Homework 2: Search Problems

Arya Farahi

02/06/2017

1 Search Formulation

In this problem you will learn to formulate problems as instances of search problems on which you can run the algorithms weve 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.

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.**

   There are 4 elements, Farmer(O) - Bag of Beans(B) - Goose(G) - Fox(F), that can be on the right or left side of the river. In total there are $2^4 = 16$ states. Note that the only valid moves would be the Farmer alone or with one more object can cross the river. So the farmer can not take more than one item at a time and it is allowed to cross without carrying anything with himself. For simplicity from now on, we only used O/B/G/F. We can use one list where the first element is O, the second is B, the third is G, and finally the fourth is F. Each element can be R or L means that whether the object is on the right (1) or left (-1) side of the river, for example one state is [1,1,1,1].

2. **The initial state, expressed in your representation.**

   We assume everyone is on the Right side of the river, so the initial state would be [1,1,1,1].

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

   Then the goal state would [-1,-1,-1,-1].

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

   The possible actions are defined whether the Farmer(O) is one the right side or left side. In each action the state of O changes from R to L (L to R), it is also allowed one other object changes from R to L (L to R). For example if the state is [1,1,1,-1] then the successors would be the following [-1,1,1,-1], [-1,-1,1,-1], [-1,1,-1,-1]. Always the first element would flip the sign and according to first element sign one other element may flip the sign.

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.**

   Well, let's assume that the boat is consuming fuel, and its consumption would be proportional to the weight of boat itself and weight of objects in it. Now the cost path would be how much money the farmer need to pay for the fuel in each trip, then the path cost would be different if the farmer carrying a goose or fox or beans or nothing.

Also, give numbers for the following:

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

   There are in total $2^4 = 16$ states.

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

   Only one $[-1, -1, -1, -1]$, we want them all to be in the left side, assuming they all started on the right side.

3. **How many of the representable states are legal (dont violate any constraints)?**

   The farmer should be on the same side when at least one of the following happens the goose and fox are together, the goose and the beans are together. The following states are legal: $[1, 1, 1, 1]$, $[1, 1, 1, -1]$, $[1, 1, -1, 1]$, $[1, -1, 1, 1]$, $[1, -1, 1, -1]$, $[-1, -1, -1, -1]$, $[-1, -1, -1, 1]$, $[-1, -1, 1, -1]$, $[-1, 1, -1, -1]$, $[-1, 1, -1, 1]$. There are in total 8 legal states.

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?**

   All of them are reachable within this framework.

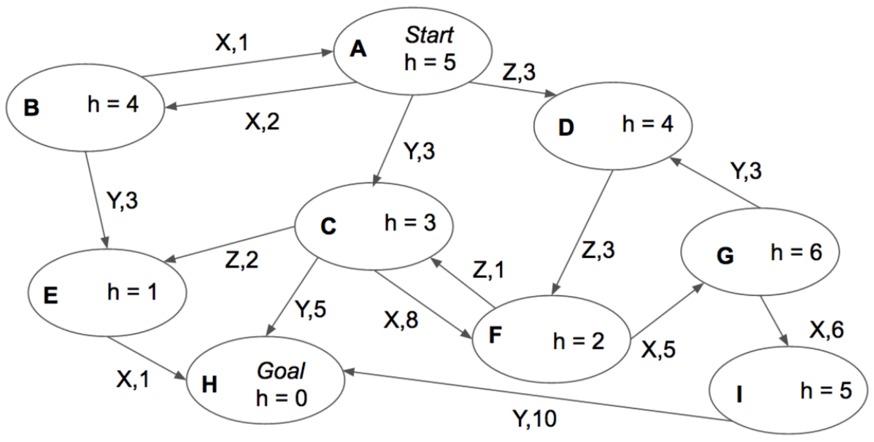

Figure 1: The example graph you will use for this task.

2 Uninformed and Informed Search

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.**

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.**

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 were looking for, heres the execution of the start of breadth first (graph) search:

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:

Breadth First Search

Look at table 1. The last line is the goal state.

