Huan Tran

Homework 2

1. **[20 points] The functions *g* and *h* each play a different role in A*. What are those roles? What happens when you emphasize or de-emphasize one of them by using different weights in *f(n)*? Consider the case in which *f(n) = (1 - w)g(n) + wh(n)*, with *0 <= w <=1*. This weighted A* search is described briefly in the textbook (4th edition, sect 3.5.4). Be specific and analyze what happens for different values of *w*.**
   - *g(n)* returns the cost of the path from the root to the current node
   - h(n) returns an estimated cost of the cheapest path from the current state at node n to a goal state
   - When one of g or h is de-emphasised or emphasized, the search begins to behave differently depending on the value of the weight. If we emphasize the h, the search could potentially stop yielding optimal answer but since it marches toward the goal more aggressively, the search would use less memory, or expand less nodes and reduce the time to find a goal

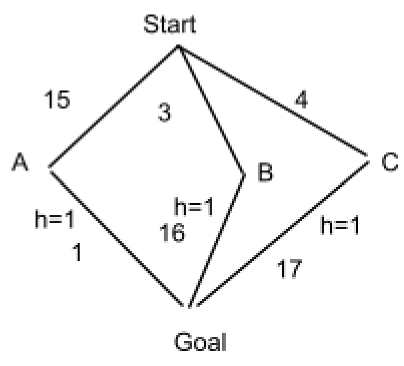
w = 0	f(n) = g(n) which makes this search becomes a uniform-cost search (complete and optimal)
0 < w < 0.5	g(n) is weighted more than h(n), which makes this search optimal
w = 0.5	f(n) = 0.5 * [g(n) + h(n)] which makes this search becomes a regular A* search (complete and optimal)
0.5 < w < 1	h(n) is weighed more than g(n), which makes this search not optimal. However, the heuristic could be more accurate, thus reducing the number of nodes expanded and lessening the time to reach a goal.
w=1	f(n) = h(n) which makes this search becomes a greedy best-first search (incomplete and not optimal)

2. **[20 points] You want to reduce the memory used by A*. You come up with the following idea: you keep in the queue only the N best nodes (i.e, the nodes with lower costs), for some positive value of N. When the queue is full and a**

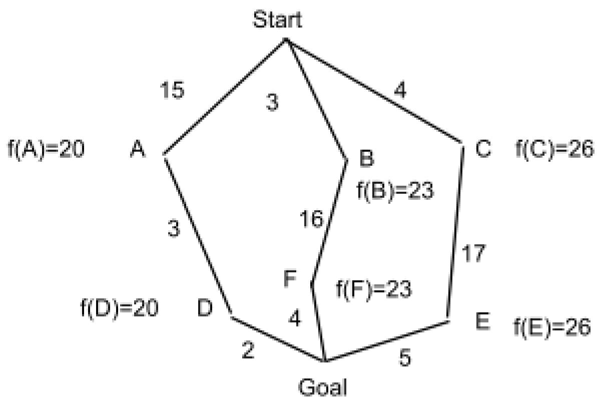
new node has to be stored, the worst node is deleted from the queue and removed from consideration.

1. **If an admissible heuristic is used, will the modified algorithm find the optimal solution? Explain why (or why not). Are there any additional constraints?**

   No, there are cases where this new algorithm will not find an optimal solution. One of the edge cases is when N = 1, which means at any time, there could only be one node in the queue. The example below shows a case where the algorithm will not return an optimal answer. There are several reasons behind this non-optimality but the main problem with this method is that the node that lead to the optimal path could be deleted from the queue. In the case of N=1, the only one node can be in the queue at a time, and there is no guarantee from the heuristic that the f(n) value of the node leading to the optimal path would be smallest when it is on the queue.

