Uninformed Search

1 Search Strategies

The search strategy is responsible for deciding which node to expand next. Search strategies fall into two classes.

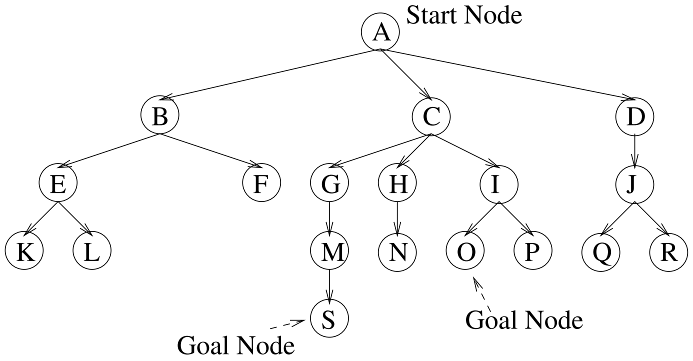
In *uninformed search* (also called *undirected search* and *blind search*), the strategy has no problem-specific knowledge that would allow it to prefer to expand one node over another. It can distinguish goal nodes from other nodes, and it will know the length of the path or the cost of the path from the initial node to each node on the agenda. But it will know nothing about the probable length or cost of extending a path so that it leads to a goal node.

In *informed search* (also called *directed search* and *heuristic search*), we make available to the strategy some problem-specific knowledge about the likely length or cost of the paths from each node on the agenda to a goal node.

This lecture looks exclusively at uninformed search.

It turns out that we can use our general search algorithm but easily implement different search strategies simply by altering the way the agenda works. Our algorithm always expands the node that is on the front of the agenda. But we have not yet said where, when it is adding nodes to the agenda, it adds them. By changing this, we get our different strategies.

We'll be illustrating the strategies in the lecture on the following state space:



In some ways, this isn't a very standard state space. First, I've shown it explicitly, rather than specifying a start state and some operators. Second, to keep the lecture simple, I've not allowed more than one path to each node. This means we don't have to think about avoiding re-exploration: the searches will all be finite. But remember, this is the state space; it's not the search tree (which is what we'll show in the lecture).

2 Breadth-First Search

In *breadth-first search*, we treat the agenda as a *queue*. Nodes come off the front (as always), and new nodes are added to the back. This means that all nodes at depth i in the search tree are expanded before any at $i + 1$. (All paths of length 1 are considered before any of length 2, and all of length 2 are considered before any of length 3, and so on.)

Here's a Java class that implements `IAgenda` to give an agenda that acts as a queue:


```
public class QueueAgenda
   implements IAgenda
{
   public QueueAgenda()
   {  queue = new LinkedList();
   }

   public boolean isEmpty()
   {  return queue.isEmpty();
   }

   public void addNode(Node theNode)
   {  queue.add(theNode); // 'add' adds to the end.
   }

   public Node removeFrontNode()
   {  return (Node) (queue.remove(0));
   }

   protected List queue;
}
```

We can add the following to the `StateSpaceSearch` class definition from the previous lecture:

```
   public List breadthFirstSearch()
   {  agenda = new QueueAgenda();
      return getSolutionPath();
   }
```

And then to run a breadth-first search on the water jugs state space, we would write code such as:

```
WaterJugsState startWJ = new WaterJugsState(0, 0);

// Allowing re-exploration:
sss = new StateSpaceSearch(startWJ);
printPath(sss.breadthFirstSearch());

// Avoiding undos:
sss = new StateSpaceSearchWithNoUndos(startWJ);
printPaths(sss.breadthFirstSearch());

// Avoiding cycles:
sss = new StateSpaceSearchWithNoCycles(startWJ);
printPaths(sss.breadthFirstSearch());
```

Think about the completeness, optimality and time- and space-complexity of breadth-first search.

3 Depth-First Search

In *depth-first search*, we treat the agenda as a *stack*. Nodes are popped from the front of the agenda (as always), and new nodes are pushed onto the front of the agenda. This means that the strategy always chooses to expand one of the nodes that is at the deepest level of the search tree. It only expands nodes on the agenda that are at a shallower level if the search has hit a dead-end at the deepest level. (A path is expanded as much as possible —until it reaches a goal node or can be expanded no more— prior to extending other paths.)