Table 1: Breadth First Search

Fringe	Examinde	Successors	Explored Set
[A]	A	AB, AC, AD	{A}
[AB, AC, AD]	AB	~~ABA~~, ABE	{A, B}
[AC, AD, ABE]	AC	ACF, ACH. ACE	{A, B, C}
[AD, ABE, ACF, ACH, ACE]	AD	ADF	{A, B, C, D}
[ABE, ACF, ACH, ACE, ADF]	ABE	ABEH	{A, B, C, D, E}
[ACF, ACH, ACE, ADF, ABEH]	ACF	ACFG,~~ACFC~~	{A, B, C, D, E, F}
[ACH, ACE, ADF, ABEH]	ACH	-	-

Table 2: Depth First Search

Fringe	Examinde	Successors	Explored Set
[A]	A	AB, AC, AD	{A}
[AB, AC, AD]	AB	~~ABA~~, ABE	{A, B}
[ABE, AC, AD]	ABE	ABEH	{A, B, E}
[ABEH, AC, AD]	ABEH	-	-

Depth First Search

Look at table 2. The last line is the goal state.

Iterative Deepening Search (show all iterations)

Look at table 3. The last line is the goal state.

Show the execution of the following search algorithms:

- **Uniform-Cost Search**

  Look at table 4. The last line is the goal state.

- **Greedy Best-First Search**

  Look at table 5. The last line is the goal state.

- **A* Search**

  Look at table 6. The last line is the goal state.

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 (non-negative real) value for h(F) that would make the heuristic:

Table 3: Iterative Deepening Search

Fringe	Examinde	Successors	Explored Set	Level
[A]	A	-	{A}	0
[A]	A	AB, AC, AD	{A}	1
[AB, AC, AD]	AB	-	{A, B}	1
[AC, AD]	AC	-	{A, B, C}	1
[AD]	AD	-	{A, B, C, D}	1
[A]	A	AB, AC, AD	{A}	2
[AB, AC, AD]	AB	~~ABA~~, ABE	{A, B}	2
[ABE, AC, AD]	ABE	-	{A, B, E}	2
[AC, AD]	AC	ACF, ACH, ~~ABA~~	{A, C, B, E}	2
[ACF, ACH, AD]	ACF	-	{A, C, B, E, F}	2
[ACH, AD]	ACH	-	-	2

Table 4: Uniform-Cost Search Tree

Fringe	Examinde	Successors	Explored Set
[A(0)]	A	AB(2), AC(3), AD(3)	{A}
[AB(2), AC(3), AD(3)]	AB	~~ABA(3)~~, ABE(5)	{A, B}
[AC(3), AD(3), ABE(5)]	AC	ACE(5), ACH(8), ACF(11)	{A, B, C}
[AD(3), ABE(5), ACE(5), ACH(8), ACF(11)]	AD	ADF(6)	{A, B, C, D}
[ABE(5), ACE(5), ADF(6), ACH(8), ACF(11)]	ABE	ABEH(6)	{A, B, C, D, E}
[ACE(5), ADF(6), ABEH(6), ACH(8), ACF(11)]	ACE	ACEH(6)	{A, B, C, D, E}
[ADF(6), ABEH(6), ACEH(6), ACH(8), ACF(11)]	ADF	~~ADFC(7)~~, ADFG(11)	{A, B, C, D, E, F}
[ABEH(6), ACEH(6), ACH(8), ACF(11), ADFG(11)]	ABEH	-	-

Table 5: Greedy Best-First Search

Fringe	Examinde	Successors	Explored Set
[A(5)]	A	AC(3), AB(4), AD(4)	{A}
[AC(3), AB(4), AD(4)]	AC	ACH(0), ACE(1), ACF(2)	{A, C}
[ACH(0), ACE(1), ACF(2), AB(4), AD(4)]	ACH	-	-

- **Inadmissible, explain your reasoning:**

  h(F)=7, The shortest path to the goal is 6 FCH. If heuristic reads 7 then it is larger than the shortest path and it is over estimating the path FCH, so it is inadmissible.

- **Admissible but inconsistent, explain your reasoning:**

  If h(F)=5 then the value is still admissible because the length of shortest path , which is FCH, is 6. But is not consistent. Because h(F) > C(C→F) + h(C), $6 > 3 + 1 = 4$.

