4 Symbolic AI

One approach to creating artificial intelligence is based on creating a symbolic representation of the problem and solve it using logic and search. By *symbolic* we mean a human-understandable representation, for example through well defined states, rules, and actions to take. This general approach is known as *symbolic AI*. Once we have formulated a symbolic representation of a problem, we can solve it by using logic to deduce the logical consequences of the rules of the system, or by searching through the space of possibilities to find the most optimal solution.

4.1 Logic

Symbolic AI is based on the rules of logic. Many different formal systems of logic have been developed through the history of mathematics. Here we will consider the fundamental algebraic logic that deals with reasoning about binary variables.

4.1.1 Boolean algebra

In Boolean algebra (named after George Boole) variables can only take two values: true or false. Often the values are denoted by 1 (true) and 0 (false), but these values should not be confused with the usual integers 0 and 1. Our usual mathematical operations, such as addition and mulitiplication, do not really make sense for values which can only either be true or false. Instead, we have the three operations *and*, *or*, and *not*. Perhaps a little confusingly, the operations *and* and *or* are usually denoted by a multiplication and addition symbol, whereas the negation is often denoted by a line above the expression.

Definition 4.1 Boolean operators

$$
\begin{aligned}
\text{and:} \quad a \cdot b &= \begin{cases} 1 & a = b = 1 \\ 0 & \text{otherwise} \end{cases} \\
\text{or:} \quad a + b &= \begin{cases} 0 & a = b = 0 \\ 1 & \text{otherwise} \end{cases} \\
\text{not:} \quad \overline{a} &= \begin{cases} 1 & a = 0 \\ 0 & a = 1 \end{cases}
\end{aligned}
$$

Example 4.1 Truth table

The two Boolean variable a and b can each take two values: 0 or 1. Thus in total there are four different combinations of values that a and b can take. We can list these combinations in a table with four rows. In addition, we can also list the results of applying the functions *and* and *or* to the two variables. This results in the *truth table* show in Figure 4.1.

a	b	$a \cdot b$	$a + b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Figure 4.1: Truth table for the *and* and *or* operators.

The Boolean *and* and *or* operators follow almost the same algebraic rules as the usual multiplication and addition operators: The commutative, associative, and distributive laws hold as we are used to. On top of that there are a number of other laws of Boolean algebra, that can be used to manipulate and simplify Boolean expressions.

Definition 4.2 Properties of Boolean algebra

Commutative law	$a + b = b + a$
	$a \cdot b = b \cdot a$
Associative law	$a + (b + c) = (a + b) + c$
	$a \cdot (b \cdot c) = (a \cdot b) \cdot c$
Distributive law	$(a \cdot b) + (a \cdot c) = a \cdot (b + c)$
	$(a + b) \cdot (a + c) = a + (b \cdot c)$
Double negative law	$\overline{\overline{a}} = a$
Identities	$a + 0 = a, \quad a + 1 = 1$
	$a \cdot 1 = a, \quad a \cdot 0 = 0$
Complement law	$a + \overline{a} = 1$
	$a \cdot \overline{a} = 0$
Absorbtion law	$a + a \cdot b = a$
	$a + \overline{a} \cdot b = a + b$
DeMorgan's law	$\overline{a \cdot b} = \overline{a} + \overline{b}$
	$\overline{a + b} = \overline{a} \cdot \overline{b}$

4.1.2 *Boolean functions*

A Boolean function $f(x_1, x_2, \ldots, x_n)$ takes as input n of these true/false values and produces an output in the form of a single true/false value. Any such a function can be created by combining the three Boolean operators.

Example 4.2 Crossing the road with a Boolean function

Let us consider a Boolean function that describes the logic behind how an individual might decide whether or not to cross the road: This individual will walk if there is a green light and no traffic to be seen. In addition, she will also walk if there is no traffic and no police to be seen, regardless of the color of the traffic light. This behavior can be described by the function:

$$\text{walk} = (\text{green and not traffic}) \text{ or } (\text{not traffic and not police})$$

Using the Boolean notation introduced above, we might equally write

$$w = (g \cdot \overline{t}) + (\overline{t} \cdot \overline{p}),$$

where we now have used short-hand notation for each of the Boolean variables (w = walk etc.). We can simplify this expression by using the laws of Boolean algebra

$$w = \overline{t} \cdot (g + \overline{p}).$$

What would happen, if there is no traffic ($t = 0$), the light is not green ($g = 0$), and the police is present ($p = 1$)? We can simply insert these truth-values in the equation, and compute w as follows:

$$w = \overline{0} \cdot (0 + \overline{1}) = 1 \cdot (0 + 0) = 1 \cdot 0 = 0.$$

So in this situation our pedestrian would not walk ($w = 0$).

