

Perimeter Search

John F. Dillenburg and Peter C. Nelson*

Department of Electrical Engineering
and Computer Science (M/C 154)
University of Illinois at Chicago
Chicago, Illinois 60607-7053
U.S.A.

September 27, 1993

*Please address all correspondence to Peter C. Nelson

Abstract

A technique for improving heuristic search efficiency is presented. This admissible technique is referred to as perimeter search since it relies on a perimeter of nodes around the goal. The perimeter search technique works as follows: First, the perimeter is generated by a breadth-first search from the goal to all nodes at a given depth d . The path back to the goal along with each perimeter node's state descriptor are stored in a table. The search then proceeds normally from the start state. During each node generation, however, the current node is compared to each node on the perimeter. If a match is found, the search can terminate with the path being formed with the path from the start to the perimeter node together with the previously stored path from the perimeter node to the goal. Both analytical and experimental results are presented to show that perimeter search is more efficient than IDA* and A* in terms of time complexity and number of nodes expanded for two problem domains.

Notation

s, g	Start node and goal node respectively.
n, m	Nodes in the search space.
$h(n)$	Estimate of the cost of a least cost path from n to g .
$h^*(n,m)$	The cost of a least cost path from n to m .
$h(n,m)$	Estimate of $h^*(n,m)$.
$c(n,m)$	Cost of the arc between nodes n and m .
L	Cost of the least cost path from s to g .
P_d	Set of nodes in a perimeter of depth d .
$ P_d $	Size of a perimeter of depth d .
d	Distance from g to nodes in perimeter.
t	Time per node expansion.
f	Fraction of time spent evaluating the heuristic during a node expansion.
n_d	Number of node expansions with perimeter search and a perimeter of depth d .
n_{BFS}	Number of node expansions by breadth-first search when generating perimeter.
B	Brute force branching factor ($h(n) = 0$).
b	Effective branching factor ($h(n) > 0$).
η	The number of goals.

1. Introduction

The perimeter search algorithm can be categorized as a bidirectional search algorithm. As with other bidirectional search algorithms [3][8][11][12], perimeter search requires a state space with reversible operators and a heuristic which can estimate the distance between any two nodes. Also like other bidirectional search algorithms, perimeter search attempts to reduce the complexity of heuristic search by splitting the search space between two separate searches; one from the start, and another from the goal. Unlike other bidirectional search algorithms, perimeter search avoids some of the pitfalls of attempting to do the two searches simultaneously. After performing a depth-limited breadth-first search from the goal, perimeter search proceeds as a unidirectional search from the start.

Pohl's original bidirectional heuristic path algorithm (BHPA) [11], attempts to reduce the complexity of heuristic search by performing two simultaneous searches, one from the start to the goal, the other from the goal to the start. The algorithm finds successively better paths as the two searches collide with each other. The algorithm terminates when no further improvements can be made to the path length. A best-case scenario for BHPA would be for the two searches to "meet in the middle" initially. This would reduce an $O(b^d)$ search into two $O(b^{d/2})$ searches. Unfortunately, the two search spaces often do not meet at a midpoint. The result of this, combined with satisfying BHPA's admissibility conditions, is that BHPA can end up expanding more nodes than a unidirectional algorithm.

De Champeaux and Sint eliminated the "meet in the middle" problem with their BHFFA algorithm [3] by using a heuristic which guides the nodes in each wavefront toward the closest node in the opposing wavefront. This guarantees that the first node found by BHFFA is the best one, but the time complexity of the heuristic used increases proportionally with the size of the opposing wavefront. Compounding this problem is the fact that the heuristic value for a node cannot be stored and must be recomputed every time the opposing wavefront changes.

Politowski and Pohl [12] created a bidirectional search algorithm similar to BHFFA, but with the heuristic changed so that the search was guided toward only one special node in the opposing wavefront called the d-node. The d-node is the node in the opposing wavefront furthest from the start(goal). The d-node retargeting algorithm works by performing n node expansions towards the d-node, switching directions, recomputing the heuristic values for the new d-node, and repeating the process until the wavefronts meet. Unfortunately, the d-node retargeting algorithm is inadmissible and requires a careful choice for the value of n . Choosing too large a value for n leads to a unidirectional search. Making n too small leads to lower quality paths and more overhead for recomputing the heuristic values [12].

