SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(2), 219–232 (FEBRUARY 1994)

A Faster Scrabble Move Generation Algorithm

steven a. gordon
Department of Mathematics, East Carolina University, Greenville, NC 27858, U.S.A.
(email: magordon@ecuvax.cis.ecu.edu)

SUMMARY

Appel and Jacobson[1] presented a fast algorithm for generating every possible move in a given position in the game of Scrabble using a DAWG, a finite automaton derived from the trie of a large lexicon. This paper presents a faster algorithm that uses a GADDAG, a finite automaton that avoids the non-deterministic prefix generation of the DAWG algorithm by encoding a bidirectional path starting from each letter of each word in the lexicon. For a typical lexicon, the GADDAG is nearly five times larger than the DAWG, but generates moves more than twice as fast. This time/space trade-off is justified not only by the decreasing cost of computer memory, but also by the extensive use of move-generation in the analysis of board positions used by Gordon[2] in the probabilistic search for the most appropriate play in a given position within realistic time constraints.

key words: Finite automata Lexicons Backtracking Games Artificial intelligence

INTRODUCTION

Appel and Jacobson[1] presented a fast algorithm for generating every possible move given a set of tiles and a position in Scrabble (in this paper Scrabble refers to the SCRABBLE® brand word game, a registered trade mark of Milton Bradley, a division of Hasbro, Inc.). Their algorithm was based on a large finite automaton derived from the trie[3,4] of the entire lexicon. This large structure was called a directed acyclic word graph (DAWG).

Structures equivalent to a DAWG have been used to represent large lexicons for spell-checking, dictionaries, and thesauri.[5–7] Although a left-to-right lexical representation is well-suited for these applications, it is not the most efficient representation for generating Scrabble moves. This is because, in Scrabble, a word is played by 'hooking' any of its letters onto the words already played on the board, not just the first letter.

The algorithm presented here uses a structure similar to a DAWG, called a GADDAG, that encodes a bidirectional path starting from each letter in each word in the lexicon. The minimized GADDAG for a large American English lexicon is approximately five times larger than the minimized DAWG for the same lexicon, but the algorithm generates moves more than twice as fast on average. This faster

CCC 0038–0644/94/020219–14 *Received 29 March 1993*
© 1994 by John Wiley & Sons, Ltd. *Revised 30 August 1993*

algorithm makes the construction of a program that plays Scrabble intelligently within realistic time constraints a more feasible project.

Bidirectional string processing is not a novel concept. One notable example is the Boyer–Moore string searching algorithm.[8–10] In addition to moving left or right, this algorithm also sometimes skips several positions in searching for a pattern string within a target string.

The advantage of a faster algorithm

The DAWG algorithm is extremely fast. There would be little use for a faster algorithm if the highest scoring move was always the 'best' one. Although a program that simply plays the highest scoring play will beat most people, it would not fare well against most tournament players. North American tournament Scrabble differs from the popular version in that games are always one-on-one, have a time limit of 25 minutes per side, and have a strict word challenge rule. When a play is challenged and is not in the official dictionary, OSPD2,[11] the play is removed, and the challenger gets to play next. Otherwise, the play stands and the challenger loses his/her turn. The most apparent characteristic of tournament play is the use of obscure words (e.g. XU, QAT and JAROVIZE). However, the inability of a program which knows every word and always plays the highest scoring one to win even half of its games against expert players indicates that strategy must be a significant component of competitive play.

Nevertheless, there would still be no need for a faster algorithm if expert strategy could be modeled effectively by easily computed heuristic functions. Modeling the strategy of Scrabble is made difficult by the presence of incomplete information. In particular, the opponent's rack and the next tiles to be drawn are unknown, but the previous moves make some possibilities more likely than others. Gordon[2] compares the effectiveness of weighted heuristics and simulation for evaluating potential moves. Heuristics that weigh the known factors in the proportions that perform most effectively over a large random sample of games give an effective, but unintelligent, strategy. Simulating candidate moves in a random sample of plausible scenarios leads to a strategy that responds more appropriately to individual situations. Faster move generation facilitates the simulation of more candidate moves in more scenarios within competitive time constraints. Furthermore, in end game positions, where the opponent's rack can be deduced, faster move generation would make an exhaustive search for a winning line more feasible.

NON-DETERMINISM IN THE FAST ALGORITHM

Appel and Jacobson acknowledged that the major remaining source of inefficiency in their algorithm is the unconstrained generation of prefixes. Words can only be generated from left to right with a DAWG. Starting from each anchor square (a square on the board onto which a word could be hooked) the DAWG algorithm handles prefixes (letters before the anchor square) differently to suffixes (those on or after the anchor square). The DAWG algorithm builds every string shorter than a context-dependent length that can be composed from the given rack and is the prefix of at least one word in the lexicon. It then extends each such prefix into complete words as constrained by the board and the remaining tiles in the rack.

