A SCRABBLE CROSSWORD GAME PLAYING PROGRAM*

Stuart C. Shapiro

Department of Computer Science

State University of New York at Buffalo

Amherst, New York 14226

&

Howard R. Smith

Department of Computer Sciences

The University of Texas at Austin

Austin, Texas 78712

Keywords and Phrases: SCRABBLE Crossword Game, game playing, lexicon, dictionary, letter table, trie, data structures, searching.

CR Categories: 3.64, 3.66, 3.74

---

*The work reported herein was done while both authors were at the Computer Science Department, Indiana University, Bloomington, Indiana 47401.

Abstract

A program has been designed and implemented in SIMULA 67 on a DECSystem-10 to play the SCRABBLE Crossword Game interactively against a human opponent. The heart of the design is the data structure for the lexicon and the algorithm for searching it. The lexicon is represented as a letter table, or trie using a canonical ordering of the letters in the words rather than the original spelling. The algorithm takes the trie and a collection of letters, including blanks, and in a single backtrack search of the trie finds all words that can be formed from any combination and permutation of the letters. Words using the higher valued letters are found before words not using those letters, and words using a collection of letters are found before words using a sub-collection of them. The search procedure detaches after each group of words is found and may be resumed if more words are desired.

1. Introduction

The SCRABBLE Crossword Game is a well known game considered to require a fair amount of intelligence, strategic and tactical skill, facility with words, and luck. Unlike most games in the artificial intelligence literature, it can be played by two or more players and is neither a zero sum game nor a game of perfect information. For these reasons, game tree searching procedures do not seem applicable. When humans play the game, a large easily accessible vocabulary seems to be the most important determiner of victory. One might, therefore, think that it would be easy to write a program that plays the SCRABBLE Crossword Game at the championship level. However, several issues are not so clear: How should the lexicon be organized for maximum usefulness?; How should a program decide where to play?; How can a program take advantage of the small literature on the strategy and tactics of the game (e.g. [1])?; What is the relative importance of a good memory for words vs. skillful decisions about what letters to use or not to use and where to play?

Our interest has focused on the organization of the lexicon. This lexicon differs from those usually used in natural language processing programs because of the use to which it is to be put. Usually one is confronted with a possible word. One must determine if it is a word, and, if so, segment it into affixes and stem, and retrieve lexical information associated with the stem. One must allow for the possibility that the word is lexically ambiguous and that the segmentation can be done in several different ways. The problem for the SCRABBLE Crossword Game lexicon is, given a set of letters, find all the words that can be made from any combination and permutation of them. This is a very different problem, but similar to the information retrieval problem of,

given a set of keys find all records that contain any subset of them, or the pattern recognition problem of, given a set of features, some of which may be spurious, find all objects that might have given rise to them.

We have designed a lexicon and an algorithm for searching it. Around them, we have designed and implemented a program that plays the SCRABBLE Crossword Game interactively at a competitive, human level. We chose SIMULA 67 [2,3,4] for the programming language for reasons to be discussed below. In the sections below, we first briefly describe the program organization and the board evaluation technique, then we discuss the lexicon and the lexicon search algorithm in more detail.

2. Program Organization

Because our major concern was with the lexicon and associated algorithms, the rest of the program is the minimal required to play a reasonable game. We will describe the program as it is, and in the final section of this paper indicate desirable enhancements.

The program is written to play a two person game - the program player against one human opponent. Figure 1 shows the overall organization of the major program modules, which we shall describe below.

2.1 The Game Manager

The game manager handles interaction with the human opponent and keeps track of all game information. It deals out tiles to the human opponent and the program player at the beginning of the game and after each play. Dealing is done randomly (without replacement) from a list initialized correctly at the beginning of the game. The seed of the pseudo-random number generator is chosen by the human at the start of

