HomeWork Assignment 2

Comp 590.133

4th February 2014

Due: 11:59pm, 25th February 2014

Getting Started

What to submit: Written parts of assignment and descriptions of the programming part of the assignment are expected to be written in a file called answers.pdf (preferably in pdf format but Word or ascii are ok too.) Please also provide a short description on how to run your code in a README file.

Where to submit: Upload the document(s) and all your python code files (along with a README) to */afs/cs.unc.edu/project/courses/comp590-133-s14/students/***<your-login>***/HW2* directory by logging into classroom/snapper nodes. Where **<your-login>** should be replaced by your CS login. (The same way you submitted homework 1.)

Evaluation: Your code will be auto-graded for technical correctness.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: You are not alone! If you find yourself stuck on something, contact the course staff for help. If you can't make our office hours, let us know and we will schedule more. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask. One more piece of advice: if you don't know what a variable does or what kind of values it takes, print it out. Also, we encourage you to work in a team of size 2.

Part 1: Constraint Satisfaction Problems

Graduation Dinner (3 points)

You are helping to figure out the seating at your graduation dinner table. Your mom has already worked out seating on your side of the table, but wants you to decide where to put the relatives on the other side of the table. The relatives are your cousin Jasmine, your cousin Jason, your aunt Trudy, her husband Randy, and your aunt Misty. There are 5 seats, numbered 1-5 with 1 being the seat closest to the kitchen. You have the following constraints:

* No two relatives can sit in the same seat.
* Your cousins can't be seated next to each other because they will start a food fight.
* Jason as the eldest cousin gets the privilege of being seated closer to the kitchen than his sister Jasmine so he can get to the food first.
* Your aunt Misty shouldn't be seated next to Jason, Trudy or Randy because she invariably instigates an argument about politics and disrupts dinner.
* Additionally, Misty and Trudy have to be seated with at least 2 seats between them as a buffer.

Please answer the following written questions (no implementation):

a) Formulate this problem as a binary CSP where the variables are relatives. In particular, state the domains and binary constraints on variables (e.g. different(A,B), notAdjacent(A,B), A<B, etc ...).

b) If you ran an arc consistency algorithm on this problem what would the domains be for each variable?

c) Which variable or variables would be assigned first according to MRV using the arc-consistent domains from part b?

d) If we assume that Randy is seated in the middle (seat 3) and run arc consistency, what will the remaining domains be?

e) List all solutions to this CSP with Randy in seat 3, or state that none exist.

Hide & Seek (7 points)

a: Simple Backtracking.

Emma, one of the computer science freshman student from UNC, has N friends. Her friends want to play "hide and seek" in a park (a grid of size N x N) near the north campus. To start the game, each one of her N friends want to stand in the park (at some coordinate) such that no one can see each other. All her friends can see only in straight lines, in the 8 possible directions. The park contains trees and looking through them is not possible. Your job is to help Emma write code in python to help her friends stand in their initial positions.

Let's define the CSP such that variables are grid locations, ie {A_{11},A_{12},A_{13},....} with domains states of the grid, ie {friend, tree, empty}. Describe the constraints on the variables. Then, implement backtracking search to find an assignment that satisfies this problem. (see below for the explanation of the problem input and format of the solution). In your report, explain your approach. Also, experiment with different inputs and include these in your report as well as the number of backtracks you had to make for each input you tried. Then, plot the number of backtracks as N grows (for 4<=N<15).

Input: First line of the input contains 2 integers. First value corresponds to N - the number of friends. Second value corresponds to K - the number of trees in the park. The next K lines contain the X, Y location of the trees in the park. You can assume that the input always yields a valid answer.

Output: Return (x,y) location of each of her N friends.

Sample Input:

```
8 15
7 7
3 1
7 1
4 8
8 3
6 5
4 4
2 1
8 2
6 3
3 7
7 6
2 3
1 2
6 4
```

