International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM)
ISSN: 2150-7988 Vol.2 (2010), pp.234-242
http://www.mirlabs.org/ijcisim

Spider Search: An Efficient and Non-Frontier-Based Real-Time Search Algorithm

Chao Lin Chu
Department of Computer Science
California State University, Los Angeles
5151 State University Drive
Los Angeles, CA 90032-8530
Email: usaminichu@gmail.com

Debora K. Shuger
Department of English
University of California, Los Angeles
149 Humanities Building, Box 951530
Los Angeles, CA 90095-1530
Email: shuger@humnet.ucla.edu

Russell J. Abbott
Department of Computer Science
California State University, Los Angeles
5151 State University Drive
Los Angeles, CA 90032-8530
Email: rabbott@calstatela.edu

Abstract

Real-time search algorithms are limited to constant-bounded search at each time step. We do not see much difference between standard search algorithms and good real-time search algorithms when problem sizes are small. However, having a good real-time search algorithm becomes important when problem sizes are large. In this paper we introduce a simple yet efficient algorithm, Spider Search, which uses very low constant time and space to solve problems when agents need deep (but not exhaustive) path analysis at each step. Moreover, we did some experimental tests to compare Spider search with other searches. We expect that Spider search is the first in a new class of tree-based rather than frontier-based search algorithms.

1 Introduction

In real-time search one is limited to a fixed amount of time at each step [1, 2, 3, 6], with the result that programmers usually reduce or limit the planning depths in order to meet the time limit requirement. Standard search algorithms cannot guarantee that they meet this requirement, although they can sometimes find optimal solutions [7]. These standard algorithms (including Breadth-first Search, Depth-first Search, and A* Search) have some use. However, they are not practical when the spaces grow very big.

A more advanced algorithm to solve real-time problems would be LRTA* [5], which limits the planning depth of each step and tries to find the local best at each step. This can be very useful for some problems but it is not practical for the problem we are trying to solve.

The problem we are trying to solve in this paper is the Boat/Torpedo problem. This problem concerns a boat that tries to avoid torpedoes and get home safely; the torpedoes, however, are very fast. How, we ask, can the boat be smart enough, how think far enough ahead, to avoid them? We defined the boat and torpedoes as being able only to move left and right at each step. Therefore, the tree is going to be of size $2^{1000} - 1$ if the boat reaches the home base at the 999^{th} step.

Standard static searches, like Breadth-first Search, Depth-first Search and A*, cannot solve this question, nor can LRTA* do so efficiently. The reason they cannot is that the boat agent has to plan ahead about 30 steps or more in order to avoid collision with torpedoes. If the planning path is too short, the boat agent cannot really tell which path is the best and sometimes it makes a wrong decision.

To solve this difficulty, we introduce a simple yet efficient algorithm, the Spider Search Algorithm [4], which can reach the depth of 30 or more, while still using a very low upper-bounded constant. The idea behind this approach is, first, to use controlled random probability numbers to evenly distribute paths in a tree space.(generation) This idea is important because if we do not do this, we will miss some parts of a tree and produce incorrect actions. We call

these paths the population. Then we calculate the fitness value of each state and pick the best leaf from the population in this step.(selection) This allows the boat agent to know the path to the best leaf node. The neighbor of the root on this path will be the next state. We can see that the upper-bounded limit constant will be ((depth + 1) * population), which is usually less than 1000 nodes per step. This will guarantee that this program runs in real-time.

2 Problem Formulation

First, the basic idea of this Boat/Torpedo problem is that the boat has to be smart enough and capable of thinking sufficiently far ahead to avoid the torpedoes. (This problem is originally from University of Illinois at Urbana-Champaign.)

The representation of the boat, torpedoes, and home in a state is(x,y), in which x is double precision from -1 to 1 and y is double precision from -1 to 1. The location of the home base is fixed, but the torpedoes and the boat are moving constantly during the whole program. The torpedoes can move faster than the boat, but the boat has a tighter turning radius than the torpedoes. For simplicity′s sake, the boat and torpedoes can only turn left and right. (Had we not imposed this limitation, we would need a bigger space than the current one, making the problem more complicated.) Therefore, we made the boat and torpedoes able only to turn full left and full right. The torpedoes turn as best they can towards a point somewhere in front of where the boat is currently located.