Table 6: A* Search

Fringe	Examinde	Successors	Explored Set
[A(5)]	A	AB(6), AC(6), AD(7)	{A}
[AB(6), AC(6), AD(7)]	AB	~~ABA(8)~~, ABE(6)	{A, B}
[AC(6), ABE(6), AD(7)]	AC	ACE(6), ACH(8), ACF(13)	{A, B, C}
[ABE(6), ACE(6), AD(7), ACH(8), ACF(13)]	ABE	ABEH(6)	{A, B, C, E}
[ACE(6), ABEH(6), AD(7), ACH(8), ACF(13)]	ACE	ACEH(6)	{A, B, C, E}
[ABEH(6), ACEH(6), AD(7), ACH(8), ACF(13)]	ABEH	-	-

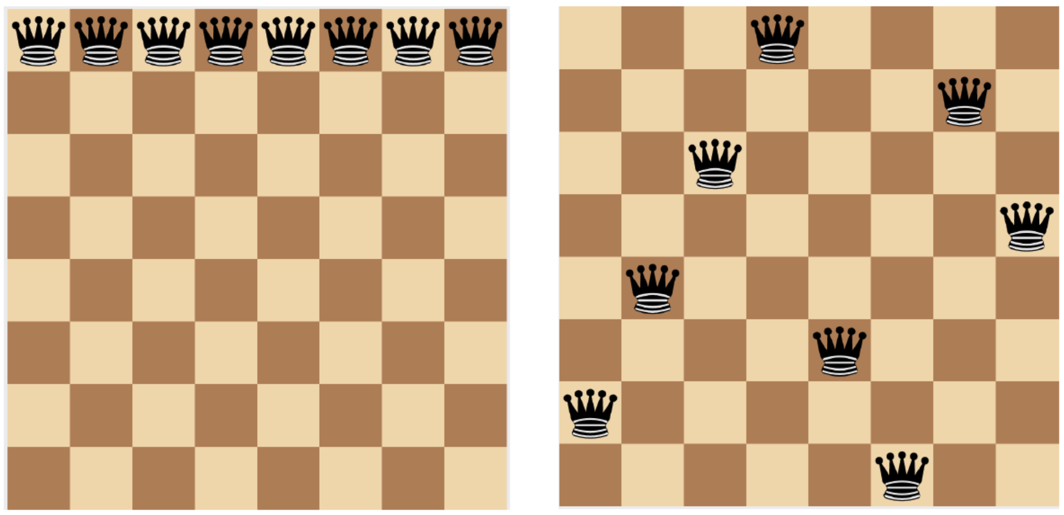

Figure 2: Left: the initial state, Right: A possible solution

3 Local Search

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.

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?

We will have a list of eight element which element would represent a queen, and each element gives us the location of the queen on the board, for example the initial state would be [1, 1, 1, 1, 1, 1, 1, 1], and the solution state on the right side of figure 2 is [7, 5, 3, 1, 6, 8, 2, 4]. The output of [1, 1, 1, 1, 1, 1, 1, 1] is 28 and [7, 5, 3, 1, 6, 8, 2, 4] is 0. Note that within this representation the first element is corresponding to the first queen and because queens do not change the column it would be always the only queen in that column. the following is the implementation of the cost function.

```
def cost_function(state):
    cost = 0
    for i in range(8):
        queen1 = state[i]
        for j in range(i+1, 8):
            queen2 = state[j]
            # check whether they are on the same row
            if queen1 == queen2:
                cost += 1
            # check whether they are on the diagonal line
            if abs(queen1 - queen2) == abs(i - j):
                cost += 1
    return cost
```

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?

Nope it did not find a solution. This is the final state after the algorithm terminates: [2, 8, 1, 7, 4, 6, 1, 5], and the number of moves are 6. Here is the list of moves:

```
0: [1, 1, 1, 1, 1, 1, 1, 1]
1: [1, 8, 1, 1, 1, 1, 1, 1]
2: [2, 8, 1, 1, 1, 1, 1, 1]
3: [2, 8, 1, 7, 1, 1, 1, 1]
4: [2, 8, 1, 7, 1, 6, 1, 1]
5: [2, 8, 1, 7, 1, 6, 1, 5]
6: [2, 8, 1, 7, 4, 6, 1, 5]
```

and the cost of final state is 1.