When each letter of a prefix is generated, the number of letters that will follow it is variable, so where it will fall on the board is unknown. The DAWG algorithm therefore only generates prefixes as long as the number of unconstrained squares left of an anchor square. Nevertheless, many prefixes are generated that have no chance of being completed, because the prefix cannot be completed with any of the remaining tiles in the rack, the prefix cannot be completed with the letter(s) on the board that the play must go through, or the only hookable letters were already consumed in building the prefix.

They suggest eliminating this non-determinism with a 'two-way' DAWG. A literal interpretation of their proposal is consistent with their prediction that it would be a huge structure. The node for substring x could be merged with the node for substring y if and only if $\{(u,v) \mid uxv \text{ is a word}\} = \{(u,v) \mid uyv \text{ is a word}\}$, so minimization would be ineffective.

A MORE DETERMINISTIC ALGORITHM

A practical variation on a two-way DAWG would be the DAWG for the language $L = \{\mathsf{REV}(x)\Diamond y \mid xy \text{ is a word and } x \text{ is not empty}\}$, where $\Diamond$ is just a delimiter. This structure would be much smaller than a complete two-way DAWG and still avoid the non-deterministic generation of prefixes. Each word has as many representations as letters, so, before minimization, this structure would be approximately n times larger than an unminimized DAWG for the same lexicon, where n is the average length of a word.

Each word in the lexicon can be generated starting from each letter in that word by placing tiles leftward upon the board starting at an anchor square while traversing the corresponding arcs in the structure until $\Diamond$ is encountered, and then placing tiles rightward from square to the right of the anchor square while still traversing corresponding arcs until acceptance. A backtracking, depth-first search[12] for every possible path through the GADDAG given the rack of tiles and board constraints generates every legal move.

Being the reverse of the directed acyclic graph for prefixes followed by the directed acyclic graph for suffixes, it will be called a GADDAG. Reversing the prefixes allows them to be played just like suffixes, one tile at a time, moving away from anchor squares. The location of each tile in the prefix is known, so board constraints can be considered, eliminating unworkable prefixes as soon as possible. Requiring the prefix to be non-empty allows the first tile in the reverse of the prefix to be played directly on the anchor square. This immediately eliminates many otherwise feasible paths through the GADDAG.

A DAGGAD, the DAWG for $\{y\Diamond\mathsf{REV}(x) \mid xy \text{ is a word and } y \text{ is not empty}\}$, would work just as well—tiles would be played rightward starting at an anchor square and then leftward from the square left of the anchor square.

The following conventions allow a compressed representation of a GADDAG, as well as partial minimization during construction:

1. If the y in $\mathsf{REV}(x)\Diamond y$ is empty, the $\Diamond$ is omitted altogether.
2. A state specifies the arcs leaving it and their associated letters.
3. An arc specifies
   (a) its destination state
   (b) its letter set—the letters which, if encountered next, make a word.

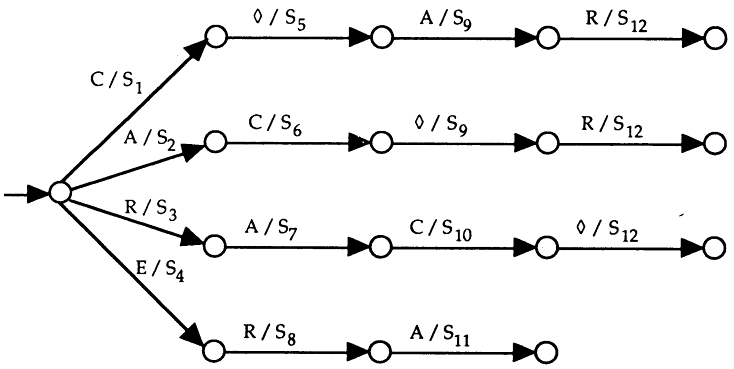

Figure 1. Subgraph of unminimized GADDAG for 'CARE' (see Table I for letter sets)

Placing letter sets on arcs avoids designating states as final or not.

Figure 1 is the subgraph of an unminimized GADDAG that contains the representations of the word CARE. The letter sets on the arcs in Figure 1 can be found in Table I. CARE has four distinct paths, C◇ARE, AC◇RE, RAC◇E, and ERAC, corresponding to hooking the C, A, R, and E, respectively, onto the board.

The move generation algorithm

Figure 2 illustrates the production of one play using each path for CARE through the GADDAG in Figure 1 on a board containing just the word ABLE. A play can

Table I. Letter sets for Figures 1, 5, and 6.