The boat and torpedoes know each other′s location, speed, and angle, but only the boat knows the torpedoes′ turning strategy. The boat is ′smart′ in that the boat agent is able to plan possible paths far enough into the future to predict the torpedoes′ various possible locations and to tell whether, if it takes that path, it will be hit by them. Then the boat agent can choose its next move in order to avoid destruction.

The boat′s length is 0.06 and the torpedo′s length is 0.04 in the space described above. As we can see, they are very small in this 2 x 2 square space. In addition, the boat and torpedoes can only move 0.007 and 0.01799 at each step, and the distance (0.0149317) that the program uses to decide if the boat is hit or gets home is also very small. So the ability to look 5 or 10 steps ahead is simply not enough for the boat to determine whether a path will allow it to escape from an attacking torpedo. Indeed, we found that we needed to make the boat able to look 30 steps ahead.

3 AI Approaches

Before we talk about AI approaches, we need to understand the program structure. The program is time-stepped. At each time tick, the boat expands the tree and selects the best node. It then turns left or right, depending on the first step in the path to the node selected as best. This may sound easy, but expanding the tree and picking the best node both presented major questions.

The two key issues are: (a) What kind of search strategy should the boat use when expanding the tree? and (b) Once the boat has finished expanding the tree, how should it pick the best node?

Real-time search problems need search algorithms that can perform constant-bounded search and yield reasonable results. Since real-time search is very sensitive to the size of the tree that the agent is going to explore, real-time search algorithms must limit path depths severely. This has the advantage of reducing computing time. This kind of algorithm may not be able to find the optimal solution, but it does allow the agent to work in real-time with some minor accuracy differences.

However, some real-time problems cannot be solved by simply reducing the path depths to be explored: for example, the Boat/Torpedoes problem. The shorter path we plan, the less accurate the result we get. The boat agent needs to plan ahead about 30 or more steps in order to avoid a collision. If the boat can only go left and right at each step, this means that the tree size is going to be $2^{30+1} - 1$ at each step. This will take a lot of computation, making some algorithms non-responsive in real-time.

The steps that the boat and torpedo take are so small that any frontier-based search will require an unacceptable number of steps to reach a point where the result will be useful. Therefore, we designed a search approach that is not frontier-based. We named it Spider Search because the paths created by Spider Search algorithm looked like a spider. This algorithm can achieve useful levels of path length and still use constant-bounded space and time.

3.1 Other Search Algorithms

Breadth-First Search uses level by level search from the root to higher levels. [7] This approach will guarantee that we get the optimal results since it searches for all possible paths in the space. However, this brute force algorithm is not efficient.

In addition, the Boat/Torpedo problem is more complex than the standard search problem. This is because the boat has two goals: to reach home and not to get hit by torpedoes. Not getting hit has higher priority when expanding the tree. This means that if the boat gets hit, the program will stop and the boat won′t get another chance to search. Among those nodes that would result in the boat not being hit, those closer to home have higher priority.

We know that most searches have a single goal. This problem could likewise be formulated as having a single

goal (to get home) were the amount of time available to the boat unlimited. The torpedoes would function like barriers to be avoided, rendering the problem more like a maze search. But given the real time constraint, the boat can′t look ahead indefinitely to see where all the barriers are, and if it makes a wrong decision, it cannot backtrack and take another path. So the goal of not being hit becomes more than just a constraint on the search path; it becomes the highest priority at each search step. That makes the problem very different from standard search problems.

A* Search is an offline search much like Breadth-first Search and Depth-first Search, and hence faces similar time and space size problem. However, A* Search is a bit better than the other two because it uses a heuristic function to tell if it is worthwhile to further expand the node. Hence, A* Search is pretty much like a Best First Search. [7] It sounds like we can limit the depth, for example depth 30, and run the A* search. However, it has a problem. The problem is that most of the leaf nodes are dead near the depth of 30. This means that A* is going to extend the best node until the node gets killed (at which time, it is near the full depth of 30). Then it starts to extend the second best node. Eventually, it is going to fill up this tree. As we can see, it can have a maximum of $2^{30+1} - 1$ nodes in the tree in each step. Although we can say it is constant-bounded, this constant is far too big for this problem.