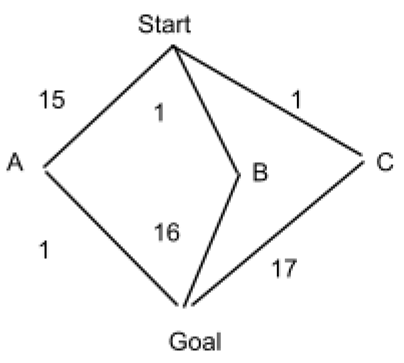
 Start 15 3 4 A B C h=1 1 h=1 16 h=1 17 Goal	For the tree in the picture , after the algorithm expands the root node, only node B will be kept in the queue since f(B)=4 is smallest among three nodes: A with f(A)=16, B with f(B)=4, and C with f(C)=5. However, the optimal path to the goal is through node A, which will not be returned since node A was ignored earlier.
An example when this new algorithm will not return an optimal solution	

2. **If a perfect heuristic is used, i.e. *for all n h(n)=h*(n)* , where *h*(n)* is the cost of the optimal path from *n* to goal, will the modified algorithm find the optimal solution? Explain.**

 Start 15 3 4 f(A)=20 A B f(B)=23 C f(C)=26 3 16 17 F f(F)=23 f(D)=20 D 4 E f(E)=26 2 5 Goal	Yes, when h(n)=h*(n) the new algorithm will alway return the optimal solution even in the extreme case (N=1). It is clear that if the algorithm could return an optimal solution with N = 1, it will always return an optimal solution with N > 1. So let assume N = 1 and h(n) = h*(n). We know that: f(n) = g(n) + h(n) = g(n) + h*(n) = sum of all nodes on the path with the lowest cost to the goal and contains the current node. This means f(n) of any nodes will give you the exact cost of the locally optimal path to the goal the node is on (like in the picture), and if you follow these locally optimal paths, all the nodes on each of the locally optimal paths will have the same f(n) value. Because of the above property, once a node on the globally optimal path is generated, it will always be placed at the first position in the queue and expanded first. Therefore, the first position on the queue always contains a node on the globally optimal path, and since N=1, only this optimal path will be explored and returned. Hence, the new algorithm will return the optimal cost even when N=1. thus it will do so when N > 1

3. **[10 points] Answer the following questions on Uniform Cost search briefly but precisely:**
    1. **Is it possible for Uniform Cost to expand more nodes than Breadth-First search? Feel free to use an example to support your answer.**

Yes, it is possible. One of the reason is the Breadth-First search uses early-goal test while Uniform cost search uses late-goal test

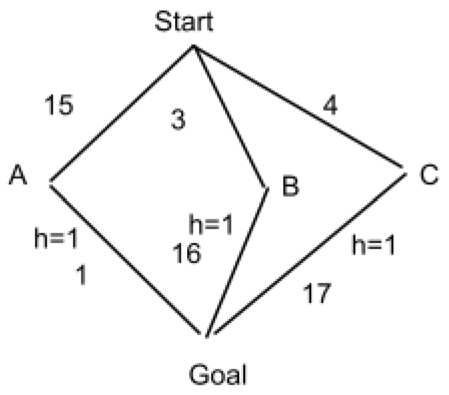
 Start 15 1 1 A B C 1 16 17 Goal	For the tree in the picture ● Breadth-first search only expands the root node and the node A. The algorithm stops after expanding A because when the algorithm generated the child node of A, it found the goal and since it uses an early-goal test, it stops right after it has found the goal. ● Uniform-cost search will expand the root, B, C, A before it stops searching.
An example when Uniform cost search expands more nodes than Breadth-First search	

2. **Does Uniform Cost search expand more nodes than A*? Why (or why not)?**

This is not always true because looking at the f(n) functions, for uniform cost search, f(n) = g(n) while for A*, f(n) = g(n) + h(n). With a bad heuristic function like h(n) = constant, A* behaves identically to a Uniform-cost search, thus both A* and the uniform-cost search will expand the same number of nodes. Moreover, if the heuristic function is admissible but not consistent, when a node is reached, it is not guaranteed to be on an optimal path, which means the same node with several different states could be on the queue, thus the same node could be expanded more than once. Given this behavior, it is possible that a uniform-cost search with an additional feature to remove non-optimal duplicated node on the queue could expand less node than the given A* search since this modified Uniform-cost search only expands each node at most once.

