*Programming Techniques and Data Structures*

*Daniel Sleator Editor*

# The World's Fastest Scrabble Program

**ANDREW W. APPEL AND GUY J. JACOBSON**

*ABSTRACT: An efficient backtracking algorithm makes possible a very fast program to play the SCRABBLE® Brand Crossword Game. The efficiency is achieved by creating data structures before the backtracking search begins that serve both to focus the search and to make each step of the search fast.*

## 1. INTRODUCTION

The SCRABBLE® Brand Crossword Game[1] (hereafter referred to as "Scrabble") is ill-suited to the adversary search techniques typically used by computer game-players. The elements of chance and limited information play a major role. This, together with the large number of moves available at each turn, makes subjunctive reasoning of little value. In fact, an efficient generator of legal moves is in itself non-trivial to program, and would be useful as the tactical backbone of a computer crossword-game player.

The algorithm described here is merely that: a fast move generator. In practice, combining this algorithm with a large dictionary and the heuristic of selecting the move with the highest score at each turn makes a very fast program that is rarely beaten by humans. The program makes no use of any strategic concepts, but its brute-force one-ply search is usually sufficient to overwhelm its opponent.

## 2. COMPUTER SCRABBLE-PLAYERS IN THE LITERATURE

A number of computer programs have been written to play SCRABBLE®. The best publicized is a commercial product named MONTY®[2], which is available for various microcomputers and as a hand-held device the size of a giant calculator. According to *Scrabble Players News*, it uses both strategic and tactical concepts [2]. The human SCRABBLE experts who reviewed MONTY beat it consistently, but said that it was a fairly challenging opponent.

Peter Turcan of the University of Reading in England has written a SCRABBLE-player [8, 9] for some unspecified micro-computer. It appears that he generates moves by iterating over the words in his lexicon in reverse order of length. He somehow decides for each word whether and where it can be played on the current board with the current rack. His program doesn't attempt any adversary search, but it does use an evaluation function more sophisticated than the score of the prospective move. It takes the score and conditionally adds terms depending on simple strategic features of the new position and tiles left in the rack.

[1] As used in this paper, the mark SCRABBLE refers to one of the crossword-game products of Selchow and Richter Company.

This research was sponsored in part by a grant from the Amoco Foundation, in part by an NSF Graduate Student Fellowship, in part by NSF grant MCS-830805, in part by Presidential Young Investigator grant DCR-8352081, and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

[2] Registered trademark of Ritam Corporation.

Peter Weinberger of AT&T Bell Laboratories wrote a SCRABBLE-playing program that was originally designed to work on a PDP-11 (which could not hold the entire lexicon in memory), but now operates on a VAX [10]. His move generator first constructs a set of position descriptors, one for each place on the board where a legal move might be made. For each position, he compiles information about the letters and positions that might fit; other information is compiled about the rack and board as a whole. He then looks at each word in the dictionary, discarding as many as possible by quick global tests. The remaining words are subjected to quick local tests (for each position) to discard those that don't fit at particular positions. The words that pass these tests are examined letter-by-letter at each position to see if they make legal plays. The emphasis throughout is on heuristic tests that can quickly and correctly eliminate words from consideration; this strategy was motivated by the need to sequentially access the lexicon. Weinberger's program has no concept of strategy, and simply chooses the highest-scoring play available.

Stuart Shapiro *et al.* of SUNY Buffalo have implemented several SCRABBLE-playing programs in SIMULA and Pascal on a PDP-10 [6, 7]. They represent their lexicon as a tree-structure of letters where each path down the tree has an associated list of words that can be formed using exactly those letters on the path. The letters along each path appear in a canonical order corresponding (approximately) to the point value of the tile in SCRABBLE, by decreasing value. The reason for the putting the higher-valued letters higher in the tree is to help find the most valuable words first, in case a full search cannot be completed.

Shapiro's move generator iterates over board positions, taking the tiles in the rack and at the proposed board position, and searching down from the root for acceptable words that can be formed with those letters. His programs do not generally examine all possible board positions where words could be played, only those positions judged worthwhile. Once a move is found yielding at least some pre-determined threshold score, that move is chosen.

