6601 – Assignment 1: Search

Instructor: Thad Starner TA: Daniel Kohlsdorf, Titilayo Craig

Weiren Wang weirenwang@gatech.edu

https://www.linkedin.com/in/weirenwang

https://github.com/JeffreyWeirenWang

Warm Up

Assignment 3.18: Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, O(n2) vs O(n))

Suppose that there is a tree to search, the tree has only one branch. Depth-first search could reach goal after searching the single branch search in the tree. The complexity is O(n). The iterative searching needs to search 1+2+3+…+n which is, $O(n^2)$. The other situation is a tree has b branches. The answer is in leftist branch. DFS take O(n) to find. IDS take $O(b^{d-1})$.

Assignment 3.21: Prove each of the following statements, or give a counterexample:

a. **Breadth first search is a special case of uniform-cost search.**
   True. When all step costs are the same, uniform-cost search is similar to breadth-first search, except that the latter stops as soon as it generates a goal, whereas uniform-cost search examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

b. **Depth first search is a special case of best-first tree search.**

True. When the evaluation function in best-first search is proportional to the depth of the state which searching.

c. **Uniform-cost search is a special case of A* search.**

True. A* is estimated by function f(n) = g(n) + h(n). While h(n) = 0 for all states, the A* algorithm is the same as the uniform algorithm.

Practice

Queues: Figure 3.14 in your textbook shows the uniform cost search algorithm. Argue in terms of computational complexity why a priority queue such as a fibonacci heap is superior to searching the max in an unordered queue or to sorting. HINT: you have to use both INSERT and POP for your argument. You do not need to prove your result but construct a clear argument.

Fibonacci heap actual insert complexity is O(1), pop min/max complexity is O(n), decrease key(or increase) for updating the queue is O(n). In terms of amortized cost, the insert complexity is O(1), pop min/max complexity is O(log n), and decrease key is O(1).

The properties for searching is inserting several nodes and popping out max/min nodes every time. In the initial state, the queue is empty and then insert one source node. During the searching or expanding the graph, inserting the nodes and pop out the max/min from the queue. The overall complexity is O(nlog n + e) for Fibonacci heap is better than the overall complexity of min Min heap which is O(nlog n + elog n).

BDD: If you search large spaces, keeping track of nodes you visited (explored queue in the algorithm) already might be expensive in terms of memory. How could you use binary decision diagrams (BDDs) to keep track of visited nodes with the goal to save memory? How do you you have to represent states?.

The BDD is generally achieving by merge isomorphic subgraphs and eliminate

nodes whose two children are isomorphic. For every node, ranked their neighbor by cost. Then the entry to each neighbor could be represented as a decision tree. Building a truth table, the visited node value is 1, unvisited node value is 0. Suppose one node could be reached from start, then we need to record the path from start. Instead of tracking whole states, we just need to track which way we need to go to reach the visited node 1. In this way, memory could be saved because of

Implementation

```
1.  procedure UniformCostSearch(Graph, root, goal)
2.          node := root, cost = 0
3.        frontier := priority queue containing node only
4.        explored := empty set
5.        do
6.                  if frontier is empty
7.                          return failure
8.                  node := frontier.pop()
9.                  if node is goal
10.                         return solution
11.                 explored.add(node)
12 .                for each of node's neighbors n
13.                         if n is not in explored
14.                             if n is not in frontier
15.                                 frontier.add(n)
16.                 else if n is in frontier with higher cost
17.       replace existing node with n
```

The uniform-cost search is complete and optimal if the cost of each step exceeds some positive bound ε. The worst-case time and space complexity is O(b1 + C*/ε), where C* is the cost of the optimal solution and b is the branching factor. When all step costs are equal, this becomes O (bd + 1).

In order to decide the paths to three cities A, B and C. The initial way is to run the uniform cost search three times. In each time, set the [start, end] tuple as [A, B] [B, C] and [C, A]. Because all of the ways are two-way pass, we don't need to consider direction in this case. A smarter way could let us search in only two times. Suppose we start from A. Then the first city we touched while searching is B. Then let A continue searching until A touched C. Then let B to be the start and C to be the end to search. In the first search round, we get distance between [A,B], [A,C]. In the second search round, we get distance between [B,C]. For a three cities problem, two rounds search is enough to find the solution.

Bidirectional Search

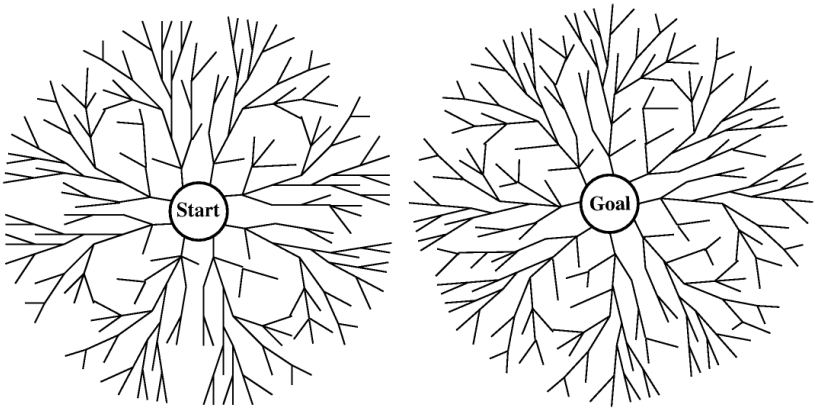
```
BIDIRECTIONAL_SEARCH
 1   Q_I.Insert(x_I) and mark x_I as visited
 2   Q_G.Insert(x_G) and mark x_G as visited
 3   while Q_I not empty and Q_G not empty do
 4       if Q_I not empty
 5           x ← Q_I.GetFirst()
 6           if x = x_G or x ∈ Q_G
 7               return SUCCESS
 8           forall u ∈ U(x)
 9               x' ← f(x, u)
10               if x' not visited
11                   Mark x' as visited
12                   Q_I.Insert(x')
13               else
14                   Resolve duplicate x'
15       if Q_G not empty
16           x' ← Q_G.GetFirst()
17           if x' = x_I or x' ∈ Q_I
18               return SUCCESS
19           forall u^-1 ∈ U^-1(x')
20               x ← f^-1(x', u^-1)
21               if x not visited
22                   Mark x as visited
23                   Q_G.Insert(x)
24               else
25                   Resolve duplicate x
26   return FAILURE
```

The bidirectional search is complete and optimal.

The core idea of bidirectional search is let the start and end to search simultaneously. We keep track of the frontier of each node. When two frontier has intersections, the point could be reached both by start and end.

Thus start and end could reach each other.

In the three city problems, we define a start end tuple structure, [start, end]. Then based on this, we use [A, B], [B, C], [C , A] to be the start point and end point to test.



Tridirectional Search

The Tridirectional search is complete and optimal.

The core idea of Tridirecional search is very similar to bidirectional search. Instead of start from two points, we start from three points this time. Keep tracking tree frontiers, once they find the interaction with each other. Then stop. Because both of the three start place can reach to the intersection place. And we are using breadth first search, the solution is complete and optimal. For the three cities problem, the Tridirectional Search is the best fit.

Evaluation

From the data, we can find that the tridirectional search is the best search compared to bidirectional search and uniformdirectional search.

Tridirectional search is better than bidirectional search in this case because the three places searched outside simultaneously. Once their frontier intersect, the place is the optimal choice.

For bidirectional search, we need to run the bidirectional search 3 times so that we could find paths to connect three cities. These three paths connect each two cities. we then select two paths to be the optimal choice.