4.2 Rule based systems

According to our definition of intelligence, an intelligent system has the ability to learn and apply knowledge and skills to achieve goals. For now, let us set aside the question of how we learn and aquire knowledge and skills, and focus on the subproblem of how we can apply knowledge to solve problems. To answer this, we need to define more precisely what we mean by *knowledge*, and consider how we can represent that in a computer system.

One way to think of knowledge is to envision a human *expert* who posesses all the knowledge that we need to solve our task. If we had access to such an expert, we might be able to extract the knowledge, and distill it into some practical and operational format. This is the idea behind *expert systems*.

One way to represent knowledge is as a collection of *facts* and *rules*, which describe all the things that exist as well as their properties and relations. Of course, in practise we would need to limit the scope to handle only a limited number of facts and rules within some problem domain. In a classical expert system, the rules are specified through a (possibly large) number of *if-statements*. An if-statement contains two parts called the *condition* and the *consequent*.

if <condition>

then <consequent>

The condition is a logical statement, and if the condition is true the consequent is executed. In this case, we say that the rule has *fired*. If the condition is false (the rule does not fire), the consequent is not executed and the rule is not applied.

Example 4.3 Crossing the road with an expert system

Continuing Example 4.2, let let us define the possible observable facts, that we need to take into consideration:

light	The state of the traffic light can either be red, amber, or green.
road	The state of the road can be traffic or none.
surveillance	The state of surveillance can be police, or none.

With these three observables which can each take two or three different values, our system can be in $3 \cdot 2 \cdot 2 = 12$ different possible states. Now, let us define a set of expert rules inspired by the individual in Example 4.2.

if *light* is green and *road* is none
then *walk* is true

if *surveillance* is none and *road* is none
then *walk* is true

At first we might think that these two rules describe the same function as in Example 4.2, but they do not. The difference is that the expert rules are only used to positively define the cases that are described in the two conditions—all other situations are left undefined. In a rule based expert system, we need to define all the needed actions as rules. We might then add another rule, which says when to not cross the road:

if *road* is traffic
then *walk* is false

Even with that rule in place, there are still some situations that are left undefined. To get a better overview, we can list all the 12 different possibles states in a table, and write in the cases that are handled by our expert rules. If we do this, we get the following table for the *walk* variable:

	road	traffic		none	
	surveillance	police	none	police	none
light	red	0	0	?	1
	amber	0	0	?	1
	green	0	0	1	1

Here we see, that there are still two situations in which the result is undefined, namely when there is no traffic, there is police present, and the traffic light is not green.

4.2.1 *Resolving conflicts*

Once we start creating more complex expert systems, it becomes difficult to avoid conflicting rules. If we have two rules which fire at the same time, but which have conflicting consequences, we need some mechanism to decide which of the rules should have preference. It would not be very useful to require that all expert rules are free from conflict, since this would make the process of writing down the rules very cumbersome and unintuitive. Another idea is to use a simple conflict resolution mechanism. One such mechanism is to order the rules according to their priority. If two or more rules are in conflict, the rule with greatest priority wins, and the rest are ignored.

Example 4.4 Conflict resolution

Let us consider a simple system with just two rules

if *weather* is rain
then *recommend* is stay_home

if *weather* is sun or *clothes* is rain_coat
then *recommend* is go_out

Here we might have a conflict: If it rains and we are wearing a rain coat, both rules will fire. According to the first rule, the recommendation is to stay home, and according to the second rule, we should go out.

If we assume that the rules are listed in increasing order of priority (i.e. the second rule has priority over the first), then the conflict is resolved, and the recommendation is to go out in the rain with the rain coat on.

4.2.2 *Forward and backward chaining*

In the examples considered so far, the variables considered in the conditions have not been affected by any of the consequents. When that is the case, an expert system can simply be implemented by running each rule that applies and observing the outcome. However, if some of the consequents influence the conditions of other rules, the situation is more complicated.

We can distinguish between two modes of reasoning in a rule based system, called forward and backward chaining.

Forward chaining is resoning from the data: The rules are applied from the top, and once the first applicable rule fires its consequent generates a new piece of information which might influence which rules are applicable. Then the rules are applied again from the top, and again the top most rule that applies is fired. Each rule is allowed to fire only once, and the process continues until no more rules can fire.