One of the most recent developments in bidirectional search algorithms is Kwa's BS* algorithm [8]. The BS* algorithm is derived from Pohl's BHPA algorithm with refinements added

to eliminate excessive node expansions. The BS^* algorithm is provably admissible and has been shown to outperform BHPA. Unfortunately, the test results given in [8] and in section 5 show that BS^* is inferior to A^* not only in terms of running time, but also in terms of node expansions. This is probably due to the extra overhead necessary for nipping and pruning useless nodes from the OPEN lists.

The best admissible bidirectional search algorithms considered here, BHFFA and BS^* , take two different approaches to dealing with the “meet in the middle” problem. The BHFFA algorithm forces the two wavefronts to meet in the middle, but in doing so increases the cost of the heuristic exponentially. The BS^* algorithm relies upon the ability to eliminate useless nodes once the wavefronts have met in order to reduce the number of node expansions. Perimeter search takes a third approach to dealing with this problem: generating one of the wavefronts with a breadth-first search and limiting the size of the wavefront.

Limiting the size of one of the wavefronts has many important advantages. First, the wavefront only needs to be generated once for each possible goal. This allows the cost of performing the breadth-first search to be amortized over many problem instances, if necessary. In most cases the optimum wavefront size will be so small that the time it takes to generate it is insignificant. Next, fixing a depth limit on one side of the search avoids the wasteful necessity of retargeting toward a continually changing opposing wavefront as is done in the BHFFA and the d -node retargeting algorithms. Finally, the time complexity of the perimeter search algorithm can be analyzed and the optimum wavefront size can be calculated. Section 3 will show how these advantages allow perimeter search to substantially outperform other admissible heuristic search algorithms.

2. Perimeter Search Algorithms

The perimeter search algorithm consists of two searches performed in sequence. The first search, a depth-limited breadth-first search from the goal, generates the perimeter, see figure 1. The second search proceeds from the start using $\text{Min}_{m \in P}(h(n,m) + h^*(m,g))$ as the heuristic, where n is the node being expanded, P is the set of perimeter nodes, and $h^*(m,g)$ is the cost of the least cost path from perimeter node m to g . Since the two searches are completely independent, any type of heuristic search algorithm can be used for the second search. Discussion will be limited to using A^* [6] and IDA^* [7]. Perimeter search with A^* will be denoted as PS^* and perimeter search with IDA^* will be denoted as $IDPS^*$.

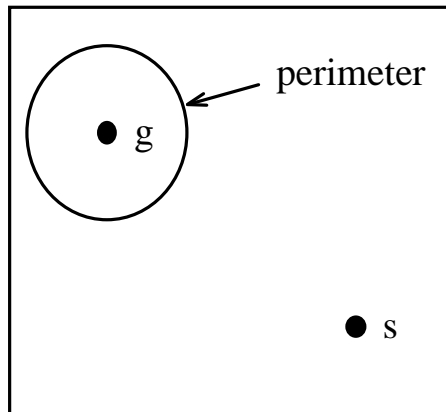


Figure 1. Hypothetical search space showing perimeter around goal g .

In addition to modifying the heuristic used in the second search, a test needs to be put in place which compares the node selected for expansion against the nodes in the perimeter. If a match is found, the second search can terminate with the solution path being formed from the path to the perimeter node together with the previously stored path from the perimeter node to the goal.

Both PS^* and $IDPS^*$ use a heuristic similar to the one in BHFFA to force the heuristic search towards the perimeter. That is, $\text{Min}_{m \in P}(h(n,m) + h^*(m,g))$ is used instead of $h(n)$ as the heuristic. For problem domains with unity cost arcs, $h^*(m,g)$ is equal to the perimeter depth d . Since d is constant throughout the search, it can be omitted leaving $\text{Min}_{m \in P}h(n,m)$ as the heuristic for unity cost problems.

Assuming $h(n,n) = 0$, it is only necessary to check for a collision with the perimeter when $\text{Min}_{m \in P}h(n,m) = 0$. Once a node in the perimeter has been selected for expansion, the search can terminate. The optimal path from s to g can be formed by appending the path from p to g onto the path from s to p , where p is the perimeter node encountered. Since PS^* and $IDPS^*$ are based upon admissible algorithms, and since $\text{Min}_{m \in P}(h(n,m) + h^*(m,g))$ is an admissible heuristic, both PS^* and $IDPS^*$ are admissible.

