Homework 2: Search Problems (100 pts)

Due: Monday, February 6 at 11:59pm

Instructions

- Submit a pdf of your work to gradescope, making sure to mark all pages relevant to each outlined section on gradescope (including relevant code)
- Append all written code to the end of your pdf
- In addition, submit all code files to the corresponding assignment on Canvas
- You have 3 late days, with a 10 point grade reduction per day. No submissions will be accepted after the third late day.

Overview

This assignment covers topics in problem solving such as search (including uninformed and informed search methods), local search methods, and constraint satisfaction problems. You will apply search methods to solve a variety of problems, some of which you will have to formulate yourself. Some problems will require you to apply these methods by hand while others will require you to implement them as a computer program.

Task 1: Search Formulation (20 points)

In this problem you will learn to formulate problems as instances of search problems on which you can run the algorithms we've talked about in class. We will be considering the fox, goose, and bag of beans puzzle, which, like the similar Missionaries and Cannibals problem discussed in class, is a famous riddle that has existed in many forms across many cultures for centuries. https://en.wikipedia.org/wiki/Fox,_goose_and_bag_of_beans_puzzle

You need to transport a fox, a goose, and a bag of beans across a river. Your boat only holds yourself and one of the three at a time, so

you must leave the other two on the bank of the river—either the initial bank you start on, or the far bank to which you eventually want to get. If left alone together, the fox would eat the goose, and likewise the goose would eat the beans. (So for example, you cannot leave the fox and the goose alone on the far bank of the river while you go back for the beans, nor can you take the fox across the river while leaving the goose and the beans alone.) The challenge of the riddle is to figure out how to eventually get all three to the far bank of the river without any of them devouring the other.

a. Develop the problem formulation that can be used to search for a solution, providing a detailed description of the following:

1. A specific state representation that captures important details and leaves out unimportant ones.

2. The initial state, expressed in your representation.

3. The goal test, expressed in terms of your representation.

4. The possible actions in any state and what successor state(s) they lead to, again in terms of your state representation.

5. A path cost function defined in terms of the actions and/or states. There might be more than one reasonable choice for this; briefly justify your choice.

b. Also, give numbers for the following:

1. How many different states are representable given your choice of representation?

2. How many different representable states satisfy the goal test?

3. How many of the representable states are “legal” (don’t violate any constraints)?

4. How many different legal states are reachable (without passing through illegal states) given your choice of representation, the initial state, and the set of actions?

Task 2: Uninformed and Informed Search (30 points)

The components of a search problem can be encoded into a state space graph like the one shown below.

- Each vertex (A-I) represents a state of the world.
- The initial state is labeled with “Start”.
- All states (here there is just one) that satisfy the goal test are labeled with “Goal”.
- Directed edges show which actions (X, Y, or Z) transition between which states. The cost of each action is also shown.
- Values of one particular heuristic are shown (e.g. h=5) for each state.

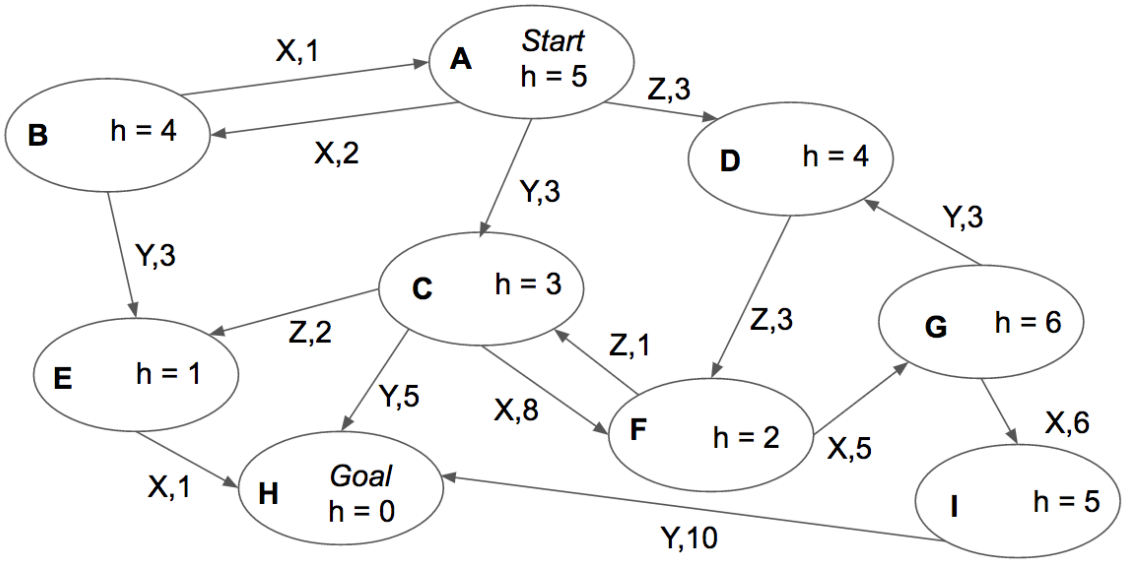

Figure 1: The example graph you will use for this task.

You will work through the execution of various search algorithms as they attempt to find a path to get from A, the start state, to a goal state. Use the general search process discussed in class (goal test only applied on examination/expansion), summarized in the textbook in Figure 3.7. When writing out your solutions to the problems below, use this notation:

- You can indicate a node by the path it represents (e.g. ADF)
- Write out the fringe as a list. For example, the fringe after expanding node A could be written as: [AB, AC, AD]

• The "front" of the fringe is on the left. In the above, AB would be expanded next. When expanding a node and generating its successors, assume the available actions (X, Y, or Z) are processed in alphabetical order.

Here are two examples to rule out any possible confusion.

Expanding Node ADF with Breadth First Search

Actions	Successor Nodes	Updated Fringe (a queue)
X, Z	ADFG, ADFC	[…, ADFG, ADFC]

Expanding Node ADF with Depth First Search

Actions	Successor Nodes	Updated Fringe (a stack)
X, Z	ADFG, ADFC	[ADFC, ADFG, …]

To show your work, show which node is expanded and which successor nodes are generated at each step. Also write out the contents of the fringe after and, if appropriate, the contents of the explored set. If a successor is generated but immediately discarded or if a node on the fringe is replaced, cross it out. As an example of the format we're looking for, here's the execution of the start of breadth first (graph) search:

Breadth First Search

Fringe	Examined/ Expanded	Successors	Explored Set
[A]	A	AB, AC, AD	{A}
[AB, AC, AD]	AB	~~ABA~~, ABE	{A, B}

If something goes wrong that prevents your algorithm from finding a solution, show your work up to that point and include a description of why the algorithm fails.

Show the execution of the following (graph) search algorithms:
a. Breadth First Search

b. Depth First Search
c. Iterative Deepening Search (show all iterations)

Now, we will consider "best-first" algorithms that use an *evaluation function* f(x) to prioritize node examination/expansion. If node ACF has f(ACF) = 5, you should now write it as ACF(5). When adding new nodes to the fringe, first add them to end and then make sure to sort the nodes in a *stable* fashion. That is, if two nodes have equal priority, the one that has been in the fringe longer will be examined/expanded first. And if two successor nodes have the same value, they will appear in their generated order.

Show the execution of the following search algorithms:
d. Uniform-Cost Search
e. Greedy Best-First Search
f. A* Search

The heuristic in the graph above turns out to be admissible (and consistent). Consider specifically the heuristic value for state F, h(F)=2.

Find a different (nonnegative real) value for h(F) that would make the heuristic:
g. Inadmissible
h. Admissible but inconsistent
In both cases, explain your reasoning.

Task 3: Local Search (30 points)

The goal of the 8 queens problem, as discussed in the book (pg. 71 & pg. 109) is to find a placement of 8 queens on a chessboard where no two queens can attack each other. (Recall that queens can move as far as they want in vertical, horizontal, or diagonal lines.) Choosing any programming language you prefer, you are going to implement the (steepest-descent) hill climbing and random-restart (steepest descent) hill climbing algorithms to search for a solution to this problem.

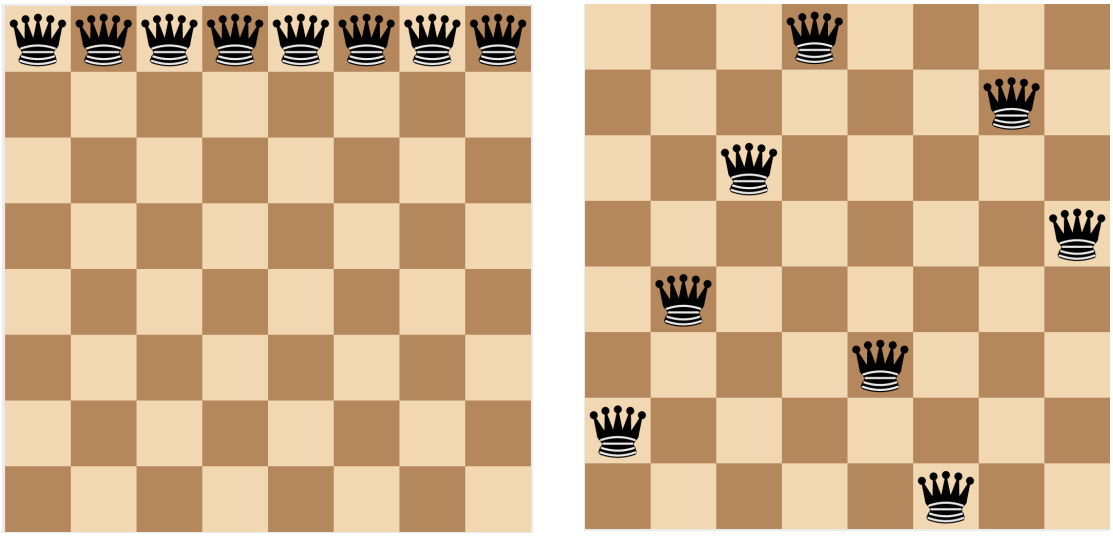

Figure 2. Left: the initial state Right: A possible solution

Suppose each queen is assigned to a particular column, and at each step of the search you are allowed to move a single queen to another square in the same column. *The heuristic cost function is the number of pairs of queens attacking each other, either directly (meaning that one queen could capture the other, if this were an actual game and they were of different colors) or indirectly (meaning that another queen is in the way, but otherwise the queens could capture each other).*

Decide on a state representation and implement a cost function that takes as input a given state and returns the heuristic cost value.

a. How would you represent the initial state and the solution state in Figure 2? What are the outputs of your function for each of these two states?

Next, implement a function that calculates the cost after each possible single move, given a current state, and returns the best move. If no possible single move provides improvement, then the function should return the current state. Break ties by choosing the move in the column farthest to the left. Given a tie within that column, choose the move closest to the top of the board.

The steepest-descent algorithm starts with an initial state and chooses a move that provides the greatest improvement. It then

repeats this process with the state resulting from the chosen move, over and over again. The algorithm should halt when no single move provides an improvement.

b. Using the initial state given in Figure 2, run your steepest-descent algorithm. Was a solution found? What is the final state after the algorithm terminates? What is the number of moves taken to reach that final state?

Using the steepest-descent algorithm from above, implement the random-restart hill climbing algorithm. For each (re)start, you should choose a new, random initial state and begin steepest-descent with this new initial state. This process should continue in a loop until a solution (no attacking queens) is found, or until some limit on the number of allowed (re)starts is reached. In your implementation, set the limit on allowed (re)starts to 10.

c. Run your random-restart algorithm at least 5 times. (Note, because of randomness it most likely won't behave the same way each run!) For each run of the algorithm, record the current best solution (the number of attacking queens is sufficient) found for each (re)start (including the final iteration right before the algorithm terminates), the final state (best overall solution found across the (re)starts) when the algorithm terminates, as well as the total number of moves taken (summed over the (re)starts during a run). Was a solution found in each run? If not, how often was one found?

d. Describe briefly how you can modify the steepest descent algorithm above using simulated annealing to solve the 8 queens problem. You do not need to provide pseudo-code or implement it. What is an advantage compared with the steepest-descent algorithm used in part b? Does it have any disadvantages?

Task 4: Constraint satisfaction (20 points)

You have been given the task of creating the University of Michigan's football in conference schedule for the next (odd numbered) year. The potential opponents and their home/away situation for each weekend are listed below.

Team	Weeks available for a Michigan home game	Weeks available for a Michigan away game
Michigan State		2
Nebraska	6	1, 6
Minnesota	4	3,4
Iowa	1,2,4,6	2
Northwestern	5,6	3,7
Indiana	6, 7	
Illinois	1	
Ohio State	5, 9	
Penn State	7	6
Purdue	7,9	
Wisconsin	7	

The first five teams listed are all in the Legends Division of the Big Ten Conference, and the final six are all in the Leaders Division of the Big Ten Conference.

The schedule must meet the following constraints.

- Michigan can only play against one team each week
- Michigan only has one bye week (a week where they don't play any opponent)
- Michigan cannot play the same team twice
- Michigan must play Michigan State, and Ohio State
- Michigan State is an away game for Michigan
- Ohio is a home games for Michigan
- Michigan must play all five other teams in the Legends Division of the Big Ten Conference
- Michigan must play three teams from the Leaders Division of the Big Ten Conference
- Michigan's last game of the season must be against Ohio State
- Michigan must play exactly four home games and four away games

The variables in this problem are W1, W2, ... W9 representing the nine weeks of in conference play. Write values as the name (or abbreviation) of an opposing school followed by an (a) or (h) for an

away or home game, respectively, or just “bye” for the value corresponding to a bye week. For example, assigning Michigan to play Ohio State at home on week 9 could be written by assigning W9 the value OSU (h).

Initially each week's domain is all the listed teams (11) at home + all the listed teams away + the possibility of not playing that week (for a total of 23 possibilities). The product of the (constrained) domain sizes for every variable is the number of candidate solutions, as it is an upper bound on the number of possible assignments that a simple CSP solver needs to check. Before applying the unary constraints there are $23^9 \approx 10^{12}$ candidate solutions.

a. Apply the unary constraints to limit the domains of all the variables. Show the resulting domains. Now how many candidate solutions are there?

b. If you were to perform a backtracking search with no mechanisms for forward checking, in the order of W1, W2, ... W9, expanding the variables in the order they are shown in the above chart (MSU(h), MSU(a), Neb(h), Neb(a), ... Wisc(a)) followed by the bye week option, how many value-to-variable assignments will be made during the search? How many backtracking steps (from a variable to the previous variable) will take place? (For this and other parts of the problem, you can show your work for getting your answers if you like, to receive partial credit, but it isn’t required. You can also use a program if desired, but it isn’t essential.)

c. Starting with the domains found in part 1, do a backtracking search this time using forward checking and the MRV heuristic to order the variables. Ties in the MRV heuristic should be broken so that the variables are assigned in the same relative order as in part 2. How many value-to-variable assignments will be made? How many backtracking steps (from a variable to the previous variable) will take place?