Here's a Java class that implements `IAgenda` to give an agenda that acts as a stack:

```
public class StackAgenda
   implements IAgenda
{
   public StackAgenda()
   {  stack = new Stack();
   }

   public boolean isEmpty()
   {  return stack.empty();
   }

   public void addNode(Node theNode)
   {  stack.push(theNode);
   }

   public Node removeFrontNode()
   {  return (Node) (stack.pop());
   }

   private Stack stack;
}
```

Think about the completeness, optimality and time- and space-complexity of depth-first search.

4 A Recursive Version of Depth-First Search (Optional)

There is an alternative way to implement depth-first search, which does not require us to implement our own agenda. Instead, we implement a recursive algorithm. When a node is expanded, the search algorithm is called recursively on each successor node. The reason we don't need an explicit agenda is that the unexpanded options are implicit in the local state of each invocation on the call stack. When a path can be expanded no further, the call stack is popped. The effect is a backtracking implementation of depth-first search.

Here's an implementation (not carefully tested!):

```
private List getSolutionPath(Node currentNode)
{  if (currentNode.getState().isGoal())
   {  return currentNode.getPathFromRoot();
   }
   Iterator iter = currentNode.expand().iterator();
   List path = null;
   while (iter.hasNext() && path == null)
   {  path = getSolutionPath((Node) iter.next());
   }
   return path;
}
```


To run this, you would also make use of `Node`, `SuccessorState`, `IState` and a class that defines some problem-specific state such as `WaterJugsState`. None of the code that mentions agendas is necessary.

5 Depth-Bounded Depth-First Search

Depth-Bounded Depth-First Search (also referred to simply as *depth-bounded search*) is just like depth-first search, but paths whose lengths have reached some limit, l, are not extended further. This is implemented by having an agenda that operates as a stack but any node whose depth in the tree is greater than l is not added to the agenda.

Here's a subclass of `DepthFirstAgenda` which incorporates this depth limit. Mostly it inherits from `StackAgenda`, but it overrides `addNode`:

```
public class DepthBoundedAgenda
   extends StackAgenda
{
   public DepthBoundedAgenda(int theDepthLimit)
   {  super();
      depthLimit = theDepthLimit;
   }

   public void addNode(Node theNode)
   {  if (theNode.getDepth() <= depthLimit)
      {  super.addNode(theNode);
      }
   }

   private int depthLimit;
}
```

Depth-bounded search can avoid some of the problems of depth-first search.

Think about the completeness, optimality and time- and space-complexity of depth-first search.

6 Iterative-Deepening Depth-Bounded Depth-First Search

Iterative-deepening depth-bounded depth-first search (also referred to simply as *iterative-deepening search*) carries out *repeated* depth-bounded searches, using successively greater depth limits. It first tries a depth-bounded search with a depth bound of 0, then it tries a depth-bounded search with a depth limit of 1, then of 2, and so on.

Since this search strategy is based on depth-bounded search, its implementation does not require a new kind of agenda. Instead, we repeatedly invoke depth-bounded searches until a solution is found.

The following code would be added to the `StateSpaceSearch` class definition:

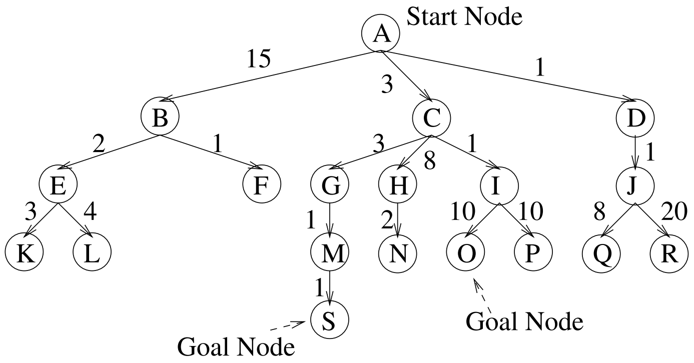
```
public List iterativeDeepeningSearch()
{  List solutionPath = null;
   int depthBound = 0;
   do
   {  agenda = new DepthBoundedAgenda(depthBound);
      solutionPath = getSolutionPath();
      depthBound++;
   } while (solutionPath == null);
   return solutionPath;
}
```

7 Least-Cost Search

The strategies we have looked at so far do not take path cost into account. Some of them are capable of finding the *shortest-path*. But none of them is, in general, capable of finding the *cheapest* path. (Of course, if all action costs are uniform, e.g. if all actions have a cost of 1, then the cheapest path and the shortest path amount to the same thing. We want to consider here the more general case where different actions can have different costs, and so the shortest path may not be the cheapest path.)