The perimeter search algorithm does not require the perimeter depth d to be a natural number. If the problem domain has real valued costs, then the depth-limited breadth-first search used to generate the perimeter should only add those nodes to the perimeter which have a depth greater than or equal to d . See figure 2 for an algorithm which can generate perimeters for any search domain.

```

procedure makePerimeter(State goal, Real d, var Perimeter P)
  add goal to the back of the OPEN queue, set  $g(goal) = 0$ 
  while OPEN is not empty
    Remove  $n$  from the front of the OPEN queue
    if CLOSED set does not include  $n$  then
      Add  $n$  to CLOSED set
      if  $g(n) < d$  then
        For each successor  $n_i$  of  $n$  do begin
          Add  $n_i$  to OPEN, set  $g(n_i) = g(n) + c(n, n_i)$ 
        end for
      else
        Add  $n$  to perimeter  $P$ 
      end if
    end if
  end while
end procedure makePerimeter

```

Figure 2. Algorithm for generating perimeter P of depth d

The perimeter need not be generated using a breadth-first depth limited search as shown in figure 2. In fact, any set of nodes which encompass the goal will work as a perimeter. One such alternative method of generating the perimeter would be to use A^* . This would generate a constant evaluation perimeter because all nodes on the perimeter would have approximately the same f value. The algorithm listed in figure 2 generates constant depth perimeters because all the nodes on the perimeter have approximately the same g value. This paper will focus primarily on constant depth perimeters for the following reasons: First, analyzing the number of node expansions is much easier with constant depth perimeters. Second, constant evaluation perimeters are more expensive to generate since an agenda must be maintained by A^* . Third, constant evaluations perimeters can only be used for one start/goal pair whereas constant depth perimeters can be reused as long as the goal remains the same. Finally, empirical results show that there is no significant advantage to using constant evaluation perimeters instead of constant depth perimeters.

3. Analyzing Perimeter Search

A simple analysis of the perimeter search algorithm can be used to help decide when perimeter search will improve the search efficiency for a particular problem domain. The same analysis can also be used to determine the optimal value of d to use. In the absence of such an analysis, empirical results based on small problem instances may be used in order to determine if perimeter search is useful for a particular problem domain.

Let t be the amount of CPU time necessary per node expansion with one heuristic evaluation to a single goal. This analysis assumes that t remains constant throughout the search. Let f be the fraction of time t spent evaluating the heuristic $h(n,m)$. Let n_d be the number of nodes expanded by the forward search when a perimeter of depth d is used. Finally, let η be the number of goals.

The time needed to evaluate h once for each goal during a single node expansion is $tf\eta$. The time needed for the rest of the operations during a node expansion is $t(1-f)$. Since evaluating $\text{Min}_{m \in p} h(n,m)$ requires $|P_d|$ heuristic evaluations for each goal (in the worst case, see section 4), the total amount of time spent evaluating the heuristic over all node expansions is $|P_d|tf\eta n_d$. The total amount of time spent on non-heuristic operations is $(1-f)tn_d$. Let n_{BFS} be the number of nodes expanded when creating the perimeter for one goal. Assuming the cost per node expansion is t for the backwards search, the total running time of a perimeter search algorithm is

$$R^*(t, P_d, f, n_d, d, \eta) = (|P_d|f\eta + 1 - f)tn_d + t\eta n_{BFS}. \quad (1)$$

Assuming a tree search, n_d can be estimated as b^{L-d} where b is the effective branching factor and L is the cost of a least cost path from s to g . The perimeter size and the number of nodes expanded to generate the perimeter can be estimated with B^d where B is the brute force branching factor. Substituting these into equation (1) yields

$$R(t, B, f, b, d, L, \eta) = (B^d f\eta + 1 - f)tb^{L-d} + t\eta B^d. \quad (2)$$

Equation (2) is plotted in figure 3 for $b=1.44$, $B=2.13$, $t=1$, $L=53$, and various values of f . The values for b and L correspond to empirical estimates of the branching factor and average path length for the 15-puzzle problem when using IDA* and the Manhattan distance heuristic [10]. The value for B was computed exactly, see [14].

