COMP20003 Algorithms and Data Structures
Second (Spring) Semester 2017

[Assignment 2]
Solving the FLOW FREE Game: Graph Search

Handed out: Monday, 3 of October
Due: 12:00 Noon, Friday, 20 of October

Purpose

The purpose of this assignment is for you to:

- Increase your proficiency in C programming, your dexterity with dynamic memory allocation and your understanding of data structures, through programming a search algorithm over Graphs.
- Gain experience with applications of graphs and graph algorithms to solving games, one form of artificial intelligence.

Assignment description

In this programming assignment you'll be expected to build a *solver* for the Flow Free game. The game is yet another proof that William Rowan Hamilton was ahead of his time: not only Candy Crash is addictive and NP-complete, Flow free is another instance of a NP-complete game being wildly successful and addictive. You can play the game for free in your android or apple phone https://en.wikipedia.org/wiki/Flow_Free or solve it compiling the code given to you.

The Flow Free game

The game presents a grid of squares with colored dots occupying some of the squares. The objective is to connect dots of the same color by drawing 'pipes' between them such that the **entire grid** is occupied by pipes. However, pipes **may not intersect**. Difficulty is determined by the size of the grid, ranging from 5x5 to 15x15 squares.

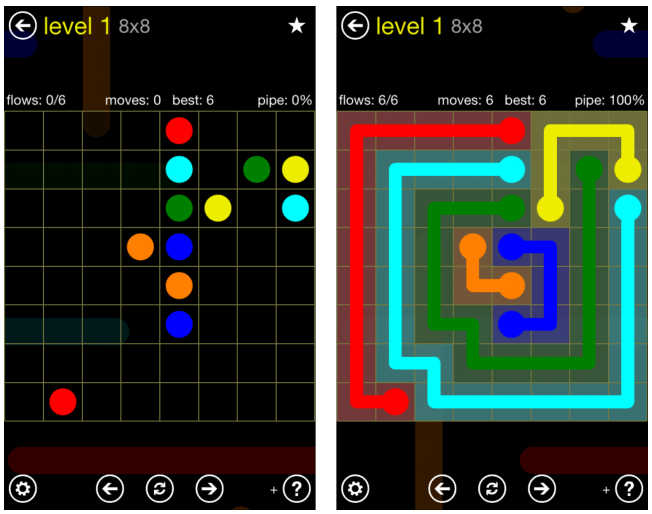

Figure 1: An initial configuration and solution of an 8x8 grid

```
GraphSearch(Graph, start)
 1  node ← start
 2  pq ← priority Queue Containing start node Only
 3  while frontier ≠ empty
 4  do
 5      node ← pq.pop()
 6      for each next color in ordering
 7      do
 8          for each move direction
 9          do
10              if move direction is valid
11                then
12                      newNode ← applyMoveDirection(node)
13
14              if newNode is a dead − end
15                then
16                      deleteNewNode
17                      continue
18
19              if newNode is thesolution
20                then
21                      solution = NewNode
22                      break
23
24              pq.push(newNode)
25
26  free priority queue
27  return bestAction
```

Figure 2: variant of Dijkstra for Flow Free

The Algorithm

Each possible configuration of the Flow Free grid is called a *state*. Each position in the grid can be free or contain a color. The Flow Free Graph $G = \langle V, E \rangle$ is implicitly defined. The vertex set V is defined as all the possible grid configurations (states), and the edges E connecting two vertexes are defined by the legal movements (right, left, up, down) for each color. The code provided makes sure that only legal moves are generated. A move is legal only if it extends a color pipe. A color cannot be placed if it's not adjacent to the pipe of the same color. Furthermore, to decrease the number of edges in the Graph, the pipe can only be started from one of the two initial colors, making one of them the start and the other the goal of the pipe.

Your task is to find the path leading from the initial configuration to the goal configuration. A path is a sequence of movements of pipes. You are going to use Dijkstra to explore all the possiblities and to find the solution path.

The initial node (vertex) corresponds to the initial configuration, along with an ordering of which is the next color to try. Therefore, every node has at most 4 children, and the next node along the path will generate the children for the next color. Different color orderings can be tried with the command line options (type `./flow -h` to see all options). Different orderings have different impact on the

number of nodes until you find a solution.

