CS360 – Project #2

Similar to Project 1, this project has a theoretical part and a programming part. This project will likely take more time to complete than Project 1, so start working on it as soon as possible.

Theoretical Part: A* (25 Points)

Answer the following questions:

Question 1

A* needs to break ties to decide which state to expand next if several states have the same smallest f-value. It can either break ties in favor states with smaller g-values or in favor of states with larger g-values. Compare both versions of A* (using a consistent heuristic) with respect to the number of expanded states. Which version will typically perform better than the other one and why?

Hint 1: You can try to simulate both versions of A* by searching an empty 4-neighbor grid with the Manhattan distance heuristic from the bottom left corner to the upper right corner to get some intuition for the answer. Remember that the Manhattan distance between two cells (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$.

Hint 2: Consider a version of A* that expands every state at most once. In the following, assume that there is only one goal state and that the version of A* that breaks ties in favor of states with smaller g-values expands non-goal states (if available) that have the same f-value and g-value as the goal state before the goal state. (If you want to have fun, then think about what happens in case this assumption is wrong - but you don't need to worry about that unless you want to.) Let c* be the cost of a cost-minimal path from the start state to the goal state.

a) Consider a version of A* that breaks ties in favor of states with smaller g-values. Argue that this version of A* expands all non-goal states whose f-values are equal to c*.

b) Consider a version of A* that breaks ties in favor of states with larger g-values. Argue that this version of A* might only expand some of the non-goal states whose f-values are equal to c*.

c) Based on a) and b), argue why a version of A* that breaks ties in favor of states with larger g-values expands no more states than a version of A* that breaks ties in favor of states with smaller g-values. (If you couldn't argue a) and/or b), assume that these statements are true.) You will need to take into account that both versions of A* can also expand states whose f-values are not equal to c*.

If the above text is too confusing to you, make again the above assumption and try to argue more directly that every state expanded by the version of A* that breaks

ties in favor of states with larger g-values is also expanded by the version of A* that breaks ties in favor of states with smaller g-values.

Question 2

Show a detailed example that demonstrates that A* used with an inconsistent but admissible heuristic has not necessarily found a cost-minimal path from the start state to a state when it is about to expand (any node labeled with) the state for the first time. Show the resulting search tree, including the node expansion order, the g-, h- and f-values of all nodes, and the pruned nodes (with an x through them), as well as the path found by the A* search and compare it to the cost-minimal path.

Programming Part: Puzzle Generation (75 Points)

In this part of the project, we want you to write a puzzle generator in C++.

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.

Below are some features expected from a 'good' puzzle:

- 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.

Note that the start cell is always reachable (therefore, it cannot be a white hole) and the goal cell is always reaching (therefore, it cannot be a black hole). Furthermore, the goal cannot have any forced forward moves (since we have found the path by that point) and the start cannot have any forced backward moves.

Puzzle Generator

You are to develop a program in C++ that 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). Your program needs to generate high quality puzzles to get good marks (see the sections below). It also needs to output the following information about the puzzle it generated: Whether the puzzle has a solution, whether it has a unique solution, solution length (use 0 if no solution exists), the number of black and white holes, the number of forced forward and backward moves, and finally, its score (next section).

Your program should terminate within 1 minute of wall clock time. We have provided three files that you should use in your code:

main.cpp The main function reads the four command line arguments and passes them to the function `GeneratePuzzle`, which is where your implementation goes. You can modify `main.cpp` however you like, but you are not allowed to change anything within the main function. We suggest that you create a class for puzzle generation and call its constructor from the function `GeneratePuzzle`. `Timer.h` needs to be included in `main.cpp` as well, since we will use it to time your projects (the elapsed time printed by the main function is the official time that we will use to make sure that your program does not exceed the time limit).