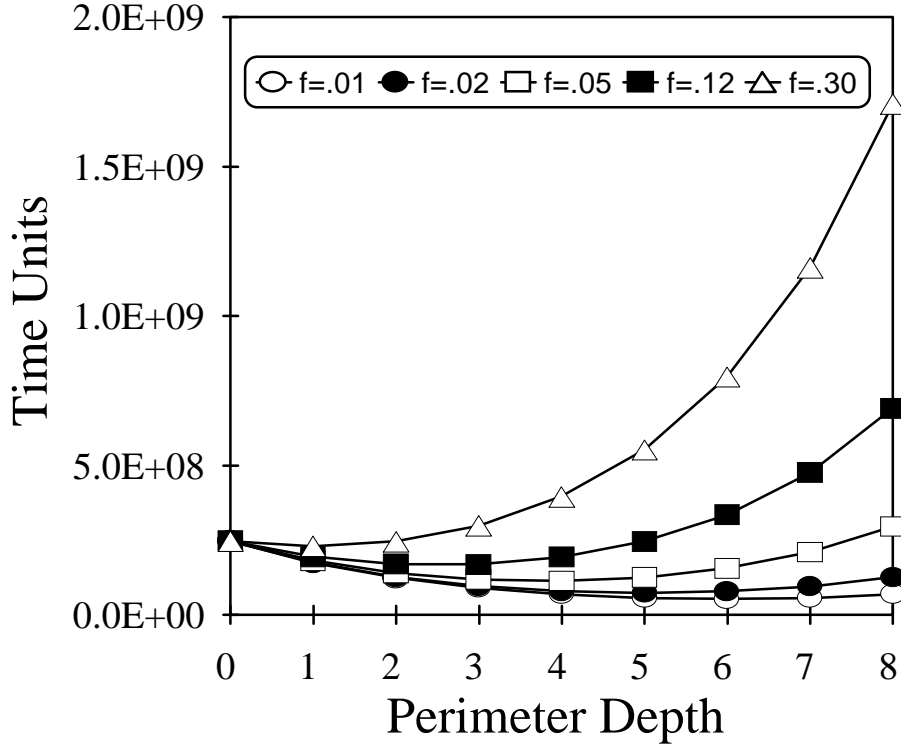


Figure 3. Analytical time complexity of perimeter search for the 15-puzzle problem

Figure 3 shows that the usefulness of perimeter search for the 15-puzzle problem depends critically on the fractional amount of time spent evaluating the heuristic. For values of f greater than about 0.3, perimeter search will no longer improve the search efficiency. The usefulness of perimeter search can be determined for other problem domains as well. Taking the derivative of equation (2) with respect to d yields

$$\frac{dR}{dd} = b^{L-d} t f \log b - b^{L-d} t \log b + b^{L-d} B^d t f \eta \log B - b^{L-d} B^d t f \eta \log b + B^d t \eta \log B. \quad (3)$$

Perimeter search is no longer useful when the minimum R occurs at $d \leq 0$. Substituting $d = 0$ into equation (3) and solving for $dR/dd = 0$ reveals that perimeter search will be useful if

$$f \leq \frac{b^L \log b - \eta \log B}{b^L (\log b - \eta \log b + \eta \log B)}. \quad (4)$$

For the case with one goal and for sufficiently large values of L , equation (4) simplifies to

$$f \leq \frac{\log b}{\log B}. \quad (5)$$

Equation (5) provides a simple test to determine when perimeter search will improve search performance. Note that equation (5) only applies to problem domains where d can be any real number. For problem domains like the 15-puzzle problem where d must be a natural number, $d = 1$ is substituted into equation (3) and $dR/dd = 0$ is solved to yield

$$f_{nat} \leq \frac{b^L \log b - bB\eta \log B}{b^L (\log b - B\eta \log b + B\eta \log B)}. \quad (6)$$

For the case with one goal and for sufficiently large values of L , equation (6) simplifies to

$$f_{nat} \leq \frac{\log b}{\log b - B \log b + B \log B}. \quad (7)$$

Substituting in the appropriate values of $b = 1.44$ and $B = 2.13$ into equation (7) reveals that perimeter search will be useful for the 15-puzzle problem if f is less than or equal to 0.304.

Equations (4) through (7) are only valid under the assumption of a tree search. For problem domains where this is not a valid assumption, a different analysis must be done based either on empirical equations for n_d and n_{BFS} , or else exact equations for n_d and n_{BFS} can be determined using methods similar to those found in [10]. Such an analysis has been done for the grid search problem domain revealing that perimeter search would not reduce the number of nodes expanded and should therefore not be used [4].