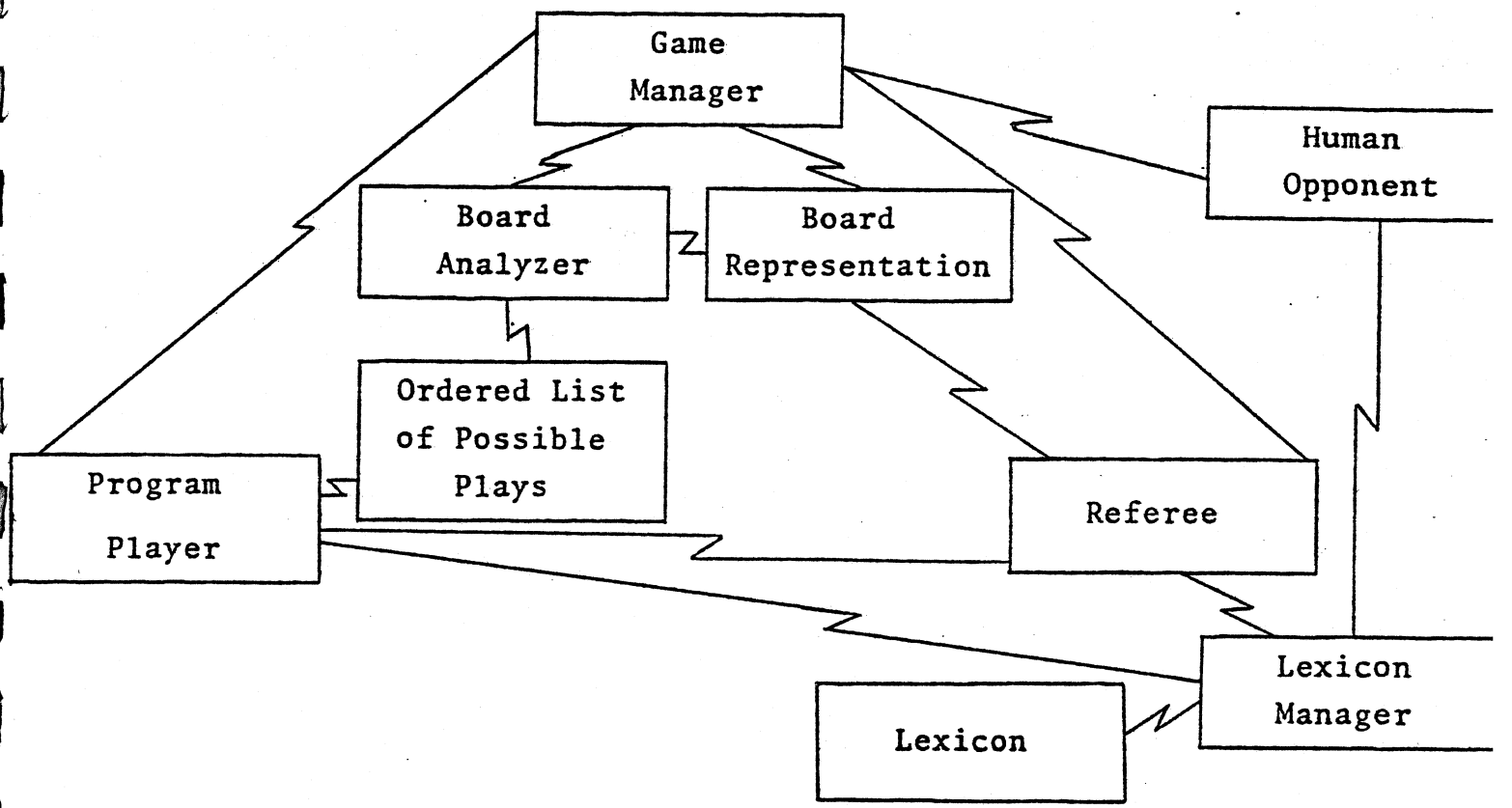

FIGURE 1: Organization of Major Program Modules

the game so that a game can be repeated to see the effect of learning new words. Whether the program or the human plays first is decided by randomly drawing a tile for each, according to the rules. The game manager's basic cycle is as follows:

1. Print out the current board.
2. Print out the human's rack and request an instruction. (The human's options at this point are described in Section 2.3).
3. If the human specifies a word to play, pass it to the referee for checking and scoring.
4. If the referee says that the play is illegal (see Section 2.5), print an appropriate message and go to step (2).
5. If the move was legal, update the board representation and the human's score, deal the human new tiles to replace the ones just used, print the score for the move just made and the new rack so the human can start thinking about the next move.
6. Pass the human's move to the board analyzer for updating the list of possible plays.
7. If the inspect made is on, print the program player's rack.
8. Get the program's move, update the board and the program's score, print the move and the score of the move.
9. Pass the program's move to the board analyzer.
10. Go to step (1).

The game ends when the human so specifies in step (2). Each player's total score is decreased by the value of the tiles in his/her rack,

and the final scores are printed.

2.2 The Board Representation

The board is represented as a 15 x 15 array of SQUAREs. In interaction with the human, the rows are numbered 1-15 and the columns A-O, in the official manner [1,pp33-4]. Each SQUARE has a value as shown below:

Kind of SQUARE	value
normal	0
double letter	1
triple letter	2
double word	3
triple word	4

If a SQUARE is occupied by a tile, it also contains the tile, an indication of whether the word using that tile is horizontal or vertical, and the value of a play using that tile as assigned by the board analyzer. Figure 2 shows how the board is printed before any play has been made.

