Principles of Artificial Intelligence
Fall 2005
Handout #2
Goal-Based Agents

Vasant Honavar
Artificial Intelligence Research Laboratory
Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011

Last revised: August 26, 2006

1 Goal-Based Agents

In this chapter, we consider the design of goal-based agents. The specification and design of goal-based agents involves answering the following questions:

1. *What is the goal to be achieved?* This involves describing a situation we want to achieve, a set of properties that we want to hold (when the agent succeeds at its goal), etc. This requires defining a *goal test* so which captures what it means to have achieved/satisfied the goal. While in some domains (e.g., chess) it is rather straightforward to specify the goal test, in others, it is not as obvious and requires considerable thought. In general, the cognitive processes having to do with goal selection and goal specification in humans and animals are poorly understood. Consequently, the design of autonomous systems that select, prioritize, and update their goals is largely an unsolved problem in AI. In what follows, we assume that the system designer or user will specify the goal to be achieved.

2. *What are the actions that are available to the agent?* We need to specify precisely all of the primitive actions (including their preconditions and their expected effects on the environment) that are sufficient (at least in principle) to achieve the goal. Early AI systems assumed that given an action (also called an operator) and a description of the current state of the world, the action completely specifies the conditions under which the action can be applied to the current world as well as the the exact state of the world after the action is executed in the current world. Actions were viewed as atomic, discrete events that can be thought of as occurring at an instant in time. Thus, the world is in a state before the action is executed, and it makes a transition to a different state (specified implicitly or explicitly by the effects of the action) once the action is executed. For example, if "Mary is in class" and she performs the action "go home" she will end up "at home." Given this representation of action, we have no way of describing a point in time when Mary is neither in class nor at home (i.e., she is on her way home). It is also typical to assume that no two actions are executed simultaneously. It is generally assumed that the world itself has no dynamics of its own besides the aspects that are explicitly modeled by the agent. While these assumptions are quite reasonable for manipulating "symbolic worlds" of the sort encountered in mathematical problem-solving, they can be problematic for agents that need to cope with dynamic, uncertain, real-world environments. Relaxing the simplifying assumptions associated with action specification to deal with such scenarios is a topic of much current research in intelligent agents and multi-agent systems.

3. *What information is necessary to encode about the the world to sufficiently describe aspects of the world that are relevant for accomplishing the goal?* Early AI systems assumed that all the information necessary for choosing an action is available in each percept so that each state is a complete description of the current world. What to encode in a state is the knowledge representation problem. That is, we must decide what information from the raw perceptual or sensory data is relevant to keep, and what form the data should be represented in so as to make explicit the most important features of the data for solving the goal. Is the color of the boat relevant to solving the Missionaries and Cannibals problem? Is sunspot activity relevant to predicting the stock market? Do the facial expressions of the opponent give important clues while playing a game of poker? What to represent is a very hard problem that is usually left to the system designer. A closely related issue has to do with the level of abstraction or detail used to describe the world. If the granularity of the representation is too fine, we will "miss the forest for the trees." If it is too coarse, we will miss critical details for solving the problem. The processes used by humans and animals in coming up with the right representation for each task are far from being fully understood and constitute a major topic of current research.

We formalize some aspects of goal-based problem-solving in what follows.

2 Problem Solving as Search

A wide range of problems in AI—including, among others, *theorem proving*, *game playing*, *planning*, and *learning*—can be formulated at an abstract level as essentially search problems.

As noted above, representation is a key issue in problem solving. Consider 17 sticks arranged in 6 squares as shown in Figure 1. Suppose we are asked to remove just 5 sticks so that we are left with only 3 squares (with no extra sticks). The number of possible ways is $\binom{17}{5}$. However, the problem could be represented as that of removing 3 squares from the current configuration of 6 squares. The number of possibilities is just $\binom{6}{3}$ which is considerably smaller. Also note that the preceding discussion implicitly assumed that the squares have to be of the same size. Thus, representation plays a key role in the formulation of search problems. We have not precisely defined what we mean by representation. For a more detailed discussion of the nature of representation, see the relevant sections of the AI overview chapter.

