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

Name1 and **Name2**
Department of Computer Science
University of School1

Name3
Department of Computer Science
University of School2

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. There are some well known real-time search algorithms, for instance, Learning Real-time A* (LRTA*). But LRTA* is inadequate for problems that require deep path analysis at each time step. In general, any frontier-based search will be inadequate when deep path analysis is required. 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. We expect that Spider search is the first in a new class of tree-based rather than frontier-based search algorithms.

Introduction

In real-time search one is limited to a fixed amount of time at each step, 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. 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*, which limits the planning depth of each step and tries to find the local best at each step. This can be very useful for problems that dont require a big exploring depth at each step. This means that problems with huge spaces and requiring deep paths at each time step of planning can cause the LRTA* to become non-responsive as a real-time search algorithm.

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 represent the boat, torpedoes, and home base as (x, y), in which x and y are each a double precision value from -1 to 1. Each state will contain the locations of the boat, torpedoes, and home. Home has a fixed location, but the boat and torpedoes are constantly moving. 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} if the boat reaches the home base at the 1000^{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. Therefore, if we limit 30 depth and run LRTA*, we find that it is still constant-bounded but with a very big constant(2^{30}).

To solve this difficulty, we introduce a simple yet efficient algorithm, the Spider Search Algorithm, 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. This idea is important because if we dont́ 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. 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 * population), which is usually less than 1000 nodes per step. This will guarantee that this program runs in real-time.

Problem Formulation

First, the basic idea of this Boat/Torpedo problem is that the boat tries to avoid torpedoes and get home safely. However, the torpedoes are faster than the boat, so the boat has to be smart enough and capable of thinking sufficiently far ahead to avoid them. (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 simplicityś 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 only able 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 otherś location, speed, and angle, but only the boat knows the torpedoes´turning strategy. The boat is śmart´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 boats length is 0.06 and the torpedoes 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.

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, 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} 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.

The standard searches discussed below and LRTA* are both frontier-based searches. We will now explain why they are not useful for this problem.

Standard Search Algorithms

Breadth-First Search Breadth-First Search uses level by level search from the root to higher levels. 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 the Boat/Torpedo problem, the boat usually needs several hundred to several thousand steps to reach the goal. Breadth-First Search will need unacceptable amounts of computation, 2^{500} or more node expansions, to finish. Therefore, this approach is not applicable to this type of problem.

Depth-First Search Although Depth-First Search can reach the depth sufficient for the boat to tell if a path is good or bad, it faces the same space size problem that Breadth-First Search does. It can reach the desired depth, but cannot explore the space widely enough within some node limits. This is dangerous because we dont́ want to explore just one part of the tree space. Hence, this is not an ideal solution for Boat/Torpedo problem.

A* Search 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. This heuristic function can prevent unnecessary exploration of the tree, and hence results in improved performance. However, A* search picks only the best node to expand. This may leave a lot of the search space unexplored. Although it is complete and optimal, it is still not suitable for this Boat/Torpedo problem.

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 wont́ 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 cant́ look ahead indefinitely to see where all the barriers are, and if it makes a wrong decision, it cant́ 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.

Real-Time Search Algorithm

LRTA* 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. Since we know the distance from a current state to a target state, we can know the estimated distance from a current state to the goal state via this target state. Those two features are good ways to reduce the amount of time and space the agent must use at each step. Nevertheless, it doesńt work in situations where an agent needs a large amount of path depth planning to move to the next state.

With respect to the Boat/Torpedo problem, LRTA* presents two problems: unexplored area of space and tree size.

In order for the boat to escape, it usually needs about a 30 or more step look-ahead. If LRTA* uses a depth of 30 to search and it finds the best node at depth of 30 without getting hit, it stops the current step and return an action. Although it saves computation time this is risky, since it leaves most of the nodes in the space unexplored.

The other problem with LRTA* is that most of the leaf nodes are dead near the depth of 30. This means that LRTA* 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} nodes in the tree. Although we can say it is constant-bounded, this constant is far too big for this problem.

Spider Search Algorithm

To solve the issues that LRTA* cannot, we introduce Spider Search Algorithm: a non-frontier based real-time search algorithm (Figure 1). 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 doń consider whether or not a path will move the boat toward home. Also, we doń 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.

