More Games of No Chance
MSRI Publications
Volume **42**, 2002

Searching for Spaceships

DAVID EPPSTEIN

Abstract. We describe software that searches for spaceships in Conway's Game of Life and related two-dimensional cellular automata. Our program searches through a state space related to the de Bruijn graph of the automaton, using a method that combines features of breadth first and iterative deepening search, and includes fast bit-parallel graph reachability and path enumeration algorithms for finding the successors of each state. Successful results include a new $2c/7$ spaceship in Life, found by searching a space with 2^{126} states.

1. Introduction

John Conway's Game of Life has fascinated and inspired many enthusiasts, due to the emergence of complex behavior from a very simple system. One of the many interesting phenomena in Life is the existence of gliders and spaceships: small patterns that move across space. When describing gliders, spaceships, and other early discoveries in Life, Martin Gardner wrote (in 1970) that spaceships "are extremely hard to find" [10]. Very small spaceships can be found by human experimentation, but finding larger ones requires more sophisticated methods. Can computer software aid in this search? The answer is yes – we describe here a program, `gfind`, that can quickly find large low-period spaceships in Life and many related cellular automata.

Among the interesting new patterns found by `gfind` are the "weekender" $2c/7$ spaceship in Conway's Life (Figure 1, right), the "dragon" $c/6$ Life spaceship found by Paul Tooke (Figure 1, left), and a $c/7$ spaceship in the Diamoeba rule (Figure 2, top). The middle section of the Diamoeba spaceship simulates a simple one-dimensional parity automaton and can be extended to arbitrary lengths. David Bell discovered that two back-to-back copies of these spaceships form a pattern that fills space with live cells (Figure 2, bottom). The existence of infinite-growth patterns in Diamoeba had previously been posed as an open problem (with a \$50 bounty) by Dean Hickerson in August 1993, and was later

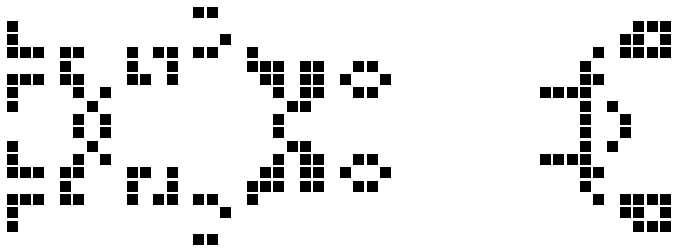

Figure 1. The "dragon" (left) and "weekender" (right) spaceships in Conway's Life (B3/S23). The dragon moves right one step every six generations (speed $c/6$) while the weekender moves right two steps every seven generations (speed $2c/7$).

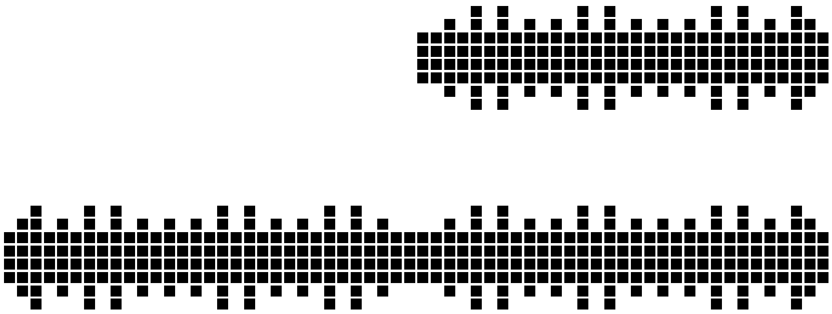

Figure 2. A $c/7$ spaceship in the "Diamoeba" rule (B35678/S5678), top, and a spacefilling pattern formed by two back-to-back spaceships.

included in a list of related open problems by Gravner and Griffeath [11]. Our program has also found new spaceships in well known rules such as HighLife and Day&Night as well as in thousands of unnamed rules.

As well as providing a useful tool for discovering cellular automaton behavior, our work may be of interest for its use of state space search techniques. Recently, Buckingham and Callahan [8] wrote "So far, computers have primarily been used to speed up the design process and fill gaps left by a manual search. Much potential remains for increasing the level of automation, suggesting that Life may merit more attention from the computer search community." Spaceship searching provides a search problem with characteristics intriguingly different from standard test cases such as the 15-puzzle or computer chess, including a large state space that fluctuates in width instead of growing exponentially at each level, a tendency for many branches of the search to lead to dead ends, and the lack of any kind of admissable estimate for the distance to a goal state. Therefore, the search community may benefit from more attention to Life.

The software described here, a database of spaceships in Life-like automata, and several programs for related computations can be found online at http://www.ics.uci.edu/~eppstein/ca/.

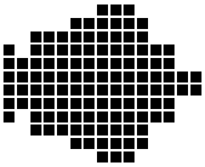

Figure 3. Large $c/60$ spaceship in rule B36/S035678, found by brute force search.

2. A Brief History of Spaceship Searching

According to Berlekamp et al. [5], the $c/4$ diagonal glider in Life was first discovered by simulating the evolution of the R-pentomino, one of only 21 connected patterns of at most five live cells. This number is small enough that the selection of patterns was likely performed by hand. Gliders can also be seen in the evolution of random initial conditions in Life as well as other automata such as B3/S13 [14], but this technique often fails to work in other automata due to the lack of large enough regions of dead cells for the spaceships to fly through. Soon after the discovery of the glider, Life's three small $c/2$ orthogonal spaceships were also discovered.

Probably the first automatic search method developed to look for interesting patterns in Life and other cellular automata, and the method most commonly programmed, is a brute force search that tests patterns of bounded size, patterns with a bounded number of live cells, or patterns formed out of a small number of known building blocks. These tests might be exhaustive (trying all possible patterns) or they might perform a sequence of trials on randomly chosen small patterns. Such methods have found many interesting oscillators and other patterns in Life, and Bob Wainwright collected a large list of small spaceships found in this way for many other cellular automaton rules [19]. Currently, it is possible to try all patterns that fit within rectangles of up to 7×8 cells (assuming symmetric initial conditions), and this sort of exhaustive search can sometimes find spaceships as large as 12×15 (Figure 3). However, brute force methods have not been able to find spaceships in Life with speeds other than $c/2$ and $c/4$.