Once the utility of perimeter search has been verified, the next step is to determine the value of d which yields the minimum time complexity. This can be achieved by setting equation (3) equal to zero and solving for d . Unfortunately, there is no closed form solution to this equation and a numerical analysis must be used instead. Table 1 lists out some optimum d values for the 15-puzzle problem with one goal. Also shown in the table is the speedup obtained by using perimeter search. Speedup is defined here as the ratio

$$Speedup = \frac{R(t, B, f, b, 0, L, 1)}{R(t, B, f, b, d, L, 1)}. \quad (8)$$

f	Optimum d	$Speedup$
0.01	6	4.6
0.02	5	3.3
0.05	4	2.2
0.1	3	1.6
0.2	2	1.2
0.3	1	1.1

Table 1. Optimum values of d for the 15-puzzle problem and the corresponding search speedup

4. Minimizing Heuristic Evaluations

Table 1 shows that the performance of perimeter search is dependent on the amount of time spent evaluating the heuristic. Many of the methods for improving the heuristic evaluation

time are implementation dependent. However, there is a general technique which can be used with IDPS* or multiple goal IDA* to minimize the number of heuristic evaluations.

At first it may seem that using $\text{Min}_{m \in P} h(n, m)$ as a heuristic requires $|P|$ heuristic evaluations per node expansion. This is not the case, however, if the heuristic is monotonic. Recall that if a heuristic is monotonic, then

$$|h(n) - h(m)| \leq c(m, n) \quad (9)$$

holds for all nodes n and m , where $c(n, m)$ is the cost of traversing the arc between m and n . With a monotonic heuristic, the estimate of the distance from a node n to a perimeter node p can only change by at most $c(m, n)$, where m is the parent of n . The result is that a heuristic value does not need to be recomputed if its value cannot change enough to affect the minimum.

The technique for reducing the number of heuristic evaluations is now as follows. The heuristic is used once at the beginning of the search to estimate the distance from the start to each perimeter node and this estimate is then stored along with the node. Next, when a node n is generated from a node m , the heuristic value of the current minimum is recomputed using h . The arc cost $c(m, n)$ is then subtracted from the rest of the stored heuristic values. If any of the stored heuristic values falls below the minimum, it is recomputed using h also. Using this technique with IDPS* and a perimeter containing 4 nodes, the number of heuristic evaluations was reduced by 74% when solving 25 random 15-puzzles .

Note that this technique for reducing the number of heuristic evaluations requires $O(|P|)$ memory to store the heuristic values for every perimeter node. This is the reason the technique cannot be used with PS*, it would require $O(|P|)$ memory for each node in the OPEN list. IDPS* does not have this problem because the heuristic values only need to be stored with each node along the current path.

5. Test Results

The theoretical analysis of section 3 indicate that perimeter search can improve search efficiency quite dramatically. This section will support the theoretical analysis through experimental results which show that perimeter search improves search efficiency for the 15-puzzle and Think-A-Dot problems. All tests in this section were run on Sun Sparcstation IPX workstations. The elapsed time was calculated based on the number of CPU seconds required to solve a problem. The elapsed time for each PS* and IDPS* test includes the time needed to generate a perimeter.

5.1 15-Puzzle Problem

The 15-puzzle problem is a sliding tile problem based on a game in which fifteen numbered tiles are placed in a four by four grid. The objective of the game is to arrange

the tiles in a specified pattern by moving the tiles, one at a time and without crossing over each other, into the one blank position. The IDA* and IDPS* algorithms were used to solve random instances of the 15-puzzle problem. The A*, PS*, and BS* algorithms could not be used due to their large memory requirements.

One of the best heuristics available for the 15-puzzle problem is the Manhattan distance heuristic. The Manhattan distance heuristic estimates the number of moves needed to solve a sliding tile puzzle by summing up the distance between each tile and its “home” position. All 15-puzzle problems were solved using the Manhattan distance heuristic.

Perimeter search was shown in section 3 to be useful for the 15-puzzle problem only if $f \leq 0.3$. The actual value of f for the Manhattan distance heuristic was estimated to be approximately 0.02 by solving twenty “easy” problems. Easy problem instances were generated by making a limited number of random moves from the goal state. The low f value results from computing the Manhattan distance heuristic incrementally using a lookup table based on the current blank position, the new blank position, and the tile being moved. This low f value indicates that perimeter search should provide a speedup of at least 3, see Table 1.

