Turnpike Structures for Optimal Maneuvers

by

Arthur T. Benjamin

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland
1989

Abstract

This dissertation is concerned with problems of optimally maneuvering a collection of objects ("pieces") from one location to another, subject to various restrictions on the allowable movements. We illustrate and prove that when the "distance" from the origin to the destination is large, and the movement rules and environment satisfy certain "homogeneity" properties, there exist near-optimal trajectories with very special (*turnpike*) structure.

These results are obtained by representing the problem through a configuation graph. Here, we have a node for each configuration and an arc for every "different" legal move. Each arc is endowed with a scalar weight for the move's cost, and a vector weight for the move's "progress" towards the desired final placement. Optimal solutions to the maneuvering problem are shown to be equivalent to minimum-cost walks from the origin node to the destination node with a prescribed amount of progress. The turnpike solution to the graph problem (which spends most of its time going around at most m cycles) is constructed by utilizing a basic optimal solution of an associated linear program.

We begin to explore potential applications of our configuration graph models to other types of problems (not all having to do with optimal maneuvering), and discuss further research directions.

Acknowledgments

I thank my family, friends, faculty and staff of the Mathematical Sciences Department for their support and encouragement. I am especially grateful to Roy Mathias, Ingrid Busch, and Professor Edward Scheinerman for their attention and active participation throughout the research process. Joel Auslander, Dan Wilkerson, and Professor Gregory Sullivan also provided valuable technical contributions. I thank Deena Dizengoff for her assistance with the figures. Most of all, I am indebted to Professor Alan J. Goldman for his support, leadership, and inspiration.

TABLE OF CONTENTS

LIST OF FIGURES

Thesis Outline

This dissertation is concerned with problems of optimally maneuvering a collection of objects ("pieces") from one location to another, subject to various restrictions on the allowable movements. We illustrate and prove that when the "distance" from the origin to the destination is large, and the movement rules and environment satisfy certain "homogeneity" properties, there exist near-optimal trajectories with very special (*turnpike*) structure.

In the first chapter, we examine some illustrative problems arising from Chinese-checkers-type games (e.g., jumping and sliding problems). The results and conjectures arising from the analysis of these games (obtained by the author, his thesis advisor, and others) are what led to the more general and theoretical results of the following chapters.

In chapter two, we rigorously express the ideas of the opening paragraph. For example, two placements of the pieces belong to the same equivalence class, if they are translates of each other. Using the vocabulary developed there, such *placements* have the same *configuration*, but are *placed* at different *positions*. It is proven that when our movement environment is the integer lattice $\mathbf{Z}^m$, and the origin and destination are far enough apart, then if the problem obeys certain "natural" conditions (namely: Finiteness (of the configuration set), Homogeneity (with respect to cost, time, and space), Brute Force Ability, and Cycle Positivity), we can construct a near-optimal solution which accrues almost all of its cost repeating at most m different simple patterns of movement.

These results are obtained by representing the problem through a configuation graph (abbr: $\mathcal{C}$-Graph). Here, we have a node for each configuration and an arc for every "different" legal move. Each arc is endowed with a scalar weight for the move's cost, and a vector weight for the move's "progress" towards the desired final placement. Optimal solutions to the maneuvering problem are shown to be equivalent to minimum-cost walks from the origin node to the destination node with a prescribed amount of progress. The turnpike solution to the graph problem (which spends most of its time going around at most m cycles) is constructed by utilizing a basic optimal solution of an associated linear program.

In chapter three, we discuss ways of solving the associated linear program when the $\mathcal{C}$-Graph is of manageable size. In particular, we report favorable computational results from use of a column generation scheme which utilizes a "negative cycle detector" as a major subroutine. We analyze a modified Bellman-Ford algorithm for finding negative cycles, and suggest ways of speeding up the detection. Possible algorithmic performance is bounded by showing that the decision problem associated with the $\mathcal{C}$-Graph is NP-complete.

Chapter four returns to the maneuvering problems of the second chapter, but with the added restriction that the pieces must stay within certain boundaries. Here, the space homogeneity assumption is violated, but only at the borders. It is shown how a border-ignoring turnpike trajectory can be systematically modified to accommodate this situation.

In the fifth chapter, we examine maneuvering problems over movement environments more general than $\mathbf{Z}^m$ (e.g., finitely generated abelian groups). Results analogous to those of chapter two are obtained by redefining configurations to be equivalence classes under an appropriate binary relation (analogous to the translation relation of chapter two).

In chapter six, we begin to explore potential applications of our $\mathcal{C}$-Graph models to other types of problems (not all having to do with optimal maneuvering), and discuss further research directions.

The appendices contain computational results and technical extensions of the material presented in the main chapters. A list of references follows.

Chapter 1
Motivating Problems

In the present chapter, we consider a series of attractive special cases arising from *jumping* problems and *sliding* problems. In every instance, the optimal trajectories have exhibited a special repetitive structure. The desire to "explain" and generalize this common feature provoked the investigations in the chapters that follow.

A reader who is only interested in the "serious" mathematical aspects of the dissertation could skim the results of this chapter without much loss of clarity in the subsequent chapters (i.e., all concepts and notations used in those chapters are defined there). On the other hand, it was the initial "evidence" provided by the analysis of these problems that led us to the more general theories developed later. Besides, most of the results presented here seem (at least to the author) to be interesting in their own right.

1.1 A 1-dimensional Jumping Problem

The first example we consider is a game resembling Chinese checkers. This solitaire puzzle is played with a finite set of indistinguishable pieces, using the set $\mathbf{Z}^1$ of integers as our game board. At each move, exactly one piece is displaced. Suppose that a piece is situated at the point $x \in \mathbf{Z}^1$. If $x+1$ is unoccupied, the piece can *shift* there; similarly for $x-1$. If $x+1$ is occupied, but $x+2$ is not, then the piece can *hop* over the occupant of $x+1$ to arrive at $x+2$, where it may either remain or hop over another adjacent piece, etc. (Similarly for a hop over $x-1$ to $x-2$.) A *move* consists either of a shift or a *jump* (a sequence of one or more hops by a single piece). Our objective is to transfer, in the minimum number of moves, all pieces from some configuration near the origin 0 to a specified destination configuration near d where $d > 0$ is large.

In 1983, Castells and Goldman solved the above problem when the p pieces initially occupy the points $0, 1, \ldots, p-1$, and must maneuver to occupy the points $d, d+1, \ldots, d+p-1$ in as few moves as possible. The solution consisted of maneuvering the pieces so that the back piece could jump over the remaining $p-1$ pieces. This was followed by shifting the

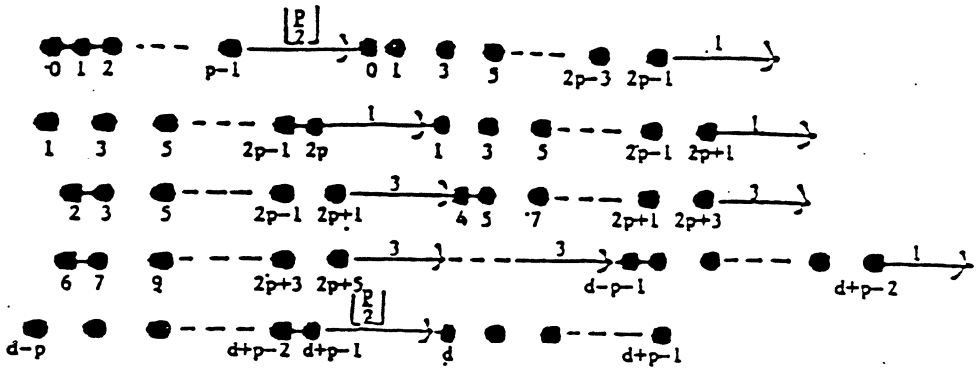


Figure 1: Solution to a 1-dimensional jumping problem when $d \geq 2p-1$.

(new) back piece forward and the (new) front piece forward, resulting in the original configuration (now translated two spaces forward). When the front piece reached position d, the pieces were "collapsed" to the desired destination placement in the obvious way. This was shown to be optimal by means of an elegant induction argument which employed a metric that diminished as the front and back pieces neared their destinations. The trajectory is illustrated in Figure 1 when $d+p$ is odd. (The trajectory when $d+p$ is even is similar.) Notice that when d is large, almost all of the time is spent repeatedly performing the same three-move sequence.

Next, we consider a 1-dimensional "forward jumping" problem with designated origin $\mathcal{O} = \{\sigma_1, \ldots, \sigma_p\}_<$ (notation: $\sigma_1 < \cdots < \sigma_p$) and destination $\mathcal{D} = \{\delta_1, \ldots, \delta_p\}_<$, with $\delta_1 \geq \sigma_p$. We shall further assume that our pieces are only allowed to move in the forward direction. For this problem, we define a configuration $\{x_1, \ldots, x_p\}_<$ to be *connected* if $x_i - x_{i-1} \leq 2$, for $i = 2, \ldots, p$. We define a *trajectory* to be a sequence $X_0, X_1, \ldots, X_n$ of configurations, where configuration X_i can be reached from configuration X_{i-1} in one move. We say that a trajectory is *connected* if all of its configurations are connected. We shall always use the symbols $\mathcal{O}$ and $\mathcal{D}$ to represent the Origin and Destination configurations respectively.

Claim 1.1 *In the above problem, if $\mathcal{O}$ is connected and $\mathcal{D}$ is connected, then there is a minimum length trajectory from $\mathcal{O}$ to $\mathcal{D}$ which is connected.*

Proof. We can obviously find a (generally,disconnected) "brute force"

trajectory with length $\sum_{i=1}^{p}(\delta_i - \sigma_i)$ by repeatedly shifting the front piece from σ_p to δ_p, then shifting the next piece on σ_{p-1} to δ_{p-1}, and so on. Since a feasible trajectory exists, a minimum length trajectory must exist. Let S be the set of all minimum length trajectories from $\mathcal{O}$ to $\mathcal{D}$. To avoid trivial cases, we shall assume $p > 1$ and the length of each minimum trajectory to be $n \geq 2$. We assert that S contains a connected trajectory.

Suppose, to the contrary, that no such connected trajectory exists. Thus every minimum length trajectory contains a disconnected configuration. For each trajectory $T \in S$, $T = (\mathcal{O} = X_0, X_1, \ldots, X_{n-1}, X_n = \mathcal{D})$ (where X_k is the k^{th} configuration, reachable in one move from X_{k-1}), let $i_T = \min_{i=0,\ldots,n}\{i : X_i \text{ is disconnected}\}$. Since $\mathcal{O}$ and $\mathcal{D}$ are connected, $1 \leq i_T \leq n-1$, and the configuration $X_{i_T-1} \equiv \{a_1, a_2, \ldots, a_p\}_<$ must have been connected, and became disconnected after moving forward the piece located at, say, position a_j. Define $j_T = j$, and give T the label (i_T, j_T). Now choose T^* to be any trajectory with label $(i_{T^*}, j_{T^*}) \equiv (i, j)$ where $i_{T^*} = \max_{T \in S}\{i_T\}$ and $j_{T^*} = \min_{T \in S}\{j_T : i_T = i\}$. In other words, T^* delays disconnecting until the last possible moment, and does so with the rearmost piece possible, without loss of optimality. Let $T^* = (\mathcal{O} = X_0^*, X_1^*, \ldots, X_{n-1}^*, X_n^* = \mathcal{D})$. Thus, $X_{i-1}^* = \{a_1, \ldots, a_p\}_<$ is connected, but after moving the piece on a_j forward, we reach $X_i^* = \{c_1, \ldots, c_p\}$ which is disconnected. Notice that since the piece on a_j either shifted forward to $1 + a_j$ or jumped over a piece on $1 + a_j$, we must have $c_k = a_k$ for $k = 1, \ldots, j-1$ and $c_j = 1 + a_j$. It is clear that $j \neq 1$ and that the only disconnecting "gap" created by this move must exist between the pieces on $c_{j-1} = a_{j-1}$ and c_j (i.e., $c_j - c_{j-1} = 3$). Thus, since $\mathcal{D}$ is connected and backwards movement is prohibited, we must eventually move forward one of the pieces located on a_k for some $k \in \{1, 2, \ldots, j-1\}$. Suppose the next time we move one of these pieces is on the t^{th} move where $t > i$. Let $X_{t-1} = \{b_1, \ldots, b_p\}_<$. (Now here's the key idea.) Since $b_1 = a_1, \ldots, b_{j-1} = a_{j-1}$ and $b_j - b_{j-1} \geq c_j - a_{j-1} = 3$, the piece situated at b_k may not move beyond $1 + b_{j-1}$ since no piece occupies $2 + b_{j-1}$. Therefore, all pieces situated beyond $1 + b_{j-1}$ are not relevant toward executing this move. Consequently, this same "move" (that is, physically moving the piece on $a_k = b_k$) could have been executed just before the move i actually made in T^*, rather than at move t. Since moves $i+1, \ldots, t-1$ do not concern the pieces on b_1 through b_{j-1}, we would still reach the same configuration X_t

after the t^{th} move. Hence, we have a new minimum trajectory that postpones the "offending" i^{th} move another turn. If this new i^{th} move preserves connectivity, then we have contradicted the definition of i_{T^*}. If this move disconnects the configuration, then we have contradicted the definition of j_{T^*} since $k < j$. Either way, we are provided with the desired contradiction. □

What does such a claim do for us? It assures us that, for this particular problem, when origin and destination are connected, we can restrict our attention to connected configurations *without loss of optimality.* We can therefore fit each configuration into a *box* of length $2p-1$. If two placements of our pieces are considered, in some sense, to be *equivalent*, should they look the same when "left-justified" in our box (i.e., they are translates of each other), then we have reduced the number of possible "different" configurations down to 2^{p-1} (if the first piece is fixed at z_1, then $z_{i+1} = 1+z_i$ or $2 + z_i$, $i = 1, \ldots, p-1$), a quantity which not only is *finite*, but does not depend on the "distance" between $\mathcal{O}$ and $\mathcal{D}$. The usefulness of such a bound will become apparent in the next chapter.

1.2 Higher-Dimensional Jumping Problems

We now direct our attention to the higher dimensional version of the jumping problem from the last section. Here, the m-dimensional integer lattice $\mathbf{Z}^m$ is our game board, and at each move, exactly one piece is displaced. The movement rules are analogous to those given previously. Specifically, letting e_i denote the i-th unit vector of $\mathbf{Z}^m$, $i = 1 \ldots m$, if a piece is situated at the point $x \in \mathbf{Z}^m$, and the point $x + e_i$ is unoccupied, the piece can shift there; similarly for $x - e_i$. If $x + e_i$ is occupied, but $x + 2e_i$ is not, then the piece can hop over the occupant of $x + e_i$ to arrive at $x + 2e_i$, where it may either remain or hop over another adjacent piece, etc. (Similarly for a hop over $x - e_i$ to $x - 2e_i$.) A move consists either of a shift or a jump (a sequence of one or more hops by a single piece). Our objective is to transfer, in the minimum number of moves, the pieces from some configuration near the origin $\mathbf{0}_m$ to a specified destination far away, parameterized by the scalar d.

For example, if we consider the two piece, two-dimensional jumping problem with origin $\{(0,0),(1,0)\}$ and destination $\{(d,d-1),(d,d)\}$, the

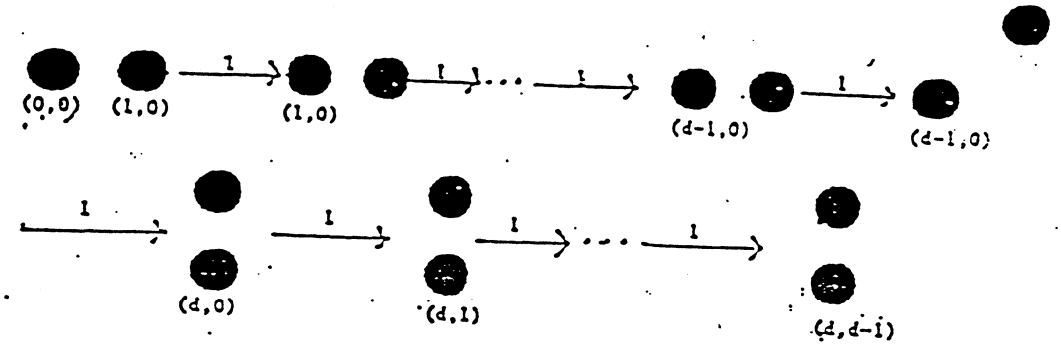


Figure 2: Solution to a two piece, 2-dimensional jumping problem.

trajectory presented in Figure 2 is indeed optimal. The notation $X \xrightarrow{\mu} Y$ in the figure denotes using μ moves to reach configuration Y from configuration X. One point in the configuration is labeled with its position in $\mathbf{Z}^2$, and the positions of the remaining pieces are thereby automatically determined. Notice that almost all of the time is spent in configurations $\bullet\bullet$ and $\begin{smallmatrix}\bullet\\\bullet\end{smallmatrix}$.

In 1985, Belur and Goldman solved the three piece, two dimensional jumping problem. Here, the prescribed origin configuration was the "lower triangle" situated at the points $(0,1)$,$(0,0)$ and $(1,0)$. Our destination configuration is the "upper triangle" situated at the points $(d-1,d)$, (d,d),and $(d,d-1)$ for some prescribed positive d.

The solution is portrayed in Figure 3. Notice that the second and fifth configurations are merely translates of each other (in the direction $(1,1)$), and the same sequence of three moves is used to reach the subsequent configurations. The sequence requires $3d-1$ moves. This was shown to be minimum by *projecting* down to a simpler 1-dimensional problem, similar to the one analyzed in the previous section.

Based on the "speed of light" results of the following subsection, an optimal solution to the analogous two dimensional *four*-piece problem (for d sufficiently large) would spend most of its time performing the two-move maneuver of Figure 4.

For the general p-piece problem ($p > 4$) in two dimensions, the following two solutions are conjectured to be optimal. The first solution is to use

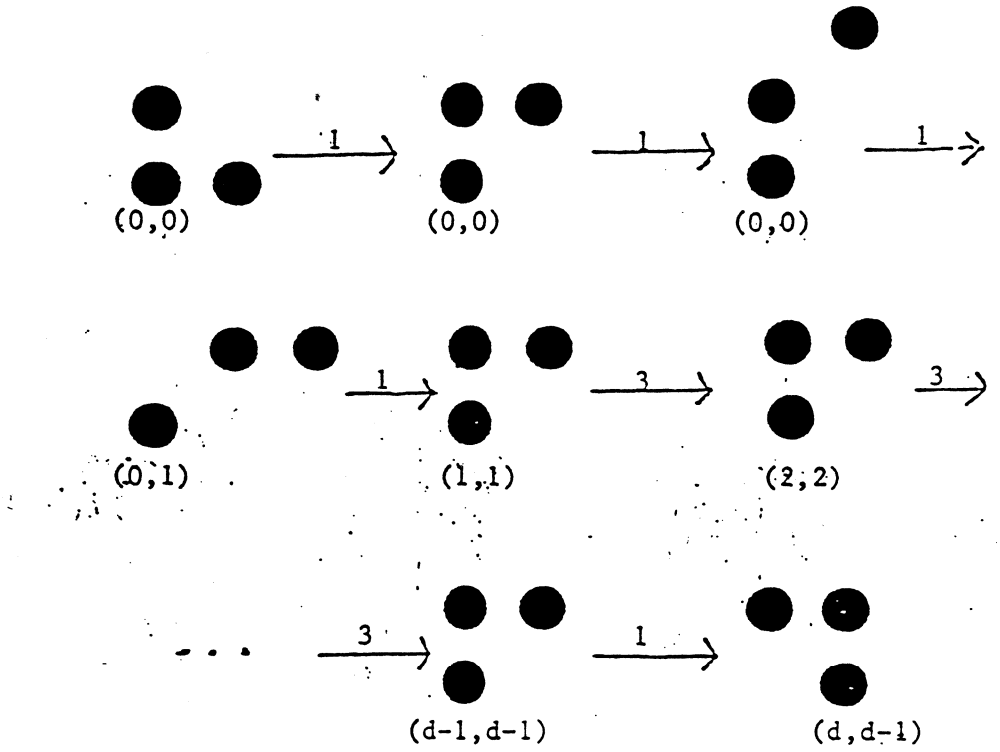


Figure 3: Solution to the 3-piece 2-dimensional jumping problem.

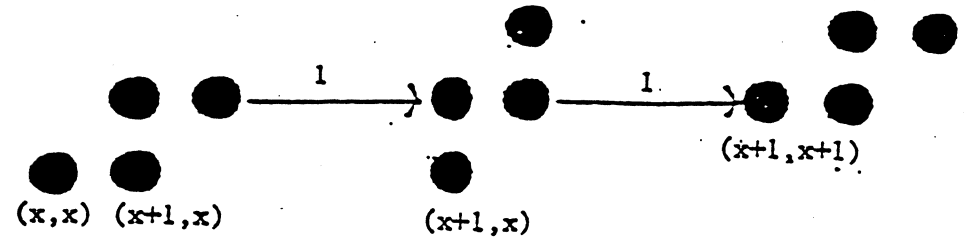


Figure 4: Solution to the four piece, 2-dimensional jumping problem (intermediate phase).

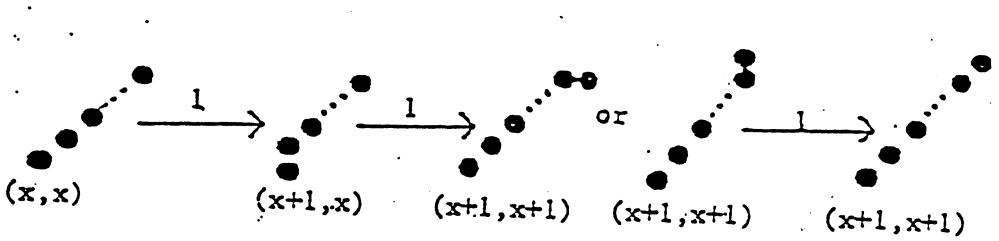


Figure 5: Conjectured solution to the p-piece 2-dimensional jumping problem (intermediate phase).

the aforementioned optimal 1-dimensional configuration to crawl along the $y = 0$ axis, then after "turning the corner", to crawl along the $x = d$ axis in a similar way. The other solution is to maneuver into a *diagonal* configuration, and repeatedly use a three-move procedure (see Figure 5) to translate it in the direction $(1,1)$ until we are near (d,d). (In Figure 5, the two alternatives for the third configuration correspond to the different possible parities of p.)

1.3 Speed of Light Configurations

As in the last section, we are interested in efficiently moving a collection of p indistinguishable pieces over the integer lattice $\mathbf{Z}^m$, using the movement rules of the jumping problem described in the last section.

We begin with some definitions. A *placement* of pieces is a size p subset of $\mathbf{Z}^m$, usually denoted as $X = \{x_1, \ldots, x_p\}$. We define the *center* of a placement X to be $c(X) = \frac{1}{p}\sum_{i=1}^{p} x_i$, which is a vector in $\frac{1}{p}\mathbf{Z}^m$. For placements X and Y, the scalar quantity $d(X,Y) = (c(Y) - c(X))^T 1_m$ is then defined as the *displacement from X to Y*, where 1_m is the vector of all ones. Loosely, displacement measures the distance between placements, where the directions e_i are viewed as *positive* directions. Note that d can be negative, or non-trivially zero. For $n \geq 1$, a *(n-move) trajectory* $X_0, X_1, \ldots, X_n$ is a sequence of placements where X_{i+1} is reachable from X_i in a single move. The *speed* of an n-move trajectory from X to Y is defined as $d(X,Y)/n$. The results of this section came about from an attempt to characterize the *speediest* trajectories.

Theorem 1.1 *The maximum trajectory speed is* $\leq 2 - \frac{2}{p}$, *where* $p \geq 2$ *is*

the number of pieces. When $p = 1$, the speed is bounded by 1.

Proof. By linearity and telescoping, we have $d(X_0, X_n)/n = \sum_{i=1}^{n} d(X_{i-1}, X_i)/n$ for any trajectory X. That is, the speed of the trajectory $X_1, \ldots, X_n$ is equal to the average of the speeds of its moves. Hence it suffices to prove that any single move (say from X to Y) has its speed (equal to $d(X, Y)$ since $n = 1$) bounded by the above constant. Note that by definition, if $X = \{x_1, \ldots, x_p\}$ and $Y = \{y_1, \ldots, y_p\}$, then $d(X, Y) = \frac{1}{p} \sum_{i=1}^{p} (y_i - x_i)^T 1_n$. Thus, if the placement Y is reached from X in a single move, say by moving the piece at x_i to y_j, then $d(X, Y) = \frac{1}{p}(y_j - x_i)^T 1_n$. Thus, when $p = 1$, $d(X, Y) = \pm e_i^T 1_m = \pm 1$. Now suppose $p \geq 2$. A shifting move has speed (equal to its displacement) of $\frac{1}{p} \leq 1 \leq 2 - \frac{2}{p}$, since $p \geq 2$. Note further that since a hop from x to $x + 2e_i$ implies that there must have been another piece at $x + e_i$, a jump from X to Y can have an excess of at most $p - 1$ positive-displacement hops over negative-displacement hops, and so has speed at most $\frac{1}{p} 2(p-1) = 2 - \frac{2}{p}$, as asserted. □

When $X = \{0_n, e_1, 3e_1, 5e_1, \ldots, (2p-3)e_1\}$ and we "long jump" the piece at 0_n to $(2p-2)e_1$, the upper bound is attainable. Note however, that this speed is not sustainable; the above move cannot be followed immediately by another long jump. In fact, the next theorem shows that a "repeatable" trajectory has speed at most 1.

We say that placement Y is a *translate* of X if there exists $a \in \mathbf{Z}^m$ such that $Y = X + a$ (i.e., $\{y_1, \ldots, y_p\} = \{x_1 + a, \ldots, x_p + a\}$). Such placements X and Y are said to be represented by the same *configuration.* Next we define the following convenient (though somewhat abusive) notation. For $x \in \mathbf{Z}^m$ define $\|x\|$ to be $x^T 1_m$, and for all integers M, let the *border* M be $\{x \in \mathbf{Z}^m : \|x\| = M\}$. When $M = B(X) \equiv \min_{i=1}^{p} \|x_i\|$, for placement X, then such a border is called the *back border of* X and is said to have *value* $B(X)$. Replacing min with max provides us with analogous *front border* definitions, and notation $F(X)$ analogous to $B(X)$. When we say that a piece is situated *on* border M, we mean that it occupies a point *in* border M.

Theorem 1.2 *Let Y be a translate of X. Then any trajectory from X to Y has speed at most* 1.

Proof. Suppose $Y = X + a$ for some $a \in \mathbf{Z}^m$. For ease of notation, assume that $y_i = x_i + a$, $i = 1 \ldots p$, whence $d(X,Y) = \frac{1}{p}\sum_{i=1}^{p}(y_i - x_i)^T 1_n$ $= \frac{1}{p}\sum_{i=1}^{p} a^T 1_n = \|a\|$. Next observe that at any move, the value of the back border cannot increase by more than 1. For example, let x be the location of a piece on the back border of some placement. If the piece at x does not move, then clearly, the back border's value cannot increase. If the piece at x does move (in such a way that the border value increases), then the new placement must have a piece at $x + e_i$ for some i. (If the move was a shift, this is clear; if the move was a jump, the piece at x had to make an initial hop over a piece situated at $x + e_i$ for some i.) Hence, after this move, the new back border's value could not have increased by more than 1. Since $B(Y) = \min_{i=1}^{p} \|(x_i + a)\| = \min_{i=1}^{p} \|x_i\| + \|a\| = B(X) + \|a\|$, it follows from the previous observation that the number of moves n needed for a trajectory from X to Y is at least $\|a\| = d(X,Y)$. If $d(X,Y) \leq 0$, then since $n \geq 1$, the trajectory has non-positive speed. Otherwise, since $n \geq d(X,Y)$, its speed is $d(X,Y)/n \leq 1$. □

A placement X is called a *speed of light placement* if there exists a non-zero vector $a \in \mathbf{Z}^m$ and a speed one trajectory (called a *speed of light trajectory*) from X to $X + a$. Since the rules of our game depend only on the relative positions of our pieces, it is clear that if X is a speed of light placement, and Y is a translate of X, then Y is also a speed of light placement. Thus, a *speed of light configuration* is well defined, and has the obvious definition. In figure 6, we illustrate speed of light configurations for the two dimensional case (where $p = 1, 2,$ and 4, respectively). In fact, the next theorem demonstrates that these are the *only* such configurations in two dimensions, and essentially the only ones for higher dimensions, too.

Theorem 1.3 *The following are speed of light configurations:*

the atom $\{x\}$ *(when* $p = 1$*),*

the frog $\{x, x + e_i\}$ $1 \leq i \leq m$ *(when* $p = 2$*), and*

the slug $\{x, x + e_i, x + e_i + e_j, x + 2e_i + e_j\}$ $1 \leq i \neq j \leq m$ *(when* $p = 4$*).*

No other speed of light configurations exist.

Proof. The first part of the theorem is straightforward. The atom can translate itself (in the direction e_i) by shifting itself from $\{x\}$ to $\{x + e_i\}$,

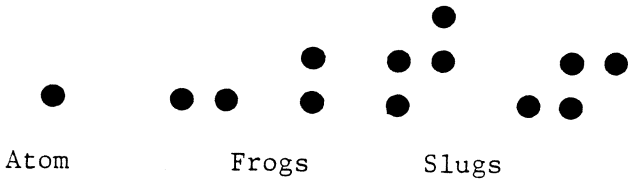


Figure 6: Speed of Light Configurations

a speed one maneuver when $p = 1$. The frog $\{x, x + e_i\}$ translates itself (in the direction e_i) in a single hop to $\{x + e_i, x + 2e_i\}$, a speed one maneuver when $p = 2$. When $p = 4$, the slug performs two consecutive double jumps to go from $\{x, x + e_i, x + e_i + e_j, x + 2e_i + e_j\}$ to $\{x + e_i + e_j, x + 2e_i + e_j, x + 2e_i + 2e_j, x + 3e_i + 2e_j\}$, translating itself in the direction $e_i + e_j$ in two moves.

We establish the second part of the theorem by proving a series of necessary conditions that must be satisfied by speed of light objects.

Lemma 1.1 *Every move in a speed of light trajectory must simultaneously increase the values of the back border and the front border.*

Proof. As argued in proving theorem 1.2, a m-move trajectory from X to $X + a$ has speed $\|a\|/n$, where $n \geq \|a\|$. Note that $B(X + a) = B(X) + \|a\|$, $F(X + a) = F(X) + \|a\|$. We observe (as in the proof of theorem 1.2) that the functions B and F cannot increase by more than 1 each move. Hence, in order for $n = \|a\|$, we must simultaneously increase the values of both borders each move. □

Lemma 1.2 *Let X be a speed of light placement. Then there is a unique piece on border $B(X)$, and a speed of light move must "jump" that piece to a point on border $F(X) + 1$.*

Proof. This follows immediately from Lemma 1.1 and the fact that only one piece moves each turn. □

In fact, we have

Lemma 1.3 *Given a speed of light placement X and $B(X) \leq M \leq F(X)$, there is at most one occupied point $x \in X$ with $\|x\| = M$.*

Proof. Suppose, to the contrary, that more than one piece is situated on border M. By Lemma 1.2, the first $M - B(X)$ moves of the trajectory involve moving pieces from borders with values less than M to borders with values greater than M, after which our new back border has value M. But then this border has more than 1 piece, contradicting lemma 1.2. (Note that we are using the fact that all intermediate placements in a speed of light trajectory are themselves speed of light placements, and that the "movement pattern" of such a trajectory can be repeated to produce an arbitrarily long speed of light trajectory, if desired.) □

Notice that when $p = 1$, the atom is the only configuration. Henceforth, we shall assume that $p \geq 2$.

Lemma 1.4 *When $p \geq 2$, every move in a speed of light trajectory is a jump.*

Proof. Since $p \geq 2$, we have $F(X) > B(X)$ by lemma 1.2. Since a shift can not take a back border piece beyond border $B(X) + 1 \leq F(X)$, it cannot expand the front border, as required. □

Notice that in a speed of light trajectory, for a piece on border M to make "forward progress", it must hop over a piece on border $M + 1$ and land on border $M + 2$. (Here we are using the fact that all moves are jumps and that each hop changes the piece's border value by an even amount, ruling out the possibility of hopping *over* border M during the jump.)

Lemma 1.5 *Every speed of light placement X has at least one piece on every border between $B(X)$ and $F(X)$.*

Proof. By definition, the back border and front border are occupied. Suppose, to the contrary, that there is no piece on some smallest border M where $B(X) < M < F(X)$. The first $M - B(X) - 1$ moves of a speed of light trajectory starting at X must jump pieces from the back border (initially $B(X)$, then $B(X)+1$, then... $M-2$) to a new front border beyond $F(X) > M$. At move number $M - B(X)$, the back border has value $M - 1$ and a unique piece there, but there is no piece on border M over which

that "unique piece" can hop. Hence, the speed of light trajectory could not repeat itself, a contradiction. □

Thus, by lemmas 1.3 and 1.5, we have

Lemma 1.6 *Every speed of light placement X must have exactly one piece on each border between border $B(X)$ and border $F(X)$. Consequently, $F(X) = B(X) + p - 1$.*

Lemma 1.7 *If X is a speed of light placement with $p \geq 2$ pieces, then p must be even.*

Proof. By Lemmas 1.2 and 1.6, the first move must jump a piece from border $B(X)$ to border $B(X) + p$. Since that piece can only jump to a border whose value has the same parity as $B(X)$, p must be even. □

We note here that by lemma 1.4, when $p = 2$, all speed of light configurations must be of the form $\{x, x + e_i\}$ for some $1 \leq i \leq m$, i.e., the frogs. By lemma 1.7, $p \neq 3$. Henceforth we shall restrict our attention to the case where $p \geq 4$.

It remains to prove that the only possible remaining speed of light configurations are of the slug variety. The argument we give is a little tricky. Let x be an arbitrary point in $\mathbf{Z}^m$. Let $\|x\| = M$. Now imagine that we have a speed of light trajectory which makes one move every second. Suppose, without loss of generality, that the sole front border piece of our speed of light placement X is presently ($t = 0$) situated at x. We shall focus our attention on only those points in $\mathbf{Z}^m$ that occupy borders of value M or higher. Thus at $t = 0$, all that we see is a single piece, situated at x (see Figure 7).

When $t = 1$, the front border's value has increased to $M + 1$. Since x is the only occupied point in border M, the new front border piece must have made its final hop over x to land on the point $x + e_i$ for some $1 \leq i \leq m$. At $t = 1$, we see, as in Figure 8, a piece at x and one at $x + e_i$.

When $t = 2$, the front border's value has increased to $M + 2$. The piece which landed there hasd to make its final hop over the (only) piece on border $M + 1$, located at $x + e_i$. Hence the piece on border $M + 2$ must be situated at $x + e_i + e_j$ for some $1 \leq j \leq m$. We now show that $j \neq i$ by the following

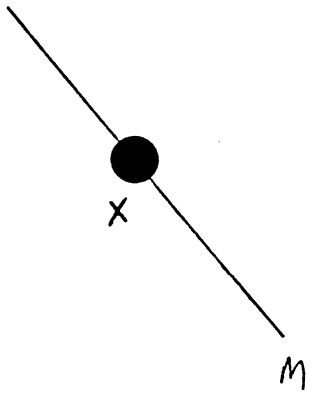


Figure 7: What we see when $t = 0$.

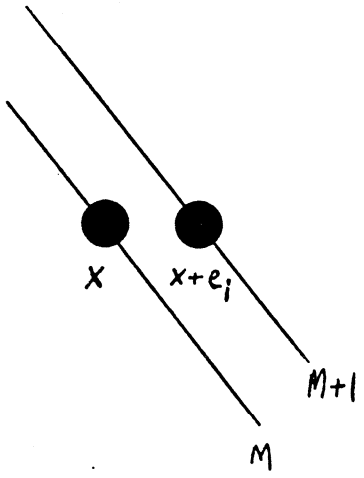


Figure 8: What we see when $t = 1$.

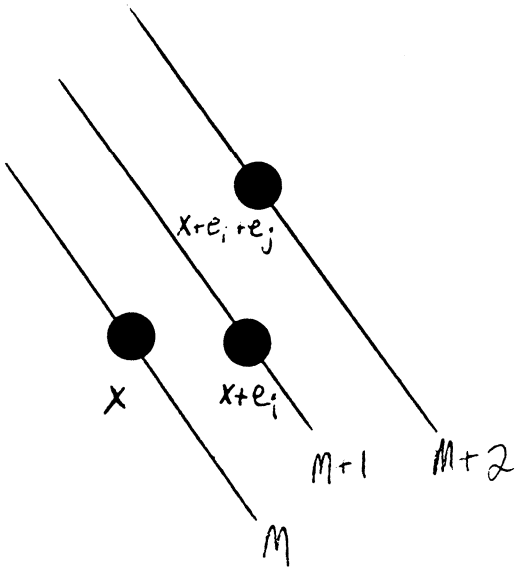


Figure 9: What we see when $t = 2$.

Lemma 1.8 *If the number of pieces is at least* 4, *the direction of the final hop of a speed of light move must be different from the direction of the final hop in the previous speed of light move.*

Proof. Suppose that in the previous move, the jumping piece's final hop was over a piece situated at y to land on $y + e_i$, becoming the unique front-border piece. Then the jumping piece of the current move must have hopped over $y + e_i$ to reach border $\|y\| + 2$; if its final hop had the same direction as that of the previous move, it must be on $y + 2e_i$. But if that were the case, then the piece on $y + 2e_i$ must have originated from y. But the piece at y could not have moved because (since $p > 2$) it is not a back border piece. □

Thus, at $t = 2$, we see three pieces, situated at x, $x + e_i$, $x + e_i + e_j$, where $1 \leq i \neq j \leq m$. (See Figure 9.)

When $t = 3$, a piece is jumped to the new front border $M + 3$ and, since it had to hop over the piece at $x + e_i + e_j$, it must end up at $x + e_i + e_j + e_k$ for some $1 \leq k \leq m$. By Lemma 1.8, we know that $k \neq j$. We now show that, in fact, $k = i$. The new front border piece, before it made its final hop over $x + e_i + e_j$ to $x + e_i + e_j + e_k$, must have been at the point $x + e_i + e_j - e_k$ on border $M + 1$. But how did it get there? It had to hop over the sole piece on border M, situated at x. But this requires $e_i + e_j - e_k$ to be a unit vector, which is only possible when $k = i$ or $k = j$. And since $k \neq j$, we have $k = i$. Hence at $t = 3$, we see four pieces, situated at x, $x + e_i$, $x + e_i + e_j$, and $x + 2e_i + e_j$, as in Figure 10.

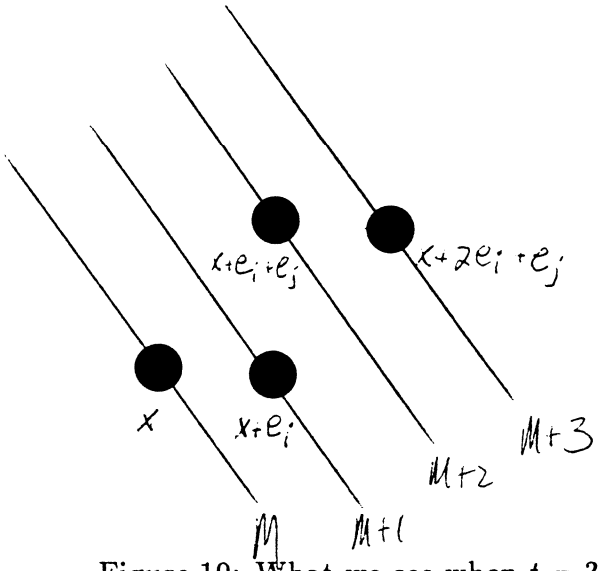


Figure 10: What we see when $t = 3$.

When $t = 4$, the back border piece, wherever it is, jumps to border $M+4$, landing on $x+2e_i+e_j+e_k$ for some $1 \leq k \leq m$. Hence, its final hop originated from $x+2e_i+e_j-e_k$ on border $M+2$, and the hop before that (over the piece on $x+e_i$) must have come from $x+e_j-e_k$. But in order for the hop over $x+e_i$ to be legal, $(x+2e_i+e_j-e_k)-(x+e_i) = e_i+e_j-e_k$ must be a unit vector. Since $k \neq i$ by Lemma 1.8, we have $k = j$. Thus, the hop over $x+e_i$ was to $x+2e_i$ and therefore originated from x, which is occupied. Thus, the piece at x must be the back border piece when $t = 4$. Hence, $p = 4$, and the pieces seen in Figure 10 are all the pieces (and are in the slug configuration). By reversing the last four moves, we see that our original speed of light placement X was also a slug. Specifically, at $t = 0$, the pieces occupied the points, y, $y+e_j$, $y+e_j+e_i$, and $y+2e_j+e_i$, where $y = x-2e_j-e_i$. The theorem is proven. □

As a consequence of this theorem, we see that no speed of light configurations exist when the number of pieces is three or greater than four. In these cases, it is easy to see that there exist configurations which are translatable with speed 2/3. For example, see Figures 1, 3, and 5. The question of whether the speed 2/3 is *optimal* for the two (and higher) dimensional problem, when $p = 3$ or $p > 4$, remains open. If this is the case, then (for d sufficiently large) the first conjectured solution to the 2-dimensional problem (see the last paragraph of the previous section) would be "almost optimal", when $p = 3$ or $p > 4$.

1.4 A Sliding Problem

Here we consider a 1-dimensional *sliding problem* on $\mathbf{Z}^1$ with p indistinguishable pieces. A piece situated at point x can slide to point y in one move, provided that y is unoccupied, and all points between x and y are occupied. (Note that if $y = x + 1$ or $y = x - 1$, then the slide from x to y is simply a shift.) The pieces originally occupy the points in $\mathcal{O}$, where $\mathcal{O} = \{a_1, a_2, \ldots, a_p\}_<$ (i.e., $a_1 < a_2 < \cdots < a_p$), and we wish to maneuver them to occupy the points of $\mathcal{D} = \{b_1, b_2, \ldots, b_p\}_<$ in as few moves as possible. Let us define for any placement $X = \{x_1, x_2, \ldots, x_p\}_<$, the *back border* of X to be x_1 and its *front border* to be x_p.

Claim 1.2 *When $b_1 \geq a_p$, the unique solution to this problem is: each move, slide forward the piece that is farthest back and not on a destination point. The optimal sequence of moves requires $\Delta \equiv b_p - a_1 - (p-1)$ moves.*

We denote the optimal trajectory described above by T. T begins with an opening leading to a "long slide configuration" (i.e., all p pieces are scrunched together–no gaps) with front border at a_p. This is followed by a sequence of "long slides" (i.e., sliding the back piece to the front) until the back border reaches b_1. At this point we ignore the piece on b_1 and proceed as before, depositing subsequently-fixed pieces on $b_2, b_3, \ldots, b_p$ in turn.

Proof. First, we make some useful observations:

- The front and back borders cannot change by more than 1 in any move (i.e., they are "discrete continuous" functions).
- A move can simultaneously increase (or decrease) *both* borders, if and only if the pieces are in the long slide configuration (i.e., the difference between the front border and back border is $p - 1$).
- At any given move, a particular piece can move to exactly 2 locations—the next unoccupied spot in front of or in back of the piece.
- Since all moves are reversible, there is an obvious one-to-one correspondence between sequences of moves (*trajectories*) that take $\mathcal{O}$ to $\mathcal{D}$ and trajectories that take $\mathcal{D}$ to $\mathcal{O}$. This correspondence preserves the *length* (i.e., the number of moves) of the trajectory.

We begin by proving the claim when the origin is already in the most efficient configuration.

Lemma 1.9 *The above theorem is true when $a_p - a_1 = p - 1$ (i.e., when we start out in the long slide configuration), and the destination obeys the (weaker) data assumption that $b_1 \geq a_1$. Furthermore, T expands the front border each turn.*

Proof. First observe that since $a_p = a_1 + (p-1)$, we have $\Delta = b_p - a_p$, i.e., Δ is the difference between the front borders of the origin and destination. Thus, by discrete continuity of the front border, any trajectory from $\mathcal{O}$ to $\mathcal{D}$ requires at least Δ moves.

We now prove inductively, that there exists a trajectory from $\mathcal{O}$ to $\mathcal{D}$ of length Δ. Notice that

$$\Delta = b_p - a_1 - (p-1) \geq (b_1 + (p-1)) - a_1 - (p-1) = b_1 - a_1 \geq 0,$$

and that equality holds if and only if $b_1 = a_1$ and $b_p = b_1 + (p-1)$, (i.e., $b_p = a_p$); that is, $\Delta = 0$ if and only if $\mathcal{O} = \mathcal{D}$. Thus, the base case $\Delta = 0$ is vacuously satisfied by the "null" trajectory. Now suppose, inductively, that there exists a trajectory of length Δ for all origin-destination pairs satisfying the conditions of the lemma with $\Delta \leq k$ ($k \geq 0$). Next, consider an origin-destination pair satisfying the lemma's conditions with $\Delta = k+1$. Let $\mathcal{D}'$ be the position obtained by sliding the front piece of $\mathcal{D}$ sdrawkcab. This produces a position with the front border diminished by 1. (The data assumption remains true since $\Delta \geq 1$.) Thus, for the pair $(\mathcal{O}, \mathcal{D}')$, we have $\Delta = k$. By the induction hypothesis, there exists a trajectory which takes $\mathcal{O}$ to $\mathcal{D}'$ in exactly k moves. By sliding the former front piece forward again, we reach the destination $\mathcal{D}$. Thus, we have found a trajectory which takes the pieces from $\mathcal{O}$ to $\mathcal{D}$ in $k+1$ moves; the induction is complete.

Next, we show that the trajectory T (described in the Claim) takes $\Delta = b_p - a_p$ moves when the conditions of the lemma are satisfied. We proceed by induction on p. When $p = 1$, our trajectory merely moves the sole piece forward $\Delta = b_1 - a_1$ times, as asserted. Inductively, our trajectory with p pieces spends its first $b_1 - a_1$ moves moving the current back piece to the front, at which point the new back piece is at destination point b_1. The rest of the trajectory proceeds as if we started with $p-1$ pieces in compact

formation, with the back piece at position $b_1 + 1$ and hence with front piece at position $b_1 + 1 + (p-2)$. By the induction hypothesis, this subproblem takes $b_p - (b_1 + 1 + (p-2)$ moves. Thus, the total number of moves used is $(b_1 - a_1) + (b_p - (b_1 + 1 + (p-2)) = b_p - (a_1 + (p-1)) = b_p - a_p = \Delta$, as desired.

To prove uniqueness, notice that since $\Delta = b_p - a_p$, any optimal trajectory *must* increase the front border at every move. Consider any intermediate configuration in an optimal trajectory. Let x be the location of that piece which is farthest back and not on a destination point. We now show that we must move the piece at x forward, by showing that any other move would ultimately lead to a move that did not expand the front border. Clearly, in order to expand the front border each time, we must always be moving our pieces forward. If we moved forward a piece that was behind x, then that piece left a destination point b_i. Thus, we have at most $i-1$ pieces occupying the at most $i-1$ destination points behind b_i, with b_i unoccupied. Thus, at some point we will have move a piece back to "reclaim" one of these i destination points. Such a move will not expand the front border; thus it is not optimal to move forward a piece behind x.