S_1	=	{Δ	ΔC is a word}	=	∅.
S_2	=	{Δ	ΔA is a word}	=	{A,B,D,F,H,K,L,M,N,P,T,Y}.
S_3	=	{Δ	ΔR is a word}	=	{A,E,O}.
S_4	=	{Δ	ΔE is a word}	=	{A,B,D,H,M,N,O,P,R,W,Y}.
S_5	=	{Δ	CΔ is a word}	=	∅.
S_6	=	{Δ	ΔCA is a word}	=	{O}.
S_7	=	{Δ	ΔAR is a word}	=	{B,C,E,F,G,J,L,M,O,P,T,V,W,Y}.
S_8	=	{Δ	ΔRE is a word}	=	{A,E,I,O}.
S_9	=	{Δ	CAΔ is a word}	=	{B,D,M,N,P,R,T,W,Y}.
S_{10}	=	{Δ	ΔCAR is a word}	=	{S}.
S_{11}	=	{Δ	ΔARE is a word}	=	{B,C,D,F,H,M,P,R,T,W,Y}.
S_{12}	=	{Δ	CARΔ is a word}	=	{B,D,E,K,L,N,P,S,T}.
S_{13}	=	{Δ	ΔN is a word}	=	{A,E,I,O,U}.
S_{14}	=	{Δ	ΔEE is a word}	=	{B,C,D,F,G,J,L,N,P,R,S,T,V,W,Z}.
S_{15}	=	{Δ	ΔEN is a word}	=	{B,D,F,H,K,M,P,S,T,W,Y}.
S_{16}	=	{Δ	ΔREE is a word}	=	{B,D,F,G,P,T}.
S_{17}	=	{Δ	ΔEEN is a word}	=	{B,K,P,S,T,W}.
S_{18}	=	{Δ	ΔCARE is a word}	=	{S}.
S_{19}	=	{Δ	ΔAREE is a word}	=	∅.
S_{20}	=	{Δ	ΔREEN is a word}	=	{G,P}.
S_{21}	=	{Δ	CAREΔ is a word}	=	{D,R,S,T,X}.
S_{22}	=	{Δ	ΔCAREE is a word}	=	∅.
S_{23}	=	{Δ	ΔAREEN is a word}	=	{C}.
S_{24}	=	{Δ	CAREEΔ is a word}	=	{N,R}.

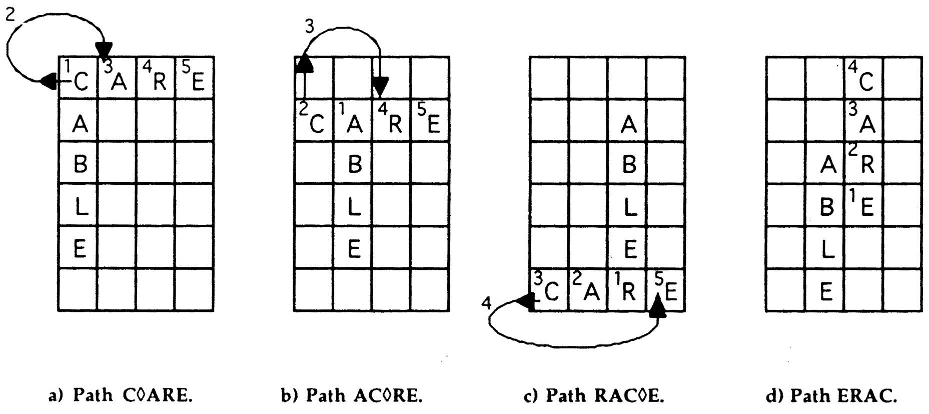

Figure 2. Four ways to play 'CARE' on 'ABLE'

connect in front (above), in back (below), through, or in parallel with words already on the board, as long as every string formed is a word in the lexicon.

Consider, for example, the steps (corresponding to the numbers in the upper left corners of the squares) involved in play (c) of Figure 2. CARE can be played perpendicularly below ABLE as follows:

1. Play R (since ABLER is a word); move left; follow the arc for R.
2. Play A; move left; follow the arc for A.
3. Play C; move left; follow the arc for C.
4. Go to the square right of the original starting point; follow the arc for ◇.
5. Play the E, since it is in the last arc's letter set.

The GADDAG algorithm for generating every possible move with a given rack from a given anchor square is presented in Figure 3 in the form of backtracking, recursive co-routines. Gen(0,NULL,RACK,INIT) is called, where INIT is an arc to the initial state of the GADDAG with a null letter set. The Gen procedure is independent of direction. It plays a letter only if it is allowed on the square, whether letters are being played leftward or rightward. In the GoOn procedure, the direction determines which side of the current word to concatenate the current letter to, and can be shifted just once, from leftward to rightward, when the ◇ is encountered.

A GADDAG also allows a reduction in the number of anchor squares used. There is no need to generate plays from every other internal anchor square of a sequence of contiguous anchor squares (e.g. the square left or right of the B in Figure 2), since every play from a given anchor square would be generated from the adjacent anchor square either to the right (above) or to the left (below). In order to avoid generating the same move twice, the GADDAG algorithm was implemented with a parameter to prevent leftward movement to the previously used anchor square.