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 wont 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?

The solution was not found in each run. 4 out of 5 the solution is found (80% of the time). In case the solution was not found the best scenario would be the final state with the the lowest cost function, if there are many then we check for the least number, and if there are many of such restarts then we choose the first restart.

```
Run =  1
  Best Trial Solution:
     Initial State :  [3, 2, 7, 5, 8, 8, 6, 2]
     Final State :  [4, 2, 7, 5, 1, 8, 6, 3]
     Cost of the finale State :  0
     Number of Moves :  3
  Last Trial Solution:
     Initial State :  [3, 2, 7, 5, 8, 8, 6, 2]
     Final State :  [4, 2, 7, 5, 1, 8, 6, 3]
     Cost of the finale State :  0
     Number of Moves :  3
  Total Number of Moves: 19
  Solution Found: Yes
------------------------------------------------
Run =  2
  Best Trial Solution:
     Initial State :  [7, 7, 6, 7, 8, 5, 7, 4]
     Final State :  [2, 8, 6, 1, 3, 5, 7, 4]
     Cost of the finale State :  0
     Number of Moves :  5
  Last Trial Solution:
     Initial State :  [7, 7, 6, 7, 8, 5, 7, 4]
     Final State :  [2, 8, 6, 1, 3, 5, 7, 4]
     Cost of the finale State :  0
     Number of Moves :  5
  Total Number of Moves: 32
  Solution Found: Yes
---------------------------------
Run =  3
  Best Trial Solution:
     Initial State :  [6, 1, 1, 1, 2, 8, 2, 3]
     Final State :  [5, 7, 1, 4, 2, 8, 6, 3]
     Cost of the finale State :  0
     Number of Moves :  5
  Last Trial Solution:
     Initial State :  [6, 1, 1, 1, 2, 8, 2, 3]
     Final State :  [5, 7, 1, 4, 2, 8, 6, 3]
     Cost of the finale State :  0
     Number of Moves :  5
  Total Number of Moves: 8
  Solution Found: Yes
---------------------------------
Run =  4
  Best Trial Solution:
     Initial State :  [7, 1, 3, 3, 4, 8, 2, 7]
     Final State :  [7, 5, 3, 1, 6, 8, 2, 4]
     Cost of the finale State :  0
     Number of Moves :  4
  Last Trial Solution:
     Initial State :  [7, 1, 3, 3, 4, 8, 2, 7]
     Final State :  [7, 5, 3, 1, 6, 8, 2, 4]
     Cost of the finale State :  0
     Number of Moves :  4
  Total Number of Moves: 4
  Solution Found: Yes
---------------------------------
Run =  5
  Best Trial Solution:
     Initial State :  [1, 8, 7, 5, 7, 3, 1, 3]
     Final State :  [6, 8, 2, 5, 7, 4, 1, 3]
     Cost of the finale State :  1
     Number of Moves :  3
  Last Trial Solution:
     Initial State :  [6, 6, 1, 6, 4, 1, 7, 3]
     Final State :  [2, 6, 8, 6, 4, 1, 7, 5]
```

```
      Cost of the finale State :  1
      Number of Moves :  3
   Total Number of Moves: 31
   Solution Found: No
--------------------------------
```

At the end you can see the full output of the code for 5 runs.

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?

For the implementation first we randomly select a column and and randomly select a new row for each queen. Then we calculate the cost if the cost is less than the cost of current state then we move to the new state otherwise with some probability we take the new state. The probability function has the form of $\exp(-(C_{\text{new}} - C_{\text{old}})/\text{k}T)$ where $\text{k}T$ is a free parameter. IT starts with high temperature and each step the temperature goes down, until it become zero.

Advantages: The advantage is that if the steepest-descent stuck in a local minimum then simulated annealing allows the algorithm to search the local neighbors and look for the global min. Theoretically this algorithm supposed to converge to the global min solution, and the initial condition does not matter.