### 2.1 Performance Comparison

The value of our algorithm for SCRABBLE move generation is its speed. Our program is faster than all the other programs we know of by two orders of magnitude. The large size of the lexicon searched leads us to suspect that our program would probably beat the programs that have smaller vocabularies.

Playing at its highest level, MONTY uses a lexicon of 44,000 words and takes about two minutes per move. Turcan's program, with a lexicon of 9000 words, takes about two minutes per move. Weinberger's program takes about a minute or two to do move generation using a lexicon of about 94,000 words. Shapiro's programs search from 1500 to 2000 words, finding a move in 30 or 40 seconds. Our program has a lexicon of 94,240 words, and takes one or two seconds to generate *all* legal moves, on a VAX 11-780.

## 3. THE ALGORITHM

Instead of scanning through the entire lexicon each turn for playable words, we begin by scanning over the board for places where a word could connect to letters already on the board. Then we try to build the word up incrementally, using letters from the rack and the board near the proposed place of attachment. We maintain data structures that make this search one-dimensional, so we never have to look outside of a single row or column of the board during move generation.

### 3.1 Reducing the Problem to One Dimension

We can classify each legal play in SCRABBLE as either *across* or *down*, depending on whether the tiles played are all in the same row or all in the same column[3]. Without loss of generality, we can restrict our attention to generating the *across* plays only, since the *down* plays are simply *across* plays with the board transposed.

3.1.1 *Cross-Checks.* When making an *across* play, the newly-placed tiles must also form *down* words whenever they are directly above or below tiles already on the board. However, since at most one tile can be added to any column of the board, it is easy to precompute (for each empty square) the set of letters that will form legal *down* words when making an *across* move through that square. Because there are only 26 letters in the alphabet, these *cross-check* sets can be represented efficiently as bit-vectors. The cross-checks can be computed before beginning the move generation phase, since they are independent of any particular *across* move. Since the number of empty squares whose cross-checks can change after any move is small, we need to recompute cross-checks for only a few squares after each move.

3.1.2 *Anchors.* Any *across* word must include some newly-placed tile adjacent to a tile already on the board.[4] Therefore, it is natural to use a place of adjacency as the spot to begin looking for a legal move. We will call the leftmost newly-covered square adjacent to a tile already on the board the *anchor* square of the word.

It is easy to decide which squares are potential anchor squares: they are the empty squares that are adjacent (vertically or horizontally) to filled squares. We call these candidate anchor squares the *anchors* of the row.

By first computing the cross-checks and the anchors for a given row, we can look for *across* words in that

[3] A single newly-placed tile that forms words in both directions is both *across* and *down*.

[4] Except for the first move, an easy special case.

row without considering the contents of any other row. The move generation problem is thus reduced to the following one-dimensional problem: given a rack of tiles, the contents of a row on the board, and the cross-checks and anchors for the row, generate all legal plays in that row.

## 3.2 Representation of the Lexicon

A variety of algorithmic techniques and heuristics are used to make our search fast. A key element of the algorithm is the representation of the lexicon.

3.2.1 *The Trie.* We represent the lexicon as a tree whose edges are labeled by letters. Each word in the lexicon corresponds to a path from the root. When two words begin the same way they share the initial parts of their paths. The node at the end of a word's path is called a *terminal* node; these are specially marked. (Notice that all leaves of the tree are terminal nodes, but the reverse need not be true.) This data structure is called a letter-tree or *trie* [3, 4]; an example is shown in Figure 1.

Our 94,240-word lexicon can be represented as a 117,150-node trie with 179,618 edges. By storing each node as a variable-sized array of out-edges, we can store the trie in space proportional to the number of edges (three or four bytes per edge).