The GADDAG algorithm is still non-deterministic in that it runs into many dead-ends. Nevertheless, it requires fewer anchor squares, hits fewer dead-ends, and follows fewer arcs before detecting dead-ends than the DAWG algorithm.

```
Gen(pos, word, rack, arc):                    {pos = offset from anchor square}
   IF a letter, L, is already on this square THEN
      GoOn(pos, L, word, rack, NextArc(arc, L), arc)
   ELSE IF letters remain on the rack THEN
      FOR each letter on the rack, L, allowed on this square
         GoOn(pos, L, word, rack - L, NextArc(arc, L), arc)
      IF the rack contains a BLANK THEN
         FOR each letter the BLANK could be, L, allowed on this square
            GoOn(pos, L, word, rack - BLANK, NextArc(arc, L), arc)

GoOn(pos, L, word, rack, NewArc, OldArc):
   IF pos ≤ 0 THEN                                         {moving left:}
      word ← L || word
      IF L on OldArc & no letter directly left THEN RecordPlay
      IF NewArc ≠ 0 THEN
         IF room to the left THEN Gen(pos-1, word, rack, NewArc)
         NewArc ← NextArc(NewArc, ◊)                  {shift direction:}
         IF NewArc ≠ 0, no letter directly left & room to the right THEN
            Gen(1, word, rack, NewArc)
   ELSE IF pos > 0 THEN                                   {moving right:}
      word ← word || L
      IF L on OldArc & no letter directly right THEN RecordPlay
      IF NewArc ≠ 0 & room to the right THEN
         Gen(pos+1, word, rack, NewArc)
```

Figure 3. The GADDAG move generation algorithm

Computing cross sets

Appel and Jacobson's DAWG implementation uses and maintains a structure for keeping track of which squares are potential anchor squares (horizontally and/or vertically), and for each such anchor square, the set of letters that can form valid crosswords (cross sets). Whenever a play is made, only the squares directly affected by the play need to be updated. The GADDAG implementation uses and maintains the same structure.

Computing a right cross set (i.e. the set of letters for the square to the right of a word or single letter) is easy with a DAWG—start in the initial state and follow the arcs associated with the letters of the word. Computing the left cross set of a word is equivalent to generating the set of one-letter prefixes, and thus exhibits the same non-determinism as prefix generation. For each letter of the alphabet, one must follow the arc for that letter from the initial state of the DAWG, and then follow the arcs associated with each letter of the word to see if they lead to acceptance.

A GADDAG supports the deterministic and simultaneous computation of left and right cross sets. Just start in the initial state and follow arcs for each letter in the word (reading from right to left). The left cross set is the letter set on the last arc and the right cross set is the letter set on the ◇ arc from the state that the last arc led to.

There is one rare case where the computation of a cross set is not deterministic. When a square is left of one word and right of another, then one must follow one word through the GADDAG, and then for each letter of the alphabet, follow that letter and then the letters in the other word to see if they lead to acceptance. For example, if PA and ABLE were separated by just one square, this computation would allow a word to be played perpendicular to them if it placed an R or a Y between them.

Partial and full minimization

For all strings x, y, and z, REV(x)◇yz is a path through the GADDAG if and only if xyz is a word. So, if $xy = vw$, then $\{z \mid$ REV (x)◇yz is a path$\}$ = $\{z \mid$ REV(v)◇wz is a path$\}$. Standard minimization[13] of the GADDAG as a finite automaton would therefore merge the node that REV(x)◇y leads to with the node that REV(v)◇w leads to. For example, in the instance of CARE, the node that C◇A leads to would be merged with the node that AC◇ leads to, and the nodes that C◇AR, AC◇R, and RAC◇ each lead to would also be merged into a single node.

The algorithm given in Figure 4 merges all such states during the initial construction of the GADDAG. The resulting automaton is still not fully minimized, but the comparatively slow, standard minimization process receives a much smaller automaton to finish minimizing.

Figure 5 is the subgraph of the semi-minimized GADDAG produced by this algorithm that contains the representation of the word CARE. Figure 6 is the subgraph containing the representation of the word CAREEN. (Table I lists the letter sets for Figures 5 and 6). The longer the word and the more duplicate letters, the more states this algorithm eliminates.

