**COM1005 Machines and Intelligence**

**Week 1 Practical session: Jugs Problem**

Heidi Christensen (partially based on notes by Phil Green)

The aim of this practical is to explore the Jugs problem (as introduced in the lectures) and to try out the search1 code which is a specific implementation of a general search engine.

After you’ve worked through this lab sheet you should have a good idea of how the general search engine code works and how you might tailor it to a particular search problem.

Feel free to work with a friend, but make sure that you write/modify your own code and understand the material for yourself. If you experience any problems, then ask a Demonstrator or Lecturer to help you – you will have an opportunity to do this during Thursday’s synchronous session or on the Blackboard lab class forum.

**Part 1: The Jug Problem: state space**

Let’s remind ourselves what the Jugs Problem is:

| A typical jugs problem is:   * Suppose you have a 7-pint jug, a 4-pint jug, a water tap and a sink. * You are allowed to   + fill either jug from the tap,   + empty either jug into the sink, or   + pour from one jug into the other until either the first jug is full or the second jug is empty.   Aim: show that by a sequence of such moves it is possible to measure any number of pints between 0 and 7. |
| --- |

We consider that the problem involves moving between a number of problem **states.**

A **State Space** contains all allowed problem states. It consists of a **graph** in which states are connected by **arcs** (legal moves from one state to another).

For the Jugs Problem, the State Space contains all possible configurations that can be reached from the initial state where both jugs are empty. If we represent each State as the tuple containing

( jug1Content, jug2Content)

That is, the content of jug1 and the content of jug2, we can illustrate an *incomplete* State Space as follows:

Diagram, schematic

Description automatically generated

In general, a State is a data structure containing everything needed to define a position in the problem. The StateSpace is a graph with a node for each state and links (arcs) connecting the nodes. For a node n there is a link to all nodes which are successors of n (successors of n can be reached by a single move).

The exploration process is called State Space Search or Graph Traversal.

The expandfunction takes a given state and returns a list of all the states that can be reached by a single move from that state. For the Jugs Problem, this mean all possible configurations that can be reached by one move from a given state.

Sometimes it is natural to think of the expandfunction as invoking operators**.** An operator is a function which embodies a 'legal move' from one state to another. For the Jugs Problem these operators would be something like:

* empty jug1 to sink
* empty jug2 to sink
* fill jug1 from tap
* fill jug2 from tap
* pour from jug1 to jug2
* pour from jug2 to jug1

Not all operators will be applicable to a given state.

⇒For each of the operators above, write down a state where you could use it, and a state where you couldn’t use it. Fill out the table below with your answers.

| Operator | State where this could be applied | State where this couldn’t be applied |
| --- | --- | --- |
| empty jug1 to sink | jug1 has any amount of liquid in it | jug1 is empty |
| empty jug2 to sink | jug2 has any amount of liquid in it | jug2 is empty |
| fill jug1 from tap | jug1 is empty | jug1 is full |
| fill jug2 from tap | jug2 is emptpy | jug2 is full |
| pour from jug1 to jug2 | jug2 has less water than jug1 | jug2 has more water than jug1 |
| pour from jug2 to jug1 | jug1 has less water than jug2 | jug1 has more water than jug2 |

The operators/expand-function represent the 'rules of the game'. The process of considering all the successors of a given state is called **developing** the state.

A **problem** is seen as finding a **path** through state space from a given **initial state** to some **goal state.**

The **solution** can be expressed either as the sequence of states traversed, the sequence of operators used or the goal state, as appropriate.

There may be **one or many goal states.** In general, we need a **goal test** that, given a state, tells us whether it is a goal or not.

It may be sufficient to find any solution, or we may be required to find the '**best**' solution. For the Jugs Problem, we have multiple goal states or *targets* (“Aim: show that by a sequence of such moves it is possible to measure any number of pints between 0 and 7”).

**States and Nodes**

Each node in the search tree will be associated with a different state in state space, but may also contain other information (for instance, a pointer to the parent node):

Diagram, schematic

Description automatically generated

Jugs search target=3. Solution path in bold.

