Searching State Spaces

1 A General Search Algorithm

How do we search for paths in our implicitly-specified graph (state space)?

We have some notion of the *current state* — the one we're currently looking at. At the start of the search, the current state is the one associated with the start node of the graph (the initial state).

We check whether the current state is a goal state (whether it satisfies the goal condition). If it does then, assuming we're looking for only one solution path, we can stop.

If the current state is not a goal state, we *expand* the current state. What this means is that we apply operators to this state to *generate* its *successor* states.

While there might be no successor states (a dead-end) or just one successor state, in general there will be multiple successors. The essence of search is to choose one state for further exploration (it becomes the new current state) and to put the others somewhere in case we want to come back to them, e.g. if the chosen one does not lead to a solution.

The data structure in which we keep states that have not yet been explored is called an *agenda*.

Given that there are multiple states waiting on the agenda, yet to be explored, the policy which determines which state to explore next is called the *search strategy* (or *control strategy*).

Here's the algorithm in pseudocode:

```
agenda initially contains only the initial state

while agenda is not empty
{  let currentNode = node removed from front of agenda
   if currentNode satisfies goal test
   {  return the path of actions that led to the currentNode
   }
   else
   {  compute the successors (states resulting from expanding the currentNode)
      *add successors to the agenda*
   }
} // end while
return failure
```

Different search strategies result from different implementations of the line enclosed in asterisks.

2 Java Implementation

To make this more concrete, here are some fragments from a Java implementation. (The full source code is available from the course web site.) The code below will be explained in the lectures.

We start with a Java interface that specifies what it means to be a state in the state space. (We'll add another method to this later.)

```
public interface IState
{
   public boolean isGoal();

   public Iterator getSuccessors();
}
```

(In fact, it is common for classes that implement this interface to also define the `equals` method. You might also want to implement the `hashCode` method for reasons that will become apparent below. You might also include the method `toString` so that the output looks nice.)

Here's an example of a class that implements the `IState` interface.

```
public class WaterJugsState
   implements IState
{
   public WaterJugsState(int theX, int theY)
   {  x = theX;
      y = theY;
   }

   public boolean isGoal()
   {  return x == 2;
   }

   public Iterator getSuccessors()
   {  List succStates = new ArrayList(3);
      if (x < 4)
      {  succStates.add(new SuccessorState(
            new WaterJugsState(4, y), "Fill4GfromTap", 1));
      }
      // Etc! 7 more rules omitted to save space.
      return succStates.iterator();
   }

   private int x;
   private int y;
}
```

The nice thing is that this is the only part of the code that is problem-specific. To solve the 8-puzzle instead of the water jugs problem requires only that we replace the above class by an 8-puzzle-specific class.

There's another class called `SuccessorState`, which is not very interesting and would not help your understanding, so we'll ignore it. (All it does, for those who 'need' to know, is bundle together several pieces of information so that they can be passed as a single object from, e.g., `WaterJugsState` to `Node`.)

The class that does most of the work is `StateSpaceSearch`. Here's a relevant fragment of it:

```
public class StateSpaceSearch
{
   public StateSpaceSearch(IState theInitialState)
   {  initialState = theInitialState;
   }

   private List getSolutionPath()
   {  addNode(agenda, new Node(initialState));
      while (! agenda.isEmpty())
      {  Node currentNode = agenda.removeFrontNode();
         if (currentNode.getState().isGoal())
         {  return currentNode.getPathFromRoot();
         }
         addNodes(agenda, currentNode.expand());
      }
      return null;
   }

   private void addNodes(IAgenda theAgenda, List theNodes)
   {  Iterator iter = theNodes.iterator();
      while (iter.hasNext())
      {  addNode(theAgenda, (Node) iter.next());
      }
   }

   protected void addNode(IAgenda theAgenda, Node theNode)
   {  theAgenda.addNode(theNode);
   }

   private IState initialState;
   private IAgenda agenda;
}
```

The `StateSpaceSearch` class definition makes use of nodes. These are defined as follows (some details omitted):

```
public class Node
{
   public Node(IState theState, String theAction, int theCost,
      Node theParent, int theDepth)
   {  state = theState;
      action = theAction;
      pathCost = theParent.getPathCost() + theCost;
      parent = theParent;
      children = new LinkedList();
      depth = theDepth;
   }
```



```
   public List expand()
   {  depth++;
      int childDepth = depth;
      Iterator iter = state.getSuccessors();
      while (iter.hasNext())
      {  SuccessorState succ = (SuccessorState) iter.next();
         children.add(new Node(succ.getState(), succ.getAction(),
            succ.getCost(), this, childDepth));
      }
      return children;
   }

   private IState state;
   private String action;
   private int pathCost;
   private Node parent;
   private List children;
   private int depth;
}
```

Finally, we say what it means to be an agenda by giving a Java interface. Any class that implements this interface can be used as an agenda.

```
public interface IAgenda
{  public boolean isEmpty();
   public void addNode(Node theNode);
   public Node removeFrontNode();
}
```

We'll see how to run the code in subsequent lectures, after we've discussed how to write classes that implement the `IAgenda` interface.

3 Search Trees

One way to think about the search algorithm is that it is making explicit parts of the implicitly-specified state space: the nodes it actually visits and the edges it actually traverses. The parts of the state space that the search algorithms visits can be shown in the form of a tree, called the *search tree*.