Disadvantages: Note that the convergence rate of simulated annealing is slower than steepest descent. For a well behaved functions steepest descent would converge faster and computationally is more efficient. There is a free parameter which need to be tuned, though maybe it is not an disadvantage in general one need to be careful about this tuning and need some intuition and experience and some trial and error to make the best choice, otherwise the algorithm may converge very very slow or it may trap in a local min, and because the temperature is very small it does not get a chance to get out of the potential. Though theoretically the algorithm supposed to converge to the global min, in practice if the temperature decreases very fast, then the algorithm may find a local min instead.

```
Run =  1
   Trial number  1
      Initial State :  [4, 4, 4, 7, 4, 7, 6, 2]
      Final State :  [1, 4, 4, 7, 3, 3, 6, 2]
      Cost of the finale State :  2
      Number of Moves :  3
   Trial number  2
      Initial State :  [3, 1, 5, 7, 2, 7, 5, 7]
      Final State :  [3, 1, 6, 4, 2, 7, 5, 8]
      Cost of the finale State :  1
      Number of Moves :  3
   Trial number  3
      Initial State :  [1, 2, 2, 8, 4, 7, 5, 2]
      Final State :  [1, 4, 6, 8, 2, 7, 5, 3]
      Cost of the finale State :  1
      Number of Moves :  4
   Trial number  4
      Initial State :  [8, 1, 2, 2, 8, 3, 7, 6]
      Final State :  [3, 1, 4, 2, 8, 3, 7, 2]
      Cost of the finale State :  2
      Number of Moves :  3
   Trial number  5
      Initial State :  [7, 4, 2, 2, 1, 4, 6, 1]
      Final State :  [7, 5, 2, 8, 1, 4, 6, 3]
      Cost of the finale State :  1
      Number of Moves :  3
   Trial number  6
```

```
     Initial State :  [3, 2, 7, 5, 8, 8, 6, 2]
     Final State :  [4, 2, 7, 5, 1, 8, 6, 3]
     Cost of the finale State :  0
     Number of Moves :  3
---------------------------------
Run =  2
  Trial number  1
     Initial State :  [5, 4, 3, 6, 1, 8, 8, 1]
     Final State :  [4, 2, 7, 6, 3, 5, 8, 1]
     Cost of the finale State :  1
     Number of Moves :  5
  Trial number  2
     Initial State :  [7, 1, 6, 6, 6, 3, 2, 2]
     Final State :  [7, 4, 1, 8, 6, 3, 6, 2]
     Cost of the finale State :  1
     Number of Moves :  4
  Trial number  3
     Initial State :  [1, 5, 2, 6, 5, 5, 2, 1]
     Final State :  [1, 5, 2, 6, 3, 7, 4, 1]
     Cost of the finale State :  1
     Number of Moves :  3
  Trial number  4
     Initial State :  [4, 4, 2, 2, 7, 4, 7, 2]
     Final State :  [4, 6, 8, 5, 7, 1, 7, 2]
     Cost of the finale State :  1
     Number of Moves :  4
  Trial number  5
     Initial State :  [8, 7, 2, 8, 4, 6, 8, 3]
     Final State :  [7, 5, 1, 1, 4, 6, 8, 3]
     Cost of the finale State :  1
     Number of Moves :  5
  Trial number  6
     Initial State :  [7, 5, 5, 1, 2, 6, 2, 5]