By default the search will expand maximum 1GB of nodes. This is checked by a variable named `max_nodes`

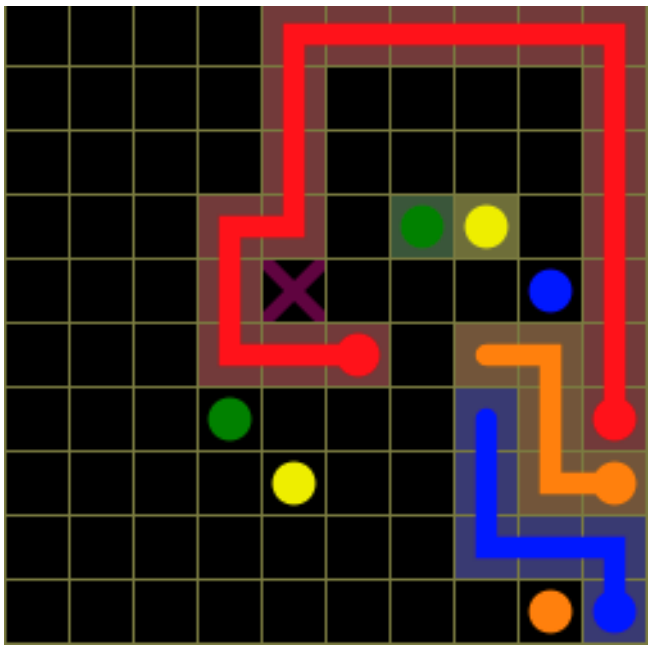

Figure 3: A dead-end configuration

When you *applyMoveDirection* you have to create a new node, that points to the parent, updates the grid resulting from applying the direction chosen, updates the priority of the node, and updates any other auxiliary data in the node. The priority of the node is given by the length of the path from the root node, i.e. how many grid cells have been painted. Check the file `node.h` as this function is already given.

A dead-end is a configuration for which you know a solution cannot exist. Figure 3 shows an example of a dead-end, as the cell marked with X cannot be filled with any color, and hence makes the problem unsolvable and we do not need to continue searching any of its successors. You can recognize dead-end cells by looking for a free cell in the grid that is *surrounded by three completed path segments (colored) or walls*. The *current position and goal position have to be treated as free space.*

Detecting unsolvable states early can prevent lots of unnecessary search. The search will eventually find a solution. If we recognize dead-ends and not generate their successors (pruning), the search for a solution will likely take less time and memory, as you'll avoid exploring all possible successors of an unsolvable state.

Deliverables, evaluation and delivery rules

Deliverable 1 – *Solver* source code

You are expected to hand in the source code for your solver, written in C. Obviously, your source code is expected to compile and execute flawlessly using the following makefile command:

```
make
```

generating an executable called `flow`. Remember to compile using the optimization flag `gcc -O3`

for doing your experiments, it will run twice faster than compiling with the debugging flag `gcc -g`. The makefile provided compiles with the optimization flag by default, and with the debugging flag if you type `make debug=1`.

Your implementation should be able to solve the *regular* puzzles provided. To solve the *extreme* puzzles you'll need further enhancements that go beyond the time for this assignment.

Base Code

You are given a base code (adapted from https://github.com/mzucker/flow_solver). You can compile the code and execute the solver by typing `./flow <puzzleName>`. You are going to have to program your solver in the file search.c. Look at the file and implement the missing part in the function called *game_dijkstra_search*. Once you implement the search algorithm, go to the file called *extensions.c* and implement the function called *game_check_deadends*

You are given the structure of a node *node.**, and also a priority queue *queues.** implementation. Look into the *engine.** and *utils.** files to know about the functions you can call to perform the search.

You are free to change any file.

Input

In order to execute your solver use the following command:
`./flow [options] <puzzleName1> ... <puzzleNameN>`

for example:
`./flow puzzles/regular_5x5_01.txt`

Will run the solver for the regular 5 by 5 puzzle, and report if the search was successful, the number of nodes generated and the time taken. if you use flag `-q` it will report the solutions more concisely. This option can be useful if you want to run several puzzles at once and study its performance.