Replacing final states with letter sets on arcs eliminates an explicit arc and state for the last letter in each path of each word. Letter sets also allow many states to be merged in minimization that otherwise would not be. For example, both WOUND and ZAGG can only be followed by the multi-letter strings ED and ING. Even though WOUND is a word and ZAGG is not, the node that W◇OUND, OW◇UND, UOW◇ND, NUOW◇D, and DNUOW◇ all lead to can be merged with the node

```
FOR each word $a_1a_2…a_n$ in the lexicon:
    st ← initialState                        {create path for $a_na_{n-1}…a_1$:}
    FOR i FROM n DOWNTO 3
        AddArc(st, $a_i$)
    AddFinalArc(st, $a_2$, $a_1$)

    st ← initialState                        {create path for $a_{n-1}…a_1◊a_n$:}
    FOR i FROM n-1 DOWNTO 1
        AddArc(st, $a_i$)
    AddFinalArc(st, ◊, $a_n$)

    FOR m FROM n-2 DOWNTO 1      {partially minimize the remaining paths:}
        forceSt ← st
        st ← initialState
        FOR i FROM m DOWNTO 1
            AddArc(st, $a_i$)
        AddArc(st, ◊)
        ForceArc(st, $a_{m+1}$, forceSt)

AddArc(st, ch) adds an arc from st for ch (if one does not already exist)
    and resets st to the node this arc leads to.

AddFinalArc(st, c1, c2) adds an arc from st for c1 (if one does not
    already exist) and adds c2 to the letter set on this arc.

ForceArc(st, ch, fst) adds an arc from st for ch to fst (an error occurs
    if an arc from st for ch already exists going to any other state).
```

Figure 4. The GADDAG construction algorithm

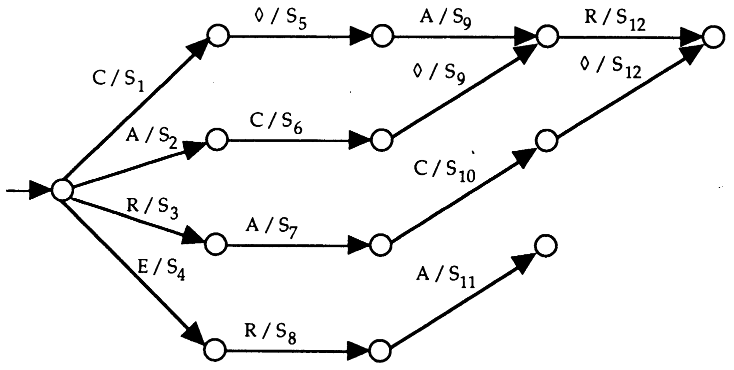

Figure 5. Subgraph of semi-minimized GADDAG for 'CARE' (see Table I for letter sets)

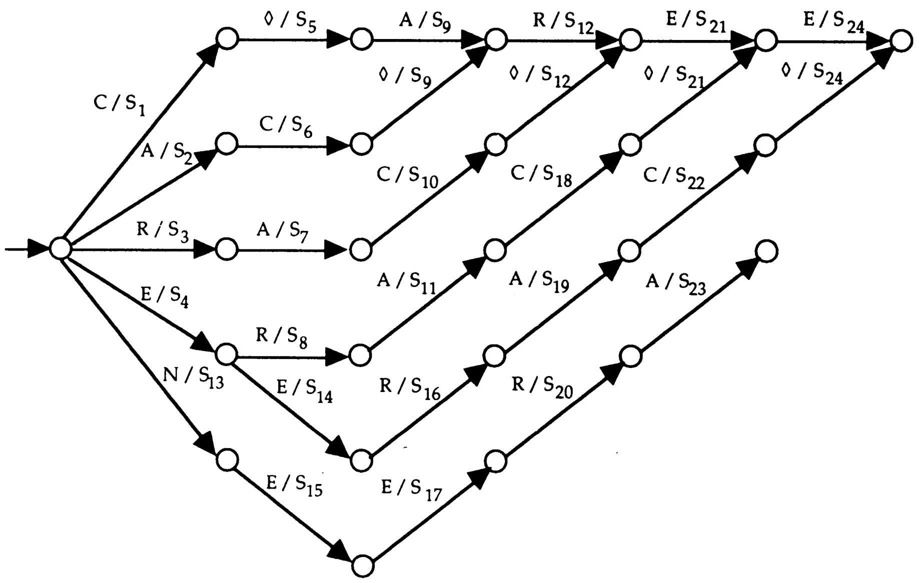

Figure 6. Subgraph of semi-minimized GADDAG for 'CAREEN' (see Table I for letter sets)

that Z◇AGG, AZ◇GG, GAZ◇G, and GGAZ◇ all lead to. Each arc leading to the former node has the letter set {S}, whereas each arc leading to the latter node has a null letter set. After merging, those arcs will all lead to the same node, but their letter sets will remain distinct. Incidentally, the node that DNUOW leads to cannot be merged with the node that GGAZ lead to, since these strings can be completed by different strings (e.g. the path DNUOWER for REWOUND and the path GGAZGIZ◇ED for ZIGZAGGED). The ◇ precludes this.