The first use of more sophisticated search techniques came in 1989, when Dean Hickerson wrote a backtracking search program which he called `LS`. For each generation of each cell inside a fixed rectangle, `LS` stored one of three states: unknown, live, or dead. `LS` then attempted to set the state of each unknown cell by examining neighboring cells in the next and previous generations. If no unknown cell's state could be determined, the program performed a depth first branching step in which it tried both possible states for one of the cells. Using this program, Hickerson discovered many patterns including Life's $c/3$, $c/4$, and $2c/5$ orthogonal spaceships. Hartmut Holzwart used a similar program to find many variant $c/2$ and $c/3$ spaceships in Life, and related patterns including the "spacefiller" in which four $c/2$ spaceships stretch the corners of a growing

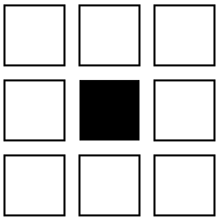

Figure 4. The eight neighbors in the Moore neighborhood of a cell.

diamond shaped still life. David Bell reimplemented this method in portable C, and added a fourth "don't care" state; his program, `lifesrc`, is available at http://www.canb.auug.org.au/~dbell/programs/lifesrc-3.7.tar.gz.

In 1996, Tim Coe discovered another Life spaceship, moving orthogonally at speed $c/5$, using a program he called `knight` in the hope that it could also find "knightships" such as those in Figure 5. `Knight` used breadth first search (with a fixed amount of depth-first lookahead per node to reduce the space needs of BFS) on a representation of the problem based on *de Bruijn graphs* (described in Section 4). The search took 38 cpu-weeks of time on a combination of Pentium Pro 133 and Hypersparc processors. The new search program we describe here can be viewed as using a similar state space with improved search algorithms and fast implementation techniques. Another recent program by Keith Amling also uses a state space very similar to Coe's, with a depth first search algorithm.

Other techniques for finding cellular automaton patterns include complementation of nondeterministic finite automata, by Jean Hardouin-Duparc [12, 13]; strong connectivity analysis of de Bruijn graphs, by Harold McIntosh [16,17]; randomized hill-climbing methods for minimizing the number of cells with incorrect evolution, by Paul Callahan (http://www.radicaleye.com/lifepage/stilledit.html); a backtracking search for still life backgrounds such that an initial perturbation remains bounded in size as it evolves, by Dean Hickerson; Gröbner basis methods, by John Aspinall; and a formulation of the search problem as an integer program, attempted by Richard Schroeppel and later applied with more success by Robert Bosch [7]. However to our knowledge none of these techniques has been used to find new spaceships.

3. Notation and Classification of Patterns

We consider here only *outer totalistic* rules, like Life, in which any cell is either "live" or "dead" and in which the state of any cell depends only on its previous state and on the total number of live neighbors among the eight adjacent cells of the Moore neighborhood (Figure 4). For some results on spaceships in cellular automata with larger neighborhoods, see Evans' thesis [9]. Outer totalistic rules are described with a string of the form $\mathrm{B}x_1x_2\ldots/\mathrm{S}y_1y_2\ldots$ where the x_i are digits listing the number of neighbors required for a cell to be born (change from dead

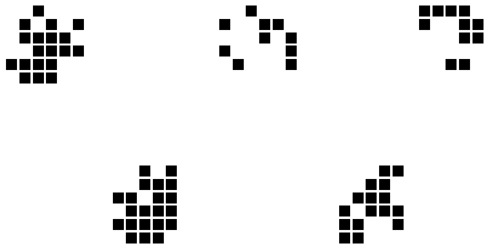

Figure 5. Slope 2 and slope 3/2 spaceships. Left to right: (a) B356/S02456, $2c/11$, slope 2. (b) B3/S01367, $c/13$, slope 2. (c) B36/S01347, $2c/23$, slope 2. (d) B34578/S358, $2c/25$, slope 2. (e) B345/S126, $3c/23$, slope 3/2.

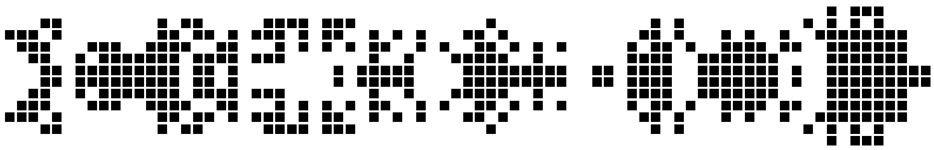

Figure 6. Long narrow c/6 spaceship in Day&Night (B3678/S34678).

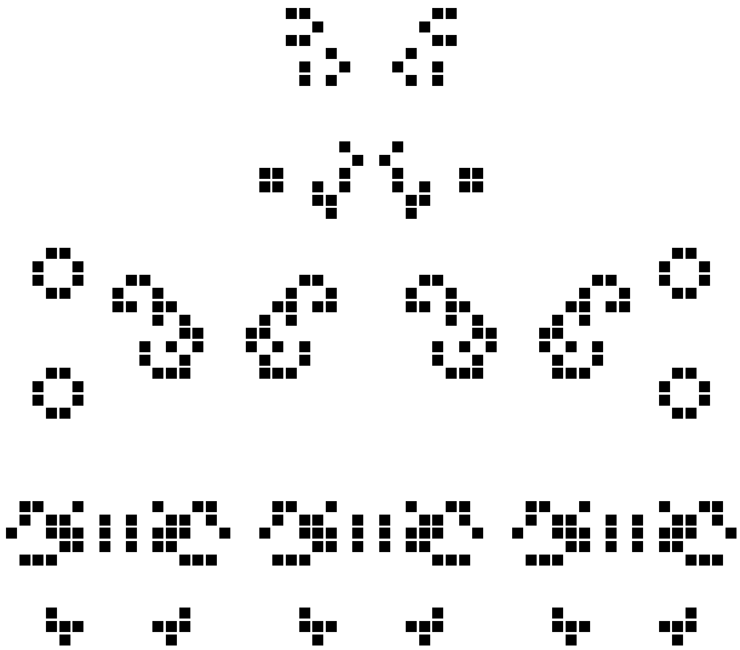

Figure 7. $9c/28$ spaceship formed from 10 R-pentomino puffers in B37/S23.

to live) while the y_i list the number of neighbors required for a cell to survive (stay in the live state after already being live). For instance, Conway's Life is described in this notation as B3/S23.