Next, suppose we move forward a piece, located at z, that is in front of x. Such a move necessarily creates a "gap" at z, separating the piece at x from the front border piece. When it comes time to move the piece on x forward (which must eventually happen since x is not a destination point), a gap will still exist between x and the front border piece because the only moves which could close the gap are backward moves in front of x (which can not be optimal) or forward moves of pieces behind x (which as previously argued also cannot be optimal); thus we will be unable to expand the front border on this move. Hence, the only possible optimal move is to slide x forward. Thus T is the unique optimal trajectory, as asserted, and the lemma is proved. □

To prove the theorem, let us consider an arbitrary $(\mathcal{O}, \mathcal{D})$ pair satisfying the conditions of the theorem (here, $b_1 \geq a_p$). We aim to prove that T is the unique optimal trajectory taking $\mathcal{O}$ to $\mathcal{D}$. We define a scoring function, D (like the one used by Castells and Goldman (1983)) for any intermediate configuration $\{c_1, \ldots, c_p\}_<$ as $D(c_1, \ldots, c_p) = |b_p - c_p| + |b_1 - c_1|$. Thus, $D_0 \equiv d(\mathcal{O}) = (b_p - a_p) + (b_1 - a_1)$, and $D(\mathcal{D}) = 0$. Recall that $\Delta = b_p - a_1 - (p-1)$.

Thus it suffices to prove that T takes the pieces from $\mathcal{O}$ to $\mathcal{D}$ in Δ moves, that Δ is the minimum number of moves needed to take D from D_0 down to zero, and that this can be done in precisely one way.

To see that T takes Δ moves, note that the first $[a_p - a_1 - (p-1)]$ moves of T expand the back border (i.e., we move the back piece forward; this follows from $b_1 - a_1 \geq a_p - a_1$) but do not expand the front border because the front piece and back piece are more than $p-1$ apart. After these moves, the front piece and back piece are exactly $p-1$ apart, and hence all spaces in between are occupied. Let us call this configuration (with back piece presently at the point $a_p - (p-1)$) configuration LS (for Long Slide). From this placement of LS, our lemma asserts that T takes $b_p - a_p$ moves to attain destination $\mathcal{D}$. Thus, T takes our pieces from $\mathcal{O}$ to $\mathcal{D}$ in $a_p - a_1 - (p-1) + b_p - a_p = b_p - a_1 - (p-1) = \Delta$ moves.

Next, we notice that from any configuration, D can decrease by at most 2 in a single move. The only way D can decrease by exactly 2 is when both the front and back borders move closer to their destinations. This can only occur when the front border is less than b_p and we make a "long slide" forward, or the back border is greater than b_1, and we make a long slide backwards.

Note that when $p > 1$, an optimal trajectory must eventually make long slide moves, for otherwise D would diminish by at most one each move and therefore at least $d_0 = b_p - a_1 + b_1 - a_p \geq b_p - a_1 > \Delta$ moves would be required when $p > 1$. Next, we show that long slide positions with front border strictly less than a_p are undesirable. Roughly, if we use L (leftward) moves retracting the front border to $a_p - L$, we earn the right to perform L more long jumps, but at the cost of having L more spaces to travel than before–a net waste of L moves.

More rigorously, let $a_p - L$ be the front border of the leftmost long slide configuration in our trajectory. Denote the back border, $a_p - L - (p-1)$, by a'. At least L moves are spent bringing the front border down from a_p to $a_p - L$. If $a' > a_1$, then at least $a' - a_1$ other moves are spent bringing the back border up from a_1 to a', and by our lemma, at least $b_p - a' - (p-1)$ other moves are spent maneuvering from LS (with back border a') to $\mathcal{D}$, resulting in a total of at least $L + (a' - a_1) + (b_p - a' - (p-1) = L + \Delta > \Delta$ moves. If $a' \leq a_1$, then the total number of moves needed is at least $L + 0 + (b_p - a' - (p-1) \geq L + b_p - a_1 - (p-1) = L + \Delta > \Delta$. Likewise,

it does not pay to have the *back* border of LS strictly greater than b_1 (say at $b_1 + R$), since we would need at least $b_1 + R - a_1$ moves to reach LS, then at least R backward moves to bring the back border back, and (by our lemma) at least $b_p - (b_1 + R) - (p - 1)$ forward moves to go from LS to $\mathcal{D}$, thus requiring a total of at least $(b_1 + R - a_1) + R + [b_p - (b_1 + R) - (p - 1)] = R + b_p - a_1 - (p - 1) = R + \Delta > \Delta$ moves, which would be suboptimal. That is, to perform a LS in an optimal trajectory we must have its *front* border less than $b_1 + (p - 1)$ and (from earlier this paragraph) at least at a_p. Therefore, since it is obviously sub-optimal to repeat the same placement in a trajectory, any optimal trajectory can perform at most $b_1 + (p - 1) - a_p$ long slides, which is precisely the number that T uses. Hence T must be optimal, since it diminishes D by 2 the maximum number of times, and diminishes it by 1 at all other moves (because $b_1 \geq a_p$, all the pre-LS moves bring the back border closer to b_1 (without affecting the front border) and all the post-LS moves bring the front border closer to b_p without affecting the back border). Furthermore, any optimal trajectory must perform all of these long slides, and therefore reach configuration LS (with front border a_p). We know from our lemma that T is the unique optimal trajectory from LS to $\mathcal{D}$. (Note $b_1 \geq a_p \geq a_p - (p - 1)$.) Similarly, by our lemma, "the reversal of T", call it T', is the unique optimal trajectory from LS to $\mathcal{O}$. (Here we need the data assumption $b_1 \geq a_p$ since the first "destination point" is a_p.) Our lemma asserts that T' will always increase its front border (i.e., bring it closer to $\mathcal{O}$). Thus the reversal of T' (from $\mathcal{O}$ to LS) always increases its back border, which is never at a destination point since $b_1 \geq a_p$. Thus, the reversal of T' from LS to $\mathcal{O}$ is just T from $\mathcal{O}$ to LS. Therefore, T is the unique optimal trajectory from $\mathcal{O}$ to $\mathcal{D}$, and requires exactly Δ moves. The claim is proved. □

The solution is illustrated in Figure 11; most of its time is spent (when $b_1 \gg a_p$) performing the "long slide", a one-move sequence.

1.5 Summary

All of the preceding solutions share a common feature. When the distance between the origin and destination is sufficiently large, most of the cost (i.e., the number of moves) is spent repeatedly translating one or two configurations (such as •• and $\overset{\bullet}{\bullet}$ in the two piece jumping problem solution,

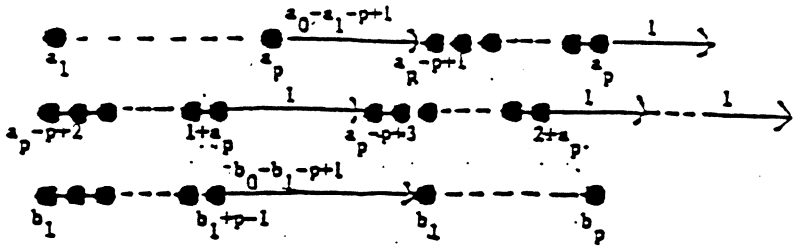


Figure 11: Solution to a 1-dimensional sliding problem.

or LS in the sliding-problem solution). This resembles the phenomenon that if one had to efficiently drive a great distance (say from Baltimore to Los Angeles), one would spend most of the time on (perhaps only one or two) high speed interstate highways or *turnpikes*.[1] Instances of this turnpike theme have been identified in the operations research literature, making both theoretical and algorithmic contributions toward solving knapsack problems (see Gilmore and Gomory 1966 and Shapiro and Wagner 1967) and Markov decision problems (see Shapiro 1968). The theme has been somewhat more prominent in mathematical economics (see Cass 1966 and McKenzie 1986). In a similar spirit, we are led by the procedding examples to attempt a unification, by identifying and proving turnpike theorems for general maneuvering problems.

[1]Strictly speaking, a turnpike is a high speed highway where some toll is charged, as opposed to a *free*way. We shall not make use of this distinction.

Chapter 2
Graphs, Maneuvers and Turnpikes

2.1 One Dimensional Turnpike Theory

The general problem of finding a minimum cost sequence of moves from one subset of $\mathbf{Z}^1$ to another can be viewed as a minimum cost path problem on an infinite directed graph, where each node represents a possible placement of the pieces on $\mathbf{Z}^1$, and there exists an arc with weight c directed from node X to node Y if it is possible to reach Y from X in a single move with cost c. (In our earlier examples, each arc had unit cost.) Of course, unless we make some additional assumptions about the rules for movement (and hence, the associated graph), we cannot hope to make any useful statements about these problems in general.

Before presenting such assumptions, we clarify the concepts of configuration and placement, and develop a useful notation. At each moment in time (i.e., before each move) our pieces are arranged in some *configuration* X, whose back piece is situated at the position $a \in \mathbf{Z}^1$. We will refer to (X, a) as a *placement* of configuration X at the point a. For example, Figure 12 illustrates the situation where $p = 3$, $X = \bullet\bullet \quad \bullet$ and $a = 4$. Thus, if we consider placements to be "related" if they are translates of each other, then configurations are precisely the equivalence classes under the translation relation. In this more discriminating terminology, "configuration" matches the intuitive notion of "formation", while a placement is a "*placed* configuration". The notation $(X, a) \xrightarrow{c} (Y, b)$ denotes moving from (X, a) to (Y, b) with cost c (e.g., in c moves). If no c is identified, it is assumed that $c = 1$. [The choice of representing the position by its back piece is a fairly arbitrary one. The front piece, second piece, or location of its center of gravity (as in section 1.3) would also be acceptable, and in certain proofs, may be easier to work with. For instance, if one piece has special properties, its location may be a natural position parameter.] We are now ready to state our assumptions on 1-dimensional movement, which we abstract from the properties of the particular cases discussed in Chapter 1.

Figure 12: Configuration X and placement (X, a).

2.1.1 Rules-for-Movement Assumptions over $\mathbf{Z}^1$

We are interested in moving a collection of objects (called *pieces*) from one subset of $\mathbf{Z}^1$ to another at minimum cost, subject to restrictions on the allowable moves. We assume our rules for movement obey the following assumptions, to be discussed after their statements:

Finiteness. Without loss of optimality, we can prescribe a finite set $\mathcal{C}$ of allowable configurations for our pieces. From each configuration, there are a finite number of legal moves available.

Time Homogeneity. For all $(X, a) \in \mathcal{C} \times \mathbf{Z}^1$, the legal moves available from (X, a) do not depend on the particular moment in time.

Cost Homogeneity. For all $(X, a) \in \mathcal{C} \times \mathbf{Z}^1$, the legal moves available from (X, a) do not depend on the total cost accrued previously in reaching (X, a).

Space Homogeneity. For all $(X, a) \in \mathcal{C} \times \mathbf{Z}^1$, the legal moves available from (X, a), as well as their costs, do not depend on a. In terms of our notation, this says that for any $a, b, c, \delta \in \mathbf{Z}^1$, and $X, Y \in \mathcal{C}$, $(X, a) \xrightarrow{c} (Y, b)$ is legal if and only if $(X, a+\delta) \xrightarrow{c} (Y, b+\delta)$ is legal.

Brute Force Ability. There exist $r \geq 0$ and non-negative integral "brute force" constants $\{c_\delta : \delta \geq r\}$ such that for all $X, Y \in \mathcal{C}$, $a \in \mathbf{Z}^1$ and $\delta \geq r$, $(X, a) \xrightarrow{c_\delta} (Y, a + \delta)$ is legal.

Cycle Positivity. If $(X, a) \xrightarrow{c} (X, a + \delta)$ is legal, then $c \geq 0$. If $\delta \neq 0$, then $c > 0$.

Remarks:

- By stating that the pieces must occupy some *subset* of $\mathbf{Z}^m$, we have implicitly assumed that all pieces are indistinguishable and that points cannot be occupied by more than one piece at the same time. We can easily accommodate more general situations by changing notation. (This is discussed further in Chapter 5.)

- For some rules for movement, the finiteness property is *explicit*— for instance, if the rules themselves actually list a finite number of legal configurations or require some sort of "connectivity" or "compactness of formation". It would be desirable, as in Claim 1.1, to derive useful conditions sufficient for our rules *implicitly* to yield the finiteness condition. Also, we may wish to weaken the without-loss-of-optimality assumption to *without loss of asymptotic optimality*, meaning that the difference between the minimum trajectory cost when restricted to our finite configuration set and the minimum (unrestricted) cost is bounded above, by a constant which does not depend on the "distance" between the origin and destination.

- Even when time homogeneity is not strictly present, we can sometimes modify $\mathcal{C}$ so that time homogeneity is obeyed. For instance, suppose that our rules involve periodic "refueling" or "maintenance" restrictions like "you cannot go more than $t_i \geq 1$ time units without maneuvering into some configuration in the set $S_i \subseteq \mathcal{C}, i = 1 \ldots n$." Then one "simply" multiplies the number of configurations by $\prod_{i=1}^{n}(t_i)$ by associating, with each $X \in \mathcal{C}$, the new configurations $X^{(s_1,s_2,\ldots,s_n)}, 0 \leq s_i < t_i, i = 1, \ldots, n$. The legal moves are precisely those of the following form: supposing that $X \xrightarrow{c} Y$ when time is not a consideration (e.g., at time 0), that $J = \{j : Y \in S_j\}$, and that $s_j < t_j - 1$ for all $j \notin J$, then in our restricted problem

$$X^{(s_1,\ldots,s_n)} \xrightarrow{c} Y^{(s'_1,\ldots,s'_n)}$$

where

$$s'_j = \begin{cases} 0 & \text{if } j \in J \\ s_j + 1 & \text{if } j \notin J. \end{cases}$$

With $\mathcal{C}$ redefined to be $\{X^{(s_1,\ldots,s_n)} : X \in \mathcal{C}^{\text{old}}, 0 \leq s_i < t_i\}$, our rules now obey the time homogeneity assumption. In a similar way, one

could accommodate restrictions of the form: you cannot make more than t_i consecutive moves of "type" $i, i = 1, \ldots, n$.

- Similarly, certain cost non-homogeneities can be accommodated in the same way as time non-homogeneities.

- It would be desirable to weaken the space-homogeneity assumption to allow for *boundaries* on (or *obstacles* in) an otherwise homogeneous environment. This line of generalization will be explored in Chapter 4.

- By space homogeneity, to verify Brute Force Ability it suffices to show $(X, 0) \xrightarrow{c_\delta} (Y, \delta)$ is legal.

- We could allow the arc costs to be non-integral, and all subsequent theorems would follow, provided that we re-interpret the notation $(X, a) \xrightarrow{c_\delta} (Y, a + \delta)$ in the brute force assumption to mean that we could maneuver from placement (X, a) to placement $(Y, a + \delta)$ with cost *not exceeding* c_δ.

- The non-negativity of r (as in "$r$adius of maneuver") in the brute force assumption means that the existence of any legal "backwards" moves is not guaranteed, so that our destination had better be in the forwards direction. A stronger assumption implying "there exists c such that $(X, a) \xrightarrow{c} (Y, a)$ is legal", would exclude situations like the "forward moving jumping problem" (analyzed in Claim 1.1) where backwards movement was not allowed. (There would be no way to reach $(Y, 1)$ from $(X, 1)$ with $Y = \bullet\bullet\bullet$ and $X = \bullet\bullet\ \ \bullet$, while one could reach $(Y, 2)$ from $(X, 1)$.) We can often assume, without loss of optimality, that for any $X \in \mathcal{C}, a \in \mathbf{Z}^1$, $(X, a) \xrightarrow{1} (X, a)$ is legal; this can ease verification (and satisfaction) of the brute force ability assumption.

- The cycle positivity assumption is needed to ensure that we cannot make arbitrarily long progress without accumulating positive cost. The name, "cycle positivity", will be clear when we introduce the $\mathcal{C}$-Graph.

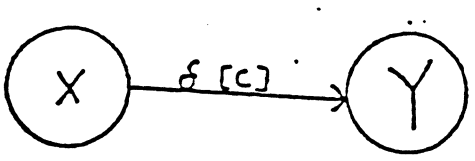


Figure 13: An arc in our $\mathcal{C}$-Graph.

2.1.2 The $\mathcal{C}$-Graph (1-Dimensional)

If our rules for movement obey the aforementioned assumptions, we can conveniently represent our problem in terms of the following Configuration-graph (abbreviated $\mathcal{C}$-Graph). Our $\mathcal{C}$-Graph consists of a vertex-set (or node-set) $\mathcal{C}$ consisting of the (finite number of) allowable configurations, and a weighted arc (or directed edge) set E, where an arc exists from node X to node Y with cost c and *progress* δ if and only if $(X, a) \xrightarrow{c} (Y, a + \delta)$ is a legal move for some $a \in \mathbf{Z}^1$ (and hence for all $a \in \mathbf{Z}^1$, by space homogeneity). In terms of our graph, the arc in Figure 13 represents the ability to move from placement (X, a) to $(Y, a + \delta)$ at cost c in a single move, for any $a \in \mathbf{Z}^1$. As before, if no c is present, then a cost of 1 is assumed. If no δ is present, then a progress of zero is assumed. Based on an earlier remark, we can assume without loss of generality, that a zero-progress, unit-cost *loop* exists from every node to itself (to accommodate the brute force assumption).

Consider the 1-dimensional forward moving jumping problem analyzed in section 1.1, specialized to the situation where we have only $p = 3$ pieces. By Claim 1.1 we need only consider four different configurations, namely:

A •••

B •• •

C • ••

D • • •

The corresponding $\mathcal{C}$-Graph appears in Figure 14 (δ values shown, $c = 1$). From the $\mathcal{C}$-Graph, we see that the brute force assumption is indeed valid with $r = 2$, and $c_\delta = 2\delta + 1$, as follows:

$$(X,0) \xrightarrow{3} (A,2) \xrightarrow{1} (C,2) \xrightarrow{1} (A,3) \xrightarrow{2} (A,4) \xrightarrow{2} (A,5) \xrightarrow{2}$$

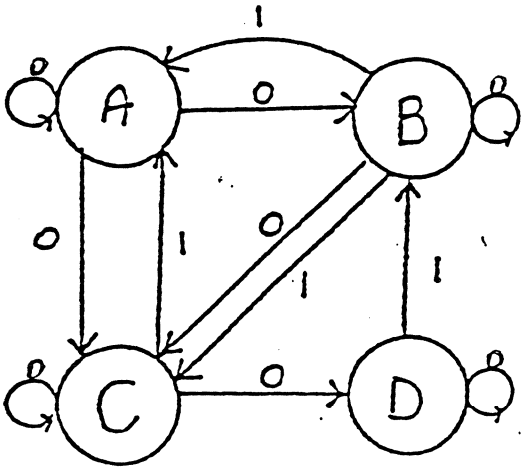


Figure 14: $\mathcal{C}$-Graph for the 3-piece, 1-dimensional jumping problem.

$$\cdots \xrightarrow{2} (A,\delta) \xrightarrow{2} (Y,\delta)$$

Notice that $(X,0) \xrightarrow{4} (A,2)$ and $(A,\delta) \xrightarrow{2} (Y,\delta)$ are possible for all X and Y (using the loops at X or Y, if necessary).

Define a *walk* on a graph to be an alternating sequence of vertices and arcs $(v_1, e_1, v_2, \ldots, e_{n-1}, v_n)$ where e_i is an arc from v_i to v_{i+1}. When the context is clear, we will omit mentioning the arcs. A walk is *closed* if $v_1 = v_n$. A closed walk is called a *cycle* (or circuit or dicycle or simple cycle) if $v_1, \ldots, v_{n-1}$ are all distinct. For a $\mathcal{C}$-Graph, we define the *(total) progress* of a walk to be the sum of the (progress) weights of the arcs of the walk, where each weight is counted as many times as the associated arc is used in the walk. Similarly, we can define the *(total) cost* of a walk. (When all arcs have unit cost, this is simply $n-1$, the number of arcs in the walk counting repetitions.)

The original problem is to reach $(\mathcal{D}, d)$ from $(\mathcal{O}, 0)$ with minimum cost, subject to our rules for movement. This is equivalent to finding a minimum cost walk from node $\mathcal{O}$ to node $\mathcal{D}$ with total progress exactly d.

Notice that a closed walk (and in particular, a cycle) beginning and ending at node X, with total progress δ and total cost c, represents the *translation* of pieces at (X,a) to $(X,a+\delta)$ with cost c for some arbitrary $a \in \mathbf{Z}^1$. Define the *speed* (or average speed or efficiency) of a cycle to be its total progress divided by its total cost. (Note that when all moves have

unit cost, this is equivalent to the cycle speed definition of Section 1.3.–i.e., the "change of center of gravity" divided by the number of moves.) In our one-dimensional setting, a *turnpike configuration* X is one that lies on a maximum-speed cycle of the $\mathcal{C}$-Graph. Recall the definition of r in the brute force assumption.

Theorem 2.1 (1-Dimensional Turnpike Theorem) *Consider the task of moving a collection of pieces over* $\mathbf{Z}^1$ *from* $(\mathcal{O},0)$ *to* $(\mathcal{D},d)$ *at minimum cost. If the rules for movement obey the previously stated assumptions, and* $d \geq 2r$, *then there exists a* turnpike *trajectory of the following form: Maneuver the pieces from* $(\mathcal{O},0)$ *into any turnpike configuration,* T. *Repeatedly translate this configuration until you are close to* $(\mathcal{D},d)$. *Then maneuver the pieces to* $(\mathcal{D},d)$. *Furthermore, the difference between the cost of this trajectory (if properly chosen) and the cost of an optimal trajectory from* $\mathcal{O}$ *to* $\mathcal{D}$ *is bounded above by a constant that does not depend on* d.

Proof. In terms of our $\mathcal{C}$-Graph, the theorem (loosely) says that we can find a near-minimum cost walk from $\mathcal{O}$ to $\mathcal{D}$ with total progress d, which spends most of its cost repeatedly traversing some one cycle of the $\mathcal{C}$-Graph.

Let C be a cycle of our $\mathcal{C}$-Graph with maximum average speed $s = p/q$. ($p > 0$ is the total progress of C; $q > 0$ is the total cost of C. Note that the finiteness, brute force, and positive cycle assumptions imply the existence of such a cycle.) Let T be an arbitrary node of C, and consider the following trajectory:

$$(\mathcal{O},0) \xrightarrow{c_r} (T,r) \xrightarrow{q} (T,r+p) \xrightarrow{q} (T,r+2p) \xrightarrow{q} \cdots \xrightarrow{q} (T,r+xp) \xrightarrow{c_\delta} (\mathcal{D},d)$$

where $x = \lfloor \frac{d-2r}{p} \rfloor$ and $\delta = d - (r + xp)$. The cost of this trajectory is $c_r + xq + c_\delta \leq c_r + c_\delta + \frac{d-2r}{s}$. This is our turnpike trajectory, which translates the configuration T through progress p (i.e., traverses cycle C) x times. Notice that $r \leq \delta \leq r + p - 1$, which does not depend on d.

Let $(\mathcal{O},0) \xrightarrow{c^*} (\mathcal{D},d)$ be a minimum cost trajectory and consider the trajectory $(\mathcal{O},0) \xrightarrow{c^*} (\mathcal{D},d) \xrightarrow{c_r} (\mathcal{O},d+r)$. This represents a closed walk (from $\mathcal{O}$ to $\mathcal{O}$) along our $\mathcal{C}$-Graph with total progress $d + r$ and total cost $c^* + c_r$.

Now we can "decompose" the arcs of any closed walk into cycles. That is, if the cycles of our $\mathcal{C}$-Graph are $C_1, \ldots, C_n$, we can find non-negative

integers $x_1, \ldots, x_n$ such that if we traverse cycle C_i x_i times, $i = 1, \ldots, n$, then every arc will be traversed exactly as many times as in the closed walk. (This can be proven by induction on the number of arcs (repetitions counted) of the walk as follows. If the closed walk is itself a cycle, we are done. Otherwise, it contains a node v which is visited twice (if the only such node is the first node, then it is re-visited before the end). Hence, our walk contains an internal closed walk which, inductively is decomposable into cycles. After removing this subwalk from our walk, the remaining walk remains closed, and this too, by induction, can be decomposed into cycles. Thus, we have decomposed our original walk into cycles.)

Thus, if our closed walk (after decomposition) traverses cycle C_i exactly x_i times, and C_i has total progress p_i and total cost $q_i > 0$ (cf. the positive cycle assumption), then our total progress is

$$d + r = \sum_{i=1}^{n} p_i x_i.$$

Our total cost is

$$c^* + c_r = \sum_{i=1}^{n} q_i x_i,$$

and so our average speed is

$$\frac{d+r}{c^* + c_r} = \frac{\sum_{i=1}^{n} p_i x_i}{\sum_{i=1}^{n} q_i x_i} \leq s,$$

where the inequality follows from $p_i \leq s q_i$. Therefore, $c^* \geq \frac{d+r}{s} - c_r$. Since the cost of our turnpike trajectory is at most $c_r + c_\delta + \frac{d-2r}{s}$, it follows that

$$\frac{d+r}{s} - c_r \leq c^* \leq \frac{d-2r}{s} + c_r + c_\delta. \tag{1}$$

Thus, the difference between the cost of our turnpike trajectory and an optimal trajectory is at most

$$c_r + c_\delta + \frac{d-2r}{s} - c^* \leq (c_r + c_\delta + \frac{d-2r}{s}) - (\frac{d+r}{s} - c_r) = 2c_r + c_\delta - \frac{3r}{s},$$

with a bound (since $\delta \leq r + p - 1$) which does not depend on d. □

Cycle	speed
AB	1/2
ABC	2/3
ABC'	1/3
AC	1/2
ACDB	2/4
BCD	2/3
BCD'	1/3

Table 1: Speeds of cycles of the three piece jumping problem.

The preceding result is analogous to theorems given by Chrétienne (1984), with non-consructive proofs in the manner of Gilmore and Gomory (1966), which imply that a "maximum valued" walk from $\mathcal{O}$ to $\mathcal{D}$ with progress d necessarily spends most of its time travelling turnpike cycles as d gets large. Those theorems were not extended to higher dimensions.

Note that we have shown the difference in cost of the optimal trajectory and our turnpike trajectory to be bounded by a constant which becomes relatively negligible as d gets large. That is, we have by equation (1)

$$\lim_{d\to\infty} \frac{c^*}{d/s} = 1.$$

Thus $c^* \approx d/s$, for large d.

2.1.3 Examples

Returning to the $\mathcal{C}$-Graph for the three piece, 1-dimensional jumping problem (see Figure 14), we notice that it contains seven simple cycles, excluding the four zero-progress loops (see Table 1). Cycles ABC' and BCD' denote cycles ABC and BCD where the zero progress arc from B to C is used instead of the unit progress arc.

The cycles ABC and BCD are turnpike cycles, with maximum speed 2/3. Thus, if we let ABC play the role of our turnpike cycle with $p = 2$ and $q = 3$ and let B be our "entering" turnpike configuration within BCD, then

our turnpike trajectory, from origin (A,0) to destination (D,99), would be

$$(A,0) \xrightarrow{5} (B,2) \xrightarrow{3} (B,4) \xrightarrow{3} (B,6) \xrightarrow{3} \cdots$$

$$\xrightarrow{3} (B,94) \xrightarrow{3} (B,96) \xrightarrow{7} (D,99)$$

with a cost of $5 + 3(47) + 7 = 153$. To illustrate the merely asymptotic nature of the "optimality" provided by such a trajectory, we observe the lesser length, $150 = 1 + 49(3) + 1 + 1$, attained (via cycle BCD) by

$$(A,0) \xrightarrow{1} (B,0) \xrightarrow{3} (B,2) \xrightarrow{3} (B,4) \xrightarrow{3} \cdots$$

$$\xrightarrow{3} (B,98) \xrightarrow{1} (C,99) \xrightarrow{1} (D,99).$$

This last trajectory is optimal, because if we could maneuver from $(A,0)$ to $(D,99)$ at a cost $c \leq 149$, then the closed walk $(A,0) \xrightarrow{c} (D,99) \xrightarrow{1} (B,100) \xrightarrow{1} (A,101)$ would have a progress/cost ratio of $101/(c+2) \geq 101/151 > 2/3$, which is impossible by Table 1.

As another example, consider the previous problem modified by the presence of a distinguished piece. The same rules apply, but now only the distinguished piece is allowed to perform a double jump. We can use the same argument as in Claim 1.1 to restrict ourselves to connected configurations. Here we have $4 \times 3 = 12$ nodes $X1, X2, X3$ depicting whether the distinguished piece is in front, middle, or back, respectively, in the configuration $X \in \{A, B, C, D\}$. In the corresponding $\mathcal{C}$-Graph (see Figure 15), the dotted lines denote arcs with progress 0, solid lines denote arcs with progress 1, and all arcs have a cost of 1. We can prove that the maximum cycle speed is $\frac{4}{7}$ as follows. First, we prove that all cycles that do not use the arc from $B3$ to $C1$ (corresponding to performing the double jump) have speed at most $\frac{1}{2}$. We see this by removing the arc from $B3$ to $C1$ and then *projecting* to the $\mathcal{C}$-Graph in Figure 16. Here we have a solid (dotted) line from node X to node Y if there exists a solid (dotted) line from X_i to Y_j for some i, j. Notice that the only simple cycle utilizing two consecutive solid lines is cycle $ACDB$, with speed $\frac{1}{2}$. All other cycles must follow a solid arc with a dotted arc and therefore have speed at most $\frac{1}{2}$ in this graph, and consequently in the original graph as well. Thus any cycle with speed greater than $\frac{1}{2}$ must use the solid arc from $B3$ to $C1$ in the original graph. By

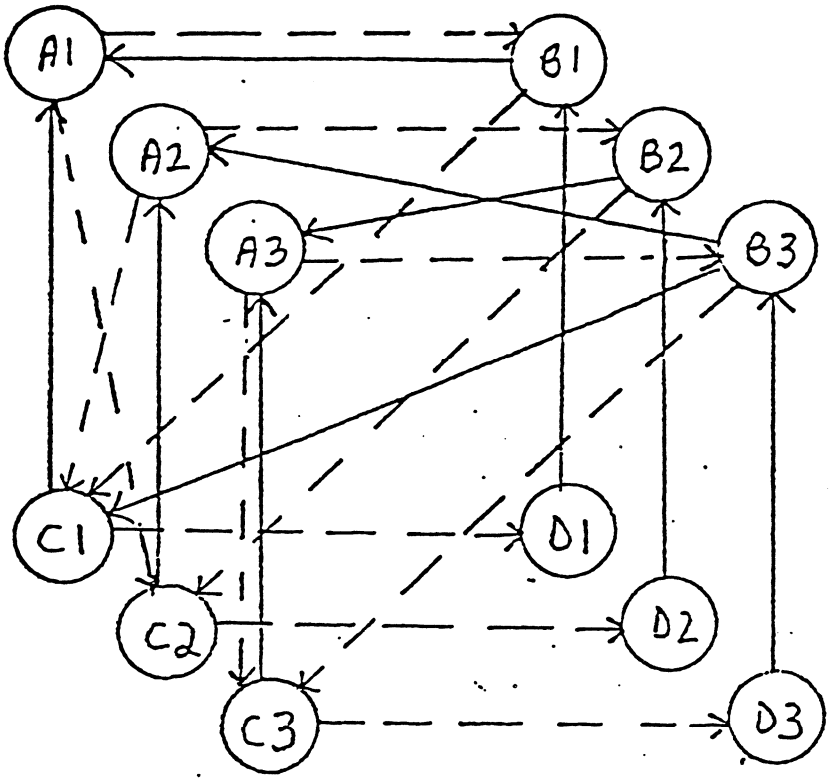


Figure 15: C-Graph for a three piece, one dimensional jumping problem with a distinguished piece (loops omitted).

branching from $C1$, we see that the minimum length path from $C1$ to $B3$ is of length 6, which by the preceding argument cannot have more dotted lines than solid. Hence the speed of the cycle is at most $\frac{1+x/2}{1+x}$, $x \geq 6$, hence at most $\frac{4}{7}$. This is attained by the cycle $C1 - A1 - C2 - A2 - B2 - A3 - B3$.

As a last, and less obvious example, consider the knapsack-type problem:

$$\text{Minimize} \sum_{j=1}^{n} f_j x_j$$

$$\text{subject to} \sum_{j=1}^{n} h_j x_j = d$$

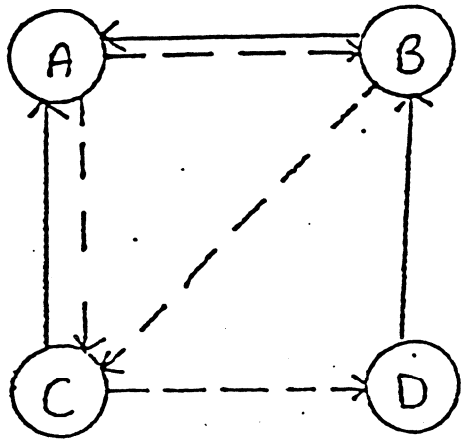


Figure 16: A "projected" problem.

$$x_j \text{ non-negative integer}$$

where we assume $h_1 = 1$ to ensure feasibility, and that $f_j, h_j > 0$ for all j. Suppose further that for all j, $\frac{h_j}{f_j} \leq \frac{h_n}{f_n}$. Then we can construct the $\mathcal{C}$-Graph with a single node X bearing n (loop) arcs, where arc j has progress h_j and cost f_j. The problem becomes that of finding a minimum cost walk from $(X, 0)$ to (X, d). (Note that $h_1 = 1$ easily gives us our brute force condition.) Our turnpike trajectory then spends most of its cost along the "minimum cycle" from X to X along the nth arc. This corresponds to the feasible solution

$$x_n = \lfloor d/h_n \rfloor, x_1 = d - h_n \lfloor d/h_n \rfloor, x_j = 0, j \neq 1, n$$

which is nearly optimal for d sufficiently large. (This is essentially the result of Gilmore and Gomory 1966.)

To make the correspondence explicit, observe that the above integer program can be represented by a simple one piece, one-dimensional problem in which our piece must move from 0 to d at minimum cost, and the jth of the n legal moves available from the single configuration propels the piece forward h_j units at a cost of f_j.

2.1.4 "One good turn deserves another", and other adages

We briefly mention a few other approaches to the 1-dimensional problem, and their drawbacks. The first natural idea is to try to prove everything using only the pigeonhole principle like this: If we only have a finite number of different configurations, then if the distance between the origin and destination is great enough, any optimal trajectory must at some point repeat the same configuration. "If it was optimal to do so the first time, then *surely* it should be optimal to do it again!" We were unable to get such an argument to work. Further, this idea would not likely extend to higher dimensional problems, since there we may want to change directions (cf. Theorem 2.2 in the next section).

2.2 Higher Dimensional Turnpike Theory

The preceding theory extends rather nicely to higher dimensions. When maneuvering our pieces over $\mathbf{Z}^m$, we make the following adjustments.

Now, $(X, \mathbf{a}) \xrightarrow{c} (Y, \bar{\mathbf{a}})$ denotes moving from configuration X placed at $\mathbf{a} \in \mathbf{Z}^m$ to configuration Y placed at $\bar{\mathbf{a}} \in \mathbf{Z}^m$ with cost $c \in \mathbf{Z}^1$. More specifically, we shall assume that $\mathbf{a} = (a_1, \ldots, a_m)$ and let $(X, \mathbf{a})$ denote that placement of X such that a_i is the minimum i^{th} coordinate among all pieces in X. (For an example, see Figure 17. As in the one dimensional case, other measures of location such as maximum coordinate, the location of some distinguishable piece, or the center of gravity will also work, and may be more natural for certain problems. (The last quantity would belong to the set $\frac{1}{p}\mathbf{Z}^m$ where p is the number of pieces.))

2.2.1 Rules-for-Movement Assumptions over $\mathbf{Z}^m$

We are interested in moving a collection of objects from one subset of $\mathbf{Z}^m$ to another at minimum cost, subject to certain restrictions on the "elementary" movements. We assume our rules for movement obey the following assumptions:

Finiteness. Without loss of optimality, we can prescribe a finite set $\mathcal{C}$ of allowable configurations for our pieces. From each configuration, there are a finite number of legal moves available.

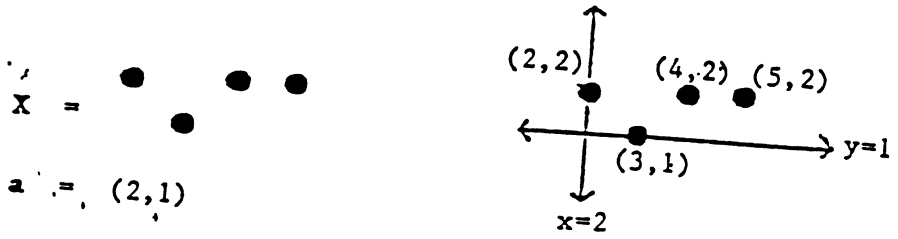


Figure 17: Configuration X and placement $(X, \mathbf{a})$.

Time, Cost and Space Homogeneity. For all $(X, \mathbf{a}) \in \mathcal{C} \times \mathbf{Z}^m$, the costs and legal moves available from $(X, \mathbf{a})$ depend only on X. That is, for all $X, Y \in \mathcal{C}, c \in \mathbf{Z}$, and $\mathbf{a}, \bar{\mathbf{a}}, \delta \in \mathbf{Z}^m$, we have $(X, \mathbf{a}) \xrightarrow{c} (Y, \bar{\mathbf{a}})$ is legal if and only if $(X, \mathbf{a} + \delta) \xrightarrow{c} (Y, \bar{\mathbf{a}} + \delta)$ is legal.

Brute Force Ability. There exist non-negative integral brute force constants $\{c_\delta\}$ such that for all $X, Y \in \mathcal{C}$, and $\mathbf{a}, \delta \in \mathbf{Z}^m$, $(X, \mathbf{a}) \xrightarrow{c_\delta} (Y, \mathbf{a} + \delta)$ is legal. In particular, $(X, \mathbf{a}) \xrightarrow{c_0} (Y, \mathbf{a})$ is legal.

Cycle Positivity. If $(X, \mathbf{a}) \xrightarrow{c} (X, \bar{\mathbf{a}})$ is legal, then $c \geq 0$. If $\mathbf{a} \neq \bar{\mathbf{a}}$, then $c > 0$.

The remarks following the one dimensional assumptions remain valid. We are assuming that our desired destination from $(\mathcal{O}, \mathbf{0})$ is $(\mathcal{D}, d\mathbf{b})$ where $\mathcal{D} \in \mathcal{C}$, d is a (large) positive integer and $\mathbf{b} \geq \mathbf{0}$. If the last condition fails, we can reverse the axes of the offending coordinates without loss of generality. Notice that here we are using a stronger brute force assumption than in the one dimensional version. We shall say more about this after the proof of the next theorem.

2.2.2 The $\mathcal{C}$-Graph (m-dimensional version)

The $\mathcal{C}$-Graph for the m-dimensional problem is similar to the 1-dimensional $\mathcal{C}$-Graph. Here, an arc is present from node X to node Y, with cost $c \in \mathbf{Z}^1$ and progress $\delta \in \mathbf{Z}^m$, if and only if in a single move, we can move from configuration $(X, \mathbf{a})$ to $(Y, \mathbf{a} + \delta)$ at cost c for any $\mathbf{a} \in \mathbf{Z}^1$. As before, if no c

is present, then a cost of 1 is assumed. We shall usually assume that a zero-progress, unit cost loop exists from every node to itself, to accommodate the brute force assumption.

Also, as before, a closed walk from node X to X represents a translation of configuration X, with total progress and total cost defined, respectively, as the sum of the walk-arcs' cost weights and the (vector) sum of their progress weights. Determining a minimum cost trajectory from $(\mathcal{O}, \mathbf{0})$ to $(\mathcal{D}, d\mathbf{b})$, $d > 0$, $\mathbf{b} \geq \mathbf{0}$, is equivalent to finding a minimum cost walk in our $\mathcal{C}$-Graph from node $\mathcal{O}$ to node $\mathcal{D}$ with total progress $d\mathbf{b}$. If $\mathcal{O} = \mathcal{D}$, the walk is closed.

Theorem 2.2 (m-Dimensional Turnpike Theorem) *Consider the problem of moving a collection of pieces over $\mathbf{Z}^m$ from $(\mathcal{O}, \mathbf{0})$ to $(\mathcal{D}, d\mathbf{b})$ at minimum cost. If the rules for movement obey the previously stated assumptions, then there exists a* turnpike *trajectory of the following form. Letting $\mathcal{O} = T_0$, proceed as follows. For $i = 0, \ldots, m-1$, brute force maneuver from configuration T_i to an appropriate configuration T_{i+1}, then repeatedly translate T_{i+1} x_{i+1} times , x_{i+1} an appropriate non-negative integer. Then brute force maneuver from T_m to $\mathcal{D}$. Furthermore, the difference between the cost of this trajectory (if appropriately chosen) and that of an optimal trajectory is bounded above by a constant that does not depend on d or $\mathbf{b}$.*

Proof. In terms of our $\mathcal{C}$-Graph, the theorem states that we can find a near minimum cost walk from $\mathcal{O}$ to $\mathcal{D}$ with total progress $d\mathbf{b}$, which spends most of its cost repeatedly traversing m particular cycles of the $\mathcal{C}$-Graph.

Suppose the cycles of our $\mathcal{C}$-Graph are cycles $C^1, \ldots, C^n$, where for $i = 1, \ldots, n$, cycle C^i has total progress $\mathbf{a}_i \in \mathbf{Z}^m$ and total cost $c^i > 0$ (not to be confused with our brute force constants c_δ).

Let A denote the $m \times n$ matrix with i^{th} column $\mathbf{a}_i$, $i = 1, \ldots, n$. We now use the brute force assumption to prove that A has full row rank, as follows. Consider any $X \in \mathcal{C}$, and any vector $\mathbf{v} \in \mathbf{Z}^m$. By the brute force assumption there is a closed walk from $(X, \mathbf{0})$ to $(X, \mathbf{v})$. As shown while proving Theorem 2.1, this closed walk can be decomposed into cycles. Therefore $\mathbf{v}$ can be expressed as a (non-negative integral) linear combination of $\mathbf{a}_1, \ldots, \mathbf{a}_n$, $i = 1, \ldots, n$. Hence any integral vector $\mathbf{v}$ can be expressed as a non-negative integral combination of some of the $\mathbf{a}_i$'s. Thus, A has full row rank.

Let $M = \max_{i,j} |a_{i,j}|$, and let $\mathbf{e}^T = (1, \ldots, 1) \in \mathbf{Z}^m$. For any linearly independent set of m column vectors $\{\mathbf{a}_{i_1}, \ldots, \mathbf{a}_{i_m}\}$, we can express $d\mathbf{b}$ as a *linear* combination of these vectors in precisely one way (namely as $d\mathbf{b} = \sum_{j=1}^{m} \mathbf{a}_{i_j} x_j$, where $x_j = (B^{-1} d\mathbf{b})_j \in \mathbf{Q}$, where $\mathbf{a}_{i_j}$ is the j^{th} column of B). If $\mathbf{x} = (x_1, \ldots, x_m)$ is non-negative, we say that the basis $B = \{\mathbf{a}_{i_1}, \ldots, \mathbf{a}_{i_m}\}$ is *feasible*, and has total cost

$$\sum_{j=1}^{m} c^{i_j} x_j = \sum_{j=1}^{m} c^{i_j} (B^{-1} d\mathbf{b})_j \equiv \mathbf{c}_B^T B^{-1} d\mathbf{b}.$$

Now since $(\mathcal{O}, \mathbf{0}) \xrightarrow{c_{d\mathbf{b}}} (\mathcal{O}, d\mathbf{b})$ is legal and decomposable into cycles, the system $A\mathbf{x} = d\mathbf{b}$ has a feasible solution. Therefore, by the Fundamental Theorem of Linear Programming (see, for instance, Chvátal 1983), A must contain at least one feasible basis. The number of feasible bases is finite; assume for ease of notation that $\{\mathbf{a}_1, \ldots, \mathbf{a}_m\}$ is a feasible basis with minimum total cost $d\mathbf{c}_B^T B^{-1} \mathbf{b}$, where $B = [\mathbf{a}_1, \ldots, \mathbf{a}_m]$ and $\mathbf{c}_B^T = (c^1, \ldots, c^m)$. Let $x_i = (dB^{-1}\mathbf{b})_i \in \mathbf{Q}_+$. Let T_i be an arbitrary configuration node on cycle C^i, $i = 1, \ldots, m$. Then our turnpike trajectory can be constructed as follows:

$$(\mathcal{O}, \mathbf{0}) \xrightarrow{c_{\mathbf{0}}} (T_1, \mathbf{0}) \xrightarrow{c^1 \lfloor x_1 \rfloor} (T_1, \lfloor x_1 \rfloor \mathbf{a}_1) \xrightarrow{c_{\mathbf{0}}} (T_2, \lfloor x_1 \rfloor \mathbf{a}_1) \xrightarrow{c^2 \lfloor x_2 \rfloor}$$

$$(T_2, \lfloor x_1 \rfloor \mathbf{a}_1 + \lfloor x_2 \rfloor \mathbf{a}_2) \xrightarrow{c_{\mathbf{0}}} \cdots \xrightarrow{c_{\mathbf{0}}} (T_i, \sum_{j=1}^{i-1} \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c^i \lfloor x_i \rfloor} (T_i, \sum_{j=1}^{i} \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c_{\mathbf{0}}}$$

$$\cdots \xrightarrow{c_{\mathbf{0}}} (T_m, \sum_{j=1}^{m-1} \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c^m \lfloor x_m \rfloor} (T_m, \sum_{j=1}^{m} \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c_\delta} (\mathcal{D}, d\mathbf{b})$$

where

$$\delta = d\mathbf{b} - \sum_{j=1}^{m} \lfloor x_j \rfloor \mathbf{a}_j = d\mathbf{b} - d\mathbf{b} + \sum_{j=1}^{m} \mathbf{a}_j f_j = \sum_{j=1}^{m} (\mathbf{a}_j f_j),$$

and $f_j \equiv x_j \bmod 1$, $0 \leq f_j < 1$, i.e., $f_j = x_j - \lfloor x_j \rfloor$. Consequently, $-mM\mathbf{e} \leq \delta \leq mM\mathbf{e}$, and since $\{\delta \in \mathbf{Z}^m : \|\delta\|_\infty \leq mM\mathbf{e}\}$ is a finite set, c_δ is bounded above by a constant $\bar{c}$ that does not depend on d or $\mathbf{b}$. The total cost of this turnpike trajectory is

$$\text{TPCOST} = mc_{\mathbf{0}} + \sum_{i=1}^{m} c^i \lfloor x_i \rfloor + c_\delta \leq mc_{\mathbf{0}} + \sum_{i=1}^{m} c^i \lfloor x_i \rfloor + \bar{c}. \qquad (2)$$

Notice that by construction, $\mathbf{x} = (x_1, \ldots, x_n)^T$ is a (basic feasible) optimal solution to the following linear program:

$$u^* = \min_{\mathbf{x}} \sum_{i=1}^{n} c^i x_i$$

$$\text{subject to } A\mathbf{x} = d\mathbf{b}, \mathbf{x} \geq \mathbf{0}$$

Furthermore, if c^* denotes the minimum cost to reach $(\mathcal{D}, d\mathbf{b})$ from $(\mathcal{O}, \mathbf{0})$, then c^* cannot exceed the cost of the turnpike trajectory. Hence, by equation (2), we must have

$$c^* \leq \text{TPCOST} \leq mc_{\mathbf{0}} + \bar{c} + u^*. \tag{3}$$

On the other hand, consider some trajectory $(\mathcal{O}, \mathbf{0}) \xrightarrow{c^*} (\mathcal{D}, d\mathbf{b})$ with minimum cost c^*. Since the trajectory $(\mathcal{O}, \mathbf{0}) \xrightarrow{c^*} (\mathcal{D}, d\mathbf{b}) \xrightarrow{c_{\mathbf{0}}} (\mathcal{O}, d\mathbf{b})$ is a closed walk on our $\mathcal{C}$-Graph, it can be decomposed into cycles. Thus, $d\mathbf{b} = \sum_{j=1}^{n} \mathbf{a}_j y_j$ for some non-negative integers y_j, $j = 1, \ldots, n$. It follows that

$$\begin{array}{ccccc} c^* + c_{\mathbf{0}} & \geq & \min \sum_{j=1}^{n} c^j x_j & \geq & \min \sum_{j=1}^{n} c^j x_j \\ & & \text{s.t. } A\mathbf{x} = d\mathbf{b} & & \text{s.t. } A\mathbf{x} = d\mathbf{b} \\ & & \mathbf{x} \geq \mathbf{0} \text{ integer} & & \mathbf{x} \geq \mathbf{0} \end{array} \quad = \quad u^*$$

That is,

$$c^* \geq -c_{\mathbf{0}} + u^*. \tag{4}$$

Combining relations (3) and (4), we have

$$-c_{\mathbf{0}} + u^* \leq c^* \leq \text{TPCOST} \leq mc_{\mathbf{0}} + \bar{c} + u^*.$$

Consequently,

$$\text{TPCOST} - c^* \leq (m+1)c_{\mathbf{0}} + \bar{c}, \tag{5}$$

which does not depend on d or $\mathbf{b}$. □

By analogy to the 1-dimensional turnpike theorem, we now proceed to prove a similar result using a weaker brute force assumption. First, we point out that the apparent weakening,"There exist $r \geq 0$ and non-negative brute force constants $\{c_i : i \geq r\}$ such that for all $X, Y \in \mathcal{C}$ and $\mathbf{a} \in \mathbf{Z}^m$, $(X, \mathbf{a}) \xrightarrow{c_{\|\delta\|}} (Y, \mathbf{a} + \delta)$ is legal whenever $\|\delta\| \geq r$", is actually

equivalent to the current assumption, since for any $X, Y \in \mathcal{C}$, $\mathbf{a}, \delta \in \mathbf{Z}^m$, with $\|\delta\| < r$, $(X, \mathbf{a}) \xrightarrow{c_t} (Y, \mathbf{a} + (1 + r/\|\delta\|)\delta) \xrightarrow{c_\mathbf{r}} (Y, \mathbf{a} + \delta)$ would be legal, where $t = \|\delta\| + r$.