4. **[10 points] Does the fact that A* is "optimally efficient" mean that A* will never expand more nodes than any other algorithm?**

No, according to the definition of “optimal efficiency”, an A* search with a consistent heuristic is optimally efficient , which means any algorithm that extends search paths from the initial state, and **uses the same heuristic information,** must expand all nodes that are surely expanded by A*. This means algorithm that does not use the same heuristic information like an uninformed-search could still expand less node than A*. For example, in the search tree below, a breadth-first search will reach the optimal goal with less nodes expended comparing to an A* search.



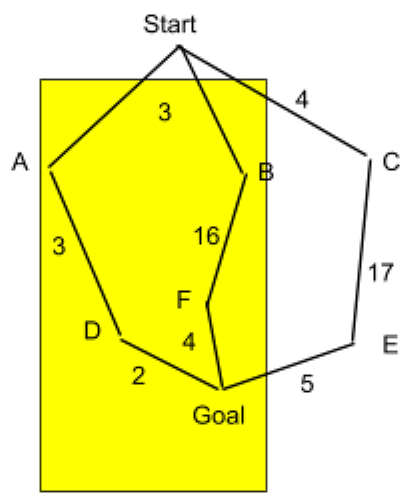
Moreover, if the heuristic function is admissible but not consistent, when a node is reached, it is not guaranteed to be on an optimal path, which means the same node with several different states could be on the queue, thus the same node could be expanded more than once. Given this behavior, it is possible that a uniform-cost search with an additional feature to remove non-optimal duplicated node on the queue could expand less node than the given A* search since this modified Uniform-cost search only expands each node at most once.

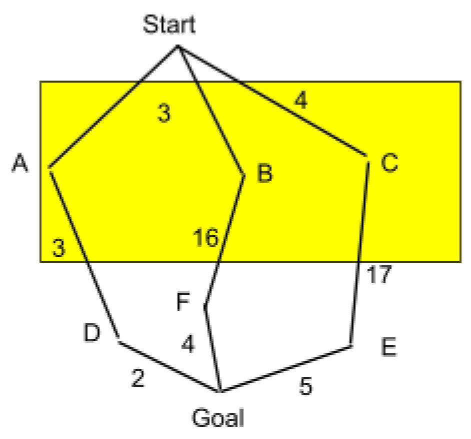
5. **[20 points] Inspired by the iterative deepening algorithm, you decide to design an "iterative broadening algorithm". The idea is to start with 2 children, and do depth-first search limiting at each node expansion the number of children to 2. If you fail to find a solution, you restart the search from the beginning increasing the number of children by 1. Repeat this process until you find a solution.**
    1. **What advantages, if any, do you see in this algorithm? What shortcomings?.**

Advantages	Shortcomings
● Given the same amount of memory usage, breadth-first search only broadens the	● Not optimal since the algorithm only returns the first goal it discovers

search path, depth-first search only lengthens the search path while this algorithm could broaden the search path and also penetrate deep down the tree. ● For a search tree that has goals lying deep and a high branching factor, this algorithm will have a higher chance to find a solution than breadth-first search or depth-first search alone since breadth-first search expands the search aggressively horizontally but subtly vertically and contrast for depth-first search	● Incomplete on state space with infinite depth since the depth-first search part still runs forever when the tree has infinite depth ● Search time is high since the algorithm might have to repeat a lot of expansions from the previous branching factor. ● High memory usages because it deepens like depth-first search and broaden like depth-first search

2. **For what type of search spaces do you think this algorithm will be useful?**

 Start 3 4 A B C 3 16 17 F D 4 E 2 5 Goal	This algorithm would be very useful when you have a state space with a large branching factor, shallow/finite depth. This algorithm is more useful in this state space because the algorithm advances the frontier both horizontally and vertically at a higher rate than only depth-first search or breadth-first search do. Moreover, given a limited memory, this algorithm could automatically adjust its expansion to go deep and then wide, which yields a higher chance of finding a solution than applying depth-first search or breadth-first

 Start 3 4 A B C 3 16 17 F D 4 E 2 5 Goal	search alone. The pictures on the left show how the algorithm advances its frontier given a limited memory. Initially, with a small branching factor, the algorithm could penetrate deep down in the tree. When branching factor increases, the algorithm expands the frontier wider and gradually reduces its reachable depth.