A *spaceship* is a pattern which repeats itself after some number p of generations, in a different position from where it started. We call p the *period* of the spaceship. If the pattern moves x units horizontally and y units vertically every p steps, we say that it has slope y/x and speed $\max(|x|,|y|)c/p$, where c denotes the maximum speed at which information can propagate in the automaton (the so-called *speed of light*). Most known spaceships move orthogonally or diagonally, and many have an axis of symmetry parallel to the direction of motion. Others, such as the glider and small $c/2$ spaceships in Life, have *glide-reflect symmetry*: a mirror image of the original pattern appears in generations $p/2$, $3p/2$, etc.; spaceships with this type of symmetry must also move orthogonally or diagonally. However, a few asymmetric spaceships move along lines of slope 2 or even 3/2 (Figure 5). According to Berlekamp et al. [5], there exist spaceships in Life that move with any given rational slope, but the argument for the existence of such spaceships is not very explicit and would lead to extremely large patterns.

Related types of patterns include oscillators (patterns that repeat in the same position), still lifes (oscillators with period 1), puffers (patterns which repeat some distance away after a fixed period, leaving behind a trail of discrete patterns such as oscillators, still lifes, and spaceships), rakes (spaceship puffers), guns (oscillators which send out a moving trail of discrete patterns such as spaceships or rakes), wickstretchers (patterns which leave behind one or more connected stable or oscillating regions), replicators (patterns which produce multiple copies of themselves), breeders (patterns which fill a quadratically-growing area of space with discrete patterns, for instance replicator puffers or rake guns), and spacefillers (patterns which fill a quadratically-growing area with one or more connected patterns). See Paul Callahan's Life Pattern Catalog (http://www.radicaleye.com/lifepage/patterns/contents.html) for examples of many of these types of patterns in Life, or http://www.ics.uci.edu/~eppstein/ca/replicators/ for a number of replicator-based patterns in other rules.

We can distinguish among several classes of spaceships, according to the methods that work best for finding them.

- Spaceships with small size but possibly high period can be found by brute force search. The patterns depicted in Figures 3 and 5 fall into this class, as do Life's glider and c/2 spaceships.
- Spaceships in which the period and one dimension are small, while the other dimension may be large, can be found by search algorithms similar to the ones described in this paper. The new Life spaceships in Figure 1 fall into this class. For a more extreme example of a low period ship which is long but narrow, see Figure 6. "Small" is a relative term—our search program has found $c/2$ spaceships with minimum dimension as high as 42 (Figure 10) as well as narrower spaceships with period as high as nine.
- Sometimes small non-spaceship objects, such as puffers, wickstretchers, or replicators, can be combined by human engineering into a spaceship. For in-

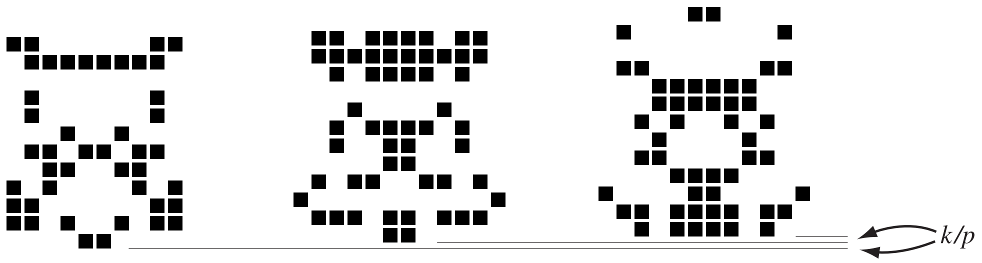

Figure 8. The three phases of the $c/3$ "turtle" Life spaceship, shifted by $1/3$ cell per phase.

stance, in the near-Life rule B37/S23, the R-pentomino pattern acts as an (unstable) puffer. Figure 7 depicts a $9c/28$ spaceship in which a row of six pentominoes stabilize each other while leaving behind a trail of still lifes, which are cleaned up by a second row of four pentominoes. Occasionally, spaceships found by our search software will appear to have this sort of structure (Figure 10). The argument for the existence of Life spaceships with any rational slope would also lead to patterns of this type. Several such spaceships have been constructed by Dean Hickerson, notably the $c/12$ diagonal "Cordership" in Conway's Life, discovered by him in 1991. Hickerson's web page http://www.math.ucdavis.edu/~dean/RLE/slowships.html has more examples of structured ships.

- The remaining spaceships have large size, large period, and little internal structure. We believe such spaceships should exist, but none are known and we know of no effective method for finding them.

4. State Space

Due to the way our search is structured, we need to arrange the rows from all phases of the pattern we are searching for into a single sequence. We now show how to do this in a way that falls out naturally from the motion of the spaceship.

Suppose we are searching for a spaceship that moves k units down every p generations. For simplicity of exposition, we will assume that $\gcd(k,p) = 1$. We can then think of the spaceship we are searching for as living in a cellular automaton modified by shifting the grid upward k/p units per generation. In this modified automaton, the shifting of the grid exactly offsets the motion of the spaceship, so the ship acts like an oscillator instead of like a moving pattern. We illustrate this shifted grid with the turtle, a $c/3$ spaceship in Conway's Life (Figure 8).

Because of the shifted grid, and the assumption that $\gcd(k,p) = 1$, each row of each phase of the pattern exists at a distinct vertical position. We form a

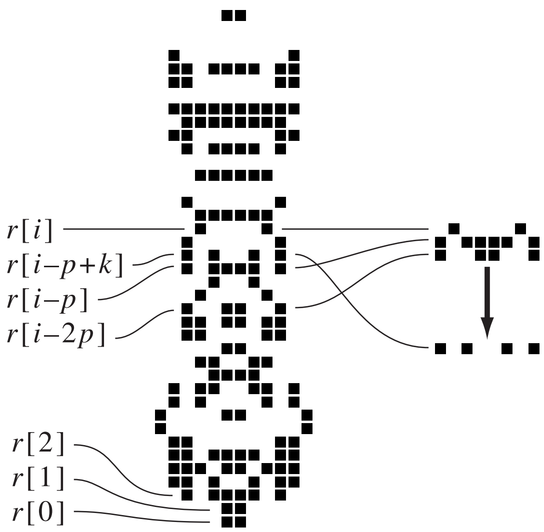

Figure 9. Merged sequence of rows from all three phases of the turtle, illustrating equation $(*)$.

doubly-infinite sequence

$$\ldots r[-2], r[-1], r[0], r[1], r[2] \ldots$$

of rows, by taking each row from each phase in order by the rows' vertical positions (Figure 9). For any i, the three rows $r[i-2p], r[i-p], r[i]$ form a contiguous height-three strip in a single phase of the pattern, and we can apply the cellular automaton rule within this strip to calculate

$$r[i-p+k] = \text{evolve}(r[i-2p], r[i-p], r[i]). \tag{$*$}$$

Conversely, any doubly-infinite sequence of rows, in which equation $(*)$ is satisfied for all i, and in which there are only finitely many live cells, corresponds to a spaceship or sequence of spaceships. Further, any finite sequence of rows can be extended to such a doubly infinite sequence if the first and last $2p$ rows of the finite sequence contain only dead cells.

