Homework 2

Li Miao (miao0044)

By submitting my exam I certify that I did not consult with anyone during the exam.

1. The state space for this problem is all possible positions of blocks.

The initial state is the initial positions of all blocks given.

The goal condition is to have 3 blocks located at a specific location, one on top of another.

The actions is to move a block if the block is on top of a table or on top of a stack.

1. A heuristic I propose is the height of block stacks at the specific location on the table. This is an admissible heuristic.
2. The two different solutions are all optimal and will have the same objective function values. As suggested by the theorem of A\* search that the search always return the optimal solution.
3. First of all there must be different optimal solutions in the problem for A\* to solve. The two students might have used different heuristics, and the small differences in heuristics let A\* to expand different paths while searching for solutions. So it ended up with two different optimal solutions.

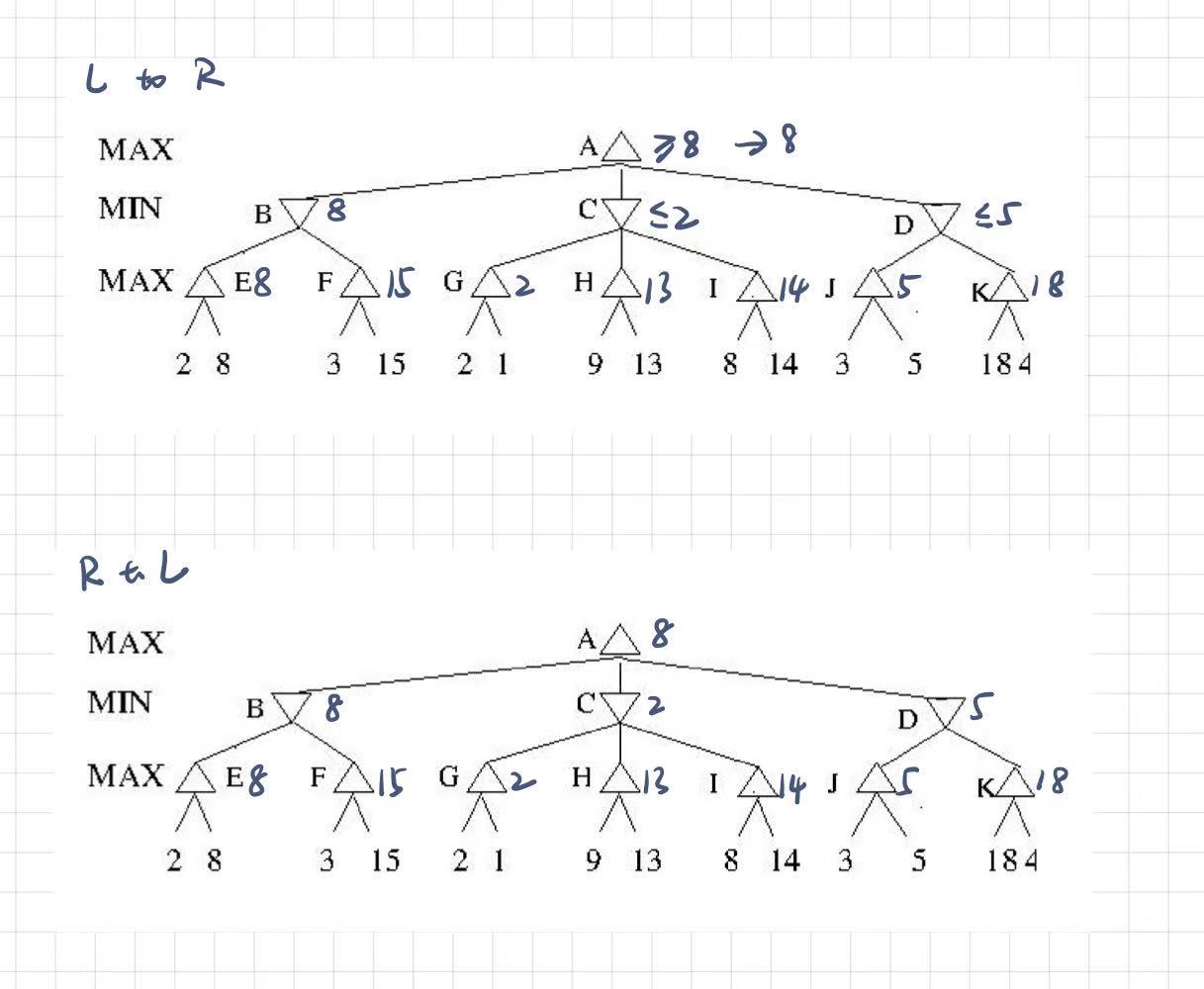
a. No. Admissible heuristics don’t always return the same value for each state. So there is no guarantee that as long as a heuristic is admissible then the algorithm will expand in the same order the same states.

b. Yes. As long as the branching factor is finite, the BFS algorithm will always find a solution if one exists. It will only take a lot of time if the depth is infinite.

c. Depth-first search, also known as backtracking algorithm, when given an incremental formulation of CSP problem. Because CSPs in an incremental formulation can be seem as a search problem. So we can borrow the idea of search algorithms to solve it. And backtracking is one of these algorithms.

d. IDA\* stands for iterative deepening A\* algorithm, and it is better than A\* in a sense that it cost a lot less memory ( space ) because it keeps a shorter list of nodes in the memory to explore. So when the state space is huge, IDA\* is more efficient with the memory required.

1. I would use backtracking search algorithms. It will conduct a search that assigns value to one variable at a time and backtracks when there is no good values to assign. Using this algorithm I can find all the possible assignments. 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.
2. Incremental formulation will work the best because we can start with one variable and start assigning values to other variables that doesn’t violate constraints. It’s a lot more efficient than permutating among variables and try to change them to fit the constraints. We also don’t care how we arrived at a solution ( the paths ), so incremental formulation is better for this problem.
3. Yes. These two heuristics are helpful in selecting the most influential variables to try out first and can help avoid meaningless trials.
4. For n variables with on average d possible values ( domain ). Then the total search space is d^n
   1. For 1 increase in the number of variable, the search space increases by d.
   2. For 1 increase in the size of domains, the search space increases by O(nd^(n-1) )
5. The pruning is shown in the charts below:



a. No. It didn’t prune the same. For the two directions, different branches were expanded.

b. Yes, the final chosen action is the same.

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

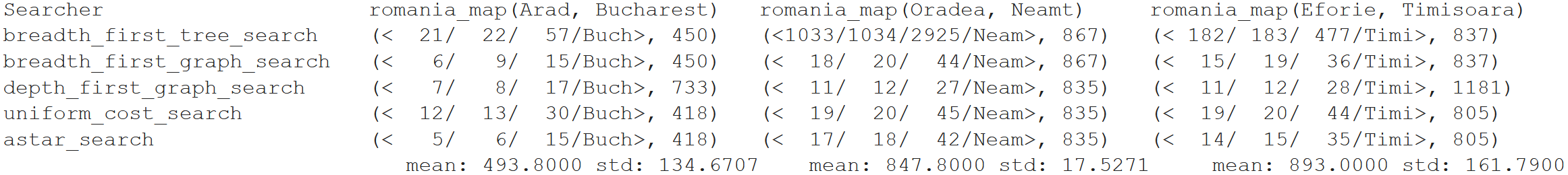
1. It has few advantages. First, it is memory efficient. Just like DFS, it only needs to store nodes from root to the current node. Also, if all nodes on the solution is in front of most of their sibling nodes, Iterative Broadening search would find the solution very fast.

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.

1. 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:



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.

1. Code:

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(mylist)), 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:

