Enumerating Backgammon Positions: The Perfect Hash

By Arthur Benjamin and Andrew M. Ross

ABSTRACT

Like many games, people place money wagers on backgammon games. These wagers can change during the game. In order to make intelligent bets, one needs to know the chances of winning at any point in the game. We were working on this for positions near the end of the game when we needed to explicitly label each of the positions so the computer could refer to them. The labeling developed here uses the least possible amount of computer memory, is reasonably fast, and works well with a technique known as dynamic programming.

INTRODUCTION

The Game of Backgammon

Backgammon is played on a board with 15 checkers per player, and 24 points (organized into four tables of six points each) that checkers may rest on. Players take turns rolling the two dice to move checkers toward their respective "home boards," and ultimately off the board. The first player to move all of his checkers off the board wins.

We were concerned with the very end of the game, when each of the 15 checkers is either on one of the six points in the home board, or off the board. This is known as the "bearoff," since checkers are borne off the board at each turn. Some sample bearoff positions are shown in Figure 1.

We wanted to calculate the chances of winning for any arrangement of positions in the bearoff. To do this, we needed to have a computer play backgammon against itself, to keep track of which side won more often. As part of its strategy, the computer needed to know how many turns it would take to bear off completely from any bearoff position. Thus, we need to know how many bearoff positions there are, and we need a good way of referring to them (a "hash" function).

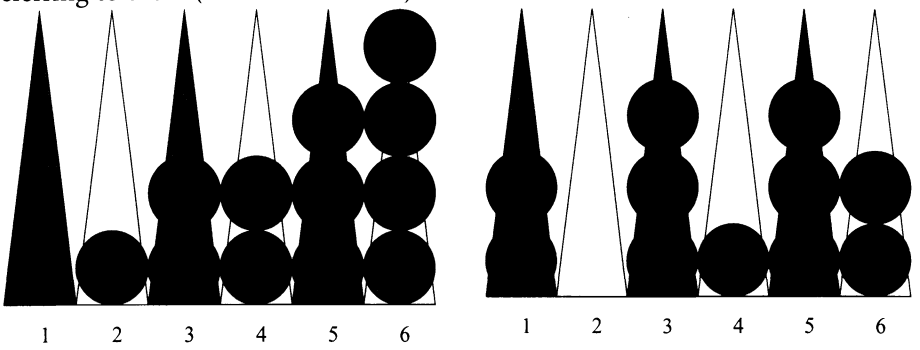

Figure 1. Sample bearoff positions. Left: 3 checkers off the board. Right: 4 checkers off the board.

The Problem At Hand

We want a way to look at a backgammon position and give it a unique and easy-to-compute number that specifies the position. The most obvious way to refer to positions would have used impractical amounts of memory (almost five thousand times as much as necessary). It was like writing a dictionary with entries for "aaaaa" through "zzzzz", with most of the entries blank. Instead, we found a way to give each position a number, such that there are no gaps between positions (no blank entries in the dictionary). Computer scientists refer to this as a "hash" function. Moreover, the function is easy to compute in either direction. It is as if we can look at any word and know which page in the dictionary it appears on, and we can tell which word appears on a page without looking it up. This ability is due to the special structure of backgammon.

SOME SIMPLE COMBINATORICS

In order to develop our enumeration scheme, we need some simple formulas from combinatorics (the study of ways to combine things). The first is the factorial function, denoted with an exclamation mark (!). This denotes the number of ways of ordering some number of objects. The number of ways to put four books onto a bookshelf is denoted $4!$. These numbers are easily calculated: $n! = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$. So, $1! = 1$, $2! = 2$, $3! = 6$, $4! = 24$, $5! = 120$, and so forth.

We can use the factorial function to answer more complicated questions. How many ways are there of choosing 4 books out of 6, and putting them on a shelf? The answer is $6!/(6-4)!$, or 360. This is only the case if we must decide on the order of the 4 books, though. If the final order doesn't matter, our value of 360 is too large: the selection ABCD gets counted as different from DCBA, even though the same books were chosen. In fact, each selection is counted $4!$ times. So, to get the number of ways to chose 4 books from 6 (denoted $\binom{6}{4}$, pronounced "six choose four" or "choose four from six"), we take

$$\frac{6!}{(6-4)!4!} = 15$$

And, in general,

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

When we are using the "choose" function, we are not allowed to take more than one of each item. We can't take the same book twice, for example. If we relax this restriction, we end up with more possibilities. How many possible four-scoop ice cream cones are there when we have six flavors to choose from? Now we could get four of the same flavor, or two of two flavors, and so forth. This problem is denoted $\left(\!\binom{6}{4}\!\right)$,

and pronounced "six multichoose four". It is possible to show that

$$\left(\!\!\binom{n}{k}\!\!\right) = \binom{n+k-1}{k}$$

This means that there are $\left(\!\!\binom{6}{4}\!\!\right) = \binom{9}{4} = 126$ ways to order a four-scooper when presented with six flavors. This is substantially more than the number of choices we had if we could not duplicate flavors.