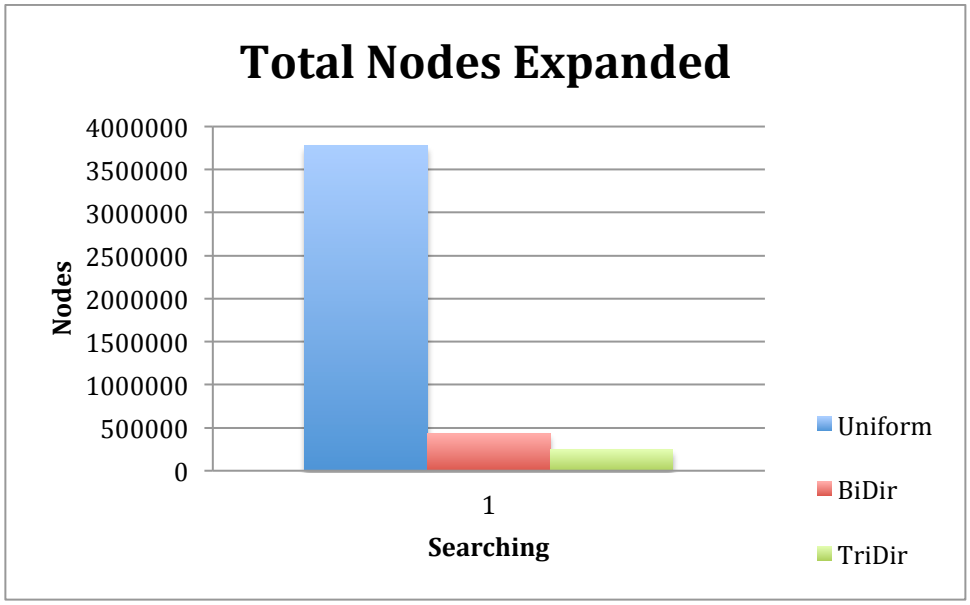
For Uniform cost search, it is the worst in this case both in time and space complexity. We need to run this algorithm at three times, at least two times to find the optimal path. At the same time, Uniform cost search is really time consuming because of its time complexity.

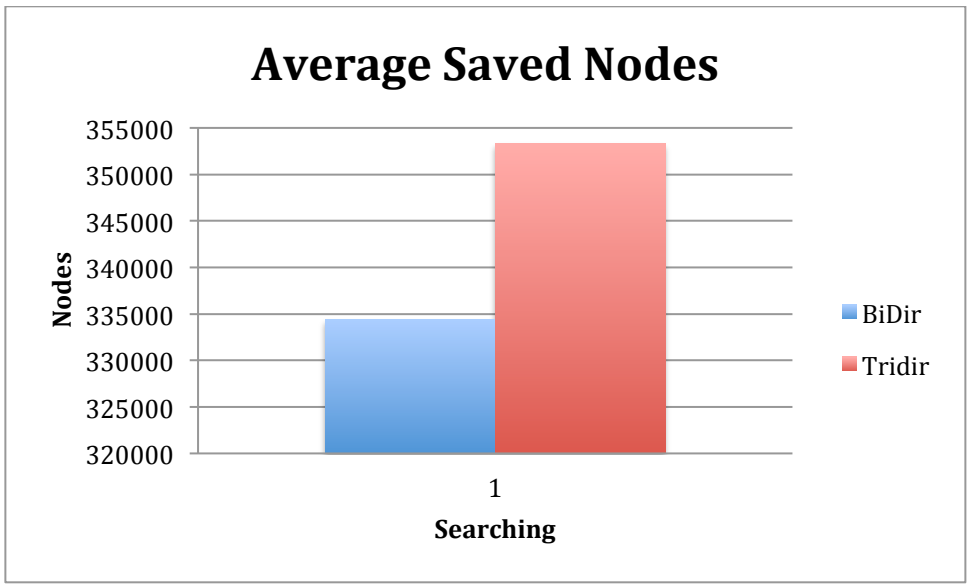
Analysis:

Criteria	Breadth-First	Uniform Cost	Bidirectional	Tridirectional
Complete	YES	YES	YES	YES
Optimal	YES	YES	YES	YES
Time	$O(b^d)$	$O(b^{1+c*/e})$	$O(b^{d/2})$	$O(b^{d/3})$
Space	$O(b^d)$	$O(b^{1+c*/e})$	$O(b^{d/2})$	$O(b^{d/3})$

b is the braching factor; d is depth of the shallowest situation; l is the depth limit.

Data Chart:


Total Nodes Expanded
4000000
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
Nodes
1
Searching
Uniform
BiDir
TriDir


Average Saved Nodes
355000
350000
345000
340000
335000
330000
325000
320000
Nodes
1
Searching
BiDir
Tridir