Homework 2

Li Miao (miao0044)

Writing:

1. In f(n) = g(n) + h(n), h(n) is the total estimated cost. g(n) is the cost so far to reach node N from root. And h(n) is the estimated cost to reach goal from N.

   If we add a weight to the equation, as f(n) = (1-w) * g(n) + w * h(n), thing might become different. There are 5 different situations:

   a. w = 0.5, where f(n) = 0.5g(n) + 0.5h(n)
   In this case, the algorithm still functions as normal A* search. It loops through massive amount of nodes, and found the optimal solution at the end.
   b. w = 0, where f(n) = g(n)
   In this case, the weight ratio didn't actually change. The search algorithm only looks the cost of reaching each node, and completely forget about the estimated cost. It becomes Uniform Cost search.
   c. 0 < w < 0.5, where g(n) weights more
   In this case, the search algorithm becomes something between Uniform Cost search and A* search.
   d. w = 1, where f(n) = h(n)
   In this case, the algorithm only looks at the estimated cost, not the cost so far. The algorithm becomes Greedy Best First search.
   e. 0.5 < w < 1, where h(n) weights more
   In this case, the algorithm becomes something between Greedy Best First search and A* search. It might find a solution with cost between C* (the optimal solution) and wC*. Usually, the solution is slightly more expensive than C*. But the algorithm would be way faster since it loops through way less nodes.

2.

   a. No, it is not guaranteed to find the optimal solution. For example, node A, node B, and node C are children of root. Let's say "root-A-C-goal" is the optimal path; node B and node C are not on the path of optimal solution. If N = 2, and the admissible heuristic underestimates h(n) of B and C by a lot, to a point h(n) of b and h(n) of C are both smaller than h(n) of A; Then node B and node C would push off node A. Since node A is the next node towards the optimal solution, and it was deleted already, the search algorithm would never get the optimal solution. The constrain of this edition is, the nodes of the optimal path cannot be pushed off from the "N best nodes" list. If they got pushed off, something like the above example might occur, and make it impossible to get the optimal solution.

b. Yes, it would always find the optimal solution. Since the heuristic is perfect, h(n) (the estimate cost to goal) is always equals to h*(n) (the actual cost to goal). The nodes on the optimal solution path always have the lowest f(n). And since h*(n) = h(n), the nodes on the optimal solution path would always be in the list of "N best nodes", which would never be deleted.

3.

a. Yes, it is possible for Uniform Cost search to expand more nodes than BFS. Uniform Cost search does not care about the number of steps involved in searching, and only concerns about the cost of paths. Due to these properties, sometimes Uniform Cost search could stuck in an infinite loop.
b. Yes, Uniform Cost search expands more nodes than A* search. Uniform Cost is an optimal search algorithm since it guarantees to find an optimal solution. A* search is known as "optimally efficient", which means no optimal search algorithm (include Uniform Cost search) is more efficient than A* search.

4.

No. A* search is "optimally efficient" means that: among all optimal search algorithms, A* search is the most efficient one.

But search algorithms like Depth First Search could get lucky:

For example, if DFS keeps expanding to nodes which are on the direct track of one of the optimal solutions. Then, it would find one of the optimal solutions extremely fast, it might expand less nodes than A* search.

However, this is not guaranteed. DFS could get unlucky too, in which case, it would expand towards a wrong way, and loop through huge amount of nodes before reaching an optimal solution; or it could final a non-optimal solution; or stuck into an infinite loop.

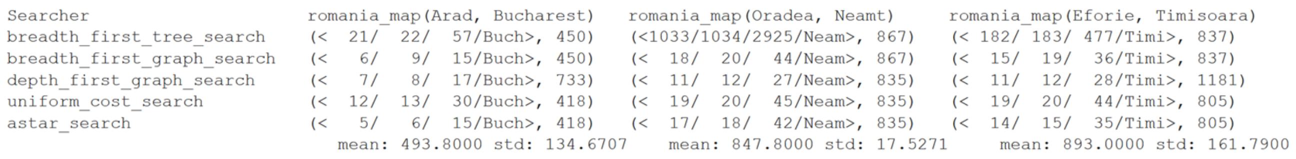
So, it is **possible** for DFS to be faster than A* to fins an optimal solution, but it's **not guaranteed**.

5.

a. It has few advantages. First, it is memory efficient. Just like DFS, it only needs to store nodes from root to the current node. Also, if all nodes on the solution is in front of most of their sibling nodes, Iterative Broadening search would find the solution very fast.
   The biggest shortcoming is similar as the shortcoming of Iteration Deepening search. If any node on the solution path has lots of sibling nodes in front of it, the search algorithm could repeat itself for many times. For example, if A has children of B, C, D, E, F, G. Let G be the last children of A, and the only node on the solution path. Then the algorithm would apply Depth First search on BC, BCD, BCDE, BCDEF, and finally BCDEFG. Which is a lot of extra work.
b. This algorithm could be very useful, if we know **all nodes on the path to goal is relatively in front of their siblings**. For example, if you are searching the word "bed" in a search space of all 3-letter combinations, Iterative Broadening search could be very efficient. If the algorithm is searching letter by letter, and let each letter be a node. Since "b" is the 2nd letter of all 26 alphabets, "e" is the 5th, and "d" is the 4th, all nodes are relatively in front of their siblings. This means, when the number of children increases to 4, the search algorithm would be able to find the solution (which is very fast, faster than DFS and BFS).