Compression

A GADDAG (or DAWG) could be represented in a various expanded or compressed forms. The simplest expanded form is a 2-dimensional array of arcs indexed by state and letter. In the current lexicon, the number of distinct letter sets, 2575, and distinct states, 89,031, are less than 2^{12} and 2^{17}, respectively. So, each arc can encode the indices of a letter set and a destination state within a 32-bit word. The array of letter sets takes just over 10 kilobytes. Non-existent arcs or states are just encoded with a 0.

The simplest compressed representation is a single array of 32-bit words. States are a bit map indicating which letters have arcs. Each arc is encoded in 32-bit words as in the expanded representation. In this combined array, arcs directly follow the states they originate from in alphabetical order.

Compression has two disadvantages. The first is that encoding destination states in arcs limits the potential size of the lexicon. Under expanded representation, a state is just the first index of a two-dimensional array, whereas, under compression, a state is the index of its bit-map in a single array of both arcs and states. This second index grows much faster, so that a larger lexicon would allow arcs to be encoded in 32 bits under expanded representation than under compression.

The second disadvantage is the time it takes to find arcs. Each arc is addressed directly under expanded representation. Under compression, the bit map in a state indicates if a given letter has an associated arc. One must count how many of the preceding bits are set and then skip that many 32-bit words to actually find the arc. Although the actual number of preceding bits that are set is usually small, each preceding bit must be examined to compute the correct offset of the associated arc.

The advantage of compression is, of course, a saving in space. The histogram of states by number of arcs in Table II indicates that most states have only a few arcs. Avoiding the explicit storage of non-existent arcs thus saves a great amount of space. A compressed GADDAG requires $S + A$ 32-bit words, compared to $27S$ for an expanded GADDAG, where S is the number of states and A is the number of arcs. Nevertheless, the automata in Reference 7 had fewer states with more than 10 arcs, since a lexicon of commonly-used words is less dense than a Scrabble lexicon.

Table III compares the sizes of DAWG and GADDAG structures built from a lexicon of 74,988 words from the OSPD2[11]. Word lengths vary from 2 to 8 letters, averaging 6·7 letters per word. Since people rarely play words longer than 8 letters, the OSPD2 only lists base words from 2 to 8 letters in length and their forms. The longer words that are listed are not representative, having, for example, a disproportionate number of ING endings. Comparing DAWG and GADDAG structures on a lexicon with just these longer words could be misleading, so longer words were omitted.

The minimized and compressed DAWG structure represents the lexicon with an impressive 4·3 bits per character, less than half the number of bits required to represent the lexicon as text (including one byte per word for a delimiter). Appel and Jacobson achieved an even better ratio by encoding many arcs in 24 bits. The GADDAG minimized more effectively than the DAWG, being just less than 5 times larger rather than the 6·7 times larger expected with equally effective minimization.

PERFORMANCE

Table IV compares the performances of the DAWG and GADDAG move generation algorithms implemented within identical shells in Pascal playing 1000 randomly generated games on a VAX4300 using both compressed and expanded representations.

Table II. Histogram of states by number of outgoing arcs in minimized DAWG and GADDAG structures

Number of arcs	DAWG		GADDAG	
per state	States	%	States	%
0	0	0·0	0	0·0
1	5708	32·0	35,103	39·4
2	5510	30·9	24,350	27·4
3	2904	16·3	11,291	12·7
4	1385	7·8	5927	6·7
5	775	4·3	3654	4·1
6	445	2·5	2187	2·5
7	299	1·7	1435	1·6
8	189	1·1	1009	1·1
9	148	0·8	821	0·9
10	98	0·5	641	0·7
11	84	0·5	493	0·6
12	53	0·3	389	0·4
13	52	0·3	325	0·4
14	41	0·2	254	0·3
15	30	0·2	198	0·2
16	20	0·1	165	0·2
17	23	0·1	150	0·2
18	16	0·1	128	0·1
19	14	0·1	102	0·1
20	10	0·1	81	0·1
21	18	0·1	87	0·1
22	12	0·1	65	0·1
23	8	0·0	59	0·1
24	6	0·0	52	0·1
25	4	0·0	26	0·0
26	4	0·0	34	0·0
27	–	–	5	0·0

Table III. Relative sizes of DAWG and GADDAG structures

	DAWG		GADDAG		Ratio (G/D)
	Unminimized	Minimized	Semi-minimized	Minimized	minimized)
States	55,503	17,856	250,924	89,031	4·99
Arcs	91,901	49,341	413,887	244,117	4·95
Letter sets	908	908	2575	2575	2·84
Expanded					
Bytes	5,775,944	1,860,656	27,110,092	9,625,648	5·17
Bits/char	91·6	29·5	429·8	152·6	
Compressed					
Bytes	593,248	272,420	2,669,544	1,342,892	4·93
Bits/char	9·4	4·3	42·3	21·3	

Table IV. Relative performance of DAWG and GADDAG algorithms playing both sides of 1000 random games on a VAX4300