Backward chaining is reasoning aimed at proving an outcome: First, the set of rules are searched to find the rules that are able to generate the outcome we are interested in. Such rules must have the desired outcome in their consequent. If there exists one or more such rules, we examine each of them in turn. We look at the condition, and if it is true, our outcome is proven. Otherwise we repeat the process by looking for rules that have these conditions as consequents and continue the recursion backward. Once there are no more rules to consider, we have either proven or not proven our outcome.

Example 4.5 Walk or drive to work

Let us consider an expert system that helps decide if we should walk or drive to work. The observable facts related to our car and the weather can take the following values:

fuel_tank	empty, not_empty
battery	flat, not_flat
birds	singing, not_singing
sun	shining, not_shining

We might then create the following set of rules, listed in order of increasing priority:

1. **if** *weather* is not_nice
   then *action* is drive

2. **if** *car* is dead or *weather* is nice
   then *action* is walk

3. **if** *fuel_tank* is empty or *battery* is flat
   then *car* is dead

4. **if** *sun* is shining and *birds* is singing
   then *weather* is nice

5. **if** *sun* is not_shining
   then *weather* is not_nice

Let us assume we are given the following initial facts: *fuel_tank* is not_empty, *battery* is not_flat, *sun* is not_shining, and *birds* is singing.

Forward chaining Let us examine what we can derive from the initial facts. In the first round, the top most rule that applies is rule 5, which gives us the new fact *weather* is not_nice. In the second round, the top most rule that applies is rule 1, which gives us the new fact *action* is drive. In the third round, no further rules apply. Thus, the two facts we have derived are the full set of facts that can be derived from the initial facts.

Backward chaining Let us see if we can prove that *action* is drive. The only rule that can generate this outcome is rule number 1, so we need to prove its condition *weather* is not_nice. The only rule that can prove this condition is rule number 5, so we need to prove its condition *sun* is not_shining. Since we have that fact in our initial set of facts, we have now proven the statement *action* is drive by backward chaining.

4.3 Search

In the expert systems discussed so far, the goal has been to write down a set of rules that would lead to the best decision or action based on the observable facts. We can think of the observable facts as the *state* or the environment we are acting in, and our job is to make an AI system that can map any state to an appropriate action. In many situations it can be difficult to

define what the optimal action is: Rather, our desire is often to act in such a way that we achieve some goal in the long run.

By taking actions, we can modify the state of the environment, and our goal might be formulated in terms of reaching a certain desirable environment state. In this setup, we can formulate the problem using a graph where each node is a state and each edge between two states corresponds to an action. The objective is then to traverse the graph to reach the goal state starting from a given initial state.

Example 4.6 Removing paper jam

Consider a printer which can either be jammed or clear as well as have an open or closed lid. In total, the printer can be in four different states as illustrated in Figure 4.2. In each state we have three possible actions: We can *open* or *close* the lid of the printer, or we can *remove jam*. We can only remove the paper jam if the lid is open, and obviously, opening the lid if it is already open has no effect, so it does not change the state of the printer.

Initially, the printer is in the jammed, closed state. Our objective is to find the sequence of actions which will bring the printer into the cleared, closed state. This is easy, of course: We just have to open the lid, remove the jam, and close the lid again. Examining the state diagram, we can easily get to this solution by tracing the path from the initial state to the goal state. The reasons why this is an easy problem are:

1. We have *semantic* information (we know stuff about printers) that helps us solve the problem.
2. We have a simple state transition diagram of managable size in which we can visually trace the optimal path.
3. There happens to be exactly one unique way to get to the goal state.

If the state diagram was more complicated, it might not be so easy to spot the solution, or to determine if a solution even exists. In such a case, we could let a computer algorithm *search* through the state space for a solution.

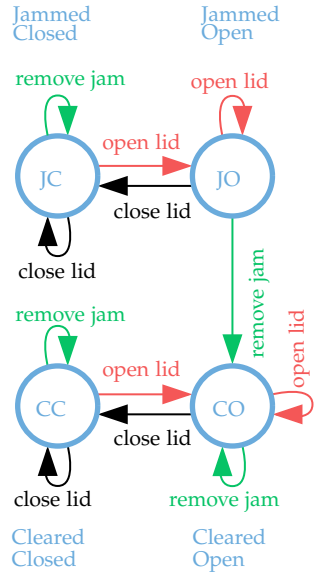

Figure 4.2: Example of a system with four states and three actions: A printer can be either Jammed or Cleared as well as Open or Closed. In combination, the printer can be in four different states, *JC*, *JO*, *CC*, and *JC*. In each state there are three possible actions: *remove jam*, *open lid*, and *close lid*. Starting in state *JC* the goal is to get to the state *CC*.