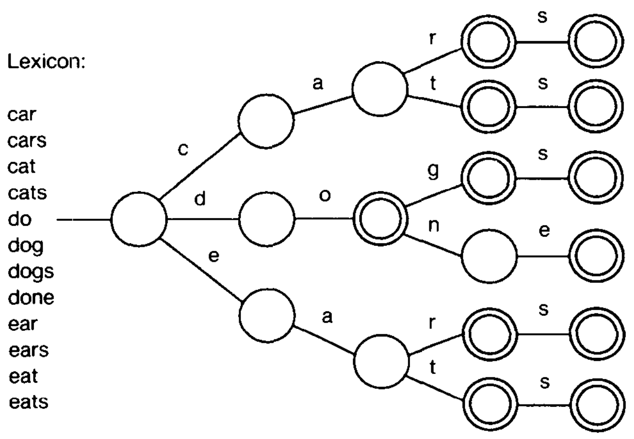

**FIGURE 1. A Lexicon and the Corresponding Trie (Terminal Nodes are Circled)**

3.2.2 *The Dawg.* The trie is rather bulky—it occupies over half a megabyte. By representing it as a graph instead of a tree, we can dramatically reduce its size without changing the move-generation algorithm at all.

The trie can be considered a finite-state recognizer of the lexicon. Nodes in the trie are the states of the finite-state machine; edges of the trie are the transitions of the machine; and terminal nodes are the accepting states.

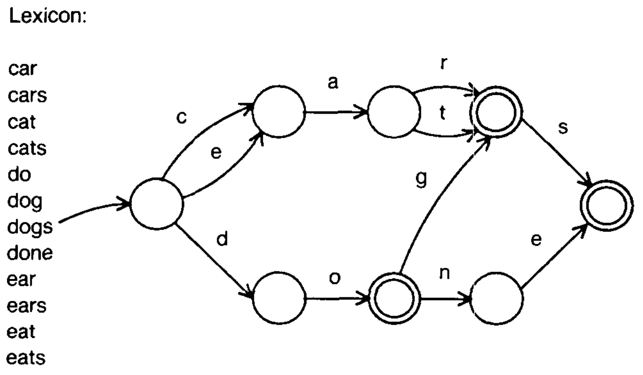

**FIGURE 2. A Dawg**

The *language* of a finite-state recognizer is the set of words that it will accept. For any language, there will be many different finite-state recognizers. In particular, there will be one with a minimum number of states. When the language contains only a finite number of words (which our lexicon certainly does), it is easy to find the minimum-size finite state recognizer quickly [5].

The minimum state recognizer will be a directed graph rather than a tree. The trie of Figure 1 may be reduced to the graph of Figure 2. Thus, a *dawg* (Directed Acyclic Word-Graph) [1] is basically a trie where all equivalent sub-tries (corresponding to identical patterns of acceptable word-endings) have been merged. This minimization produces an amazing savings in space; the number of nodes is reduced from 117,150 to 19,853. The lexicon represented as a raw word list takes about 780 Kbytes, while our dawg can be represented in 175 Kbytes. The relatively small size of this data structure allows us to keep it entirely in core, even on a fairly modest computer.

## 3.3 Backtracking

We use a simple two-part strategy to do move generation. For each anchor, we generate all moves anchored there as follows:

(1) Find all possible "left parts" of words anchored at the given anchor. (A left part of a word consists of those tiles to the left of the anchor square.)
(2) For each "left part" found above, find all matching "right parts." (A right part consists of those tiles including and to the right of the anchor square.)

The left part will contain either tiles from our rack, or tiles already on the board, but not both.

If the square preceding the anchor is vacant, we must place a (possibly empty) left part from the rack; for each such left part, we then try to extend rightwards from the anchor to form words. If the square preceding the anchor is occupied, then the contiguous tiles to the left of the anchor constitute the left part, and we can simply try to extend it rightwards with tiles from the rack.

3.3.1 *Placing Left Parts.* The left part is either already on the board or all from the rack. In the former case we compute the left part simply by looking at what's there. The latter case is nontrivial: we must find all possible left parts.

Since we defined an anchor square as the *leftmost* point of adjacency, the left part cannot extend to cover an anchor square. This puts a limit on the maximum size of a left part of a word anchored at a given square. Furthermore, the anchor squares are exactly those with nontrivial cross-checks. Thus, the squares covered by the left part all have trivial cross-check sets that allow any letter to be placed there. We can generate the left parts that can be placed before a given anchor square by doing a pruned traversal of the dawg, constrained by the tiles remaining in the rack.

Because all of the squares covered by the left part have trivial cross-check sets, we need not consider those squares when traversing the dawg—we need only to know the maximum size of the left part. This is equal to the number of non-anchor squares to the left of the current anchor square.