Diagonal spaceships, such as Life's glider, can be handled in this framework by modifying equation $(*)$ to shift rows i, $i-p$, and $i-2p$ with respect to each other before performing the evolution rule. We can also handle glide-reflect spaceships such as Life's small $c/2$ spaceships, by modifying equation $(*)$ to reverse the order of the cells in row $i-p+k$ (when k is odd) or in the two rows $i-p$ and $i-p+k$ (when p is odd). Note that, by the assumption that $\gcd(k,p) = 1$, at least one of k and p will be odd. In these cases, p should be considered as the half-period of the spaceship, the generation at which a flipped copy of the original pattern appears. Searches for which $\gcd(k,p) > 1$ can be handled by adjusting the indices in equation $(*)$ depending on the phase to which row $r[i]$ belongs.

Our state space, then, consists of finite sequences of rows, such that equation $(*)$ is satisfied whenever all four rows in the equation belong to the finite sequence. The initial state for our search will be a sequence of $2p$ rows, all of which contain only dead cells. If our search discovers another state in which the last $2p$ rows also contain only dead cells, it outputs the pattern formed by every pth row of the state as a spaceship.

As in many game playing programs, we use a *transposition table* to detect equivalent states, avoid repeatedly searching the same states, and stop searching in a finite amount of time even when the state space may be infinite. We would like to define two states as being equivalent if, whenever one of them can be completed to form a spaceship, the same completion works for the other state as well. However, this notion of equivalence seems too difficult to compute (it would require us to be able to detect states that can be completed to spaceships, but if we could do that then much of our search could be avoided). So, we use a simpler sufficient condition: two states are equivalent if their last $2p$ rows are identical. If our transposition table detects two equivalent states, the longer of the two is eliminated, since it can not possibly lead to the shortest spaceship for that rule and period.

We can form a finite directed graph, the *de Bruijn graph* [16,17], by forming a vertex for each equivalence class of states in our state space, and an edge between two vertices whenever a state in one equivalence class can be extended to form a state in the other class. The size of the de Bruijn graph provides a rough guide to the complexity of a spaceship search problem. If we are searching for patterns with width w, the number of vertices in this de Bruijn graph is 2^{2pw}. For instance, in the search for the weekender spaceship, the effective width was nine (due to an assumption of bilateral symmetry), so the de Bruijn graph contained 2^{126} vertices. Fortunately, most of the vertices in this graph were unreachable from the start state.

Coe's search program `knight` uses a similar state space formed by sequences of pattern rows, but only uses the rows from a single phase of the spaceship. In place of our equation $(*)$, he evolves subsequences of $2p+1$ rows for p generations and tests that the middle row of the result matches the corresponding row of the subsequence. As with our state space, one can form a (different) de Bruijn graph by forming equivalence classes according to the last $2p$ rows of a state. Coe's approach has some advantages; for instance, it can find patterns in which some phases exceed the basic search width (as occurred in Coe's $c/5$ spaceship). However, it does not seem to allow the fast neighbor-finding techniques we describe in Section 7.

5. Search Strategies

There are many standard algorithms for searching state spaces [20], however each has some drawbacks in our application:

- Depth first search requires setting an arbitrary depth limit to avoid infinite recursion. The patterns it finds may be much longer than necessary, and the search may spend a long time exploring deep regions of the state space before reaching a spaceship. Further, DFS does not make effective use of the large amounts of memory available on modern computers.
- Breadth first search is very effective for small searches, but quickly runs out of memory for larger searches, even when large amounts of memory are available.
- Depth first iterative deepening [15] has been proposed as a method of achieving the fast search times of breadth first search within limited space. However, our state space often does not have the exponential growth required for iterative deepening to be efficient; rather, as the search progresses from level to level the number of states in the search frontier can fluctuate up and down, and typically has a particularly large bulge in the earlier levels of the search. The overall depth of the search (and hence the number of deepening iterations) can often be as large as several hundred. For these reasons, iterative deepening can be much slower than breadth first search. Further, the transposition table used to detect equivalent states does not work as well with depth first as with breadth first search: to save space, we represent this table as a collection of pointers to states, rather than explicitly listing the $2p$ rows needed to determine equivalence, so when searching depth first we can only detect repetitions within the current search path. Finally, depth first search does not give us much information about the speed at which the search is progressing, which we can use to narrow the row width when the search becomes too slow.
- Other techniques such as the A^* algorithm, recursive best first search, and space-bounded best first search, depend on information unavailable in our problem, such as varying edge weights or admissable estimates of the distance to a solution.

Therefore, we developed a new search algorithm that combines the best features of breadth first and iterative deepening search, and that takes advantage of the fact that, in our search problem, many branches of the search eventually lead only to dead ends. Our method resembles the MREC algorithm of Sen and Bagchi [18], in that we perform deepening searches from the breadth first search frontier, however unlike MREC we use the deepening stages to prune the search tree, allowing additional breadth first searching.

Our search algorithm begins by performing a standard breadth first search. We represent each state as a single row together with a pointer to its predecessor, so the search must maintain the entire breadth first search tree. By default, we allocate storage for 2^{22} nodes in the tree, which is adequate for small searches yet well within the memory limitations of most computers.

On larger searches (longer than a minute or so), the breadth first search will eventually run out of space. When this happens, we suspend the breadth first search and perform a round of depth first search, starting at each node of the current breadth first search queue. This depth first search has a depth limit which is normally δ levels beyond the current search frontier, for a small value δ that we set in our implementation to equal the period of the pattern we are searching for. (Setting δ to a fixed small constant would likely work as well.) However, if a previous round of depth first searching reached a level past the current search frontier, we instead limit the new depth first search round to δ levels beyond the previous round's limit.