It is also possible to show that $\left(\!\!\binom{n}{k}\!\!\right)$ has another interpretation: it is the number of ways to distribute k identical objects to n distinct recipients. One can think of this as distributing four votes to the six ice cream flavors. One could get all of the votes, or four of them could get one vote each, and so forth. It is this last interpretation that will be useful in enumerating backgammon positions.

HOW MANY BEAROFF POSITIONS ARE THERE?

There are at most 15 checkers to distribute among 6 points. We say "at most" because some of the checkers might have been borne off the board already. If K_i denotes the number of checkers on the i-th point, then it must be the case that $K_1 + K_2 + \ldots + K_6 \leq 15$. If we add a "dummy" point, point 0, that counts the number of checkers off the board, we have $K_0 + K_1 + \ldots + K_6 = 15$. The number of ways to do this, as discussed above, is $\left(\!\!\binom{n}{k}\!\!\right)$, with $n = 7$ points and $k = 15$ identical checkers. So, there are $\left(\!\!\binom{7}{15}\!\!\right) = 54264$ possible bearoff positions.

STORING BEAROFF POSITIONS

Each one of these 54264 positions may be represented as a length seven vector of integers, each one between zero and fifteen (inclusive). If we wanted to store some information for each position, and declared a 7-dimensional array on the computer, it would require at least $16^7 > 268$ Megabytes of RAM, an impractical amount on almost any computer. A better method of storage is needed.

One reason for the large amount of RAM is the sparseness of the array. The positions are constrained to have the sum of their elements equal 15; that is, if K is a bearoff position vector, $\sum_{i=0}^{6} K_i = 15$. Thus, vectors such as $(0, 0, 0, 0, 0, 0, 0)$ and $(15, 15, 15, 15, 15, 15, 15)$ are illegal. To guarantee 15 checkers, we can change the vector representation to a length 15 list representation of each checker's position.

Thus, the position "all checkers off the board" is expressible as (15, 0, 0, 0, 0, 0, 0) or 000000000000000, and "all checkers on the six-point" is (0, 0, 0, 0, 0, 0, 15) or 666666666666666. Since all checkers are identical, some positions have many representations in this form. To stop this duplication, we restrict the list to be nondecreasing from left to right. We then sort the lists lexicographically, as in Table 1.

Vector	List	Integer
(15,00,00,00,00,00,00)	000000000000000	0
(14,01,00,00,00,00,00)	000000000000001	1
:	:	:
(14,00,00,00,00,00,01)	000000000000006	6
(13,02,00,00,00,00,00)	000000000000011	7
(13,01,01,00,00,00,00)	000000000000012	8
:	:	:
(13,01,00,00,00,00,01)	000000000000016	12
(12,03,00,00,00,00,00)	000000000000111	13
:	:	:
(04,03,03,02,00,02,01)	000011122233556	8887
:	:	:
(00,00,00,00,00,00,15)	666666666666666	54263

Table 1: Lexicographical Ordering of Bearoff Positions

Since the positions are now ordered and enumerated, it would be useful to have a function that returned the number of a position given to it. This could be accomplished by a linear search through a lookup table, as above, but that would be relatively slow. Fortunately, there is a faster way.

Consider the position $E = (4, 3, 3, 2, 0, 2, 1)$, shown in Figure 2. We wish to calculate its integer representation. This will be done by counting the positions that come before E. Since E has 4 checkers on the 0-point, it must come after every position that has more than 4 checkers there. The number of such positions is $\left(\!\!\binom{7}{10}\!\!\right)$; of the 15 checkers, we put 5 on the 0-point, then distribute the remaining 10 checkers among the 7 points.