However, if we assume that $\mathbf{b} > 0$, then we can prove the previous theorem under a genuinely weaker brute force assumption, namely: There exist $\mathbf{r} \geq \mathbf{0}_m$ and non-negative brute force constants $\{c_\delta\colon\ \delta \geq \mathbf{r}\}$, such that for all $X, Y \in \mathcal{C}$, $\mathbf{a} \in \mathbf{Z}^m$, and $\delta \geq \mathbf{r}$, $(X, \mathbf{a}) \xrightarrow{c_\delta} (Y, \mathbf{a} + \delta)$ is legal. This is analogous to the earlier one dimensional brute force assumption, and is motivated by the desire to include rules for movement where we are restricted to move only in "forwards" directions.

Theorem 2.3 *When $\mathbf{b} > 0$, and d is sufficiently large, Theorem 2.2 is true under the weaker brute force assumption above.*

Proof. As before, let the cycles of our $\mathcal{C}$-Graph be $C^1, \ldots, C^n$, where for $i = 1, \ldots, n$, cycle C^i has total progress $\mathbf{a}_i \in \mathbf{Z}^m$ and total cost $c^i > 0$, and let A denote the $m \times n$ matrix with i^{th} column $\mathbf{a}_i$, $i = 1, \ldots, n$. Since $\{\mathbf{v} \in \mathbf{Z}^m : \mathbf{v} \geq \mathbf{r}\}$ has dimension m, and (by the weak brute force assumption) lies in the column span of A, A has full row rank.

Let $M = \max_{i,j} |a_{i,j}|$, and let $\mathbf{e}^T = (1, \ldots, 1) \in \mathbf{Z}^m$. Let $\hat{\mathbf{b}} = d\mathbf{b} - mM\mathbf{e} - (m+1)\mathbf{r}$. As in the previous proof, for any linearly independent set of m column vectors $\{\mathbf{a}_{i_1}, \ldots, \mathbf{a}_{i_m}\}$, we can express $\hat{\mathbf{b}}$ as a linear (though not necessarily integer) combination of these vectors in precisely one way.

Since $\mathbf{b} > 0$, we must have $\hat{\mathbf{b}} > \mathbf{r}$ for sufficiently large d. Thus $(\mathcal{O}, \mathbf{0}) \xrightarrow{c_{\hat{\mathbf{b}}}} (\mathcal{O}, \hat{\mathbf{b}})$ is legal and decomposable into cycles, so that A must contain at least one feasible basis for the system $A\mathbf{x} = \hat{\mathbf{b}}$. The number of feasible bases is finite; assume for ease of notation that $\{\mathbf{a}_1, \ldots, \mathbf{a}_m\}$ is a feasible basis with minimum total cost $\mathbf{c}_B^T B^{-1} \hat{\mathbf{b}}$, where $B = [\mathbf{a}_1, \ldots, \mathbf{a}_m]$ and $\mathbf{c}_B^T = (c^1, \ldots, c^m)$. Let $x_i = (B^{-1}\hat{\mathbf{b}})_i \in \mathbf{Q}_+$. Let T_i be an arbitrary configuration node on cycle C^i, $i = 1, \ldots, m$. Then our turnpike trajectory can be constructed as follows:

$$(\mathcal{O}, \mathbf{0}) \xrightarrow{c_\mathbf{r}} (T_1, \mathbf{r}) \xrightarrow{c^1 \lfloor x_1 \rfloor} (T_1, \mathbf{r} + \lfloor x_1 \rfloor \mathbf{a}_1) \xrightarrow{c_\mathbf{r}} (T_2, 2\mathbf{r} + \lfloor x_1 \rfloor \mathbf{a}_1) \xrightarrow{c^2 \lfloor x_2 \rfloor}$$

$$(T_2, 2\mathbf{r} + \lfloor x_1 \rfloor \mathbf{a}_1 + \lfloor x_2 \rfloor \mathbf{a}_2) \xrightarrow{c_\mathbf{r}} \cdots \xrightarrow{c_\mathbf{r}} (T_i, i\mathbf{r} + \sum_{j=1}^{i-1} \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c^i \lfloor x_i \rfloor} (T_i, i\mathbf{r} + \sum_{j=1}^{i} \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c_\mathbf{r}}$$

$$\cdots \xrightarrow{c\mathbf{r}} (T_m, m\mathbf{r} + \sum_{j=1}^{m-1} \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c^m \lfloor x_m \rfloor} (T_m, m\mathbf{r} + \sum_{j=1}^{m} \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c_\delta} (\mathcal{D}, d\mathbf{b})$$

where

$$\begin{aligned}
\delta &= d\mathbf{b} - (m\mathbf{r} + \sum_{j=1}^{m} \lfloor x_j \rfloor \mathbf{a}_j) \\
&= d\mathbf{b} - m\mathbf{r} - (\hat{\mathbf{b}} - \sum_{j=1}^{m} \mathbf{a}_j f_j) \text{ (where } f_j \equiv x_j \bmod 1,\ 0 \leq f_j < 1) \\
&= d\mathbf{b} - m\mathbf{r} + \sum_{j=1}^{m} \mathbf{a}_j f_j - d\mathbf{b} + (m+1)\mathbf{r} + mM\mathbf{e} \\
&= \mathbf{r} + \sum_{j=1}^{m} (\mathbf{a}_j f_j) + mM\mathbf{e}.
\end{aligned}$$

Consequently, $\mathbf{r} \leq \delta \leq \mathbf{r} + 2mM\mathbf{e}$, and since $\{\delta \in \mathbf{Z}^m : \mathbf{r} \leq \delta \leq \mathbf{r} + 2mM\mathbf{e}\}$ is a finite set, c_δ is bounded above by a constant $\bar{c}$ that does not depend on d or $\mathbf{b}$. The total cost of the turnpike trajectory is

$$\text{TPCOST} = mc_{\mathbf{r}} + \sum_{i=1}^{m} c^i \lfloor x_i \rfloor + c_\delta \leq mc_{\mathbf{r}} + \sum_{i=1}^{m} c^i \lfloor x_i \rfloor + \bar{c}. \tag{6}$$

Notice that by construction, $\mathbf{x} = (x_1, \ldots, x_n)^T$ is a (basic feasible) optimal solution to the following linear program:

$$u^* = \min_{\mathbf{x}} \sum_{i=1}^{n} c^i x_i$$

$$\text{subject to } A\mathbf{x} = \hat{\mathbf{b}}, \mathbf{x} \geq \mathbf{0}$$

Furthermore, if c^* denotes the minimum cost to reach $(\mathcal{D}, d\mathbf{b})$ from $(\mathcal{O}, \mathbf{0})$, then c^* cannot exceed the cost of the turnpike trajectory. Hence, by equation (6), we must have

$$c^* \leq \text{TPCOST} \leq mc_{\mathbf{r}} + \bar{c} + u^*. \tag{7}$$

On the other hand, consider some trajectory $(\mathcal{O}, \mathbf{0}) \xrightarrow{c^*} (\mathcal{D}, d\mathbf{b})$ with minimum cost c^*. Since the trajectory $(\mathcal{O}, \mathbf{0}) \xrightarrow{c^*} (\mathcal{D}, d\mathbf{b}) \xrightarrow{c\mathbf{r}} (\mathcal{O}, d\mathbf{b} + \mathbf{r})$

is a closed walk on our $\mathcal{C}$-Graph, it can be decomposed into cycles. Thus, setting $\overline{\mathbf{b}} = d\mathbf{b} + \mathbf{r}$, we have $\overline{\mathbf{b}} = \sum_{j=1}^{n} \mathbf{a}_j y_j$ for some non-negative integers y_j, $j = 1, \ldots, n$. It follows that

$$\begin{array}{ccccc} c^* + c_{\mathbf{r}} & \geq & \min \sum_{j=1}^{n} c^j x_j & \geq & \min \sum_{j=1}^{n} c^j x_j \\ & & \text{s.t. } A\mathbf{x} = \overline{\mathbf{b}} & & \text{s.t. } A\mathbf{x} = \overline{\mathbf{b}} \\ & & \mathbf{x} \geq \mathbf{0} \text{ integer} & & \mathbf{x} \geq \mathbf{0} \end{array} \equiv z^*$$

That is,

$$c^* \geq -c_{\mathbf{r}} + z^*. \tag{8}$$

Combining relations (7) and (8), we have

$$-c_{\mathbf{r}} + z^* \leq c^* \leq \text{TPCOST} \leq mc_{\mathbf{r}} + \bar{c} + u^*.$$

Consequently,

$$\text{TPCOST} - c^* \leq (m+1)c_{\mathbf{r}} + \bar{c} + u^* - z^*. \tag{9}$$

But u^* and z^* denote optimal objective-function values to linear programs with parameters $(A, \hat{\mathbf{b}}, \mathbf{c})$ and $(A, \overline{\mathbf{b}}, \mathbf{c})$ respectively. Now by theorem (2.4) of Mangasarian and Shiau 1987, there exist optimal solutions $\hat{\mathbf{x}}$ and $\overline{\mathbf{x}}$ to the above linear programs satisfying

$$\|\hat{\mathbf{x}} - \overline{\mathbf{x}}\|_\infty \leq k_A \|\hat{\mathbf{b}} - \overline{\mathbf{b}}\|_\infty,$$

where k_A is a constant depending only on A. Thus,

$$\begin{aligned} u^* - z^* &= \mathbf{c}^T\hat{\mathbf{x}} - \mathbf{c}^T\overline{\mathbf{x}} \\ &= \mathbf{c}^T(\hat{\mathbf{x}} - \overline{\mathbf{x}}) \\ &\leq \|\mathbf{c}\|_\infty \|\hat{\mathbf{x}} - \overline{\mathbf{x}}\|_\infty \\ &\leq k_A \|\mathbf{c}\|_\infty \|\hat{\mathbf{b}} - \overline{\mathbf{b}}\|_\infty \\ &= k_A \|\mathbf{c}\|_\infty \|d\mathbf{b} - mM\mathbf{e} - (m+1)\mathbf{r} - (d\mathbf{b} + \mathbf{r})\|_\infty \\ &= k_A \|\mathbf{c}\|_\infty \|mM\mathbf{e} + (m+2)\mathbf{r}\|_\infty, \end{aligned} \tag{10}$$

which does not depend on d or $\mathbf{b}$.

Consequently, by relations (9) and (10), we see that the difference between the cost of our turnpike trajectory and the minimum cost trajectory is bounded by a constant which does not depend on d or $\mathbf{b}$. □

Remarks:

- The theorem of Mangasarian and Shiau is stronger than we need. The theorem says that there exist two *solutions* to the linear programs which are close together, when it suffices to show that their objective function values are close together. It would be interesting to see if this extra information could be exploited to yield stronger results.

- $\mathcal{C}$-Graphs were introduced as a tool for modeling, analyzing, and solving maneuvering problems. However, the previous theorems can be applied to any problem which can be transformed into a $\mathcal{C}$-Graph question of the form just analyzed (i.e., given a $\mathcal{C}$-Graph, nodes $\mathcal{O}$ and $\mathcal{D}$, and a progress vector $d\mathbf{b} \in \mathbf{Z}^m$, find a minimum cost walk from $\mathcal{O}$ to $\mathcal{D}$ with total progress exactly $d\mathbf{b}$). By applying simple modifications to the proofs of Theorems 2.2 and 2.3, we can construct almost optimal turnpike solutions to other $\mathcal{C}$-Graph questions as well, as in the following three remarks which were motivated by the non-maneuvering applications discussed in Chapter 6.

- If the origin and/or destination node are unspecified, then we choose $\mathcal{O}$ and $\mathcal{D}$ arbitrarily and construct the same trajectory as before. If an optimal trajectory uses endpoints $\mathcal{O}^*$ and $\mathcal{D}^*$, then by brute forcing from $(\mathcal{D}^*, d\mathbf{b})$ to $(\mathcal{O}^*, d\mathbf{b})$ with cost $c_{\mathbf{0}}$, we can decompose into cycles, and proceed as before.

- If the $\mathcal{C}$-Graph problem contains inequality constraints, then the proofs of Theorems 2.2 and 2.3 can be modified in a straightforward manner. Specifically, suppose our problem requires us to maneuver at near minimum cost, from $(\mathcal{O}, \mathbf{0})$ to a *variable* terminal placement, $(\mathcal{D}, \Delta)$ where $\mathcal{D}$ is specified and $\mathbf{\Delta} = (\mathbf{\Delta}_1, \mathbf{\Delta}_2, \mathbf{\Delta}_3)^T$ must satisfy $\mathbf{\Delta}_1 \geq d\mathbf{b}_1$, $\mathbf{\Delta}_2 = d\mathbf{b}_2$, $\mathbf{\Delta}_3 \leq d\mathbf{b}_3$. Then as before we are led to solve the linear program $\min \mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} = \hat{\mathbf{b}}$, $\mathbf{x} \geq 0$, where the appropriate choice of $\hat{\mathbf{b}}$ differs between Theorems 2.2 and 2.3. The construction, brute forcing and proof are similar to those already given; the details of the necessary modifications are given in Appendix C.

- If we have to solve a *maximization* problem, say where each arc has vector progress and a scalar *value*, a desirable quantity, then we have to make some adjustments so that the affiliated LP is not unbounded.

Naturally, if all of our cycles had negative value, then we could negate the value of each arc and solve the minimization problem as before, yet this scenario is not in the spirit of arcs having positive value. Suppose, as in Theorem 2.3 with some $\mathbf{r} \geq \mathbf{0}_m$, that for any $X, Y \in \mathcal{C}$, and $\delta \geq \mathbf{r}$, we can brute force our way from node X to node Y making progress exactly δ with a value of c_δ. (Perhaps $c_\delta \leq 0$.) Thus, we can write the $\mathcal{C}$-Graph version of the brute force assumption as before. However, in order to keep the affiliated LP bounded, we replace the cycle positivity assumption with the following

Cycle Positivity (Progress) All cycles with positive value have (vector) positive total progress.

In this manner, the affiliated linear program $\max \mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} = d\mathbf{b}$, $\mathbf{x} \geq \mathbf{0}$, remains bounded, as before, since $x_i \leq \min_{j=1\ldots m} db_j/a_{ij}$ whenever $c_i > 0$. (In the case where the LP has inequality constraints, it too is bounded unless all the constraints are of the form $A\mathbf{x} \geq \mathbf{b}$.) Theorems 2.2 and 2.3 (as well as the material in Appendix C) generalize: if $\mathbf{x}^* = (x_1^*, \ldots, x_m^*)$ is a basic optimal solution to the above LP, then our turnpike trajectory can be expressed exactly as before. We get TPVALUE $\geq mc_\mathbf{r} + \bar{c} - \sum_{i=1}^m |c_i| + u^*$, where $\bar{c}$ is a lower bound to a finite set, and u^* is the maximum value to the above LP. By analogy, we have $c^* + c_r \leq z^*$, and the result follows, as in the last proof.

Chapter 3
Implementation Ideas

The construction of the turnpike trajectories of the last chapter depended on the solution of a linear program (LP) based on the maneuvering problem's $\mathcal{C}$-Graph representation. To approach such a problem *directly* through linear programming would require an enumeration of all cycles in the $\mathcal{C}$-Graph, corresponding to the individual variables of the LP. This may be prohibitively difficult; for instance, if our $\mathcal{C}$-Graph contains a complete directed graph on n vertices, there will be more than $(n-1)!$ cycles (the number of traveling salesman tours). Assuming, for now, that our $\mathcal{C}$-Graph is itself of manageable size, (i.e., a representation of it can be stored in a computer), we present in our first section a "column generation " approach to solving the linear program without enumeration of cycles. The procedure relies heavily on efficient detection of negative cycles in a weighted directed graph. We discuss the theory of detecting negative cycles in the following section, and make some suggestions for improving the efficiency of a standard method. In section three, we provide some illustrative computational results. Section four sketches alternative approaches to the $\mathcal{C}$-Graph problem, including ideas for how to treat an unmanageably large $\mathcal{C}$-Graph. In the last section, we bound the possibilities for rapid solution by showing that a decision problem, naturally associated with our $\mathcal{C}$-Graph problem, is NP-complete.

3.1 Column Generation

For a one-dimensional maneuvering problem, the i-th arc of its $\mathcal{C}$-Graph has an associated cost γ_i and progress p_i, both scalar quantities. With the goal of making long-term efficient forward movement, our turnpike trajectory requires us to determine a *forward cycle* C (i.e., $\sum_{i\in C} p_i > 0$) with minimum average cost $\sum_{i\in C} \gamma_i / \sum_{i\in C} p_i$.

This problem can be solved without explicitly enumerating all the cycles of the $\mathcal{C}$-Graph, by the following method, similar to one utilized by Lawler (1966). Recall that all cycles of the $\mathcal{C}$-Graph are assumed to have positive cost, i.e., $\sum_{i\in C} \gamma_i > 0$ for all cycles C.

Start with any forward cycle (one must exist by the brute force assumption); say it has average cost $\lambda > 0$. Assign each arc in the C-Graph (not just in the cycle) the weight $\gamma_i - \lambda p_i$. Using your favorite method (e.g., the one discussed in the next section), find a cycle of negative weight or determine that none exists. If no such cycle exists, then every forward cycle C satisfies $\sum_{i \in C}(\gamma_i - \lambda p_i) \geq 0$, i.e., $\sum_{i \in C} \gamma_i / \sum_{i \in C} p_i \geq \lambda$, and our current cycle is optimal. Otherwise, if some cycle C has negative weight, then $\sum_{i \in C}(\gamma_i - \lambda p_i) < 0$. C is necessarily a forward cycle, since otherweise $\sum_{i \in C} \gamma_i < \lambda \sum_{i \in C} p_i \leq 0$, contradicting the positive cycle (cost) assumption. Hence, $0 < \sum_{i \in C} \gamma_i / \sum_{i \in C} p_i < \lambda$, and C is a forward cycle with smaller average cost. The process can now be repeated, with C becoming our current cycle. According to Lawler (1976, p.97) this procedure has a polynomial ($O(n^6)$) worst case time complexity, where n is the number of nodes, and the weights of the arcs do not grow with n.

We now generalize the above procedure to obtain a solution method for higher dimensional maneuvering problems. Here, arc i has scalar cost γ_i, and *vector* progress $\mathbf{p}_i \in \mathbf{Z}^m$, where m is the dimension of our movement environment. The j-th cycle (call it C) has *total cost* $c^j = \sum_{i \in C} \gamma_i$, assumed to be positive, and *total progress* $\mathbf{a}_j = \sum_{i \in C} \mathbf{p}_i \in \mathbf{Z}^m$. If our desired direction of movement is $\mathbf{b} \geq \mathbf{0}_m$, then from chapter two, we must solve

$$\min \sum_{j=1}^{n} c^j x_j$$

$$\text{s.t. } A\mathbf{x} = \mathbf{b},\ \mathbf{x} \geq \mathbf{0} \qquad (1)$$

where x_j (after truncation) denotes the number of times our turnpike trajectory utilizes the j-th cycle.

As before, we wish to solve this linear program without explicitly enumerating all the columns of our matrix A, representing cycles of our C-Graph. We now present a *column generation* scheme, analogous to the procedure above, to solve (1) directly. If necessary, begin with an initial artificial basis consisting of m "artificial cycles", each making one unit of progress in a unit direction at enormous cost. Using the columns of our A matrix generated so far, solve the linear program to optimality. Let $\mathbf{x} \in \mathbf{R}^n$ and $\lambda \in \mathbf{R}^m$ be the primal and dual solutions to this restriction of (1) to the generated columns (i.e., $x_j = 0$ if the j-th column has not been generated). From duality theory, $\mathbf{x}$ is an optimal solution to (1) if and only if

$c^j - \lambda^T \mathbf{a}_j \geq 0, j = 1 \ldots n$. This can be determined directly on our $\mathcal{C}$-Graph by assigning the i-th arc a weight of $\gamma_i - \lambda^T \mathbf{p}_i, i = 1 \ldots \|E\|$, and looking for a cycle of negative weight. If no such cycle exists, $\mathbf{x}$ solves (1); otherwise a negative cycle is generated, and its associated progress column and (unadjusted) cost is added to our set of generated columns. The new LP is solved, and the procedure is repeated. The implementation and computational experience with this method are the subjects of the next two sections.

There is also the question of increasing the efficiency of column-generation methods by judicious column-*dropping*. The ideas discussed in O'Neill & Wildhelm (1976) and O'Neill (1977) may be useful.

Incidentally, contrary to the sequence of presentation here, this higher dimensional procedure was formulated before its one-dimensional special case. It was inspired by a paper of Ford and Fulkerson, 1958.

3.2 Finding Negative Cycles

Here, we address the issue of finding a *negative cycle* in a weighted directed graph, i.e., a cycle whose arcs have weights which sum to a negative number. In addition to our $\mathcal{C}$-Graph application, detecting the existence (or non-existence, really) of a negative cycle is evidently important for the application of many shortest path algorithms, since a "shortest" walk or path may not be well defined when such a cycle exists. Ironically, these very shortest path algorithms can be modified so as to detect and determine negative cycles. In this section, we explain how the well-known Bellman-Ford method (see Lawler 1976), for computing shortest paths from a specific origin node, can be so modified. In addition, we suggest some ideas for accelerating the computations.

For digraphs without negative cycles, the Bellman-Ford shortest path algorithm computes shortest-path lengths from node 1 to all nodes $1, \ldots, n$, as follows:

Begin
Line (0): Initially, $u_0(1) := 0$, $u_0(i) := \infty$, $i = 2, \ldots, n$.
Line (1): For $k := 1$ to $n - 1$ Do
Line (2): For $i := 1$ to n Do

Line (3): For $j := 1$ to n Do
Line (4): $u_k(j) := \min\{u_{k-1}(j), u_{k-1}(i) + c_{ij}\}$
End.

Here c_{ij} (usually interpreted as some kind of cost) is the weight of *the* arc from node i to node j. (When more than one arc exists from i to j, c_{ij} denotes the cost of a cheapest one.) By induction, it is clear that $u_k(i)$ is the length of a shortest *k-walk* from node 1 to node i, where a k-walk is a walk that takes no more than k steps. In the absence of any negative cycles, a shortest walk need never repeat any nodes, and so can be assumed to be a path requiring at most $n - 1$ steps.

In practice, using a linked-list data structure, Line (3) would be replaced by "For all $j \in A(i)$ Do", where $A(i) \subseteq E$ is the set of arcs with tail i. Hence the algorithm has time complexity $O(n|E|)$. Now suppose that after running this algorithm, we set $k = n$, and execute lines (2)-(4) again. If for some j, we have *improvement* in the sense that $u_n(j) < u_{n-1}(j)$, then there exists a walk from node 1 to node j which costs less than any path from node 1 to node j. Hence that walk, and so the graph itself, must contain a negative cycle that is reachable from node 1. We observe here that all nodes on that cycle will be improved infinitely often as further iterations proceed.

Conversely, if after some iteration (and, in particular, after the n-th iteration) we have *no* improvement, then no subsequent iteration will yield improvement. For, if there was no improvement at the k-th iteration, then for all arcs ij, $u_{k-1}(j) \leq u_{k-1}(i) + c_{ij}$, and the same $|E|$ comparisons will be made each subsequent iteration: no further improvement is possible. Thus, by the observation of the previous paragraph, if a graph contains a negative cycle (that is reachable from node 1) then there will be improvement at *every* iteration.

From the above discussion we can conclude that if all nodes are reachable from node 1, a natural property for a $\mathcal{C}$-Graph, then a negative cycle is present if and only if there is improvement at the n-th iteration of the Bellman-Ford algorithm. As mentioned before, we can *reject* the existence of a negative cycle at an earlier iteration if we have no improvement then. In support of our column generation scheme, we are even more interested in *detecting* negative cycles prior to the n-th iteration.

First, we address the issue of actually finding a negative cycle from the Bellman-Ford computations. We do this by recording, for each improved node, the arc (or node when there is no ambiguity) which led to its most recent improvement. For example, if while going through node i's adjacency list, we find that $u_{k-1}(i) + c_{ij} < u_{k-1}(j)$, then we assign $u_k(j) := u_{k-1}(i) + c_{ij}$, and assign $\text{Pred}(j) := i$. Initially, we assign $\text{Pred}(j)$:=Nil for all nodes j. By induction, at the conclusion of iteration k, $\text{Pred}(j)$, if non-Nil, denotes a node on a minimum cost k-walk from node 1 to node j which is the immediate *pred*ecessor of the last appearance of node j. Note that if all minimum cost k-walks to node j contain a negative cycle, then the sequence $j, \text{Pred}(j), \text{Pred}(\text{Pred}(j)), \ldots$ must eventually intersect itself (not necessarily at node j). For otherwise we would reach Nil, giving us (in reverse) a shortest *acyclic* k-walk to node j, a contradiction.

Conversely, we assert that if the sequence j,$\text{Pred}(j)$,$\text{Pred}(\text{Pred}(j)), \ldots$ cycles (after some iteration k), then this cycle must be negative. For suppose, without loss of generality, that we reach the cycle $\sigma_1 \to \sigma_2 \to \cdots \to \sigma_t \to \sigma_1$, (i.e., $\text{Pred}(\sigma_i) = \sigma_{i-1}$, $i = 2, \ldots, t$ and $\text{Pred}(\sigma_1) = \sigma_t$). If the time of node σ_i's last improvement, was iteration $k(i) \in [1, k]$, then

$$u_k(\sigma_i) = u_{k(i)}(\sigma_i) = u_{k(i)-1}(\sigma_{i-1}) + c_{\sigma_{i-1},\sigma_i} \geq u_k(\sigma_{i-1}) + c_{\sigma_{i-1},\sigma_i} \qquad (2)$$

Summing these inequalities (treating 0 as t), we get

$$\sum_{i=1}^{t} u_k(\sigma_i) \geq \sum_{i=1}^{t} u_k(\sigma_i) + \sum_{i=1}^{t} c_{\sigma_{i-1},\sigma_i}.$$

Hence, $\sum_{i=1}^{t} c_{\sigma_{i-1},\sigma_i} \leq 0$.

Next, suppose node σ_j was the *last* node on the cycle to improve. Then since node σ_{j+1} has not yet "reacted" to node σ_j's most recent improvement, the inequality $u_k(\sigma_{j+1}) > u_k(\sigma_j) + c_{\sigma_j,\sigma_{j+1}}$ is strict. Hence $\sum_{i=1}^{t} c_{\sigma_{i-1},\sigma_i} < 0$, as asserted.

Thus, we need not necessarily wait for the n-th iteration of the Bellman-Ford computation: after *each* iteration we can trace back (via Pred) from all the improved nodes; if we find a cycle, it must be negative, and the algorithm can terminate. This traceback procedure (from *all* the improved nodes) can be done in only $O(n)$ time by exploiting the idea that once a node has been examined, we need never trace back from it again. More

specifically, at the conclusion of an iteration, we "stamp" each node with the symbol 0. In the general step, if improved node i is stamped with a 0, then we re-stamp it with the symbol i, and then examine node Pred(i). If Pred(i) is stamped with a 0, then we stamp it with an i, then examine node Pred(Pred(i)). This process continues until for some k, either $\text{Pred}^k(i) = Nil$ or node $\text{Pred}^k(i)$ is found to bear a stamp $j \neq 0$. In the first case, we have been led back to node 1 which has not been improved, and thus no negative cycle utilizes the nodes stamped with i (for this iteration). In the second case, if $j = i$, then we have discovered a (necessarily negative) "Pred cycle" (not necessarily containing node i). If $j \neq i$, then node $\text{Pred}^k(i)$ has already been examined (when we traced back from node j). Hence we need not examine $\text{Pred}^{k+1}(i)$, as we have reached a "dead end" information-wise. Thus no arc $(\text{Pred}(j), j)$ is utilized more than once, and so our trace-back algorithm is $O(n)$, which does not increase the $O(|E|)$ time complexity of the iteration.

We mention two further ideas for accelerating the algorithm. First, in analogy to other iterative methods (e.g., the Gauss-Seidel method for solving linear equations), we can use the updated u information immediately, rather than waiting an iteration to use it. That is, we can eliminate the k-subscript, and replace line (4) with

$$u(j) := \min\{u(j), u(i) + c_{ij}\}$$

or just as simply,

Line (4a): If $u(j) > u(i) + c_{ij}$ Then
Line (4b): $[u(j) := u(i) + c_{ij}$, and $\text{Pred}(j) := i]$.

After the k-th iteration, $u(j)$ may be interpreted as the length of a minimum cost walk from node 1 to node j among all walks in "the k-th class". This class contains (usually, properly) all k-walks. All of the preceding theory goes through with minor modifications. We note that the inequality analogous to (2), namely

$$u(i) \geq u(\text{Pred}(i)) + c_{\text{Pred}(i),i}$$

remains valid (*sans* k-subscripts) since when $u(i)$ changes, it is set equal to $u(\text{Pred}(i)) + c_{\text{Pred}(i),i}$ and the right hand side can change only by decreasing before $u(i)$ decreases again.

Second, we note that if a node i has not improved since the last time $A(i)$ was examined, then we need not examine it in line (4a) of this iteration. For, after the last iteration we had $u(j) \leq u(i) + c_{ij}$ for all j on $A(i)$. Since the right-hand side has not changed, and $u(j)$ can not increase, line (4b) will not be executed.

The acceleration ideas in the last three paragraphs do not increase the asymptotic complexity of the algorithm. Based on the computational evidence in Appendix B, they frequently identify (or establish the absence of) negative cycles in significantly less time than the n iterations required by the traditional version. A Pascal program (NEGCYC.PAS) implementing these ideas is presented in Appendix A.

We briefly allude to other popular methods for detecting negative cycles. First of all, the Bellman-Ford computations could possibly be speeded up even further, using ideas from Yen (1970). Desrochers (1987), and Glover and Klingman (1987) proposed a modified shortest path routine which keeps track of the numbers of arcs in its shortest walks. If the number of arcs in some shortest walk is n (the number of nodes) or greater, then a negative cycle can be detected early. Our back-tracking procedure seems to be stronger (i.e., it will find negative cycles that these methods might miss, while adding nothing to the worst case complexity) with the same order of $O(n|E|)$ complexity. Another idea is to implement an "all-pairs shortest path routine" (e.g., the Floyd-Warshall method or a matrix multiplication method, both described in Lawler, 1976), and after n iterations, to check (for each node i) whether the length of a shortest path from node i to node i is negative. We did not pursue this method, since its worst case complexity was $O(n^3)$.

A branch and bound approach, exploiting the arithmetic property that every negative cycle contains a node from which all "partial sums" of arc-weights around the cycle, commencing from that node, are negative, was proposed by Netter (1971) and Florian and Robert (1971, 1972). These methods, as pointed out by Yen (1972), have exponential worst case performances. Klein and Tibrewala (1973) suggested an improvement over the technique of Florian and Robert by iteratively transforming the costs of the arcs (in such a way that "shortest routes" are preserved) until a negative cycle was found or was shown not to exist. Chen (1975) proposed a "node elimination" method which performed better than that of Klein and Tibre-

wala for large graphs with low average degree. Despite the fact that all of the methods mentioned in this paragraph had non-polynomial worst case complexity, they did outperform the shortest-path based methods (using, I believe, a shortest-path routine described in Ford and Fulkerson (1962)) for their chosen problems. These experiments, however, probably did not incorporate early termination ideas, such as those discussed in this section, with the shortest-path computations. It would be interesting to see how the above methods compare with our modified Bellman-Ford procedure.

3.3 Implementation Details and Computational Results

In Appendix B we discuss our computer program COLGEN.PAS, which implements the ideas of the first two sections to solve $\mathcal{C}$-Graph problems of manageable size. We exercised the program to solve for all directions **b** of desired final progress, the three piece, *connected* two dimensional jumping problem, restricted to *connected* configurations. Connectivity is defined as follows. The graph of a placement consists of a node for every point in $\mathbf{Z}^m$ which is occupied. Nodes x and y are connected by an arc if $\|x - y\|_1 \leq 2$. The placement is said to be connected, if its associated graph is connected.

In $\mathbf{Z}^2$, there are 46 connected three-piece configurations (see Figure 18), and the associated $\mathcal{C}$-Graph has 294 arcs (obtained by doing a systematic enumeration of all possible moves). In Appendix B, we demonstrate how COLGEN.PAS solves this problem. For example, if we wish to maneuver from a configuration placed at $(0,0)$ to one placed at (a,b), with $a \geq b \geq 0$, then the calculated turnpike phase of our trajectory consists of approximately b repetitions of cycle $5 \rightarrow 24 \rightarrow 25 \rightarrow 5$ (the "diagonal snake" translation in the direction $(1,1)$ with cost 3), and $(a-b)/2$ repetitions of cycle $29 \rightarrow 31 \rightarrow 20 \rightarrow 29$ (the "horizontal snake" in the direction $(2,0)$ with cost 3). This was efficiently solved by our column generation scheme, whose complete details are in Appendix B. We note that the cycles generated by COLGEN.PAS were usually found in just a few iterations of the Bellman-Ford procedure. The negative cycle detection process (using the program NEGCYC.PAS) is separately illustrated in Appendix A, using some small examples.

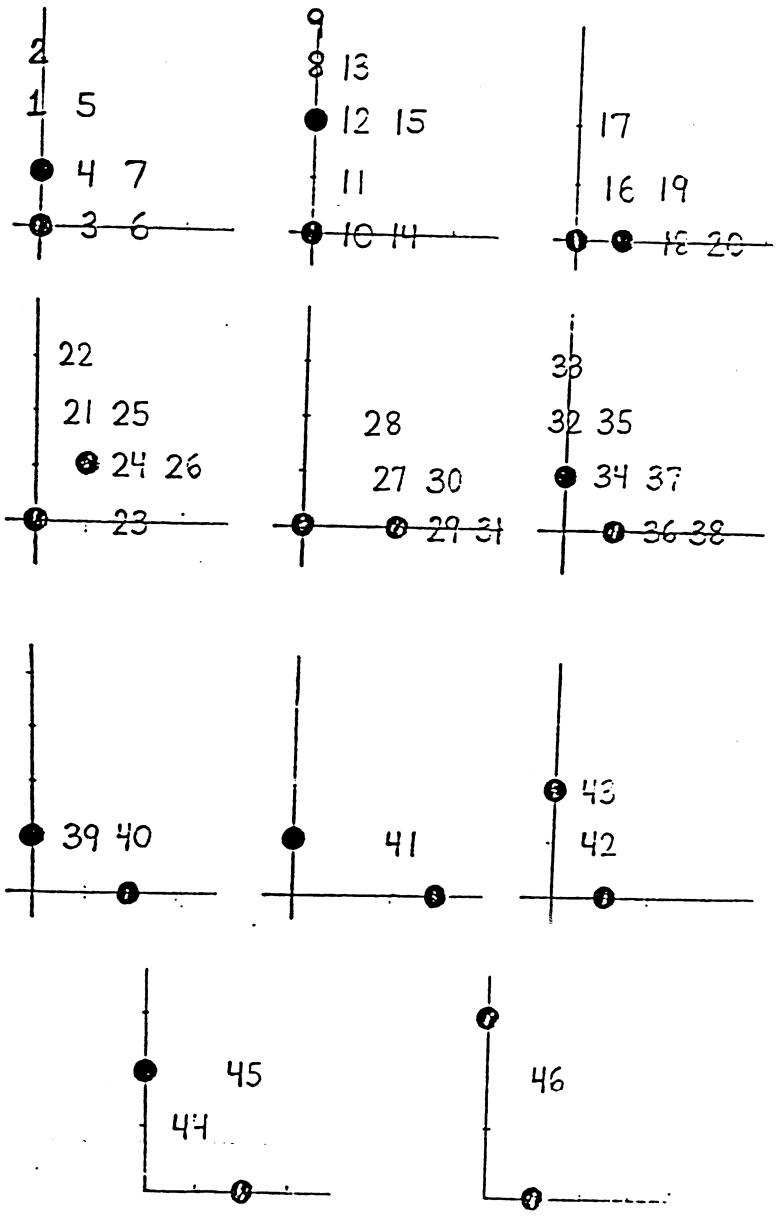


Figure 18: The connected three piece configurations. Each subfigure shows the positions of two pieces, and configuration-numbers corresponding to the different possible position of the third piece.

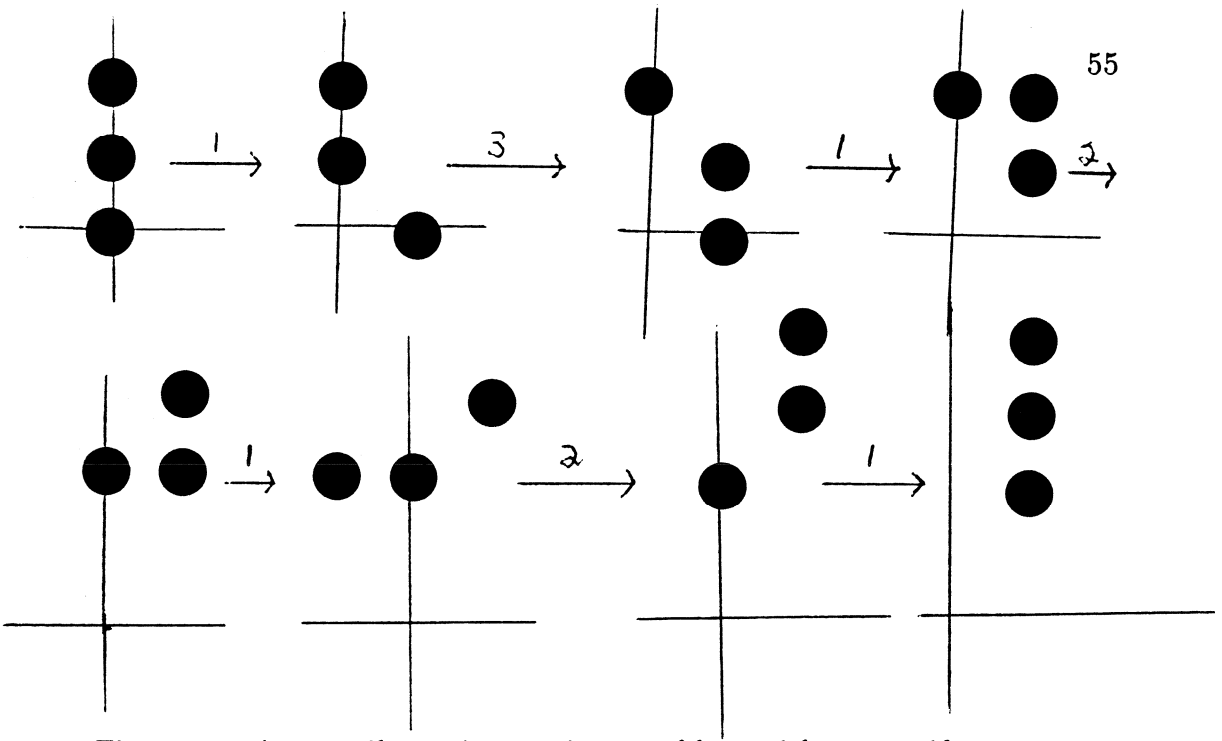


Figure 19: A turnpike trajectory in a problem with non-uniform costs.

In the problem just analyzed, all moves had unit cost. For the next problem, we varied the costs (uniformly, among the integers $1, \ldots, 9$ using a table of random digits) and re-solved the problem for all right-hand sides. [Note that in two dimensions, it suffices to consider those right hand sides of the form $(b_1, b_2)^T$, where the point (b_1, b_2) lies on the square surrounding 0_2, with vertices $(1,1)$, $(-1,1)$, $(-1,-1)$, $(1,-1)$. This made the sensitivity analysis output especially useful. It would be interesting to see if a similar "surrounding" of the origin could be efficiently accomplished in higher dimensions.] We note the presence of "retrograde" movement in some of our turnpike cycles (e.g., in Figure 3.3).

3.4 Alternative Approaches

The linear program (1) can be reformulated as *minimum cost circulation problem*

$$\min \sum_{j=1}^{\|E\|} \gamma^j y_j$$

$$\text{s.t. } P\mathbf{y} = \mathbf{b},\ B\mathbf{y} = \mathbf{0},\ \mathbf{y} \geq \mathbf{0} \qquad (3)$$

where here y_j is the amount of flow on the j-th *arc* from the arc-set E, γ^j and $\mathbf{p}_j$ are that arc's respective cost and (m-dimensional) progress, and B is the node-arc incidence matrix of the $\mathcal{C}$-Graph.

The correspondence between these two linear programs can be made explicit by noting that every circulation can be decomposed as a non-negative linear combination of cycles (with the same overall cost), and likewise every such combination of cycles is a circulation. Note that a basic optimal solution to (3) uses at most $m + \|V\|$ arcs, where V is the node set, and can thus be decomposed into $m + \|V\|$ or fewer cycles in $O((m + \|V\|)^2)$ time by a simple recursive procedure. If this decomposition uses more than m cycles (and if we can not reduce this number by inspection), we can then restrict (1) by using only those columns (representing cycles) obtained in our decomposition, and proceed to find a basic optimal solution directly. It would be interesting to compare this method with our column generation scheme.

The minimum cost circulation problem should be most efficiently treatable by special "network with side constraints" algorithms (e.g. Chen and Saigal 1977). In fact, since cycles are "extreme circulations" to the "network without side constraints" problem, our column generation scheme can be viewed as an application of Dantzig-Wolfe decomposition to the minimum cost circulation problem. (See, e.g., Chvátal, 1983.) The author acknowledges Michael Schneider for pointing this out to him. When $m = 1$, program (3) can be solved using a similar column generation scheme described by Dantzig, Blattner and Rao (1967). Other efficient solutions to the one dimensional problem of finding the minimum cost/progress ratio cycle are given by algorithms of Karp (1978) (when all arcs have the same cost) with time complexity $O(|V||E|)$, and Megiddo (1979) with time complexity $O(|V||E|^2)$. (Note: due to the absence of negative cycles in our $\mathcal{C}$-Graph's, the error in Megiddo's paper (pointed out by Ahuja, Batra and Gupta (1983)) does not arise here.

Lastly, we address the situation when our $\mathcal{C}$-Graph's arc set, or even its vertex set, is prohibitively large (i.e., we can not even store the graph in a computer). (Recall that the 1-dimensional jumping problem with p pieces has 2^{p-1} connected configurations.) Does this mean that there is no hope of finding a reasonable solution to our problem? Not at all. Operations researchers face this sort of difficulty, for example, when formulating

integer programs for crew scheduling of airlines. To consider all possible assignments of subsets of crew members to all possible flights would be overwhelming. However, one can restrict the domain of optimization to a manageable number of reasonable looking assignments. (In some versions, the "current optimum" can be tested for true optimality, in such a way that a negative outcome also generates a new member of the "manageable set"; cf. the use of "column generation", above.) In a similar way, our rules for movement may suggest certain promising translations of configurations, and we might then restrict our attention to those particular translations (i.e., cycles in the $\mathcal{C}$-Graph), and thus to the vertices and arcs that they contain. This idea can be extended to the case where we are unable to prove that the finiteness assumption is satisfied by our rules for movement. Here again, we can guess at a finite number of natural-looking efficient configurations, and determine turnpike-like combinations of translations of these, as before.

3.5 Complexity

In this section we show that the decision problem C-GRAPH naturally associated with the $\mathcal{C}$-Graph problem for one-dimensional maneuvers, is NP-complete. Given the apparent computational intractability of NP-Complete problems, this suggests that efficient algorithms for finding "almost optimal" trajectories, such as our turnpike trajectories, may be about as good as we can expect—the computational price required for "optimal" trajectories may be too high.

INPUT: A list of arcs each with an associated integral progress scalar and a positive integral cost; an origin node $\mathcal{O}$; a destination node $\mathcal{D}$; a "progress requirement" $d > 0$ and a positive integral cost requirement c. If x denotes an instance of this input, then its size $|x|$ satisfies $|x| \geq n + |E| + 2 + \log d + \log c$ (where the logarithm is to the base 2, and $n = |V|$).

DECISION PROBLEM: Does there exist a walk from $\mathcal{O}$ to $\mathcal{D}$, with total progress d, and total cost less than or equal to c?