LRTA* is a general and well-known real-time search algorithm that uses a fixed depth to limit the agent planning. This ensures that a program runs at a constant-bounded rate. It also uses a heuristic function to calculate the estimated distance from a target state to a goal state. It is pretty much like the A* search so it faces the same problem as the A* search in this problem.

3.2 Spider Search Algorithm

To solve the issues that LRTA* cannot, we introduce Spider Search Algorithm: a non-frontier based real-time search algorithm (Please see the search path on Figure 1 and the Spider Search Algorithm on Figure 2.). This algorithm works by generation and selection.

That is to say, it generates lots of diverse possibilities without biasing the generation process by referring to the goal. This means that during the generation step, we do not consider whether or not a path will move the boat toward home. Also, we do not consider if a path will keep the boat from being hit by a torpedo. However, for efficiency, we do cut off a path if a point on the path results in a hit, since there is no need to explore the points that the boat, having been destroyed, obviously cannot reach. This limitation is thus an efficiency measure rather than a means to direct the search.

Once these possibilities have been generated, we then select the best node from the resulting population based on an evaluation function. This fitness function takes into account distance from home, length of path, and whether the boat was hit. The important point is that selection is separated from generation.

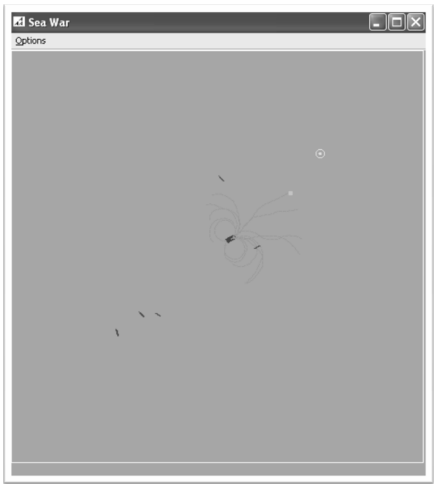

Figure 1. Spider SearchPath

This algorithm is therefore crucially different from the best-first search, which ties generation directly to the goal. Moreover, the separation makes it fit very well within a Real-Time framework. Since the generation step is separated from the selection step, it can be done without reference to the goal and in a fixed deterministic way. That doesn′t mean it has to be deterministic, but it does mean that the generation step is not search-driven. It is the search aspect of the other search approaches that make them non-real-time.

Since Spider Search does its generation outside the search paradigm, it can be real time. Moreover, the selection step can also be done in real-time because the number of elements the generation step generated is fixed, so that the selection needs only select from among them.

Let′s look at the Spider Search Algorithm in Figure 2. We pass the root node, the population size, and the depth limit to the createSpider algorithm. It then calculates several probabilities according to the population size. Then the createSpider algorithm passes the root node, these probabilities, and the depth limit to the createPath algorithm.

When the createPath gets those parameters, it performs the following steps:

(1) Find the current depth of this node in the tree.

(2) If the current depth is smaller than the depth limit and this node is neither dead nor reaching home, go to step 3. Otherwise, go to step 9.

(3) Use randomly generated numbers to decide left or right child to expend. If it decides to go right, go to step 4.

```
createSpider( root, popSize, depthLimit )
1 for each i from 0 to popSize - 1
2   turnRightProbability = i / (popSize - 1)
3   createPath( root, turnRightProbability, depthLimit)
4 end of for loop

createPath( root, turnRightProbability, depthLimit )
1    d = current depth of this node
2    if (d < depthLimit and not (dead or reached home) )
3      if( randomly generated real number from 0 to 1 <
turnRightProbability )
4         if( right child has not been created )
5            create rightChild
6         end of if
7           createPath( rightChild, turnRightProbability,
depthLimit)
8      end of if
9      else
10        if( left child has not been created )
11           create leftChild
12        end of if
13        createPath( leftChild, turnRightProbability, depth-
Limit)
14    end of else
15  end of if
```

Figure 2. Spider Search Algorithm

Otherwise, go to step 7.

(4) If the right child has not been created, create the right child.