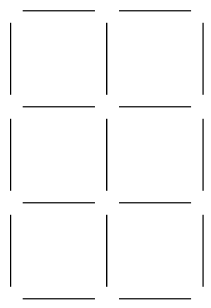

Figure 1: Importance of Representation

The 8-puzzle, as described on page 63 of Artificial Intelligence A Modern Approach by Stuart Russell and Peter Norvig, "consists of a 3 x 3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can be exchanged with the blank tile." (Figure 2) Given an initial configuration of 8 numbered tiles on a 3 x 3 board, move the tiles in such a way so as to produce a desired goal configuration of the tiles. Here, the state of the world is encoded as a 3 x 3 array of the tiles on the board.

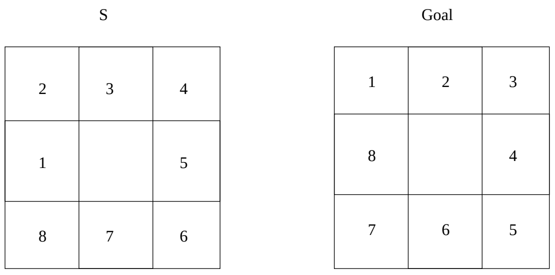

Figure 2: 8-Puzzle Problem

At any state, we have between 2 and 4 options among the following:

- exchange the blank with the square to its left.
- exchange the blank with the square to its right.
- exchange the blank with the square above it.
- exchange the blank with the square below it.

Note that this is a more efficient encoding of the operators than one in which each of four possible moves for each of the 8 distinct tiles is used. Initial State is a given configuration of the board. and the goal is a particular configuration of the board (wherein each of the tiles is ordered numerically around the periphery board and the blank tile is located at the center.

The search spaces encountered in AI problems usually grow exponentially with the length of the solution (i.e., path to a goal) and seldom will we attempt to enumerate the entire search space. Control of search to minimize the fraction of the search space that needs to be explored in arriving at a solution is a key topic of research in AI.

Here are some additional examples:

- *Missionaries and Cannibals* There are 3 missionaries, 3 cannibals, and 1 boat that can carry up to two people on one side of a river. Goal is to move all the missionaries and cannibals across the river. Constraint on any state is that missionaries can never be outnumbered by cannibals on either side of the river, or else the missionaries are killed. The states correspond to configuration of missionaries and cannibals and boat on each side of the river. The operators specify movement of the boat containing some set of occupants across the river (in either direction) to the other side.

- *Cryptarithmetic* Find an assignment of digits (0, ..., 9) to letters so that a given arithmetic expression is true. For example, SEND + MORE = MONEY Note: In this problem, unlike the two above, the solution is NOT a sequence of actions that transforms the initial state into the goal state, but rather the solution is simply finding a goal node that includes an assignment of digits to each of the distinct letters in the given problem. Such search problems are often called *pathless* search problems.

- *Water Jug Problem* Given a 5-gallon jug and a 2-gallon jug, with the 5-gallon jug initially full of water and the 2-gallon jug empty, the goal is to fill the 2-gallon jug with exactly one gallon of water. We can specify the states by ordered pairs of the form (x, y), where x = number of gallons of water in the 5-gallon jug and y is gallons in the 2-gallon jug. The initial state is $(5, 0)$. the goal State = $(\star, 1)$, where $\star$ denotes an arbitrary number. Operators can be specified as follows:
  - $(x, y) \rightarrow (0, y)$; empty the 5-gallon jug
  - $(x, y) \rightarrow (x, 0)$; empty the 2-gallon jug
  - $(x, 2) \rightarrow (x + 2, 0)$ provided $x \leq 3$; pour 2-gallon into 5-gallon jug making sure that the jug won't overflow
  - $(x, 0) \rightarrow (x - 2, 2)$ provided $x \geq 2$; pour 5-gallon into 2-gallon jug
  - $(1, 0) \rightarrow (0, 1)$; empty 5-gallon jug into the 2-gallon jug

2.1 Formalizing Problem Solving as Search

- The *states* — snapshots of the condition of a problem at each step of an attempted solution.
- *operators* — means of transforming one state into another.
- *state space* — set of all possible states given a start state and a set of operators.

More precisely, the specification of the state space problem requires:

1. A start state S.
2. A set of operators O={O1, O2,..., On} (partial functions that map one state into another).
3. A predicate ϕ that defines the set of goal states G. $\phi(g)$ = TRUE if $g \in G$.