To show that this decision problem (which we call C-GRAPH) belongs to the class NP, we must determine a polynomial p such that (a) when the input x to C-GRAPH has a "yes" answer, a "certificate" $c(x)$ can be written whose size is bounded by $p(|x|)$, and (b) the certificate is checkable,

to verify the "yes" answer, in polynomial time (i.e., within $p(|c(x)|)$.

Note that the "obvious" certificate consisting of the actual sequence of arcs traversed in the walk would satisfy condition b) but not a), since increasing d by a factor of n only adds $\log n$ to the input size, but would increase the certificate size by a factor of n. Hence we provide the following

Lemma 3.1 *Let W be a walk which satisfies the conditions of C-GRAPH under input x. Then the list L of distinct arcs of W, along with the number of times each arc is utilized in W, provides a certificate $c(x)$.*

Proof. Indeed, the directed (multi-)graph induced by this list L would be connected, with the further "balanced node" property that the number of arcs entering each node is equal to the number of arcs leaving that node (except if $\mathcal{O} \neq \mathcal{D}$ when we make the obvious adjustments at these nodes). Conversely, by the usual Eulerian trail or circuit arguments (see, for instance, Gibbons, p.155, (1985)), the existence of an arc sublist with these properties, and with total progress d and total cost $\leq c$, ensures the existence of a walk that satisfies C-GRAPH. □

Note that the size $|c(x)| \approx |L|(1 + \sum_{j \in L} \lceil \log f_j \rceil)$ where f_j is the multiplicity of arc j. Since all arcs are assumed to have positive integral cost and $c(x)$ represents a "yes" certificate, we must have $f_j \leq c$ for all j. Hence, $|c(x)| \leq |E|(1 + |E|\lceil \log c \rceil) \leq (|x|)^3$. Thus, condition a) is satisfied. Furthermore, the connectedness of the induced graph can be checked in time polynomial in $|c(x)|$ say by seeking shortest paths from node $\mathcal{O}$ to all other nodes and seeing if all vertices are reached, as can the "balanced node property". Thus properties a) and b) are both satisfied, so C-GRAPH$\in$ NP.

Now to prove that C-GRAPH is NP-Complete, we must further demonstrate that some known NP-Complete problem can be transformed with only polynomial effort into an equivalent C-GRAPH problem, whose size is polynomially bounded in the size of the NP-Complete problem.

The NP-Complete problem we shall use is PARTITION (Garey and Johnson, 1978, p.47) whose input is a finite set of integers $A = \{a_1, \ldots, a_n\}$. The problem is to decide whether there exists some subset $A' \subseteq A$ such that $\sum_{a \in A'}(a) = \sum_{a \in A - A'}(a)$. The size of such an input is $n + \sum_{i=1}^{n} \lceil \log(1 + a_i) \rceil$. Without loss of generality, we shall assume that $\sum_{a \in A}(a)$ is even.

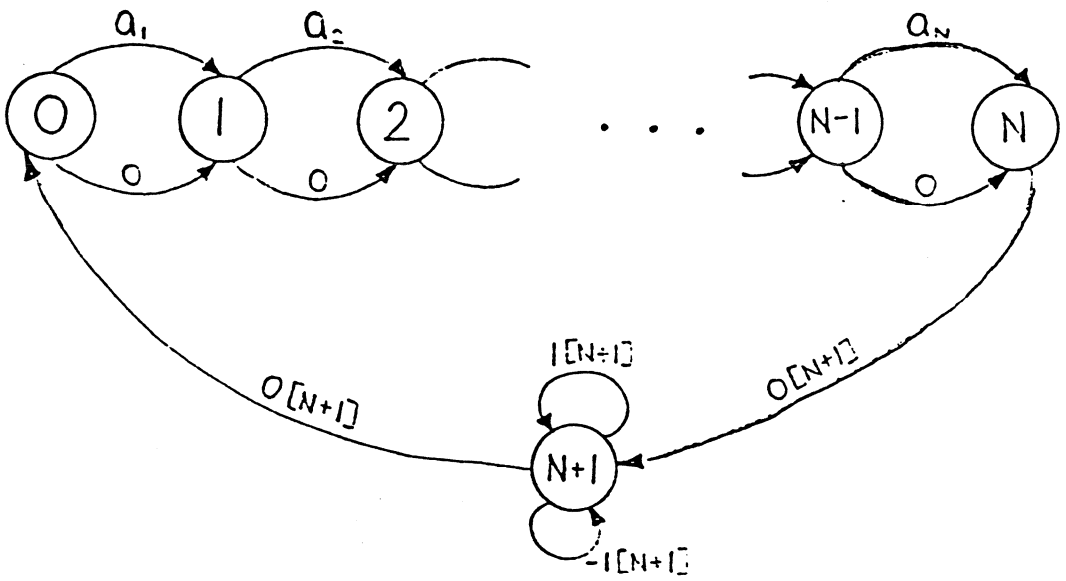


Figure 20: A C-Graph for the PARTITION problem

An equivalent C-GRAPH problem is constructed as follows. (See Figure 20.) The graph has node-set $V = \{0, 1, \ldots, n, n+1\}$. For $i = 0, 1, \ldots, n-1$, we have from node i to node $i+1$ both an arc with progress a_{i+1} and cost 1, and an arc with progress 0 and cost 1. We have an arc with progress 0 and cost $n+1$ from node n to node $n+1$ and the same kind of arc from node $n+1$ to node 0. Finally we have two loops at node $n+1$ with progress 1 and -1 respectively and cost $n+1$. The corresponding C-GRAPH problem is: "Does there exist a walk from node 0 to node n with total progress $\sum_{i=1}^{n} a_i/2$ and total cost at most n?"

The one-to-one correspondence between these two problems is immediate. (The node $n+1$ was placed in the network to accommodate the brute force assumption of C-Graphs.) Hence, a polynomial algorithm for solving C-GRAPH would result in a polynomial algorithm for PARTITION (and

consequently, all problems in NP). Thus C-GRAPH is NP-Complete.

Incidentally, we can associate a (somewhat contrived) maneuvering problem (called "RAINBOW") with the $\mathcal{C}$-Graph in Figure 20. The game consists of a single piece which "changes color" as it moves. The piece starts off red, and in its next move (where it can make progress 0 or a_1) it changes color to orange. The following move, it changes to yellow (making progress of either 0 or a_2), and so on, until it turns violet (the n-th color). From there, the piece turns black (making progress 0), and on its subsequent moves can either remain black (with progress 1 or -1) or change back to red (with progress 0). All non-black moves have a cost of 1, while all black moves are very costly. The aim is a minimum-cost transition from red to violet, with total progress $\sum_{i=1}^{n} a_i/2$.

Chapter 4
Maneuvering Within Boundaries

4.1 Maneuvering Within Boundaries

As before, our objective is to maneuver optimally within $\mathbf{Z}^m$ from $(\mathcal{O}, \mathbf{0}_m)$ to $(\mathcal{D}, d\mathbf{b})$, where $d > 0$, but now we are restricted to have all placements $(X, \mathbf{a})$ satisfy $\mathbf{a} \in dL$ for some prescribed "legal region" $L \in \mathbf{R}^m$. This new condition violates the Space Homogeneity assumption, and the previous theories therefore need not apply. We shall show, however, that if the problem obeys certain "interiority" conditions, then we can modify our previous turnpike trajectory so as to maneuver within the "boundaries of dL", without loss of asymptotic optimality. It will also be shown that this scenario can be adapted to handle the more concrete requirement that all *pieces* remain in dL during the maneuver.

We begin with the "unit problem" of maneuvering from $(\mathcal{O}, \mathbf{0})$ to $(\mathcal{D}, \mathbf{b})$ subject both to the usual type of movement rules, and to the condition that all placements $(X, \mathbf{a})$ of our trajectory satisfy $\mathbf{a} \in L \subset \mathbf{R}^m$. Without loss of generality, let $\mathbf{x}^* = (x_1^*, x_2^*, \ldots, x_m^*) \in \mathbf{R}^m$ be the basic part of an optimal basic solution to the associated linear program (LP): $\min \mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} = \mathbf{b}$, $\mathbf{x} \geq 0$. Geometrically, $\mathbf{x}^*$ can initially be thought of as a piecewise linear path in $\mathbf{R}^m$, from $\mathbf{0}_m$ to $\mathbf{b}$ that goes from $\mathbf{0}$ to $x_1^*\mathbf{a}_1$, to $x_1^*\mathbf{a}_1 + x_2^*\mathbf{a}_2$, to $\ldots$, to $\sum_{i=1}^m x_i^*\mathbf{a}_i = \mathbf{b}$, with total cost $\sum_{i=1}^m c_i x_i^*$ (one summand per linear segment).

Notice that we can in many ways "chop up and rearrange" the vectors of this path to create another one with the same origin, destination and cost (measured in the natural way). This yields an alternative solution (geometrically speaking) to the LP, albeit with perhaps more *segments*. For example, see Figure 21.

More formally, a *path based on* $\mathbf{x}^*$ *and* A is one which begins at $\mathbf{0}$ and is *represented* by a finite sequence $\{(y_i, \alpha_i)\}_{i=1}^k$ such that $y_i > 0$, $\alpha_i \in \{\mathbf{a}_1, \ldots, \mathbf{a}_m\}$, $i = 1, \ldots, k$, and $\sum_{i:\alpha_i = \mathbf{a}_j} y_i = x_j^*$. (Note that $\{i : \alpha_i = \mathbf{a}_j\}$ could be empty for some j, when $\mathbf{x}^*$ is degenerate.) Hence, at the i-th step, our path goes from its current position $\mathbf{z} \in \mathbf{R}^m$ to $\mathbf{z} + y_i\alpha_i$. For example,

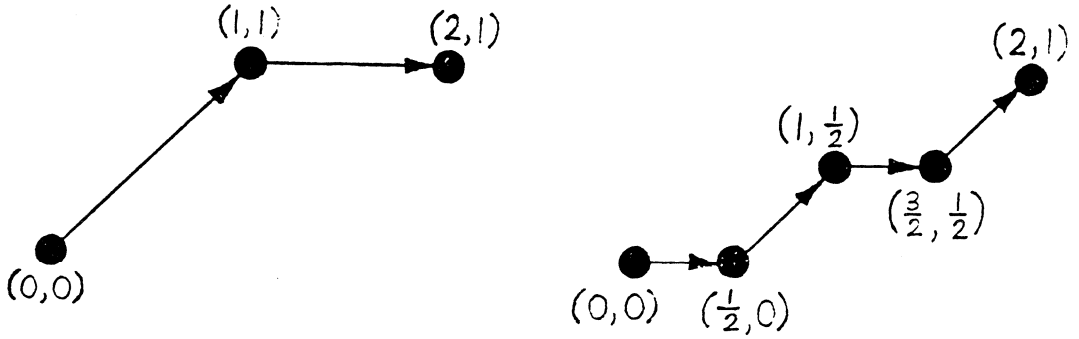


Figure 21: Two paths based on the same $\mathbf{x}^*$ and A.

two paths based on $\mathbf{x}^* = (1,1)^T$ and $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ (see Figure 21) are represented by the sequences $(1,(1,1))$, $(1,(1,0))$ and $(1/2,(1,0))$, $(1/2,(1,1))$, $(1/2,(1,0))$, $(1/2,(1,1))$, respectively.

Given $\mathbf{x}^*$, A, and L, let us suppose that we can construct a path P based on $\mathbf{x}^*$ and A, satisfying the following *interior condition*

$$N(P,M) \subseteq L,$$

where $M = \max_{i=1,\ldots,m} \|\mathbf{a}_i\|$ (any norm will do), and $N(P,M)$ is the M-neighborhood of P, i.e., the set of points in $\mathbf{R}^m$ whose distance from P (under the same norm, of course) is less than M. We shall later derive sufficient conditions for the existence of a path P satisfying this interior condition. Let P be represented by $\{(y_i,\alpha_i)\}_{i=1}^k$. Thus P has precisely k linear parts. (Generally, $k \geq m$.)

Next consider the path dP, with particular representation $dP = \{(dy_i,\alpha_i)\}_{i=1}^k$. This is easily seen to be a path, based on $d\mathbf{x}^*$ and A, from $\mathbf{0}$ to $d\mathbf{b}$. For a point $\mathbf{z} \in \mathbf{R}^m$, and a compact set $Q \subset \mathbf{R}^m$, define $\|\mathbf{z}-Q\| = \min_{y\in Q} \|\mathbf{z}-\mathbf{y}\|$. Note that $\|d\mathbf{z} - dQ\| = d\|z - Q\|$.

Claim 4.1 *If $N(P,M) \subseteq L$, then $N(dP,dM) \subseteq dL$, for $d > 0$.*

Proof. Let $\mathbf{z} \in N(dP, dM)$. That is, $\|\mathbf{z} - dP\| < dM$. Thus, $\|\frac{1}{d}\mathbf{z} - P\| = \frac{1}{d}\|\mathbf{z} - dP\| < M$. Whence, $\frac{1}{d}\mathbf{z} \in N(P, M) \subseteq L$. Hence, $\mathbf{z} \in dL$. □

Next, consider the path $\lfloor dP \rfloor = \{(\lfloor dy_i \rfloor, \alpha_i)\}_{i=1}^{k}$. This is a path from $\mathbf{0}$ to $\sum_{i=1}^{k} \lfloor dy_i \rfloor \alpha_i \equiv \bar{\mathbf{b}}$.

Claim 4.2 *$\lfloor dP \rfloor \subseteq N(dP, kM)$. In addition, $\|d\mathbf{b} - \bar{\mathbf{b}}\| < kM$.*

Proof. Let $\mathbf{z} \in \lfloor dP \rfloor$. We need to find a point $\hat{\mathbf{z}} \in dP$ such that $\|\hat{\mathbf{z}} - \mathbf{z}\| < kM$. Since $\mathbf{z} \in \lfloor dP \rfloor$, $\mathbf{z} = \sum_{i=1}^{j-1} \lfloor dy_i \rfloor \alpha_i + \eta\alpha_j$ for some $j \in \{1, \ldots, k\}$, $0 \leq \eta \leq \lfloor dy_j \rfloor$. Consider $\hat{\mathbf{z}} = \sum_{i=1}^{j-1} dy_i\alpha_i + \eta\alpha_j$. $\hat{\mathbf{z}} \in dP$, since $\eta \leq \lfloor dy_j \rfloor \leq dy_j$. Therefore,

$$\begin{aligned}
\|\bar{\mathbf{z}} - \mathbf{z}\| &= \|\sum_{i=1}^{j-1} dy_i\alpha_i - \sum_{i=1}^{j-1} \lfloor dy_i \rfloor \alpha_i\| \\
&= \|\sum_{i=1}^{j-1} (dy_i - \lfloor dy_i \rfloor)\alpha_i\| \\
&\leq \sum_{i=1}^{j-1} \|(dy_i - \lfloor dy_i \rfloor)\alpha_i\| \\
&< \sum_{i=1}^{j-1} \|\alpha_i\| \leq (j-1)M \leq (k-1)M < kM.
\end{aligned}$$

Furthermore,

$$\begin{aligned}
\|d\mathbf{b} - \bar{\mathbf{b}}\| &= \|\sum_{i=1}^{k} dy_i\alpha_i - \sum_{i=1}^{k} \lfloor dy_i \rfloor \alpha_i\| \\
&= \|\sum_{i=1}^{k} (dy_i - \lfloor dy_i \rfloor)\alpha_i\| \\
&\leq \sum_{i=1}^{k} (dy_i - \lfloor dy_i \rfloor)\|\alpha_i\| \\
&< \sum_{i=1}^{k} \|\alpha_i\| \leq kM.
\end{aligned}$$

Combining these two claims gives us the following

Lemma 4.1 *When $d \geq k$, $\lfloor dP \rfloor \subseteq dL$.*

Proof.

$$\lfloor dP \rfloor \subseteq N(dP, kM) \subseteq N(dP, dM) \subseteq dL.$$

□

Theorem 4.1 *When the interior condition is satisfied, then for all d sufficiently large (where $d\mathbf{b} \in \mathbf{Z}^m$), there exists a near-optimal "turnpike" solution to the maneuvering problem from $(\mathcal{O}, \mathbf{0}_m)$ to $(\mathcal{D}, d\mathbf{b})$, restricted by the requirement that all placements $(X, \mathbf{a})$ employed must satisfy $\mathbf{a} \in dL$. The trajectory can be considered "turnpike", because it spends almost all of its cost engaged in at most m simple movement patterns.*

Proof. Our turnpike trajectory is constructed by "following the instructions" of $\lfloor dP \rfloor$. That is, we brute force our way from $(\mathcal{O}, \mathbf{0}_m)$ to a turnpike configuration (placed at $\mathbf{0}_m$) associated with α_1, then translate *that* configuration $\lfloor dy_1 \rfloor$ times, then we brute force to a configuration similarly associated with α_2 (placed at $\lfloor dy_1 \rfloor \alpha_1$) and translate that configuration $\lfloor dy_2 \rfloor$ times, and so on. The trajectory concludes with a brute force from the configuration associated with α_k (placed at $\bar{\mathbf{b}}$) to our destination $(\mathcal{D}, d\mathbf{b})$. For $d \geq k$, our lemma assures that all intermediate placements $(X, \mathbf{a})$ satisfy $\mathbf{a} \in \lfloor dP \rfloor \subseteq dL$. (We are assuming that all brute forces can be done without leaving the legal set dL–see the remarks following the theorem.)

The cost of the above trajectory is

$$\begin{aligned} kc_{\mathbf{0}} + \sum_{j=1}^{m} c_j \sum_{\substack{i=1 \\ \alpha_i = \mathbf{a}_j}}^{k} \lfloor dy_i \rfloor + c_\delta &\leq kc_{\mathbf{0}} + \sum_{j=1}^{m} c_j \sum_{\substack{i=1 \\ \alpha_i = \mathbf{a}_j}}^{k} dy_i + \bar{c} \\ &= kc_{\mathbf{0}} + d\sum_{j=1}^{m} c_j x_j^* + \bar{c} \\ &= kc_{\mathbf{0}} + \bar{c} + u^*, \end{aligned}$$

where $\delta = d\mathbf{b} - \bar{\mathbf{b}}$, (by Claim 4.2; note $\|\delta\| \leq kM$), $\bar{c} = \max\{c_\delta : \|\delta\| \leq kM\} < \infty$, and $u^* = d\mathbf{c}^T\mathbf{x}^*$. In chapter two, it was established that the cost of any optimal trajectory from $(\mathcal{O}, \mathbf{0}_m)$ to $(\mathcal{D}, d\mathbf{b})$, ignoring the restriction $\mathbf{a} \in dL$, must be at least $u^* - c_{\mathbf{0}}$, so that the same is true when the restriction is imposed. Hence the cost of our turnpike trajectory exceeds the cost of

an optimal trajectory by at most $(k+1)c_0 + \bar{c}$, which does not depend on d. □

Remarks:

- Note that under this scenario, we must qualify our movement assumptions with the words "provided $\mathbf{a} \in dL$", whenever we use placement $(X, \mathbf{a})$. We must further assume that all required brute forces can be achieved within dL.

- It may be more natural to analyze the scenario where all of our *pieces* must remain inside our legal region dL. This situation can be handled by the above model, by considering $w = \max_{X \in \mathcal{C}} \text{Radius}(X) < \infty$, where $\text{Radius}(X)$ is defined to be $\max_{\mathbf{x} \in (X, \mathbf{0}_m) \subset \mathbf{Z}^m} \|\mathbf{x}\|$. Hence every piece in placement $(X, \mathbf{a})$ is at most a distance w from $\mathbf{a}$. It follows from Claim 4.2 that any piece corresponding to a placement $(X, \mathbf{a})$, $\mathbf{a} \in \lfloor dP \rfloor$, is at most a distance $kM + w$ from dP. Since $dM \geq kM + w$ for d sufficiently large ($d \geq k + w/m$), Claim 4.1 implies that all our pieces remain inside dL.

- Lastly, we need to develop sufficient conditions for the interior condition to be satisfied. This will be the subject of the next section.

4.2 Getting Down to the Interior of It

Let ℓ denote the straight segment (path) from $\mathbf{0}_m$ to $\mathbf{b}$.

Lemma 4.2 *If for some $r > 0$, $N(\ell, M + r) \subseteq L$, then the interior condition is satisfied for some path P based on $\mathbf{x}^*$ and A.*

Proof. Let ν be a constant depending on our norm, and satisfying $|y_i| \leq \nu \|\mathbf{y}\|$ for all $\mathbf{y} = (y_1, \ldots, y_m)$. (For the "usual" norms, $\nu = 1$ will do.) Let Q denote the path represented by $(x_1^*, \mathbf{a}_1), (x_2^*, \mathbf{a}_2), \ldots, (x_m^*, \mathbf{a}_m)$. We claim that multiplying Q by a suitably small positive constant $\epsilon > 0$ yields a path $\epsilon Q = \{(\epsilon x_i^*, \mathbf{a}_i)\}_{i=1}^m \subseteq N(\ell, r)$. Indeed choosing $\epsilon = r/(\nu m M \|\mathbf{x}^*\|)$, makes ϵQ a path from $\mathbf{0}$ to $\epsilon\mathbf{b}$ whose distance from $\mathbf{0}_m$ never exceeds $\epsilon \sum_{i=1}^m x_i^* \|a_i\| \leq \epsilon m M \nu \|\mathbf{x}^*\| \leq r$. If we "repeat these instructions", starting at $\epsilon\mathbf{b}$, we have a new path (call it $(\epsilon Q)^2 = (\epsilon Q) \cup ((\epsilon Q) + \epsilon\mathbf{b}))$ from $\mathbf{0}_m$ to $2\epsilon\mathbf{b}$, where the distance between any of our new points and $\epsilon\mathbf{b} \in \ell$ never

exceeds r. Thus, $(\epsilon Q)^2 \subseteq N(\ell, r)$. Continuing in this way, we build up the path $(\epsilon Q)^{\lfloor 1/\epsilon \rfloor}$ from $\mathbf{0}_m$ to $\epsilon\lfloor 1/\epsilon \rfloor \mathbf{b}$, which is a subset of $N(\ell, r)$. Finally, define P to consist of $(\epsilon Q)^{\lfloor 1/\epsilon \rfloor}$ followed by the appropriate translate of $(\bar{\epsilon} Q)$, where $\bar{\epsilon} = 1 - \epsilon\lfloor 1/\epsilon \rfloor$. Since $0 \leq \bar{\epsilon} < \epsilon$, no point of this last subpath of P has distance more than r from its initial point $\epsilon\lfloor 1/\epsilon \rfloor \mathbf{b} \in \ell$.

Thus, P is a path from $\mathbf{0}_m$ to $\mathbf{b}$ which is based on $\mathbf{x}^*$ and A, and is a subset of $N(\ell, r)$. If $\mathbf{x} \in N(P, M)$, there exists some point $\mathbf{p} \in P$ such that $\|\mathbf{x} - \mathbf{p}\| < M$. Since $\mathbf{p} \in P \subset N(\ell, r)$, $\|\mathbf{p} - \ell\| \leq r$. Hence $\|\mathbf{x} - \ell\| \leq \|\mathbf{x} - \mathbf{p}\| + \|\mathbf{p} - \ell\| < r + M$. Hence, $\mathbf{x} \in N(\ell, M + r)$, a subset of L, by assumption. Thus, $N(P, M) \subseteq L$, as asserted. □

Remarks:

- Note that when ϵ is chosen as in the previous proof, the number of linear segments in our path P is at least $k = m\lceil 1/\epsilon \rceil \approx \nu m^2 M \|\mathbf{x}^*\|/r$, so that P could be quite "kinky". Then the condition $d \geq k$, in Theorem 4.1's proof's specification of "sufficiently large d", might be found restrictive. Although the present arguments are adequate for our "asymptotic" purposes, sharper bounds would be of interest.

- The hypothesis of Lemma 4.2 implies $N(\ell, M) \subseteq L$, requiring ℓ to lie "comfortably interior" to L. But "comfortably" can be dropped from this hypothesis. Indeed, suppose we only had $N(\ell, \delta) \subseteq L$ for some $0 < \delta \leq M$. Then after multiplying by a suitably large constant u (say $u = \lceil 2M/\delta \rceil$), we have by Claim 4.1 that $N(u\ell, M) \subseteq N(u\ell, 2M) \subseteq uL$, where $u\ell$ is the straight line path from $\mathbf{0}$ to $u\mathbf{b}$. And now, if we treat $u\mathbf{b}$ and uL as defining our unit problem, then by Lemma 4.2 our interior condition is satisfied (with $k \approx \nu m^2 u \|\mathbf{x}^*\| \approx 2\nu m^2 M \|\mathbf{x}^*\|/\delta$). Consequently, Theorem 4.1 remains true under the weaker hypothesis, for all d sufficiently large ($d \geq k$) such that $du\mathbf{b} \in \mathbf{Z}^m$. Notice that since d can take on rational values, we lose no generality by letting $u\mathbf{b}$ be our "unit destination".

Chapter 5
Turnpiking Over More General Environments

5.1 Introduction and Notation

In this chapter, we continue the investigation of optimal maneuvering problems over a structured environment E. Whereas Chapter 4 treated the presence of maneuver-limiting "boundaries" violating the previous assumption of Space Homogeneity, here spatial homogeneity will be maintained but the "regular structure" of E will be of more complicated types. Most of the basic theory developed in Chapter 2, where $E = \mathbf{Z}^m$, can be applied or extended to these more general situations.

In section 5.2, we analyze the scenario $E = \mathbf{Z}^m \times T$, where T is a finite set. This is the case, in particular, when E is a finitely generated abelian group; see (Jacobson, 1974; Theorem 3.13). In section 5.3, we generalize even further, to the case where E is a union of cosets of some subgroup G in $\overline{E}$, where $G \subseteq E \subseteq \overline{E}$ and $\overline{E}$ is a finitely generated abelian group. Section 4 extends the results of the preceding two sections when the movement rules obey an additional homogeneity assumption.

First we clarify the notions of placement and configurations in such more general settings. We define P to be a *placement* (of p pieces) over E, if $P \subseteq E$, $|P| = p < \infty$. Actually, P could even be a multi-subset of E, since multiple-occupancy of points of E by pieces may be permissible. If all the pieces are distinguishable, then P should be thought of as a p-tuple of elements of E. More generally still, if our pieces can be partitioned permanently (i.e., transmutations are not permitted) into k distinguishable types, with λ_i pieces of type i, $i = 1 \ldots k$, $\sum_{i=1}^{k} \lambda_i = p$, then p is a k-tuple of multi-sets (with sizes $\lambda_1, \ldots, \lambda_k$, respectively) of elements of E. To avoid burdensome notation, we shall simply adopt the elementary notation $P \subseteq E$ to denote an arbitrary placement of (or over) E, and the notation $P + \mathbf{a}$, $\mathbf{a} \in E$ has the obvious interpretation when the addition of components is well-defined. [Example: if $E = \mathbf{Z}^2$, $p = 3$, $k = 2$, with $(\lambda_1, \lambda_2) = (2,1)$, and $P = (\{(0,0),(1,0)\},\{(1,1)\})$ (see Figure 22), then for $\mathbf{a} \in E$, $P + \mathbf{a} = (\{(0,0) + \mathbf{a}, (1,0) + \mathbf{a}\}, \{(1,1) + \mathbf{a}\})$.]

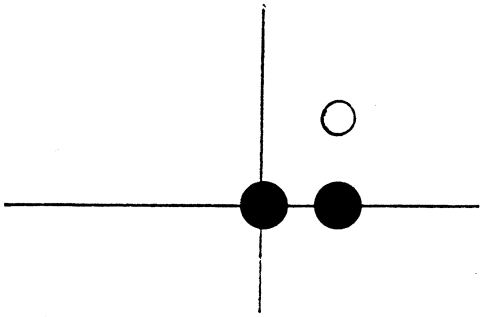


Figure 22: A Placement Example

Our objective is still to maneuver from one placement over E to another at minimum cost, subject to various movement rules. As before, we assume that there exists an equivalence relation among the placements (e.g., translation) with certain desirable properties. We define a *configuration* to be an equivalence class of placements under the above relation, and denote one placement in each equivalence class as its "canonical placement". Sometimes we will use the term configuration to refer to its canonical placement, when the context is clear. The "desirable properties" mimic those used in Chapter 2, and are made precise in each section below. The only one we mention immediately is Finiteness, expressed as before, i.e., "Without loss of (asymptotic) optimality, we can prescribe a finite set C of allowable configurations of the pieces. From each configuration, there are a finite number of legal moves available." In each section we shall let C denote the set of configurations, even though we will be using different equivalence relations.

Much of this chapter relies on the finiteness of T, our ability to brute force, and our limitation to *asymptotic analysis*. Together, these properties imply that Chapter 2's general approach and results continue to work, though it is necessary to provide the precise definitions and technical details.

5.2 $E = \mathbf{Z}^m \times T$.

Suppose that our maneuvering environment E is isomorphic to $\mathbf{Z}^m \times T$, where T is an arbitrary finite set. (A motivating example will be given

later.) For placements $P, Q \subseteq E$, we say that Q is a *free-translate* of P if and only if there exists $\mathbf{a} \in \mathbf{Z}^m$ such that $Q = P + (\mathbf{a}, 0)$. (This notation (slightly abusive) should be interpreted to mean that P and Q have the same T-component, and their non-T-components are translates of each other.) This is obviously an equivalence relation over E. Thus, after noting the coordinates of a canonical placement X, we can describe its equivalence class as $\{X + (\mathbf{a}, 0) : \mathbf{a} \in \mathbf{Z}^m\}$ and let $(X, \mathbf{a})$ represent the placement $X + (\mathbf{a}, 0)$, for $\mathbf{a} \in \mathbf{Z}^m$.

Notice that every placement P has exactly one representation as $(X, \mathbf{a})$, where X is the canonical placement in the configuration containing P. This follows from the fact that if $(\mathbf{a}, 0)$ and $(\mathbf{b}, 0)$ are distinct elements of $\mathbf{Z}^m \times \{0\}$, then for some $i \in \{1, \ldots, m\}$, $a_i \neq b_i$. Suppose, without loss of generality, that $a_j < b_j$; then the minimum j-th coordinate of $X + (\mathbf{a}, 0)$ will be less than the minimum j-th coordinate of $X + (\mathbf{b}, 0)$. Whence $(X, \mathbf{a}) \neq (X, \mathbf{b})$.

As before, we can use the notation $(X, \mathbf{a}) \xrightarrow{c} (Y, \mathbf{b})$ to denote a legal move from placement $(X, \mathbf{a})$ to placement $(Y, \mathbf{b})$ with cost c, where $X, Y \in \mathcal{C}, \mathbf{a}, \mathbf{b} \in \mathbf{Z}^m, c \in \mathbf{Z}^1$. Such a move is defined to have *progress* of $\mathbf{b} - \mathbf{a} \in \mathbf{Z}^m$.

Using the notion of free-translation, and the corresponding interpretation of $(X, \mathbf{a})$, we can write down our movement assumptions (i.e., finiteness, homogeneity, brute force and cycle positivity) *exactly* as in Chapter 2. A $\mathcal{C}$-Graph can be constructed whose nodes represent configurations, and whose arcs represent the cost and progress of legal maneuvers, and there is the same correspondence between trajectories from $(\mathcal{O}, \mathbf{0}_m)$ to $(\mathcal{D}, d\mathbf{b})$ and walks from the node $\mathcal{O}$ to node $\mathcal{D}$ with progress $d\mathbf{b} \in \mathbf{Z}^m$. Likewise, cycles represent simple free-translations. Thus, if our objective is to maneuver from placement $(\mathcal{O}, \mathbf{0}_m)$ (wlog, by space homogeneity) to placement $(\mathcal{D}, d\mathbf{b})$, $\mathcal{O}, \mathcal{D} \in \mathcal{C}$, $\mathbf{b} \geq \mathbf{0}_m$, and $d > 0$, then it is equivalent to finding a corresponding minimum cost walk from $\mathcal{O}$ to $\mathcal{D}$, with total progress $d\mathbf{b}$ along the $\mathcal{C}$-Graph.

Hence Theorems 2.1 (when m=1), 2.2, and 2.3 (when $\mathbf{b} > \mathbf{0}_m$), as well as their proofs, proceed exactly as before in this more general setting. The result is that if E is equal (or isomorphic) to $\mathbf{Z}^m \times T$, and our movement assumptions are satisfied, then there is a near optimal trajectory from $(\mathcal{O}, \mathbf{0}_m)$ to $(\mathcal{D}, d\mathbf{b})$ which (for d sufficiently large) spends almost all of its cost performing simple, free-translations of at most m placements of the

pieces.

Remarks:

- The assumption that T is a finite set, while not used explicitly, can be assumed without loss of optimality once the finiteness condition is imposed. For, suppose $E = \mathbf{Z}^m \times T$, and that our movement rules obey the stated conditions. Then, since our $\mathcal{C}$-Graph (with node-set $\mathcal{C}$) is finite, and placements with different T component sets belong to different configurations, we can restrict ourselves to those placements whose T-components are represented on the $\mathcal{C}$-Graph. Since each configuration, consisting of p pieces, uses at most p elements of T, we can replace T by a subset with size at most $p\|\mathcal{C}\|$, without loss of asymptotic optimality.

- By a well known algebraic structure theorem (see, for instance, Fuchs (1960) or Jacobson (1974)), all finitely generated abelian groups are isomorphic to $\mathbf{Z}^m \times T$ for some $m \geq 0$ and some finite abelian group T (as in Torsion). When the above group is free (i.e., isomorphic to $\mathbf{Z}^m$), we have precisely the same theorems as in Chapter 2, since $T = \{0\}$ and all translations are free-translations.

- As an application of these results, we can consider a "connected Chinese checkers" problem of maneuvering over the vertices of the equilateral triangularization of the plane $\mathbf{R}^2$, where we arbitrarily label one of the vertices as $(0,0)$. (See Figure 23.) Our p pieces maneuver as in Chinese checkers, except that (to accommodate the finiteness assumption) we are restricted to a finite set of connected configurations. Our objective is to maneuver from one placement ("placed at $\mathbf{0}_2$") to another, placed far away. The problem can be analyzed using our earlier techniques by recognizing that our movement environment is a finitely generated abelian group, isomorphic to $\mathbf{Z}^2$ via the isomorphism $f : \mathbf{Z}^2 \to E$ defined by $f(1,0) = (1,0)$ and $f(0,1) = (\cos 120°, \sin 120°)$, (so that $f(x,y) = (x + y\cos 120°, y \sin 120°)$); see Figure 23. Note that the presence of the 45° lines on $\mathbf{Z}^2$ does not add any new points to $\mathbf{Z}^2$. They make the game easier to play, and may help in defining a useful connectivity concept, but other than that, they serve no real purpose.

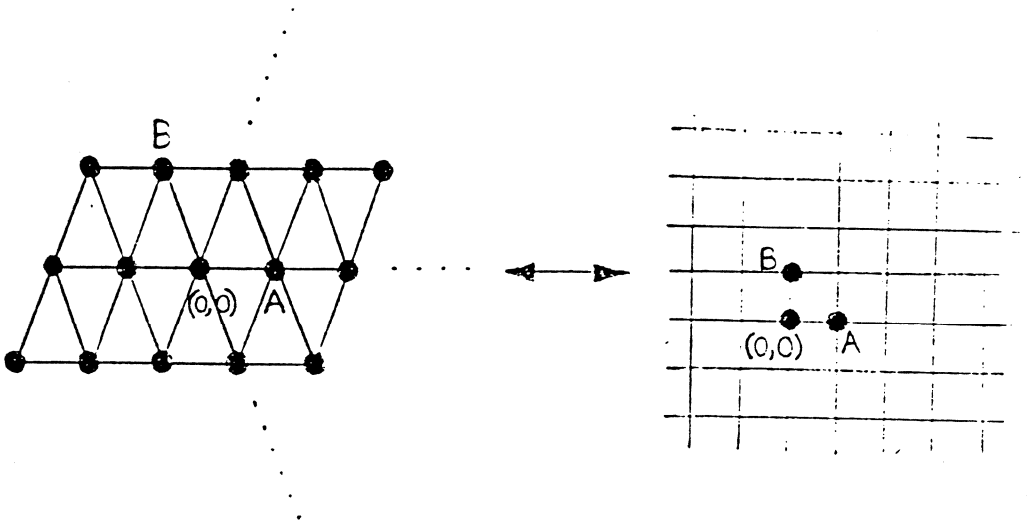


Figure 23: The vertex set of a Chinese checkers board is isomorphic to $\mathbf{Z}^2$.

- If the above gameboard and rules are extended to a third dimension, say $E = \mathbf{Z}^2 \times \mathbf{Z}_n$ where Z_n is the cyclic group of order n, then we can also apply the previous results, even if (*especially if*) placements P and $Q = P + (a_1, a_2, a_3)$, $a_3 \neq 0$, have fundamentally different maneuvers available from them. (Note that P and Q are not *free*-translates of each other.) It is precisely such possibilities that give non-trivial content to this section's generalization of Chapter 2's scenario $E = \mathbf{Z}^m$.

5.3 $G \subseteq E \subseteq \overline{E} = \mathbf{Z}^m \times T$.

Suppose for a motivating example that our movement environment E is the vertex set of the regular hexagonal tiling of $\mathbf{R}^2$ (see figure 24). Choose an arbitrary vertex as $(0,0)$, and then label each vertex by its Cartesian coordinates relative to the origin (each hexagonal side has unit length). E can be regarded as a subset (though not a subgroup) of the group $U = (\mathbf{R}^2, +)$. In fact, E can be regarded as a subset of a much smaller subgroup of U, namely $\overline{E}$, the subgroup of U generated by the points of E. Note that

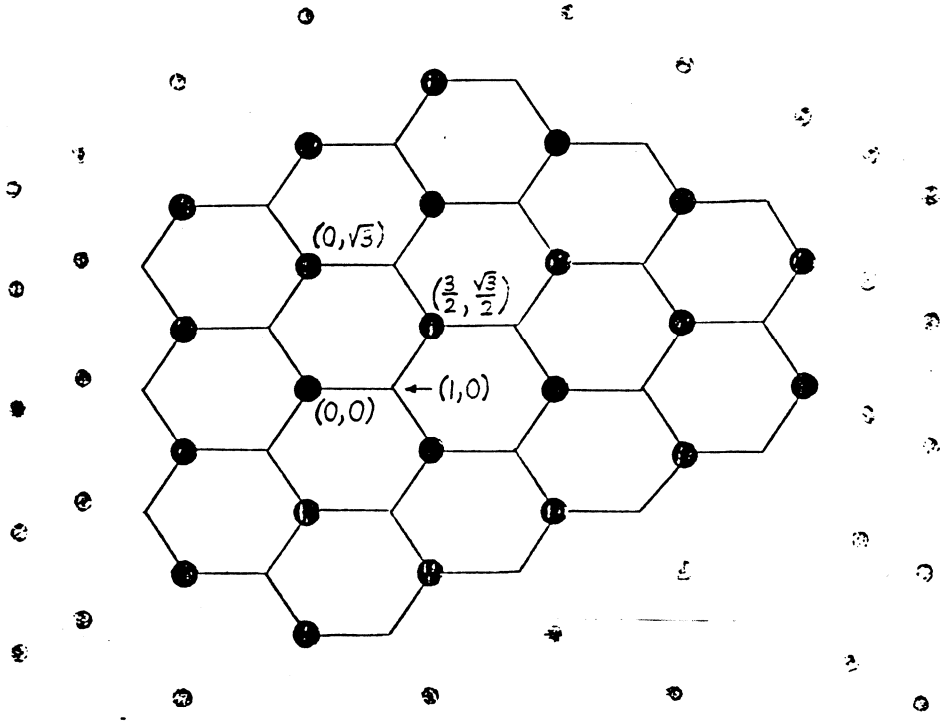


Figure 24: Hexagonal Tiling

all points in E can be reached from $(0,0)$ by a piecewise-linear path where each piece has length 1, and direction $\pm\mathbf{v}_1$, $\pm\mathbf{v}_2$, or $\pm\mathbf{v}_3$, where $\mathbf{v}_1 = (1,0)$, $\mathbf{v}_2 = (\cos 120°, \sin 120°) = (-1/2, \sqrt{3}/2)$, and $\mathbf{v}_3 = (\cos 60°, \sin 60°) = (1/2, \sqrt{3}/2)$. (Note that $\mathbf{v}_3 = \mathbf{v}_1 + \mathbf{v}_2$.) Thus $V \equiv \{a\mathbf{v}_1 + b\mathbf{v}_2 : a, b \in \mathbf{Z}^1\}$ is a group containing E; whence, $V \supseteq \overline{E}$. On the other hand, since $\mathbf{v}_1, \mathbf{v}_2 \in E$, any group containing E must contain V. Whence $\overline{E} \supseteq V$. Thus $\overline{E} = V$. (Note that V is just the triangularization from the last section.) Furthermore E contains a subgroup of $\overline{E}$, call it G, denoted by the shaded vertices in Figure 24 (and generated by the points $(3/2, \sqrt{3}/2)$ and $(0, \sqrt{3})$). Naturally, $\overline{E}$ is the disjoint union of its G-cosets, i.e., $\overline{E} = G \cup (G + (1,0)) \cup (G + (2,0))$. But unnaturally, we have, in addition, that $E = G \cup (G + (1,0))$.

One anticipates that turnpiking should occur over so regular an environment as E, if the movement rules are at all "sensible". After all, E is really just two superimposed copies of G (itself a finitely generated abelian

group). We shall prove this assertion for *any* environment E satisfying the following conditions:

1. E is a subset of some finitely generated abelian group $\overline{E}$.

2. E contains G, a (finitely generated, abelian) subgroup of $\overline{E}$, such that $E = \bigcup_{i=1}^{k}(G + e_i), e_i \in E, i = 1 \ldots, k$, where k is finite.

Condition 2 says that E is the (disjoint) union of some finite (generally, proper) subset of the cosets of G (relative to $\overline{E}$). (If $E = \overline{E}$, then E is itself a finitely generated abelian group, a case which was handled in the last section.) Since $\overline{E}$ is a finitely generated abelian group, we can assume without loss of generality that $\overline{E} = \mathbf{Z}^m \times T$, with T a finite abelian group. Therefore, $G \subseteq E \subseteq \mathbf{Z}^m \times T$, with G a subgroup of $\mathbf{Z}^m \times T$. Note further that the "free part of G", $G_F = \{x \in \mathbf{Z}^m : (x, 0) \in G\}$, is a subgroup of $\mathbf{Z}^m$. (The fact that G_F, as a subgroup of $\mathbf{Z}^m$, remains free and abelian is clear; that G_F must also be finitely generated follows from Theorem 10.3, Fuchs (1960).) Observe that by Condition 2 any $e \in E$ can be uniquely expressed as $e = g + e_i$, $g \in G$, $i \in \{1 \ldots k\}$. Further, if we *translate* $e \in E$ by $\delta \in G$, then the resulting point $e + \delta = (g + \delta) + e_i$ remains in the same coset $G + e_i$, and in particular, remains in E.

We can now define placements over E as before. By the above observation, for any placement $P \subseteq E$ and any $g \in G$, the placement $P + g \subseteq E$ is well-defined. We say that $Q \subseteq E$ is a *free-G-translate* of $P \subseteq E$, if $Q = P + g$ for some $g \in G_F \times \{0\}$. Since this defines an equivalence relation among the placements of E, we can define *configurations* (i.e., equivalence classes) under free-G-translations. As before, all placements in the same equivalence class have the same set of T components. Also as before, every placement P can be uniquely represented as $(X, \mathbf{a}) \equiv X + (\mathbf{a}, 0)$, where $\mathbf{a} \in G_F$, and the canonical placement of P is X (aesthetically "placed" at some e_i).

Let $(X, \mathbf{a}) \xrightarrow{c} (Y, \mathbf{b})$ denote a legal maneuver from placement $(X, \mathbf{a})$ to placement $(Y, \mathbf{b})$ with cost $c \in \mathbf{Z}^1$. Such a move is said to have progress $\mathbf{b} - \mathbf{a} \in G_F \subseteq \mathbf{Z}^m$. Now G_F is a (free) subgroup of $\mathbf{Z}^m$ and is therefore necessarily isomorphic to $\mathbf{Z}^i$ for some $i \leq m$ (see, e.g., Theorem 12.1, Fuchs (1960)). Thus, we could create a new $\mathcal{C}$-Graph isomorphic to the old one with all its progress data from $\mathbf{Z}^i$.

We only need to modify our movement rules slightly in order to obtain the desired turnpike theorems. Specifically, letting $\mathcal{C}$ denote our set of allowable configurations over free-G-translations, we replace $\mathbf{Z}^m$ throughout by $G_F \subseteq \mathbf{Z}^m$. This enables us to construct the appropriate $\mathcal{C}$-Graph with the usual correspondences. Here, cycles represent simple, free-G-translations, and a trajectory from $(\mathcal{O}, \mathbf{0})$ to $(\mathcal{D}, d\mathbf{b})$ corresponds to a walk from node $\mathcal{O}$ to node $\mathcal{D}$ with total progress $d\mathbf{b}$ and with the same cost. From the theorems of Chapter 2, we know there exist such walks of near-minimum cost which spend almost all the cost (for d sufficiently large) traversing at most i cycles of the $\mathcal{C}$-Graph. Here there is a near-optimal trajectory from $(\mathcal{O}, \mathbf{0})$ to $(\mathcal{D}, d\mathbf{b})$ which spends most of its cost simply and freely-G-translating at most m configurations. The proof of Theorem 2.2 proceeds just as before, by virtue of its $\mathcal{C}$-Graph representation.

As for Theorem 2.3, we make the following observations. First, we shall require that the radius of maneuver vector $\mathbf{r}$ of the brute force assumption belongs to G_F (for our applications, the $\geq$ relation does not make much sense over the torsion component), and that our isomorphism $G_F \cong \mathbf{Z}^i$ carries the "meaningful" relation "$\delta \geq \mathbf{r}$" into the numerical one on $\mathbf{Z}^i$. Thus all the progress data are expressed in terms of G_F. Now the proof of Theorem 2.3 proceeds exactly as before over this new $\mathcal{C}$-Graph (since the constants $M, \mathbf{e}$, and $\hat{\mathbf{b}}$ of that proof are now well defined.) Hence the analog of Theorem 2.3 is true in this more general environment (again, with $\leq m$ cycles replaced by $\leq i$ cycles).

Remark: The partitioning of E into a *finite* number of cosets of G is implied by the movement rules, without loss of optimality. For suppose that $E = \bigcup_{i \in I}(G + e_i)$, for some index set I. As noted before, if $x \in P \subseteq E$, where $x \in G + e_i$, then for any $(\mathbf{a}, 0) \in G_F \subseteq G$, $x + (\mathbf{a}, 0) \in G + e_i$. Thus, each configuration in our $\mathcal{C}$-Graph involves at most p (the number of pieces) cosets of G. Overall, the $\mathcal{C}$-Graph represents at most $p|\mathcal{C}| < \infty$ cosets. Hence we may assume $|I| \leq p|\mathcal{C}| < \infty$, without loss of asymptotic optimality.

