CSCI 360 – Project #2

Theoretical Part: Bayesian Networks - 4 Points

From `http://aispace.org/bayes/` download the AIspace '`Belief and Decision Networks`' Java applet. You might need to add `http://www.aispace.org/` to the '`Exception Site List`' available in the '`Security`' tab of the '`Java Control Panel`' to get the applet to run. Read the documentation available at the download site and try it out. Do this well before the deadline to ensure that it works on your computer.

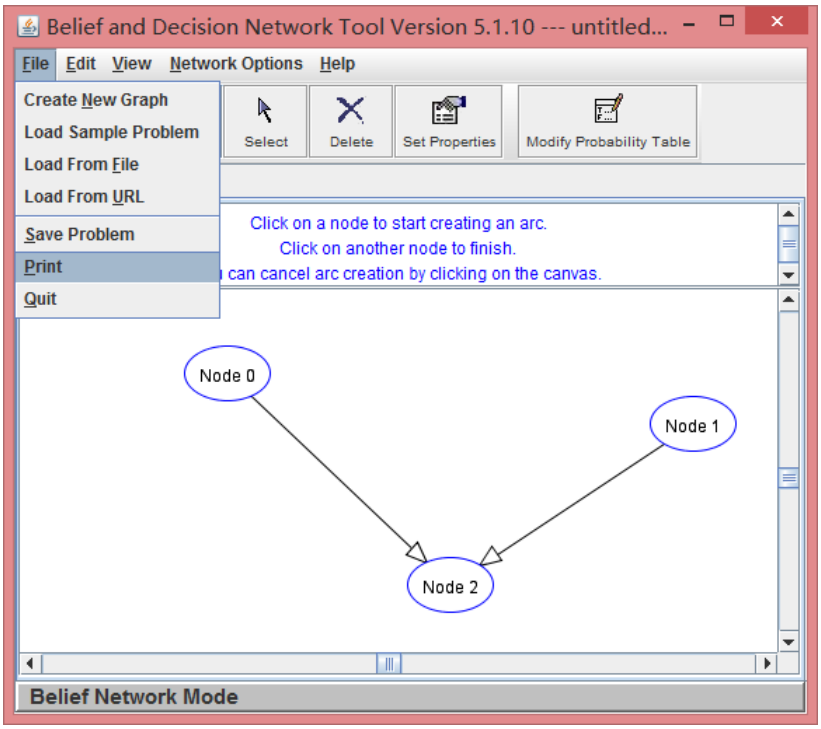

Figure 1: AIspace "Belief and Decision Networks" Java Applet

Using the Java applet, develop a Bayesian network that can be used to predict whether Donald Trump will be elected president. We expect your Bayesian network to have 10 or more nodes (that is, random variables) and a non-trivial structure. For example, you do not want to have 9 nodes that are all directly connected to the 10th node or 10 nodes that all form a line. Save your Bayesian network and include a screenshot of it in your answer. Create two interesting nontrivial scenarios where the values of some nodes are known. Show the predictions of your Bayesian network for these scenarios and explain why they make sense.

List two nodes in your Bayesian network that are independent or conditionally independent. Write down the equations that express their (conditional) independence

and verify them via queries to the Bayesian network. Provide screenshots of these queries. You can use the 'Toggle monitoring' option to display the probability distribution of a node, and see whether it changes when you make an observation about the truth value of a node. An example is given below:

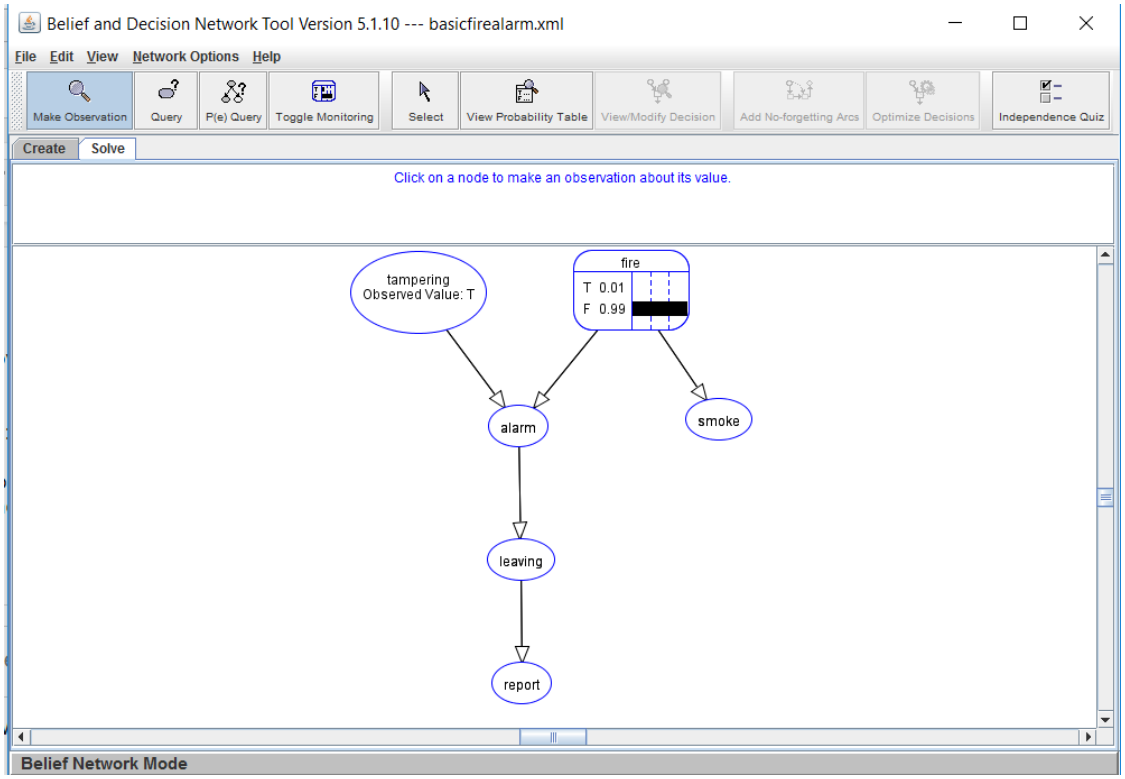

Figure 2: A Bayesian network with an observation made for node 'tempering', and monitoring enabled for node 'fire'.

Programming Part: Local Search - 6 Points

In this part of the project, you will use the local search methods you have learned in class (such as hill-climbing and simulated annealing) to generate puzzles that are optimized to be challenging and entertaining. We provide you with code that generates and evaluates the puzzles (with respect to some value function that we describe below), but you will have to implement the local search part that finds puzzles with high values. Your program needs to terminate within 1 minute of wall clock time and report the best puzzle it has found.

The Puzzle

The puzzle consists of a grid with r rows and c columns of cells. Each cell contains exactly one integer in the range from i to j (inclusive), where i and j are positive integers. The player of the puzzle has to start in the upper-left cell (= the start cell) and move with the smallest number of actions to the lower-right cell (= the goal cell).

If the player is in a cell that contains integer x, then they can perform one of at most four actions, namely either move x cells to the left (L), x cells to the right (R), x cells up (U), or x cells down (D), provided that they do not leave the grid. An example puzzle of size 5×5 is given below:

3	2	1	4	1
3	2	1	3	3
3	3	2	1	4
3	1	2	3	3
1	4	4	3	G

The shortest solution for the instance above is 19 moves: R D L R U D R L R L U D L D R R U D D.

The Value Function for the Puzzle

Below are some of the features that are expected from a 'good' puzzle that is both challenging and entertaining. We list these features just to give you an idea of the kind of puzzles that we are trying to generate. You are not expected to program anything related to these features. The code that we provide takes care of that.

- The puzzle has a solution.
- The puzzle has a unique shortest solution.
- The puzzle contains as few *black holes* (dead ends) as possible. Define a *reachable cell* as a cell that can be reached from the start with a sequence of actions. Define a *reaching cell* as a cell from which the goal can be reached with a sequence of actions. A cell is a black hole if and only if it is a reachable, non-reaching cell.
- The puzzle contains as few *white holes* as possible. A cell is a white hole if and only if it is a reaching, non-reachable cell.
- The puzzle contains as few *forced forward moves* as possible. A forced forward move occurs when there is only one action that leaves a reachable cell.
- The puzzle contains as few *forced backward moves* as possible. A forced backward move occurs when there is only one action that reaches a reaching cell.

Using these features, we develop the following value function to evaluate a given puzzle:

- We multiply the length of a shortest solution by 5.
- We add $r \times c$ (r = number of rows, c = number of columns) points if there is a unique shortest solution.
- We subtract 2 points for each white or black hole.
- We subtract 2 points for each forced forward or backward move.
- We subtract $r \times c \times 100$ points if the puzzle does not have a solution.