Here is the backtracking procedure that places left parts. It calls the procedure ExtendRight with each left part that it finds

```
LeftPart(PartialWord, node N in dawg, limit) =
  ExtendRight(PartialWord, N, AnchorSquare)
  if limit > 0 then
    for each edge E out of N
      if the letter l labeling edge E is
          in our rack then
        remove a tile labeled l from the
          rack
        let N' be the node reached by
          following edge E
        LeftPart (PartialWord · l, N',
          limit - 1)
        put the tile l back into the rack
```

To generate all moves from *AnchorSquare*, assuming that there are $k$ non-anchor squares to the left of it, we call

```
LeftPart("", root of dawg, k)
```

3.3.2 *Extending Rightwards.* We can attempt to complete the word by adding tiles to the right end one at a time, doing a pruned traversal of the sub-dawg rooted at $N$. This traversal is constrained by: the tiles remaining in the rack, the tiles already occupying the squares to the right of the anchor, and the relevant cross-checks.

In extending rightwards, we may find tiles already on the board. This does not terminate the search, as a legal move may include previously-placed tiles sandwiched between newly-placed tiles. Instead, we simply include these tiles in the newly-placed words, when possible.

Assume we have a procedure LegalMove that takes a legal play and records it for consideration. (A simple LegalMove procedure might simply keep track of the highest-scoring move, and discard the rest.) We can express the backtracking search as a recursive procedure ExtendRight:

```
ExtendRight(PartialWord, node N in dawg,
    square) =
  if square is vacant then
    If N is a terminal node then
      LegalMove(PartialWord)
    for each edge E out of N
      if the letter l labeling edge E is
          in our rack and
        l is in the cross-check set of
          square then
          remove a tile l from the rack
          let N' be the node reached by
            following edge E
          let next-square be the square to
            the right of square
          ExtendRight (PartialWord · l, N',
            next-square)
          put the tile l back into the
            rack
  else
    let l be the letter occupying square
    if N has an edge labeled by l that
        leads to some node N' then
      let next-square be the square to
        the right of square
      ExtendRight (PartialWord · l, N',
        next-square)
```

Now that we can place left parts and extend them rightwards, we have a complete algorithm to generate all the legal moves. It is easily seen that the algorithm outlined above generates each *across* move exactly once.

## 3.4 Some Details

The above description skips over a few details in the move generation algorithm that are not crucial to a basic understanding. In this section we address some of the particulars.

3.4.1 *Empty Prefixes.* When there is a tile already on the board to the left of an anchor square, no prefix is placed. Instead, we can call ExtendRight directly, starting from the node in the dawg corresponding to the partial word to the left of the anchor square.

3.4.2 *Blanks.* The problem of move generation in the SCRABBLE game is complicated by the presence of blank tiles, which may represent any letter. To deal with them, we must add a little extra code to our LeftPart and ExtendRight procedures. Whenever we look through our rack for some letter, we also check to see if we have a blank. If we do have a blank, we may use it, temporarily letting it represent the letter we seek. When we backtrack and pick it up off the board, the blank regains its polymorphic properties.

The presence of blank tiles in the rack greatly increases the number of moves possible at a given turn. The time spent in searching increases accordingly. In fact, it is almost always possible to tell when our program holds a blank tile by the noticeable delay before a move is made. When the program gets both blanks simultaneously, it seems to slip into a coma for a few seconds. Fortunately, the number of blanks in the pool is small, so most of the time there are no blanks in the rack to contend with. It is rare indeed to hold both of them at once.

3.4.3 *Data Structures.* In this section, we describe the data structures in more detail. The major data structure is the dawg, which we store as a very large array of edges. All the edges out of a given node occupy a contiguous sub-array of the large array; a node is referenced by the index in the large array of the first edge out of that node. Each edge stores a letter labeling the edge (5 bits), a reference to the node reached by following the edge (16 bits), a bit indicating if the edge is "terminal," and a bit flagging the last edge in the sub-array that constitutes a node.

Note that the terminal bit-flag (indicating when a complete word is formed) is stored per edge, rather than per node. Thus, some edges coming into a node could consider it a terminal node, while others coming into the same node would not consider it terminal. This makes the dawg more compact, since more shared list structure is possible.