5.4 Finitely Generated Abelian Groups Again

The results of sections two and three pertain to problems satisfying *free* homogeneity requirements. That is, they allow for placements which are

non-free translates to have different maneuvering capabilities. Now suppose that $E = G = \mathbf{Z}^m \times T$, where T is a finite, abelian group (i.e., $E = G$ is a finitely generated abelian group), and that our movement rules are invariant over non-free translations, as well. Since all the necessary free-homogeneity conditions (and more) are met, the results of sections two and three apply here too. However, one suspects that the $\mathcal{C}$-Graph constructed is about $|T|$ times larger than it really needs to be. For computational purposes, it would be useful to reduce the size of the $\mathcal{C}$-Graph. In this section, we make the above ideas precise.

Specifically, we assume that our rules for movement satisfy, in addition to the conditions of section two, a stronger translation-invariance condition, namely: for all placements $P, Q \subseteq E$, and all $\mathbf{a} \in G$, $P \xrightarrow{c} Q$ if and only if $(P + \mathbf{a}) \xrightarrow{c} (Q + \mathbf{a})$. (In Section 5.2, this was only assured when the T component of $\mathbf{a}$ was zero.)

Thus we define an expanded equivalence relation, namely the familiar translation relation via elements of G (rather than the previous $G_F \times \{\mathbf{0}\}$). Hence the configurations are the equivalence classes $\{X + \mathbf{a} : \mathbf{a} \in G\}$, where X is the canonical placement for the class. However, due to torsion effects (not present in the earlier sections) it may be possible that $X + \mathbf{a}_1 = X + \mathbf{a}_2$, where $\mathbf{a}_1 \neq \mathbf{a}_2$. (For example, the placement $P = \{(0,0),(0,1)\} \subset \mathbf{Z}^1 \times \mathbf{Z}_2$ is equal to $P + (0,1)$ if the two pieces are indistinguishable.) To treat this possibility, we define, for a placement $P \subseteq E$, the *stabilizer* of P to be $H_P = \{g \in G : P + g = P\}$.

Lemma 5.1 *H_P is a subgroup of G.*

Proof. Clearly, $\mathbf{0} \in H_P$. Suppose $g^1, g^2 \in H_P$. Then $P + (g^1 + g^2) = (P + g^1) + g^2 = P + g^2 = P$. Hence, $g^1 + g^2 \in H_P$. Further, $P = P + (g^1 - g^1) = (P + g^1) - g^1 = P - g^1$. Hence, $-g^1 \in H_P$. □

Lemma 5.2 *If $Q = P + \mathbf{a}$, $\mathbf{a} \in G$, then $H_Q = H_P$.*

Proof.

$$\begin{aligned} g \in H_Q &\iff Q + g = Q \\ &\iff (P + a) + g = P + a \\ &\iff P + g = P \\ &\iff g \in H_P \end{aligned}$$

□

(The converse of this lemma is easily seen to be false, by noticing, for example that two placements with stabilizer $\{\mathbf{0}\}$ need not be translates of each other.)

Thus, if X is a canonical placement, H_X will be the stabilizer of every placement in the corresponding equivalence class. For that configuration $\{X + \mathbf{a} : \mathbf{a} \in G\}$, we see that $X + \mathbf{a} = X + \mathbf{b}$ if and only if $\mathbf{a} - \mathbf{b} \in H_X$ (i.e., $\mathbf{a} \equiv \mathbf{b} \pmod{H_X}$). Note that by the same "minimum coordinate" argument used before, we have for all $g \in H_X$, $g_i = 0, i = 1 \ldots m$. Thus, H_X is a subgroup of $\{\mathbf{0}_m\} \times T$ and is therefore finite. We shall adopt the notation $(X, \mathbf{a})$ to denote placement $X + \mathbf{a}$, where it is understood that $(X, \mathbf{a} + h), h \in H_X$, represents the same placement.

As before, we let $(X, a) \xrightarrow{c} (Y, b)$ denote a cost c maneuver from placement (X, a) to placement (Y, b). Note that this maneuver can also be represented by $(X, a+h_x) \xrightarrow{c} (Y, b+h_y)$ for any $h_x \in H_X, h_y \in H_Y$. Note further that since for $X, Y \in \mathcal{C}$, H_X and H_Y are finite abelian groups, then so is $H_X + H_Y \equiv \{h_x + h_y : h_x \in H_X, h_y \in H_Y\}$. Observe that if $(X, 0) \xrightarrow{c} (Y, \delta)$ is a legal move, then so is $(X, 0) \xrightarrow{c} (Y, \delta + h)$ for any $h \in H_X + H_Y$. For if $h = h_x + h_y$, $h_x \in H_X, h_y \in H_Y$, then $(X, 0) = (X, -h_x)$, $(Y, \delta) = (Y, \delta + h_y)$. Thus, $(X, -h_x) \xrightarrow{c} (Y, \delta + h_y)$ is a legal move. Hence, by homogeneity, $(X, -h_x + h_x) \xrightarrow{c} (Y, \delta + h_y + h_x)$ or equivalently, $(X, 0) \xrightarrow{c} (Y, \delta + h)$ is a legal move.

We define a *formal trajectory* τ of length n as a formal sequence

$$(X_0, a_0) \xrightarrow{c_1} (X_1, a_1) \xrightarrow{c_2} \cdots \xrightarrow{c_n} (X_n, a_n)$$

such that $(X_{i-1}, a_{i-1}) \xrightarrow{c_i} (X_i, a_i)$ is a legal maneuver. The above trajectory is *equal* to the trajectory

$$(Y_0, b_0) \xrightarrow{c'_1} (Y_1, b_1) \xrightarrow{c'_2} \cdots \xrightarrow{c'_n} (Y'_n, b'_n)$$

if and only if $n = n', X_i = Y_i, c_i = c'_i, a_i = b_i, i = 1 \ldots n$. For notational convenience, we occasionally abbreviate τ as $(X_0, a_0) \xrightarrow{c} (X_n, a_n)$, where $c = \sum_{i=1}^{n} c_i$.

Hence the associated maneuvering sequence from placement (X_0, a_0) to placement (X_n, a_n) is represented by each of $\prod_{i=0}^{n} |H_{X_i}|$ different trajectories. The total cost of the trajectory is defined to be $\sum_{i=1}^{n} c_i$, and its total

progress is defined to be $a_n - a_0$. Note that the total cost of a trajectory is the same as the cost of the maneuvering sequence that it represents. Thus, to find a minimum cost maneuvering sequence from placement (X, a) to placement (Y, b), it suffices to find a minimum cost trajectory with first term (X, a) and last term (Y, b).

We now construct a $\mathcal{C}$-Graph in the usual way. Specifically, the graph has one node for each configuration, and an arc exists from node X to node Y with cost $c \in \mathbf{Z}^1$ and progress $a \in G$ if and only if $(X, 0) \xrightarrow{c} (Y, a)$ is a legal maneuver. (Note that such an arc implies the existence of an arc from X to Y with cost c and progress $a + \delta$, for all $\delta \in H_X + H_Y$.) Without loss of generality, no two arcs between the same pair of nodes have the same cost and progress, since the moves associated with such arcs accomplish the same movement at the same cost.

Lemma 5.3 *There is a one to one correspondence between the set of (length n) formal trajectories from $(X_0, 0)$ to (X_n, a_n) with cost c and the set of walks on the $\mathcal{C}$-Graph (of length n) from node X_0 to node X_n with total cost c and total progress a_n.*

Proof. Let $S1$ be the set of length n, cost c trajectories from $(X_0, 0)$ to (X_n, a_n), and let $S2$ be the set of length n walks from node X_0 to node X_n with total cost c and total progress a_n. Consider $\tau \in S1$, say $\tau : (X_0, 0) \xrightarrow{c_1} (X_1, a_1) \xrightarrow{c_2} \cdots \xrightarrow{c_n} (X_n, a_n)$, whence $c = \sum_{i=1}^{n} c_i$. Define $f : S1 \to S2$, by taking $f(\tau)$ to be the length n walk over nodes $X_0, X_1, \ldots, X_n$ (in that order, repetition allowed), where the i-th arc has cost c_i and progress $a_i - a_{i-1}, i = 1 \ldots n$.

$f(\tau)$ is well defined since $(X_{i-1}, a_{i-1}) \xrightarrow{c_i} (X_i, a_i)$ represents a legal maneuver, and there is exactly one arc from node X_{i-1} to node X_i with cost c_i and progress $a_i - a_{i-1}$. $f(\tau)$ has length n, total cost $\sum_{i=1}^{n} c_i = c$, and total progress $(a_1 - 0) + \sum_{i=2}^{n}(a_i - a_{i-1}) = a_n$.

Consider in $S2$ walk w over nodes $X_0, X_1, \ldots, X_n$, in that order, where the i-th arc has cost γ_i and progress α_i, $i = 1 \ldots n$. Whence, $\sum_{i=1}^{n} \gamma_i = c$ and $\sum_{i=1}^{n} \alpha_i = a_n$. Then $w = f(\nu)$, $\nu \in S1$, where $\nu : (X_0, 0) \xrightarrow{c_1} (X_1, a_1) \xrightarrow{c_2} \cdots \xrightarrow{c_n} (X_n, a_n)$, $c_k = \gamma_k$, and $a_k = \sum_{i=1}^{k} \alpha_i$. Thus f is onto.

Futhermore, any trajectory $\tau \in S1$ satisfying $f(\tau) = w$ must clearly satisfy $a_1 = \alpha_1$. Since $\alpha_2 = a_2 - a_1$, $a_2 = a_1 + \alpha_2 = \alpha_1 + \alpha_2$. Proceding

inductively, we have that $\alpha_k = a_k - a_{k-1}$. Whence, $a_k = a_{k-1} + \alpha_k = \sum_{i=1}^{k} \alpha_i$. (Note that $a_n = \sum_{i=1}^{n} \alpha_i$ is consistent.) Hence, the inverse image of w is unique, and f is a one to one correspondence, as desired. □

Hence, if our objective is to find a minimum cost trajectory (representing a sequence of maneuvers) from placement $(\mathcal{O}, \mathbf{0})$ (wlog, by space homogeneity) to placement $(\mathcal{D}, d\mathbf{b}, \tau_d)$, $(d > 0, \mathbf{b} \in \mathbf{Z}^m, \tau_d \in T)$, then by the previous lemma, for any $(\mathbf{0}, \delta) \in H_{\mathcal{D}}$, it suffices to find a minimum cost walk from node $\mathcal{O}$ to node $\mathcal{D}$ with total progress $(d\mathbf{b}, \tau_d + \delta)$. Notice that cycles in the $\mathcal{C}$-Graph correspond to G-translations of configurations.

Theorem 5.1 *Under the movement rules of Chapter 2, with* $\mathbf{Z}^m$ *replaced throughout by* G*, there exists a near optimal turnpike solution to the above maneuvering problem. Specifically, for* d *sufficiently large, we can construct a trajectory that accrues almost all of its cost simply translating at most* m *configurations. The cost of this trajectory differs from the optimal trajectory by a constant which does not depend on* d, $\mathbf{b}$ *or* τ_d.

Proof. The proof closely mimics the proofs of Theorems 2.2 and 2.3 (for Theorem 2.3's analog, we must assume $\mathbf{b} > \mathbf{0}_m$.) Note that by the brute force assumption and the finiteness of T, there exists a constant c_T such that $(X, \mathbf{a}, \tau_1) \xrightarrow{c_T} (X, \mathbf{a}, \tau_2)$ is a legal maneuver for any $X \in \mathcal{C}, \mathbf{a} \in \mathbf{Z}^m, \tau_1, \tau_2 \in T$.

To construct our turnpike trajectory, we first construct the turnpike trajectory from $(\mathcal{O}, \mathbf{0}_m)$ (not $m+1$) to $(\mathcal{D}, d\mathbf{b})$, with cost TPFREE, temporarily ignoring the torsion constraints. (We do this by removing the torsion component from all the arcs of the current $\mathcal{C}$-Graph, and then proceeding as in Chapter 2—we do *not* coalesce arcs which become identical in the process.) Upon returning the torsion components to the arcs used in the above walk, our walk represents a trajectory from $(\mathcal{O}, \mathbf{0}_m, 0)$ to $(\mathcal{D}, d\mathbf{b}, \tau_1)$ for some $\tau_1 \in T$. Next we brute force from $(\mathcal{D}, d\mathbf{b}, \tau_1)$ to $(\mathcal{D}, d\mathbf{b}, \tau_d)$, with cost c_T. Whence our turnpike trajectory

$$(\mathcal{O}, \mathbf{0}_m, 0) \overset{TPFREE}{\longrightarrow} (\mathcal{D}, d\mathbf{b}, \tau_1) \xrightarrow{c_T} (\mathcal{D}, d\mathbf{b}, \tau_d)$$

has total cost TPCOST = TPFREE $+ c_T$. As in the proof of Theorem 2.2,

$$TPCOST \leq mc_{\mathbf{0}} + \bar{c} + u^* + c_T.$$

Similarly, if $(\mathcal{O}, \mathbf{0}_m, 0) \xrightarrow{c^*} (\mathcal{D}, d\mathbf{b}, \tau_d)$ is an optimal trajectory, then by considering

$$(\mathcal{O}, \mathbf{0}_m, 0) \xrightarrow{c^*} (\mathcal{D}, d\mathbf{b}, \tau_d) \xrightarrow{c_{\mathbf{0}}} (\mathcal{O}, d\mathbf{b}, \tau_d),$$

we must have $c^* \geq -c_{\mathbf{0}} + u^*$. Whence,

$$TPCOST - c^* \leq (m+1)c_{\mathbf{0}} + \bar{c} + c_T,$$

which does not depend on d, $\mathbf{b}$, or τ_d.

The proof of the analog of Theorem 2.3 is essentially unchanged. The only difference is that the final brute force of the turnpike trajectory takes us from $(T_m, m\mathbf{r} + \sum_{j=1}^{m} \lfloor x_j \rfloor \mathbf{a}_j, \tau)$, for some $\tau \in T$, to $(\mathcal{D}, d\mathbf{b}, \tau_d)$, with cost $c_{(\delta,\tau')}$, $\tau' = \tau_d - \tau \in T$. The set $\{\delta \in \mathbf{Z}^m : \mathbf{r} \leq \delta \leq \mathbf{r} + 2mM\mathbf{e}\} \times T$ remains finite. u^* and z^* play the same roles as before, and the new theorem is proven. □

Remarks:

- When $m = 1$, we have the analog of Theorem 2.1 .
- The implementation ideas of Chapter 3 can still be used for the more general problem. Note that for every configuration X here, there are $|T|/|H_X|$ configurations in the free $\mathcal{C}$-Graph of Section 2. Hence the ratio of the number of nodes in the new $\mathcal{C}$-Graph to the number of nodes in the old one is $|\mathcal{C}|/\sum_{X \in \mathcal{C}}(|T|/|H_X|)$. If most X have $H_X = \{0\}$, then this ratio is approximately $1/|T|$.
- If our movement environment E is not all of $\overline{E} = \mathbf{Z}^m \times T$, but instead satisfies the additional two homogeneity conditions of Section 3, we say that placements P and $Q = P+g$, $g \in G$ are *G-translates*, and this defines an equivalence relation among the placements. Defining configurations and canonical placements in the usual way, we let (X, a) denote the placement $X + a, a \in G$. (Again, $X + a$ may have multiple ($|H_X|$) representations.) If our movement rules obey the usual conditions (with $\mathbf{Z}^m$ replaced by G), we can represent the problem on a $\mathcal{C}$-Graph (as in this section) with progress labels coming from G, a subgroup of $\mathbf{Z}^m \times T$, and thus $G \cong \mathbf{Z}^i \times T'$, where $i \leq m$, and T' is a finite group. The problem has now been reduced to the one considered in this section. Thus for d sufficiently large, there exists

a near optimal turnpike trajectory from $(\mathcal{O},(\mathbf{0},0_i))$ to $(\mathcal{D},(d\mathbf{b},\tau_d))$ which spends most of its cost doing simple G-translations of at most $i \leq m$ different configurations.

Chapter 6
Applications and Research Directions

6.1 Applications

In this section we present some problems that can be posed as $\mathcal{C}$-Graph problems, and thereby have turnpike solutions. Interestingly, the first two problems we consider are not even optimal maneuvering problems.

6.1.1 The Recycling Problem

We begin with an original *contraption* $\mathcal{O}$ with d units of "life" in it. After the contraption is *utilized*, it is *transformed* into another contraption, whereby it loses life units, and is subsequently utilized. The process is continued until we no longer can (or desire to) transform any further. The objective is to maximize the total amount of utility achieved during this process.

More precisely and more generally, we have a finite set $\mathcal{C}$ of contraptions. The value of a contraption $X \in \mathcal{C}$ is $u(X) \in \mathbf{Z}^1$ monetary units. A transformation takes a contraption X to a contraption Y at a cost of $t_{XY} \in \mathbf{Z}^1$ monetary units, and a destruction of $\mathbf{p} \in \mathbf{Z}^m$ life units. Hence the net monetary value of the transformation is $u(Y) - t_{XY} \equiv c_{XY}$. (There may be more than one way to transform from X to Y, but we assume this number is finite. In such a case, we would adopt new notation for c_{XY}.) Initially, we begin with contraption $\mathcal{O} \in \mathcal{C}$ with $d\mathbf{b} \in \mathbf{Z}^m$ life units ($\mathbf{b} \geq \mathbf{0}_m$, d large). We wish to make a sequence of transformations consuming at most $d\mathbf{b}$ life units, and maximizing the total monetary value achieved. We assume (as in the homogeneity assumptions) that the amount of life units remaining to a contraption does not affect its utility nor its transformation parameters. We also assume (as in the brute force assumption) that there exists a vector $\mathbf{r}$ such that for any $X, Y \in \mathcal{C}$, and $\delta \geq \mathbf{r}$, there is a sequence of transformations from X to Y with value c_δ and "destruction" δ. Lastly, as in the cycle positivity progress assumption at the end of Chapter 2, we have a "law of entropy" asserting that for any $X \in \mathcal{C}$, any sequence of transformations taking X back to itself must destroy a *positive* amount of

life units.

With these assumptions, the problem can be modeled by a $\mathcal{C}$-Graph and the theorems of the second chapter apply. (Cf. the remarks after Theorem 3 there.) Our turnpike solution spends most of its time repeatedly applying at most m different transformation cycles. Hence our recycling problem has, so to speak, a re-cycling solution.

6.1.2 Crop Rotations

A farmer has a plot of land from which he can grow a single crop each year. The land has m nutrients, $i = 1 \ldots m$. If one year he plants crop X, and the next season he plants crop Y, then he makes a profit of γ_Y, but uses up $p_{XYi} > 0$ units of each nutrient i. Assuming initial nutrient endowments $b_i \gg 0$, and that nutrients cannot replenish themselves, how should the farmer schedule his crop plantings so as to "reap" the greatest profit (while his land is usable). Using a $\mathcal{C}$rop graph analogous to the $\mathcal{C}$ontraption graph in the previous subsection, we obtain our desired near-optimal turnpike sequence of "crop rotations". Note that in this problem, the associated $\mathcal{C}$-Graph is likely to be complete (one node for each crop), and that the radius of maneuver $\mathbf{r} = (r_1, \ldots, r_m)$ can be defined as $r_i = \max_{XY} p_{XYi} > 0$. Thus the amount of soil deterioration from one season to the next is at most $\mathbf{r}$. Note further that we can weaken the positivity assumption on p_{XYi} by the appropriate cycle positivity assumption, and that the profit γ_Y can be taken to depend on the previous season's crop (and thereby be denoted as γ_{XY}).

Remark: Perhaps more realistically, suppose the farmer wishes to maximize his profit, discounted over time. That is, the money he earns during his second season is multiplied by a discount factor α, the money he earns in his third season is multiplied by α^2, etc., where $0 < \alpha < 1$. If α is small enough, the profit obtained the first few seasons may dominate the remaining profit, and therefore it may not pay to get onto a turnpike immediately (if indeed a turnpike solution is even near-optimal). Perhaps a turnpike result for this situation could be established in a manner analogous to that of Shapiro and Wagner (1967). Another possible objective could be that the farmer only intends to plant for t seasons, where t is large but finite. If such is the case, we add a new constraint to our problem. In fact, we

could even remove the pessimistic constraint that nutrients never replenish themselves, by allowing the land to "go to pasture" for some seasons at the expense of losing time. This type of situation is developed more fully in the next example.

6.1.3 Optimal (Bi)cycling

Before presenting this application, we state a weaker cycle positivity progress assumption for maximization problems when the associated LP has $m = m_1 + m_2 + m_3$ constraints, where the first m_1 constraints are the $\leq$ constraints, and the next m_2 constraints are the equality constraints, and $m_3 \neq m$.

Weaker Cycle Positivity Progress Assumption. Let C_i be a cycle with positive value c^i and progress $\mathbf{a}_i \in \mathbf{Z}^m$. Then $a_{ij} > 0$ for some $j = 1, \ldots, m_1 + m_2$.

This ensures that the associated LP remains bounded.

A team of $m-1$ marathon bicyclists wants to see how far they can travel within t time periods. The cyclists can maneuver into a finite number of different configurations. Depending on a cyclist's talents and position in a configuration, he expends a certain amount of energy during that time period (e.g., a bicyclist "in front" encounters more wind resistance than the other cyclists, and thus expends more energy here than if he were in another position). At the beginning of each time period, the cyclists can change into a new configuration (say from configuration X to configuration Y). The distance travelled during this time period (which begins with making the transition to Y) is denoted as γ_{XY}, and p_{XYi} units of energy (perspiration?) is exhausted by cyclist i, where the notation is chosen to be analogous to the cost and progress parameters used earlier. The cyclists can even maneuver into a "resting" configuration, where they can recuperate energy (but travel a distance of 0 for each time period there).

We may now construct a $\mathcal{C}$-Graph having a node for each configuration (from a finite set $\mathcal{C}$), and an arc between every pair of nodes $X, Y \in \mathcal{C}$ (perhaps $X = Y$), with distance and energy parameters γ_{XY} and p_{XY} as described above. Letting $\mathcal{O}$ denote the resting configuration, the objective is to find a walk from node $\mathcal{O}$ to itself such that the sum of the distances

(gammas) of the arcs is maximized, subject to the constraints that the number of arcs used is at most t, and that the sum of energies exhausted is at most $\mathbf{b}$, where $b_i \gg 0$ is the initial amount of energy of bicyclist i.

The LP used to construct the turnpike solution is: $\min \mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} \leq (\mathbf{b}, t)^T$, $\mathbf{x} \geq \mathbf{0}$, where if the i-th cycle is denoted as C_i, then its cost is $-\mathbf{c}^i = \sum_{XY \in C_i} \gamma_{XY}$, and its progress $\mathbf{a}_i = \sum_{XY \in C_i} (\mathbf{p}_{XY}, 1)^T$ is the i-th column of A. (The 1 denotes the one unit of time used up in that time period.) It may be the case that $\mathbf{p}_{XY} \geq \mathbf{p}_{YY}$ and $\gamma_{YY} \geq \gamma_{XY}$ for all $X, Y \in \mathcal{C}$, representing the extra energy and time expended while making the transition from X to Y. (Note that most of the time during this period is spent in configuration Y.) In such a case a simple cycle $X_1 \to X_2 \to \cdots \to X_s \to X_1$ is dominated by the set of s loop cycles $X_1 \to X_1$, $X_2 \to X_2, \ldots, X_s \to X_s$, which makes at least as much progress, and uses no more energy. Thus, we could restrict our attention to the loop cycles only, when constructing our turnpike solution. Hence, we could solve the associated LP directly without needing to resort to column generation (provided of course that the number of configurations is itself of manageable size).

In the earlier crop rotation example, it may have been the case that $\gamma_{YY} \leq \gamma_{XY}$ and $\mathbf{p}_{YY} \geq \mathbf{p}_{XY}$ if more nutrients were absorbed when a crop was immediately repeated in the following season, and the quality (and hence price) of the crop was diminished. Here, the turnpike cycles would tend *not* to include loops.

6.1.4 Other Potential Applications

$\mathcal{C}$-Graph like objects have been used to analyze VLSI designs with regular structure. (See Kosaraju and Sullivan 1988, Orlin 1984, and Iwano and Stegilitz 1987. The design gives rise to a (generally infinite) *dynamic graph* and an associated finite *static graph.* (See Figure 25, from the last of the papers just cited.) Since brute-force and positive cycle assumptions are generally not assumed for problems of this type (for example, Kosaraju and Sullivan developed and analyzed algorithms for determining whether a static graph has a non-trivial cycle with progress $\mathbf{0}_m$), $\mathcal{C}$-Graphs can be viewed as a special class of static graphs with exploitable structure.

Problems arising in robotics often involve the maneuvering of a single

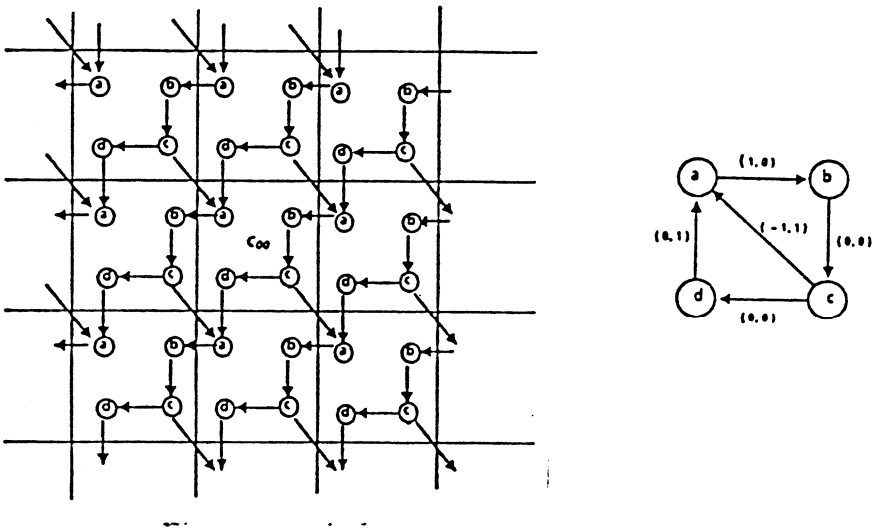


Figure 25: A dynamic graph and a static graph

piece (i.e., a robot) to a particular destination, while avoiding various obstacles in the environment. If there are too many obstacles (irregularly placed) in the environment, then one could not expect to see regular turnpike-like maneuvering. When there are not too many obstacles, Brooks (1983) has shown how one can find feasible (though not necessarily minimum cost) trajectories by using long corridors of free space, denoted as "freeways" where the robot can move freely, generally staying a good distance away from obstacles. The analogy between freeways and turnpikes is tempting. However, it seems unlikely that brute-force and space homogeneity assumptions are likely to be realistic, since often the real issue involved is the detection of *feasible* trajectories.

6.2 Research Directions

Finally, we mention a number of questions remaining open, and intended for future research:

- When is the finiteness assumption valid? Are there natural conditions sufficient to imply finiteness? For instance, we have not yet been able

to show even that a 2-dimensional jumping problem with connected origin and destination configurations must necessarily have a near-optimal trajectory whose intermediate configurations are confined to a finite set (e.g., of connected configurations).

- In Section 1.3, we characterized the fastest translating configurations for m-dimensional jumping problems (the "speed of light configurations" with speed 1, available only for one, two and four pieces), but were unable to prove that the next fastest simple translation speed was 2/3. (Such a result would resolve the last issue in the paragraph above.)

- The main theorems indicate that the turnpike solution is almost as good as an optimal solution. When can we prove the stronger claim that there exists an optimal solution with the turnpike property? The papers by Gilmore and Gomory (1966) and Chrétienne (1984) may suggest approaches to such a problem.

- Suppose we extend the "turnpike" analogy to a hierarchy of roads (corresponding, say, to cycles in the $\mathcal{C}$-Graph) with successively higher speeds. Does there exist an optimal trajectory consisting of subtrajectories which first rise through this hierarchy (starting at the level of the origin configuration) and then descend (to the destination's level)?

- Consider the deterministic Markov Decision Problem over a given directed graph with rewards on the arcs. The objective is to find a walk of length n (possible $n = \infty$) starting at a particular node $\mathcal{O}$ with maximum (perhaps discounted) total reward (see Howard, 1960). Adapting a result of Shapiro 1968, we have that for n sufficiently large, there is an optimal solution that spends most of its time traversing one particular cycle. This is related to a variant of our (unit cost 1-dimensional) $\mathcal{C}$-Graph problem, where the aim is to accumulate a prescribed amount of reward (progress) in the least amount of time. (Here, the particular destination node is unspecified.) These problems seem very much related, and perhaps results from one area can supplement results from the other. For instance, it may be possi-

ble to define a "node value" for each configuration represented in our $\mathcal{C}$-Graph. Also, we may find some natural non-deterministic extension of our problem.

- When the optimal solutions are necessarily disconnected, do they still demonstrate turnpike-like behavior? For instance, it may be optimal to break up seven pieces into two groups of sizes four and three which move separately in turnpike fashion, possibly joining up somewhere, then splitting up into configurations of sizes five and two for further separate movement, and then finally rejoining to maneuver to the goal in sequential fashion. This suggests the further generalization of exploring "parallel movement" of connected configurations.

- Our emphasis on asymptotic results, and free use of the "brute force" assumption, has permitted us to *finesse* the issues of *efficient* entry to and exit from the turnpike cycle (when $m = 1$), and of *efficient* sequencing of (and transitions between) traversals of the various cycles, when $m > 1$. Both for applicability to situations in which the asymptotic effects are not overwhelming, and for satisfaction of natural puzzle-solving curiosity, these issues call for serious study.

In addition to the theoretical issues raised above, many concrete open problems remain:

- Add more maneuvering problems to our data set, both recreational (e.g., a non-trivial higher dimensional sliding problem, or a problem arising out of the more complex environments of Chapter5, like the hexagonal tiling of the plane) and applied (e.g., in such fields as military logistics, automated manufacturing or robotic movement (the paper by Mitchell (1987) is suggestive)). The developments in Sections 5.3 and 5.4 may lend themselves to maneuvering problems in environments where obstacles are regularly arranged.

- Perform probabilistic analysis and further computational experimentation for the improved negative-cycle detection concepts of Section 3.2.

- Alternative versions of the column generation approach of Section 3.1 should be explored and compared with the alternative solution method of Section 3.4. The heuristic proposals at the end of Section 3.4 should also be tested.

- How can we automate the construction of the C-Graph from "natural" descriptions of its nodes (i.e. configurations) and arcs (i.e. legal moves)? Can this construction be usefully interwoven with the solution algorithms sketched in the preceding sections? Along these lines, it appears fruitful to explore the connection between our turnpike cycles and the *macro-operators* used in artificial intelligence. To quote Iba (1985), "Macro-operators (or macros) are a kind of super-operator which is composed of ... more elementary operators. Macros derive their power from their ability to shorten the search process. Using macros, it is possible to take 'larger strides' through the search space, since applying a single macro may be equivalent to a large number of more primitive steps.... The result is that a solution may be found in a significantly smaller number of search steps than would be required by a search using only the basic operator.... Macros allow for more efficient search as well as more economical representation and recall of solutions." AI techniques that generate and test efficient macros may provide a means of generating a manageable number of useful turnpike cycles (and configurations) based on simple descriptions of movement rules, when the generation of the entire C-Graph is undesirable.

We close by suggesting that the particular mathematical constructs identified in this dissertation, that of "high-speed cycles in a C-Graph", and its multi-dimensional generalization to *sets* of cycles, should prove generally valuable in the treatment of many optimal-maneuvering problems. The preceding results provide encouraging initial evidence, which we hope will be confirmed by the additional investigations outlined above.

Appendix A: Detecting Negative Cycles

In this appendix, we illustrate our negative cycle detection scheme with two small problems. (Both appear as exercises in Lawler, 1976.) The actual computer code. called NEGCYC.PAS and written in the language Pascal, follows.

The data structure used to represent weighted directed graphs is an array of linked lists. The i-th list contains those nodes reachable from i in one step (df., "A(i)" in Section 3.2), plus the cost of that step. (See Figure 26.) As was discussed in Section 3.2, the i-th element of array u will denote the "current" shortest walk length from origin node 1 to node i, and PRED(i) is the predecessor node of i on such a shortest walk.

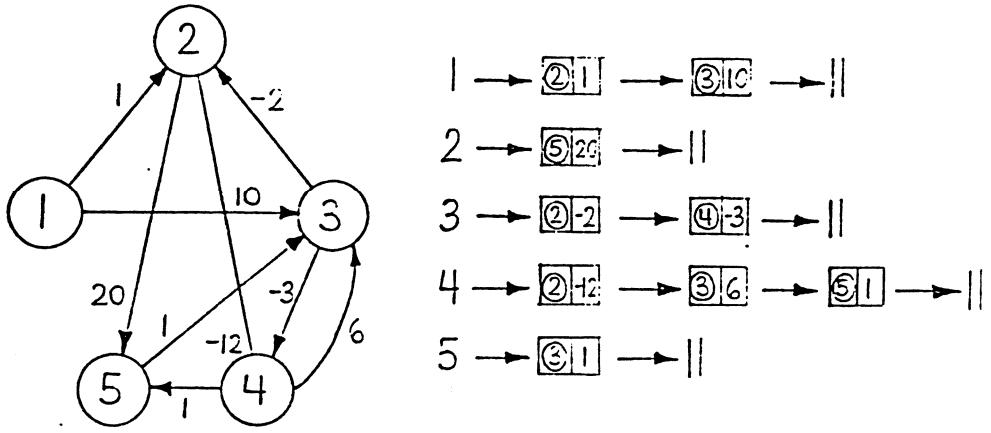


Figure 26: A weighted digraph and its data structure.

EXAMPLE 1:

Initially, we set

	1	2	3	4	5
u :	0	∞	∞	∞	∞

	1	2	3	4	5
$PRED$:	–	–	–	–	–

where a horizontal line denotes "Nil".

Iteration 1

We begin by going through the adjacency list 1: $u[2] := \min(0+1, \infty) = 1^*$ and $u[3] := \min(0+10, \infty) = 10^*$. (An asterisk denotes an improvement in the u-value.) This gives us

	1	2	3	4	5
u :	0	1	10	∞	∞

	1	2	3	4	5
$PRED$:	–	1	1	–	–

Next we go through the adjacency list of node number 2. (Note that this step would be unnecessary if node number 2 had not improved just previously.) $u[5] := \min(1+20, \infty) = 21^*$ Whence,

	1	2	3	4	5
u :	0	1	10	∞	21

	1	2	3	4	5
$PRED$:	–	1	1	–	2

Going through adjacency list 3: $u[2] := \min(10-2, 1) = 1$ and $u[4] := \min(10-3, \infty) = 7^*$, yielding

	1	2	3	4	5
u :	0	1	10	7	21

	1	2	3	4	5
$PRED$:	–	1	1	3	2

Adjacency list 4: $u[2] := \min(7-12, 1) = -5^*$, $u[3] := \min(7+6, 10) = 10$, and $u[5] := \min(7+1, 21) = 8^*$, yielding

	1	2	3	4	5
u :	0	-5	10	7	8

	1	2	3	4	5
$PRED$:	–	4	1	3	4

Adjacency list 5: $u[3] := \min(8+1, 10) = 9^*$, yielding

	1	2	3	4	5
u :	0	-5	9	7	8

	1	2	3	4	5
$PRED$:	–	4	5	3	4

Tracing back from those nodes which were improved this iteration, we see that Pred[2] = 4, Pred[4] = 3, Pred[3] = 5, while Pred[5] = 4, giving us the negative cycle $4 \xrightarrow{1} 5 \xrightarrow{1} 3 \xrightarrow{-3} 4$ in just one iteration.

EXAMPLE 2:

Next, consider the weighted digraph of Figure 27. We initialize, as before:

	1	2	3	4	5
u :	0	∞	∞	∞	∞

	1	2	3	4	5
$PRED$:	–	–	–	–	–

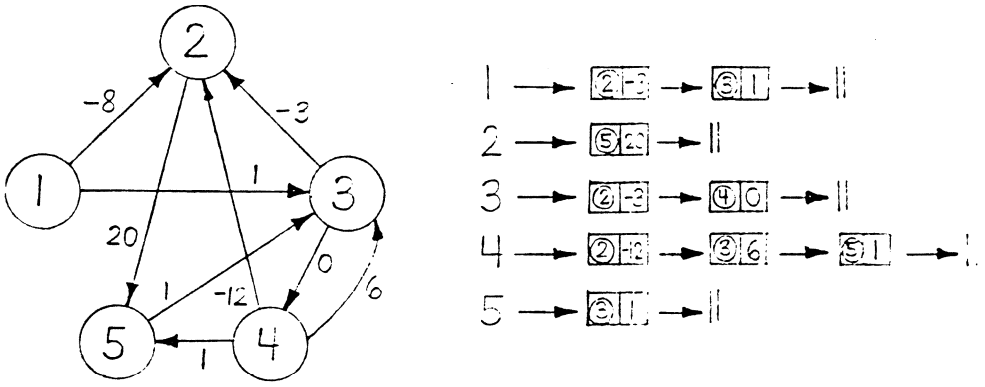


Figure 27: Another weighted digraph and its data structure.

Iteration 1

Adjacency list 1: $u[2] := -8^*$ and $u[3] := 1^*$.

	1	2	3	4	5
u :	0	-8	1	∞	∞

	1	2	3	4	5
$PRED$:	–	1	1	–	–

Adjacency list 2: $u[5] := -8 + 20 = 12^*$.

	1	2	3	4	5
u :	0	-8	1	∞	12

	1	2	3	4	5
$PRED$:	–	1	1	–	2

Adjacency list 3: $u[2] := \min(1-3, -8) = -8$ and $u[4] := 1 + 0 = 1^*$.

	1	2	3	4	5
u :	0	-8	1	1	12

	1	2	3	4	5
$PRED$:	–	1	1	3	2

Adjacency list 4: $u[2] := \min(1-12, -8) = -11^*$, $u[3] := \min(1+6, 1) = 1$, and $u[5] := \min(1+1, 12) = 2^*$.

	1	2	3	4	5
u :	0	-11	1	1	2

	1	2	3	4	5
$PRED$:	–	4	1	3	4

Adjacency list 5: $u[3] := \min(2+1, 1) := 1$. (No changes)

Tracing back from those nodes which were improved this iteration, we see Pred[2] = 4, Pred[4] = 3, Pred[3] = 1, Pred[1] = 0 does not lead to

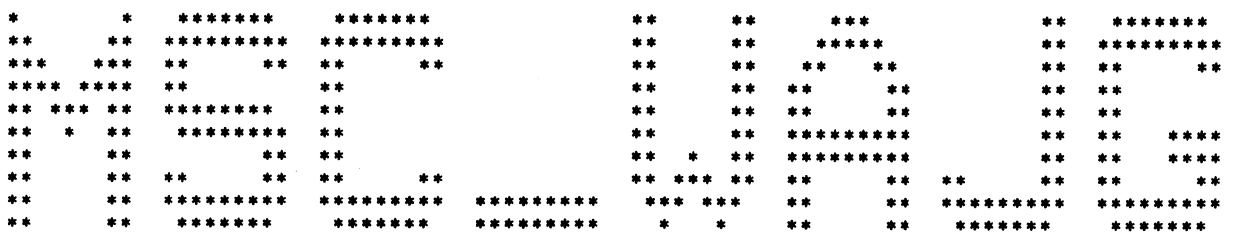
a cycle, and Pred[5] = 4, which has already been explored. Hence, no negative cycles have been detected this iteration.

Iteration 2

Since $u[1]$ has not improved since the last time we went through node 1's list, there is no need to go through it this iteration.

Adajcency list 2: $u[5] := \min(-11 + 20, 2) = 2$. (No changes)

Likewise, there is no reason to explore lists 3, 4 and 5. Since no improvement has been made this iteration, we may conclude that no negative cycles exist in our digraph. Furthermore, the array Pred and u can be used to back-trace shortest paths from node 1 to all other nodes, and the array u to read off their lengths.



FILE: NEGCYC PAS

TIME: 04/18/89 14:49:57

SYSTEM: JHUVMS

PRINTER: RM_LINE

MODE: STANDARD

```
Program NegCycles(Input,Output,Infile,Outfile); (**This program finds a negative cycle in a weighted graph, if one exists.**)
                                                (**The graph is read from file infile. **)
                                                (** Input instructions appear at the end of this documentation.**)

Const Nmax = 501;                               (*max # of nodes*)
      BigM = 100;                               (*infinity value for shortest path computations*)
      Epsilon = 0.0001;  (*10^-4*)              (*used to avoid improving u based on roundoff errors*)

Type  Nodetype = 0..Nmax;
      nvector = Array[1..nmax] of Integer;
      boolvector = Array[1..nmax] of Boolean;

      ptr = ^Arc;
      Arc = Record
              Tail: Nodetype;
              Head: Nodetype;
              Cost: Integer;
              Next: Ptr;                        (* next arc on adjacency list *)
              NextonCycle: Ptr;                 (* next arc on the negative cycle *)
            End;

      PtrArray = Array[1..Nmax] of ptr;
      Filename = Packed Array[1..12] of Char;

Var   N : Nodetype;                             (* # of nodes in the C-graph *)
      A : PtrArray;                             (* A[I] is a linked list of all arcs leaving node I *)
      Infile, Outfile : Text;
      Infilename, Outfilename: Filename;

(*************************************************************************)
Procedure Readinfile(Var Infilename: Filename);         (* Readinfile Reads the name of the input file *)

Begin
 Write('Enter the name of the input file: ');
 Readln(Infilename);
 Open(Infile, File_Name:= Infilename, History:= Old);
 Reset(Infile);
End;
(*************************************************************************)
Procedure Readoutfile(Var Outfilename: Filename);       (* Readoutfile reads the name of the output file *)

Begin
 Write('Enter the name of the output file: ');
 Readln(Outfilename);
 Open(Outfile, File_Name:= Outfilename);
 Rewrite(Outfile);
End;
(*************************************************************************)
Procedure GetData(Var N: Nodetype; Var A: PtrArray);    (* Getdata initializes variables and creates the *)
                                                        (* adjacency lists from the input file.          *)

Var   K, Tail, Head: Nodetype;
      P,Q: ptr;

Begin
 Readln(Infile, N);                                     (* read n *)
 For K:= 1 to N Do
  A[K]:= Nil;
 Read(Infile,Tail);
 While (Tail <> 0) Do
  Begin  (* While *)                                   (* Record the information from the arc Tail----->Head *)
    New(P);
    P^.Tail:= Tail;                                     (* Read tail *)
    Read(Infile, Head);                                 (* Read head *)
    P^.Head:= Head;
    Readln(Infile, P^.cost);                            (* Read cost *)
```

```
    P^.Next:= A[Tail];                                  (* Insert arc at beginning of adjacency list *)
    A[Tail]:= P;
    Read(Infile, Tail);                                 (* Input ends with a 0 *)
 End; (* While *)
End;
(*************************************************************************)
Procedure Writedata(Var A: Ptrarray;  N: Nodetype;Infilename:Filename);
                                                        (* Writedata writes the adjacency list into the output file *)
Const Screensize = 8;
Var   I: Nodetype;
      K: 0..Screensize;
      P, Q: Ptr;

Begin
 Writeln;
 Writeln(outfile,'Adjacency List for Data from ',Infilename);
 Writeln(outfile);
 For I:= 1 to N Do                                      (* Print the adjacency list for node I *)
   Begin
     Write(outfile,I:2,':');
     P:= A[I];
     Q:= A[I];
     While Q <> Nil Do
        Begin (*While*)
          For K:= 1 to Screensize Do                    (* At most 8 nodes at a time, *)
             Begin
               If P <> Nil Then
                  Begin
                    Write(outfile,'------>',P^.Head:2);
                    P:=P^.Next;
                  End;
             End;
          Writeln(Outfile);
          For K:= 1 to Screensize Do
             Begin
               If Q <> Nil Then                         (* listing the cost underneath its respective arc. *)
                  Begin
                    Write(outfile,'    ','(',Q^.cost:1,')');
                    Q:=Q^.Next;
                  End;
             End; (* For K loop *)
          Writeln(outfile);
          Writeln(outfile);
        End;    (* While *)
 End;          (* For I loop *)
End;
(*************************************************************************)
Procedure NoCycleMessage(I:Nodetype);       (* Nocyclemessage prints ``no neg. cycles found after I iterations.'' *)

Begin
 Writeln(outfile);
 Writeln(outfile);
 Writeln;
 Writeln;
 Writeln('No negative cycles detected; Termination after ',I:2,' iterations.');
 Writeln(outfile,
  'No negative cycles detected; Termination after ',I:2,' iterations.');
End;
(*************************************************************************)
Procedure BellmanFord(Var A,Pred: PtrArray; Var u: nvector;     (* Procedure BellmanFord does one iteration of the Bellman-Ford *)
                      Var Improvement: Boolean;                 (* shortest path algorithm, updating the vector u, and          *)
                      Var N: Nodetype;                          (* indicating which components of u have improved during that   *)
                      Var ImprovedSinceLastTime: Boolvector;    (* iteration (via ImprovedThisTime).  After the I-th iteration  *)
                      Var ImprovedThisTime: Boolvector);        (* of BellmanFord, u[J] is  the weight of a minimum weight      *)
                                                                (* walk from node one to node J, among all walks in the I-th    *)
                                                                (* class.  The I-th class contains (properly) all walks from    *)
```

```
                                                        (* node one to node J which take I steps or fewer. Pred[J] is   *)
                                                        (* the node  which precedes node J in some such minimum  weight*)
                                                        (* walk from node one to node J.                                *)
Var  Tail, I: Nodetype;
     P: Ptr;

Begin
 For I:= 1 to N do
   ImprovedThisTime[I]:= False;
 For Tail:= 1 to N do                                   (* Going through all arcs with tail Tail... *)
 If ImprovedSinceLastTime[Tail] Then                    (* that have improved since last time these arcs were examined, *)
  Begin
    ImprovedSinceLastTime[Tail]:= False;
    P:= A[Tail];
    While P <> Nil Do
      Begin
        If (u[Tail] + P^.cost  < u[P^.Head] - epsilon)  Then    (* If we can find a shorter walk to node Head with (Tail,Head) *)
                                                                (* as its last step, then update u[Head] and Pred[Head].       *)
          Begin
            Improvement:= True;
            u[P^.Head]:= u[Tail] + P^.cost;
            Pred[P^.Head]:= P;
            ImprovedSinceLastTime[P^.Head]:= True;
            ImprovedThisTime[P^.Head]:= True;
          End;
        P:= P^.Next;                                    (* Examine the next arc on Tail's out-adjacency list. *)
      End;  (*While*)
  End;
End;
(*************************************************************************)
Procedure Traceback(Node, N: Nodetype;
                    Var StartofCycle:  Nodetype; Pred: PtrArray;
                    Var ExaminedBy: Nvector;
                    Var NegCycleFound, NewCycFound: Boolean);   (* Traceback starts with node Node and then follows pred[Node] *)
                                                                (* to the next node, then follows pred of this next node to    *)
                                                                (* another node, and so on, until either we reach nil or some  *)
                                                                (* node repeats itself. In the latter case, we have found a    *)
                                                                (* negative cycle, starting at the repeated node (StartofCycle)*)

Var P: ptr;
    DeadEnd: Boolean;

Begin
 ExaminedBy[Node]:= Node;
 DeadEnd:= False;
 P:= Pred[Node];
 While ((P <> Nil) And (Not NewCycFound) And (Not DeadEnd)) Do
  Begin
    If ExaminedBy[P^.Tail] = Node Then                  (* If the tail of our arc has been examined during this call, *)
     Begin
       NegCycleFound:= True;                            (* then a new negative cycle has been found  *)
       NewCycFound:= True;
       StartofCycle:= P^.Tail;                          (* starting with node StartofCycle. *)
     End;
    If ExaminedBy[P^.Tail] = 0 Then                     (* If the tail of our arc has not been examined, then   *)
      ExaminedBy[P^.Tail]:= Node;                       (* examine it.                                          *)
    If ExaminedBy[P^.Tail] <> Node  Then                (* If the tail of our arc has been examined in a previous call *)
      DeadEnd:= True;                                   (* then explore no further.                             *)
    P:= Pred[P^.Tail];                                  (* Trace back one more arc via Pred *)
  End;
End;
(*************************************************************************)
Procedure PrintNegCycle(Pred:PtrArray; ImprovedNode, Iter, N, StartofCycle: Nodetype);

                                                        (* PrintNegCycle prints out the negative cycle detected in *)
                                                        (* DetectNegCyc, starting at node StartofCycle.  It also   *)
```

```
                                                    (* prints the cycle's total  cost.                    *)

Var Node: Nodetype;
    P, ArcStack: ptr;
    Totalcost: Integer;

Begin
 Writeln(outfile);
 Writeln(outfile);
 Write(outfile,'Using Improved Node ',ImprovedNode:2);
 Writeln(outfile,'  at iteration ',Iter:2);
 Writeln;
 Writeln;
 Write('Using Improved Node ', ImprovedNode:2);
 Writeln('  at iteration ',Iter:2);
 Writeln(outfile);
 Writeln(outfile);
 Writeln;
 Writeln;
 Arcstack:= Pred[StartofCycle];
 Arcstack^.NextonCycle:= Nil;
 Node:= Pred[StartofCycle]^.Tail;
 While Node <> StartofCycle Do
   Begin                                            (* Put edges of cycle on arcstack in reverse order  *)
     Pred[Node]^.NextonCycle:= Arcstack;
     Arcstack:= Pred[Node];
     Node:= Pred[Node]^.Tail;
   End;
 P:= Arcstack;
 Totalcost:= 0;
 Repeat                                             (* Pop off edges in the proper order, computing and *)
                                                    (* printing the relevant information.               *)
   Write(P^.Tail,'---->',P^.Head:3);
   Writeln('  Cost: ',P^.Cost:2);
   Write(outfile,P^.Tail,'---->',P^.Head:2);
   Writeln(outfile, '  Cost: ',P^.Cost:2);
   TotalCost:= TotalCost + P^.Cost;
   P:= P^.NextonCycle;
 Until P = Nil;
 Writeln;
 Writeln('  TotalCost: ',TotalCost:3);
 Writeln;
 Writeln;
 Writeln;
 Writeln(Outfile,'  TotalCost: ',TotalCost:3);
 Writeln(Outfile);
 Writeln(Outfile);
 Writeln(Outfile);
End;
(*************************************************************************)
Procedure DetectNegCyc(Var A:PtrArray; N:Nodetype);
                                                    (* DetectNegCycle                                   *)
                                                    (* determines if the graph contains a cycle of negative *)
                                                    (* It calls BellmanFord at most N times.  If after  *)
                                                    (* any such call, u is not improved in any component, then *)
                                                    (* the graph has no negative cycles.  When u does improve, *)
                                                    (* the arcs leading into improved nodes are traced  *)
                                                    (* back.  If a cycle is found, it must be negative, and it *)
                                                    (* is printed out.  Otherwise we call BellmanFord again. *)
Var  I, Iter, StartofCycle: Nodetype;
     P : Ptr;
     u: nvector;
     Improvement, NegCycleFound, NewCycFound: Boolean;
     Pred: Ptrarray;
     ISLT, ImprovedThisTime: Boolvector;            (* ISLT[i] is true if node i has Improved Since Last Time *)
     Examinedby: Nvector;                           (* Node i's out adjacency list was examined.        *)
```

```
                                                    (* ImprovedThisTime[i] is true if u[i] has improved during    *)
                                                    (* the present call of BellmanFord.                           *)
                                                    (* ExaminedBy makes the TraceBack procedure more efficient.   *)
Begin
 u[1]:= 0;
 ISLT[1]:= True;
 For I:= 2 to N do
  Begin
    u[I]:= BigM;
    ISLT[I]:= False;
  End;                                              (* Initialize u and ISLT *)
 Improvement:= True;
 NegCycleFound:= False;
 Iter:= 1;
 For I:= 1 to N Do
  Pred[I]:= Nil;
 While ((Improvement) And (Iter <= N) And (Not NegCycleFound))  Do
   Begin
     Improvement:= False;
     BellmanFord(A, Pred, u, Improvement, N, ISLT, ImprovedThisTime);
     If Improvement Then
       Begin                                                  (* Trace back through all improved nodes *)
         For I:= 1 to N do
           ExaminedBy[I]:= 0;
         For I:= 1 to N do
           If (ImprovedThisTime[I] And (ExaminedBy[I] = 0))  Then
             Begin
               NewCycFound:= False;
               Traceback(I, N, StartofCycle, Pred, Examinedby, NegCycleFound, NewCycFound);
               If NewCycFound Then                          (* and print out all negative cycles found. *)
                PrintNegCycle(Pred, I, Iter, N , StartofCycle);
             End;
       End;
     Iter:= Iter + 1;
   End;
 If Not Improvement Then  NoCycleMessage(Iter - 1);
 End;

(*************************************************************************)
(*********************  MAIN  PROCEDURE  *********************************)
(*************************************************************************)

Begin
 Readinfile(Infilename);
 GetData(N,A);
 Close(Infile);
 Readoutfile(Outfilename);
 Writedata(A,N,Infilename);
 DetectNegCyc(A,n);
 Close(Outfile);
End.

(*****************************************************************************************************************)
(******                                    INPUT INSTRUCTIONS                                                 *******)
(*****************************************************************************************************************)
(* THE DATA ARE ENTERED VIA SOME INPUT FILE (E.G., LAWLER1.DAT). THE DATA ARE LEFT JUSTIFIED AS FOLLOWS:          *)
(* N ----THE NUMBER OF NODES IN THE GRAPH (ACTUALLY, THE HIGHEST NUMBERED NODE IN THE GRAPH)                     *)
(* LIST THE ARCS, ONE ROW AT A TIME, AS FOLLOWS:                                                                 *)
(* TAIL   HEAD   COST                                                                                            *)
(* 0 ----INDICATES THE END OF THE INPUT                                                                          *)
(*****************************************************************************************************************)
(*  EXAMPLE-- THE FOLLOWING DATA APPEAR IN LAWLER1.DAT:
5
1 2 1
1 3 10
2 5 20
```

```
3 2 -2
3 4 -3
4 2 -12
4 3 6
4 5 1
5 3 1
0
                                                                *)
```

Appendix B
Generating Columns

In this appendix, we illustrate our column generation scheme with two large examples arising from the two-dimensional jumping game described in Chapter 1. Here we have three pieces, and restrict ourselves to the 46 "connected" configurations. Figure 28 displays each configuration placed at $(0,0)$.