When we perform a depth first search from a BFS queue node, one of three things can happen. First, we may discover a spaceship; in that case we terminate the entire search. Second, we may reach the depth limit; in that case we terminate the depth first search and move on to the next BFS queue node. Third, the depth first search may finish normally, without finding any spaceships or deep nodes. In this case, we know that the root of the search leads only to dead ends, and we remove it from the breadth first search queue.

After we have performed this depth first search for all nodes of the queue, we compact the remaining nodes and continue with the previously suspended breadth first search. Generally, only a small fraction of the previous breadth first search tree remains after the compaction, leaving plenty of space for the next round of breadth first search.

There are two common modes of behavior for this searching algorithm, depending on how many levels of breadth first searching occur between successive depth first rounds. If more than δ levels occur between each depth first search round, then each depth first search is limited to only δ levels beyond the breadth first frontier, and the sets of nodes searched by successive depth first rounds are completely disjoint from each other. In this case, each node is searched at most twice (once by the breadth first and once by the depth first parts of our search), so we only incur a constant factor slowdown over the more memory-intensive pure breadth first search. In the second mode of behavior, successive depth first search rounds begin from frontiers that are fewer than δ levels apart. If this happens, the ith round of depth first search will be limited to depth $i \cdot \delta$, so the search resembles a form of iterative deepening. Unlike iterative deepening, however, the breadth first frontier always makes some progress, permanently removing nodes from the actively searched part of the state space. Further, the early termination of depth-first searches when they reach deep nodes allows our algorithm to avoid searching large portions of the state space that pure iterative deepening would have to examine.

On typical large searches, the amount of deepening can be comparable to the level of the breadth-first search frontier. For instance, in the search for the weekender, the final depth first search round occurred when the frontier had reached level 90, and this round searched from each frontier node to an additional 97 levels. We allow the user to supply a maximum value for the deepening amount; if this maximum is reached, the state space is pruned by reducing the row width by one cell and the depth first search limit reverts back to δ.

There is some possibility that a spaceship found in one of the depth first searches may be longer than the optimum, but this has not been a problem in practice. Even this small amount of suboptimality could be averted by moving on to the next node instead of terminating the search when the depth first phase of the algorithm discovers a spaceship.

6. Lookahead

Suppose that our search reaches state

$$S = r[0], r[1], \dots r[i-1].$$

The natural set of neighboring states to consider would be all sequences of rows

$$r[0], r[1], \dots r[i-1], r[i]$$

where the first i rows match state S and we try all choices of row $r[i]$ that satisfy equation $(*)$.

However, it is likely that some of these choices will result in inconsistent states for which equation $(*)$ can not be satisfied the next time the new row $r[i]$ is involved in the equation. Since that next time will not occur until we choose row $r[i+p-k]$, any work performed in the intermediate levels of the search between these two rows could be wasted. To avoid this problem, when making choices for row $r[i]$, we simultaneously search for pairs of rows $r[i]$ and $r[i+p-k]$ satisfying both equation $(*)$ and its shifted form

$$r[i] = \mathrm{evolve}(r[i-p-k], r[i-k], r[i+p-k]). \tag{L}$$

Note that $r[i-p-k]$ and $[i-k]$ are both already present in state S and so do not need to be searched for. We use as the set of successors to row S the sequences of rows $r[0], r[1], \dots r[i-1], r[i]$ such that equation $(*)$ is true and equation (L) has a solution.

One could extend this idea further, and (as well as searching for rows $r[i]$ and $r[i+p-k]$) search for two additional rows $r[i+p-2k]$ and $r[i+2p-2k]$ such that the double lookahead equations

$$\begin{aligned} r[i-k] &= \mathrm{evolve}(r[i-p-2k], r[i-2k], r[i+p-2k]) \quad \text{and} \\ r[i+p-k] &= \mathrm{evolve}(r[i-2k], r[i+p-2k], r[i+2p-2k]) \end{aligned} \tag{LL}$$

have a solution. However this double lookahead technique would greatly increase the cost of searching for the successor states of S and provide only diminished

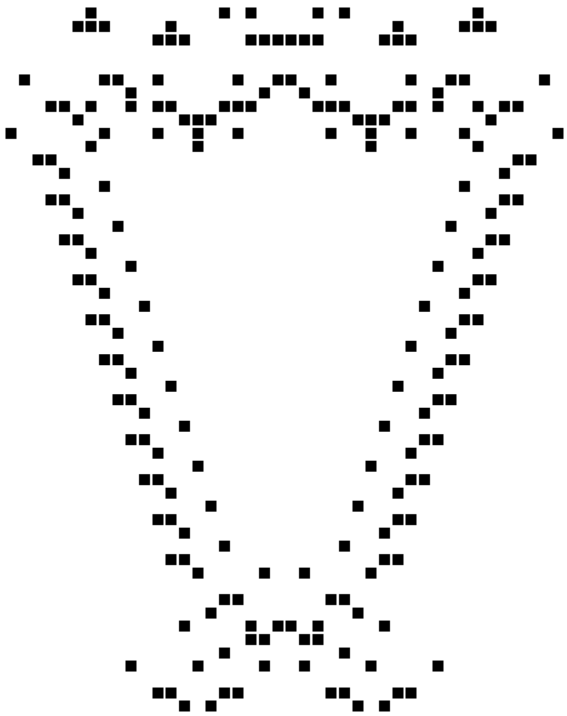

Figure 10. 42×53 $c/2$ spaceship in rule B27/S0.

returns. In our implementation, we use a much cheaper approximation to this technique: for every three consecutive cells of the row $r[i+p-k]$ being searched for as part of our single lookahead technique, we test that there could exist five consecutive cells in rows $r[i+p-2k]$ and $r[i+2p-2k]$ satisfying equations (LL) for those three cells. The advantage of this test over the full search for rows $r[i+p-2k]$ and $r[i+2p-2k]$ is that it can be performed with simple table lookup techniques, described in the next section.

For the special case $p = 2$, the double lookahead technique described above does not work as well, because $r[i+p-2k] = r[i]$. Instead, we perform a reachability computation on a de Bruijn graph similar to the one used for our state space, five cells wide with don't-care boundary conditions, and test whether triples of consecutive cells from the new rows $r[i]$ and $r[i+p-k]$ correspond to a de Bruijn graph vertex that can reach a terminal state. This technique varies considerably in effectiveness, depending on the rule: for Life, 18.5% of the 65536 possible patterns are pruned as being unable to reach a terminal state, and for B27/S0 (Figure 10) the number is 68.3%, but for many rules it can be 0%.