We need a total of 23 bits to store each edge in the array, so an edge can conveniently fit into a 32-bit machine word (or if more space efficiency is desired, into three 8-bit bytes). The unique node with no out-edges is given an index of zero by convention.

Because there are only 26 letters in the alphabet, we can conveniently store each cross-check set as a bit-vector in one 32-bit machine word, and do membership testing quickly. The rack is stored as an array of 27 small integers giving the quantity of each tile type (26 letters and the blank) present. This allows fast membership testing, addition, and removal of tiles.

### 3.5 Loose Ends

It is useful to put fictitious squares with empty cross-check sets at the end of each row to serve as sentinels, preventing us from trying to extend words off the board.

Although we have described how move generation can be reduced to one dimension, we might still have to look outside the current row to compute the scores of the moves generated. To avoid this, we compute for each empty square the sum of the values of all tiles in contiguous sequence above and below that square before beginning move generation. This extra information is enough to allow us to compute the scores of moves one-dimensionally. We conveniently compute these *cross-sums* while computing the cross-check sets.

## 4. THE PROGRAM

In 1983 we wrote a program that implements our algorithm and referees games between the computer and a human opponent. The program consists of about 1500 lines of code, and is written in the C programming language. It was written on a VAX running the UNIX operating system and was subsequently ported to a Sun workstation and an Apple Macintosh.[5]

### 4.1 Program Statistics

In a test series of 10 games in which the program played against an opponent that passed at each move, 224 moves were made at an average computation time of 1.4 seconds per move (including the time for redrawing the graphic display) on a VAX 11/780. An average of 450 legal moves were found per turn, although the number varied enormously from turn to turn. In a test match where the program played against itself for 10 games, it had an average final score of 377 per player.

## 5. WHY IS IT FAST?

The efficiency comes primarily from the high-yield backtracking search strategy. By using a backtracking algorithm with the dawg as our guide, we never consider placing a tile that isn't part of *some* word *on the board* (though we might not be able to complete the word). Furthermore, the placement of one tile may lead to the generation of several moves before that tile is picked up again. The checks for tiles in the rack and for legal cross-words prune the search. Because of this pruning, most of the words in the dictionary are never even examined in a typical turn.

There is also an implicit pruning in starting the search from the anchor squares, because this guarantees that the word will be adjacent to tiles already on the board. If we started searching rightwards from each square where a word could conceivably start, we would find that most of the time we would fail to connect the partial word to tiles already on the board.

Through the abstractions of anchor squares and cross-check sets, we reduce a two-dimensional problem into one-dimension. By precomputing the cross-check sets before beginning move generation, we can do all of the pruning operations in constant time. There are few special cases in the resulting algorithm, making it easy to code efficiently.

Finally, an important advantage of the dawg representation of the lexicon is its compactness. This allows us to keep the entire lexicon in primary memory during move generation, which avoids costly I/O.

[5] VAX is a trademark of Digital Equipment Corporation. UNIX is a trademark of AT&T Bell Laboratories. Sun is a trademark of Sun corporation. Apple and Macintosh are trademarks of Apple Computer Corporation.

## 6. APPLICATIONS

Although most of the work described here is not generally useful for anything except playing the SCRABBLE game, we have used the dawg data structure in other programs. A dawg should be considered any time a search through a large lexicon is needed. We have written multi-word anagram finders, an acrostic solver's assistant, and a code breaking tool using our dawg. The same data structure would be useful in playing many other word games, and could be used in a spelling checker.

## 7. FUTURE WORK

Our program is merely blindingly fast, not outstandingly good. It will make the same moves as any program that simply picks the highest-scoring move each turn from the same lexicon (as Weinberger's program does, albeit more slowly). In practice, this performance is enough for resounding victories over almost all human players it has faced.

### 7.1 A Two-Way Dawg

There are several ways in which our algorithm might be speeded up. One idea considered was changing the representation of the lexicon to allow extension of partial words leftwards as well as rightwards. We would use a "two-way" dawg, each of whose nodes corresponded to the middle sections of words. A node would have out-edges for each letter that could be appended to the beginning as well as the end of the partial word corresponding to that node.