Programming:

1. Result as following:



```
Searcher                     romania_map(Arad, Bucharest)   romania_map(Oradea, Neamt)     romania_map(Eforie, Timisoara)
breadth_first_tree_search    (<  21/  22/  57/Buch>, 450)   (<1033/1034/2925/Neam>, 867)   (< 182/ 183/ 477/Timi>, 837)
breadth_first_graph_search   (<   6/   9/  15/Buch>, 450)   (<  18/  20/  44/Neam>, 867)   (<  15/  19/  36/Timi>, 837)
depth_first_graph_search     (<   7/   8/  17/Buch>, 733)   (<  11/  12/  27/Neam>, 835)   (<  11/  12/  28/Timi>, 1181)
uniform_cost_search          (<  12/  13/  30/Buch>, 418)   (<  19/  20/  45/Neam>, 835)   (<  19/  20/  44/Timi>, 805)
astar_search                 (<   5/   6/  15/Buch>, 418)   (<  17/  18/  42/Neam>, 835)   (<  14/  15/  35/Timi>, 805)
                                mean: 493.8000 std: 134.6707    mean: 847.8000 std: 17.5271     mean: 893.0000 std: 161.7900
```

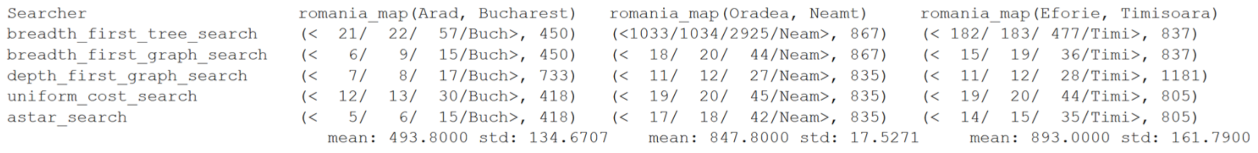
From the table, we can see that both Uniform Cost Search and A* Search found the optimal solution all 3 times. Depth First Graph Search did get lucky the second time, finding the optimal solution while being the fastest. While Breadth First Tree Search and Breadth First Graph Search are both BFS, they always find the same solution, which is just a little bit worse than the optimal solution. Depth First Graph Search is the tricky one: it could get lucky and find the optimal solution, or could get unlucky and find a not-so-good solution.

Depth First Graph Search is almost always the fastest (expect the first time A* being faster). A* Search is just a tiny bit slower than DFGS but still very fast. Interestingly, while Breadth First Tree Search and Breadth First Graph Search are both BFS, there is a huge difference between them. Breadth First Tree Search ended up did way more goal checking and is way slower than and Breadth First Graph Search. Finally, Uniform Cost Search is not so fast, ended up somewhere between Breadth First Tree Search and Breadth First Graph Search.

2. Code:

```python
from search import *
from utils import name, print_table
import statistics

def compare_searchers(problems, header,
                      searchers=[breadth_first_tree_search,
                                 breadth_first_graph_search,
                                 depth_first_graph_search,
                                 uniform_cost_search,
                                  astar_search]):
    def do(searcher, problem):
        path_indx = 0
        p = InstrumentedProblem(problem)
        x = searcher(p)
        return p, x.path_cost

    table = [[name(s)] + [do(s, p) for p in problems] for s in searchers]

    print_table(table, header)
    print ("\t\t\t", end=" ")

    for i in range(len(problems)):

        mylist = []
        for j in range(len(searchers)):
                path_cost = table[j+1][i+1][1]
                mylist.append(path_cost)

        print("\tmean: %.4f std: %.4f" %(mean(mylist), statistics.stdev(my
list)), end=" ")

    print("")

compare_searchers(
    problems=[
            GraphProblem('Arad', 'Bucharest', romania_map),
            GraphProblem('Oradea', 'Neamt', romania_map),
            GraphProblem('Eforie', 'Timisoara', romania_map)
        ],
    header=['Searcher',
             'romania_map(Arad, Bucharest)',
             'romania_map(Oradea, Neamt)',
             'romania_map(Eforie, Timisoara)'])
```

The results are already posted above in problem 1. Just posting it again here:



```
Searcher                    romania_map(Arad, Bucharest)   romania_map(Oradea, Neamt)     romania_map(Eforie, Timisoara)
breadth_first_tree_search   (<  21/  22/  57/Buch>, 450)   (<1033/1034/2925/Neam>, 867)   (< 182/ 183/ 477/Timi>, 837)
breadth_first_graph_search  (<   6/   9/  15/Buch>, 450)   (<  18/  20/  44/Neam>, 867)   (<  15/  19/  36/Timi>, 837)
depth_first_graph_search    (<   7/   8/  17/Buch>, 733)   (<  11/  12/  27/Neam>, 835)   (<  11/  12/  28/Timi>, 1181)
uniform_cost_search         (<  12/  13/  30/Buch>, 418)   (<  19/  20/  45/Neam>, 835)   (<  19/  20/  44/Timi>, 805)
astar_search                (<   5/   6/  15/Buch>, 418)   (<  17/  18/  42/Neam>, 835)   (<  14/  15/  35/Timi>, 805)
                               mean: 493.8000 std: 134.6707    mean: 847.8000 std: 17.5271     mean: 893.0000 std: 161.7900
```