7. Fast Neighbor-Finding Algorithm

To complete our search algorithm, we need to describe how to find the successors of each state. That is, given a state

$$S = r[0], r[1], \ldots r[i-1]$$

we wish to find all possible rows $r[i]$ such that (1) $r[i]$ satisfies equation $(*)$, (2) there exists another row $r[i+p-k]$ such that rows $r[i]$ and $r[i+p-k]$ satisfy equation (L), and (3) every three consecutive cells of $r[i+p-k]$ can be part of a solution to equations (LL).

Note that (because of our approximation to equations (LL)) all these constraints involve only triples of adjacent cells in unknown rows. That is, equation $(*)$ can be phrased as saying that every three consecutive cells of $r[i]$, $r[i-p]$, and $r[i-2p]$ form a 3×3 square such that the result of the evolution rule in the center of the square is correct; and equation (L) can be phrased similarly. For this reason, we need a representation of the pairs of rows $r[i], r[i+p-k]$ in which we can access these triples of adjacent cells.

Such a representation is provided by the graph depicted in Figure 11. This graph (which like our state space can be viewed as a kind of de Bruijn graph) is formed by a sequence of columns of vertices. Each column contains 16 vertices, representing the 16 ways of forming a 2×2 block of cells. We connect a vertex in one column with a vertex in the next column whenever the two blocks overlap to form a single 2×3 block of cells; that is, an edge exists whenever the right half of the left 2×2 block matches the left half of the right 2×2 block.

If we have any two rows of cells, one placed above the other, we can form a path in this graph in which the vertices correspond to 2×2 blocks drawn from the pair of rows. Conversely, the fact that adjacent blocks are required to match implies that any path in this graph corresponds to a pair of rows.

Since each triple of adjacent cells from the two rows corresponds to an edge in this graph, the constraints of equations $(*)$, (L), and (LL) can be handled simply by removing the edges from the graph that correspond to triples not satisfying those constraints. In this constrained subgraph, any path corresponds to a pair of rows satisfying all the constraints. Thus, our problem has been reduced to finding the constrained subgraph, and searching for paths through it between appropriately chosen start and end terminals. The choice of which vertices to use as terminals depends on the symmetry type of the pattern we are searching for.

This search can be understood as being separated into three stages, although our actual implementation interleaves the first two of these.

In the first stage, we find the edges of the graph corresponding to blocks of cells satisfying the given constraints. We represent the set of 64 possible edges between each pair of columns in the graph (as shown in Figure 11) as a 64-bit quantity, where a bit is nonzero if an edge exists and zero otherwise. The set

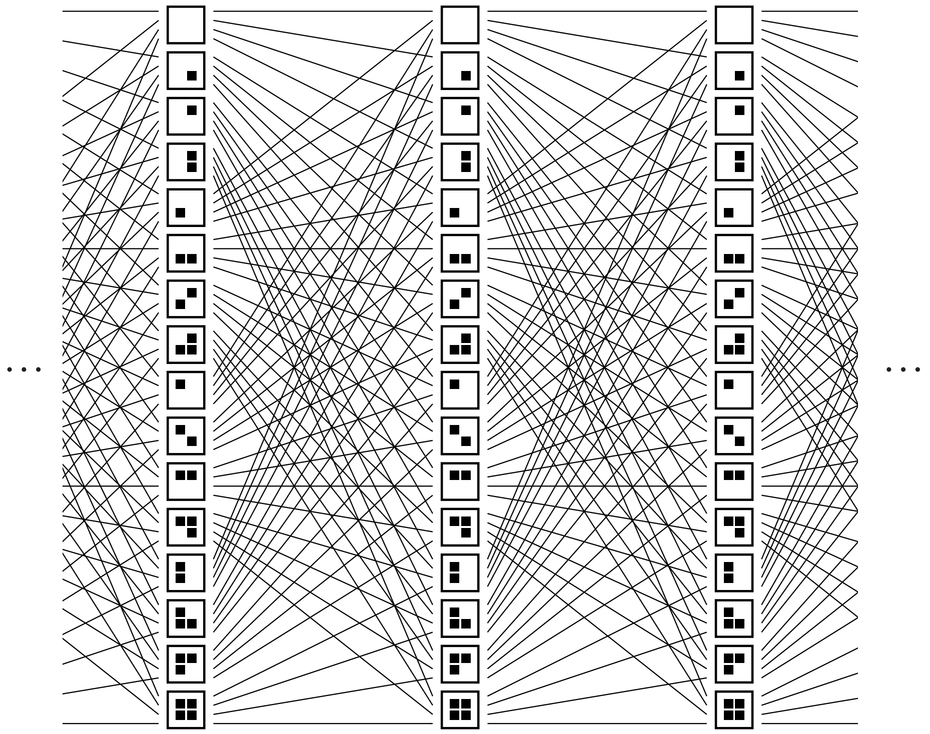

Figure 11. Graph formed by placing each of 16 2×2 blocks of cells in each column, and connecting a block in one column to a block in the next column whenever the two blocks overlap to form a 2×3 block. Paths in the graph represent pairs of rows $r[i], r[i+p-k]$; by removing edges from the graph we can constrain triples of adjacent cells from these rows.

of edges corresponding to blocks satisfying equation $(*)$ can be found by a table lookup with an index that encodes the values of three consecutive cells of $r[i-p]$ and $r[i-2p]$ together with one cell of $r[i-p+k]$. Similarly, the set of edges corresponding to blocks satisfying equation (L) can be found by a table lookup with an index that encodes the values of three consecutive cells of $r[i-k]$ and $r[i-p-k]$. In our implementation, we combine these two constraints into a single table lookup. Finally, the set of edges corresponding to blocks satisfying equations (LL) can be found by a table lookup with an index that encodes the values of five consecutive cells of $r[i-2k]$ and $r[i-p-2k]$ together with three consecutive cells of $r[i-k]$. The sets coming from equations $(*)$, (L), and (LL) are combined with a simple bitwise Boolean and operation. The various tables used by this stage depend on the cellular automaton rule, and are precomputed based on that rule before we do any searching.