     Final State :  [7, 3, 8, 3, 1, 6, 2, 5]
     Cost of the finale State :  1
     Number of Moves :  4
  Trial number  7
     Initial State :  [3, 7, 4, 8, 2, 6, 7, 5]
     Final State :  [3, 7, 4, 8, 1, 2, 7, 5]
     Cost of the finale State :  2
     Number of Moves :  2
  Trial number  8
     Initial State :  [7, 7, 6, 7, 8, 5, 7, 4]
     Final State :  [2, 8, 6, 1, 3, 5, 7, 4]
     Cost of the finale State :  0
     Number of Moves :  5
---------------------------------
Run =  3
  Trial number  1
     Initial State :  [7, 5, 3, 5, 5, 4, 1, 1]
     Final State :  [7, 5, 3, 8, 6, 4, 2, 1]
     Cost of the finale State :  1
     Number of Moves :  3
  Trial number  2
     Initial State :  [6, 1, 1, 1, 2, 8, 2, 3]
     Final State :  [5, 7, 1, 4, 2, 8, 6, 3]
     Cost of the finale State :  0
     Number of Moves :  5
---------------------------------
Run =  4
  Trial number  1
     Initial State :  [7, 1, 3, 3, 4, 8, 2, 7]
     Final State :  [7, 5, 3, 1, 6, 8, 2, 4]
     Cost of the finale State :  0
     Number of Moves :  4
---------------------------------
Run =  5
  Trial number  1
     Initial State :  [1, 8, 7, 5, 7, 3, 1, 3]
     Final State :  [6, 8, 2, 5, 7, 4, 1, 3]
     Cost of the finale State :  1
     Number of Moves :  3
  Trial number  2
     Initial State :  [2, 2, 4, 2, 8, 6, 4, 2]
     Final State :  [7, 1, 4, 2, 8, 6, 4, 2]
     Cost of the finale State :  2
     Number of Moves :  2
  Trial number  3
     Initial State :  [4, 5, 7, 2, 1, 4, 1, 7]
     Final State :  [4, 5, 7, 2, 6, 3, 1, 8]
     Cost of the finale State :  1
     Number of Moves :  3
  Trial number  4
     Initial State :  [6, 4, 6, 7, 4, 6, 5, 4]
     Final State :  [6, 4, 1, 7, 4, 6, 8, 2]
     Cost of the finale State :  2
     Number of Moves :  3
  Trial number  5
     Initial State :  [1, 2, 7, 8, 8, 6, 7, 7]
     Final State :  [3, 1, 7, 5, 8, 6, 4, 2]
     Cost of the finale State :  1
     Number of Moves :  5
  Trial number  6
     Initial State :  [7, 4, 3, 6, 3, 3, 4, 1]
     Final State :  [2, 5, 3, 6, 3, 7, 4, 1]
     Cost of the finale State :  2
     Number of Moves :  3
  Trial number  7
     Initial State :  [7, 4, 8, 6, 3, 6, 4, 6]
     Final State :  [7, 1, 8, 5, 3, 6, 4, 2]
     Cost of the finale State :  2
     Number of Moves :  3
  Trial number  8
     Initial State :  [4, 2, 8, 4, 8, 5, 5, 4]
     Final State :  [7, 2, 8, 6, 8, 1, 5, 4]
     Cost of the finale State :  2
     Number of Moves :  3
  Trial number  9
     Initial State :  [3, 6, 2, 5, 6, 8, 6, 4]
     Final State :  [3, 6, 2, 5, 1, 8, 4, 2]
     Cost of the finale State :  2
     Number of Moves :  3
  Trial number  10
     Initial State :  [6, 6, 1, 6, 4, 1, 7, 3]
     Final State :  [2, 6, 8, 6, 4, 1, 7, 5]
     Cost of the finale State :  1
     Number of Moves :  3
---------------------------------
```