Timer.h This is a very simple timer implementation that uses the function `gettimeofday` to measure the wall clock time in seconds. "`Timer t;`" constructs a timer, "`t.StartTimer();`" starts the timer, and "`t.GetElapsedTime();`" returns the elapsed time in seconds. You should use this class to make sure your program does not exceed the time limit.

makefile We have included a sample makefile for your project. You are free to modify the makefile (without changing the name of the executable), but your code should compile with the command "make", and clean up any object files and executables that it generated while compiling with the command "make clean". We also included a "make test" command which compiles and runs the program

with command line arguments "5 5 2 4". You might need to change the make file if you add more 'cpp' files. For instance:

```
all:
$(CC) $(CFLAGS) -o $(EXEC) MyClass.cpp main.cpp
```

When printing your puzzle, use 0 to denote the goal. A sample output is given below. Your program's output should match this format (double check that the text and whitespaces do):

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

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

Solution: Yes
Unique: Yes
Solution length: 19
# of black holes: 0
# of white holes: 0
# of forced forward moves: 6
# of forced backward moves: 8
Puzzle score: 92

Total time: 59.831465 seconds
```

You can develop your code in whatever environment you want, but your submission has to compile and execute on 'aludra.usc.edu'.

Puzzle Evaluation

We will score the quality of your puzzles as follows:

- We multiply the length of a shortest solution by 5.
- We add $r \times c$ 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.

Note that these are all separate metrics. A cell can be a black hole and has a forced backward move, in which case it will lower the puzzle score by 4, since it is counted in both metrics.

Grading

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 get points deducted, for each second that they are over the time limit.

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 30 bonus points.

Hints

- The best way to approach puzzle generation might be through local search methods, such as hill climbing, simulated annealing or genetic algorithms.
- You can use two different methods of hill climbing: Evaluate all successors and pick the best one, or pick a random successor and move to it if it improves your solution. For the latter method, you can also move to a successor that makes your solution worse, with a certain probability (to escape local maxima).
- 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.
- You will need to implement a function to find the value of a given puzzle. Notice that the puzzle itself is a directed graph (for instance, in the example above, there is an edge from the start cell to the fourth cell in the first row since the value of the start cell is 3). You can use a forward breadth-first search (from the start cell) to figure out the lengths of shortest paths from the start cell to every cell in the puzzle. This identifies all reachable cells from the start cell, as well as the length of the shortest solution to the puzzle. To identify all reaching cells, you can use a backward breadth-first search (from the goal cell), using reversed edges (for instance, in the example above, there is a reversed edge from the fourth cell in the first row to the start cell). Once you identify the sets of reaching and reachable cells, you can identify the black and white holes and the forced forward and backward moves.
- Figuring out whether there is a unique shortest solution to the puzzle can be a bit tricky. You can modify your (forward) breadth-first search to identify which reachable cells have unique shortest paths from the start cell. A cell s has a unique shortest path from the start cell if a) its parent is unique (that is, any cell that has a legal move to s has a larger g-value than the parent of s), and b) its parent has a unique shortest path from the start cell.

- If you are not able to figure out how to evaluate some metrics of the puzzle (for example, whether there is a unique solution, the number of black holes etc.), you can leave it out of your scoring function and try to find the best solution with your partial scoring function. We will still grade your solutions with our own scoring function, but your puzzle's score might be good enough to exceed the threshold (although you will still lose points for not calculating all metrics).

- Once you have implemented your evaluation function, you can experiment with different local search methods (and different parameters), and use the one that gives you the best results within one minute. Also try to experiment with puzzles of different sizes and value constraints. If you find that different methods perform better at generating different types of puzzles, you can let your program decide which method to call, based on the given command line arguments. Alternatively, you can try different search strategies within the given time limit (for instance 30 seconds for hill climbing and 30 seconds for simulated annealing) and output the best puzzle generated.

- When using local search in class, we usually start with a valid initial configuration and try to improve it. In this project, it can be hard to randomly generate an initial puzzle that is solvable (especially so, since the values for cells are restricted). You can instead try to generate a random puzzle that is not necessarily solvable, and penalize the score of puzzles that are unsolvable with a very large offset. This would make your program prefer solvable puzzles to unsolvable ones. Alternatively, you can keep randomly generating puzzles until you generate a solvable one, but you also need to make sure that you don't make your puzzle unsolvable during search.

Submission

For the theoretical questions, you can either submit you solutions in PDF via blackboard or in hard copy to me or Sven (on time). You should include your name and student ID in your submission.

For the programming part, you need to submit a 'tar' file containing your source code and a readme file explaining your method of generating a good puzzle. You should also put your name and student ID as comments at the beginning of your `main.cpp` file.