The test results for the 15-puzzle problem are summarized in figure 4 and table 2. Each point in figure 4 represents the average elapsed CPU time required to solve 100 random 15-puzzles. Note that the point for perimeter depth 0 represents IDA* search and the remaining points are IDPS* searches with different perimeter depths. Table 2 shows the mean and standard deviation of six random variables: the number of node expansions for IDA* and IDPS*, the amount of CPU time for IDA* and IDPS*, and the speedup computed using node expansions and CPU time. Note that the last column in the table represents the mean of a ratio and not the ratio of two means. The test results show that IDPS* at depth 4 attained the highest speedup; it was 3.8 times faster than IDA*. This speedup surpasses the theoretical speedup of 3.3, see table 1.

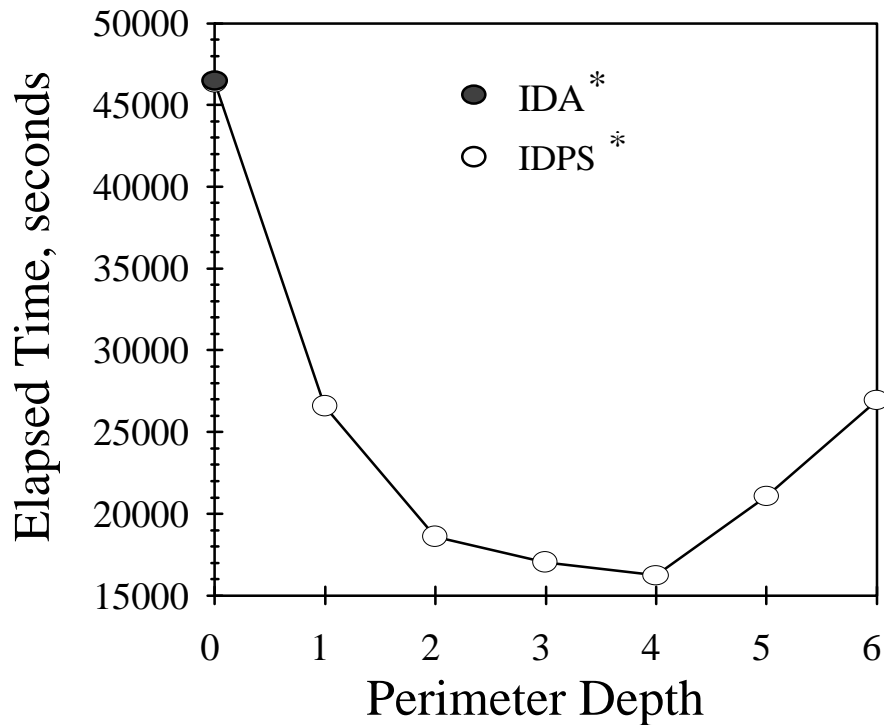


Figure 4. Test results for 15-puzzle

Number of Tests = 100 Mean Path Length = 50.6

		IDA*	IDPS*, $d = 4$	$\frac{IDA^*}{IDPS^*}$
Nodes Expansions	Mean	131,809,315	35,990,122	4.96
	Deviation	305,101,594	102,444,848	3.27
Elapsed Time, sec	Mean	46,416	16,225	3.85
	Deviation	115,069	44,772	2.68

Table 2. Summary of test results for the 15-puzzle problem.

5.2 Think-A-Dot

The Think-A-Dot problem [9] is based on a game in which a ball drops through a series of levers, see figure 5. Each lever causes the ball to drop to either the left or the right. After a ball passes by a lever, the lever switches directions so that the next ball to pass by will go the opposite direction. The objective of the game is to place all the levers in a given position by dropping balls, one at a time, into one of the four positions in the top layer. The number of moves needed to solve a Think-A-Dot problem can be estimated by finding the maximum number of incorrectly oriented levers in a single layer.

The utility of using the perimeter search algorithm was determined by estimating the values of B , b , and f by solving several easy problem instances. The value for L was assumed to be large enough so that equation (7) could be used. The other values, when substituted into equation (7), indicated that perimeter search could be used to improve search times. The optimal value for d was determined to be 4 by finding the minimum value of equation (2).

The A^* , BS^* , and PS^* algorithms were used to solve 100 random Think-A-Dot problems. An augmented red-black tree [1][5] was used for maintaining the OPEN lists used by these algorithms, resulting in $O(\log n)$ complexity for all operations performed on the lists, where n is the number of nodes in the list. The test results are summarized in figure 6 and table 3. Each point in figure 6 represents the average elapsed CPU time required to solve 100 random Think-A-Dot problem instances. Table 3 contains the mean and deviation of eight random variables: the number of node expansions for A^* , BS^* , and PS^* , the amount of CPU time for A^* , BS^* and PS^* , and the speedup relative to A^* computed using node expansions and CPU time. The results for A^* are shown at perimeter depth 0 in figure 6. The results show that PS^* at depth 4 is 1.7 times faster than A^* . The results of the BS^* tests are given in table 3. These results show that the BS^* algorithm expanded more nodes and took more time to solve the problems than either A^* or PS^* . The perimeter search algorithm PS^* had, on average, the least number of node expansions and required the least amount of running time to solve the problems.