In the first example, THREEPC, all moves have unit cost. In the second example, RANDCG, costs of the moves vary from 1 to 9 (and were generated from a uniform distribution using a table of random digits—with the restriction that every move was given the same cost as its reversal move). We solve the above maneuvering problems for *all* "right hand side" goal directions. This can be done easily in two dimensions, since the edges of the unit square completely surround the origin, and a solution to an LP with right hand side $d\mathbf{b}$ is just "d times" a solution with right hand side $\mathbf{b}$. Thus, since our problem data are rational, we can use integral right hand sides. In the style of Appendix A, the $\mathcal{C}$-Graph is represented by an array of linked lists, where the i-th list consists of the arcs leaving node i. (More exactly, the arc's entry on the list is a record containing information such as the arc's cost and progress, and pointing to the next arc on the list or indicating none exists.)

As columns are generated, they are printed out, are given a name (starting with the first non-artificial cycle, #5), and their names, total cost, and progress are recorded and placed on a list of generated columns (called NonBasicList). Our simplex method goes through NonBasicList sequentially, and selects to enter the basis the first element on the list with negative reduced cost. (Alternative selection rules may be worth exploring.) If the variable leaving the basis is not artificial, it is placed at the top of the NonBasicList, and the new basis inverse is computed directly without pivots; this is practical since $m = 2$ or 3. We next supply input and annotated output for the two aforementioned problems. These are followed by documented Pascal code for the column generation scheme, COLGEN.PAS.

Annotated Output for Problem 1 — Threepc.out

1. Observe the presence of multiple arcs, representing different moves.

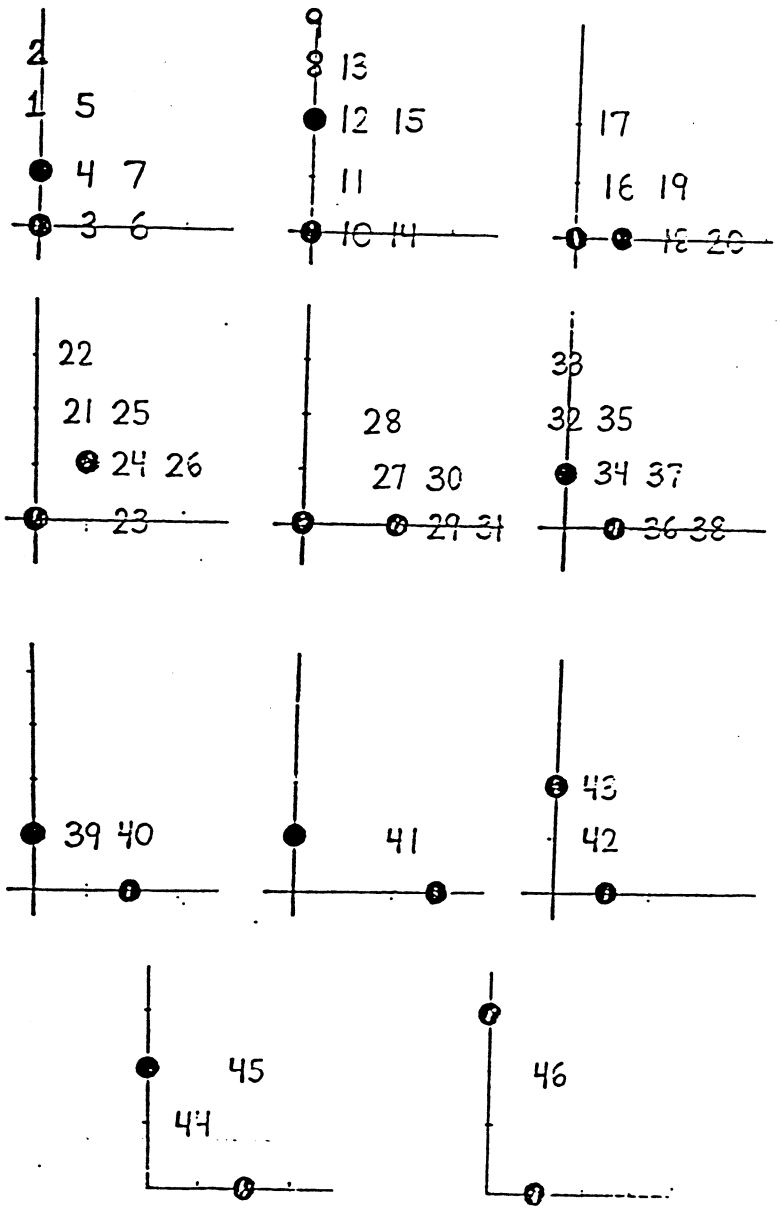


Figure 18: The connected three piece configurations. Each subfigure shows the positions of two pieces, and configuration-numbers corresponding to the different possible position of the third piece.

2. Artificial cycles have numbers from 1 to 4. The initial "goal direction" is $(1,0)$.

3. The non-basic list starts off empty except for a dummy header.

4. Using $\lambda = (100,100)$, the initial dual solution, we find a negative cycle in the first iteration of Bellman-Ford. (The details of finding such negative cycles are given in Chapter 3 and Appendix A.) Cycle #5 is the vertical one-dimensional Castells-Goldman cycle (with three pieces); it makes a progress of $(0,2)$ in 3 moves, and thereby has *speed* $2/3$. "Weights" are defined in section 3.1.

5. The cycle is brought into our basis degenerately, replacing artificial cycle #2.

6. With a new dual vector of $(100,1.5)$, our negative cycle detector finds two cycles. Cycle #6 requires 7 moves to translate all 3 pieces by the vector direction $(2,0)$. That is less efficient than simply doing 6 horizontal shifts, so we won't describe it further. It replaces the remaining artificial cycle #1 in the basis.

7. Cycle #7 translates the "snake" (configuration 25 in Figure 28 in the direction $(1,1)$ in 3 moves. This will not be useful for the current right hand side (abbr:RHS), but could be useful later.

8. Under the new dual vector $(3.5,-.5)$, we find cycle #8 which translates the anti-snake $\ddots$ in the direction $(1,-1)$ in 3 moves, and cycle #9 which performs 3 horizontal shifts to translate the snake in the direction $(1,0)$.

9. An interesting, fairly efficient cycle with speed $3/5$ (relative to the direction $(+x,-y)$. (See the illustration on output.) Notice how a piece shifts "upwards" in the move $36 \longrightarrow 39$, even though the overall translation is "downwards". Combining this with the anti-snake cycle yields an interesting (albeit sub-optimal) trajectory.

10. At last, we generate the horizontal one-dimensional Castells-Goldman cycle #11 making progress $(2,0)$ in 3 moves.

11. In the process of finding our (degenerate) turnpike solution (which utilizes only cycle number #11) we found many other efficient ones, notably cycles #5, #7, and #8. By dual feasibility, any RHS vector in the cone generated by the progress vectors in our optimal basis has an optimal solution with the same basis. Thus cycles #11 and #7 are an optimal basis for any direction with angle 0 to 45 degrees.

12. For example, with RHS $(2,1)$ (angle approximately 26 degrees) our basis remains optimal as illustrated.

13. When entering the new RHS $(1,1)$, our simplex procedure discovered an alternative optimum using cycles #7 and #5, (the latter degenerately), with progress directions $(1,1)$ and $(0,1)$ respectively.

14. Whence, by dual feasibility, this basis is optimal for all angles between 45 and 90 degrees, as illustrated by the next two right hand sides. For RHS $(0,1)$, it is cycle #7 that is degenerate in the basis.

15. Next, we examine another corner of the square. Having no generated columns that make progress in the negative horizontal direction, a partially artificial solution is given after simplexing.

16. Our negative cycle procedure detects 2 "hop-shift" cycles (of speed $1/2$), and also cycle #15, which is the reversal of the anti-snake cycle #8.

17. Note that cycles #13 and #14, though distinct, are considered the same in terms of our turnpike model, since they make the same progress at the same cost.

18. Cycle #15 yields a degenerate optimal solution for direction $(-1,1)$, and along with cycle #5, provides us with optimal solutions for angles of 90 to 135 degrees.

19. For direction $(-1,0)$, cycle #16, the reversal of cycle #11, is generated for the optimal solution. Along with cycle #15, this solves the problem for angles of 135 to 180 degrees.

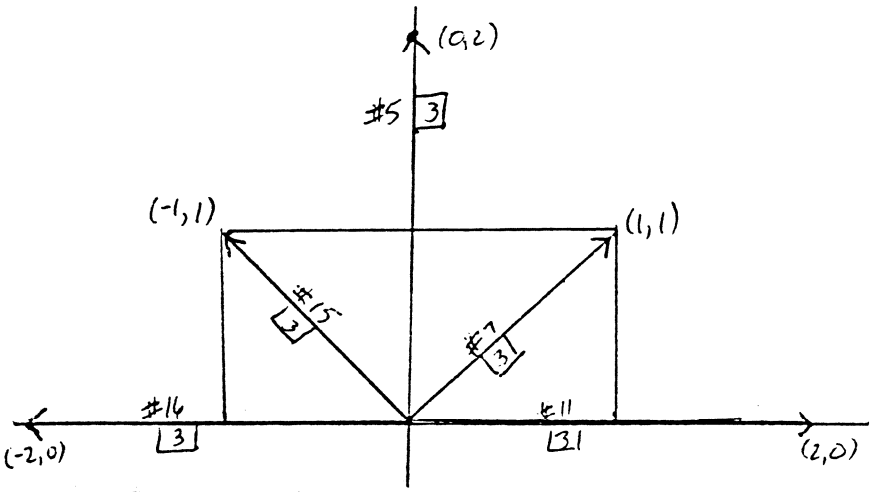


Figure 29: Optimal bases for problem 1

20. Because all moves in our problem are reversible with the same cost, the reversal of an optimal cycle pair for θ degrees is optimal for (360-θ) degrees. Hence, we have solved this problem for all angles. We verify this with a few more right hand sides.

21. Note that throughout this procedure, all negative cycles were detected in at most 4 (usually 1 or 2) iterations of the Bellman-Ford algorithm, and absence of negative cycles was verified within 6 iterations.

22. The optimal bases can be represented by the graph in Figure 29. Within each of the indicated 45-degree angular sectors, an optimal basis is given by the two cycles associated with the boundary rays of the sector.

Annotated Output for Problem 2 — Randcg.out

1. The cycles generated during Phase 1, while artificial variables are in the basis, are not guaranteed to be at all efficient. Cycles #5–#8 are in fact pretty inefficient.

2. For RHS (1,0), there is a degenerate optimal solution that utilizes only cycle #9 and consists of 3 inexpensive horizontal shifts, as illustrated. The non-utilized cycle (#11) has progress (1,2). Hence the RHS directions in the cone generated by these 2 vectors have an optimal solution with these as basis cycles. For an example, we examined the RHS (1,1).

3. While we know that the direction $(1,2)$ has a degenerate optimal solution using only cycle #11 (and #9, degenerately), when we re-ran the simplex method from an artificial basis, it generated an alternative optimum.

4. Cycle #13 is illustrated on the output. Note that it is cheaper to go from configuration #16 to configuration #19 to configuration #21 at a cost of 1+2=3 (with a net progress of $(0,0)$ than to go from #16 to #21 directly (with the same progress) at a cost of 6.

5. Thus all directions in the cone generated by $(1,2)$ and $(-1,3)$ have this same optimal basis.

6. Running the LP with RHS $(-1,3)$ did not generate an alternative optimum. Hence, we tried RHS $(-1,1)$, and hoped for the best...

7. It generates some inefficient cycles...

8. then some efficient ones...

9. then finally cycle #19, illustrated on the output.

10. The transition from configuration #21 to configuration #19 (with cost 13 and progress $(-2,3)$) is made as in the reverse of cycle #13.

11. Since cycle #13 remains (albeit degenerately) in our basis, cycles #19 and #13 form optimal bases for all right hand sides in the cone generated by $(-1,1)$ and $(-1,3)$.

12. Re-running the RHS $(-1,1)$ did not produce an alternate optimal basis.

13. Thus we tried RHS $(-1,0)$, and found that cycle #17 (the reverse of cycle #9) provided an optimal (degenerate) solution together with cycle #19, thereby solving all right hand sides between $(-1,1)$ and $(-1,0)$.

14. As before, since our moves were reversible *with the same cost*, we have solved the problem for all right hand sides.

15. For example, with the RHS $(-1, -1)$, our solution's basis is the reversal of cycles #9 and #11, as asserted.

16. The optimal bases can be represented by the graph in Figure 30.

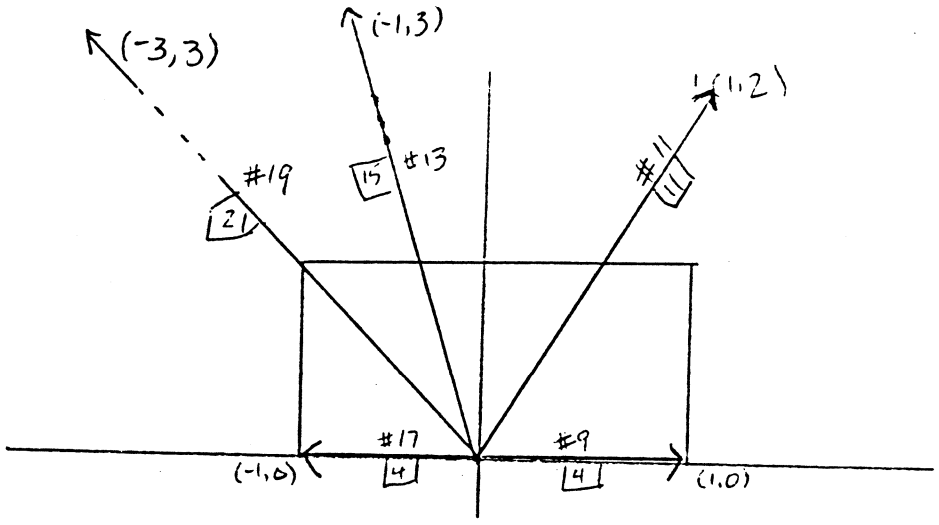
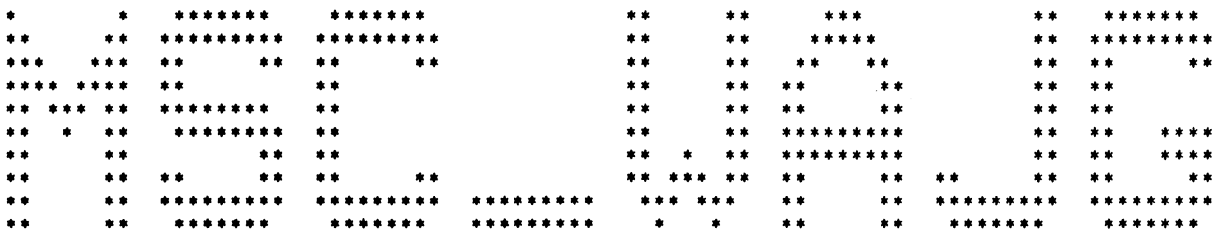


Figure 30: Optimal bases for problem 2



FILE: THREEPC OUT

TIME: 03/03/89 11:46:59

SYSTEM: JHUVMS

PRINTER: RM_HQLIN

MODE: STANDARD

Adjacency List for Data from threepc.dat

```
 1:------>42------>35------>21------> 8------> 2------>32------>11------> 8
   (-1,0)1   (-1,0)1   (-1,0)1   (0,-1)1   (0,-1)1   (0,0)1   (0,0)1   (0,0)1

------> 5------> 2
   (0,0)1   (0,0)1

 2:------>22------> 9------> 8------> 1------>33------> 8------> 1
   (-1,0)1   (0,-1)1   (0,1)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1

 3:------>36------>32------>23------>16------>11------> 4------>36------>32
   (-1,0)1   (0,-1)1   (-1,0)1   (-1,0)1   (0,-1)1   (0,-1)1   (0,0)1   (0,0)1

------> 6------> 4------>16------>10
   (0,0)1   (0,0)1   (0,0)1   (0,0)1

 4:------>37------>34------>24------>12------> 5------> 3------>34------> 7
   (-1,0)1   (-1,0)1   (-1,0)1   (0,-1)1   (0,-1)1   (0,1)1   (0,0)1   (0,0)1

------> 5------>24------>11------> 3
   (0,0)1   (0,0)1   (0,0)1   (0,0)1

 5:------>25------>13------>24------> 4------>35------>21------>12------> 4
   (-1,0)1   (0,-1)1   (0,1)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

------> 1
   (0,0)1

 6:------>38------> 7------> 7------>36------>23------>14------> 3
   (-1,0)1   (0,-1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

 7:------>26------>15------> 6------>37------>24------> 6------> 4
   (-1,0)1   (0,-1)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

 8:------>46------> 1------>13------> 9------> 2------> 2------> 1
   (-1,0)1   (0,1)1   (0,0)1   (0,0)1   (0,-1)1   (0,0)1   (0,0)1

 9:------> 2------> 8
   (0,1)1   (0,0)1

10:------>33------>17------>17------>32------>14------>11------> 3
   (0,-1)1   (-1,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

11:------> 3------>42------>12------>21------>10------> 4------> 1
   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

12:------>43------> 4------>43------>15------>13------>11------> 5
   (-1,0)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

13:------> 5------>12------> 8
   (0,1)1   (0,0)1   (0,0)1

14:------>10------> 6
   (0,0)1   (0,0)1

15:------> 7------>12
   (0,1)1   (0,0)1

16:------>35------>34------>27------>21------>19------> 3------>34------>23
   (0,-1)1   (0,-1)1   (-1,0)1   (0,-1)1   (-1,0)1   (1,0)1   (0,0)1   (0,0)1

------>21------>19------>17------> 3
   (0,0)1   (0,0)1   (0,0)1   (0,0)1

17:------>28------>22------>10------>35------>21------>16------>10
```

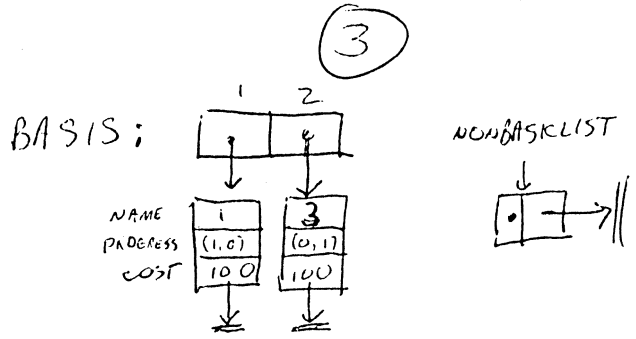
```
   (-1,0)1   (0,-1)1   (1,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

18:------>39------>37------>29------>24------>20------>36------>29------>23
   (0,-1)1   (0,-1)1   (-1,0)1   (0,0)1   (-1,0)1   (0,0)1   (0,0)1   (0,0)1

------>20------>19
   (0,0)1   (0,0)1

19:------>30------>25------>21------>16------>37------>27------>24------>18
   (-1,0)1   (0,-1)1   (1,0)1   (1,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

------>16
   (0,0)1

20:------>31------>26------>29------>18------>29------>18
   (-1,0)1   (0,-1)1   (1,0)1   (1,0)1   (0,0)1   (0,0)1

21:------> 1------>16------>25------>22------>19------>17------>16------>11
   (1,0)1   (0,1)1   (0,0)1   (0,0)1   (-1,0)1   (0,0)1   (0,0)1   (0,0)1

------> 5
   (0,0)1

22:------> 2------>17------>21
   (1,0)1   (0,1)1   (0,0)1

23:------> 3------>39------>24------>27------>18------>16------> 6
   (1,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

24:------> 4------>18------>26------>25------>23------>19------> 7------> 5
   (1,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,-1)1

------> 4
   (0,0)1

25:------> 5------>19------>24------>21
   (1,0)1   (0,1)1   (0,0)1   (0,0)1

26:------> 7------>20------>24
   (1,0)1   (0,1)1   (0,0)1

27:------>40------>16------>40------>30------>28------>23------>19
   (0,-1)1   (1,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

28:------>17------>27
   (1,0)1   (0,0)1

29:------>41------>18------>31------>30------>20------>20------>18
   (0,-1)1   (1,0)1   (0,0)1   (0,0)1   (-1,0)1   (0,0)1   (0,0)1

30:------>19------>29------>27
   (1,0)1   (0,0)1   (0,0)1

31:------>20------>29
   (1,0)1   (0,0)1

32:------>44------> 3------>36------>42------>35------>33------>10------> 3
   (-1,0)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

------> 1
   (0,0)1

33:------>10------>32------> 2
   (0,1)1   (0,0)1   (0,0)1

34:------>43------>42------>40------>39------> 4------>16------>39------>42
```

```
   (0,-1)1   (0,-1)1   (-1,0)1   (-1,0)1   (1,0)1   (0,1)1   (0,0)1   (0,0)1

------>37------>35------>16------> 4
   (0,0)1   (0,0)1   (0,0)1   (0,0)1

35:------> 1------>16------>43------>34------>32------>17------> 5
   (1,0)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

36:------>44------> 3------>39------>38------>37------>32------>18------> 6
   (0,-1)1   (1,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

------> 3
   (0,0)1

37:------> 4------>18------>40------>36------>34------>19------> 7
   (1,0)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

38:------> 6------>36
   (1,0)1   (0,0)1

39:------>41------>34------>18------>44------>40------>42------>36------>34
   (-1,0)1   (1,0)1   (0,1)1   (0,0)1   (0,0)1   (1,-1)1   (0,0)1   (0,0)1

------>23
   (0,0)1

40:------>45------>34------>27------>41------>39------>37------>27
   (0,-1)1   (1,0)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

41:------>39------>29------>40
   (1,0)1   (0,1)1   (0,0)1

42:------>46------> 1------>34------>44------>43------>39------>34------>32
   (0,-1)1   (1,0)1   (0,1)1   (0,0)1   (0,0)1   (-1,1)1   (0,0)1   (0,0)1

------>11
   (0,0)1

43:------>45------>12------>34------>46------>42------>35------>12
   (-1,0)1   (1,0)1   (0,1)1   (0,0)1   (0,0)1   (0,0)1   (0,0)1

44:------>32------>36------>42------>39
   (1,0)1   (0,1)1   (0,0)1   (0,0)1

45:------>43------>40
   (1,0)1   (0,1)1

46:------>42------>43
   (0,1)1   (0,0)1
```

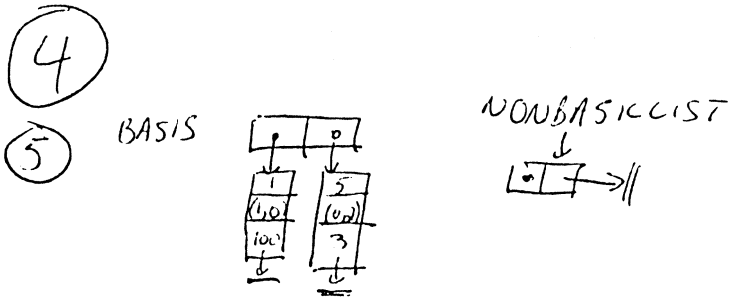
(1)

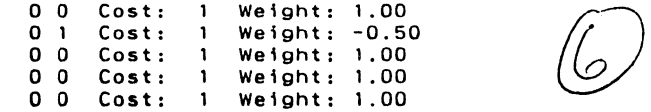
```
Original RHS vector:          1          0
Initial artificial basis:
Cycle number     1 Progress:     1    0      Cost:    100
Cycle number     3 Progress:     0    1      Cost:    100
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [100.00,100.00]


Cycle number     5
Discovered after iteration  1
```

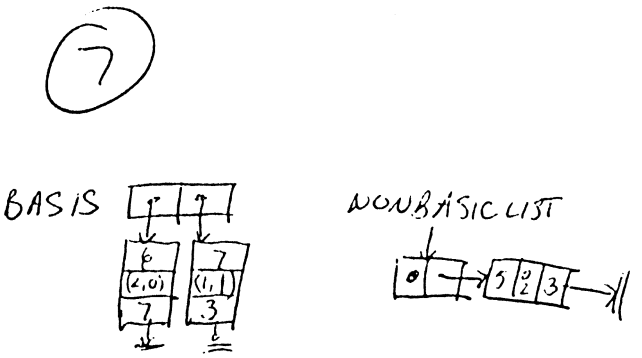
(2)



```
When lambda equals    100.00   100.00

         8----> 9    Progress:  0 0  Cost:   1  Weight: 1.00
         9----> 2    Progress:  0 1  Cost:   1  Weight: -99.00
         2----> 8    Progress:  0 1  Cost:   1  Weight: -99.00

  Total Weight: -197.00
Total Progress:    0  2  Total Cost:    3


New Basis:    1  5    Solution Cost: 100.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [100.00,  1.50]


Cycle number     6
Discovered after iteration  1


When lambda equals    100.00     1.50

        42---->44    Progress:  0 0  Cost:   1  Weight: 1.00
        44---->36    Progress:  0 1  Cost:   1  Weight: -0.50
        36---->37    Progress:  0 0  Cost:   1  Weight: 1.00
        37---->40    Progress:  0 0  Cost:   1  Weight: 1.00
        40---->41    Progress:  0 0  Cost:   1  Weight: 1.00
        41---->39    Progress:  1 0  Cost:   1  Weight: -99.00
        39---->42    Progress:  1-1  Cost:   1  Weight: -97.50

  Total Weight: -193.00
Total Progress:    2  0  Total Cost:    7


Cycle number     7
Discovered after iteration  1


When lambda equals    100.00     1.50

         5---->24    Progress:  0 1  Cost:   1  Weight: -0.50
        24---->25    Progress:  0 0  Cost:   1  Weight: 1.00
        25----> 5    Progress:  1 0  Cost:   1  Weight: -99.00

  Total Weight: -98.50
Total Progress:    1  1  Total Cost:    3


New Basis:    1  7    Solution Cost: 100.00
New Basis:    6  7    Solution Cost:  3.50
Basic Solution: x = [  0.50,  0.00]
Dual Solution: lambda = [  3.50, -0.50]


Cycle number     8
Discovered after iteration  1


When lambda equals      3.50   -0.50

        42---->44    Progress:  0 0  Cost:   1  Weight: 1.00
```

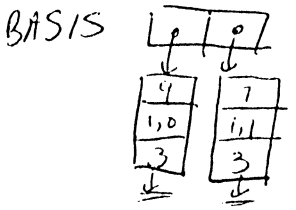


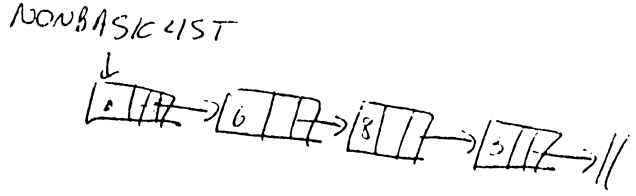




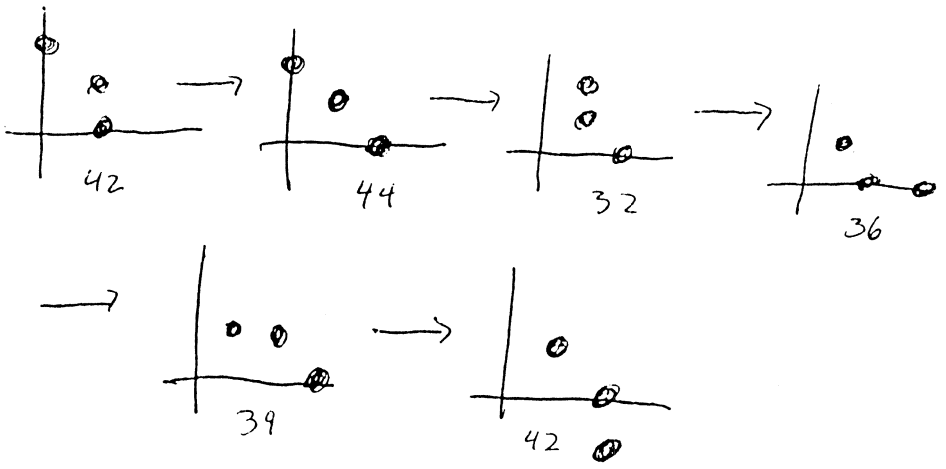
```
        44---->39   Progress:  0 0  Cost:  1  Weight: 1.00
        39---->42   Progress:  1-1  Cost:  1  Weight: -3.00

  Total Weight: -1.00
Total Progress:   1 -1  Total Cost:   3



Cycle number    9
Discovered after iteration  1


When lambda equals      3.50   -0.50

         5---->21   Progress:  0 0  Cost:  1  Weight: 1.00
        21---->25   Progress:  0 0  Cost:  1  Weight: 1.00
        25----> 5   Progress:  1 0  Cost:  1  Weight: -2.50

  Total Weight: -0.50
Total Progress:   1  0  Total Cost:   3


New Basis:   9  7   Solution Cost:  3.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [  3.00,  0.00]


Cycle number   10
Discovered after iteration  1


When lambda equals      3.00    0.00

        42---->44   Progress:  0 0  Cost:  1  Weight: 1.00
        44---->32   Progress:  1 0  Cost:  1  Weight: -2.00
        32---->36   Progress:  0 0  Cost:  1  Weight: 1.00
        36---->39   Progress:  0 0  Cost:  1  Weight: 1.00
        39---->42   Progress:  1-1  Cost:  1  Weight: -2.00

  Total Weight: -1.00
Total Progress:   2 -1  Total Cost:   5


New Basis:  10  7   Solution Cost:  2.67
Basic Solution: x = [  0.33,  0.33]
Dual Solution: lambda = [  2.67,  0.33]


Cycle number   11
Discovered after iteration  2


When lambda equals      2.67    0.33

        29---->31   Progress:  0 0  Cost:  1  Weight: 1.00
        31---->20   Progress:  1 0  Cost:  1  Weight: -1.67
        20---->29   Progress:  1 0  Cost:  1  Weight: -1.67

  Total Weight: -2.33
Total Progress:   2  0  Total Cost:   3
```

8





9



10

```
Cycle number   12
Discovered after iteration  2


When lambda equals      2.67    0.33

         4---->24   Progress:  0 0  Cost:  1  Weight: 1.00
        24----> 4   Progress:  1 0  Cost:  1  Weight: -1.67

  Total Weight: -0.67
Total Progress:   1  0  Total Cost:   2


New Basis:  12  7   Solution Cost:  2.00
New Basis:  11  7   Solution Cost:  1.50
Basic Solution: x = [  0.50,  0.00]
Dual Solution: lambda = [  1.50,  1.50]


When lambda equals   1.50  1.50

No negative cycles detected; Termination after  4 iterations.


The above solution is optimal.

**********************************************
```

11

```
New Right-hand side:          2         1
Initial artificial basis:
Cycle number    1 Progress:    1   0     Cost:   100
Cycle number    3 Progress:    0   1     Cost:   100
New Basis:   1  7   Solution Cost: 103.00
New Basis:  11  7   Solution Cost:  4.50
Basic Solution: x = [  0.50,  1.00]
Dual Solution: lambda = [  1.50,  1.50]


When lambda equals   1.50  1.50

No negative cycles detected; Termination after  4 iterations.


The above solution is optimal.

**********************************************
```

12

```
New Right-hand side:          1         1
Initial artificial basis:
Cycle number    1 Progress:    1   0     Cost:   100
Cycle number    3 Progress:    0   1     Cost:   100
New Basis:   7  3   Solution Cost:  3.00
New Basis:   7  5   Solution Cost:  3.00
Basic Solution: x = [  1.00,  0.00]
```

```
Dual Solution: lambda = [  1.50,  1.50]

When lambda equals   1.50  1.50

No negative cycles detected; Termination after  4 iterations.

The above solution is optimal.

*********************************************
```

(13)

```
New Right-hand side:          1         2
Initial artificial basis:
Cycle number     1 Progress:     1   0      Cost:   100
Cycle number     3 Progress:     0   1      Cost:   100
New Basis:    1  5    Solution Cost: 103.00
New Basis:    7  5    Solution Cost:  4.50
Basic Solution: x = [  1.00,  0.50]
Dual Solution: lambda = [  1.50,  1.50]

When lambda equals   1.50  1.50

No negative cycles detected; Termination after  4 iterations.

The above solution is optimal.

*********************************************
```

(14)

```
New Right-hand side:          0         1
Initial artificial basis:
Cycle number     1 Progress:     1   0      Cost:   100
Cycle number     3 Progress:     0   1      Cost:   100
New Basis:    1  5    Solution Cost:  1.50
New Basis:    7  5    Solution Cost:  1.50
Basic Solution: x = [  0.00,  0.50]
Dual Solution: lambda = [  1.50,  1.50]

When lambda equals   1.50  1.50

No negative cycles detected; Termination after  4 iterations.

The above solution is optimal.

*********************************************



New Right-hand side:         -1         1
Initial artificial basis:
Cycle number     2 Progress:    -1   0      Cost:   100
Cycle number     3 Progress:     0   1      Cost:   100
```

```
New Basis:   2  5   Solution Cost: 101.50
Basic Solution: x = [  1.00,  0.50]
Dual Solution: lambda = [-100.00,  1.50]


Cycle number   13
Discovered after iteration  2


When lambda equals  -100.00    1.50

        19---->16   Progress:  0 0  Cost:  1  Weight: 1.00
        16---->19   Progress: -1 0  Cost:  1  Weight: -99.00

  Total Weight: -98.00
Total Progress:  -1  0  Total Cost:   2



Cycle number   14
Discovered after iteration  2


When lambda equals  -100.00    1.50

        43---->12   Progress:  0 0  Cost:  1  Weight: 1.00
        12---->43   Progress: -1 0  Cost:  1  Weight: -99.00

  Total Weight: -98.00
Total Progress:  -1  0  Total Cost:   2



Cycle number   15
Discovered after iteration  2


When lambda equals  -100.00    1.50

        39---->44   Progress:  0 0  Cost:  1  Weight: 1.00
        44---->42   Progress:  0 0  Cost:  1  Weight: 1.00
        42---->39   Progress: -1 1  Cost:  1  Weight: -100.50

  Total Weight: -98.50
Total Progress:  -1  1  Total Cost:   3


New Basis:  15  5   Solution Cost:  3.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [ -1.50,  1.50]


When lambda equals  -1.50  1.50

No negative cycles detected; Termination after  6 iterations.


The above solution is optimal.

**********************************************
```

15

16

17

18

```
New Right-hand side:          -1          0
Initial artificial basis:
Cycle number     2 Progress:    -1    0      Cost:    100
Cycle number     3 Progress:     0    1      Cost:    100
New Basis:    2  5    Solution Cost: 100.00
New Basis:    2 15    Solution Cost: 100.00
New Basis:   14 15    Solution Cost:   2.00
Basic Solution: x = [   1.00,   0.00]
Dual Solution: lambda = [ -2.00,   1.00]


Cycle number   16
Discovered after iteration  4


When lambda equals     -2.00     1.00

        20---->31    Progress: -1 0  Cost:   1  Weight: -1.00
        31---->29    Progress:  0 0  Cost:   1  Weight:  1.00
        29---->20    Progress: -1 0  Cost:   1  Weight: -1.00

  Total Weight: -1.00
Total Progress:  -2  0  Total Cost:    3


New Basis:  16 15    Solution Cost:   1.50
Basic Solution: x = [   0.50,   0.00]
Dual Solution: lambda = [ -1.50,   1.50]


When lambda equals  -1.50  1.50

No negative cycles detected; Termination after  6 iterations.


The above solution is optimal.

*********************************************
```

19

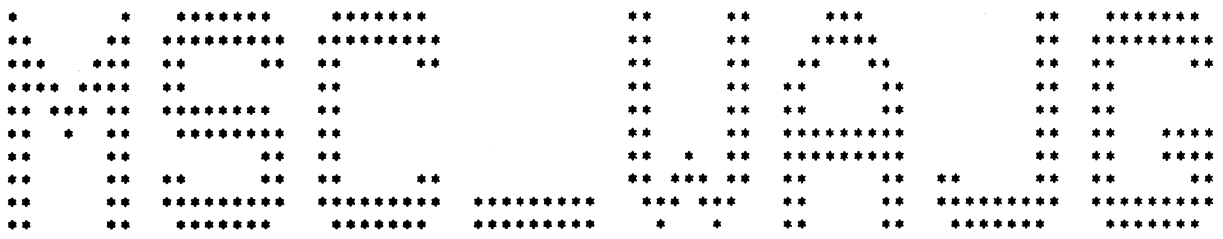
```
New Right-hand side:          -1         -1
Initial artificial basis:
Cycle number     2 Progress:    -1    0      Cost:    100
Cycle number     4 Progress:     0   -1      Cost:    100
New Basis:   16  4    Solution Cost: 101.50
New Basis:   16 10    Solution Cost:   9.50
New Basis:   16  8    Solution Cost:   6.00
Basic Solution: x = [   1.00,   1.00]
Dual Solution: lambda = [ -1.50, -4.50]


Cycle number   17
Discovered after iteration  1


When lambda equals     -1.50    -4.50

         2----> 9    Progress:  0-1  Cost:   1  Weight: -3.50
```

20

```
         9----> 8   Progress:  0 0  Cost:  1  Weight: 1.00
         8----> 2   Progress:  0-1  Cost:  1  Weight: -3.50

  Total Weight: -6.00
Total Progress:   0 -2  Total Cost:   3


New Basis:  16 17   Solution Cost:  3.00
Basic Solution: x = [  0.50,  0.50]
Dual Solution: lambda = [ -1.50, -1.50]


When lambda equals  -1.50 -1.50

No negative cycles detected; Termination after  6 iterations.


The above solution is optimal.

*********************************************




New Right-hand side:          0        -1
Initial artificial basis:
Cycle number     1 Progress:    1   0     Cost:   100
Cycle number     4 Progress:    0  -1     Cost:   100
New Basis:   1 17   Solution Cost:  1.50
New Basis:   8 17   Solution Cost:  1.50
Basic Solution: x = [  0.00,  0.50]
Dual Solution: lambda = [  1.50, -1.50]


When lambda equals   1.50 -1.50

No negative cycles detected; Termination after  4 iterations.


The above solution is optimal.

*********************************************




New Right-hand side:          2        -1
Initial artificial basis:
Cycle number     1 Progress:    1   0     Cost:   100
Cycle number     4 Progress:    0  -1     Cost:   100
New Basis:   1 17   Solution Cost: 201.50
New Basis:   1  8   Solution Cost: 103.00
New Basis:  10  8   Solution Cost:  5.00
New Basis:  11  8   Solution Cost:  4.50
Basic Solution: x = [  0.50,  1.00]
Dual Solution: lambda = [  1.50, -1.50]


When lambda equals   1.50 -1.50

No negative cycles detected; Termination after  4 iterations.
```

The above solution is optimal.

