

# Lab 10: Using Breadth-First Search to Solve Puzzles

---

**Due** Monday by 11:59pm

**Points** 100

**Submitting** an external tool

---

In today's lab we will build on code written to perform breadth-first search on a graph. An advantage of using breadth-first search is that the shortest path through the (unweighted) graph is found. 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. Also note that if depth-first search is to be performed instead the code may be modified by using a stack instead of a queue.

Download the starter code which contains two .cpp files. You will modify these by implementing the **build\_graph()** function to solve two fun puzzles that are well known in the world of recreational mathematics.

## 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 ... 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 ... 15 than length-4 binary strings. The representation of a particular state (which is already included in the code) is each item is assigned a bit position (#of bits left of rightmost bit in integer value): 0 for wolf, 1 for goat, 2 for cabbage and 3 for yourself; a value of 0 indicates the item is to the left of the river, 1 indicates the item is to the right. The starting state is 0 (0000): all items to left and the ending state is 15 (1111): all items to right.

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

**wolf cabbage |river| goat you**

Additionally the action to reach each state will be displayed before the state of the node. The starting node is the initial state so there is no action to reach this node so "Initial state: " is displayed before the state instead of an action. Subsequent states will display an action before them, the string for this action to be displayed is generated by **state\_string()** and will be stored in the **edge\_label** map as the value from a source,target state pair with a particular action. As a result the first part of the solution should print as follows (must match exactly):

**Initial state: wolf goat cabbage you |river|**

**Cross with goat: wolf cabbage |river| goat you**

...

To facilitate this the **state\_string()**, **print\_path()**, and **neighbor\_label()** functions are provided. The representation of a state of the puzzle is provided as well as the **main()** function. You will be implementing the **build\_graph()** function and you may also write any additional helper functions, modifying **wolfGoatCabbage.cpp**.

## 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 for both jugs 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).

As with the previous puzzle the representation of a state is provided, this time being a pair of integers, the first being the units of water in jug A and the second being the units of water in jug B. The state of each node will be displayed in the following way:

**[0,4]**

And like in puzzle 1 the action to reach each state will be displayed before the state of the node.

The starting node is the initial state so there is no action to reach this node so "Initial state: " is displayed before the state instead of an action. Subsequent states will display an action before them, the string for this action to be displayed will be the appropriate string from **const string actions[]** and will be stored in the **edge\_label** map as the value from a source,target state pair with a particular action. As a result the first part of the solution should print as follows (must match exactly):


**Initial state: [0,0]**

**Fill B: [0,4]**

...

To facilitate this the **print\_path()** function is provided along with the strings for each action that changes to another state for **edge\_label**. You will implement the **build\_graph()** function and you may add additional helper functions, this time modifying **waterJugs.cpp**.

[lab 10 slides.pptx \(https://clemons.instructure.com/courses/195558/files/18830655?wrap=1\)](https://clemons.instructure.com/courses/195558/files/18830655?wrap=1)   
([https://clemons.instructure.com/courses/195558/files/18830655/download?download\\_frd=1](https://clemons.instructure.com/courses/195558/files/18830655/download?download_frd=1))

[Lab 10 - Using Breadth-First Search to Solve Puzzles Starter Code.zip](https://clemons.instructure.com/courses/195558/files/18830656?wrap=1)  
(<https://clemons.instructure.com/courses/195558/files/18830656?wrap=1>)   
([https://clemons.instructure.com/courses/195558/files/18830656/download?download\\_frd=1](https://clemons.instructure.com/courses/195558/files/18830656/download?download_frd=1))

The main function provided in the starter code of each .cpp source file can be used to test your implementation for solving each puzzle. **When submitting wolfGoatCabbage.cpp and waterJugs.cpp to Gradescope for this assignment be sure to comment out the main() function in both source files.** There are two test cases, one for wolfGoatCabbage.cpp and one for waterJugs.cpp containing the original unmodified main() function from the starter code version of the respective source file that will be used by Gradescope. Gradescope will compare the output generated by your implemented version with the output from a solution implementation, so the output (including spacing) must match exactly.

This tool needs to be loaded in a new browser window

Load Lab 10: Using Breadth-First Search to Solve Puzzles in a new window