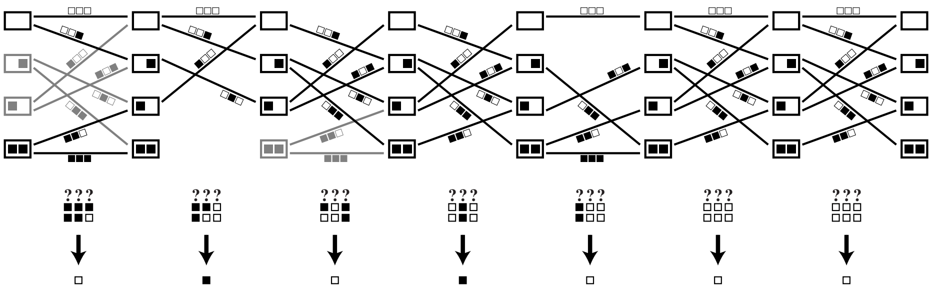

Figure 12. Simplified example (without lookahead) of graph representing equation $(*)$ for the rows from Fig. 9. Gray regions represent portions of the graph not reachable from the two start vertices.

In the second stage, we compute a 16-bit quantity for each column of vertices, representing the set of vertices in that column that can be reached from the start terminal. This set can be computed from the set of reachable vertices in the previous column by a small number of shift and mask operations involving the sets of edges computed in the previous stage.

In the third stage, we wish to find the actual rows $r[i]$ that correspond to the paths in the constrained graph. We perform a backtracking search for these rows, starting with the end terminal of the graph. At each step, we maintain a 16-bit quantity, representing the set of vertices in the current column of the graph that could reach the end terminal by a path matching the current partial row. To find the possible extensions of a row, we find the predecessors of this set of vertices in the graph by a small number of shift and mask operations (resembling those of the previous stage) and separate these predecessors into two subsets, those for which the appropriate cell of $r[i]$ is alive or dead. We then continue recursively in one or both of these subsets, depending on whichever of the two has a nonempty intersection with the set of vertices in the same column that can reach the start vertex.

Because of the reachability computation in the second stage, the third stage never reaches a dead end in which a partial row can not be extended. Therefore, this algorithm spends a total amount of time at most proportional to the width of the rows (in the first two stages) plus the width times the number of successor states (in the third stage). The time for the overall search algorithm is bounded by the width times the number of states reached.

A simplified example of this graph representation of our problem is depicted in Figure 12, which shows the graph formed by the rows from the Life turtle pattern depicted in Figure 9. To reduce the complexity of the figure, we have only incorporated equation $(*)$, and not the two lookahead equations. Therefore, the vertices in each column represent the states of two adjacent cells in row $r[i]$ only, instead of also involving row $r[i+p-k]$, and there are four vertices per column instead of sixteen. Each edge represents the state of three adjacent cells of $r[i]$, and connects vertices corresponding to the left two and right two of these cells; we have marked each edge with the corresponding cell states.

Due to the symmetry of the turtle, the effective search width is six, so we need to enforce equation $(*)$ for six cells of row $r[i-p+k]$. Six of the seven columns of edges in the graph correspond to these cells; the seventh represents a cell outside the search width which must remain blank to prevent the pattern from growing beyond its assigned bounds. Below each of these seven columns, we have shown the cells from previous rows $r[i-p]$, $r[i-2p]$, and $r[i-p+k]$ which determine the set of edges in the column and which are concatenated together to form the table index used to look up this set of edges.

The starting vertices in this example are the top and bottom left ones in the graph, which represent the possible states for the center two cells of $r[i]$ that preserve the pattern's symmetry. The destination vertex is the upper right one; it represents the state of two cells in row $r[i]$ beyond the given search width, so both must be blank. Each path from a start vertex to the destination vertex represents a possible choice for the cells in row $r[i]$ that would lead to the correct evolution of row $r[i-p+k]$ according to the rules of Conway's life. There are 13 such paths in the graph shown. The reachability information computed in the second stage of the algorithm is depicted by marking unreachable vertices and edges in gray; in this example, as well as the asymmetric states in the first column, there is one more unreachable vertex.

For this simplified example, a list of all 13 paths in this graph could be found in the third stage by a recursive depth first search from the destination vertex backwards, searching only the black edges and vertices. Thus even this simplified representation reduces the number of row states that need to be considered from 64 to 13, and automatically selects only those states for which the evolution rule leads to the correct outcome. The presence of lookahead complicates the third stage in our actual program since multiple paths can correspond to the same value of row $r[i]$; the recursive search procedure described above finds each such value exactly once.

8. Conclusions

To summarize, we have described an algorithm that finds spaceships in outer totalistic Moore neighborhood cellular automata, by a hybrid breadth first iterative deepening search algorithm in a state space formed by partial sequences of pattern rows. The algorithm represents the successors of each state by paths in a regularly structured graph with roughly $16w$ vertices; this graph is constructed by performing table lookups to quickly find the sets of edges representing the constraints of the cellular automaton evolution rule and of our lookahead formulations. We use this graph to find a state's successors by performing a 16-way bit-parallel reachability algorithm in this graph, followed by a recursive backtracking stage that uses the reachability information to avoid dead ends.

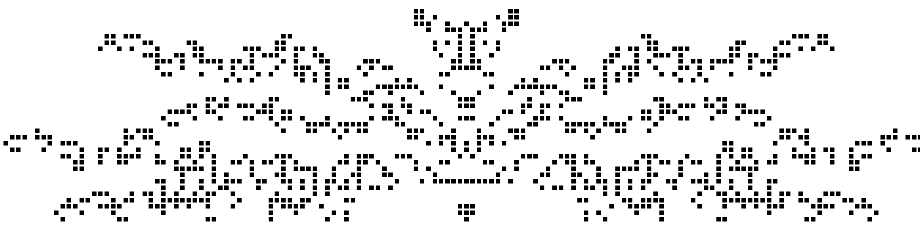

Figure 13. Large period-114 $c/3$ blinker puffer in Life, found by David Bell, Jason Summers, and Stephen Silver using a combination of automatic and human-guided searching.

Empirically, the algorithm works well, and is able to find large new patterns in many cellular automaton rules.