A solution to the state space problem is a sequence of *operator applications* leading from the start state S to one of the goal states $g \in G$. So a natural way to think of a state space is to view it as a (potentially infinite) directed graph whose nodes are states and whose arcs correspond to operator applications.

Note that:

- Each node is a data structure that contains a state description plus other information such as the parent of the node, the name of the operator that generated the node from that parent, and other bookkeeping data
- Each arc corresponds to an instance of one of the operators. When the operator is applied to the state associated with the arc's source node, then the resulting state is the state associated with the arc's destination node
- Each node has a set of successor nodes corresponding to all of the legal operators that can be applied at the source node's state. The process of expanding a node means to generate all of the successor nodes and add them and their associated arcs to the state-space graph
- One or more nodes are designated as start nodes
- A goal test predicate is applied to a state to determine if its associated node is a goal node
- A solution is a sequence of operators that is associated with a path in a state space from a start node to a goal node State-space search is the process of searching through a state space for a solution by making explicit a sufficient portion of an implicit state-space graph to include a goal node. Hence, initially only the start node S is in the graph. When S is *expanded*, its successors are generated and the graph is extended by adding the successors and the associated arcs. This process continues until a goal node is found.
- Each node implicitly or explicitly represents a partial solution path from the start node to the given node. In general, from this node there are many possible paths (and therefore solutions) that have this partial path as a prefix

It is possible to associate with each arc a fixed, positive cost corresponding to the cost of applying the operator. The cost of a solution is the sum of the arc costs on the solution path. A common variant of the state-space search problem involves finding the cheapest (minimum cost) solution.

Search problems, although conceptually simple are computationally hard. Therefore, much work in AI has to do with reducing the amount of search effort.

3 Review of Some Familiar Search Algorithms

Given the following graph enumerate the results of a Depth First Search (DFS) and a Breadth First Search (BFS) on it.

```
            N1
          / |  \
         /  |   \
       N2  N3   N4
      /  \  | \
     /    \ |  \
   N5   N6 N7 N8
```

DFS: N1-N2-N5-N6-N3-N7-N8-N4 or N1-N4-N2-N5-N6-N3-N7-N8 etc.
BFS: N1-N2-N3-N4-N5-N6-N7-N8 etc.

3.1 Pseudocode for a General Class of Search Algorithms

```
1. Let L = list of nodes yet to be examined (initialized to start node $S$).
2. If L is empty return FAIL.
   Else, pick a node ($n$) in L.
3. If $n$ is a goal node (i.e. $n \in G$ ), stop and return the path from $S$ to $n$.
4. Otherwise remove the node $n$ from L.
   Add to L all of $n$'s children labeling each with the corresponding path from $S$.
   Return to step 2.
```

NOTE: If L is a *queue* we get BFS and if L is a *stack* we get DFS. In general, search algorithms differ in terms of steps 2 and 4 are handled.

3.2 Analysis of Breadth-First Search and Depth-First Search

The search strategies are analyzed in terms of four criteria:

- Space complexity: amount of memory during run-time
- Time complexity: amount of time to find a solution
- Completeness: the search strategy is guaranteed to find a solution, if one exists (at a finite depth)
- Admissibility: the search strategy is guaranteed to find the optimal (cheapest)solution if such a solution exists.

- Optimality with respect to some specified criteria (e.g., amount of search effort relative to the quality of solution).

NOTE: Admissibility implies completeness.

To make the analysis concrete we make the following assumptions.

- Assume uniform branching factor b
- A single goal node exists at depth d (also the total depth of the search tree)
- Any node at depth d is equally likely to be a goal node.

3.2.1 Breadth First Search

We need all nodes at depth k to be on the list before we can look at any node on depth k+1. To look at the nodes on level d we need to store b^{d-1} nodes. So, in the worst case (i.e. if the last node at level d is the goal) we need to store b^d nodes. Thus space requirement is $O(b^d)$ (prove this rigorously).