2.3 The Human Opponent

When it is the human's turn to move, the following options are available:

0. terminate the game
1. play a word
2. pass without exchanging tiles
3. pass and exchange some tiles
4. have the total scores of both players printed

```
   A B C D E F G H I J K L M N O
 1 4     1       4       1     4   1
 2   3       2       2       3     2
 3     3       1   1       3       3
 4 1     3       1       3     1   4
 5           3               3     5
 6   2       2       2       2     6
 7     1       1   1       1       7
 8 4     1       3       1     4   8
 9     1       1   1       1       9
10   2       2       2       2     10
11           3               3     11
12 1     3       1       3     1   12
13     3       1   1       3       13
14   3       2       2       3     14
15 4     1       4       1     4   15
   A B C D E F G H I J K L M N O
```

FIGURE 2: How the board looks to the human before any words are played.

5. have the board and human's rack printed

6. access the lexicon manager (see Section 2.7)

7. get general instructions for interacting with the program

8. get this list of options.

When the human plays a word, a notation close to the official notation of the SCRABBLE Crossword Game PLAYERS [1,pp33-4] is used. The word is typed in with each letter already on the board preceeded by a "$". When a blank tile is played, a "#" is typed followed by the letter it is being used as. For example, the play, $F#IRM, means that the player is using a blank, an R and an M from the rack and an F from the board, is using the blank as an I, and is spelling the word, "firm". The location of the word is indicated by the official notation. For example, to place a word horizontally on row 12 from column K to column N, the notation is 12K-N. To place a word vertically on column C from row 13 to row 15, the notation is C13-15.

2.4 The Program Player

There are two basic ways to choose a play: 1) pick a place to play, then find a word that will fit; 2) pick a word to make, then find a place to put it. In either case, one might assign high priority to using or to not using certain tiles in the rack in order to keep the rack balanced or to build toward being able to make a bonus word (using all seven tiles on the rack). The second method is easier to use if the word chosen is made entirely from tiles in the rack, than if tiles from the board are also to be used (see Appendix, moves 3, 17 and 29). This is most appropriate when a bonus word can be made entirely from the rack. We decided to use the first method.

We adopted the following simplified approach. Possible places to play are tiles already on the board. If a tile is already being used in a horizontal word, only a vertical play will be considered and vice versa. This means that often valuable moves like 3, 17, 19, 21, 33, 35 and 39 in the Appendix will not be considered by this version of the program.

In certain situations, choice of a tile to play on puts obvious (to humans) limitations on possible words. For example, if we decide to play on a K in column N only words with a K as last letter or second to last letter are possible. One might decide to build such limitations into the lexicon search algorithm. However, this complicates situations where we decide to play on a centrally located tile. We decided to ignore such limitations in the lexicon lookup and use the referee to check each word found to see if it really fits where the player wants to put it.

The program player uses three parameters, MAXLOC, MAXWDS, and MINPT. It considers the best MAXLOC positions from the ordered list maintained by the board analyzer. For each position, it passes the tiles in its rack plus the tile at the position to the lexicon search algorithm, and considers the first MAXWDS words found. Each word is passed to the referee along with its proposed location. The referee determines if the word really fits there, and if so, returns a score for the play. As soon as a play worth at least MINPT points is found, it is played. If no such play is found, the highest scoring play is made. If none of the MAXLOC x MAXWDS words are playable, the player exchanges all the tiles on its rack for new ones. Of course, if fewer than seven tiles are left to draw from, this exchange is illegal and the program just passes. In the game recorded in the Appendix, MAXLOC

was 6, MAXWDS was 5 and MINPT was 14. Of the program's 25 moves, 6 were worth MINPT or more, 10 were worth fewer than MINPT, the program exchanged its rack 7 times, and passed twice. The average score for the 16 words actually played was 13.125. The average score for the human's 25 words was 13.88.

When the inspect mode is on, all positions and words the program player considers are printed on the terminal.

2.5 The Referee

The referee is called by the program player with a proposed move and by the game manager with the human's move. In either case, a word and a location are passed to the referee, which checks the following (not necessarily in the order shown):

1. The location and the length of the word are consistent.
2. The word will not extend off the board.
3. All letters except those preceeded by "$" are on the player's rack.
4. The word contains at least one tile from the board or is adjacent to a tile on the board in at least one place.
5. The location places each letter preceeded by a "$" where such a tile is already on the board and every other tile on an empty square.
6. The two ends of the word are not adjacent (in the direction of the word) to non-empty squares.

If the referee is judging the human's move, it passes the word and all cross-words formed by the play to the lexicon search routine. If one of these words is not in the lexicon, the human is asked if it

really is a word. If the human responds "yes", the word is given to the lexicon manager for immediate insertion into the lexicon. If the response is "no", the current version of the program allows the play anyway.

If the play is legal, the referee computes its score, including the main word and all newly formed cross-words, and returns this to the caller.