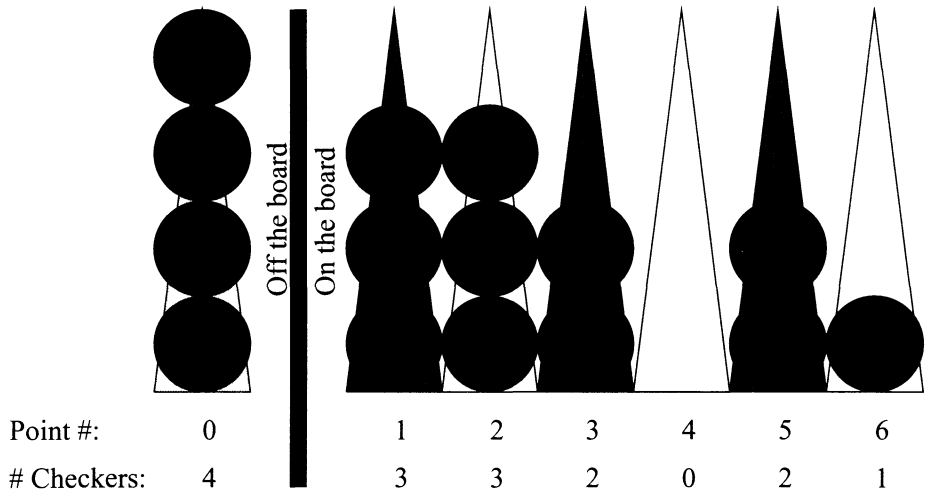

Figure 2. Position 8887

As E is after every position with more than 4 checkers on the 0-point, it is also after every position with exactly 4 checkers on the 0-point and more than 3 on the 1-point. If we put 4 on the 0-point and 4 on the 1-point, we have $7 = 15-(4+3+1)$ checkers to distribute among the six points $1 \ldots 6$, or $\left(\!\binom{6}{7}\!\right)$ possible positions. Remember, these positions have exactly 4 checkers on the 0-point, so they come between the previous "four or more" and position E. We therefore add them, and their sum is still less than the integer representation of E.

For the third point, we may calculate that there are exactly $\left(\!\binom{5}{4}\!\right)$ positions with 4 on the 0-point, 3 on the 1-point, and more than 3 on the 2-point.

Continuing this reasoning gives the sum

$$\left(\!\binom{7}{10}\!\right)+\left(\!\binom{6}{7}\!\right)+\left(\!\binom{5}{4}\!\right)+\left(\!\binom{4}{2}\!\right)+\left(\!\binom{3}{2}\!\right)+\left(\!\binom{2}{0}\!\right) = 8887$$

So 8887 is the integer representation of E. Notice that the top component decreases by one each term, and the bottom component decreases by the number of checkers on that point. This leads to a general formula for the integer representation of vector K. Define S_i as the number of possible positions that match exactly all the points before

i, and have more checkers on the i-th point than the current position does. Thus, to calculate S_i for position K, one uses the formula

$$S_i = \left(\!\!\binom{7-i}{15-(K_1+\ldots+K_i+1)}\!\!\right)$$

The integer representation of K is then the sum of the S_i terms: $\sum_{i=0}^{5} S_i$. This gives an algorithm to compute position number that runs rather quickly.

PROPERTIES OF THE ENUMERATION

Because each position's number is determined only by the number of positions that come before it, there can be no gaps in the list. So, each one of the 54264 positions has a number somewhere between 0 and 54263. This means that no memory is wasted on impossible positions.

There is a quick algorithm to convert an integer back into a position, by successively subtracting various values to find how many checkers are on point 6, then point 5, and so forth. This means that we can go from positions to integers and back again without using a large lookup table.

We mentioned before that we wanted to store information about each position. This enumeration gives us the ability to have an array of 54264 entries, each of which says something about its respective position. For example, one simple measurement of a position is its "Raw Pip Count", or RPC. This is just the sum of the distances that the checkers must travel to get off the board. For Figure 2, this is $6 + 2 \cdot 5 + 2 \cdot 3 + 3 \cdot 2 + 3 \cdot 1 = 31$. If we plot the RPC of each position versus its integer representation, we get Figure 3. This shows something of the structure of this enumeration: the positions that are close to being done (low RPC's) have low hash values, and those that are far from finishing (high RPC's) have large hash values. However, there is no simple function we could apply to approximate the RPC of a position given its hash value.