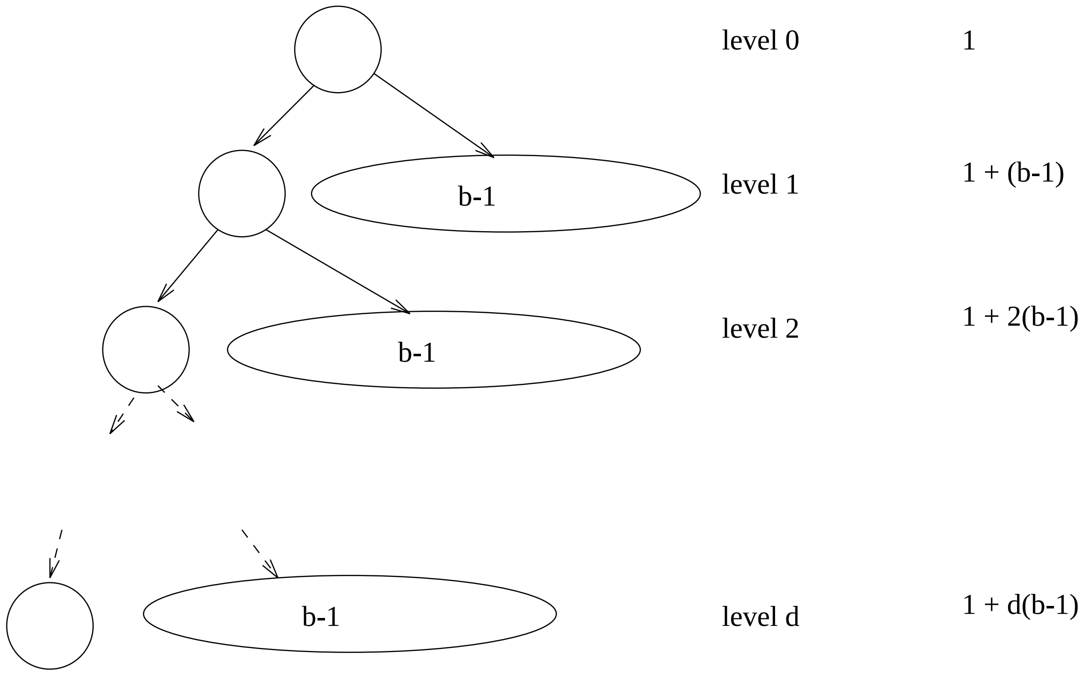

Figure 3: Space requirement for DFS

3.2.2 Average Time Requirement

This is roughly the total number of nodes examined on average. We know that the goal node is randomly distributed at level d with every node at that level being equally likely to be the goal. Before looking at any node on level d we must examine the following number of nodes.

$b^0 + b^1 + b^2 + \ldots + b^{d-1} = \frac{b^d - 1}{b-1}$

(If you don't see this, prove the formula for sum of such a series by mathematical induction). At depth d we must examine 1 node in the best case and b^d nodes in the worst case. Since we have assumed that goal node is uniformly distributed at depth d the average time required for BFS is:

$\frac{b^d-1}{b-1} + \frac{1+b^d}{2}$

(Note that you cannot simply take the average of the best and worst case time requirements in general to get the average case. Here we were able to do so because each node at depth d was assumed equally likely to be the goal. So the average case turned out to be the sum of the arithmetic series from 1 to b^d divided by b^d). For large b and d this is roughly $\frac{b^d}{2}$.

Observations:

- Almost all of the work is done at the fringe of the tree (at depth d) and on an average at least half of the nodes at depth d are to be examined.
- BFS is complete, even for infinite graphs
- BFS is not admissible in general, but is admissible if graph consists of uniform cost arcs (we will see this later).

3.2.3 Depth First Search

As the DFS proceeds, at each level a total of b - 1 nodes get added to the list L at each level. Thus the total space requirement as seen from the figure below is: $1 + d(b - 1)$ (which is $O(bd)$) . Interestingly, the space requirement is linear. However DFS as it stands, can get caught in infinite paths and not find a goal node (even if one exists).

To compute the time complexity of DFS, we assume that the search graph is finite and the graph exists only till depth 'd'.

Best Case: d+1 nodes (when the goal node is the first node on level d).
Worst Case: $b^0 + b^1 + \ldots + b^d$ i.e. all nodes in the tree (when the goal node is the last node on level d).

Question: Can we compute the average case by adding the best and worst cases and dividing by 2?

At first it does not look like the goal is uniformly distributed between the numbers representing the best and worst cases but after careful analysis we will see that it is indeed possible to compute the average case directly from the best and worst cases. The figure below shows how fringe nodes can be paired together. Due to the symmetry of these nodes it is indeed possible to compute the average directly from the best and worst cases. (This idea of pairing is due to Karl Gauss who as a child, used it to compute the sum of an arithmetic series).