2.6 The Board Analyzer

The board analyzer is called by the game manager after each move to update the values of the playable tiles and the ordered list of these positions. The analyzer begins by setting to 0 the value of each tile already on the board that has now been used in both directions. It then computes a value for each tile newly placed on the board by the play. If the play was vertical (horizontal), the value is the sum of the value of the square and the value of each unoccupied square in the same row (column) except that no square is included that is more than seven squares away from the tile being evaluated or that is beyond the closest occupied square. Those occupied squares are then reevaluated and the ordered list of playable positions is updated for the program player's next move. The values calculated by the board analyzer are stored in each evaluated SQUARE as mentioned in Section 2.2. For example, Figure 3 shows a section of the board after placing REA$L at 12A-D. The value of the L at 12D is changed to 0, the values of the R, E, and A in row 12 are set to 9, 8, and 4 respectively, and the W at 9C is re-evaluated from 11 to 8.

We realize that the board evaluation scheme needs improvement, but it does provide a rough approximation to an adequate ordering of the

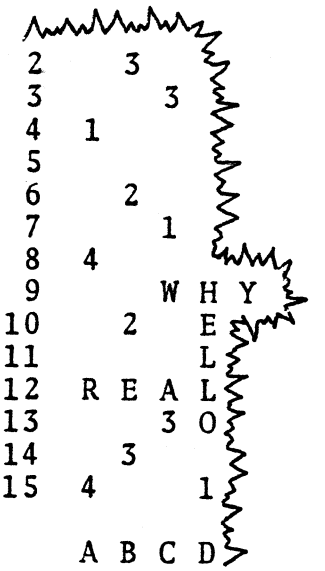

FIGURE 3: A section of the board after playing REA$L at 12A-D

playable positions and we were more concerned with the lexicon, to which we now turn.

2.7 The Lexicon Manager

The lexicon manager can perform the following operations:

1. Insert a word into the lexicon
2. Delete a word from the lexicon
3. Check if a particular word is in the lexicon
4. Find words that can be formed from any permutation and combination of a set of tiles.

The referee uses operation 3 to check the human's plays and operation 1 when the human insists that a word that isn't in the lexicon should be. The referee also uses operation 3 to check cross-words formed by plays proposed by the program player. The program player uses operation 4 to find possible words to play. The human can interact with the lexicon manager by choosing option 6 (see Section 2.3). He/She can then repeatedly use options 1, 2 or 4 before returning to the game. This option has been used by the authors for early editing of the lexicon. For example, we entered all legal two letter words [1,p.156] this way.

Option 4 takes advantage of the SIMULA 67 detach facility by taking a set of tiles and a number, n. It returns approximately n words (see below), and then can either search the lexicon using a different set of tiles or can resume the search for more words using the original set of tiles. The availability of detach was the main reason for choosing SIMULA 67 as the programming language.

3. The Lexicon Data Structure

The lexicon was designed for the particular problem, "given a collection of letters, find all words that can be formed from any combination and permutation of the letters, and find these words in approximately the order of their value in the SCRABBLE Crossword Game." Two key ideas were combined in the design of the data structure - letter tables [5, 6, 8], or tries [7], and canonical ordering.

A letter table is a binary tree in which each node has the following four fields: 1) a letter; 2) information depending on the application; 3) a success link to a subtree; 4) a fail link to a subtree. To look up a word in a letter table one starts at the root and follows fail links until a node is found whose letter is the first letter in the word. Then one looks up the remainder of the word in the success subtree of that node. When a node is found for the last letter in the word, the information field contains information about that word, e.g. lexical features. Note that as a word is looked up, all words formed from initial sub-sequences of the word are also found.

Consider the problem of designing a human readable anagram lexicon to find all words that can be formed from permutations of a given collection of letters. Take any word to be listed and reorder its letters according to any canonical ordering, e.g. alphabetical. List these re-spellings lexicographically based on the same ordering, and under each re-spelling list the original word. For example, to find anagrams of "live", look up "eilv", and find "evil", "live", and "vile".

Combining these two ideas, the data structure of our lexicon is a letter table using a canonical ordering of letters instead of the original spelling order. The information field of each node contains a list of the words formed from permutations of the letters which access

the node. So that words can be found in approximately the order of their values in the SCRABBLE Crossword Game, the canonical ordering used is the value of the letters in that game. Among letters with the same value, the one with the least frequency in English is sorted first to increase the bushiness of the tree. The ordering used is:

ZQJXKVWYFHBPMCGDULSIRNOATE

Figure 4 shows a small lexicon using this data structure. The fields are shown as indicated below.

letter	words
fail	succ

Each node is numbered for reference in Section 4.

The lexicon is maintained in a sequential file on disk, ordered in the same way it is searched: letter field, words field, success subtree, fail subtree. This file is scanned as the tree is searched and subtrees that are not needed are not stored in main memory.