If you append the option `-A` it will animate the solution found. If you append the option `-d` it will use the dead-end detection mechanism that you implemented. Feel free to explore the impact of the other options, specifically the ordering in which the colors are explored. By default, the color that has less free neighbors (most constrained), is the one that is going to be considered first.

Output

Your solver will print the following information if option -q is used:

1. Puzzle Name
2. SearchFlag (see utils.c, line 65-68 to undestand the flags)
3. Total Search Time, in seconds.
4. Number of generated nodes.
5. A final Summary

For example, the output of your solver `./flow -q ../puzzles/regular_*` could be:

```
../puzzles/regular_5x5_01.txt s 0.000 18
../puzzles/regular_6x6_01.txt s 0.000 283
```

```
../puzzles/regular_7x7_01.txt s 0.002 3,317
../puzzles/regular_8x8_01.txt s 0.284 409,726
../puzzles/regular_9x9_01.txt s 0.417 587,332

5 total s 0.704 1,000,676
```

These numbers depend on your implementation of the search, the ordering you use, and whether you prune dead-ends. If we use dead-end pruning we get the following results `./flow -q -d ../puzzles/regular_*` :

```
../puzzles/regular_5x5_01.txt s 0.000 17
../puzzles/regular_6x6_01.txt s 0.000 254
../puzzles/regular_7x7_01.txt s 0.001 2,198
../puzzles/regular_8x8_01.txt s 0.137 182,136
../puzzles/regular_9x9_01.txt s 0.210 279,287

5 total s 0.349 463,892
```

Remember that in order to get full marks, your solver has to solve at least the regular puzzles.

Evaluation

Assignment marks will be divided into three different components.

1. Solver (10)
2. Dead-End Detection (4)
3. Code Style (1)

Please note that you should be ready to answer any question we might have on the details of your assignment solution by e–mail, or even attending a brief interview with me, in order to clarify any doubts we might have.

Code Style

You can improve the base code according to the guidelines given in the first assignments and in LMS. Feel free to add comments wherever you find convenient.

Delivery rules

You will need to make *one* submission for this assignment:

- Your C code files (including your `Makefile`) will be submitted through the LMS page for this subject: *Assignments → Assignment 2 → Assignment 2: Code*.

Program files submitted (Code)

Submit the program files for your assignment and your `Makefile`.

Your programs *must* compile and run correctly on the CIS machines. You may have developed your program in another environment, but it still *must* run on the department machines at submission time. For this reason, and because there are often small, but significant, differences between compilers, it is suggested that if you are working in a different environment, you upload and test your code on the

department machines at reasonably frequent intervals.

A common reason for programs not to compile is that a file has been inadvertently omitted from the submission. Please check your submission, and resubmit all files if necessary.

Plagiarism

This is an individual assignment. The work must be your own.

While you may discuss your program development, coding problems and experimentation with your classmates, you must not share files, as this is considered plagiarism.

"Borrowing" of someone else's code without acknowledgement is plagiarism, e.x. taking code from a book without acknowledgement. Plagiarism is considered a serious offense at the University of Melbourne. You should read the University code on Academic honesty and details on plagiarism. Make sure you are not plagiarizing, intentionally or unintentionally.

You are also advised that there will be a C programming component (on paper, not on a computer) on the final examination. Students who do not program their own assignments will be at a disadvantage for this part of the examination.

Administrative issues

When is late? What do I do if I am late? The due date and time are printed on the front of this document. The lateness policy is on the handout provided at the first lecture and also available on the subject LMS page. If you decide to make a late submission, you should send an email directly to the lecturer as soon as possible and he will provide instructions for making a late submission.

What are the marks and the marking criteria Recall that this project is worth 15% of your final score. There is also a hurdle requirement: you must earn at least 15 marks out of a subtotal of 30 for the projects to pass this subject.

Finally Despite all these stern words, **we are here to help!** There is information about getting help in this subject on the LMS pages. Frequently asked questions about the project will be answered in the LMS discussion group.

Have Fun!

N.L.