4.3.1 A search algorithm

The idea behind most search algorithms is fairly simple. We start at an initial state and explore all the neighboring states that we can go to by taking different actions. This gives us a new set of states to explore. If one of the neighboring states is a goal state, we have found a solution. If not, we need to explore one step further, so we take one of the neighbor state and explore all its neighbors and so on until we find the goal state. Since the state diagram can have cycles, we need to keep track of which states we have already seen, and make sure we do not explore each state more than once.

To make the algorithm more specific, we envision that we always have a *stack* of states that we need to explore. In the beginning, the only state in our stack is the initial state. In each step of the algorithm we remove the state at the top of our stack and check if it is the goal state. If not, we find all its neighbors, check which of them have already been examined, and add all un-examined neighbor states to the bottom of our stack. Then we simply repeat the process: Remove the state at the top of the stack, check if it is the goal state, and if not add all its unexamined neighbors to the stack. The algorithm ends either when we have found the goal state, or if the stack becomes empty, which means that we have explored all possible paths without reaching the goal, which proves that the problem cannot be solved. The algorithm can be described by the following Python code.

Definition 4.3 Breadth-first search

```
def bf_search(links, start, goal):
    """Given a dict where keys are states,
    and values are lists of connected states,
    finds a path from start to goal using
    breadth-first search"""
    state_list = [start]
    visited = [start]
    parent = {}
    while state_list:
        state = state_list[0]
        if state==goal:
            return parent
        state_list.remove(state)
        for neighbor in links[state]:
            if not neighbor in visited:
                visited.append(neighbor)
                state_list.append(neighbor)
                parent[neighbor] = state
```

4.3.2 *Breadth-first and depth-first search*

In our description of our search algorithm, at each step we removed the state at the top of the stack, found its non-visited neighbors and added them to the bottom of the stack. This means that for examples neighbors two levels away from the initial node will only be examined after all neighbors one level from the initial node have been considered. Thus, the algorithm expands in a *breadth-first* manner. Alternatively, we could have designed the algorithm to explore in a *depth-first* manner, simply by taking the next node to examine from the bottom of the stack rather than the top.

Example 4.7 Missionaries and cannibals

Three missionaries and three cannibals are on the left side of a river with a boat. The boat can only hold one or two people at once, and the goal is for all six people to cross the river to the right side. However, at no time must the missionaries be outnumbered by the cannibals at either side of the river. This type of problem can be solved using search, and to do so we need to define a representation of the states as well as the possible actions in each state.

State representation We can define the state by three numbers (m, c, b) that represent the number of missionaries (m) and cannibals (c) on the left side, and whether or not the boat (b) is on the left side. The initial state is $(3, 3, 1)$ and the goal state is $(0, 0, 0)$.

Actions The five possible state transitions are to transport one or two missionaries, one or two cannibals, or one of each to the other side. If the boat is on the left side, these transitions correspond to subtracting the following values from the current state:

$$(1,0,1) \quad (2,0,1) \quad (0,1,1) \quad (0,2,1) \quad (1,1,1)$$

Each of these transitions is possible only if it does not render m or c negative: For example it not possible to transport two cannibals, if there is only one left.

If the boat is on the right side, the situation is similar and the possible transitions correspond to adding the values above to the current state. Here, each transition is possible if it does not render m or c greater than 3.

Based on this, we can solve the problem by searching the state space for a solution. The full state diagram for the problem is shown in Figure 4.3.

Here is a question to think about: How would breadth-first and depth-first search solve this problem? Consider in which sequence the two methods will visit the states.

4.3.3 *Monte Carlo search*

In many practical problems the state space can be so large that it is not feasible to search through all possible action sequences. But still, if a problem can be represented as a graph search, it might be possible to use a suitable approximation to look for a solution.

One such approximative method is Monte Carlo[1] search. Rather than explore all possible state paths, the idea is to explore a smaller number of sample paths by choosing a random action at each step. These K randomly selected paths can either be explored all the way to the end (for example until we reach a dead end) or they can be explored only up to som finite

[1] Monte Carlo methods are a broad class of algorithms that rely on random sampling. These methods are named after the famous casino in Monte Carlo, Monaco.