4. The Lexicon Search Algorithm

The search algorithm takes the lexicon tree and a list of letters in the canonical order. The list of letters may contain one or more occurrences of the blank, "#", which may match any letter and are initially placed at the front of the list. The search algorithm finds all words in the tree that are formed from any permutation and combination of the letters in the list. It finds words using the higher valued letters before words not using them. It also finds words using a collection of letters before words using a proper sub-collection of those letters. For example, if a word can be formed using all the letters

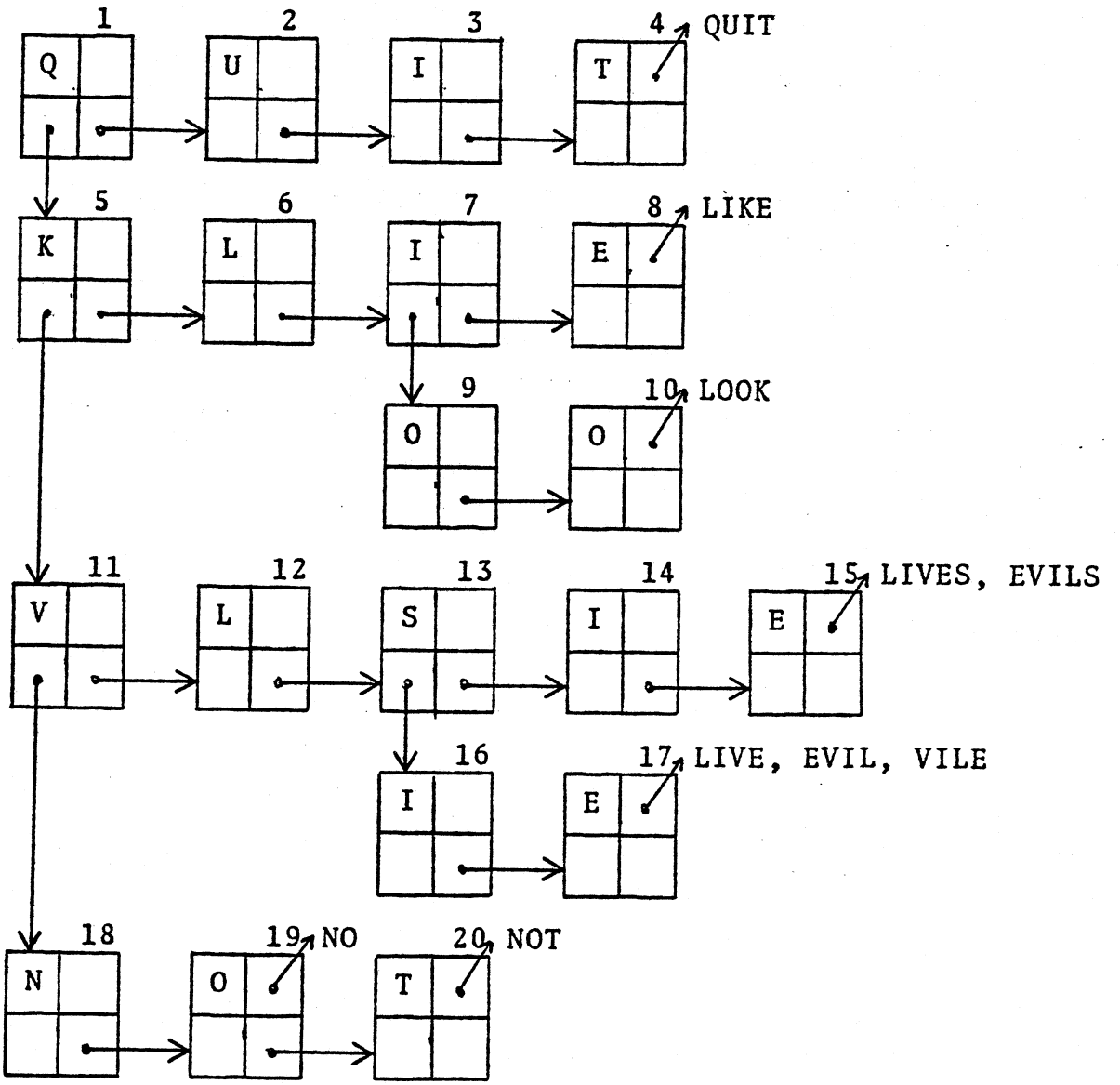

FIGURE 4: An example of the lexicon data structure

in the list, this will be the first word found.