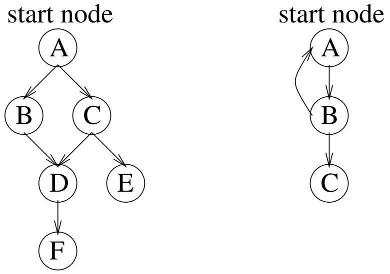
It's important to distinguish the state space from the search tree. The state space is all the states reachable by sequences of actions from the initial state. The search tree is different because:

- Some search strategies may leave parts of the state space unexplored. In other words, there may be nodes and edges in the state space that never get visited and so do not appear in the search tree. This, of course, is a 'good thing': it improves efficiency, although, if we skimp too much, we may end up missing the solution paths, which, in general, is a 'bad thing'.
- Some search strategies may re-explore parts of the state space. This can happen when two or more paths in the state space lead to the same node. Unless steps are taken to deal with this, then some nodes in the state space may get visited and expanded more than once, and so they appear in the search tree more than once. This is, in general, a 'bad thing', although the cost of eliminating it can be high. (A common special case of this is when the state space is cyclic. Unless steps are taken to deal with cycles, the search tree may then be infinite.)

Let's explore the second bullet point in more detail.

4 Avoiding Re-exploration

Strictly, we cannot draw search trees unless we know what the search strategy is. But, to illustrate the second bullet point above, without getting too bogged down in wondering what the exact search strategy is (i.e. how to decide what to visit next), in the lecture we will draw possible search trees for the following two state spaces:



To avoid re-exploration of parts of the state space, we must be more selective about which states we add to the agenda. Some of the successors of the current state should be discarded. There are various ways of deciding which to discard, and they vary in how effective they are in avoiding re-exploration and in how much time & space they cost us. (Sometimes it might be better to allow some re-exploration, rather than pay the price of eliminating it).

Three options for avoiding (some or all) re-exploration are common:

- Discard any successor that is the same as the current node's parent.
- Discard any successor that is the same as another node on that path.
- Discard any successor if it is the same as any previously generated node.

Question. *How effective at avoiding re-exploration are these three options? What do they cost in time & space?*

We can handle the first two of these in the Java implementation by defining a subclass of `StateSpaceSearch`, with a different definition for adding nodes to the agenda.

Discard any successor that is the same as the current node's parent. Here's what `StateSpaceSearchWithNoUndos` contains:

```
protected void addNode(IAgenda theAgenda, Node theNode)
{  Node parent = theNode.getParent();
   Node grandParent = null;
   if (parent != null)
   {  grandParent = parent.getParent();
   }
   if (((parent == null) ||
        (! parent.getState().equals(theNode.getState()))) &&
       ((grandParent == null) ||
        (! grandParent.getState().equals(theNode.getState()))))
   {  super.addNode(theAgenda, theNode);
   }
}
```


Discard any successor that is the same as another node on that path. Here's what `StateSpaceSearchWithNoCycles` contains:

```
protected void addNode(IAgenda theAgenda, Node theNode)
{  Node ancestor = theNode;
   do
   {  ancestor = ancestor.getParent();
      if ((ancestor != null) &&
           (ancestor.getState().equals(theNode.getState())))
      {  return; // node is not added
      }
   } while (ancestor != null);
   super.addNode(theAgenda, theNode);
}
```

Discard any successor if it is the same as any previously generated node. This is the most expensive option. The time it takes to carry out these checks can be made manageable by storing nodes in a hash table: checking is then just a hash table lookup operation (which takes constant or near constant time). The memory space needed may remain a problem.

[Advanced point. In fact, I'm oversimplifying this third option. (*Ignore this if it makes no sense.*) From the above, you might assume that if the newly-generated node is the same as a previously-generated node, we always throw away the new node. *But, this is not correct.* If the cost of the path to the new node is less than the cost of the path to the previously-visited node, then we should not throw away the new node (because the path to it is cheaper.) To handle all of this properly makes the algorithm so complicated and its costs (both space and time) so much higher that this option is hardly ever implemented. If you want to read a proper description of such an algorithm, consult a textbook such as Nils Nilsson: *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann, 1998.]

5 Evaluating a Search Strategy

We're going to look at several search strategies over the next few lectures. We need some criteria in terms of which we can compare them. The criteria used in AI are these:

Completeness: A search strategy is complete if it guarantees to find a solution when there is one.
Note that is a somewhat one-sided definition. It doesn't impose any requirement on the strategy in the case where there is no solution. In these circumstances, maybe the strategy will say 'There's no solution', or maybe it will run forever. All that matters for completeness is, if there is at least one solution, then it gets found.

Optimality: A search strategy is optimal if it guarantees that it will find the highest-quality solution.
What we mean here is that the first solution path it finds must be the highest-quality one. We're not entertaining the idea that it finds a solution, and then continues to search for other solutions so that it can choose the best of them afterwards.
Note that an algorithm cannot be optimal if it isn't complete. Or, if you prefer, optimality implies completeness. The notion of 'highest-quality' here concerns the path cost. Recall that we have a function g, the path cost function, which sums the costs of the actions along a path. We'll be writing $g(n)$ to mean the cost of the path from the start node to node n. For optimality, we want the strategy to find the cheapest solution path. (In the case where the actions haven't been assigned any costs, then we want to find the shortest path — this, of course, is equivalent to treating all actions as having uniform cost, and $g(n)$ is then just the length of the path to n.)

Time complexity: How long does it take to find a solution? We'll generally report worst-case results, but best-case and average-case are also of interest.

Space complexity: How much memory is needed to perform the search worst-, best- and average-case)?