**Part 2: Programming a state space search algorithm**

In order to programme we will need

* A way **of representing the states** (**SearchState** class).
* An **expand()** method**,** perhaps calling operators**.**
* A method acting as a goal predicate,(**goalPredicate**())
* A method implementing a strategy for navigating the State Space, effectively picking the next node to expand from**.**

**A Skeleton Algorithm for State Space Searching**

| **Expressed imperatively:**  We maintain   * a list of nodes awaiting development. This is called the **open** list or agenda**.** * a list of nodes already developed. This is called the **closed** list**.**   Initially,   * we put the start node on **open.** * **closed** is empty   We repeatedly   * Check if **open** is empty. If so, exit with failure * Use the search strategy to select (**selectNode()**) some node (the **currentNode**) from openfor development. * Remove **currentNode** from **open.** Add it to **closed.** * Check if **currentNode** is a goal - if so, exit with success. * use expand to find **currentNode’s** successors, * add all successors which are not already on **open** or **closed** to **open.** |
| --- |

We will develop a generalised search engine in Java, which we can then use in any search problem domain. The user has to set up what is necessary for their domain: we specify below how to do this.

*This policy of abstracting the problem-solver is typical of AI (and Computer Science in general). It saves repeated work & forces us to understand what we’re doing.*

⇒Find the code for week 3’s lab class. (search1.zip) file. This is the basic version of the search engine. Open **Search.java** and find where the above algorithm is implemented. This is the *core* of the search engine. Look through the different lines and try to understand what happens at each stage.

**The General Search Engine**

* We need a class which contains a method **runSearch** implementing the algorithm above. We’ll call this class **Search**.
* **runSearch** uses **open** and **closed**, so these will be variables in **Search**.
* **runSearch** deals with search nodes, so we’ll have another class **SearchNode**.
* We’ll need to keep track of the **currentNode**, so this will be another variable in **Search**.
* To programme a search in a particular problem domain (e.g. Jugs problems), our ‘user’ needs to create a subclass of ∫ for this domain – e.g. **JugsSearch**.

⇒open **JugsSearch.java** in the code and note how this class extends the methods given in the abstract class in **Search.java**.

If you ever needed to develop a search engine for a different problem, you would need to do something similar.

Diagram

Description automatically generated

* *This subclass will have variables carrying the information needed to define a search problem in this domain* – for jugs, the jug capacities and the target amount. These are the **problem parameters.**
* To run a search, we create an instance of the subclass, giving it the defining information, and call **runSearch**.
* So we will never make an instance of Search itself – it will be a *parent* class*.*

**The code to support the Search Tree**

* The class **SearchNode** makes the distinction between a node in a search and a problem state clear: the user is concerned with states. The search engine sees only nodes.

⇒remind yourself what the difference between a State Space and a Search Tree is and complete the sentences below with the description

A State Space describes: Shows the flow of states

A Search Tree describes: A tree constructed from the search algorithm.

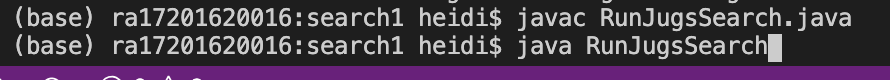
⇒open **JugsState.java** and see what has been added to specialise this for the Jugs Problem.

You should now be familiar with the code and how the Jugs Problem specific classes have inherited from the general search engine classes and have been specialised for the domain.

**Part 3: Exploring the Search Tree**

It’s time to compile and run the code.

⇒open **RunJugsSearch.java** and see how it makes an instance of **JugsSearch** and initiates it. Now compile the code and run it.

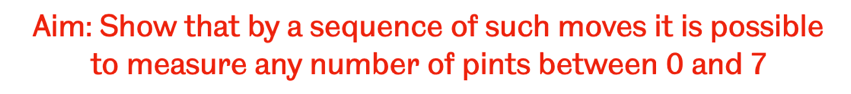
****

Look at the output and draw out the search tree that the algorithm went through. Make sure you understand what was on the **open** list (and why). Why did the search engine stop when it did?

**Part 4: Solving the Jug Problem**

Now, let’s remind ourselves what the aim of the Jugs Problem was :

Icon

Description automatically generated

⇒copy **RunJugsSearch.java** to something like **RunJugsSearch2.java** and add code that allows you to provide evidence to demonstrate that we can solve the Jugs Problem for c1=7 and c2=4 for all targets {0, …, 7}

⇒Choose different parameters for c1 and c2. Modify the code so that you can keep track of the average number of iterations across the different target values. What does that mean in terms of efficiency?

|  | C1 | C2 | How many iterations on average? |
| --- | --- | --- | --- |
| Experiment 1 | 7 | 4 | 11 |
| Experiment 2 | 10 | 2 | 2 |
| Experiment 3 | 8 | 5 | 8 |
| Experiment 4 | 3 | 4 | 7 |

**Bonus Part: Unpicking the search strategy**

This code implements a very simple search strategy.

⇒Looking at the generated search trees, how do you think it works? Find in the code where this is implemented – how would you go about implementing something a bit more ‘intelligent’?

We will look at different search strategies and their advantages later on in the module.