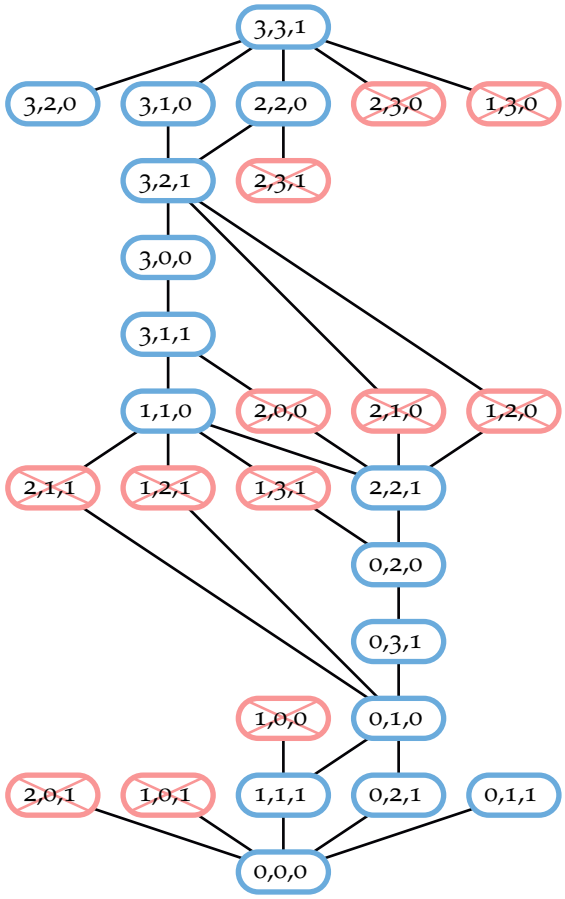

Figure 4.3: State diagram for the missionaries and cannibals problem. Each state is denoted by three numbers (m, c, b) denoting the number of missionaries m, cannibals c, and the position of the boat b where 0 and 1 indicate the left and right side of the river. Crossed-out boxes are invalid states where the cannibals outnumber the missionaries on one side of the river.

maximum number of steps.

Once we have explored K random paths, we need a mechanism to score them to measure which paths are most promising. If the problem we are trying to solve has a well-defined cost or reward associated with the states, we can use that to score the explored paths. Otherwise, we might score the paths according to their length, under the assumption that reaching far-away areas of the state space is desirable. Deciding upon a reasonable way to score the paths depends on the nature of the problem, and in some cases it is obvious what to do whereas in other cases it might be more difficult.

Once we have scored the random paths we have explored, we take the *first action* that leads to the highest score *on average*. This will move us one single step in the state diagram. Since we only have a small random sample of paths, we cannot trust that any of them are optimal, so we do not want to follow any of them in their full length. But taking just a single step chosen from the best performing set of paths is reasonable. After moving to the new state, the Monte Carlo search continues by exploring K new random paths from the new initial state. The algorithm continues in this manner until we reach a goal state or get stuck in a dead end.

Example 4.8 Shut the Box

The dice game *Shut the Box* (see Fig. 4.4) consists of nine wooden boxes numbered 1–9. The player rolls two dice and is allowed to close any combination of boxes such that their sum equals the sum of the dice. For example, if the player rolls 1+5=6 she can choose to close box 6, 1+5, 2+4, or 1+2+3. Once a box is closed it remains shut for the rest of the game. If the the boxes 7+8+9 are all closed, the player rolls only a single die. The game ends when all boxes are closed or if there is no combination of open boxes that match the sum of the dice. The final score is the sum of the open boxes at the end of the game, and the objective is to get as low a score as possible. A perfect game ends with a score of zero.

Figure 4.4: The dice game *Shut the Box*.

If the game is played completely at random, the average score is around 20 and a perfect game occurs with approximatly 2% chance. How do I know that? I wrote a small computer program that simulates the game and used it to

play 100 000 random games. But we can do much better than random with Monte Carlo search.

Consider a state in the game where the boxes 1+4+6+7+9 are open and we roll 5+2=7. We now have two possible moves: close box 1+6 or close box 7.

- First, we consider closing box 1+6 leaving 4+7+9 open. We now play, say, 1 000 random games from that position and record the final scores. We then compute the average score over the random games. (In a simulation I got an average score of 16.2.)
- Next, we consider closing box 7 leaving 1+4+6+9 open. Again we play 1 000 random games and compute the average score. (In a simulation I got 12.5.)