In the algorithm presented below, the procedure detach is to be understood as causing control to be returned to the original caller of the procedure in such a way that the current value of the local variables are available to the original caller, and so that the procedure may be resumed at the point right after the detach. Other variables and functions used are:

tree: a node of the lexicon tree

letters: a list of letters ordered as discussed above

nil: the empty list and the empty tree

letter(tree)
words(tree)
succ(tree)
fail(tree) } : the four fields of tree

hd(letters): the first letter on the list

tl(letters): the list without the first letter

next-let(letters): the first non-# letter on the list, or nil if there is none

move-to-front(letter, letters): equal to letters, but with the first occurrence of letter moved to the front of the list

before(letter1, letter2): true if letter1 is before letter2 in the canonical ordering or if letter2 is nil, false otherwise

remove-next-let(letters): equal to letters with the first none-# letter removed.

The search algorithm is:

```
procedure findwords(tree,letters);
    while tree ≠ nil and letters ≠ nil do
        if letter(tree) = next-let(letters) then
            begin {lexicon node matches first non-# letter}
                letters ← move-to-front(next-let(letters), letters);
                findwords(succ(tree), tl(letters));
                if words(tree) ≠ nil then detach;
                letters ← tl(letters); {now ignore this letter and}
                tree ← fail(tree)      {continue searching}
            end
        else if hd(letters) = '#' then
            {lexicon node doesn't match first non-# letter,
                but it can match the #}
            if before(letter(tree), next-let(letters)) then
                begin {allow the # to match the node}
                    findwords(succ(tree), tl(letters));
                    if words(tree) ≠ nil then detach;
                    tree ← fail(tree) {the # can match something else
                end
            else letters ← remove-next-let(letters)
                    {we are past this letter, so ignore it}
        else if before(hd(letters), letter(tree)) then
                letters ← tl(letters) {we're past this letter,
                        so ignore it}
        else tree ← fail(tree) {keep looking}
```

Table 1 shows a trace of this procedure on the lexicon of Figure 4 and the list of letters, #VSIE. Note that success links are followed recur-

TABLE 1

Recursion Level	Node of Tree	Letter of Tree	Letters	Test Results	Action
1	1	Q	#VSIE	Q≠V, hd(LETTERS)=#, before(Q,V)	# matches Q, follow succ(1)
2	2	U	VSIE	U≠V, hd(LETTERS)≠#, before(V,U)	ignore V
2	2	U	SIE	U≠S, hd(LETTERS)≠#, ¬before(S,U)	follow fail(2)
2	nil		SIE		Return
1	1	Q	#VSIE	words(1) = nil	follow fail(1)
1	5	K	#VSIE	K≠V, hd(LETTERS)=#, before(K,V)	# matches K, follow succ(5)
2	6	L	VSIE	L≠V, hd(LETTERS)≠#, before(V,L)	ignore V
2	6	L	SIE	L≠S, hd(LETTERS)≠#, ¬before(S,L)	follow fail(6)
2	nil		SIE		Return
1	5	K	#VSIE	words(5) = nil	follow fail(5)
1	11	V	#VSIE	V=V	move V to front, follow succ(11)
2	12	L	#SIE	L≠S, hd(LETTERS)=#, before(L,S)	# matches L, follow succ(12)
3	13	S	SIE	S=S	follow succ(13)
4	14	I	IE	I=I	follow succ(14)
5	15	E	E	E=E	follow succ(15)
6	nil		nil		Return
5	15	E	E	words(15) = LIVES, EVILS	Detach
5	15	E	E		ignore E and follow fail(15)
5	nil		nil		Return
4	14	I	IE	words(14) = nil	ignore I and follow fail(14)
4	nil		E		Return
3	13	S	SIE	words(13) = nil	ignore S and follow fail(13)
3	16	I	IE	I=I	follow succ(16)
4	17	E	E	E=E	follow succ(17)
5	nil		nil		Return

TABLE 1 (Continued)

Recursion Level	Node of Tree	Letter of Tree	Letters	Test Results	Action
4	17	E	E	words(17) = LIVE, EVIL, VILE	Detach
4	17	E	E		ignore E and follow fail(17)
4	nil		nil		Return
3	16	I	IE	words(16) = nil	ignore I, follow fail(16)
3	nil		E		Return
2	12	L	#SIE	words(12) = nil	follow fail(12)
2	nil		#SIE		Return
1	11	V	V#SIE	words(11) = nil	ignore V, follow fail(11)
1	18	N	#SIE	N≠S, hd(LETTERS)=#, ¬before(N,S)	ignore S
1	18	N	#IE	N≠I, hd(LETTERS)=#, ¬before(N,I)	ignore I
1	18	N	#E	N≠E, hd(LETTERS)=#, before(N,E)	# matches N, follow succ(18)
2	19	O	E	O≠E, hd(LETTERS)≠#, ¬before(E,O)	follow fail(19)
2	nil		E		Return
1	18	N	#E	words(18) = nil	follow fail(18)
1	nil		#E		Return

sively since we will later back up along these links, but fail links are followed iteratively since once following a fail link from a node we are no longer interested in that node or its success subtree. This is the method by which only those pieces of the tree in which we are interested are retained in main memory. Also note that since words are found in groups, if n words were wanted, at least n words, but possibly a few more, will be found.