This work raises a number of interesting research questions, beyond the obvious one of what further improvements are possible to our search program:

- The algorithm described here, and the two search algorithms previously used by Hickerson and Coe, use three different state spaces. Do these spaces really lead to different asymptotic search performance, or are they merely three different ways of looking at the same thing? Can one make any theoretical arguments for why one space might work better than another?
- Is it possible to explain the observed fluctuations in size of the levels of the state space? A hint of an explanation for the fact that fluctuations exist comes from the idea that we typically run as narrow as possible a search as we can to find a given period spaceship. Increasing the width seems to increase the branching factor, and perhaps we should expect to find spaceships as soon as the state space develops infinite branches, at which point the branching factor will likely be quite close to one. However this rough idea depends on the unexplained assumption that the start of a spaceship is harder to find than the tail, and it does not explain other features of the state space size such as a large bulge near the early levels of the search.
- We have mentioned in Section 4 that one can use the size of the de Bruijn graph as a rough guide to the complexity of the search. However, our searches typically examine far fewer nodes than are present in the de Bruijn graph. Further, there seem to be other factors that can influence the search complexity; for instance, increasing k seems to decrease the running time, so that e.g. a width-nine search for a $c/7$ ship in Life would likely take much more time than the search for the weekender. Can we find a better formula for predicting search run time?
- What if anything can one say about the computational complexity of spaceship searching? Is the problem of determining whether a given outer totalistic rule has a spaceship of a given speed or period even decidable?

- David Bell and others have had success finding large spaceships with "arms" (Figure 13) by examining partial results from an automatic search, placing "don't care" cells at appropriate connection points, and then doing secondary searches for arms that can complete the pattern from each connection point. To what extent can this human-guided search procedure be automated?
- What other types of patterns can be found by our search techniques? For instance, one possibility would be a search for predecessors of a given pattern. The rows of each predecessor satisfy a consistency condition similar to equation ($*$), and it would not be difficult to incorporate a lookahead technique similar to the one we use for spaceship searching. However due to the fixed depth of the search it seems that depth first search would be a more appropriate strategy than the breadth first techniques we are using in our spaceship searches.
- Carter Bays has had some success using brute force methods to find small spaceships in various three-dimensional rules [1, 2, 4], and rules on the triangular planar lattice [3]. How well can other search methods such as the ones described here work for these types of automata?
- To what areas other than cellular automata can our search techniques be applied? A possible candidate is in document improvement, where Bern, Goldberg, and others [6] have been developing algorithms that look for a single high-resolution image that best matches a given set of low-resolution samples of the same character or image. Our graph based techniques could be used to replace a local optimization technique that changes a single pixel at a time, with a technique that finds an optimal assignment to an entire row of pixels, or to any other linear sequence such as the set of pixels around the boundary of an object.

Acknowledgements

Thanks go to Matthew Cook, Nick Gotts, Dean Hickerson, Harold McIntosh, Gabriel Nivasch, and Bob Wainwright for helpful comments on drafts of this paper; to Noam Elkies, Rich Korf, and Jason Summers for useful suggestions regarding search algorithms; to Keith Amling, David Bell, and Tim Coe, for making available the source codes of their spaceship searching programs; and to Richard Schroeppel and the members of the Life mailing list for their support and encouragement of cellular automaton research.

References

[1] C. Bays. Candidates for the game of Life in three dimensions. *Complex Systems* 1(3):373–400, June 1987.

[2] C. Bays. The discovery of a new glider for the game of three-dimensional Life. *Complex Systems* 4(6):599–602, December 1990.

[3] C. Bays. Cellular automata in the triangular tessellation. *Complex Systems* 8(2):127–150, April 1994.

[4] C. Bays. Further notes on the game of three-dimensional Life. *Complex Systems* 8(1):67–73, February 1994.

[5] E. R. Berlekamp, J. H. Conway, and R. K. Guy. What is Life? *Winning Ways For Your Mathematical Plays*, vol. 2, chapter 25, pp. 817–850. Academic Press, 1982.

[6] M. Bern and D. Goldberg. Scanner-model-based document image improvement. *Proc. 7th IEEE Int. Conf. Image Processing*, vol. 2, pp. 582-585, 2000.

[7] R. A. Bosch. Integer programming and Conway's game of Life. *SIAM Review* 41(3):594–604, 1999, http://epubs.siam.org/sam-bin/dbq/article/33825.

[8] D. J. Buckingham and P. B. Callahan. Tight bounds on periodic cell configurations in Life. *Experimental Mathematics* 7(3):221–241, 1998, http://www.expmath.com/restricted/7/7.3/callahan.ps.gz.

[9] K. M. Evans. *Larger Than Life: it's so nonlinear.* Ph.D. thesis, Univ. of Wisconsin, Madison, 1996, http://www.csun.edu/~kme52026/thesis.html.

[10] M. Gardner. The Game of Life, part I. *Wheels, Life, and Other Mathematical Amusements*, chapter 20, pp. 214–225. W. H. Freeman, 1983.

[11] J. Gravner and D. Griffeath. Cellular automaton growth on $\mathbb{Z}^2$: theorems, examples, and problems. *Advances in Applied Mathematics* 21(2):241–304, August 1998, http://psoup.math.wisc.edu/extras/r1shapes/r1shapes.html.

[12] J. Hardouin-Duparc. À la recherche du paradis perdu. *Publ. Math. Univ. Bordeaux Année* (4):51–89, 1972/73.

[13] J. Hardouin-Duparc. Paradis terrestre dans l'automate cellulaire de Conway. *Rev. Française Automat. Informat. Recherche Opérationnelle Sér. Rouge* 8(R-3):64–71, 1974.

[14] J.-C. Heudin. A new candidate rule for the game of two-dimensional Life. *Complex Systems* 10(5):367–381, October 1996.

[15] R. E. Korf. Depth-first iterative-deepening: an optimal admissable tree search. *Artificial Intelligence* 27:97–109, 1985.

[16] H. V. McIntosh. A zoo of Life forms, http://delta.cs.cinvestav.mx/~mcintosh/comun/zool/zoo.pdf. Manuscript, October 1988.

[17] H. V. McIntosh. Life's still lifes, http://delta.cs.cinvestav.mx/~mcintosh/comun/still/still.pdf. Manuscript, September 1988.

[18] A. K. Sen and A. Bagchi. Fast recursive formulations of best-first search that allow controlled use of memory. *Proc. 11th Int. Joint Conf. Artificial Intelligence*, vol. 1, pp. 297–302, 1989.

[19] R. T. Wainwright. Some characteristics of the smallest reported Life and alienlife spaceships. Manuscript, November 1994.

[20] W. Zhang. *State-Space Search: Algorithms, Complexity, Extensions, and Applications.* Springer Verlag, 1999.

David Eppstein
Deptartment of Information and Computer Science
University of California
Irvine, CA 92697-3425
United States
eppstein@ics.uci.edu