(5) Pass the right child node, the turnRightProbability, and the depth limit to itself. (createPath algorithm)

(6) Go to step 9.

(7) If the left child has not been created, create the left child.

(8) Pass the left child node, the turnRightProbability, and the depth limit to itself. (createPath algorithm)

(9) End of this algorithm.

The ideas behind this algorithm are the following:

3.2.1 Population

Population is a parameter that the user or program can control, but it is usually fixed throughout the whole program. In special cases, we could change the population size dynamically according to deal with features specific to a particular problem. However, it should not be changed while the program is running. Also, in order to keep the bounded constant the same, we should change the depth of the paths to a lower value as a way to raise the population size.

3.2.2 Randomness and Controlled Probabilities

Since the boat can turn only left and right, we can use a binary tree to represent the paths. The left child represents the boat moving left from the root state and right child represents the boat moving right.

The problem is that the space is so big. How can we explore this tree evenly and efficiently? We use randomness to pick the paths we desire. However, if we use the full random to get left and right, the resulting paths are going to reside mainly in the middle of the tree, since one would expect that each path will include approximately half left turns and half right turns. (We tried Genetic Algorithm and found out this.) To solve this, if Spider Search has a depth of 5 and a population of 6, it will have about the following paths: LLLLL, RLLLL, RRLLL, RRRLL, RRRRL, RRRRR, which are evenly distributed. (See Figure2.) Although the algorithm we created is very simple, it actually did a very good job in spreading out the distribution of the paths in the tree. You will see how it performed later on in this paper.

3.2.3 Limited Depth

We can see the depth limit, which we have not yet described, in Figure 2. It is based on the same idea as LRTA*. Its function is to reduce the tree size at each step of agent planning. This will guarantee that it is constant-bounded. In our Boat/Torpedo program, we used a depth limit of 30 for every step. Yet, if we searched all the nodes in the tree to depth 30, we would have to search $2^{30+1} - 1$ nodes in each step, which is not acceptable for a real-time situation. However, as mentioned earlier, the program distributes the paths evenly in the tree space, which allows it to extend the depths to consider a longer range of possibilities without requiring the program to search so many nodes that it slows down the process. Through a simple calculation, we can see how fast the Spider Search is. We used a population size of 10 and depth limit of 30. The maximum nodes we expect is (30 + 1) * 10 = 310. However, it actually uses fewer than this many nodes because paths near the root of the tree share nodes. Therefore, it actually uses fewer than 310 nodes per step. For even better performance, we used a fitness function to further reduce the number of the nodes searched since we can cut off the search beyond a dead node. We will explain it in the next sub-section.

3.2.4 Fitness Function

In the Boat/Torpedo problem, we used a fitness function to decide, at any given step, which path is the best, so that the boat agent can know how to move to the next state. The fitness function picks the nodes with lower fitness values as better ones. This is because we used the distance to the home base as one of the factors determining fitness. The

```
calculateFitnessValue()
1  if( this state reaches home )
2    fitnessValue = current depth
3    return
4  end of if
5  fitnessValue = 10000000 + current state to home distance - current depth / 10
6  if( current state boat is killed )
7    fitnessValue = fitnessValue + 10000000
8  end of if
9  return
```

Figure 3. Fitness Function

lower value is thus better. This is like a heuristic function, but the distances to home are actual distances to home instead of estimated ones. We use the following rules to calculate the fitness value, and the actual algorithm is in Figure 3.

(1) If a path reaches home, the shorter the better. So the fitness function penalizes length.

(2) If we haven′t reached home, add a penalty of 10,000,000 plus the distance home. However, if we haven′t reached home, a longer path is better since it provides more time to search. So give a bonus for longer paths. The factor of 10 in effect divides the paths into 10 buckets. Within each one, the shorter the distance to home, the better the path.

(3) A dead path is worse than any live path. So add another penalty of 10,000,000 to the fitness value. Of course the boat agent stops expanding nodes if the current node′s condition is either home or dead. This further reduces the search time since it cuts off unnecessary tree nodes.