Generating Good Puzzles

We provide you with code that generates puzzles (but not necessarily good ones). The program reads the parameters r, c, i and j from the command line and outputs a single puzzle of size $r \times c$ where each cell contains an integer between i and j (inclusive). Running 'make' compiles and runs the code using the parameters `5 5 1 4`, which should generate an output similar to the one given below. To run the program with different parameters, you can modify the makefile to change the default parameters, or execute the program directly from the command line.

```
./PuzzleGenerator 5 5 1 4
Generating a 5x5 puzzle with values in range [1-4]

Puzzle:
3 2 1 4 2
3 4 1 2 1
4 3 2 2 1
3 2 1 3 4
1 4 2 3 0

Solution: Yes
Unique: Yes
Solution length: 14
# of black holes: 1
# of white holes: 3
# of forced forward moves: 5
# of forced backward moves: 9
Puzzle value: 59

Total time: 4.901224 seconds
```

The provided code has three classes. The `Timer` class is used to measure the wall clock time in seconds. The `Puzzle` class has a simple interface for generating random puzzles, generating a random successor or all possible successors of a puzzle (by changing the value of a single cell), and calculating the value of a puzzle. The `PuzzleGenerator` class utilizes the `Puzzle` class to generate 'good' puzzles. You will be modifying the `GeneratePuzzle()` member function of the `PuzzleGenerator` class to generate a 'good' puzzle using local search. We have already implemented a member function for `PuzzleGenerator`, called `RandomWalk`, for generating better than average puzzles. You should look at the source code of `RandomWalk()` to understand how to use the `Timer` and `Puzzle` classes. You are free to add new member functions and variables to the `PuzzleGenerator` class, but you should not modify any other source files.

Hints

- You are free to implement whichever local search algorithm you like. Note that a crossover function is not provided in the puzzle, so genetic algorithms might not be a good idea.
- Implementing a simple hill climbing algorithm would be pretty easy and it would get you some points (and generate decent puzzles when the puzzle size is small) but it is likely that it won't find very high quality puzzles (especially for larger puzzles, where the number of successors at each iteration can be very large). Nevertheless, it might be a good starting point.
- Your program has plenty of time to generate a good puzzle. Try including random restarts (and storing the best solution you have found so far), to try and find higher quality puzzles. If your current search for a good puzzle does not seem promising, you might want to consider terminating it early to save some time.
- You might not want to evaluate all the successors of a state. Hill climbing considers all the successors (for a 10×10 puzzle with cell values between 1-9, there are around 800 successors), which takes a lot of time to process. Simulated annealing randomly picks one and decides whether to move to that one. You can also 'sample' your successors by randomly generating a small number of them and picking the best one. It is also possible to change your successor selection strategy dynamically as the search progresses.
- A randomly generated puzzle might not be solvable. In this case, its score will be extremely low (you can also check if a puzzle is solvable by using the `HasSolution` member function of the `Puzzle` class). You can either discard it and try to generate a puzzle with a solution, or you can start searching. Even if the puzzle is unsolvable, its score can get higher with respect to the other quality metrics, and, once you reach a configuration that is solvable, its score will jump up.
- Most of the hints that we have listed above are about reminding you of the different things that you can do. It is up to you to mix and match the different techniques and test them out to create a powerful local search algorithm for generating high-quality puzzles.

Grading

You are not allowed to use any external libraries for your code. We have tested and verified that the provided code (and a solution to the project that extends the provided code) compiles and runs on `aludra.usc.edu`. You are free to develop your code on any platform that you choose, although we have not tested if our code works on non-Linux operating systems (and we might not be able to help you get your code

working on your preferred platform). We will test your code on a Linux machine that supports C++11.

We will test your program with several benchmarks using different parameters. For each benchmark, if the quality of your puzzle is above a certain threshold, you get full points. Otherwise, you get a grade based on the ratio of your puzzle's quality to the threshold quality. You can assume that r and c will be between 5 and 10 (inclusive). Puzzles that do not have a solution do not get any marks at all. Programs that exceed the time limit (1 minute) get points deducted for each second that they are over the time limit (rounded up).

For each benchmark, the five highest quality puzzles will get bonus points. We haven't decided how to give the bonus points yet, but it will be so that, if a program generates the highest quality puzzle for each benchmark, it will get a total of 3 bonus points (so, it is possible to get a 9 out of 6 for this part).

Submission

You need to submit your solutions through the blackboard system by Wednesday, Oct. 26, 11:59pm.

For the theoretical part, you should submit a PDF file named 'Part1.pdf' that contains the required screenshots. You also need to submit an xml file named 'Part1.xml', that contains your Bayesian network (using `'File'` → `'Save program'` saves your network as an xml file).

For the programming part, you should submit your modified 'PuzzleGenerator.cpp' and 'PuzzleGenerator.h' files.

If you have any questions about the project, you can post them on piazza (you can find the link on the course wiki) or come to the TAs' office hours.