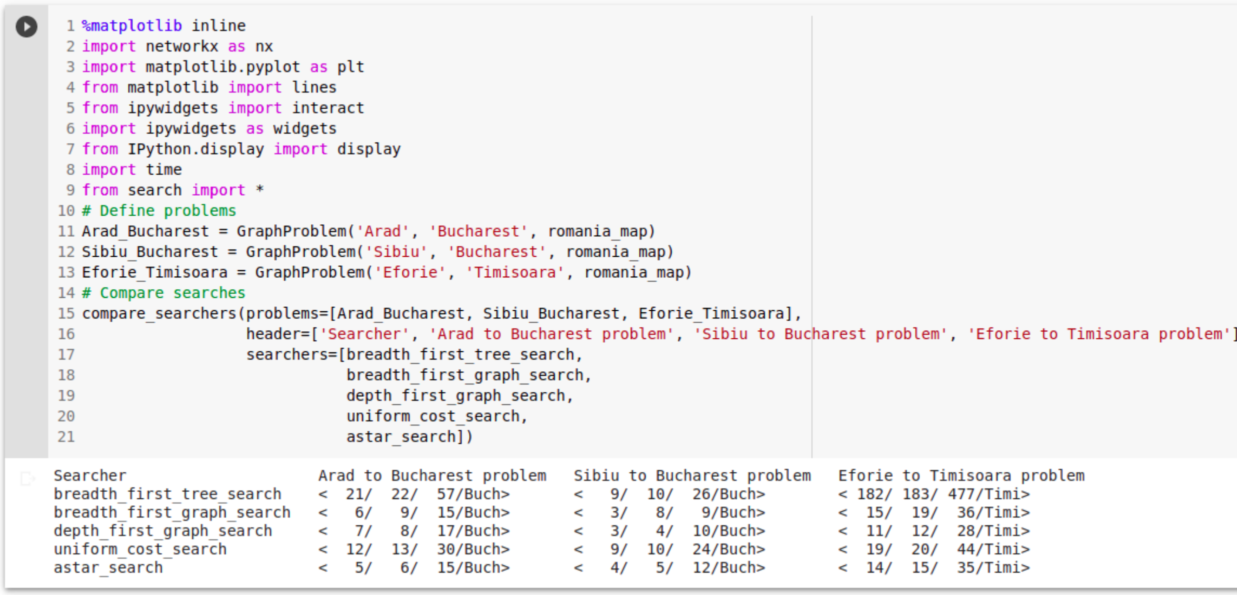
Programming part (20 points) --- This part is now finalized

For this programming part we will use the Romanian map shown in the textbook and already defined in the aima code. You can use the notebook from the previous assignment, adding the call to the compare_searchers function. For the second question, you need to modify the compare_searchers function to include the path cost. Submit the code you wrote and the results you got.