Based on this, we choose the second of the two possible moves, and close box 7.

We can now repeat the process. We throw the dice, and say we get 5+5=10. Again it happens that we have two options, to close 1+9 or 4+6. To decide which to go with, we repeat the Monte Carlo simulation as above. If we play the game this way using Monte Carlo search, the average score is around 11 and a perfect game occurs with approximately 6% chance.

As opposed to the search algorithms discussed previously, Monte Carlo search does not come with any guarantees. If a goal state can be reached Monte Carlo search might not find it, and if Monte Carlo search terminates without finding a goal state it does not mean that there is no solution.

Problems

1. Write down the truth table for the Boolean function $f(a, b) = (a \cdot \bar{b}) + (\bar{a} \cdot \bar{b})$.

2. Simplify the expression $f(a, b) = (a \cdot \bar{b}) + (\bar{a} \cdot \bar{b})$ as much as you can.

3. With n variables, how many different Boolean functions are possible? Self-check: With a single input, $n = 1$, there are four possible functions (can you list them?) and with 5 inputs there are more than 4 billion.

4. Which expert rule should you add to the three rules in Example 4.3 to make the system behave exactly as the Boolean function in Example 4.2, including the two undefined cases?

5. With the expert system in Example 4.5 and the initial facts *fuel_tank* is not_empty, *battery* is not_flat, *sun* is shining, and *birds* is singing, can you prove the statement *action* is walk using backward chaining? Which rules do you examine?

6. How can you modify the breadth-first search algorithm in Definition 4.3 so that it becomes a depth-first search algorithm?

7. Fig. 4.5 shows a state diagram with 16 states numbered 0 to 15 and possible transitions marked with arrows. If you want to search for a path from state 0 to state 15 using breadth-first and depth-first search search, which paths would be discovered? Assume that last seen states are visited first in the search.

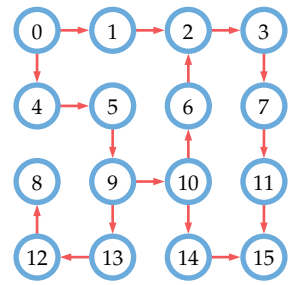

Figure 4.5: State diagram for a system with 16 states.

Solutions

1. The truth table for $f(a,b) = (a \cdot \bar{b}) + (\bar{a} \cdot \bar{b})$ is given by

a	b	$f(a,b)$
0	0	1
0	1	0
1	0	1
1	1	0

2. $f(a,b) = \bar{b}$.

3. There are 2^{2^n} possible Boolean functions with n inputs.

4. We know from Example 4.3 that the individual will not walk in the two undefined cases, so we might add the following rule which targets those specific cases:

   if *surveillance* is police and *road* is none and *light* is not green
   then *walk* is false
   Alternatively, we can rely on conflict resolution and add a simple rule at the top of the list (with lowest priority)

   if *surveillance* is police
   then *walk* is false

5. Yes, we can prove the statement. First we examine rule 2, which has the condition *car* is dead or *weather* is nice. These statements are possible consequents of rule 3 and rule 4 respectively, so these need to be examined. The condition of rule 3 is not true according to our initial facts, but the condition of rule 4 is true. This, in turn, makes rule 2 fire, which proves the statement.

6. We can do this by changing the line `node = node_list[0]` to `node = node_list[-1]` so that the search algorithm explores the node at the end of the stack rather than at the top of the stack.

7. For breadth-first search, we start at state 0. Let us keep track of the list of states we consider—to begin with, we just have state 0, so we write [0]. Now we explore state 0, which has two options, 1 and 4. We thus remove state 0 and append state 1 and 4 to the list, [0]→[1,4]. Now we explore the first state on the list, state 1, which

can only lead to state 2. Thus we remove state 1 and append state 2, [1,4]→[4,2]. Continuing this process we get [0]→[1,4]→[4,2]→[2,5]→[5,3]→[3,9]→[9,7]→[7,10,13]→[10,13,11]→[13,11,14,6]→[11,14,6,12]→[14,6,12,15], and we are done. Thus the search found the path 0→1→2→3→7→11→15.

For depth-first search, we get the sequence [0]→[1,5]→[1,9]→[1,10,13]→[1,10,12]→[1,10,8]→[1,10]→[1,6,14]→[1,6,15] and we are done. Thus the search found the path 0→4→5→9→10→14→15.