	DAWG		GADDAG		Ratio
	overall	per move	overall	per move	(D/G)
CPU time					
Expanded	9:32:44	1·344s	3:38:59	0·518s	2·60
Compressed	8:11:47	1·154s	3:26:51	0·489s	2·36
Page faults					
Expanded	6063		32,305		0·19
Compressed	1011		3120		0·32
Arcs Traversed	668,214,539	26,134	265,070,715	10,451	2·50
Per sec (compressed)	22,646		21,372		
Anchors used	3,222,746	126·04	1,946,163	76·73	1·64
Number of moves	25,569		25,363		
Average score	389·58		388·75		

The VAX had 64M of memory, a light, mostly interactive work load, and effectively unlimited image sizes, so performance was not significantly affected by either memory management or competing work load. The DAWG algorithm traversed 2·5 times as many arcs in its structure as the GADDAG algorithm did in its. CPU times reflect a slightly smaller ratio. In other words, both algorithms traverse about 22,000 arcs/s, but the GADDAG algorithm traverses the same number of arcs to generate five moves as the DAWG algorithm traverses to generate two moves.

The shell used the greedy evaluation function (i.e. play the highest scoring move). Ties for high score were broken by using the move found first. Since ties occurred frequently and the algorithms do not generate moves in exactly the same order, the actual games played diverged quickly.

Each expanded implementation ran slightly slower than the respective compressed implementation. The page faults due to the larger memory demands of the expanded structures evidently take more time to process on the VAX than searching for arcs in the compressed structures. On a dedicated machine with enough memory, the expanded implementations might run faster.

Some additional speed-up could be expected from reordering bit maps and arcs into letter frequency order (i.e. ◇, E, A, I, …) to reduce the average number of bits preceding a letter.[4] In practice, reordering had little effect on CPU times. This may be because A and E are already near the beginning and the GADDAG implementation already placed ◇ before all the letters, so most of the advantage of reordering had already been achieved.

Performance with blanks

Appel and Jacobson note that their program hesitates noticeably when it has a blank. A blank can stand for any letter, so many more words can usually be made from a rack with a blank that a rack without one. Table V presents a logarithmic histogram of the number of arcs traversed per move with and without a blank for both algorithms.

Most plays are generated faster than average, but some plays take much longer

Table V. Logarithmic histogram of arcs traversed per move with and without blanks in 1000 random games

Arc range	DAWG With blank	DAWG Without	DAWG Total	GADDAG With blank	GADDAG Without	GADDAG Total
0–255	0	24	24	0	45	45
256–511	0	896	896	0	847	847
512–1K	0	877	877	0	1618	1618
1K–2K	0	1075	1075	0	2680	2680
2K–4K	0	1809	1809	8	6260	6268
4K–8K	7	4654	4661	45	7935	7980
8K–16K	29	8348	8357	126	3605	3731
16K–32K	107	5209	5316	353	337	690
32K–64K	201	602	803	734	1	735
64K–128K	569	1	570	559	0	559
128K–256K	774	0	774	183	0	183
256K–512K	349	0	349	18	0	18
512K–1M	23	0	23	7	0	7
1M–2M	13	0	13	2	0	2
2M–4M	2	0	2	0	0	0
Total	23,495	2074	25,569	23,328	2035	25,363
Average arcs	11,961	179,850	26,134	5467	67,586	10,451

than average. The worst case requires about 1 minute for the GADDAG algorithm versus about 2 minutes for the DAWG algorithm (the frequency of this case, twice in both sides of 1000 games, suggests these racks contain both blanks).

As measured by arcs traversed, the GADDAG algorithm is 2·19 times faster on racks without blanks, whereas it is 2·66 times faster on racks with blanks. The GADDAG algorithm still hesitates when it encounters blanks, but a little less.

Performance under heuristics

The performance advantage of GADDAG algorithm on racks with blanks suggests that the better the rack, the greater the advantage. Gordon[2] presents three heuristics that improve the average quality of racks by considering the utility of the tiles left on the rack when choosing a move. Rack Heuristic1, from Reference 14, estimates the utility of each letter in the alphabet. Rack Heuristic2 has an additional factor to discourage keeping duplicate letters. Rack Heuristic3 includes another factor to encourage a balance between vowels and consonants. Table VI shows that when

Table VI. Relative performance of DAWG and GADDAG algorithms aided by rack evaluation heuristics

	DAWG			GADDAG			Ratio (D/G)		
	Secs/ move	Arcs/ move	Page faults	Secs/ move	Arcs/ move	Page faults	Secs/ move	Arcs/ move	Page faults
Greedy vs. Greedy	1·154	26,134	1011	0·489	10,451	3120	2·36	2·50	0·32
RackH1 vs. Greedy	1·390	32,213	1013	0·606	12,612	3136	2·29	2·55	0·32
RackH2 vs. Greedy	1·448	33,573	1016	0·630	13,060	3139	2·30	2·57	0·32
RackH3 vs. Greedy	1·507	35,281	1017	0·655	13,783	3141	2·30	2·56	0·32

measured by arcs traversed per move, the performance advantage of GADDAG algorithm over the DAWG algorithm increases slightly under these heuristics. However, the ratio of seconds per move actually decreases slightly.