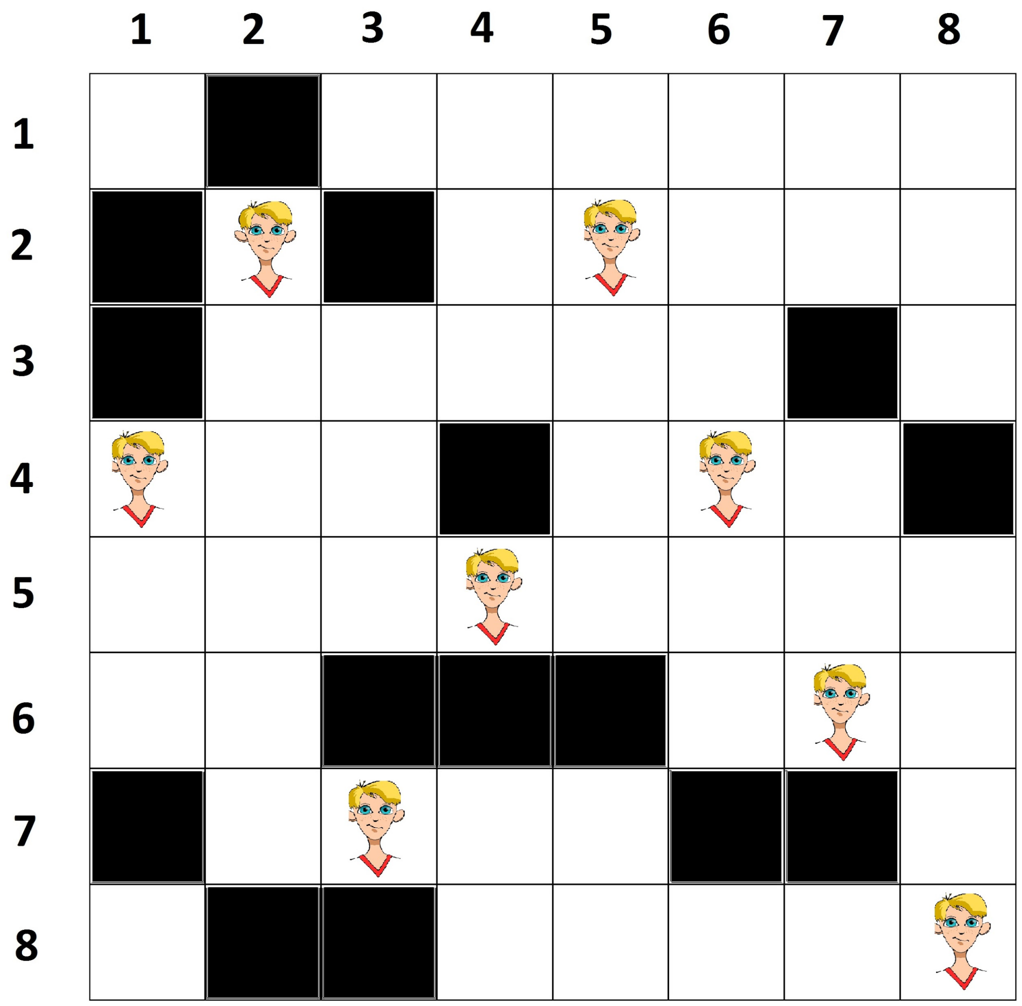
Sample Output:

```
2 2
2 5
4 1
```

```
4 6
5 4
6 7
7 3
8 8
```

Note 1: In above explanation, indices start from 1.
Note 2: Output can be in any order.
Note 3: Any possible solution is accepted.

Here is a possible solution for an 8x8 board (The black cells represent trees):



b: Heuristics.

In this part, you should experiment with various heuristics to find answers more efficiently. Here are some example heuristics you can try:

- Distance from the previously placed friend. (Local heuristic.)
- Mean distance from previously placed friends. (Global heuristic.)

Carefully mention in the report:

- The number of backtracks made with each of the heuristics.
- The total time taken by each of the heuristics for different inputs (note: sometimes your algorithm may not find a solution so try multiple random inputs).
- Are these heuristics better than simple backtracking ?
- Can you think of a better heuristic than the ones mentioned above ?

