CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean
Webpage: `http://www.cs.clemson.edu/~bcdean/`
Handout 15: Lab #9

Fall 2018
TTh 12:30-1:45
Earle 100

1 Finding Paths in Graphs

In today's lab we will build on the code developed in lecture for performing a depth-first or breadth-first search in a graph (the code is currently set to perform a breadth-first search, which has the advantage of finding shortest paths, although recall that it is easy to modify by substituting a stack for a queue if depth-first search is desired instead). Our goal is to illustrate how this algorithm can execute in a generic fashion on graphs representing many different problems, without changing the code for the core algorithm at all. Only the graph needs to change. In the following directory:

`/group/course/cpsc212/f18/lab09/`

you will find code (`ladder_generic.cpp`) for a generic version of the "word ladder" problem we solved in lecture. Please make two copies of this file. You will then modify these to solve two fun puzzles that are well known in the world of recreational mathematics.

2 Puzzle 1: The Wolf, Goat, and Cabbage

You are standing with a wolf, goat, and cabbage, next to a river, and you would like to transport all three of these items to the other side. However, you only have access to a small boat that can fit at most one item (besides yourself). If you leave the wolf and goat alone unattended, bad things happen to the goat. If you leave the goat and cabbage unattended, bad things happen to the cabbage.

For this puzzle, you will have $2^4 = 16$ states (nodes), since each of the 4 objects in question (yourself, the wolf, the goat, and the cabbage) can be either on the left side of the river or the right side. A natural way to represent a node is using an integer in the range $0 \ldots 15$, where each of the integer's 4 bits represents one object (0 for left side, and 1 for right side). One could also potentially use a length-4 string of zeros and ones. The downside of using integers is the need for bitwise manipulation (e.g., looking at specific bits), but on the other hand it's much easier to enumerate through integers in $0 \ldots 15$ than length-4 binary strings.

When you print out the sequence of nodes in a solution, each node should be printed on a single line in a human-readable format such as this:

```
wolf cabbage |river| you goat
```

3 Puzzle 2: Water Jugs

You are standing next to a river with a very content-looking wolf, a head of cabbage, and two water jugs, which have integer sizes $A = 3$ and $B = 4$. In order to boil the cabbage for your dinner, you would like to measure out exactly 5 units of water.

To solve this problem, you should use a search through a graph where each node corresponds to a pair of integers (a, b), indicating that you are in the state where jug 1 contains a units of water and jug 2 contains b units of water. You want to start from the state $(0, 0)$ where both jugs are empty, and your goal is to reach a state (a, b) with $a + b = 5$. There are 3 possible actions you can take to move between states: filling one of the jugs to its capacity, emptying out one of the jugs, or pouring the contents of one jug into another (until the first becomes empty or the second reaches capacity).

4 Extensions

Your solutions to the puzzles above should print out the sequence of nodes along a shortest path representing a solution. To make these even more intelligible, please modify the code (for both problems) so that each edge stores a textual description of what the state transition of the edge represents (e.g., "Fill jug 2", or "Take the goat across the river"). Using this, you can then print out a transcript of the solution that is much easier to follow, including edge descriptors as well as node descriptors. For example, the output of the water jugs problem might now look like this:

```
Initial state : a=0, b=0
Fill jug 2 : a=0, b=4
Pour 2->1 : a=3, b=1
Empty jug 1 : a=0, b=1
Pour 2->1 : a=1, b=0
Fill jug 2 : a=1, b=4
```

Note that you will only need to change the code for printing a path for this part, not the code for searching. A suggestion for the structure to use for storing edge descriptors might be:

```
map<pair<state,state>,string> edge_label;
```

With this sort of structure, you could say something like

```
cout << edge_label[make_pair(a,b)]
```

to print out the label of an edge from node a to node b.

5 If You Get Bored...

It turns out there is a bug in the `diameter.cpp` code that was written in class to find the pair of words between which there is the longest ladder. This code is also included in the lab directory above. Can you figure out what is the bug? This is not part of the graded part of this lab, although it will earn you the respect and admiration of your TAs.

6 Submission and Grading

Please name your submitted files `wgc.cpp` and `jugs.cpp`.

For this lab, you will receive 8 points for correctness and 2 points for having well-organized, readable code. Zero points will be awarded for code that does not compile, so make sure your code compiles on the lab machines before submitting!

Final submissions are due by 11:59pm on the evening of Friday, November 9. No late submissions will be accepted.