

# CSC172 Lab: State Space Search

November 18, 2014

## 1 Introduction

The labs in CSC172 follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at [http://en.wikipedia.org/wiki/Pair\\_programming](http://en.wikipedia.org/wiki/Pair_programming) .

Every student must hand in his own work, but every student must list the name of the lab partner (if any) on all labs.

You and your partner(s) should switch off typing, as explained by your lab TA. As one person types, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook doesn't mention this technique. So, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

## 2 Jug Puzzles

There's a large class of jug and water puzzles: There are jugs containing water, sometimes a faucet, we can pour jug to jug, fill or dump a jug: usually (maybe for all such puzzles?) a pour empties one jug or fills another.

Two examples from many at:

<http://www.creativepuzzels.com/spel/speel1/speel2/water2.htm>

1. An eight gallon container is full and both a three gallon container and a five gallon container are empty. Without using any other containers, divide the water into two equal amounts.

12. You have a faucet that delivers an unlimited amount of water. You have two empty containers, one of 5 liters, the other of 3 liters. How many operations are required to fill one of the containers with exactly 4 liters of water?

### 3 State Space Search

A *state* represents the values of relevant variables characterizing a puzzle configuration at any time. For us, that's the number of units of liquid in each jug. The *state space* is all possible states.

An *Operation* generally changes the state of the puzzle. For us, that's pouring one jug into another. Commonsense assumptions keep us in the spirit of the puzzle: for example, in an operation, the "from" jug must be completely emptied or only enough poured out to fill the "to" jug completely, else we lose track of the size of jug contents.

The search begins in a *start state* and performs operations in a more or less (for us today, less) smart way to achieve a *goal state* – one that meets the requirements of a successful solution. It creates a *search tree* of *nodes*, which contain a state and other search-related information (more below). How to search is a big question in AI: we'll use simple, "uninformed" depth-first search (Section 10.)

### 4 State Representation

Clever representation can save much trouble in search problems. Is the state for a 2-jug puzzle two variables like Jug1 and Jug2? How many operations are there and what are their names? If we add a jug or faucet to the puzzle, do we have to rewrite the whole program with new names for states and operations? Here's a general representation.

For N jugs and a faucet (and a drain for dumping water into):

State is an array indexed 0 to N; element state[i] contains contents of jug number i. The jug indexed by 0 we arbitrarily say contains 1 unit, but acts as if it had infinite contents and can accept more: it is both the 'source' (faucet) and 'sink' (if either needed). We'll need an N+1 -long array C of jug capacities: C[0] = 1 arbitrarily.

Use the same representation for N jugs and no faucet; just don't ever use the 0 indices of the states and C.

E.g., start state for problem 1 above: C is [1,3,5,8], start-state is [0,0,0,8]. A goal state is any state with 4 gallons in each of 2 jugs. Index 0 unused since water is conserved.

Start state for problem 12: C is [1,5,3], A is [1, 0,0]. Here Jug 0 is faucet (source) and drain (sink). A goal state is any state with 4 litres in any jug.

## 5 Node Representation

A (search tree) *node* is a class with three fields:  
state (an N+1 long array for N jugs)  
operator (2-long array: the op. that produced this state)  
parent (the node just above this one in the tree)

## 6 Operator Syntax, Semantics

My idea for operator notation is simply a pair [from, to] of jug indices. So, filling jug 3 from the source is [0,3], emptying 1 is [1,0], pouring from 2 to 3 is [2,3].

An [i,j] operation may not “make sense” (make any progress) anytime, or may be useless in the current state. Here’s some pseudocode for a bad\_op(op, state) function that returns TRUE if op doesn’t make sense in state. (There may be more cases but I couldn’t think of any). I’ve made C a global.

```
boolean bad_op ( [i,j], state[]) // op and state array inputs\\
(( i==j ) OR           // don't pour into self
 (state[i] == 0) OR    // don't pour from empty jug
 (state[j] == C[j])); // nor into full one
```

Here’s pseudocode to apply an operator to a state to get new state: make sure you believe it, or fix it!).

```
New-A apply_op ([i,j] state[])
[0,i] sets state[i] = C[i];
[i,0] sets state[i] = 0; // manage source and sink

[i,j]: sets state[i] = max( 0, state[i] - (C[j] - state[j]));
      // empty i or fill j or both)
Also (i,j): sets state[j] = min( C[j], (state[j] + state[i]));
```

I think...check!

## 7 Avoiding Loops

We don't want to revisit a state (gets us in a loop). Hash table natural to mark and check visited states. In lisp (and Perl, say, probably Java too), just one statement each to create table, mark, check: eg below easily understood even for non-lispers.

```
(setf visited (make-hash-table :test #'equal'))
(defun visited (state)
  (gethash state visited))
(defun mark-visited (state)
  (setf (gethash state visited) 1))
```

## 8 isGoal

The goal is not usually just one state. E.g. "get exactly six quarts in the largest jug" ignores contents of other jugs. Both our example problems have several possible goal states (Section 4), and for each problem we'll need a predicate `isGoal(state)` to test whether we've succeeded.

## 9 Globals, Utilities

N (number of jugs)  
MaxLevel (allowed depth of search) – see below  
C (N+1 array of capacities of jugs)  
StartState (N+1 array)  
Visited (hash-table)

`isGoal()`, `mark-visited()`, `visited()` `bad_op()`, `apply_op()`.

## 10 The Search Algorithm

You know depth first search. Its space requirements often assume that all possible successor states are calculated (and saved, then explored) at each node. *Backtracking* seems to mean DFS that generates new states to explore one at a time. This process can be as simple minded as a for loop or two, or really complicated (which of 30 chess moves should be tried first (the clock is ticking...)). I've done Jugs both ways, but the latter seems easiest here.

Depth-first search can go on forever so it must be governed. A common technique is *depth-limited search*: search down to some maximum level, where we give up along that branch and backtrack. If no solution is found, the maximum level can be increased either

manually or automatically by the program. This *iterative deepening* redoes the whole previous search but that's not so bad since even in a binary tree, half the nodes are at the lowest level (cute, eh?).

**Driver:**

```
{ Initialize Globals;
  start-node = new node(StartState, [0,0], null);
  stateSearch(start-node, 0); // *start* at level 0.
}
```

**Search:**

```
{ stateSearch(node, Level)

  if isGoal(node.state)
    { announce success;
      from this level, follow parent pointers
      back to top and print out ops along path;

      return; // up a level to continue search:
              // any more answers?
    }

  if (Level >= MaxLevel) return;

  for i = 0, N (or 1, N)
    for j = 0, N (or 1, N)
      { if visited(node.state) continue;
        if bad_op([i,j], node.state) continue;
        new_state = apply_op([i,j], node.state);
        new_node = new node(new_state, [i,j], node);
        mark_visited (new_node.state); %% is this OK?
        stateSearch(new_node, Level+1);
      }
}
```

Code Not guaranteed! Easiest to print path backwards.

## 11 Experiments

Debugging: may want to print out information like ops, states, level for small problems to assure things working.

Compare answers with those at the creativepuzzle link above. Are your solutions unique? Are some shorter than others? Can you easily add tests to avoid wasted search if that's what caused longer solutions? Make up some puzzles of your own... There's a way to tell if a solution exists: saw it somewhere on web under "jug puzzles" I think.

Are our representations of states and operation general enough for all the puzzles on the puzzle link?

## 12 Grading

172/grading.html

Grading will be holistic, based on process (coding and debugging) followed, discoveries made (like misprints or thinkos in the pseudocode), results and experiments. (README file counts for 10%)