Part 2: Minimax

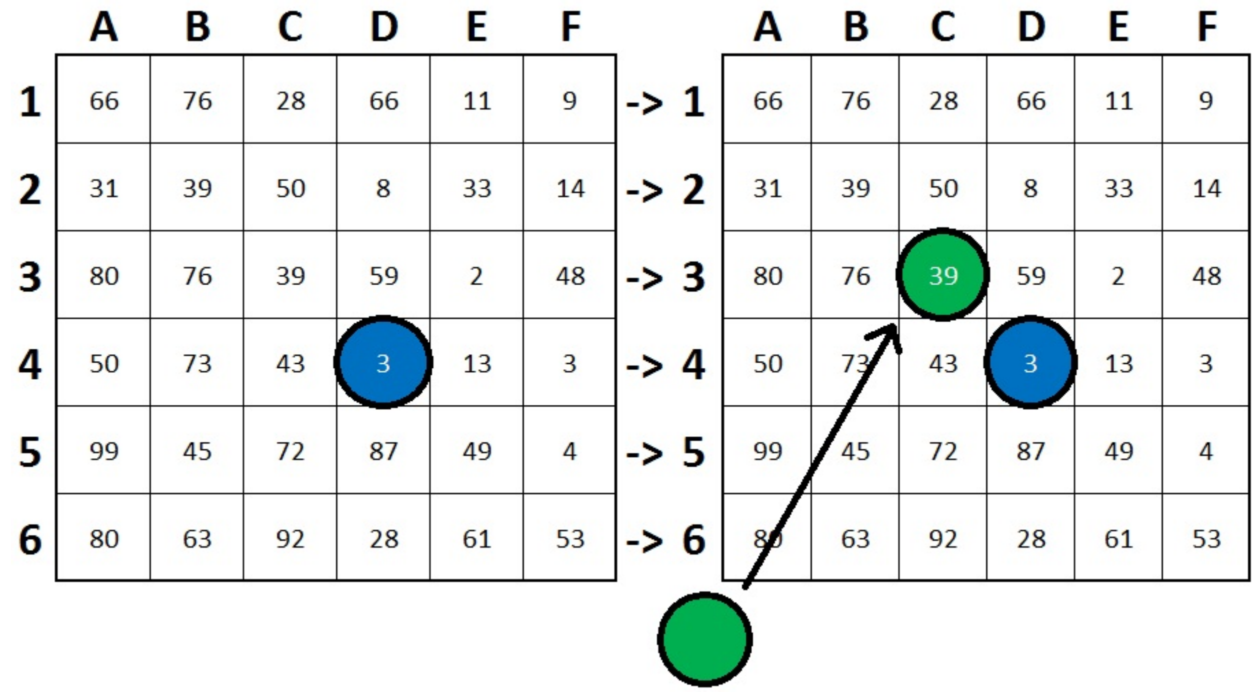
Candy game (10 points)

The goal of the game is to implement an agent to play a simple "candy game."

Rules of the game:

1. The game is played on a board of size 6x6.
2. Each cell of the board has a specified positive **value** (1 to 99).
3. There are two players, Blue and Green. Blue takes the first turn and Green takes the second and so on.
4. Each time Blue/Green "own" a cell, they place a colored candy piece in the cell to represent their ownership of the cell.
5. Blue/Green never run out of candy!
6. The objective of the game is to own cells with maximum total **value.**
7. The game ends when all the cells are occupied.
8. The values of the cells are variables that can be changed for each game, but stay constant during a game.
9. On a turn, a player can make one "**Drop and own"** move as follows: If any cell is unoccupied, Blue or Green can own it by placing a candy.
10. **Candy capture:** If a "Drop and own" move places a candy horizontally or vertically adjacent to other candies of the same color, then enemy candies that are horizontally or vertically adjacent to the placed candy are converted. Diagonal candies are not affected.

Here is a picture showing some example moves:

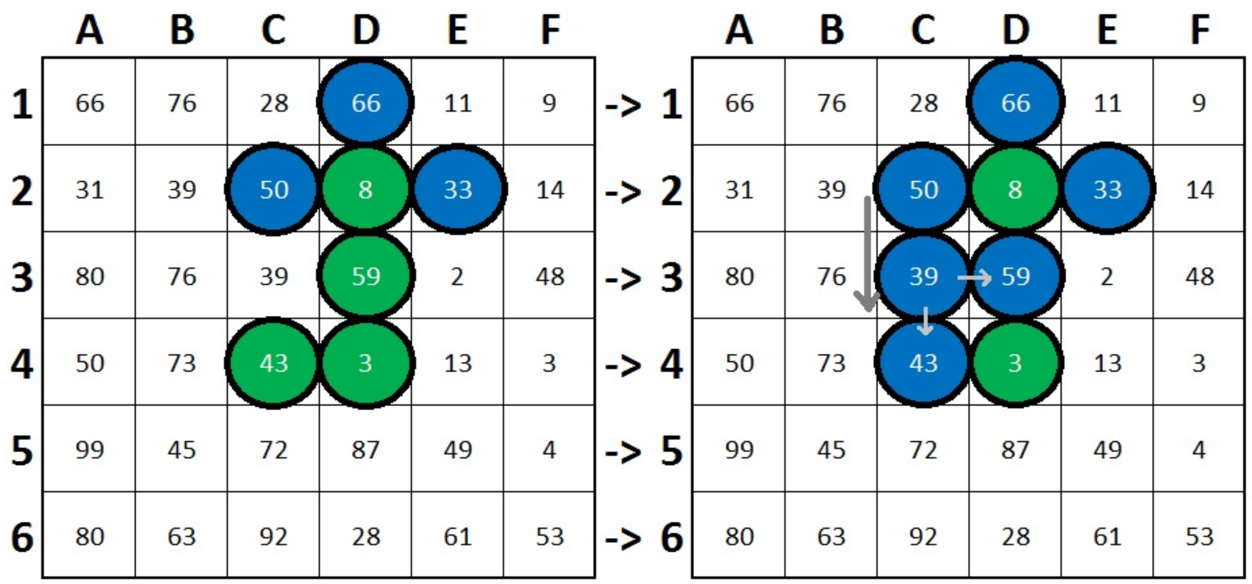


	A	B	C	D	E	F
1	66	76	28	66	11	9
2	31	39	50	8	33	14
3	80	76	39	59	2	48
4	50	73	43	3	13	3
5	99	45	72	87	49	4
6	80	63	92	28	61	53

	A	B	C	D	E	F
-> 1	66	76	28	66	11	9
-> 2	31	39	50	8	33	14
-> 3	80	76	39	59	2	48
-> 4	50	73	43	3	13	3
-> 5	99	45	72	87	49	4
-> 6	80	63	92	28	61	53

The figure above shows two “drop and own” moves. On the left, Blue makes a move, dropping its candy on [D,4] and getting 3 points. Then, as shown on the right, Green drops its candy on [C,3] and gets 39 points.

The following figure shows what happens when Blue drops its candy onto [C,3]. Because this is horizontally adjacent to the blue candy at [C,2], the green candies at [D,3] and [C,4] are “Candy Captured” and converted to blue, as shown on the right. Notice that in its next move, Green will not be able to “Candy Capture” any of Blue's candies.



	A	B	C	D	E	F
1	66	76	28	🔵66	11	9
2	31	39	🔵50	🟢8	🔵33	14
3	80	76	39	🟢59	2	48
4	50	73	🟢43	🟢3	13	3
5	99	45	72	87	49	4
6	80	63	92	28	61	53

	A	B	C	D	E	F
-> 1	66	76	28	🔵66	11	9
-> 2	31	39	🔵50	🟢8	🔵33	14
-> 3	80	76	🔵39	🔵59	2	48
-> 4	50	73	🔵43	🟢3	13	3
-> 5	99	45	72	87	49	4
-> 6	80	63	92	28	61	53

Your task is to implement different agents to play this game, one using **minimax search** and one using **alpha-beta search**. Your program should use depth-limited search with an evaluation function -- which you, of course, need to design yourself and explain in the report. Try to determine the maximum depth to which it is feasible for you to do the search (for alpha-beta pruning, this depth should be larger than for minimax). The worst-case number of leaf nodes for a tree with a depth of three in this game is roughly 42,840. Thus, you should at least be able to do minimax search to a depth of three.

For each of these five game boards (**link**), run the following matchups. Blue moves first.

1. Minimax vs minimax;
2. Alpha-beta vs alpha-beta;
3. Alpha-beta vs minimax;
4. minimax vs alpha-beta;

Carefully mention in the report:

1. The sequence of moves (e.g. "Blue: drop A3, Green: drop C1, etc. "), the final state of the board (who owns each square) and the total score of each player.
2. The total number of game tree nodes expanded by each player.
3. The average number of nodes expanded per move and the average amount of time to make a move.

For bonus points:

1. Design an interface for the game that would allow you to play against the computer. How well do you do compared to the AI? Does it depend on the depth of search, evaluation function, etc.?
2. Design your own game boards and show results on them. Try to play the game on a larger board. How large can you go?

3. Describe and implement any advanced techniques from class lectures or your own reading to try to improve efficiency or quality of gameplay.