1. [10 points] Compare the performance (using the function compare_searchers) of the following search algorithms:

    1. breadth_first_tree_search,
    2. breadth_first_graph_search,
    3. depth_first_graph_search,
    4. uniform_cost_search
    5. astar_search
       to solve these problems:
    6. find a path from Arad to Bucharest
    7. find a path from Sibiu to Bucharest
    8. find a path from Eforie to Timisoara

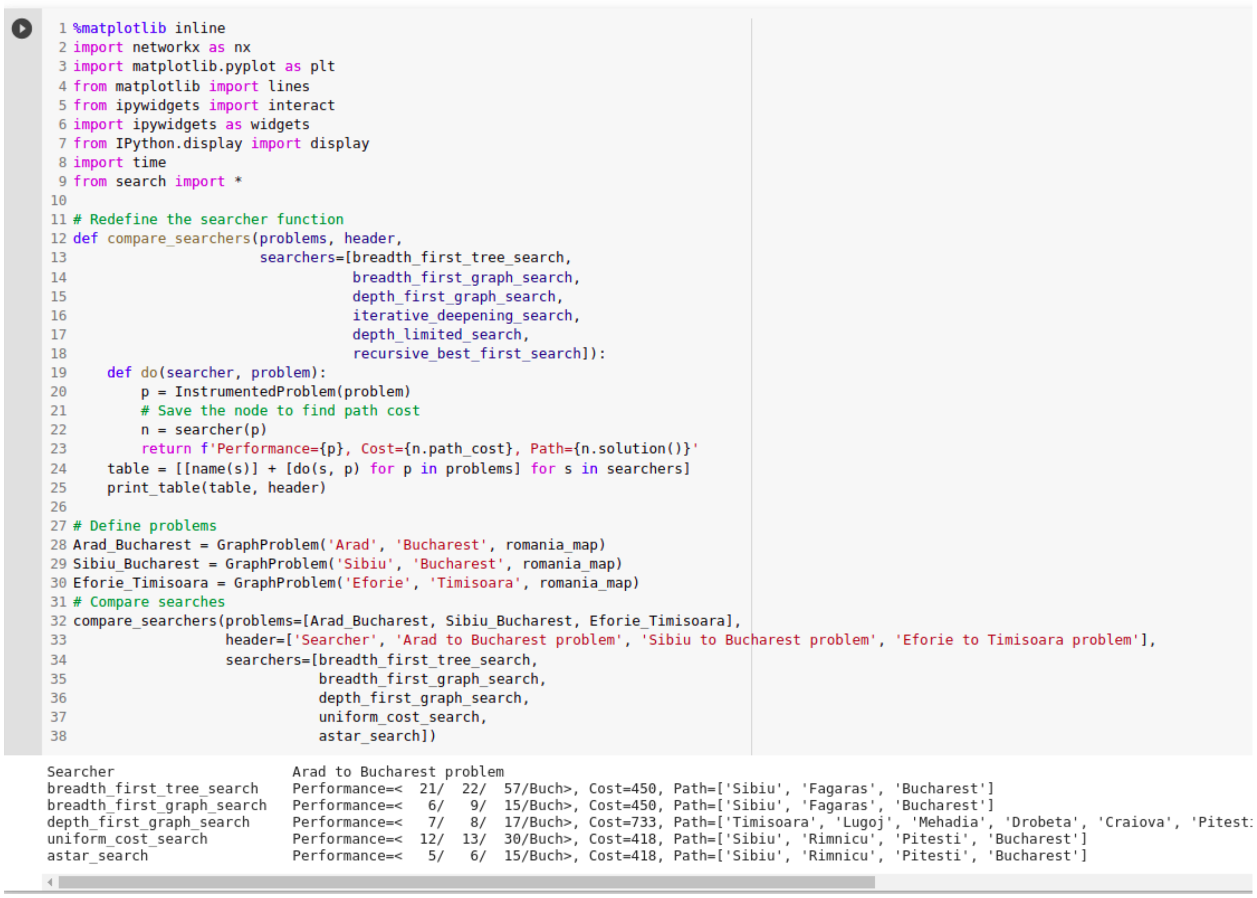
Source code is attached along with this pdf



```python
%matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import lines
from ipywidgets import interact
import ipywidgets as widgets
from IPython.display import display
import time
from search import *
# Define problems
Arad_Bucharest = GraphProblem('Arad', 'Bucharest', romania_map)
Sibiu_Bucharest = GraphProblem('Sibiu', 'Bucharest', romania_map)
Eforie_Timisoara = GraphProblem('Eforie', 'Timisoara', romania_map)
# Compare searches
compare_searchers(problems=[Arad_Bucharest, Sibiu_Bucharest, Eforie_Timisoara],
                  header=['Searcher', 'Arad to Bucharest problem', 'Sibiu to Bucharest problem', 'Eforie to Timisoara problem']
                  searchers=[breadth_first_tree_search,
                             breadth_first_graph_search,
                             depth_first_graph_search,
                             uniform_cost_search,
                             astar_search])
```

```
Searcher                     Arad to Bucharest problem   Sibiu to Bucharest problem   Eforie to Timisoara problem
breadth_first_tree_search    <  21/  22/  57/Buch>       <   9/  10/  26/Buch>        < 182/ 183/ 477/Timi>
breadth_first_graph_search   <   6/   9/  15/Buch>       <   3/   8/   9/Buch>        <  15/  19/  36/Timi>
depth_first_graph_search     <   7/   8/  17/Buch>       <   3/   4/  10/Buch>        <  11/  12/  28/Timi>
uniform_cost_search          <  12/  13/  30/Buch>       <   9/  10/  24/Buch>        <  19/  20/  44/Timi>
astar_search                 <   5/   6/  15/Buch>       <   4/   5/  12/Buch>        <  14/  15/  35/Timi>
```