One deficiency of this search algorithm is that when the program player picks a board position to play on and passes the tile at that position along with the tiles on its rack to the search algorithm, words that do not use the tile on the board may be returned. We want to be able to mark certain letters in the list as required to be used in the retrieved words. We will define the procedure findwords2 that takes a node of the tree and an ordered list of nodes, n with the following fields:

letter(n):	a letter corresponding to an element of the second argument of findwords
req(n):	true if letter(n) is required to be in the retrieved words, false otherwise
done(n):	true if no node further on in the list is required, false otherwise

Given such a list with the letter and req fields set, it is easy to define the procedure setdone which returns the list with the done fields set properly. In the procedure below, letters is such a list after setdone has been applied, next-let(letters) returns the letter field of the first node in letters whose letter field is not #, next-req(letters) returns the req field of the first node in letters whose

letter field is not #, remove-next-let(letters) removes from letters the first node whose letter field is not #, and move-to-front(letter, letters) moves to the front of letters the first node whose letter field is letter and if that node has a true req field returns setdone of the changed list. Lines in findwords2 that are different from corresponding lines in findword are marked by a "→" in the left margin.

```
procedure findwords2(tree, letters);
    while tree ≠ nil and letters ≠ nil do
        if letter(tree) = next-let(letters) then
            begin {lexicon node matches first non-# letter}
                letters ← move-to-front(next-let(letters), letters);
                findwords(succ(tree), tl(letters));
→               if words(tree) ≠ nil and done(hd(letters)) then
                        detach;
→               if req(hd(letters)) then tree ← nil
→               else begin letters ← tl(letters);
→                          tree   fail(tree)
→                    end
            end
        else if hd(letter) = '#' then
            {lexicon node doesn't match first non-# letter, but
                but it can match the #}
            if before(letter(tree), next-let(letters)) then
                begin {allow the # to match the node}
                    findwords(succ(tree), tl(letters));
→                   if words(tree) ≠ nil and done(hd(letters))
                            then detach
                    tree ← fail(tree) {the # can match something else
                end
→           else if next-req(letters) then letters ← nil
→           else letters ← remove-next-let(letters)
        else if before(hd(letters), letter(tree)) then {we've passed
                this letter}
→           if req(hd(letters)) then letters ← nil
→           else letters ← tl(letters)
        else tree ← fail(tree) {keep looking}
```

Notice that the req field keeps us from ignoring required letters, and the done field prevents words from being returned unless all required letters are included.

5. Conclusions

The lexicon data structure and search algorithm seem successful. The program player quickly makes use of high valued tiles on the board or on its rack. There is still much room for improvement besides increasing the program's vocabulary. Implementing findwords2 would make the program player's search for words to play more efficient and more likely to yield playable words without changing the MAXLOC and MAXWDS parameters. The board evaluation scheme is still rather primitive. For example, the value of double letter squares should be twice the average value of the (still unplayed) tiles instead of 1. The program player should be modified to allow for playing words parallel to adjacent words and for extending existing words. Strategies such as maintaining a balanced rack and retaining high frequency poly-graphs to increase the future probability of a bonus word might be added. Even without these improvements, the program plays a competitive game against a human opponent.

Acknowledgements

The authors appreciate the work of Ben Shneiderman, Margaret Ambrose and Barbara Rasche who along with the senior author, worked on a preliminary version of this program. Computer services were provided by IUPUI Computing Facilities as part of the Indiana University Computing Network.

References

1. Conklin, D. K. (Ed.) The Official SCRABBLE Players Handbook. Harmony Books Division, Crown Publishers, Inc., New York, 1976.

2. Dahl, O.-J. and Hoare, C.A.R. Hierarchical program structures. Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. (Eds.) Structured Programming. Academic Press, London, 1972, 175-220.

3. Dahl, O.-J., Myhrhang, B.; Nygaard, K. The Simula 67 Common Base Language. Norwegian Computing Centre, Forskningsveien 1B, Oslo 3, 1968.

4. Dahl, O.-J., and Nygaard, K. Simula - an Algol-based simulation language. Comm. ACM 9, 9 (Sept., 1966), 671-678.

5. De La Briandais, R. File searching using variable length keys. Proc. WJCC, AFIPS Press, Montvale, N.J., 1959, 295-298.

6. Hays, D.G. Introduction to Computational Linguistics. American Elsevier, N.Y., 1967, 92-94.

7. Knuth, D.E. The Art of Computer Programming Vol. 3/Sorting and Searching. Addison-Wesley, Reading, Mass., 1973, 481-487.