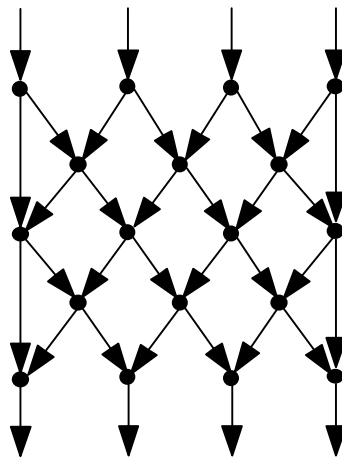


Figure 5. Think-A-Dot Problem used in tests

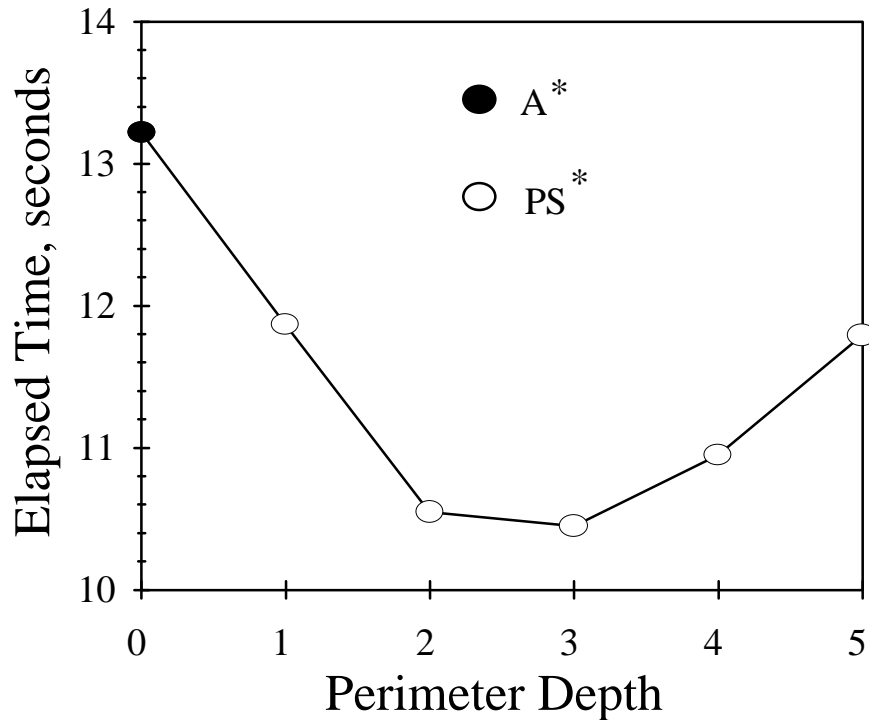


Figure 6. Test results for Think-A-Dot

Number of Tests = 100

Mean Path Length = 18.4

		A*	BS*	PS*, $d = 4$	$\frac{A^*}{PS^*}$
Nodes Expansions	Mean	5,369	5,479	2,673	4.11
	Deviation	4,214	4,782	2,916	5.13
Elapsed Time, sec	Mean	13.2	38.7	10.9	1.69
	Deviation	10.17	52.3	11.8	0.53

Table 3. Summary of test results for the Think-A-Dot problem.

6. Conclusion

A new search technique has been presented called perimeter search. Perimeter search can be viewed as a bidirectional search algorithm. Like other bidirectional search algorithms, perimeter search attempts to reduce search complexity by splitting the search space into two separate parts. Unlike other bidirectional search algorithms, perimeter search does not attempt to perform two searches simultaneously. This allows perimeter search to avoid the overhead involved in retargeting or pruning as in the d-node retargeting and BS* algorithms.

Analytical equations were presented which provide a method to predict when perimeter search will be useful for a particular problem domain. These equations can also be used to calculate the perimeter size which yields optimum performance.