4 Constraint satisfaction

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 in the HW. 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 $2^{39} \approx 10^{12}$ 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?

Here are the number of possible choices for each weak:

```
W1 = [Neb(a), Iowa(h), Illinois(h),  bye]
W2 = [MSU(a), Iowa(h), Iowa(a), bye]
W3 = [Minn(a), NWest(a), bye]
W4 = [Minn(h), Minn(a), Iowa(h), bye]
W5 = [NWest(h), OSU(h), bye]
W6 = [Neb(h), Neb(a), Iowa(h), NWest(h), Indi(h), PSU(a), bye]
W7 = [NWest(a), Indi(h), PSU(h), Purdue(h), Wisc(h), bye]
W8 = [bye]
W9 = [OSU(h), Purdue(h), bye]
```

After applying unary constraints in total there will be $4 \times 4 \times 3 \times 4 \times 3 \times 7 \times 6 \times 1 \times 3 = 72576$ states.

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 isnt required. You can also use a program if desired, but it isnt essential.)

There will be 23 value-to-variable assignments.

Here is the result:

```
Assignment 1: [Neb(a), ]
Assignment 2: [Neb(a), MSU(a), ]
Assignment 3: [Neb(a), MSU(a), Minn(a), ]
Assignment 4: [Neb(a), MSU(a), Minn(a), Iowa(h), ]
Assignment 5: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), ]
Assignment 6: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), ]
Assignment 7: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), PSU(h), ]
Assignment 8: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), PSU(h), bye,]
Backtrack 1: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), PSU(h),]
Backtrack 2: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), ]
Assignment 9: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), Purdue(h), ]
Assignment 10: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), Purdue(h), bye, ]
Backtrack 3: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), Purdue(h),]
Backtrack 4: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), ]
Assignment 11: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), Wisc(h), ]
Assignment 12: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), Wisc(h), bye, ]
Backtrack 5: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), Wisc(h), ]
Backtrack 6: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), ]
Assignment 13: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), bye, ]
Backtrack 7: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h),  ]
Backtrack 8: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), ]
Assignment 14: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), PSU(a), ]
Assignment 15: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), PSU(a), Indi(h), ]
Assignment 16: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), PSU(a), Indi(h), bye ]
Assignment 17: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), PSU(a), Indi(h), bye, OSU(h)]
complete
Solution: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), PSU(a), Indi(h), bye, OSU(h)]
```

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?

```
MRV : WEEK-8: [BYE]
Assignment 1: [, , , , , , , bye, ]
------------------------------------------------------------
Remove BYE from the domain of remaining variables (W1...W9).
W1 = [Neb(a), Iowa(h), Illinois(h)]
W2 = [MSU(a), Iowa(h), Iowa(a)]
W3 = [Minn(a), NWest(a)]
W4 = [Minn(h), Minn(a), Iowa(h)]
W5 = [NWest(h), OSU(h)]
W6 = [Neb(h), Neb(a), Iowa(h), NWest(h), Indi(h), PSU(a)]
W7 = [NWest(a), Indi(h), PSU(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-3: [Minn(a), NWest(a)]
Assignment 2: [, , Minn(a), , , , , bye, ]
------------------------------------------------------------
```

```
Remove Minn from the domain of remaining variables.
W1 = [Neb(a), Iowa(h), Illinois(h)]
W2 = [MSU(a), Iowa(h), Iowa(a)]
W4 = [Iowa(h)]
W5 = [NWest(h), OSU(h)]
W6 = [Neb(h), Neb(a), Iowa(h), NWest(h), Indi(h), PSU(a)]
W7 = [NWest(a), Indi(h), PSU(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-4: [Iowa(h)]
Assignment 3: [, , Minn(a), Iowa(h), , , , bye, ]
------------------------------------------------------------
Remove Iowa from the domain of remaining variables.
W1 = [Neb(a), Illinois(h)]
W2 = [MSU(a)]
W5 = [NWest(h), OSU(h)]
W6 = [Neb(h), Neb(a), NWest(h), Indi(h), PSU(a)]
W7 = [NWest(a), Indi(h), PSU(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-2: [MSU(a)]
Assignment 4: [, MSU(a), Minn(a), Iowa(h), , , , bye, ]
------------------------------------------------------------
Remove MSU from the domain of remaining variables.
W1 = [Neb(a), Illinois(h)]
W5 = [NWest(h), OSU(h)]
W6 = [Neb(h), Neb(a), NWest(h), Indi(h), PSU(a)]
W7 = [NWest(a), Indi(h), PSU(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-1 : [Neb(a), Illinois(h)]
Assignment 5: [Neb(a), MSU(a), Minn(a), Iowa(h), , , , bye, ]
------------------------------------------------------------
Remove Neb from the domain of remaining variables.
W5 = [NWest(h), OSU(h)]
W6 = [NWest(h), Indi(h), PSU(a)]
W7 = [NWest(a), Indi(h), PSU(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-5 : [NWest(h), OSU(h)]
Assignment 6: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), , , bye, ]
------------------------------------------------------------
Remove NWest from the domain of remaining variables.
W6 = [Indi(h), PSU(a)]
W7 = [Indi(h), PSU(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-6 : [Indi(h), PSU(a)]
Assignment 7: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), , bye, ]
------------------------------------------------------------
Remove Indi from the domain of remaining variables.
W7 = [PSU(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-9 : [OSU(h), Purdue(h)]
Assignment 8: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), , bye, OSU(h)]
------------------------------------------------------------
Remove OSU from the domain of remaining variables.
W7 = [PSU(h), Purdue(h), Wisc(h)]
MRV : WEEK-7 : [PSU(h), Purdue(h), Wisc(h)]
------------------------------------------------------------
Backtrack 1: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), , bye, ]
```

```
W7 = [PSU(h), Purdue(h), Wisc(h)]
W9 = [Purdue(h)]
MRV : WEEK-9 : [Purdue(h)]
------------------------------------------------------------
Backtrack 2: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), , , bye, ]
W6 = [PSU(a)]
W7 = [Indi(h), PSU(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-6 : [PSU(a)]
Assignment 9: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), PSU(a), , bye, ]
------------------------------------------------------------
Remove PSU from the domain of remaining variables.
W7 = [Indi(h), Purdue(h), Wisc(h)]
W9 = [OSU(h), Purdue(h)]
MRV : WEEK-9 : [OSU(h), Purdue(h)]
Assignment 10: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), PSU(a), , bye, OSU(h)]
------------------------------------------------------------
Remove PSU from the domain of remaining variables.
W7 = [Indi(h), Purdue(h), Wisc(h)]
MRV : WEEK-9 : [OSU(h), Purdue(h)]
Assignment 11: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), PSU(a), Indi(h), bye, OSU(h)]
------------------------------------------------------------
Gaol: [Neb(a), MSU(a), Minn(a), Iowa(h), NWest(h), Indi(h), , bye, OSU(h)]
```