8. Lamb, S.M. and Jacobsen, W.H., Jr. A high-speed large-capacity dictionary system. Mechanical Translation, 6 (Nov., 1961), 76-107. Reprinted in Hays, D.G. (Ed.) Readings in Automatic Language Processing. American Elsevier, New York, 1966, 51-72.

Appendix

Table 2 provides a record of a complete game played between the program player and a human, DPF. DPF had previously played several rushed rounds against the program and consistently scored lower than the program. However, in the game presented here, he took advantage of his knowledge of the program's weaknesses. The game was played with the inspect mode on and took 17 minutes, 28.56 seconds of CPU time on a DECSystem-10 and 2 hours, 50 minutes, 39.05 seconds of real time. At the end of the game, there were 1488 words in the lexicon.

In Table 2, a letter already on the board is underlined and blanks are represented as a square. Figure 5 shows the board at the end of the game.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	
1	4			1				4	E			Q	U	E	4	1
2		3				2		E	V	E				L	O	2
3			3				1		I		W	I	N	K		3
4	1			3				1	L	O	I	N			1	4
5					3						S	N	O	W		5
6		2				2				D	E			R	E	6
7			1				T		H				V	A	N	7
8	4			1	C		H	O	O	F		1	A	P	T	8
9			1	B	U	S	Y		1	I	F		N		R	9
10		2			L	A				R	A	G	E	2	E	10
11					T	I				M	I			Y	E	11
12	1			3		D	I	M	E		L	A	Z	E	S	12
13	J		C	O	G		1	O	N				I	T		13
14	A	3	A			2	R	A	D	A	R		P	I	T	14
15	R	E	B	U	D	G	E	T	S			1			O	15
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	

FIGURE 5: The board at the end of the game of Table 2

Table 2

	Rack	Word	Location	Score DPF Play	Score DPF Total	Score Program Play	Score Program Total
1.	IFOHIOT	HOOF	8G-J	20	20		
2.	GSMHR□B	F[I]RM	J8-11			10	10
3.	LATIFIT	FAIL	K9-12	28	48		
4.	EZIGSHB	HO	I7-8			9	19
5.	ITYTTIT	THY	G7-9	14	62		
6.	AEZIGSB	LAZE	12K-N			26	45
7.	YCITTIT	YET	N11-13	6	68		
8.	UNOIGSB	BUSY	9D-G			9	54
9.	DLCITIT	CULT	E8-11	12	80		
10.	PEDNOIG	ZIP	M12-14			32	86
11.	ONADIIT	SAID	F9-12	13	93		
12.	RPEDNOG	exchange rack					86
13.	AREONIT	PI	14M-N	22	115		
14.	GMRDNEI	DIME	12F-I			10	96
15.	AAREONT	MOAT	H12-15	18	133		
16.	ANRGRDN	RADAR	14G-K			8	104
17.	TEEAREN	ENTREE	O6-11	23	156		
18.	UAJTNGN	exchange rack					104
19.	ACISEAE	LAZES	12K-O	23	179		
20.	PUB□SGD	BUDG[E]T	15C-H			11	115
21.	RACIEAE	REBUDGET	15A-H	33	212		
22.	INQIVPS	ON	13H-I			8	123
23.	UACIEAE	CAB	C13-15	14	226		
24.	EIQIVPS	exchange rack					123
25.	VNUIEAE	VAN	7M-O	10	236		
26.	ROGNLGI	exchange rack					123
27.	ANUIEAE	VANE	M7-10	8	244		
28.	EQVLUNI	exchange rack					123
29.	ITOUIAE	TO	O14-15	11	255		
30.	DEEOOKW	exchange rack					123
31.	JNIUIAE	JAR	A13-15	10	265		
32.	PGGRLVQ	RAP	N6-8			16	139
33.	ISNIUIE	ENDS	I12-15	17	282		
34.	EOGGLVQ	exchange rack					139
35.	WINIUIE	WRAP	N5-8	9	291		

Table 2 (Continued)

	Rack	Word	Location	Score DPF Play	DPF Total	Program Play	Program Total
36.	EDOWNSE	SNOW	5J-N			14	153
37.	OINIUIE	INN	L3-5	6	297		
38.	ONKEDWE	WINK	3K-N			22	175
39.	QGOIUIE	RAGE	10J-M	5	302		
40.	LUIOEDE	ELK	N1-3			14	189
41.	LQOIUIE	QUE	1L-N	22	324		
42.	EGUIODE	pass					189
43.	XOLOIIE	WISE	K3-6	9	333		
44.	EGUIODE	DE	6J-K			7	196
45.	EVXOLOI	LOIN	4I-L	4	337		
46.	EGUIOE	COG	13C-E			6	202
47.	EVXOI	EVIL	I1-4	8	345		
48.	EUIE	EVE	2H-J			8	210
49.	XO	LO	2N-O	2	347		
50.	UI	pass					210
51.	X	pass			347		
					339		208