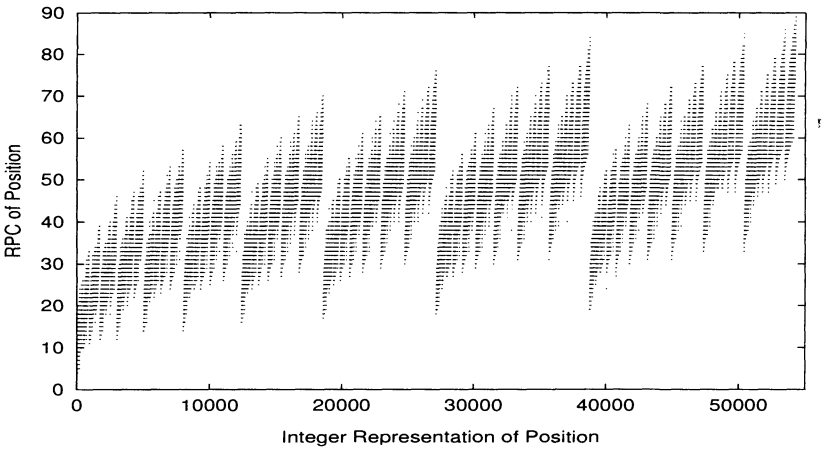

Figure 3. The raw pip counts of all bearoff positions

We wanted to keep track of how good each position is. This depends on the way the dice roll, in addition to the arrangement of the checkers. We call the value of each position its "Expected Pip Count", or EPC. These are shown in Figure 4. Again, the structure of the enumeration appears in the graph. Also, there is a general correspondence between EPC and hash value, but there is no simple function to go between the two. The EPC translates directly to the expected number of turns until the player is finished. To get the expected number of turns, one divides the EPC of the position by $49/6 \approx 8.16667$, which is the average value of a dice roll.[1]

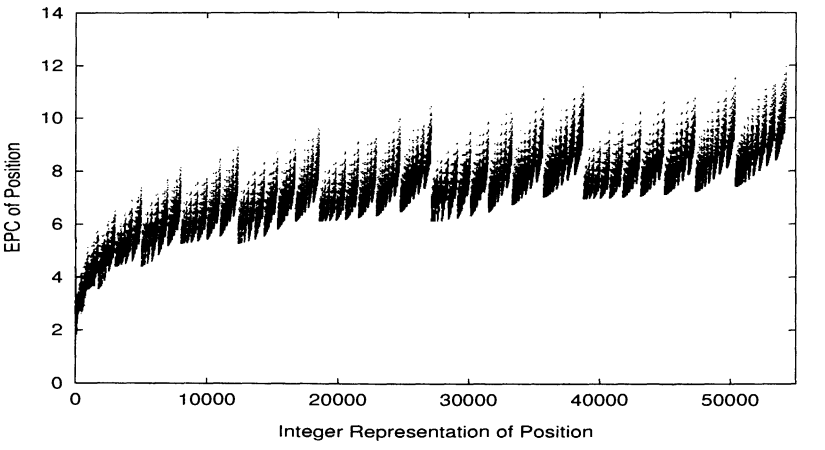

Figure 4. The expected pip counts of all bearoff positions

1. Backgammon rules give twice as much weight to rolling doubles, which is why the average is higher than the usual 7.

The EPC for any position P is calculated by looking at the EPC for all positions which can be reached from P by legal moves. This is where the enumeration shines: if position P has integer representation A, then any legal move produces an integer representation less than A. We can easily compute by hand the EPC's for six simple positions that are very near the end of the game. Then, we march up the integers, generating rolls and moves for each position and storing the EPC's as we calculate them. Conveniently, each legal move from the current position produces a position with a smaller integer representation, so we've already computed the EPC for it. Therefore, we don't have to call the EPC-calculating program recursively, since we are being smart about the order in which we calculate positions. This sort of technique is known as dynamic programming. It is very efficient and useful in all sorts of problems, from deciding which items to take on a trip (the Knapsack problem) to determining the life cycles of lizards. For an example of dynamic programming in biology, see "Dynamic Models of Behavior: An Extension of Life History Theory" by Colin W. Clark.[1]

CONCLUSIONS

Once the EPC for each of the bearoff positions has been calculated, the computer can play perfect backgammon in the bearoff. Given its current position and a dice roll, it can look at each of the positions it can move to, and choose the one with the smallest EPC. This means it minimizes the number of turns it expects to make before it finishes. It needs the hash developed here to make a compact lookup table so it can play optimal backgammon. Then, it can compute the chances of winning by playing against itself many times and keeping track of which side wins. Once the computer has calculated these winning chances, it is possible to create formulas for them which can be calculated in one's head while playing backgammon. Thus, one might not play the game any better (unless one can memorize a table of 54264 numbers), but one knows more about how to bet on it.

---

1. "Dynamic Models of Behavior: An Extension of Life History Theory" by Colin W. Clark is in *Trends in Ecology and Evolution*, volume 8, number 6, June 1993, pages 205-209.