Tests results performed on the 15-puzzle and Think-A-Dot problems show that perimeter search can decrease the search complexity quite substantially. For instance, the 15-puzzle results show that IDPS* is about 3.8 times faster than IDA*. The Think-A-Dot tests show that PS* is 1.7 times faster than A*. Also note that PS* expands only one quarter of the nodes expanded by A* for the Think-A-Dot problem resulting in a corresponding reduction in the amount of memory required.

The IDPS* algorithm can be viewed as a memory bounded search algorithm, with the memory bound equal to the perimeter size. Unlike memory bounded algorithms like MA* and MREC [2][13], IDPS* keeps a bottom-up memory rather than a top-down memory. A top-down memory prevents the same node from being reexpanded while memory is available. A bottom-up memory does not prevent nodes from being reexpanded, but rather eliminates the last and most expensive iterations of an IDA* search.

A near-optimal version of the perimeter search algorithm exists and may be useful for certain problems. The near-optimal perimeter search algorithm differs from the optimal algorithm in that the heuristic used is $h(n)$ and not $\text{Min}_{m \in ph} ph(n, m)$. Both versions of perimeter search must check to see if a node on the perimeter has been encountered. When a perimeter node is encountered using $\text{Min}_{m \in ph} ph(n, m)$ as the heuristic, we are guaranteed to have an optimal path from s to g . When a perimeter node is encountered using $h(n)$ as the heuristic, we are only guaranteed an optimal path from s to the particular perimeter node encountered. This perimeter node may or may not be on an optimal path from s to g and so the algorithm is inadmissible. It can be shown, however, that the solution cost obtained using near-optimal perimeter search is at most $L + d$, where L is the cost of the least cost path from s to g and d is the perimeter depth [4].

One of the main advantages of bidirectional search algorithms is the possibility of performing the forward and backward searches in parallel. Even though the perimeter search algorithm does not perform simultaneous searches from the start and from the goal, there are still plenty of opportunities for parallelization. A single processor can be assigned to each node in the perimeter. The main advantage of this type of parallel search is that almost no communication is required between the processors. The processors only need to communicate the current upper bound on the solution length when a perimeter node is located.

Acknowledgment

We would like to thank the anonymous referees for their valuable contributions to this paper.

References

- [1] R. Bayer, Symmetric Binary B-trees: Data Structures and Maintenance Algorithm, *Acta Informatica* 1 (1972) 290-306.
- [2] P.P. Chakrabarti, S. Ghose, A. Acharaya, and S.C. De Sarkar, Heuristic Search in Restricted Memory, *Artificial Intelligence* 41 (1989) 197-221.
- [3] D. De Champeaux and L. Sint, An Improved Bidirectional Heuristic Search Algorithm, *Journal of the Association for Computing Machinery* 24 (1977) 177-191.
- [4] J.F. Dillenburg, Techniques for Improving the Efficiency of Heuristic Search, Ph.D. Dissertation, University of Illinois at Chicago, Chicago, Illinois, 1993.
- [5] J.F. Dillenburg and P.C. Nelson, Efficient Data Structures for Heuristic Search, under review.
- [6] P. Hart, R. Duda and B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, *IEEE Transactions on SSC* 4 (1968), 100-107.
- [7] R.E. Korf, Depth-First Iterative-Deepening: An Optimal Admissible Tree Search, *Artificial Intelligence* 27 (1985) 97-109.
- [8] J.B. Kwa, BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm, *Artificial Intelligence* 38 (1989) 95-109.
- [9] J. Mostow and A.E. Prieditis, Discovering Admissible Heuristics by Abstracting and Optimizing: A transformational Approach, *Proceedings of the 11th International Joint Conference of Artificial Intelligence (IJCAI-89)* 701-707.
- [10] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley, Reading, MA, 1984).
- [11] I. Pohl, Bi-directional Search, in: B. Meltzer and D. Michie, Eds., *Machine Intelligence* 6, (Edinburgh University Press, Edinburgh, 1971), 127-140.
- [12] G. Politowski and I. Pohl, D-Node Retargeting in Bidirectional Heuristic Search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-84)*, Austin, TX (1984) 274-277.

- [13] A. Sen and A. Bagchi, Fast Recursive Formulations for Best-First Search that Allow Controlled use of Memory, *Proceedings of the 11th International Joint Conference of Artificial Intelligence (IJCAI-89)* 297-302.
- [14] L. Taylor and R.E. Korf, Pruning Duplicate Nodes in Depth-First Search, *Proceedings of the National Conference on Artificial Intelligence (AAAI-93)*, Washington D.C. (1993) 756-761.