21 22



FILE: RANDCG OUT

TIME: 02/27/89 09:14:31

SYSTEM: JHUVMS

PRINTER: RM_LINE

MODE: STANDARD

Adjacency List for Data from randcg.dat

```
 1:------>42------>35------>21------> 8------> 2------>32------>11------> 8
   (-1,0)7   (-1,0)1   (-1,0)1   (0,-1)1   (0,-1)4   (0,0)1   (0,0)9   (0,0)8

------> 5------> 2
   (0,0)6   (0,0)7

 2:------>22------> 9------> 8------> 1------>33------> 8------> 1
   (-1,0)4   (0,-1)8   (0,1)8   (0,1)4   (0,0)5   (0,0)6   (0,0)7

 3:------>36------>32------>23------>16------>11------> 4------>36------>32
   (-1,0)6   (0,-1)6   (-1,0)8   (-1,0)5   (0,-1)2   (0,-1)3   (0,0)6   (0,0)8

------> 6------> 4------>16------>10
   (0,0)8   (0,0)7   (0,0)7   (0,0)5

 4:------>37------>34------>24------>12------> 5------> 3------>34------> 7
   (-1,0)6   (-1,0)9   (-1,0)5   (0,-1)2   (0,-1)2   (0,1)3   (0,0)3   (0,0)8

------> 5------>24------>11------> 3
   (0,0)5   (0,0)3   (0,0)9   (0,0)7

 5:------>25------>13------>24------> 4------>35------>21------>12------> 4
   (-1,0)5   (0,-1)8   (0,1)5   (0,1)2   (0,0)8   (0,0)3   (0,0)1   (0,0)5

------> 1
   (0,0)6

 6:------>38------> 7------> 7------>36------>23------>14------> 3
   (-1,0)5   (0,-1)7   (0,0)4   (0,0)3   (0,0)2   (0,0)8   (0,0)8

 7:------>26------>15------> 6------>37------>24------> 6------> 4
   (-1,0)2   (0,-1)7   (0,1)7   (0,0)4   (0,0)4   (0,0)4   (0,0)8

 8:------>46------> 1------>13------> 9------> 2------> 2------> 1
   (-1,0)4   (0,1)1   (0,0)3   (0,0)1   (0,-1)8   (0,0)6   (0,0)8

 9:------> 2------> 8
   (0,1)8   (0,0)1

10:------>33------>17------>17------>32------>14------>11------> 3
   (0,-1)8   (-1,0)9   (0,0)7   (0,0)9   (0,0)5   (0,0)7   (0,0)5

11:------> 3------>42------>12------>21------>10------> 4------> 1
   (0,1)2   (0,0)8   (0,0)2   (0,0)8   (0,0)7   (0,0)9   (0,0)9

12:------>43------> 4------>43------>15------>13------>11------> 5
   (-1,0)2   (0,1)2   (0,0)6   (0,0)5   (0,0)2   (0,0)2   (0,0)1

13:------> 5------>12------> 8
   (0,1)8   (0,0)2   (0,0)3

14:------>10------> 6
   (0,0)5   (0,0)8

15:------> 7------>12
   (0,1)7   (0,0)5

16:------>35------>34------>27------>21------>19------> 3------>34------>23
   (0,-1)7   (0,-1)2   (-1,0)1   (0,-1)3   (-1,0)1   (1,0)5   (0,0)7   (0,0)6

------>21------>19------>17------> 3
   (0,0)6   (0,0)6   (0,0)1   (0,0)7

17:------>28------>22------>10------>35------>21------>16------>10
```

```
   (-1,0)3   (0,-1)1   (1,0)9   (0,0)8   (0,0)6   (0,0)1   (0,0)7

18:------>39------>37------>29------>24------>20------>36------>29------>23
   (0,-1)6   (0,-1)4   (-1,0)9   (0,0)3   (-1,0)2   (0,0)7   (0,0)4   (0,0)8

------>20------>19
   (0,0)4   (0,0)3

19:------>30------>25------>21------>16------>37------>27------>24------>18
   (-1,0)6   (0,-1)6   (1,0)2   (1,0)1   (0,0)1   (0,0)7   (0,0)2   (0,0)3

------>16
   (0,0)6

20:------>31------>26------>29------>18------>29------>18
   (-1,0)7   (0,-1)9   (1,0)2   (1,0)2   (0,0)8   (0,0)4

21:------> 1------>16------>25------>22------>19------>17------>16------>11
   (1,0)1   (0,1)3   (0,0)2   (0,0)9   (-1,0)2   (0,0)6   (0,0)6   (0,0)8

------> 5
   (0,0)3

22:------> 2------>17------>21
   (1,0)4   (0,1)1   (0,0)9

23:------> 3------>39------>24------>27------>18------>16------> 6
   (1,0)8   (0,0)1   (0,0)4   (0,0)6   (0,0)8   (0,0)6   (0,0)2

24:------> 4------>18------>26------>25------>23------>19------> 7------> 5
   (1,0)5   (0,0)3   (0,0)8   (0,0)3   (0,0)4   (0,0)2   (0,0)4   (0,-1)5

------> 4
   (0,0)3

25:------> 5------>19------>24------>21
   (1,0)5   (0,1)6   (0,0)3   (0,0)2

26:------> 7------>20------>24
   (1,0)2   (0,1)9   (0,0)8

27:------>40------>16------>40------>30------>28------>23------>19
   (0,-1)8   (1,0)1   (0,0)2   (0,0)2   (0,0)4   (0,0)6   (0,0)7

28:------>17------>27
   (1,0)3   (0,0)4

29:------>41------>18------>31------>30------>20------>20------>18
   (0,-1)4   (1,0)9   (0,0)3   (0,0)4   (-1,0)2   (0,0)8   (0,0)4

30:------>19------>29------>27
   (1,0)6   (0,0)4   (0,0)2

31:------>20------>29
   (1,0)7   (0,0)3

32:------>44------> 3------>36------>42------>35------>33------>10------> 3
   (-1,0)4   (0,1)6   (0,0)3   (0,0)3   (0,0)2   (0,0)7   (0,0)9   (0,0)8

------> 1
   (0,0)1

33:------>10------>32------> 2
   (0,1)8   (0,0)7   (0,0)5

34:------>43------>42------>40------>39------> 4------>16------>39------>42
```

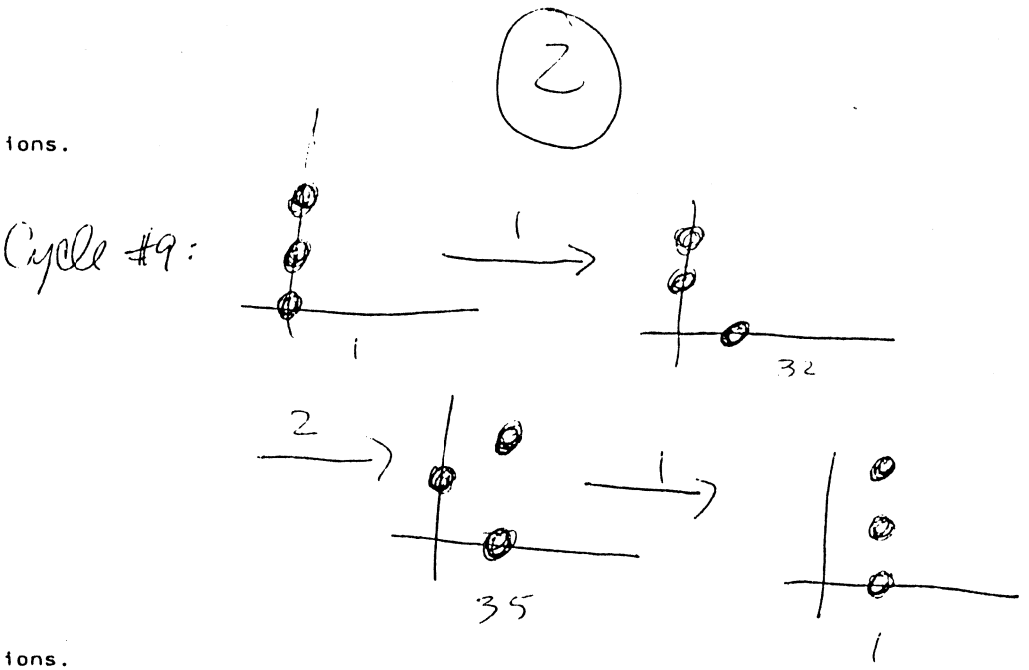
```
   (0,-1)3   (0,-1)1   (-1,0)4   (-1,0)8   (1,0)9   (0,1)2   (0,0)2   (0,0)8

------>37------>35------>16------> 4
   (0,0)4   (0,0)9   (0,0)7   (0,0)3

35:------> 1------>16------>43------>34------>32------>17------> 5
   (1,0)1   (0,1)7   (0,0)7   (0,0)9   (0,0)2   (0,0)8   (0,0)8

36:------>44------> 3------>39------>38------>37------>32------>18------> 6
   (0,-1)1   (1,0)6   (0,0)2   (0,0)3   (0,0)2   (0,0)3   (0,0)7   (0,0)3

------> 3
   (0,0)6

37:------> 4------>18------>40------>36------>34------>19------> 7
   (1,0)6   (0,1)4   (0,0)4   (0,0)2   (0,0)4   (0,0)1   (0,0)4

38:------> 6------>36
   (1,0)5   (0,0)3

39:------>41------>34------>18------>44------>40------>42------>36------>34
   (-1,0)4   (1,0)8   (0,1)6   (0,0)9   (0,0)4   (1,-1)7   (0,0)2   (0,0)2

------>23
   (0,0)1

40:------>45------>34------>27------>41------>39------>37------>27
   (0,-1)9   (1,0)4   (0,1)8   (0,0)5   (0,0)4   (0,0)4   (0,0)2

41:------>39------>29------>40
   (1,0)4   (0,1)4   (0,0)5

42:------>46------> 1------>34------>44------>43------>39------>34------>32
   (0,-1)2   (1,0)7   (0,1)1   (0,0)8   (0,0)4   (-1,1)7   (0,0)8   (0,0)3

------>11
   (0,0)8

43:------>45------>12------>34------>46------>42------>35------>12
   (-1,0)8   (1,0)2   (0,1)3   (0,0)1   (0,0)4   (0,0)7   (0,0)6

44:------>32------>36------>42------>39
   (1,0)4   (0,1)1   (0,0)8   (0,0)9

45:------>43------>40
   (1,0)8   (0,1)9

46:------>42------> 8------>43
   (0,1)2   (1,0)4   (0,0)1
```

```
Original RHS vector:          1          0
Initial artificial basis:
Cycle number     1 Progress:     1    0      Cost:    100
Cycle number     3 Progress:     0    1      Cost:    100
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [100.00,100.00]


Cycle number     5
Discovered after iteration  1
```

```
When lambda equals   100.00  100.00

        46---->42   Progress:  0 1  Cost:  2  Weight: -98.00
        42---->43   Progress:  0 0  Cost:  4  Weight: 4.00
        43---->46   Progress:  0 0  Cost:  1  Weight: 1.00

  Total Weight: -93.00
Total Progress:   0  1  Total Cost:   7



Cycle number    6
Discovered after iteration  1


When lambda equals   100.00  100.00

        24---->25   Progress:  0 0  Cost:  3  Weight: 3.00
        25----> 5   Progress:  1 0  Cost:  5  Weight: -95.00
         5---->24   Progress:  0 1  Cost:  5  Weight: -95.00

  Total Weight: -187.00
Total Progress:   1  1  Total Cost:  13


New Basis:   1  6   Solution Cost: 100.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [100.00,-87.00]


Cycle number    7
Discovered after iteration  1


When lambda equals   100.00  -87.00

        42---->44   Progress:  0 0  Cost:  8  Weight: 8.00
        44---->39   Progress:  0 0  Cost:  9  Weight: 9.00
        39---->42   Progress:  1-1  Cost:  7  Weight: -180.00

  Total Weight: -163.00
Total Progress:   1 -1  Total Cost:  24



Cycle number    8
Discovered after iteration  1


When lambda equals   100.00  -87.00

         5---->21   Progress:  0 0  Cost:  3  Weight: 3.00
        21---->25   Progress:  0 0  Cost:  2  Weight: 2.00
        25----> 5   Progress:  1 0  Cost:  5  Weight: -95.00

  Total Weight: -90.00
Total Progress:   1  0  Total Cost:  10
```

```
New Basis:   8  6   Solution Cost: 10.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [ 10.00,  3.00]


Cycle number    9
Discovered after iteration  1


When lambda equals    10.00    3.00

         1---->32   Progress:  0 0  Cost:  1  Weight: 1.00
        32---->35   Progress:  0 0  Cost:  2  Weight: 2.00
        35----> 1   Progress:  1 0  Cost:  1  Weight: -9.00

  Total Weight: -6.00
Total Progress:   1  0  Total Cost:   4


New Basis:   9  6   Solution Cost:  4.00
New Basis:   9  5   Solution Cost:  4.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [  4.00,  7.00]


Cycle number   10
Discovered after iteration  3


When lambda equals     4.00    7.00

        21---->25   Progress:  0 0  Cost:  2  Weight: 2.00
        25---->19   Progress:  0 1  Cost:  6  Weight: -1.00
        19---->21   Progress:  1 0  Cost:  2  Weight: -2.00

  Total Weight: -1.00
Total Progress:   1  1  Total Cost:  10


New Basis:   9 10   Solution Cost:  4.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [  4.00,  6.00]


Cycle number   11
Discovered after iteration  3


When lambda equals     4.00    6.00

         1---->32   Progress:  0 0  Cost:  1  Weight: 1.00
        32---->42   Progress:  0 0  Cost:  3  Weight: 3.00
        42---->34   Progress:  0 1  Cost:  1  Weight: -5.00
        34---->16   Progress:  0 1  Cost:  2  Weight: -4.00
        16---->19   Progress: -1 0  Cost:  1  Weight: 5.00
        19---->21   Progress:  1 0  Cost:  2  Weight: -2.00
        21----> 1   Progress:  1 0  Cost:  1  Weight: -3.00

  Total Weight: -5.00
Total Progress:   1  2  Total Cost:  11


New Basis:   9 11   Solution Cost:  4.00
```

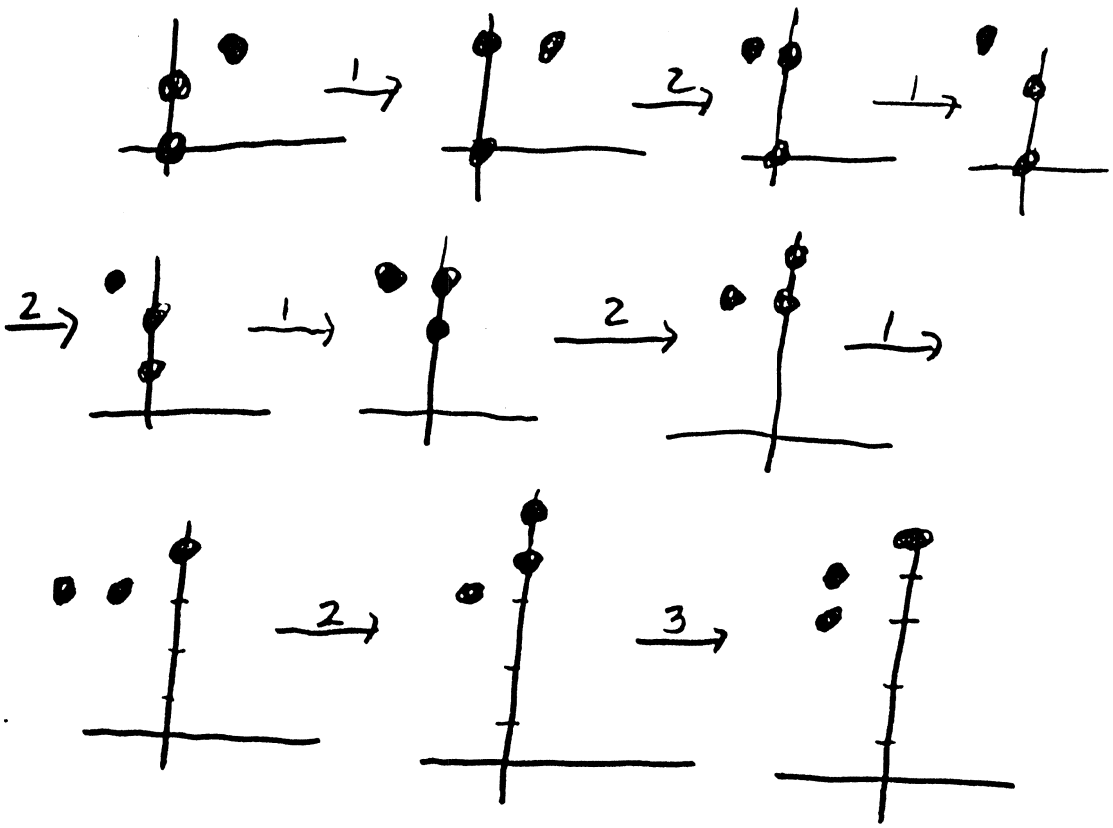
```
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [  4.00,  3.50]


When lambda equals   4.00  3.50

No negative cycles detected; Termination after  5 iterations.


The above solution is optimal.

*********************************************
```




```
New Right-hand side:          1          1
Initial artificial basis:
Cycle number     1 Progress:     1   0      Cost:    100
Cycle number     3 Progress:     0   1      Cost:    100
New Basis:    1 11    Solution Cost: 55.50
New Basis:    9 11    Solution Cost:  7.50
Basic Solution: x = [  0.50,  0.50]
Dual Solution: lambda = [  4.00,  3.50]


When lambda equals   4.00  3.50

No negative cycles detected; Termination after  5 iterations.


The above solution is optimal.

*********************************************
```


```
New Right-hand side:          1          2
Initial artificial basis:
Cycle number     1 Progress:     1   0      Cost:    100
Cycle number     3 Progress:     0   1      Cost:    100
New Basis:   11  3    Solution Cost: 11.00
New Basis:   11  5    Solution Cost: 11.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [ -3.00,  7.00]


Cycle number   12
Discovered after iteration  2


When lambda equals    -3.00    7.00

        21---->16   Progress:  0 1  Cost:  3  Weight: -4.00
        16---->19   Progress: -1 0  Cost:  1  Weight: -2.00
        19---->21   Progress:  1 0  Cost:  2  Weight: 5.00

  Total Weight: -1.00
Total Progress:   0  1  Total Cost:   6


New Basis:   11 12    Solution Cost: 11.00
```

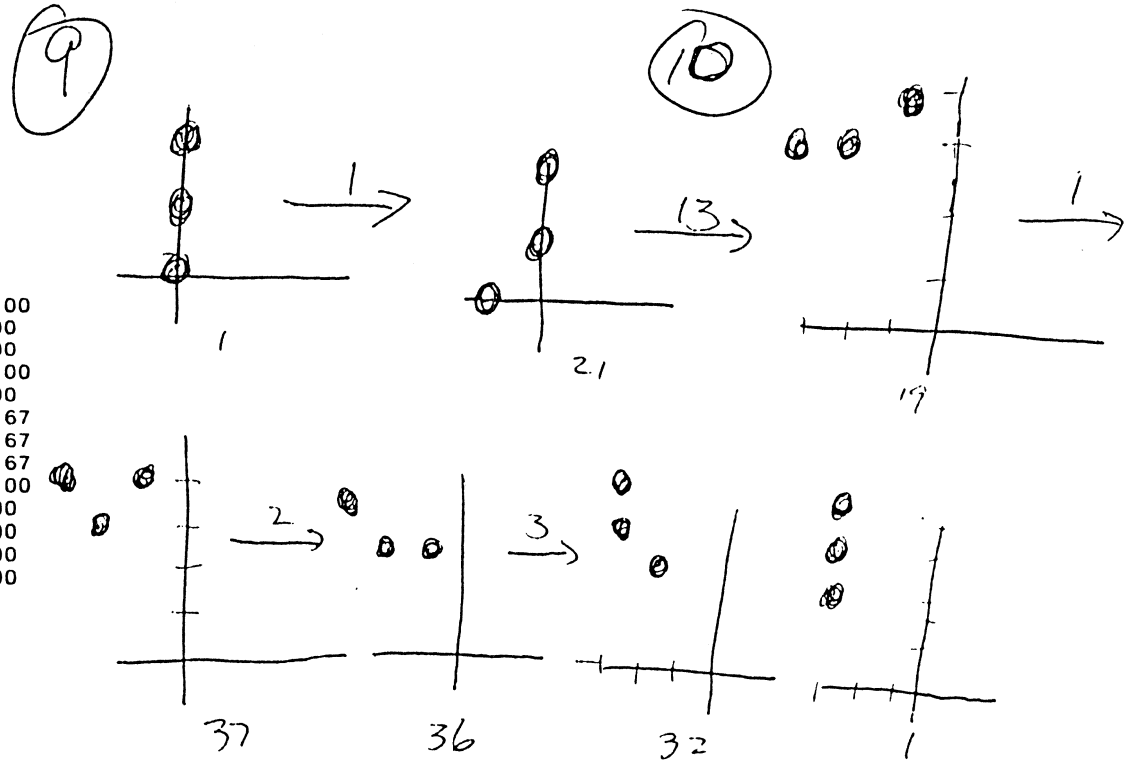
```
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [ -1.00,  6.00]


Cycle number   13
Discovered after iteration  3


When lambda equals    -1.00    6.00

         5---->12   Progress:  0 0  Cost:  1  Weight: 1.00
        12---->43   Progress: -1 0  Cost:  2  Weight: 1.00
        43---->46   Progress:  0 0  Cost:  1  Weight: 1.00
        46---->42   Progress:  0 1  Cost:  2  Weight: -4.00
        42---->34   Progress:  0 1  Cost:  1  Weight: -5.00
        34---->16   Progress:  0 1  Cost:  2  Weight: -4.00
        16---->19   Progress: -1 0  Cost:  1  Weight: 0.00
        19---->21   Progress:  1 0  Cost:  2  Weight: 3.00
        21---->  5  Progress:  0 0  Cost:  3  Weight: 3.00

  Total Weight: -4.00
Total Progress:  -1  3  Total Cost:  15


New Basis:  11 13   Solution Cost: 11.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [  0.60,  5.20]

When lambda equals   0.60  5.20

No negative cycles detected; Termination after  7 iterations.

The above solution is optimal.

*********************************************
```




```
New Right-hand side:          -1          3
Initial artificial basis:
Cycle number     2 Progress:    -1    0      Cost:    100
Cycle number     3 Progress:     0    1      Cost:    100
New Basis:  13  3   Solution Cost: 15.00
New Basis:  13 11   Solution Cost: 15.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [  0.60,  5.20]


When lambda equals   0.60  5.20

No negative cycles detected; Termination after  7 iterations.


The above solution is optimal.

*********************************************
```

```
New Right-hand side:          -1          1
Initial artificial basis:
Cycle number     2 Progress:    -1    0      Cost:    100
Cycle number     3 Progress:     0    1      Cost:    100
New Basis:    2 11    Solution Cost: 155.50
New Basis:    2 13    Solution Cost: 71.67
Basic Solution: x = [  0.67,  0.33]
Dual Solution: lambda = [-100.00,-28.33]


Cycle number   14
Discovered after iteration  1


When lambda equals  -100.00  -28.33

         8----> 2   Progress:  0-1  Cost:  8  Weight: -20.33
         2----> 9   Progress:  0-1  Cost:  8  Weight: -20.33
         9----> 8   Progress:  0 0  Cost:  1  Weight: 1.00

  Total Weight: -39.67
Total Progress:   0 -2  Total Cost:  17


New Basis:  14 13    Solution Cost: 32.00
Basic Solution: x = [  1.00,  1.00]
Dual Solution: lambda = [-40.50, -8.50]


Cycle number   15
Discovered after iteration  2


When lambda equals   -40.50   -8.50

        43---->12   Progress:  0 0  Cost:  6  Weight: 6.00
        12---->43   Progress: -1 0  Cost:  2  Weight: -38.50

  Total Weight: -32.50
Total Progress:  -1  0  Total Cost:   8




Cycle number   16
Discovered after iteration  2


When lambda equals   -40.50   -8.50

        25---->21   Progress:  0 0  Cost:  2  Weight: 2.00
        21---->19   Progress: -1 0  Cost:  2  Weight: -38.50
        19---->25   Progress:  0-1  Cost:  6  Weight: -2.50

  Total Weight: -39.00
Total Progress:  -1 -1  Total Cost:  10


New Basis:  16 13    Solution Cost: 12.50
New Basis:  15 13    Solution Cost: 10.33
Basic Solution: x = [  0.67,  0.33]
Dual Solution: lambda = [ -8.00,  2.33]
```

```
Cycle number   17
Discovered after iteration  2


When lambda equals     -8.00     2.33

         1---->35   Progress: -1 0  Cost:   1  Weight: -7.00
        35---->32   Progress:  0 0  Cost:   2  Weight: 2.00
        32----> 1   Progress:  0 0  Cost:   1  Weight: 1.00

  Total Weight: -4.00
Total Progress:  -1  0  Total Cost:   4



Cycle number   18
Discovered after iteration  2


When lambda equals     -8.00     2.33

        19---->16   Progress:  0 0  Cost:   6  Weight: 6.00
        16---->19   Progress: -1 0  Cost:   1  Weight: -7.00

  Total Weight: -1.00
Total Progress:  -1  0  Total Cost:   7


New Basis:  18 13   Solution Cost:  9.67
New Basis:  17 13   Solution Cost:  7.67
Basic Solution: x = [  0.67,  0.33]
Dual Solution: lambda = [ -4.00,  3.67]


Cycle number   19
Discovered after iteration  7


When lambda equals     -4.00     3.67

         1---->21   Progress: -1 0  Cost:   1  Weight: -3.00
        21----> 5   Progress:  0 0  Cost:   3  Weight: 3.00
         5---->12   Progress:  0 0  Cost:   1  Weight: 1.00
        12---->43   Progress: -1 0  Cost:   2  Weight: -2.00
        43---->46   Progress:  0 0  Cost:   1  Weight: 1.00
        46---->42   Progress:  0 1  Cost:   2  Weight: -1.67
        42---->34   Progress:  0 1  Cost:   1  Weight: -2.67
        34---->16   Progress:  0 1  Cost:   2  Weight: -1.67
        16---->19   Progress: -1 0  Cost:   1  Weight: -3.00
        19---->37   Progress:  0 0  Cost:   1  Weight: 1.00
        37---->36   Progress:  0 0  Cost:   2  Weight: 2.00
        36---->32   Progress:  0 0  Cost:   3  Weight: 3.00
        32----> 1   Progress:  0 0  Cost:   1  Weight: 1.00

  Total Weight: -2.00
Total Progress:  -3  3  Total Cost:  21


New Basis:  19 13   Solution Cost:  7.00
Basic Solution: x = [  0.33,  0.00]
Dual Solution: lambda = [ -3.00,  4.00]
```



```
When lambda equals  -3.00  4.00

No negative cycles detected; Termination after  8 iterations.

The above solution is optimal.

**********************************************
```

11

```
New Right-hand side:          -1          1
Initial artificial basis:
Cycle number     2 Progress:    -1    0      Cost:    100
Cycle number     3 Progress:     0    1      Cost:    100
New Basis:    2 13    Solution Cost: 71.67
New Basis:   19 13    Solution Cost:  7.00
Basic Solution: x = [  0.33,  0.00]
Dual Solution: lambda = [ -3.00,  4.00]

When lambda equals  -3.00  4.00

No negative cycles detected; Termination after  8 iterations.

The above solution is optimal.

**********************************************
```

12

13

```
New Right-hand side:          -1          0
Initial artificial basis:
Cycle number     2 Progress:    -1    0      Cost:    100
Cycle number     3 Progress:     0    1      Cost:    100
New Basis:    2 13    Solution Cost: 100.00
New Basis:    2 19    Solution Cost: 100.00
New Basis:   17 19    Solution Cost:  4.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [ -4.00,  3.00]

When lambda equals  -4.00  3.00

No negative cycles detected; Termination after  8 iterations.

The above solution is optimal.

**********************************************
```

14

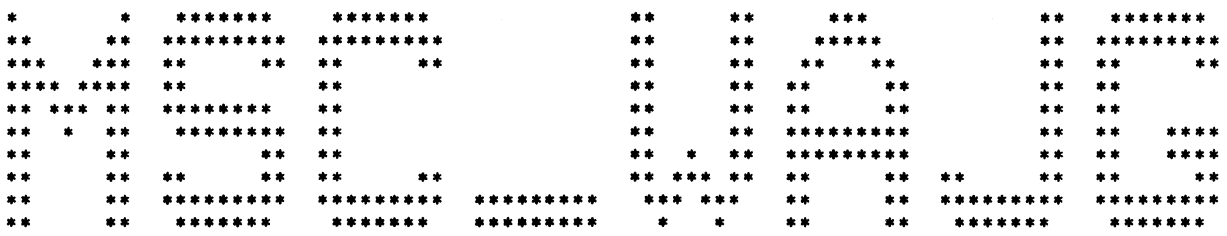
```
New Right-hand side:          -1         -1
Initial artificial basis:
Cycle number     2 Progress:    -1    0      Cost:    100
Cycle number     4 Progress:     0   -1      Cost:    100
```

```
New Basis:  17  4   Solution Cost: 104.00
New Basis:  16  4   Solution Cost: 10.00
New Basis:  16 14   Solution Cost: 10.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [ -1.50, -8.50]


Cycle number   20
Discovered after iteration  2


When lambda equals    -1.50   -8.50

         1---->21   Progress: -1 0  Cost:  1  Weight: -0.50
        21---->19   Progress: -1 0  Cost:  2  Weight: 0.50
        19---->16   Progress:  1 0  Cost:  1  Weight: 2.50
        16---->34   Progress:  0-1  Cost:  2  Weight: -6.50
        34---->42   Progress:  0-1  Cost:  1  Weight: -7.50
        42---->46   Progress:  0-1  Cost:  2  Weight: -6.50
        46---> 8    Progress:  1 0  Cost:  4  Weight: 5.50
         8---> 2    Progress:  0-1  Cost:  8  Weight: -0.50
         2---> 1    Progress:  0 0  Cost:  7  Weight: 7.00

  Total Weight: -6.00
Total Progress:   0 -4  Total Cost:  28



Cycle number   21
Discovered after iteration  2


When lambda equals    -1.50   -8.50

         4---> 5    Progress:  0-1  Cost:  2  Weight: -6.50
         5---> 4    Progress:  0 0  Cost:  5  Weight: 5.00

  Total Weight: -1.50
Total Progress:   0 -1  Total Cost:   7


New Basis:  16 21   Solution Cost: 10.00
Basic Solution: x = [  1.00,  0.00]
Dual Solution: lambda = [ -3.00, -7.00]


Cycle number   22
Discovered after iteration  3


When lambda equals    -3.00   -7.00

        16---->21   Progress:  0-1  Cost:  3  Weight: -4.00
        21---->19   Progress: -1 0  Cost:  2  Weight: -1.00
        19---->16   Progress:  1 0  Cost:  1  Weight: 4.00

  Total Weight: -1.00
Total Progress:   0 -1  Total Cost:   6


New Basis:  16 22   Solution Cost: 10.00
Basic Solution: x = [  1.00,  0.00]
```

```
Dual Solution: lambda = [ -4.00, -6.00]


Cycle number   23
Discovered after iteration  3


When lambda equals     -4.00   -6.00

          1---->21    Progress: -1 0  Cost:   1  Weight: -3.00
         21---->19    Progress: -1 0  Cost:   2  Weight: -2.00
         19---->16    Progress:  1 0  Cost:   1  Weight: 5.00
         16---->34    Progress:  0-1  Cost:   2  Weight: -4.00
         34---->42    Progress:  0-1  Cost:   1  Weight: -5.00
         42---->32    Progress:  0 0  Cost:   3  Weight: 3.00
         32---->  1   Progress:  0 0  Cost:   1  Weight: 1.00

  Total Weight: -5.00
Total Progress:  -1 -2  Total Cost:  11


New Basis:  16 23   Solution Cost: 10.00
New Basis:  17 23   Solution Cost:  7.50
Basic Solution: x = [  0.50,  0.50]
Dual Solution: lambda = [ -4.00, -3.50]


When lambda equals  -4.00 -3.50

No negative cycles detected; Termination after  5 iterations.


The above solution is optimal.
```