3.2.5 Keeping the Best Candidate for Next Step Planning

This is the action between each step. We keep the best path for the next step because when the boat gets to this next step that path may still be the best. If it turns out not to be the best, it will simply be replaced by another path that is better. This can also reduce the time spent searching some areas of the tree. Although you may think that throwing away all other paths and searching again might cause the boat to perform badly, it does not. The boat runs smoothly and uses a bounded constant time and space.

Let us look at the detail of this algorithm in Figure 2. We pass the root, popSize and depthLimit to the createSpider function. Then it creates a turnRightProbability for each path according to the popSize. After knowing the turnRightProbability, it passes the root, turnRightProbability and depthLimit to the createPath function.

This createPath function only creates one child (L or R) by comparing random 0 to 1 real number to the turnRightProbability. Then it will call itself recursively (passing the child, turnRightProbability, depthLimit to itself) until reaching the depthLimit, being dead or reaching home. Then it will create well distributed paths in this tree. (We call it the Spider.)

4 Experimental Testing

4.1 Define the Space

For the boat and torpedoes to move freely in this 2x2 rectangle space, we took out the border barrier producing a toroidal world. We also defined the following parameters:

(1)boat speed: 0.007

(2)torpedo speed: 0.01799

(3)Boat turning angle: 0.1

(4)Torpedo turning angle: 0.05

(5)TouchEpsilon: 0.0149317.

The boat and torpedo speeds represent distance moved per step. The boat and torpedo turning angles are radians. TouchEpsilon is the distance at which contact is assumed to be made. If the distance between the boat and a torpedo is less than that value, the boat is killed; if the distance between the boat and home is less than that value, the boat reaches home.

At the beginning of each run, we placed home and the boat at the furthest possible distance apart to make sure that we didn′t give the boat an undue advantage. Then we randomly placed the torpedoes–but in different quadrants from the boat to make sure that the torpedoes would not kill the boat instantly. This random scattering of the torpedoes makes the problem harder for the boat because they come at it from many angles at once.

4.2 Gathering Data

4.2.1 Experimental Testing: Part 1

Please see Figure 4. The lighter color one represents Spider Search and the darker one represents Breadth First Search. In this test, we used depth 10 to run Breadth First Search. The reason was that since it had a real-time constraint, depth 10 performed acceptable speed on our testing computer. More than that it would generate too many nodes in each time step and slow down the Breadth First Search.

To compare Spider and Breadth First Searches, we used the same depth 10 on Spider Search. Also, we used population size of 10 to run Spider Search. Then we fixed all the parameters except the torpedo quantity. We then compared

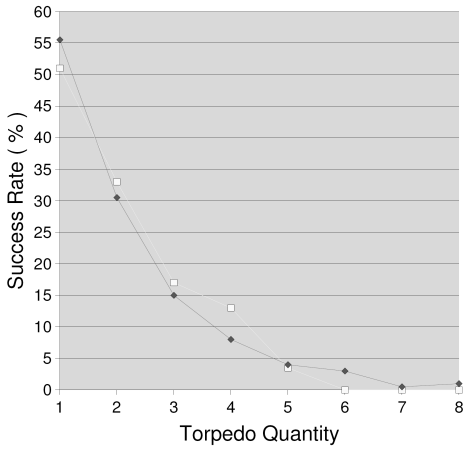

Figure 4. BreadthFirst and Spider Search Compare

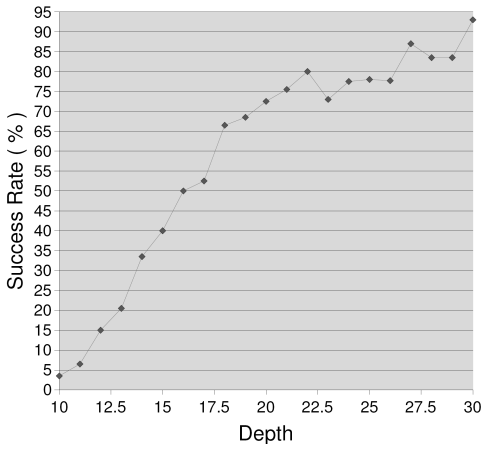

Figure 5. Depth and Success Rate of Spider Search using 5 Torpedoes and popSize 10

how BreadthFirst and Spider Searches performed. We ran the program using 1 to 8 torpedoes and running 200 trials for each torpedo quantity. Then we counted it as one success when the boat reached home. According to the 200 runs, we calculated the success rates.