Suppose we were about to start generating moves from some anchor square that had a tile immediately to its right. The search would start from the node in the two-way dawg corresponding to the sequence of tiles to the right of the anchor square. We would never build a partial word that could not be extended into *some* legal word. Currently, using our rightward-only dawg, we sometimes try to place many prefixes that (because of tiles already on the board to the right of the anchor square) cannot be completed to form words. Furthermore, with a two-way dawg we can choose which direction to extend a partial word; choosing the more-constrained direction would help to prune the search.

Because a two-way dawg has a node for each substring of each word, we imagine that it would be a great deal larger than the one-way dawg, although we have never tried to build one. It would be interesting to see just how much space it would occupy, and by how much it would speed up move generation.

### 7.2 Looking at Fewer Left Parts

Because there are no cross checks involved in searching for left parts, there is less pruning in this part of the search than in extending rightward. Our algorithm often places left parts that cannot be extended by even a single tile. We could move the anchor-square cross-check pruning to the beginning of left-part generation (where it will prune larger subtrees) by first making a list of all possible left-parts and arranging them by last letter and length. Then, for all anchor squares, we could start extending rightward from just those left parts which fit.

This might take a lot of storage, as there can be tens of thousands of left parts in the worst case. But we could now invert the order of iteration nesting to save space: we could make the iteration over left parts the outer loop, and the iteration over anchor squares the inner loop (with the backtracking over right parts inside that loop). To do this efficiently, it would be helpful to first make a list of anchor descriptors, keeping for each one the maximum left-part length and the anchor cross-check set.

It is possible to calculate empirically—using the dawg and a large set of representative game positions—the approximate savings this would yield, since we can see what effect it would have to test the cross-check set of the anchor square before looking for left parts. By these methods, we estimate that the algorithm would be made about 30 percent faster with this modification.

### 7.3 Adversary Search

One tantalizing possibility, given a fast move generator, is to attempt some form of adversary search. When the pool is exhausted and all tiles are either in the racks or on the board, SCRABBLE becomes a game of perfect information. In theory, we could simply use our move generator to search out the entire game tree; in practice, the branching factor makes this prohibitively expensive. However, we could conceivably get around this high expense by using the same sorts of techniques commonly used by adversary search algorithms. These include alpha-beta pruning, hashing of board positions, and using approximate evaluation functions to discard obviously bad moves.

Even in the middle of the game, where real adversary search is impossible, some form of "sampled" search might give the program some concept of strategy. For example, we could do a two-ply search, giving the opponent several different randomly selected racks. This would help us determine when a proposed play for us leaves the board too "open" for the opponent.

### 7.4 Evaluation Function Heuristics

Another way to improve the play of the program is to use a more sophisticated evaluation function. Some of the programs described in Section 2 use heuristics to improve their move evaluation, rather than just use the point count as an evaluation function. These heuristics include:

- Preserving tiles and tile combinations in the rack that are likely to lead to higher scores in the next move(s).
- Avoiding leaving the board open for the opponent to make high-scoring moves.

A variety of heuristics have been tried by other authors. It would be relatively easy, given our fast move generator, to experiment with these and other heuristics to significantly improve the performance of the program.

## APPENDIX: An Illustrative Game

To illustrate the effectiveness of a brute-force single-ply search in Scrabble, we include a sample game where the Macintosh version of our program faced Guy Jacobson. Guy took several minutes for each move; the Mac took several seconds. Guy played first.

| | *Guy* | | *Mac* | |
|---|---|---|---|---|
| 1 | ALACK | 32 | AJEE | 25 |
| 2 | OUTGREW | + 66 = 98 | HYALINE | + 51 = 76 |

So far, Guy is outscoring the program. His lead will not last. The G in OUTGREW is a blank; note that the tile bears no number.