15

16



FILE: COLGEN PAS

TIME: 04/18/89 14:50:33

SYSTEM: JHUVMS

PRINTER: RM_LINE

MODE: STANDARD

```
Program Colgen(Input,Output,Infile,Outfile);

(** This program takes a C-graph (read from file infile) and a RHS vector, and finds the optimal cycle combination.  When m=0,
it is any cycle with negative weight and the program NEGCYC.PAS would be a more suitable program.  When m=1, this is a cycle with
minimum cost/progress ratio.  When m > 1, this is a basic optimal solution to a linear program.  Given the dual vector lambda of
a feasible (perhaps artificial) solution to the LP, the procedure DETECTNEGCYC assigns (reduced cost) weights to the arcs of the
C-Graph, and generates (if they exist) negative cycles, which become new LP variables.  When m= 2 or 3, the program re-solves the
LP, producing a new lambda, and the process is repeated until no negative cycles are found (when m > 3, the user must do the
simplex computations externally, the program CGRAPH.PAS may be better suited for this purpose) at which point we have an optimal
solution.  Input instructions appear at  the end of the document **)

Const Nmax = 501;                                   (*max # of nodes*)
      Mmax = 3;                                     (*max # of dimensions*)
      MaxStanDirs = 20;                             (*max # of standard directions*)
      Epsilon = 0.0001;  (*10^-4*)                  (*used to avoid improving u based on roundoff errors*)

Type  Nodetype = 0..Nmax;
      mtype = 0..mmax;
      mvector = Array[1..mmax] of Integer;
      lambdatype= Array[1..mmax] of Real;
      nvector = Array[1..nmax] of Integer;
      realvector = Array[1..nmax] of Real;
      boolvector = Array[1..nmax] of Boolean;

      ptr = ^Arc;
      Arc = Record
              Tail: Nodetype;
              Head: Nodetype;
              Progress: mvector;
              Cost: Integer;
              Weight: Real;                         (* weight = cost - (lambda)^T (progress) *)
              Next: Ptr;                            (* next arc on adjacency list *)
              NextonCycle: Ptr;                     (* next arc on the negative cycle *)
            End;

      PtrArray = Array[1..Nmax] of ptr;

      colptr = ^column;                             (* represents columns in simplex calculations *)
      column = Record
                 Name: integer;
                 Progress: mvector;
                 Cost: integer;
                 Next: colptr;
               End;

      Basistype = Array[1..3] of Colptr;
      Bmatrix = Array[1..3,1..3] of Real;

      Filename = Packed Array[1..12] of Char;

Var   m : 0..mmax;                                  (* dimension of progress arcs *)
      N : Nodetype;                                 (* # of nodes in the C-graph *)
      lambda : lambdatype;                          (* dual variable vector      *)
      done: Boolean;
      A : PtrArray;                                 (* A[I] is a linked list of arcs leaving node I *)
      Answer: Char;
      Infile, Outfile : Text;
      Infilename, Outfilename: Filename;
      BigM, Infinity, Best, Cyclenumber: Integer;   (* Infinity values and cycle counters *)
      Newcyclelist : Colptr;                        (* List of new cycles detected to be used in simplex calculations *)
      NonbasicList:  Colptr;                        (* List of non-basic variables for simplex calculations *)
      Basis: Basistype;                             (* Array of Basic columns *)
      RHS:   mvector;                               (* RightHandSide columns *)
      Binverse: Bmatrix;                            (* Basis inverse *)
      x: lambdatype;                                (* Primal variable vector*)
      Unbounded: Boolean;                           (* Is LP unbounded?*)
```

```
      DegenerateCycling: Boolean;                    (* Does Degenerate Cycling occur in simplex? *)

(*************************************************************************)
Procedure Readinfile(Var Infilename: Filename);          (* Readinfile Reads the name of the input file *)

Begin
 Write('Enter the name of the input file: ');
 Readln(Infilename);
 Open(Infile, File_Name:= Infilename, History:= Old);
 Reset(Infile);
End;
(*************************************************************************)
Procedure Readoutfile(Var Outfilename: Filename);        (* Readoutfile reads the name of the output file *)

Begin
 Write('Enter the name of the output file: ');
 Readln(Outfilename);
 Open(Outfile, File_Name:= Outfilename);
 Rewrite(Outfile);
End;
(*************************************************************************)
Procedure GetData(Var m: mtype; Var N: Nodetype; Var A: PtrArray;       (* Getdata initializes variables and creates the *)
                  Var Done: Boolean; Var Lambda: Lambdatype;            (* adjacency lists from the input file.          *)
                  Var Cyclenumber, Infinity, BigM: Integer);

Var   Answer: Char;
      Reversible, Unitcosts, Bignumbers : Boolean;
      I, StanDirs, DirNumber: 0..MaxStanDirs;            (* StanDirs is the number of standard directions *)
      J: 0..mmax;
      D: Array[1..MaxStanDirs] of mvector;               (* D[I] is the Ith standard direction *)
      K, Tail, Head: Nodetype;
      P,Q: ptr;

Begin
 Done:= False;
 For J:= 1 to m Do
  Lambda[J]:= 0;
 Readln(Infile, m);                                      (* read m *)
  Cyclenumber:= 2*m;                                     (* Initialize cyclenumber *)
 Readln(Infile, N);                                      (* read n *)
 Readln(Infile, Answer);
 Reversible:= ((Answer = 'R') or (Answer = 'r'));        (* Is the graph reversible ? *)
 Readln(Infile, Answer);
 Unitcosts:= ((Answer = 'U') or (Answer = 'u'));         (* Do all arcs have cost 1 ? *)
 Readln(Infile, Answer);
 Bignumbers:= ((Answer = 'S') or (Answer = 's'));        (* Standard Infinity, BigM values? *)
 If Bignumbers Then
  Begin
    Infinity:= 100;
    BigM:= 100;
  End
 Else Begin
   Read(Infinity);
   Readln(BigM);
 End;

 Readln(Infile, StanDirs);                               (* How many standard directions ? *)
 For I:= 1 to StanDirs Do
  Begin
    For J:= 1 to m Do
     Read(Infile, D[I][J]);                              (* D[I] is the Ith Standard Direction *)
    Readln(Infile);
  End;
 For K:= 1 to N Do
  A[K]:= Nil;
 Read(Infile,Tail);
```

```
 While (Tail <> 0) Do
  Begin  (* While *)                                    (* Record the information from the arc Tail----->Head *)
    New(P);
    P^.Tail:= Tail;                                    (* Read tail *)
    Read(Infile, Head);                                (* Read head *)
    P^.Head:= Head;
    Dirnumber:=1;
    If StanDirs > 0 Then
     Begin                                             (* Read progress, if a standard direction is indicated *)
       Read(Infile, Dirnumber);
       If Dirnumber > 0 Then P^.progress:= D[Dirnumber];
     End;
    If (StanDirs = 0) or (Dirnumber = 0) Then          (* Read progress, if a non-standard direction is indicated *)
      For J:= 1 to m Do
        Read(Infile, P^.progress[J]);
    If Unitcosts or EOLN(Infile) Then                  (* Assign unit cost if appropriate *)
      Begin
        P^.cost:=1;
        Readln(Infile);
      End
    Else Readln(Infile, P^.cost);                      (* Otherwise, read cost *)
    P^.Next:= A[Tail];                                 (* Insert arc at beginning of adjacency list *)
    A[Tail]:= P;
    If Reversible Then                                 (* If graph is reversible, then insert arc Head----->Tail on adjacency *)
      Begin                                            (* list with the same cost, and the (vector) negative amount of progress *)
        New(Q);
        Q^.Tail:= Head;
        Q^.Head:= Tail;
        For J:= 1 to m Do
         Q^.Progress[J]:= -P^.Progress[J];
        Q^.cost:= P^.cost;
        Q^.Next:= A[Head];
        A[Head]:= Q;
      End;
    Read(Infile, Tail);                                (* Input ends with a 0 *)
 End; (* While *)
End;
(*************************************************************************)
Procedure Writedata(Var A: Ptrarray; m:mtype; N: Nodetype;Infilename:Filename);
                                                       (* Writedata writes the adjacency list into the output file *)
Const Screensize = 8;
Var   I: Nodetype;
      J: mtype;
      K: 0..Screensize;
      P, Q: Ptr;

Begin
 Writeln;
 Writeln(outfile,'Adjacency List for Data from ',Infilename);
 Writeln(outfile);
 For I:= 1 to N Do                                     (* Print the adjacency list for node I *)
   Begin
     Write(outfile,I:2,':');
     P:= A[I];
     Q:= A[I];
     If Q = Nil then Writeln(Outfile);
     While Q <> Nil Do
        Begin (*While*)
          For K:= 1 to Screensize Do                   (* At most 8 nodes at a time, *)
             Begin
               If P <> Nil Then
                  Begin
                    Write(outfile,'------>',P^.Head:2);
                    P:=P^.Next;
                  End;
             End;
```

```
          Writeln(Outfile);
          For K:= 1 to Screensize Do
             Begin
               If Q <> Nil Then                          (* listing the progress and cost underneath their respective arcs. *)
                  Begin
                    Write(outfile,'   ','(');            (* The progress is in parenthesis, unless m = 0, in which case the *)
                    For J:= 1 to m-1 Do                  (* cost is parenthesized.                                          *)
                     Begin
                       Write(outfile,Q^.Progress[J]:1,',');
                     End;
                    If m > 0 Then Write(outfile,Q^.Progress[m]:1,')',Q^.cost:1)
                      Else Write(outfile,Q^.cost:1,')');
                    Q:=Q^.Next;
                  End;
             End; (* For K loop *)
          Writeln(outfile);
          Writeln(outfile);
        End;    (* While *)
 End;          (* For I loop *)
End;
(*************************************************************************)
Procedure Readlambda(var lambda: lambdatype; m: mtype);          (* Readlambda reads the dual vector, lambda *)
Var  J: mtype;                                                   (* Not used when m=2 or 3    *)

    Begin
      Write('Please enter lambda: ');
      For J:= 1 to m Do
        Read(lambda[J]);
      Readln;
      Writeln;
      Writeln(Outfile);
    End;
(*************************************************************************)
Procedure NoCycleMessage(I:Nodetype; lambda: lambdatype; m: mtype);
Var J: mtype;                                                    (* Nocyclemessage prints ``no neg. cycles found after I iterations.'' *)

Begin
 Writeln(outfile);
 Writeln(outfile);
 If m > 0 then
  Begin
    Write(outfile, 'When lambda equals ');
    For J:= 1 to m do
     Write(outfile, lambda[J]:6:2);
    Writeln(outfile);
    Writeln(outfile);
    Writeln;
    Writeln;
    Write('When lambda equals ');
    For J:= 1 to m do
     Write(lambda[J]:6:2);
  End;
 Writeln;
 Writeln;
 Writeln('No negative cycles detected; Termination after ',I:2,' iterations.');
 Writeln(outfile,
  'No negative cycles detected; Termination after ',I:2,' iterations.');
End;
(*************************************************************************)
Procedure BellmanFord(Var A,Pred: PtrArray; Var u: realvector;  (* Procedure BellmanFord does one iteration of the Bellman-Ford *)
                      Var Improvement: Boolean;                 (* shortest path algorithm, updating the vector u, and          *)
                      Var N: Nodetype;                          (* indicating which components of u have improved during that   *)
                      Var ImprovedSinceLastTime: Boolvector;    (* iteration (via ImprovedThisTime), as well as those which     *)
                      Var ImprovedThisTime: Boolvector);        (* have improved since the last time their adjacency list was   *)
                                                                (* scanned (via ImprovedSinceLastTime).After the I-th iteration *)
                                                                (* of BellmanFord, u[J] is  the weight of a minimum weight      *)
```

```
                                                    (* walk from node one to node J, among all walks in the I-th    *)
                                                    (* class.  The I-th class contains (properly) all walks from   *)
                                                    (* node one to node J which take I steps or fewer. Pred[J] is  *)
                                                    (* the arc which precedes node J in some such minimum weight   *)
                                                    (* walk from node one to node J.                               *)
Var  Tail, I: Nodetype;
     P: Ptr;

Begin
 Improvement:= False;
 For I:= 1 to N do
   ImprovedThisTime[I]:= False;
 For Tail:= 1 to N do                               (* Going through all arcs with tail Tail... *)
 If ImprovedSinceLastTime[Tail] Then                (* that have improved since last time these arcs were examined, *)
  Begin
    ImprovedSinceLastTime[Tail]:= False;
    P:= A[Tail];
    While P <> Nil Do
      Begin
        If (u[Tail] + P^.weight < u[P^.Head] - epsilon)  Then   (* If we can find a shorter walk to node Head with (Tail,Head) *)
                                                                (* as its last step, then update u[Head], Pred[Head] and       *)
          Begin                                                 (* the improvement variables. *)
            Improvement:= True;
            u[P^.Head]:= u[Tail] + P^.weight;
            Pred[P^.Head]:= P;
            ImprovedSinceLastTime[P^.Head]:= True;
            ImprovedThisTime[P^.Head]:= True;
          End;
        P:= P^.Next;                                    (* Examine the next arc on Tail's out-adjacency list. *)
      End;  (*While*)
  End;
End;
(*************************************************************************)
Procedure Traceback(Node, N: Nodetype;
                    Var StartofCycle:  Nodetype; Pred: PtrArray;
                    Var ExaminedBy: Nvector;
                    Var NewCycFound: Boolean);              (* Traceback starts with node Node and then follows pred[Node] *)
                                                            (* to the next node, then follows pred of this next node to    *)
                                                            (* another node, and so on, until either we reach nil or some  *)
                                                            (* node repeats itself. In the latter case, we have found a    *)
                                                            (* negative cycle, starting at the repeated node (StartofCycle)*)

Var P: ptr;
    DeadEnd: Boolean;

Begin
 ExaminedBy[Node]:= Node;
 DeadEnd:= False;
 P:= Pred[Node];
 While ((P <> Nil) And (Not NewCycFound) And Not DeadEnd) Do
  Begin
    If ExaminedBy[P^.Tail] = Node Then                  (* If the tail of our arc has been examined during this call, *)
     Begin
       NewCycFound:= True;
       StartofCycle:= P^.Tail;                          (* starting with node StartofCycle. *)
     End;
    If ExaminedBy[P^.Tail] = 0 Then                     (* If the tail of our arc has not been examined, then   *)
      ExaminedBy[P^.Tail]:= Node;                       (* examine it.                                          *)
    If ExaminedBy[P^.Tail] <> Node  Then                (* If the tail of our arc has been examined in a previous call *)
      DeadEnd:= True;                                   (* then explore no further.                             *)
    P:= Pred[P^.Tail];                                  (* Trace back one more arc via Pred *)
  End;
End;
(*************************************************************************)
Procedure PrintNegCycle(Pred:PtrArray; m:mtype; ImprovedNode, Iter, N, StartofCycle: Nodetype;
                        lambda:lambdatype; cyclenumber: integer; Var newcyclelist: Colptr);
```

```
                                                    (* PrintNegCycle prints out the negative cycle detected in *)
                                                    (* DetectNegCyc, starting at node StartofCycle.  It also   *)
                                                    (* prints lambda and the cycle's total progress, cost and  *)
                                                    (* weight, and updates the NewCycleList. Finally, it adds  *)
                                                    (* the new cycle to NewCycleList.                          *)

Var Node: Nodetype;
    P, ArcStack: ptr;
    Q: Colptr;
    J: Mtype;
    Totalprogress: mvector;
    Totalcost: Integer;
    Totalweight, ratio: Real;

Begin
 Writeln(outfile);                                  (* Print cycles number and the iteration it was detected *)
 Writeln(outfile);
 Writeln(outfile, 'Cycle number ',Cyclenumber:4);
 Writeln(outfile,'Discovered after iteration ',Iter:2);
 Writeln;
 Writeln;
 Writeln('Cycle number ',Cyclenumber:4);
 Writeln('Discovered after iteration ',Iter:2);
 Writeln(outfile);
 Writeln(outfile);

 If m > 0 then
  Begin                                             (* Print lambda *)
   Write(outfile, 'When lambda equals ');
   For J:= 1 to m do
    Write(outfile, lambda[J]:8:2);
   Writeln(outfile);
   Writeln(outfile);
   Writeln;
   Writeln;
   Write('When lambda equals ');
   For J:= 1 to m do
    Write(lambda[J]:8:2);
  End;
 Writeln;
 Writeln;

 Arcstack:= Pred[StartofCycle];
 Arcstack^.NextonCycle:= Nil;
 Node:= Pred[StartofCycle]^.Tail;
 While Node <> StartofCycle Do
   Begin                                    (* Put edges of cycle on arcstack in reverse order  *)
     Pred[Node]^.NextonCycle:= Arcstack;
     Arcstack:= Pred[Node];
     Node:= Pred[Node]^.Tail;
   End;
 P:= Arcstack;

 For J:= 1 to m do
  Totalprogress[J]:= 0;
 Totalcost:= 0;
 Totalweight:= 0;                                   (* Pop off edges in the proper order, computing and *)
 Repeat                                             (* printing the relevant information.               *)
   Write(P^.Tail,'---->',P^.Head:3,'   Progress: ');
   For J:= 1 to m Do Write(P^.Progress[J]:2);
   Writeln('  Cost: ',P^.Cost:2, '  Weight: ', P^.Weight:3:2);
   Write(outfile,P^.Tail,'---->',P^.Head:2,'   Progress: ');
   For J:= 1 to m Do Write(outfile,P^.Progress[J]:2);
   Writeln(outfile, '  Cost: ',P^.Cost:2, '  Weight: ', P^.Weight:3:2);
   For J:= 1 to m do
    TotalProgress[J]:= TotalProgress[J] + P^.Progress[J];
```

```
    TotalCost:= TotalCost + P^.Cost;
    TotalWeight:= TotalWeight + P^.Weight;
    P:= P^.NextonCycle;
  Until P = Nil;
  Writeln;
  Writeln('  Total Weight: ',TotalWeight:3:2);
  Write('TotalProgress: ');
  For J:= 1 to m do
   Write(TotalProgress[J]:3);
  Writeln('  Total Cost: ',TotalCost:3);
  Writeln;
  Writeln;
  Writeln(outfile);
  Writeln(outfile,'  Total Weight: ',TotalWeight:3:2);
  Write(Outfile,'Total Progress: ');
  For J:= 1 to m do
   Write(Outfile,TotalProgress[J]:3);
  Writeln(Outfile,'  Total Cost: ',TotalCost:3);
  Writeln(Outfile);
  Writeln(Outfile);
  Writeln(Outfile);

  New(Q);                                              (* Add new cycle to NewCycleList *)
  Q^.Name:= Cyclenumber;
  Q^.Cost:= TotalCost;
  For J:= 1 to m Do
   Q^.Progress[J]:= TotalProgress[J];
  Q^.Next:= NewCycleList;
  NewCycleList:= Q;

  If m = 1 then                                        (* When m=1, print cost/progress ratio *)
   Begin
     Ratio:= TotalCost/TotalProgress[1];
     Writeln('Cost/Progress ratio is ',ratio:5:3);
     Writeln(outfile, 'Cost/Progress ratio is ',ratio:5:3);
   End;
End;
(*************************************************************************)
Procedure DetectNegCyc(Var A:PtrArray; lambda: lambdatype; m:mtype; N:Nodetype;
                       Var NewCycleList: colptr; Var CycleNumber: Integer);
                                                       (* DetectNegCycle assigns weights to the arcs (via lambda)    *)
                                                       (* and determines if the graph contains a cycle of negative   *)
                                                       (* weight.  It calls BellmanFord at most N times.  If after   *)
                                                       (* any such call, u is not improved in any component, then    *)
                                                       (* the graph has no negative cycles.  When u does improve,    *)
                                                       (* the arcs leading into the last improved node are traced    *)
                                                       (* back.  If a cycle is found, it must be negative, and it    *)
                                                       (* is printed out, and stored temporarily on NewCycleList.    *)
                                                       (* If there was improvement, but no negative cycles were      *)
                                                       (* detected that iteration, then Bellman-Ford is called again.*)
Var  I, Iter, StartofCycle: Nodetype;
     P : Ptr;
     J : mtype;
     u : Realvector;
     Improvement, NegCycleFound, NewCycFound: Boolean;
     Pred: Ptrarray;
     ISLT, ImprovedThisTime: Boolvector;               (* ISLT[i] is true if node i has Improved Since Last Time     *)
     Examinedby: Nvector;                              (* Node i's out adjacency list was examined.                  *)
                                                       (* ImprovedThisTime[i] is true if u[i] has improved during    *)
                                                       (* the present call of BellmanFord.                           *)
                                                       (* ExaminedBy makes the TraceBack procedure more efficient.   *)
Begin

  For I:= 1 to N Do
    Begin                                              (* Assign weights to edges *)
      P:= A[I];
```

```
      While P <> Nil Do
        Begin
          P^.weight:= P^.cost;
          For J:= 1 to m Do
            P^.weight:= P^.weight - lambda[J] * P^.progress[J];
          P:= P^.Next;
        End;
    End;

 u[1]:= 0;                                                    (* Initialize u and ISLT *)
 ISLT[1]:= True;
 For I:= 2 to N do
  Begin
    u[I]:= infinity;
    ISLT[I]:= False;
  End;

 Improvement:= True;                                          (* Initialize Bellman-Ford variables *)
 NegCycleFound:= False;
 NewCyclelist:= Nil;
 Iter:= 1;
 For I:= 1 to N Do
  Pred[I]:= Nil;

 While ((Improvement) And (Iter <= N) And (Not NegCycleFound))  Do      (* Do one iteration of Bellman-Ford *)
   Begin
     BellmanFord(A, Pred, u, Improvement, N, ISLT, ImprovedThisTime);
     If Improvement Then                                              (* If any nodes improve, then *)
       Begin                                                          (* Trace back through all improved nodes *)
         For I:= 1 to N do
           ExaminedBy[I]:= 0;
         For I:= 1 to N do
           If (ImprovedThisTime[I] And (ExaminedBy[I] = 0))  Then
             Begin
               NewCycFound:= False;
               Traceback(I, N, StartofCycle, Pred, Examinedby, NewCycFound);
               If NewCycFound Then                                    (* and print out all negative cycles found. *)
                 Begin
                   NegCycleFound:= True;
                   Cyclenumber:= Cyclenumber + 1;
                   PrintNegCycle(Pred, m, I, Iter, N , StartofCycle, lambda, cyclenumber, newcyclelist);
                 End;
             End;
       End;
     Iter:= Iter + 1;
   End;

 If Not Improvement Then  NoCycleMessage(Iter - 1, lambda, m);
 End;
(*************************************************************************************************************)
Procedure GetNewLambda(Var Lambda: LambdaType; Newcyclelist: Colptr; Var Best: Integer);
                                                              (* Among the cycles on Newcyclelist, this procedure reassigns *)
                                                              (* lambda to be the new min cost/progress ratio, and          *)
                                                              (* reassigns best to the name of a min ratio cycle on the list*)
                                                              (* Used only when m=1.                                        *)
Var Q: colptr;
    Ratio: Real;

Begin
  Q:= Newcyclelist;
  While Q <> Nil Do
   Begin
     Ratio:= Q^.Cost/Q^.Progress[1];
     If Ratio < Lambda[1] Then
       Begin
         Lambda[1]:= Ratio;
```

```
          Best:= Q^.Name;
        End;
      Q:= Q^.Next;
    End;
End;
(*************************************************************************************************************)
Procedure InitializeLP(Var Nonbasiclist: Colptr; Var Basis: Basistype; Var RHS: mvector; m:mtype);
                                                    (* This procedure reads the right hand side vector and initializes *)
                                                    (* the basic and nonbasic lists.                                   *)
Var I: mtype;
    Q: colptr;

Begin
 New(Q);
 Q^.name:= 0;      (* dummy header *)
 Q^.next:= nil;
 NonbasicList:= Q;
 For I:= 1 to m do
  Basis[I]:= Nil;
 Write('Please enter the right-hand side on this line: ');
 For I:= 1 to m do
  Read(RHS[I]);
 Readln;
 Writeln;
 For I:= 1 to 5 Do
  Writeln(Outfile);
 Write(Outfile, 'Original RHS vector:');
 For I:= 1 to m do
  Write(Outfile, RHS[I]);
 Writeln(Outfile);
End;
(*************************************************************************************************************)
Procedure GetFeasible(Var Nonbasiclist: Colptr; Var Basis: Basistype; Var x, lambda: lambdatype;
                      Var Binverse: Bmatrix;  RHS: mvector;
                      m: mtype; Bigm:integer);
                                                (* This procedure constructs an initial artificial basic feasible solution *)
Var Q: Colptr;                                  (* Artificial columns have names 1,...,2m.                                 *)
  I,J: mtype;

Begin

 If Basis[1] <> Nil Then                        (* If basis is non-empty, then *)
   Begin
     For I:= 1 to m Do                          (*Put non-artificial cols. on nonbasiclist*)
      If (Basis[I]^.Name > 2*m) Then
        Begin
          Basis[I]^.next:= Nonbasiclist^.next;
          Nonbasiclist^.next:= Basis[I];
        End;
   End;

  For I:= 1 to m Do
   Begin                                        (* Construct I-th artificial basis column *)
     New(Q);
     Q^.cost:= Bigm;
     Q^.next:= Nil;
     For J:= 1 to m Do
       If (J <> I) Then Q^.Progress[J]:= 0
       Else If (RHS[I] >= 0) Then Q^.Progress[J]:=1
       Else Q^.Progress[J]:= -1;
     If RHS[I] >= 0 Then Q^.name:= 2*I - 1
      Else Q^.Name:= 2*I;;
     Basis[I]:= Q;
   End;

   Writeln('Initial artificial basis: ');       (* Print initial basis *)
```

```
    For I:= 1 to m Do
     Begin
       Write('Cycle number ',Basis[I]^.Name:4, ' Progress: ');
       For J:= 1 to m Do
          Write(Basis[I]^.Progress[J]:4);
       Writeln(' Cost: ',Basis[I]^.cost:5);
     End;
    Writeln(outfile, 'Initial artificial basis: ');
    For I:= 1 to m Do
     Begin
       Write(outfile,'Cycle number ',Basis[I]^.Name:4, ' Progress: ');
       For J:= 1 to m Do
          Write(outfile, Basis[I]^.Progress[J]:4);
       Writeln(outfile, '     Cost: ',Basis[I]^.cost:5);
     End;
    Writeln(Outfile);

     For I:= 1 to m Do
       If RHS[I] >= 0 Then x[I]:= RHS[I]                    (* Compute x, lambda and Binverse *)
         Else x[I]:= -RHS[I];
     For I:= 1 to m Do
      lambda[I]:= Basis[I]^.Progress[I] * BigM;
     For J:= 1 to m Do
       For I:= 1 to m Do
         Binverse[I,J]:= Basis[J]^.Progress[I];
End;
(*************************************************************************)
Procedure ChooseEnteringVariable(NonbasicList:colptr; lambda:lambdatype;
                                 Var EnteringVariable: colptr;
                                 Var NoSuchVariable: Boolean; m:mtype);

Var Q1,Q2: Colptr;               (* This procedure takes that first non-basic *)
    Reducedcost: Real;           (* column with negative reduced cost, points *)
    I:mtype;                     (* to it with enteringvariable, and removes  *)
                                 (* it from the nonbasic list.                *)
                                 (* If all columns have non-negative reduced  *)
                                 (* cost, then NoSuchVariable:= True.         *)
Begin
 NoSuchVariable:=True;
 Q1:= Nonbasiclist;
 Q2:=Q1^.next;
 While (Q2 <> Nil) And NoSuchVariable Do
  Begin
    ReducedCost:=Q2^.cost;
    For I:= 1 to m Do
     ReducedCost:= ReducedCost - (Lambda[I] * Q2^.Progress[I]);
    If ReducedCost < -Epsilon Then
     Begin
       NoSuchVariable:= False;
       EnteringVariable:= Q2;
       Q1^.next:= Q2^.next;      (* EnteringVariable is removed from nonbasiclist*)
     End;
    Q1:= Q1^.next;
    Q2:= Q2^.next;
  End;
End;
(*************************************************************************)
Procedure ImproveBasis(m:mtype; Var Nonbasiclist: colptr; Var Basis: Basistype;
                       Var Binverse: Bmatrix; EnteringVariable: colptr;
                       Var x,lambda: lambdatype; RHS: mvector;
                       BigM: integer; Var Unbounded: Boolean; Var DegenPivots: Integer);

Var I,J, LeavingVariable: mtype;                    (* Given the current Basis and the entering variable, this procedure   *)
    Ratio, Determinant, Solutioncost: real;         (* chooses a leaving variable (or concludes unboundedness) and updates *)
    Alpha: lambdatype;                              (* the basis, x, and lambda.                                           *)
    B: Bmatrix;
```

```
Begin

 Unbounded:= True;
 For I:= 1 to m do
   Begin
     Alpha[I]:= 0;                                    (* Alpha = Binverse * a, where a is the entering column vector.      *)
     For J:= 1 to m Do
       Alpha[I]:= Alpha[I] + Binverse[I,J] * EnteringVariable^.Progress[J];
   End;

 Ratio:= BigM * BigM;                                 (* Do min-ratio test to determine the leaving column.                *)
 For I:= 1 to m Do
   If (Alpha[I] > Epsilon) And (x[I] < (Ratio * Alpha[I]) - Epsilon) Then
      Begin
        Ratio:= x[I]/Alpha[I];
        If Ratio = 0 Then DegenPivots:= DegenPivots +1
           Else DegenPivots:= 0;
        Unbounded:= False;
        LeavingVariable:= I;
      End;

   If Not Unbounded Then
     Begin                                          (*Interchange LeavingVariable with EnteringVar.*)
       Basis[LeavingVariable]^.Next:= NonbasicList^.next;
       Nonbasiclist^.next:= Basis[LeavingVariable];
       Basis[LeavingVariable]:= EnteringVariable;
       Basis[LeavingVariable]^.next:= Nil;          (* This line is not really necessary *)

       For I:= 1 to m Do
         For J:= 1 to m Do
           B[I,J]:= Basis[J]^.Progress[I];          (* Create B matrix *)

       If m = 3 Then
          Begin
            Binverse[1,1]:= B[2,2]*B[3,3] - B[3,2]*B[2,3];      (* Compute Binverse matrix directly *)
            Binverse[1,2]:= B[3,2]*B[1,3] - B[1,2]*B[3,3];
            Binverse[1,3]:= B[1,2]*B[2,3] - B[2,2]*B[1,3];
            Binverse[2,1]:= B[3,1]*B[2,3] - B[2,1]*B[3,3];
            Binverse[2,2]:= B[1,1]*B[3,3] - B[3,1]*B[1,3];
            Binverse[2,3]:= B[2,1]*B[1,3] - B[1,1]*B[2,3];
            Binverse[3,1]:= B[2,1]*B[3,2] - B[3,1]*B[2,2];
            Binverse[3,2]:= B[3,1]*B[1,2] - B[1,1]*B[3,2];
            Binverse[3,3]:= B[1,1]*B[2,2] - B[2,1]*B[1,2];
            Determinant:=B[1,1]*B[2,2]*B[3,3] + B[1,2]*B[2,3]*B[3,1]
                        +B[1,3]*B[2,1]*B[3,2] - B[1,3]*B[2,2]*B[3,1]
                        -B[1,1]*B[2,3]*B[3,2] - B[1,2]*B[2,1]*B[3,3];
          End;

       If m = 2 Then
           Begin
             Binverse[1,1]:= B[2,2];
             Binverse[2,2]:= B[1,1];
             Binverse[1,2]:= -B[1,2];
             Binverse[2,1]:= -B[2,1];
             Determinant:= B[1,1]*B[2,2]-B[1,2]*B[2,1];
           End;

       For I:= 1 to m Do
         For J:= 1 to m Do
           Binverse[I,J]:= Binverse[I,J]/Determinant;

       For I:= 1 to m Do                            (*Directly compute Lambda = c * Binverse, where c is the basic cost vector *)
         Begin
           Lambda[I]:= 0;
           For J:= 1 to m Do
```

```
            Lambda[I]:= Lambda[I] + Basis[J]^.Cost * Binverse[J,I];
          End;

        For I:= 1 to m Do                              (*Directly compute x = Binverse * b, where b is the right hand side vector *)
          Begin
            x[I]:= 0;
            For J:= 1 to m Do
              x[I]:= x[I] + Binverse[I,J] * RHS[J];
          End;

        Solutioncost:= 0;
        For I:= 1 to m Do                              (*Directly compute solutioncost = lambda * b *)
          Solutioncost:= Solutioncost + lambda[I] * RHS[I];

        Write('New Basis: ');                          (* Print names of basic variables and solutioncost *)
        For I:= 1 to m Do
          Write(Basis[I]^.Name:3);
        Writeln('   Solution Cost: ',Solutioncost:5:2);
        Write(outfile,'New Basis: ');
        For I:= 1 to m Do
          Write(outfile,Basis[I]^.Name:3);
        Writeln(outfile,'   Solution Cost: ',Solutioncost:5:2);

      End;
End;

(*************************************************************************************************************************)
Procedure Simplex(Var Unbounded, DegenerateCycling: Boolean;
                  Var Basis: Basistype;  Var Binverse: Bmatrix;
                  Var Nonbasiclist, newcyclelist: colptr;
                  Var x, lambda: lambdatype; RHS: mvector; m: mtype);


Const MaxDegenPivots = 50;                             (* Given a current feasible basis and a list of nonbasic variables,   *)
                                                       (* this procedure calls ChooseEnteringVariable and ImproveBasis       *)
                                                       (* repeatedly until an optimal basis is found (unless unboundedness   *)
                                                       (* or degenerate cycling is detected).  The primal and dual solutions *)
                                                       (* are printed.                                                       *)
Var Enteringvariable: colptr;
    Nosuchvariable: Boolean;
    I: mtype;
    DegenPivots: Integer;

Begin
 Unbounded:= False;
 DegenerateCycling:= False;
 DegenPivots:= 0;

 Repeat
   ChooseEnteringVariable
                    (NonbasicList,lambda,EnteringVariable,NoSuchVariable,m);

   If NoSuchVariable Then                              (* If all reduced costs are non-negative, then *)
     Begin                                             (* Printout current optimal solution. *)
       Write('Basic Solution: x = [');
       For I:= 1 to m-1 Do
         Write(x[I]:6:2,',');
       Writeln(x[m]:6:2,']');
       Write('Dual Solution: lambda = [');
       For I:= 1 to m-1 Do
         Write(lambda[I]:6:2,',');
       Writeln(lambda[m]:6:2,']');
       Write(outfile,'Basic Solution: x = [');
       For I:= 1 to m-1 Do
         Write(outfile,x[I]:6:2,',');
       Writeln(outfile,x[m]:6:2,']');
```

```
        Write(outfile,'Dual Solution: lambda = [');
        For I:= 1 to m-1 Do
          Write(outfile,lambda[I]:6:2,',');
        Writeln(outfile,lambda[m]:6:2,']');
      End

    Else                                                (* Otherwise, Improve the Basis. *)
      ImproveBasis(
    m,NonbasicList,Basis,Binverse,EnteringVariable,x,lambda,RHS,Bigm,Unbounded,DegenPivots);
  Until NoSuchVariable or Unbounded;

  If Unbounded Then
   Begin
    Write('Unbounded solution detected while entering ');
    Writeln('column', EnteringVariable^.name, 'into the basis.');
    Write(outfile,'Unbounded solution detected while entering ');
    Writeln(outfile,'column', EnteringVariable^.name, 'into the basis.');
   End;

  If DegenPivots = MaxDegenPivots Then
   Begin
    Writeln('Termination due to degenerate cycling.');
    Writeln(outfile);
    Writeln(outfile, 'Termination due to degenerate cycling.');
    DegenerateCycling:= True;
   End;

End;
(*************************************************************************)
Procedure Append(Var list1, list2: colptr);
                                (* This procedure appends the contents of  *)
                                (* list1 to the beginning of list2 (after  *)
                                (* the dummy header). It is used to append *)
                                (* the NewCycleList to the NonbasicList.   *)
Var Q: colptr;

Begin
 Q:= list1;
 While Q^.next <> nil Do
  Q:= Q^.next;                  (* Point to last element on list1 *)
 Q^.next:= list2^.next;         (* Point this to the first (non-header) element*)
 List2^.next:= list1;           (* Remember dummy header on list 2 *)
End;
(*************************************************************************************************)
Procedure EnterNewRHS(Var RHS: mvector; m: mtype);

Var I: mtype;

        Begin           (*Enter new RHS*)
          Write('Please enter the right-hand side on this line: ');
          For I:= 1 to m Do Read(RHS[I]);
          Readln;
          Writeln;
          Writeln(Outfile);
          Writeln(Outfile,'*********************************************');
          For I:= 1 to 5 do
            Writeln(Outfile);
          Write(Outfile,'New Right-hand side: ');
          For I:= 1 to m do
            Write(Outfile, RHS[I]);
          For I:= 1 to 3 do
           Writeln(Outfile);
        End;


(*************************************************************************)
```

```
(*********************  MAIN  PROCEDURE  ******************************)
(*************************************************************************)

Begin
 Readinfile(Infilename);                          (* Read C-Graph and other problem parameters *)
 GetData(m,N,A,done,lambda,Cyclenumber, Infinity, BigM);
 Close(Infile);
 Readoutfile(Outfilename);
 Writedata(A,m,N,Infilename);

 If m = 0 Then                                    (* If m=0, then just find a negative cycle. *)
  Begin
    Writeln;
    Writeln('You may prefer to use the program Cycles.Pas');
    DetectNegCyc(A,lambda,m,n,NewCycleList,Cyclenumber);
  End;

 If m = 1 Then                                    (* If m=1, then find the cycle with min cost/progress ratio *)
  Begin
    Cyclenumber:= 0;
    Best:= 0;
    Readlambda(lambda,m);                         (* Enter initial lambda *)

    Repeat                                        (* Assign C-graph weights and find a negative cycle (if one exists). *)
      DetectNegCyc(A,lambda,m,n,Newcyclelist,Cyclenumber);
      If NewCycleList <> Nil Then GetNewLambda(lambda, newcyclelist, best);
    Until Newcyclelist = Nil;                     (* When no negative cycle exists, we are done. *)

    Writeln(Lambda[1]:5:3,' is the min. cost ratio, and is attained at cycle number ', Best:4);
    Writeln(outfile, Lambda[1]:5:3,' is the min. cost ratio, and is attained at cycle number ', Best:4);
  End;

 If m > 3 Then                                                (* Could also use the program C-GRAPH.PAS *)
  Begin
   Writeln('LP calculations must be solved externally.');     (* Perhaps using Lindo *)
   Repeat
    Readlambda(lambda,m);                                     (* Using lambda, assign C-Graph weights, and *)
    DetectNegCyc(A,lambda,m,n,Newcyclelist,Cyclenumber);      (* find a negative cycle, if one exists. *)
    Write('Would you like to run the program with');          (* Using simplex information, you can enter a new value of *)
    Write(' a new value of lambda? (Y/N): ');                 (* lamabda, if desired. *)
    Readln(Answer);
    If (Answer = 'N') or (Answer = 'n') Then Done:= True;
   Until Done;
  End;

 If (m=2) or (m=3) Then                                       (* When m= 2 or 3, the program performs the simplex calculations*)
  Begin
   InitializeLP(Nonbasiclist, basis, RHS, m);                 (* First, we read the RHS vector, *)
   Repeat
     GetFeasible(Nonbasiclist, Basis, x, lambda, Binverse, RHS, m, bigm);      (* Create an artificial basis, if necessary *)
     Simplex(
       Unbounded,DegenerateCycling,basis,Binverse,nonbasiclist,newcyclelist,x,lambda,RHS,m);
                                                                               (* Using generated columns, solve the LP *)
     If Not (Unbounded or DegenerateCycling) Then

       Repeat                                                 (* (Re-)assign C-Graph weights, and find negative cycles, *)
         Detectnegcyc(A,lambda,m,n,Newcyclelist,Cyclenumber); (* if they exist. *)
         If Newcyclelist = Nil Then                           (* If none exists, then the previous solution is optimal. *)
           Begin
             Writeln('The above solution is optimal.');
             Writeln(outfile);
             Writeln(outfile);
             Writeln(outfile, 'The above solution is optimal.');
           End
         Else                                                 (* Otherwise, add the new columns (cycles) to the genereated *)
          Begin                                               (* columns list, and re-solve the LP, using SIMPLEX.         *)
```

```
          Append(Newcyclelist,Nonbasiclist);
          Simplex(Unbounded,DegenerateCycling,basis,Binverse,
                  nonbasiclist,newcyclelist,x,lambda,RHS,m);
        End;
      Until (NewCycleList = Nil) or Unbounded or DegenerateCycling;

      Writeln('Would you like to use another Right Hand Side? (Y/N) ');
      Readln(Answer);
      If (Answer = 'N') or (Answer = 'n') Then Done:= True;      (* Re-enter, a new right-hand side, if desired *)
      If Not Done Then EnterNewRHS(RHS, m);

  Until Done;
 End;
Close(Outfile);
End.

(*****************************************************************************************************************)
(******                                        INPUT INSTRUCTIONS                                          *******)
(*****************************************************************************************************************)
(* THE DATA ARE ENTERED VIA SOME INPUT FILE (E.G., FOURNODE.DAT). THE DATA ARE LEFT JUSTIFIED AS FOLLOWS:          *)
(* M ----THE DIMENSION OF THE PROGRESS CONSTRAINTS                                                               *)
(* N ----THE NUMBER OF NODES IN THE GRAPH (ACTUALLY, THE HIGHEST NUMBERED NODE IN THE GRAPH)                     *)
(* IF G IS REVERSIBLE, THEN TYPE AN 'R' HERE (OPTIONALLY FOLLOWED BY ANYTHING ELSE). OTHERWISE TYPE ANY OTHER CHARACTER. *)
(* IF ALL ARCS HAVE COST 1, THEN TYPE A 'U' HERE.   "                                  "                         *)
(* IF THE STANDARD INFINITY VALUES (INFINITY = 100; BIGM = 100) ARE TO BE USED TYPE AN 'S' HERE.  OTHERWISE    "    .    *)
(* IF AN S WAS NOT TYPED ON THE ABOVE LINE, ENTER THE VALUES ON THE  SAME LINE.                                  *)
(* STANDIR ----THE NUMBER OF STANDARD DIRECTIONS                                                                 *)
(* LIST THE STANDARD DIRECTIONS, ONE ROW AT A TIME (UNLESS STANDIR = 0)                                          *)
(* LIST THE ARCS, ONE ROW AT A TIME, AS FOLLOWS:                                                                 *)
(* TAIL   HEAD   DIRNUMBER(IF STANDIRS > 0)  PROGRESS(IF DIRNUMBER = 0 OR STANDIRS = 0) COST(IF NO UNITCOSTS OR COST<>1) *)
(* 0 ----INDICATES THE END OF THE INPUT                                                                          *)
(*****************************************************************************************************************)
(*  EXAMPLE-- THE FOLLOWING DATA APPEAR IN FOURNODE.DAT:
2
4
REVERSIBLE
Not Unit Costs
Not Standard Infinite values
100, 200
3
0 0
0 1
1 0
1 2 1
1 3 2 2
2 3 3
2 4 0 1 1 3
3 4 0 1 2
0

THIS REPRESENTS THE FOLLOWING C-GRAPH:
Adjacency List for Data from fournode.dat

 1:------> 3------> 2
   (0,1)2   (0,0)1

 2:------> 4------> 3------> 1
   (1,1)3   (1,0)1   (0,0)1

 3:------> 4------> 2------> 1
   (1,2)1   (-1,0)1   (0,-1)2

 4:------> 3------> 2
   (-1,-2)1   (-1,-1)3

NOTE: IF THE GRAPH IS NOT UNIT COST AND THE COST OF SOME ARC IS 1, THEN IF YOU CHOOSE NOT TO TYPE IN THE COST OF 1, BE SURE NOT
```

```
TO LEAVE ANY SPACE AFTER THE PREVIOUS ENTRY.  ALSO, DO NOT SEPARATE ENTRIES BY COMMAS.

IN THE SPECIAL CASE OF M = 0 (NO PROGRESS VECTORS), THE INPUT IS AS FOLLOWS:
0
N
X
X
S
0
TAIL HEAD COST
     :
     :
TAIL HEAD COST
0
                      *)
```

Appendix C
Turnpikes with Inequalities

C.1 Inequality Version of Theorem 2.2

Here we provide the necessary technical details for modifying Theorem 2.2 of Chapter 2, when our problem requires us to maneuver from $(\mathcal{O}, \mathbf{0})$ to $(\mathcal{D}, \Delta)$ at near minimum cost, where $\boldsymbol{\Delta} = (\boldsymbol{\Delta}_1, \boldsymbol{\Delta}_2, \boldsymbol{\Delta}_3)^T \in \mathbf{Z}^m$ is unspecified, but must satisfy $\boldsymbol{\Delta}_1 \geq d\mathbf{b}_1$, $\boldsymbol{\Delta}_2 = d\mathbf{b}_2$, $\boldsymbol{\Delta}_3 \leq d\mathbf{b}_3$. We define $\mathbf{b} = (\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3)^T$.

Here, we solve the linear program $\min \mathbf{c}^T\mathbf{x}$ subject to $A^1\mathbf{x} \geq d\mathbf{b}_1$, $A^2\mathbf{x} = d\mathbf{b}_2$, and $A^3\mathbf{x} \leq d\mathbf{b}_3$, $\mathbf{x} \geq 0$, where the matrix A is partitioned into submatrices A^1, A^2, and A^3 in the obvious way.

As before, we let $\mathbf{x}^* = (x_1, \ldots, x_m)$ denote the basic part of an optimal basic solution to the linear program. With $d\mathbf{b}^* = \sum_{j=1}^m x_j\mathbf{a}_j$, our turnpike trajectory is now

$$(\mathcal{O}, \mathbf{0}_m) \xrightarrow{mc_0 + \sum_{j=1}^m c^j \lfloor x_j \rfloor} (T_m, \sum_{j=1}^m \lfloor x_j \rfloor \mathbf{a}_j) \xrightarrow{c_\delta} (\mathcal{D}, \lceil d\mathbf{b}^* \rceil);$$

it performs $m+1$ brute forces and goes around m cycles (fewer, if the solution is degenerate), in the usual way. Note that $\lceil d\mathbf{b}^* \rceil$ satisfies our destination requirement, since $d\mathbf{b}^*$ does and $d\mathbf{b} \in \mathbf{Z}^m$.

To prove near-optimality, the only non-obvious detail to be shown is that $\|\delta\|$ is bounded above by a constant that does not depend on d or $\mathbf{b}$. This is accomplished after observing that

$$\begin{aligned}
\delta &= \lceil d\mathbf{b}^* \rceil - (\sum_{j=1}^m \lfloor x_j \rfloor \mathbf{a}_j) \\
&= \lceil d\mathbf{b}^* \rceil - (\sum_{j=1}^m x_j\mathbf{a}_j) + (\sum_{j=1}^m (x_j - \lfloor x_j \rfloor)\mathbf{a}_j) \\
&= \lceil d\mathbf{b}^* \rceil - d\mathbf{b}^* + \sum_{j=1}^m f_j\mathbf{a}_j, \text{ (where } 0 \leq f_j < 1) \\
&< (1, 1, \ldots, 1)^T + \sum_{j=1}^m |\mathbf{a}_j|.
\end{aligned}$$

C.2 Inequality Version of Theorem 2.3

Here we consider the same problem as in the last section, but with the additional assumption that $\mathbf{b} > 0$ and the additional restriction that brute forcing with progress δ can only be assured when $\delta \geq \mathbf{r}$, where $\mathbf{r} = (\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3)^T \in \mathbf{Z}_+^m$ is our prescribed radius-of-maneuver vector. As in Chapter 2, we define $M = max_{i=1\ldots n}\|\mathbf{a}_i\|$, and $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3)$ to be the vector with m ones.

Our turnpike trajectory is based on the linear program $\min \mathbf{c}^T\mathbf{x}$ subject to $A^1\mathbf{x} \geq \hat{\mathbf{b}}_1$, $A^2\mathbf{x} = \hat{\mathbf{b}}_2$, and $A^3\mathbf{x} \leq \hat{\mathbf{b}}_3$, $\mathbf{x} \geq 0$, where $\hat{\mathbf{b}} = (\hat{\mathbf{b}}_1, \hat{\mathbf{b}}_2, \hat{\mathbf{b}}_3) = (d\mathbf{b}_1 + mM\mathbf{e}_1, d\mathbf{b}_2 - (m+1)\mathbf{r}_2 - mM\mathbf{e}_2, d\mathbf{b}_3 - (m+1)\mathbf{r}_3 - mM\mathbf{e}_3) \gg \mathbf{0}_m$ for d sufficiently large. Let $\mathbf{x}^* = (x_1, \ldots, x_m)$ denote the basic part of an optimal basic solution to the above linear program. (Such a solution is guaranteed to exist for the usual reasons.) Define $\mathbf{b}^* = A\mathbf{x}^*$.

Our turnpike trajectory, constructed in the usual way, goes as follows:

$$(\mathcal{O}, \mathbf{0}_m) \overset{mc\mathbf{r}+\sum_{j=1}^m c^j\lfloor x_j\rfloor}{\longrightarrow} (T_m, m\mathbf{r} + \sum_{j=1}^m \lfloor x_j\rfloor \mathbf{a}_j) \xrightarrow{c_\delta} (\mathcal{D}, m\mathbf{r} + \sum_{j=1}^m \lfloor x_j\rfloor \mathbf{a}_j + \delta),$$

where $\delta = (\delta_1, \delta_2, \delta_3)^T$, with $\delta_1 = \mathbf{r}_1$, $\delta_2 = d\mathbf{b}_2 - (m\mathbf{r}_2 + \sum_{j=1}^m \lfloor x_j\rfloor \mathbf{a}_j^2)$, and $\delta_3 = \mathbf{r}_3$. It remains to prove the following:

1. $\mathbf{r} \leq \delta \leq \mathbf{r} + 2mM\mathbf{e}$.
2. $m\mathbf{r} + \sum_{j=1}^m \lfloor x_j\rfloor \mathbf{a}_j + \delta$ is a feasible destination.
3. The trajectory is almost optimal.

For the first item, note that $\delta_1 = \mathbf{r}_1$ and $\delta_3 = \mathbf{r}_3$, while

$$\begin{aligned}
\delta_2 &= d\mathbf{b}_2 - (m\mathbf{r}_2 + \sum \lfloor x_i\rfloor \mathbf{a}_i^2) \\
&= d\mathbf{b}_2 - m\mathbf{r}_2 + \sum (x_i - \lfloor x_i\rfloor)\mathbf{a}_i^2 - \sum x_i\mathbf{a}_i^2 \\
&= d\mathbf{b}_2 - m\mathbf{r}_2 + \sum (x_i - \lfloor x_i\rfloor)\mathbf{a}_i^2 - \mathbf{b}_2^*.
\end{aligned}$$

Letting $f_i = x_i - \lfloor x_i\rfloor$, and recalling $\mathbf{b}_2^* = \hat{\mathbf{b}}_2$, we obtain

$$\begin{aligned}
\delta_2 &= d\mathbf{b}_2 - m\mathbf{r}_2 + \sum f_i\mathbf{a}_i^2 - d\mathbf{b}_2 + (m+1)\mathbf{r}_2 + mM\mathbf{e}_2 \\
&= \mathbf{r}_2 + \sum f_i\mathbf{a}_i^2 + mM\mathbf{e}_2.
\end{aligned}$$

Since $|\sum f_i \mathbf{a}_i^2| \leq \sum f_i |\mathbf{a}_i^2| \leq mM\mathbf{e}_2$, we have $\mathbf{r}_2 \leq \delta_2 \leq \mathbf{r}_2 + 2mM\mathbf{e}_2$, as desired.

For the second item, we see first that

$$\begin{aligned} m\mathbf{r}_1 + \sum \lfloor x_j \rfloor \mathbf{a}_j^1 + \delta_1 &= (m+1)\mathbf{r}_1 + \sum \lfloor x_j \rfloor \mathbf{a}_j^1 \\ &= (m+1)\mathbf{r}_1 - \sum (x_j - \lfloor x_j \rfloor)\mathbf{a}_j^1 + \sum x_j \mathbf{a}_j^1 \\ &= (m+1)\mathbf{r}_1 - \sum f_j \mathbf{a}_j^1 + \mathbf{b}_1^* \\ &\geq (m+1)\mathbf{r}_1 - mM\mathbf{e}_1 + \hat{\mathbf{b}}_1 \\ &= (m+1)\mathbf{r}_1 - mM\mathbf{e}_1 + d\mathbf{b}_1 + mM\mathbf{e}_1 \\ &= d\mathbf{b}_1 + (m+1)\mathbf{r}_1 \\ &\geq d\mathbf{b}_1. \end{aligned}$$

Next, by definition, $m\mathbf{r}_2 + \sum \lfloor x_j \rfloor \mathbf{a}_j^2 + \delta_2 = d\mathbf{b}_2$.

And furthermore,

$$\begin{aligned} m\mathbf{r}_3 + \sum \lfloor x_j \rfloor \mathbf{a}_j^3 + \delta_3 &= (m+1)\mathbf{r}_3 + \sum \lfloor x_j \rfloor \mathbf{a}_j^3 \\ &= (m+1)\mathbf{r}_3 - \sum (x_j - \lfloor x_j \rfloor)\mathbf{a}_j^3 + \sum x_j \mathbf{a}_j^3 \\ &= (m+1)\mathbf{r}_3 - \sum f_j \mathbf{a}_j^3 + \mathbf{b}_3^* \\ &\leq (m+1)\mathbf{r}_3 + mM\mathbf{e}_3 + \hat{\mathbf{b}}_3 \\ &= (m+1)\mathbf{r}_3 + mM\mathbf{e}_3 + d\mathbf{b}_3 - (m+1)\mathbf{r}_3 - mM\mathbf{e}_3 \\ &= d\mathbf{b}_3, \end{aligned}$$

completing the proof of the second item.

To prove near optimality, we first note that by the first item, $c_\delta \leq \bar{c} \equiv \max\{c_\rho : \mathbf{r} \leq \rho \leq \mathbf{r} + 2mM\mathbf{e}\}$, where $\bar{c}$ does not depend on d or $\mathbf{b}$.

Whence,

$$\begin{aligned} \text{TPCOST} &= mc_{\mathbf{r}} + \sum_{j=1}^{m} c^j \lfloor x_j \rfloor + c_\delta \\ &\leq mc_{\mathbf{r}} + \sum_{j=1}^{m} c^j x_j + \bar{c} \\ &= mc_{\mathbf{r}} + u^* + \bar{c}, \end{aligned}$$

where u^* is the optimal value of the linear program introduced above.

Now suppose that $(\mathcal{O}, \mathbf{0}_m) \xrightarrow{c^*} (\mathcal{D}, \mathbf{\Delta})$ is an optimal trajectory. Continue it to create the usual closed walk to $(\mathcal{O}, \mathbf{\Delta} + \mathbf{r})$, with total cost $c^* + c_{\mathbf{r}}$, and with total progress $\mathbf{\Delta} + \mathbf{r}$ which compares properly to the vector $\overline{\mathbf{b}} \equiv d\mathbf{b} + \mathbf{r}$. Replacing the previous linear program's right hand side by $\overline{\mathbf{b}}$, and denoting its optimal value as z^*, we can write $c^* + c_{\mathbf{r}} \geq z^*$.

Whence, $z^* - c_{\mathbf{r}} \leq c^* \leq \text{TPCOST} \leq mc_{\mathbf{r}} + \bar{c} + u^*$, i.e.,

$$\begin{aligned} \text{TPCOST} - c^* &\leq (m+1)c_{\mathbf{r}} + \bar{c} + u^* - z^* \\ &\leq (m+1)c_{\mathbf{r}} + \bar{c} + k_A \|\overline{\mathbf{b}} - \hat{\mathbf{b}}\| \end{aligned}$$

by the theorem of Mangasarian and Shiau (which is valid for the inequality case, too). Since $\overline{\mathbf{b}} - \hat{\mathbf{b}}$ is independent of d and $\mathbf{b}$, we have established near optimality.

Note that if all of the destination constraints are of the $\leq$ variety, then our turnpike trajectory is pure brute force, since the associated LP has $\mathbf{0}_n$ as an optimal solution.

References

[1] AHUJA, R.K., J.L. BATRA, AND S.K. GUPTA. 1983. Combinatorial Optimization with Rational Objective Functions: A Communication. *Mathematics of Operations Research* **8**, p. 314.

[2] AUSLANDER, J., A.T. BENJAMIN, AND D. WILKERSON. 1988. Speed of Light Configurations in n-Dimensional Jumping Games. Unpublished Manuscript.

[3] BELUR, A. AND A.J. GOLDMAN. 1985. Solitaire Chinese Checkers. Independent Research Report, Mathematical Sciences Department, The Johns Hopkins University.

[4] BROOKS, R. 1983. Solving the Find-Path Problem by Good Representation of Free-Space. *IEEE Transactions on Systems, Man, and Cybernetics* **13**, 190–197.

[5] CASS, D. 1966. Optimum Growth in an Aggregative Model of Capital Accumulation: A Turnpike Theorem. *Econometrica* **34**, 833–850.

[6] CASTELLS, C., AND A.J. GOLDMAN. 1983. Analysis of Some Jumping Games. Independent Research Report, Mathematical Sciences Department, The Johns Hopkins University.

[7] CHEN, I. 1975. A Node Elimination Method for Finding Negative Cycles in a Directed Graph. *INFOR* **13**, 147–158.

[8] CHEN, S., AND R. SAIGAL. 1977. A Primal Algorithm for Solving a Capacitated Network Flow Problem with Additional Linear Constraints. *Networks* **7**, 59-79.

[9] CHRÉTIENNE, P. 1984. Chemins Extrémaux d'un Graphe Doublement Valué. *R.A.I.R.O. Recherche opérationelle* **18**, 221–245.

[10] CHVÁTAL, V. 1983. *Linear Programming.* W.H. Freeman & Co.

[11] DANTZIG, G.B., W. BLATTNER, AND M.R. RAO. Finding a Cycle in a Graph with Minimum Cost to Time Ratio with Application to

a Ship Routing Problem in *Theory of Graphs,* P.Rosenstiehl, editor, Dunod, Paris, and Gordon and Breach, New York, 77–84.

[12] DESROCHERS, M. 1987. A Note on the Partitioning Shortest Path Algorithm. *O.R. Letters* **6,** 183–187.

[13] FLORIAN, M. AND P. ROBERT. 1971. A Direct Search Method to Locate Negative Cycles. *Mgmt. Sci.* **17,** 307–310.

[14] FLORIAN, M. AND P. ROBERT. 1972. Rejoinder: Direct Search Method for Finding Negative Cycles. *Mgmt. Sci.* **19** 335–336.

[15] FORD, L.R., AND D.R. FULKERSON. 1958. A Suggested Computation for Maximal Multi-Commodity Network Flows. *Mgt. Sci.* **5** 97–101.

[16] FUCHS, L. 1960. *Abelian Groups.* Pergamon Press.

[17] GAREY, M.R., AND D.S. JOHNSON. 1978. *Computers and Intractability.* W.H. Freeman and Co., New York.

[18] GIBBONS, A. 1985. *Algorithmic Graph Theory.* Cambridge University Press.

[19] GILMORE, P.C., AND R.E. GOMORY. 1966. The Theory and Computation of Knapsack Functions. *Opns. Res.* **14,** 1045–1074.

[20] GLOVER, F. AND D. KLINGMAN. 1987. New Sharpness Properties, Algorithms and Complexity Bounds for Partitioning Shortest Path Procedures, Management Science/Information Science Report 87-3, Graduate School of Business Administration, University of Colorado, Boulder, CO.

[21] HOWARD, R.A. 1960. *Dynamic Programming and Markov Processes,* The M.I.T. Press.

[22] IBA, G.A. 1985. Learning by Discovering Macros in Puzzle Solving. *9th IJCAI* 640–642.

[23] IWANO, K., AND K. STEIGLITZ. 1987. Testing for Cycles in Infinite Graphs with Periodic Structure. *Proc. of 19th ACM STOC* 46–55

[24] JACOBSON, N. 1974. *Basic Algebra I.* W.H. Freeman and Company.

[25] KARP, R.M. 1978. A Characterization of the Minimum Cycle Mean in a Digraph. *Discrete Math.* **23,** 309–311.

[26] KLEIN, M. AND R.K. TIBREWALA. Finding Negative Cycles. *INFOR* **11,** 59–65.

[27] KOSARAJU, R. AND G. SULLIVAN. 1988. Detecting Cycles in Dynamic Graphs in Polynomial Time. *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing* 398–406.

[28] LAWLER, E.L. 1967. Optimal Cycles in Doubly Weighted Directed Linear Graphs in *Theory of Graphs,* P.Rosenstiehl, editor, Dunod, Paris, and Gordon and Breach, New York, 209–214.

[29] LAWLER, E.L. 1976. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, New York.

[30] MANGASARIAN, O.L. AND T.-H. SHIAU. 1987. Lipschitz Continuity of Solutions of Linear Inequalities, Programs and Complementarity Problems, *SIAM J. Control and Optimization* **25,** 583–595.

[31] MCKENZIE, L. 1986. Optimal Economic Growth, Turnpike Theorems and Comparative Dynamics, in *Handbook of Mathematical Economics, v. III,* K.J. Arrow and M.D. Intrilligator, eds. Elsevier Science Publishers, 1281–1355.

[32] MEGIDDO, N. 1979. Combinatorial Optimization with Rational Objective Functions. *Math. Opns. Res.* **4,** 414-424.

[33] MITCHELL J. 1987. Shortest Rectilinear Paths Among Obstacles, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY.

[34] NETTER, J.P. 1971. An Algorithm to Find Elementary Negative-Cost Circuits with a Given Number of Arcs–The Travelling Salesman Problem. *Opns. Res* **19,** 234–237.

[35] O'NEILL, R.P. 1977. Column Dropping in the Dantzig-Wolfe Convex Programming Algorithm: Computational Experience. *Opns. Res.* **25,** 148–155.

[36] O'NEILL, R.P. AND W.B. WIDHELM. 1976. Acceleration of Lagrangian Column-Generation Algorithms. *Mgmt. Sci.* **23,** 50–58.

[37] ORLIN, J. 1984. Some Problems on Dynamic/Periodic Graphs, in *Progress in Combinatorial Optimization.* W.R. Pulleyblank, editor. Academic Press, 279–293.

[38] SHAPIRO, J.F. 1968. Turnpike Planning Horizons for a Markovian Decision Model. *Mgmt. Sci.* **14,** 319–341.

[39] SHAPIRO, J.F., AND H.M. WAGNER. 1967. A Finite Renewal Algorithm for the Knapsack and Turnpike Models. *Opns. Res.* **15,** 319–341.

[40] YEN, J.Y. 1970. An Algorithm for Finding Shortest Routes from all Source Nodes to a Given Destination in General Networks. *Quart. Appl. Math* **27** 526–530.

[41] YEN, J.Y. 1972. On the Efficiency of a Directed Search Method to Locate Negative Cycles in a Network. *Mgmt. Sci* **19,** 333-335.

VITA

Arthur Todd Benjamin was born in Cleveland, Ohio on March 19, 1961. In 1983, he graduated with the B.S. with University Honors in Applied Mathematics from Carnegie-Mellon University, and in 1985 he received the M.S.E. in Mathematical Sciences from The Johns Hopkins University. While attending Hopkins, he was awarded a Rufus P. Isaacs Fellowship and a National Science Foundation Graduate Fellowship. His paper "Graphs, Maneuvers and Turnpikes", based on this dissertation research, was awarded the 1988 George E. Nicholson Prize from the Operations Research Society of America. He will be joining the Department of Mathematics at Harvey Mudd College in Claremont, California as an Assistant Professor in August, 1989.