As the number of torpedoes increased in Figure 4, the success rate started to drop gradually. This is because the more torpedoes, the more dead nodes in the boat planning trees. As long as there is no live node in the boat′s searching tree, the boat is going to be hit. There is no way for it to survive, not even if we had used an optimal but slow search algorithm like Breadth-first Search. This is the nature of this problem (and of reality).

In addition, if torpedoes come from different angles and attack the boat together, there is sometimes no way for the boat to survive. This is because each torpedo can cover some parts of the boat′s possible paths. When all of the boat′s possible paths are covered by torpedoes′ paths, the boat is going to be dead.

From Figure 4 curve, we can see that Spider Search performed the same as the optimal but slow Breadth First Search. However, Spider Search used a lot less nodes in each time step. This means a lot faster. The Breadth First Search used $2^{10+1}-1$ nodes (2047) in each time step, while Spider Search only used maximum (10 + 1) * 10 = 110 nodes.

4.2.2 Experimental Testing: Part 2

From Figure 4, we knew that Spider Search and Breadth First Search were very close on torpedo 5 so we did a second test (Figure 5) to see if we can improve the Spider Search by increasing the depth.

Table 1. Depth and Node Size

$Depth$	$PopSize$	$Spider$	$BreadthFirst$
10	10	110	$2^{10+1}-1$
11	10	120	$2^{11+1}-1$
12	10	130	$2^{12+1}-1$
|	|	|	|
28	10	290	$2^{28+1}-1$
29	10	300	$2^{29+1}-1$
30	10	310	$2^{30+1}-1$

In this test, we fixed the torpedo quantity to 5 and population size to 10 on Spider Search. The only variable was the depth. We tested it from 10 to 30 in 1 unit increment and ran 200 trials for each depth. Then we counted it as one success when the boat reached home. According to the 200 runs, we calculated the success rates.

From the result in Figure 5, we can see that the success rates of Spider Search amazingly grew from 3.5 to 93 when increasing the depth. This means that longer path planning can produce better result. However, the best of the Spider Search is efficiency. In Table 1, it shows the maximum node sizes of Spider Search in each time step and the expected node sizes of Breadth First Search. (Of course, we didnot have the computing power to run $2^{30+1}-1$ nodes per step on Breadth First Search.) At depth of 30, Spider Search only used maximum 310 nodes, which was a lot less than Breadth First Search at depth of 10 (2047 nodes). Despite of using less nodes than BFS per step, Spider Search's success rates grew from 3.5 to 93. This means that it fits very well within the real-time constraint.