2. [10 points] Add to the compare function above the cost of each solution to a problem for each of the algorithms. This requires reading the code to find out where the path cost is saved and to modify the compare_searchers function to include the path cost.



```python
%matplotlib inline
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib import lines
from ipywidgets import interact
import ipywidgets as widgets
from IPython.display import display
import time
from search import *

# Redefine the searcher function
def compare_searchers(problems, header,
                      searchers=[breadth_first_tree_search,
                                 breadth_first_graph_search,
                                 depth_first_graph_search,
                                 iterative_deepening_search,
                                 depth_limited_search,
                                 recursive_best_first_search]):
    def do(searcher, problem):
        p = InstrumentedProblem(problem)
        # Save the node to find path cost
        n = searcher(p)
        return f'Performance={p}, Cost={n.path_cost}, Path={n.solution()}'
    table = [[name(s)] + [do(s, p) for p in problems] for s in searchers]
    print_table(table, header)

# Define problems
Arad_Bucharest = GraphProblem('Arad', 'Bucharest', romania_map)
Sibiu_Bucharest = GraphProblem('Sibiu', 'Bucharest', romania_map)
Eforie_Timisoara = GraphProblem('Eforie', 'Timisoara', romania_map)
# Compare searches
compare_searchers(problems=[Arad_Bucharest, Sibiu_Bucharest, Eforie_Timisoara],
                  header=['Searcher', 'Arad to Bucharest problem', 'Sibiu to Bucharest problem', 'Eforie to Timisoara problem'],
                  searchers=[breadth_first_tree_search,
                             breadth_first_graph_search,
                             depth_first_graph_search,
                             uniform_cost_search,
                             astar_search])
```

```
Searcher                     Arad to Bucharest problem
breadth_first_tree_search    Performance=<  21/  22/  57/Buch>, Cost=450, Path=['Sibiu', 'Fagaras', 'Bucharest']
breadth_first_graph_search   Performance=<   6/   9/  15/Buch>, Cost=450, Path=['Sibiu', 'Fagaras', 'Bucharest']
depth_first_graph_search     Performance=<   7/   8/  17/Buch>, Cost=733, Path=['Timisoara', 'Lugoj', 'Mehadia', 'Drobeta', 'Craiova', 'Pitest
uniform_cost_search          Performance=<  12/  13/  30/Buch>, Cost=418, Path=['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest']
astar_search                 Performance=<   5/   6/  15/Buch>, Cost=418, Path=['Sibiu', 'Rimnicu', 'Pitesti', 'Bucharest']
```

Source code and result attached along with this pdf