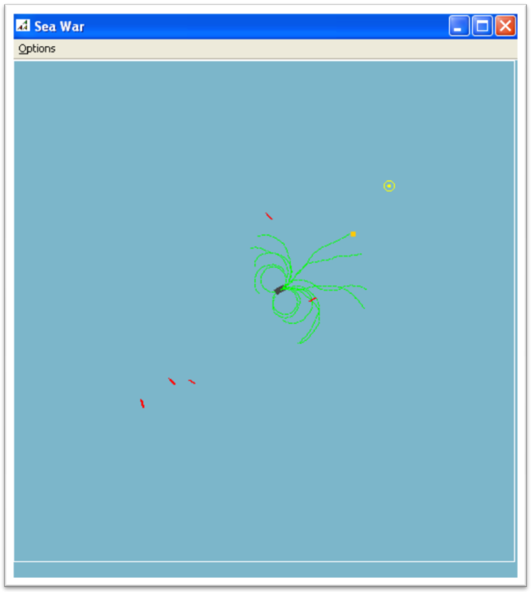

Figure 1: Spider SearchPath

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 doesńt 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.

The ideas behind this algorithm are the following:

- Population and fitness ideas from Genetic Algorithm.
- Use probabilities to control randomness of path distribution.
- Limited Depth idea from LRTA*, but depth is a lot bigger than LRTA*ś depth.
- Keep the best candidate for next step planning.

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 shouldńt 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.

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 mov-

```
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)
3       if( randomly generated real number from 0 to 1 < turnRightProbability )
4          if( right child has not 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 created )
11            create leftChild
12         end of if
13         createPath( leftChild, turnRightProbability, depthLimit)
14    end of else
15    end of if
```

Figure 2: Spider Search Algorithm

ing 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 that each path will include approximately half left turns and half right turns. 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, ..., RRRRR, which are evenly distributed. (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.