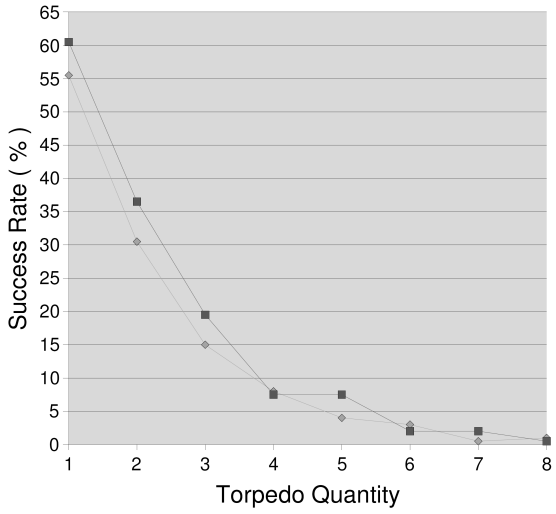

Figure 6. BestFirst and BreadthFirst Search Comparison

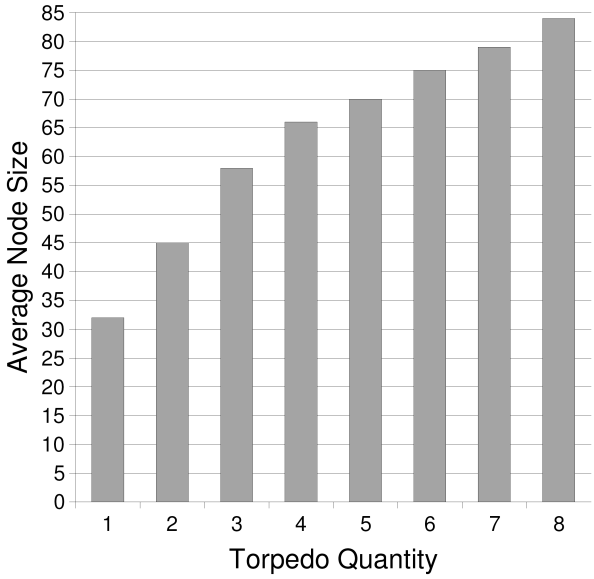

Figure 7. BestFirst Search Average Node Size

4.2.3 Experimental Testing: Part 3

In order to see if the Best First Search has the same maximum node size as the Breadth First Search, we did a third test.

To compare Best First and Breadth First Searches, we used the same depth 10 on Best First Search. Then we fixed all the parameters except the torpedo quantity. We then compared how Breadth First and Best First Searches performed. We ran the program using 1 to 8 torpedoes and running 200 trials for each torpedo quantity. Then we counted it as one success when the boat reached home. According to the 200 runs, we calculated the success rates.

Please see Figure 6. The darker color one represents Best First Search and the lighter one represents Breadth First Search. In this figure, we used the Breadth First Search results from experimental test one and the Best First Search results from this experimental test three. From Figure 6 curve, we can see that Best First Search performed almost the same as the slow Breadth First Search.

Then we focused on the result from the Best First Search. In this test, we also collected the node size of Best First Search in each step of each run. We then calculated the average node size of each torpedo quantity. In the Figure 7, we can see that the average node size grew with the growing torpedo quantity. This means that Best First Search needs to process more nodes averagely while the boat faces more dangers. The answer is easy because the more dead nodes in the search tree, the higher probability the Best First Search needs to find another best node.

People may notice that the average node sizes of Best First Search were smaller than the maximum node size of Spider Search in the same depth 10. Does this mean that the Best First Search performs better than the Spider Search? The answer is no. We will see why in the next.

In the Figure 8, it shows that the Best First Search node size of each step in one run. (One run means from start point to either home or dead.) It used very few nodes from 1 to 34 steps because it only processed the best nodes without being aware that it was in danger. Once it found out that it was in danger, the node size jumped to near full tree size, like step 35 in Figure 8. In this case, it was 2047. ($2^{10+1} - 1$) Then the node size started to drop and finally died at step 44. We can see that the bottleneck of the Best First Search is the maximum node size. In this experimental test 3, we only used depth 10 on the Best First Search so it only showed a little bit slow down when the boat was in danger. We can imagine that it will stop response when we use depth 30 and the boat is in danger. The Best First Search will search for $2^{30+1} - 1$ nodes in just one single step.

To be more clear, we recorded the maximum node size of the Best First Search in each torpedo quantity. In Table 2, it showed the maximum node size comparision of the Best First and Breadth First Searches. The maximum node sizes of the Best First Search were the same as the node sizes of the Breadth First Search no matter which torpedo quantity. This proves that the Best First Search faces the same node size problem that the Breadth First Search has.

From our visual observation in the experimental test 3, the Best First Search did slow down when the boat faced