In *least-cost* search, the agenda is a *priority-ordered queue*, ordered by cost. When nodes are added to the agenda, they are added in such a way as to keep the agenda sorted by cost, with the cheapest node at the front. Hence the node that comes off the front of the agenda is always the cheapest unexplored node. (The path that is extended is always the one that currently has the least cost.)

In the lectures, we'll use this state space for our example:



An (inefficient) Java implementation of such an agenda is:

```
public class PriorityQueueAgenda
   extends QueueAgenda
{
   public PriorityQueueAgenda(NodeComparator theComparator)
   {  super();
      comparator = theComparator;
   }

   public void addNode(Node theNode)
   {  Iterator iter = queue.iterator();
      int i = 0;
      while (iter.hasNext())
      {  Node queuedNode = (Node) iter.next();
         if (comparator.isLessThan(theNode, queuedNode))
         {  queue.add(i, theNode);
            return;
         }
         i++;
      }
      queue.add(theNode);
   }

   private NodeComparator comparator;
}
```

You'll see that, for generality, this priority-ordered queue implementation compares nodes using an object called a `NodeComparator`. This has an abstract class definition as follows:

```
public abstract class NodeComparator
{
   public abstract boolean isLessThan(Node aNode, Node otherNode);
}
```

And then the concrete subclass that we use to get least-cost search is:

```
public class NodeCostComparator
   extends NodeComparator
{
   public boolean isLessThan(Node aNode, Node otherNode)
   {  return aNode.getPathCost() < otherNode.getPathCost();
   }
}
```

You'll see that by using an abstract class for the priority-ordered queue comparator, we'll have an easy time implementing some further search strategies in a subsequent lecture.

A least-cost search is run using this method (included in `StateSpaceSearch`):

```
   public List leastCostSearch()
   {  agenda = new PriorityQueueAgenda(new NodeCostComparator());
      return getSolutionPath();
   }
```

Exercises

1. (*Past exam question*) A farmer, a wolf, a goat and a sack of cabbages are on the left bank of a river. There is a boat on the left side of the river too. It must be crewed by the farmer, and has room for only one of the other three. At no point can the farmer leave the wolf and goat together unattended. Similarly, the goat and the cabbages cannot be left together unattended. The goal is to ferry all four across to the right bank.

   One (iconic) problem representation is to represent the farmer, wolf, goat, cabbages and boat as variables, F, W, G, C and B respectively, and then to represent states by two sets: those items on the left bank, and those on the right. Therefore, the initial state is $\{FWGCB\} - \{\}$ and the goal state is $\{\} - \{FWGCB\}$. There are 8 operators: one for moving farmer and wolf from left bank to right, another for moving them back, two more for moving farmer and goat, two more for moving farmer and cabbages, and two for moving the farmer unaccompanied.

   (a) Draw the *state space*.

   (b) Your state space is to be searched using an agenda-based search algorithm. However, when the current node is expanded, if a successor is the same as any node already visited on that *path*, it is discarded.

   Hence, draw the *search trees* that are built by *depth-first* search and *breadth-first* search. (Multiple answers are possible. You need give only one such answer in each case.)

   (c) The algorithm is changed so that a successor is discarded if it is the same as *any* previously visited node.

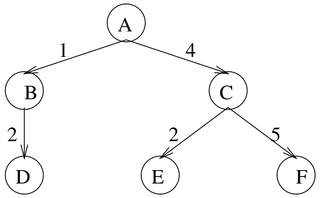
   Would the search tree for *depth-first* search be any different from the one you gave above. If so, draw it.

   Would the search tree for *breadth-first* search be any different from the one you gave above. If so, draw it.

2. Give one advantage of *depth-first* search over *breadth-first* search.

3. Give one advantage of *depth-first iterative deepening* search over *depth-first* search.

4. (*Part of a past exam question*) Consider the following state space in which the states are shown as nodes labeled A through F. A is the initial state and E is the goal state. The numbers alongside the edges represent the costs of moving between the states.



   Show how each of the following search strategies finds a solution in this state space by writing down, in order, the names of the nodes removed from the agenda. Assume the search halts when the goal state is removed from the agenda. (In some cases, multiple answers are possible. You need give only one such answer in each case.)

   (a) Breadth-first;

   (b) Depth-first;

   (c) Iterative-deepening; and

   (d) Least-cost search.