A The Python code

Only the relevant part of the code is provided here. For the full code the grader may want to check the git repository for this homework: `https://github.com/afarahi/AI_HW1/tree/master/scr/HW2`. You can run the code by entering `python main.py HW2`

```
import numpy.random as npr

def cost_function(state):

    cost = 0

    for i in range(8):

        queen1 = state[i]

        for j in range(i+1, 8):

            queen2 = state[j]

            # check whether they are on the same row
            if queen1 == queen2:
                cost += 1

            # check whether they are on the diagonal line
            if abs(queen1 - queen2) == abs(i - j):
                cost += 1

    return cost


class steepest_descent:

    def __init__(self):

        pass

    def _move(self, state):

        new_state = state[:]
        cost = cost_function(state)

        for i in range(8):
            for j in range(1, 9):

                alt_state = state[:]
                alt_state[i] = j
                alt_cost = cost_function(alt_state)

                if alt_cost < cost:
                    new_state = alt_state[:]
                    cost = alt_cost

        return new_state

    def run(self, initial_state):
        self.state = initial_state
        self.moves = [self.state[:]]

        state = [0, 0, 0, 0, 0, 0, 0, 0]

        while self.state != state:

            state = self.state[:]
            self.state = self._move(self.state)

            self.moves += [self.state[:]]

        self.moves = self.moves[:-1]

        return self.state


def hw2_task3_pipeline():

    print "Cost function of initial state: ", cost_function([1, 1, 1, 1, 1, 1, 1, 1])
    print "Cost function of solution state: ", cost_function([7, 5, 3, 1, 6, 8, 2, 4])

    queens_problem = steepest_descent()
    print "steepest descent after termination : ", queens_problem.run([1, 1, 1, 1, 1, 1, 1, 1])
    print "Moves: ", queens_problem.moves
    print "number of moves : ", len(queens_problem.moves)-1


    npr.seed(148)

    # number of trials
    num_trials = 10

    for i in range(5):

        counter = 0
        tot_moves = 0
        final_state = [1, 1, 1, 1, 1, 1, 1, 1]

        print "Run = ", i+1

        while cost_function(final_state) != 0 and counter < num_trials:

            # setup a random initial state:
            initial_state = list(npr.randint(1, 9, size=8))
            final_state = queens_problem.run(initial_state)

            counter += 1
            tot_moves += len(queens_problem.moves) - 1

            # print out results
            print "  Trial number ", counter
            print "     Initial State : ", initial_state
            print "     Final State : ", final_state
            print "     Cost of the finale State : ", cost_function(final_state)
            print "     Number of Moves : ", len(queens_problem.moves) - 1

        print "Total Moves : ", tot_moves
        print "---------------------------------"
```