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 75 for every step. Yet, if we searched all the nodes in the tree to depth 75, we would have to search 2^{75} 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

```
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

how fast the Spider Search is. We used a population size of 14 and depth limit of 75. The maximum nodes we expect is 75*14 = 1050. 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 1050 nodes per step. For even better performance, we used a fitness function to further reduce the number of the nodes searched. We will explain it in the next sub-section.

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 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:

- If a path reaches home, the shorter the better. So the fitness function penalizes length.
- If we havent́ reached home, add a penalty of 10,000,000 plus the distance home. However, if we havent́ 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.
- 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 nodes condition is either home or dead. This further reduces the search time since it cuts off unnecessary tree nodes.

Keeping the Best Candidate for Next Step Planning This is the action between each steps. 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 per-

Table 1: Torpedo Quantity and Success Rate

$Torpedo\ Quantity$	$Success\%$
1	100
2	100
3	100
4	99
5	97
6	89
7	67
8	40
9	27
10	15
11	9
12	3
13	1
14	1
15	0
16	0

Table 2: Depth and Success Rate of 8 Torpedoes

$Depth$	$PopSize$	$Success$	$Spider$	$LRTA*$
10	14	0.5	140	$2^{10+1}-1$
15	14	14.0	210	$2^{15+1}-1$
20	14	37.0	280	$2^{20+1}-1$
25	14	48.5	350	$2^{25+1}-1$
30	14	59.5	420	$2^{30+1}-1$
35	14	57.5	490	$2^{35+1}-1$
40	14	57.0	560	$2^{40+1}-1$
45	14	56.0	630	$2^{45+1}-1$
50	14	52.5	700	$2^{50+1}-1$
55	14	55.5	770	$2^{55+1}-1$
60	14	44.5	840	$2^{60+1}-1$
65	14	45.0	910	$2^{65+1}-1$
70	14	49.5	980	$2^{70+1}-1$
75	14	41.0	1050	$2^{75+1}-1$

form badly, it does not. The boat runs smoothly and uses a bounded constant time and space. We will see the results in the next section.

Experimental Testing

Define the Space

Since the boat, the torpedoes, and home are represented by an ordered pair (x, y), where x and y are in the range -1 to 1, we can see that the boat and torpedoes move inside a 2x2 rectangle. For the boat and torpedoes to move freely in this space, we took out the border barrier producing a toroidal world. We also defined the following parameters.

- Boat speed: 0.007
- Torpedo speed: 0.01799
- Boat turning angle: 0.1
- Torpedo turning angle: 0.05
- 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.

Gathering Data

Experimental Testing: Part 1 We fixed all the parameters except the torpedo quantity. Lookahead depth was 75 and population size was 14. We ran the program using from 1 to 16 torpedoes, running 100 trials for each torpedo quantity.

Experimental Testing: Part 2 Then we fixed the torpedo quantity at 8. We ran depths from 10 to 75 in 5 unit increments, and each 200 times. Throughout these tests, the population sizes were fixed at 14.

Result

Part 1 The first column is the Torpedo Quantity and the second column is the Success Rate, i.e., how many times the boat reached home in 100 runs. (Table 1)

Part 2 The first column gives the depths at each step. The second column shows the population sizes corresponding to the depths in the same rows. The success rate measures the percent of times the boat reached home in 200 runs. The last two columns give the expected maximum node sizes for Spider and LRTA* Search algorithms. Please see table 2.

Analysis

Part 1 From the results of part 1, we can see when there were between 1 and 3 torpedoes, the boat was smart enough to achieve a 100% success rate. As the number of torpedoes increased, the success rate started to drop gradually. If there were more than 14 torpedoes, the boat didn ́t have a chance. This is because the more torpedoes, the more dead nodes in the boat planning trees. As long as there are no live nodes 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 boatś possible paths. When all of the boatś possible paths are covered by torpedoes´paths, the boat is going to be dead.

Despite the nature of this problem, we can still see that the Spider Search performed very well. Although we can increase the population size and depth and as a result shift the curve to the right, it will still go down to 0 eventually. As mentioned before, this Spider algorithm does not seek to produce optimal solutions. Instead, it tries to strike a balance between the best path and best performance, as it must, if it is to be a real time search algorithm.

Part 2 From the results of part 2, we observed the following interesting points.

- The success Rates increased when the depth increased from 10 to 30. This means that we need to have at least a 30 step lookahead to achieve a good result.
- The success rate started to drop after depth of 30. This is because the bigger depth will create a bigger tree, but we fixed the population to 14. So the population will become smaller compared to the growing tree. This means that we can improve the success rate in depth of 75 by using population of 75 + 1. However, this will require more time and space. In this problem, we found that a depth of 75 and population of 14 were good enough to see how Spider Search performed.
- For the maximum node sizes, LRTA* will grow exponentially with the depth. For Spider Search to have the greatest accuracy, its node size will be O(d * (d + 1)) where d is the depth. (A population of more than d + 1 may cause some leaf nodes to end up in similar locations.) If we want faster speed more than accuracy, we can fix the population size (as in Table 2) and still get reasonable results. Its node size will be O(cd) where c is a constant of population size. The most important thing is that the depth is not changed in a single run so the required time and space is a small constant throughout each run, for instance, 75 * 14 = 1050.

Future Work

Enhancement of the Boat

In the current version, the boat has a fixed speed. If we can make the boat smart enough to accelerate or decelerate to avoid torpedoes, that will be an interesting approach. On the other hand, giving the boat more flexibility in how it moves creates more branches per node. Given a constant bound on the number of nodes one can search, it would be interesting to see whether this really improves the boatś performance.

Enhancement of the Torpedoes

Torpedoes are not smart in this current version. If we allowed them to work as a team, there is no way that the boat could survive. How could we make the torpedoes communicate with each other? That will be another challenging issue to solve.

Look for Other Problems can be Solved by This Algorithm

This algorithm is not trying to find the optimal solution but to find the reasonably best solution given time constraints. This is critical for real time search algorithms. If there are other problems that fall within this category and have not yet been solved efficiently, Spider Search may be able to help.

Adapt Spider Search into Other Search Algorithm

Since Spider Search is good at evenly distributing the search paths, perhaps we can combine Spider Search with other search algorithms and make them more effective. This will also be a good future research direction.

Conclusion

Real-time Search Algorithms are an important research area. 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. As you can see from this paper Spider Search can solve this type of problem efficiently while preserving a very low constant. Yet we donť know what other kinds of problems benefit from it. Maybe we could plug Spider Search into other search algorithms to make them more effective. That would be a good research area in the future.

References

Bulitko, V., and Lee, G. 2006. Learning in Real-Time Search: A Unifying Framework. *Journal of Artificial Intelligence Research* 25: 119–157.

Bulitko, V., Sturtevant, N., and Kazakevlch, M. 2005. Speeding Up Learning in Real-time Search via Automatic State Abstraction. *AAAI* 214: 1349–1354.

Bulitko, V., Sturtevant, N., Lu, J., and Yau, T. 2007. Graph Abstraction in Real-time Heuristic Search. *Journal of Artificial Intelligence Research* 30: 51–100.

Korf, R. E. 1990. Real-time Heuristic Search. *Artificial Intelligence* 42(2-3):189–211.

Rayner, D. C., Davison, K., Bulitko, V., Anderson, K., and Lu, J. 2007. Real-Time Heuristic Search with a Priority Queue. *IJCAI* 382: 2372–2377.

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