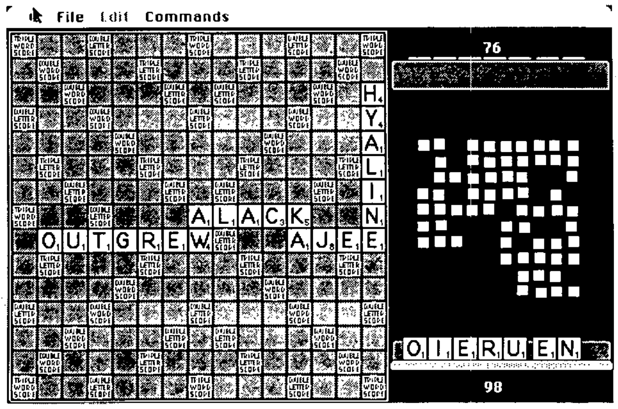

**FIGURE 3. Guy Enjoys an Early Lead**

| | *Guy* | | *Mac* | |
|---|---|---|---|---|
| | | 98 | | 76 |
| 3 | JUNIOR | + 26 = 124 | FENCES | + 50 = 126 |
| 4 | BATH | + 18 = 142 | RITZ | + 46 = 172 |
| 5 | ZEDS | + 42 = 184 | SLOGGING | + 82 = 254 |

The program paused noticeably before playing SLOGGING across two double word scores because it had a blank. Note the aggressive, open style of the program here. It cheekily plays a Z in line with a triple word score. Obviously, it has no concept of defensive strategy.

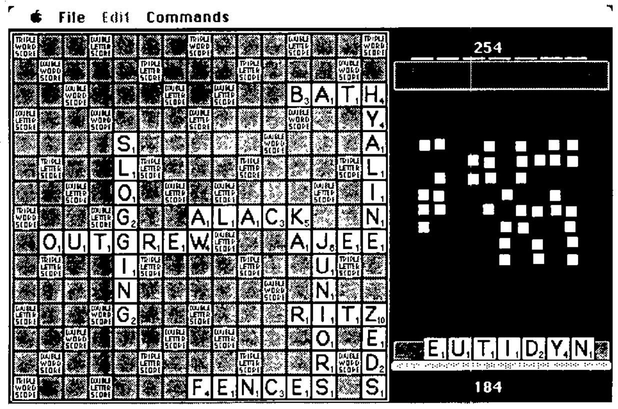

**FIGURE 4. The Tables Have Turned**

| | *Guy* | | *Mac* | |
|---|---|---|---|---|
| | | 184 | | 254 |
| 6 | YIELD | + 21 = 205 | VAGUE | + 34 = 288 |
| 7 | RUNTIER | + 66 = 271 | SONDE | + 27 = 315 |
| 8 | ME | + 19 = 290 | PAVAN | + 22 = 337 |
| 9 | EON | + 19 = 309 | QUITTOR | + 32 = 369 |
| 10 | TWO | + 26 = 335 | XI | + 50 = 419 |
| 11 | ROILY | + 16 = 351 | PAH | + 31 = 450 |
| 12 | BI | + 16 = 367 | HOD | + 14 = 464 |
| 13 | SI | + 5 = 372 | (*stuck with* FM) | |
| | | + 7 = 379 | | − 7 = 457 |

The program finishes Guy off handily by a score of 457 to 379. It has a much larger vocabulary, and seems to find ways to score even with inferior tiles. Note that the program is stuck with tiles in its rank at the end of the game because it doesn't plan ahead.

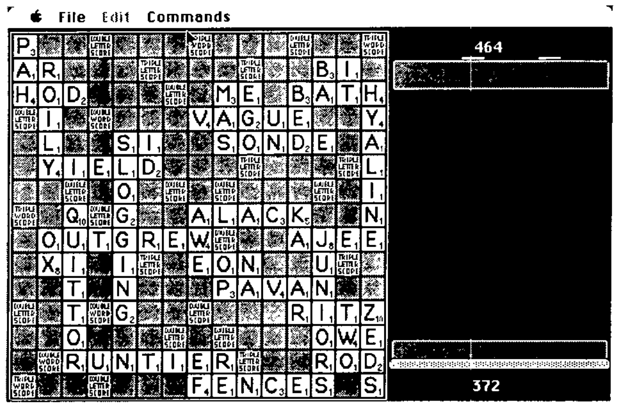

**FIGURE 5. After Guy's Last Move**

(*continued on p. 585*)

*Acknowledgments.* The author would like to express his gratitude to the referees for their helpful suggestions.