Each heuristic is computed for each move found by the move generation algorithms. Heuristic processing time is therefore a function of the number of moves generated rather than the number of arcs traversed generating those moves. Heuristic processing time per move is bounded by the increase in average CPU times for the GADDAG algorithm. The fact that average CPU times increased by twice as much for the DAWG algorithm suggests that at least half of this increase was due to poorer performance generating moves from the better racks that resulted from using the heuristics.

APPLICATIONS AND GENERALIZATIONS

Most text processing applications outside of Scrabble and crossword puzzles are well-suited to left-to-right processing, and would not seem to benefit from the GADDAG structure. However, the algorithm could be applied outside of text processing.

Ignoring the textual details, the GADDAG algorithm fits objects into an environment. The environment must be represented by a grid and the lexicon of potential objects must be described by strings in a finite alphabet. The constraints for ‘hooking’ one object onto another must be expressed in terms of symbols being on the same or adjacent squares. Objects could be represented in two or more dimensions by encoding strings with multiple delimiters.

Consider the following oversimplified, biochemical application along the lines of algorithms than can be found in Reference 15. The GADDAG structure could encode a lexicon of molecules from the point of view of each atom in each molecule that can be ‘hooked’ onto atoms in other molecules. Then, given an environment consisting of already existing molecules and their locations, the GADDAG algorithm would find every location that a molecule, composed from a given collection of atoms, can be ‘hooked’ onto another molecule in the environment without conflicting with any of the other molecules.

FUTURE WORK

The lexicon should be expanded to include all nine-letter words, and the effect on the relative size of the data structures and the performance of the algorithms should be remeasured. Even longer words, up to 15 letters, could also be added.

Research into improving the modeling of Scrabble strategy continues on three fronts: weighted heuristics for the evaluation of possible moves, the use of simulation to select the most appropriate candidate move in a given position, and exhaustive search for the optimal move in end games.

CONCLUSION

Although the Appel and Jacobson algorithm generates every possible move in any given Scrabble position with any given rack very quickly using a deterministic finite automaton, the algorithm itself is not deterministic. The algorithm presented here achieves greater determinism by encoding bidirectional paths for each word starting at each letter. The resulting program generates moves more than twice as fast, but takes up five times as much memory for a typical lexicon. In spite of the memory usage, a faster algorithm makes the construction of a program that plays intelligently within competitive time constraints a more feasible project.

REFERENCES

1. A. Appel and G. Jacobson, 'The world's fastest Scrabble program', *Commun. ACM*, **31**,(5), 572–578, 585 (1988).
2. S. Gordon, 'A comparison between probabilistic search and weighted heuristics in a game with incomplete information', *Technical Report*, Department of Mathematics, East Carolina University, August 1993. Also, to appear in *AAAI Fall Symposium Series*, Raliegh, NC (October 1993).
3. E. Fredkin, 'Trie memory', *Commun. ACM*, **3**,(9), 490–500 (1960).
4. D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973, pp.481–500, 681.
5. M. Dunlavey, 'On spelling correction and beyond', *Commun. ACM*, **24**(9), 608 (1981).
6. K. Kukich, 'Techniques for automatically correcting words in text', *ACM Comput. Surv.*, **24**(4), 382–383, 395 (1992).
7. C. Lucchesi and T. Kowaltowski, 'Applications of finite automata representing large vocabularies', *Software—Practice and Experience*, **23**, 15–30 (1993).
8. A. Apostolico and R. Giancarlo, 'The Boyer–Moore–Galil string searching strategies revisited', *SIAM J. Comput.*, **15**,(1), 98–105 (1986).
9. S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, 2nd edn, Addison-Wesley, Reading, MA, 1988, pp.209–230.
10. R. Boyer and J. Moore, 'A fast string searching algorithm', *Commun. ACM*, **20**,(10), 762–772 (1977).
11. Milton Bradley Company, *The Official Scrabble Players Dictionary*, 2nd edn, Merriam-Webster Inc., Springfield,MA, 1990.
12. R.Tarjan, 'Depth-first search and linear graph algorithms', *SIAM J. Comput.*, **1**,(2), 146–160 (1972).
13. A. Nerode, 'Linear automaton transformations', *Proc. AMS*, **9**, 541–544 (1958).
14. N. Ballard, 'Renaissance of Scrabble theory 2', *Games Medleys*, **17**, 4–7 (1992).
15. D. Sankoff and J. B. Kruskal, *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.