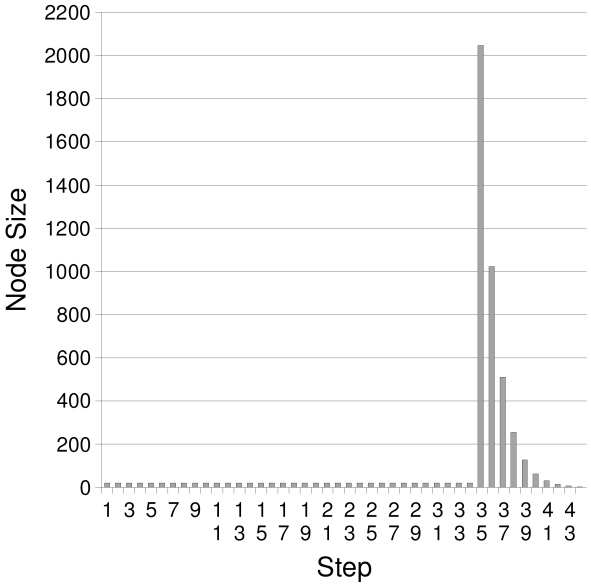

Figure 8. BestFirst Search Node Size in One Run

Table 2. Maximum Node Sizes of BreadthFirst and BestFirst Searches

$Torpedo$	$Depth$	$BestFirst$	$BreadthFirst$
1	10	$2^{10+1}-1$	$2^{10+1}-1$
2	10	$2^{10+1}-1$	$2^{10+1}-1$
3	10	$2^{10+1}-1$	$2^{10+1}-1$
4	10	$2^{10+1}-1$	$2^{10+1}-1$
5	10	$2^{10+1}-1$	$2^{10+1}-1$
6	10	$2^{10+1}-1$	$2^{10+1}-1$
7	10	$2^{10+1}-1$	$2^{10+1}-1$
8	10	$2^{10+1}-1$	$2^{10+1}-1$

dangers. Not to mention that it is going to freeze and try to find a way out from those $2^{30+1}-1$ nodes if we limit the depth to 30. This means that it can′t be real time because the maximum node size is way too big to fit in the real time constraint.

From experimental test 1 to 3, we understand that the Breadth First Search and the Best First Search are not useful in this problem while Spider Search uses very low constant space and time to solve it. Therefore, the Spider Search fits very well within the real-time constraint.

5 Future Work and Conclusion

Real-time Search Algorithms are important research areas. They are constant-bounded and good for fast response. There are many of them, for example, LRTA*, which are good for general usage. Nevertheless, they are less useful when agent path planning needs considerable depth and width. In our experimental test 1, we can see that the Breadth First Search is slow because it searched for all the nodes in a tree. Also in experimental test 3, the Best First Search can′t fit in the real time constraint because its maximum node size bottleneck. However, as we can see from the experimental test 2, Spider Search can solve this type of problem efficiently while preserving a very low constant. Yet we do not know what other kinds of problems benefit from it. Also, maybe we could plug Spider Search into other search algorithms to make them more effective in the future.

References

[1] V. Bulitko and G. Lee. Learning in real-time search: A unifying framework. *Journal of Artificial Intelligence Research*, 25:119–157, 2006.

[2] V. Bulitko, N. Sturtevant, and M. Kazakevlch. Speeding up learning in real-time search via automatic state abstraction. *AAAI*, 214:1349–1354, 2005.

[3] V. Bulitko, N. Sturtevant, J. Lu, and T. Yau. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research*, 30:51–100, 2007.

[4] C. L. Chu, R. J. Abbott, and D. K. Shuger. Spider search: An efficient and non-frontier-based real-time search algorithm. *2008 Eighth International Conference on Intelligent Systems Design and Applications*, 2:487–492, 2008.

[5] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, 1990.

[6] D. C. Rayner, K. Davison, V. Bulitko, K. Anderson, and J. Lu. Real-time heuristic search with a priority queue. *IJCAI*, 382:2372–2377, 2007.

[7] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach*. Pearson Education, Inc., Upper Saddle River, New Jersey, 2003.

Biographical notes: Chao Lin Chu is a Teaching Assistant in Computer Science at the Tunghai University, Taiwan. He received his M.S. degree in Computer Science from California State University, Los Angeles, U.S.A. and his B.S. degree in Animal Science from National Chung Hsing University, Taiwan. His research interests include Artificial Intelligence, Real-time Searches, and Genetic Algorithm.

Debora Shuger is a Professor of English at UCLA. She received her BA and MA in English from Vanderbilt University and her PhD in English from Stanford University. Her research interests are early modern/Renaissance/late 16th and 17th century England. She writes about Tudor-Stuart literature; religious, political, and legal thought; neo-Latin; and censorship of that period.

Russ Abbott is a Professor of Computer Science at California State University, Los Angeles. He received his BA in Mathematics from Columbia University, MS in Computer Science from Harvard University, and PhD in Computer Science from the University of Southern California. His research interests are complex systems and the application of concept in computer science to problems in philosophy.