**REFERENCES**
Note: Reference [6] is not cited in text.
1. Adel'son-Vel'skii, G.M., and Landis, E.M. An algorithm for the organization of information. *Dokl. Akad. Nauk USSR, 146,* 2 (1962), 263–266.
2. Bentley, J.L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18,* 9 (Sept. 1975), 509–517.
3. Chang, H., and Iyengar, S.S. Efficient algorithms to globally balance a binary search tree. *Commun. ACM 27,* 7 (July 1984), 695–702.
4. Day, A.C. Balancing a binary tree. *Comput. J. 19,* 4 (Nov. 1976), 360–361.
5. Gonnet, G.H. Balancing binary trees by internal path reduction. *Commun. ACM 26,* 12 (Dec. 1983), 1074–1081.
6. Karlton, P.L., Fuller, S.H., Kaehler, E.B., and Scroggs, R.E. Performance of height-balanced trees. *Commun. ACM 19,* 1 (Jan. 1976), 23–28.
7. Knuth, D.E. *The Art of Computer Programming,* vol. 1, *Fundamental Algorithms.* Addison-Wesley, Reading, Mass. 1973, pp. 399–404.
8. Knuth, D.E. *The Art of Computer Programming,* vol. 3, *Sorting and Searching.* Addison-Wesley, Reading, Mass. 1973, pp. 422–427.
9. Martin, W.A., and Ness, D.N. Optimal binary trees grown with a sorting algorithm. *Commun. ACM 15,* 2 (Feb. 1972), 88–93.
10. Nievergelt, J., and Reingold, E.M. Binary search trees of bounded balance. *SIAM J. Comput.* 2, 1 (1973),˙33–43.
11. Stout, Q.F., and Warren, B.L. Tree rebalancing in optimal time and space. *Commun. ACM 29,* 9 (Sept. 1986), 902–908.

**CR Categories and Subject Descriptors:** E.1 [**Data Structures**]: Trees; F.2.2 [**Nonnumerical Algorithms and Problems**]: Sorting and searching.
**General Terms:** Algorithms
**Additional Key Words and Phrases:** Balanced trees, binary search tree, internal path length

Received 8/85; revised 3/87; accepted 12/87

Author's Present Address: Thomas E. Gerasch, SPARTA, Inc., 7926 Jones Branch Drive, Suite 1070, McLean, VA 22102.

**Appel and Jacobson** *(continued from p. 578)*

**REFERENCES**
1. Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., and McConnell, R. Linear size finite automata for the set of all subwords of a word; an outline of results. *Bul. Eur. Assoc. Theor. Comp. Sci.* 21 (1983), 12–20.
2. Cosma, J., Jackson, D. et al. Introducing MONTY plays scrabble. *Scrabble Players News,* (June 1983), 7–10.
3. de la Briandais, R. File searching using variable-length keys. *Proceedings of the Western Joint Computer Conference,* 1959, pp. 295–298.
4. Fredkin, E. Trie Memory. *CACM* 3, 9 (Sept. 1960), 490–500.
5. Nerode, A. Linear automaton transformations. *Proc. AMS* 9 (1958), 541–544.
6. Shapiro, S.C. A scrabble crossword game playing program. *Proceedings of the Sixth IJCAI,* 1979, pp. 797–799.
7. Stuart, S.C. Scrabble crossword game playing programs. *SIGArt Newsletter,* 80 (April 1982), 109–110.
8. Turcan, P. Computer scrabble. *SIGArt Newsletter,* 76 (April 1981), 16–17.
9. Turcan, P. A competitive scrabble program. *SIGArt Newsletter,* 80 (April 1982) 104–109.
10. Weinberger, Peter. private communication.

**CR Categories and Subject Descriptors:** E.1 [**Data Structures**]: General—*trees*; *graphs*; *tries*; H.3.1 [**Information Storage and Retrieval**]: Context Analysis and Indexing—*dictionaries*; I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems—*games*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*backtracking*; *graph and tree search strategies*
**General Terms:** Algorithms

Authors' Present Addresses: Andrew W. Appel, Department of Computer Science, Princeton University, Princeton, NJ 08544; Guy J. Jacobson, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213.