Thus, the average case time complexity is:
$\frac{b^d}{2} * \frac{B+W}{b^d} = \frac{B+W}{2}$ Here, $b^d/2$ represents the number of pairs of nodes (each of which is $B+W$ in magnitude, B is the best case time complexity and W is the worst case time complexity. Thus, average time = $\frac{d+1}{2} + \frac{b^{d+1}-1}{2*(b-1)}$. For large values of b and d this is nearly $b^d/2$.

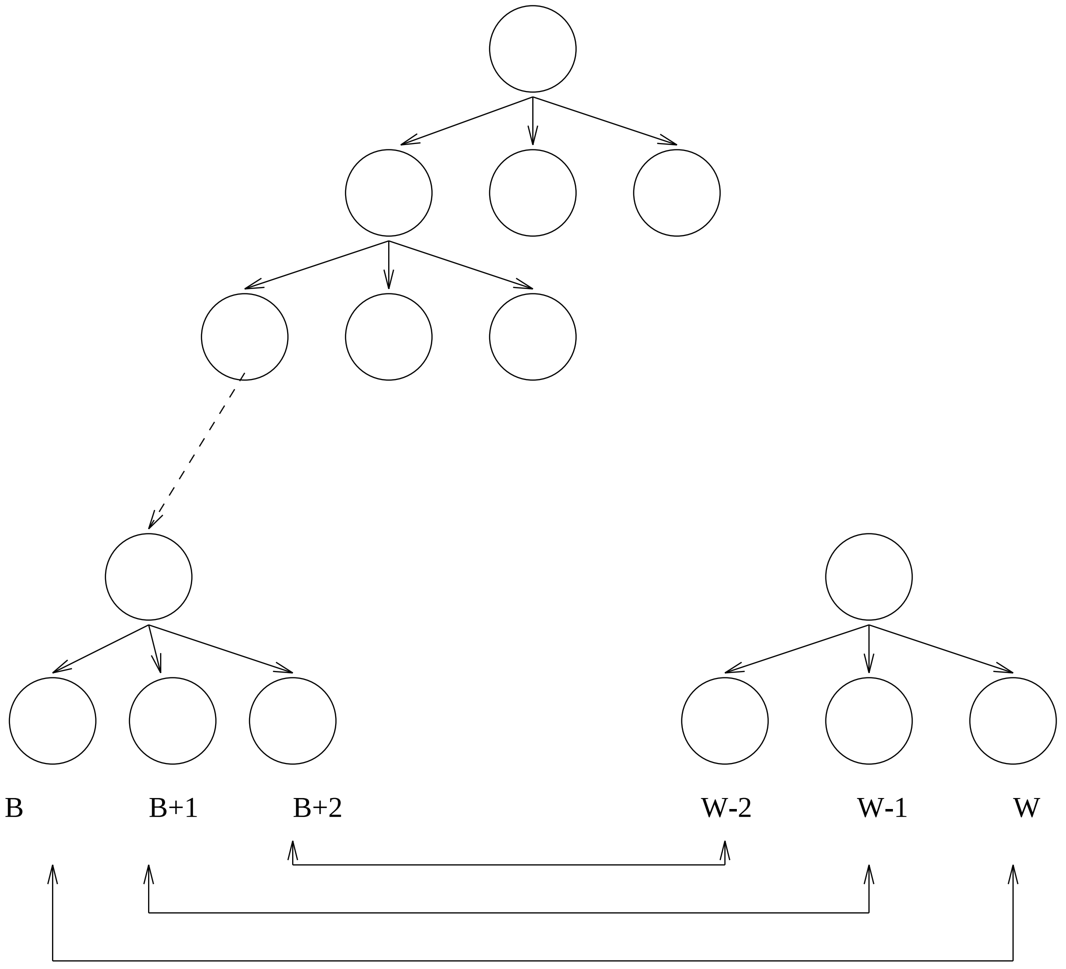

Figure 4: Average Case Time Complexity for DFS

Observations:

- Most of the work is done at depth d at which the goal is located.
- BFS has exponential space requirements but is guaranteed to find the goal if one exists.
  DFS has linear space requirement but is not guaranteed to find the goal (because of potentially infinite search paths). Thus, DFS is not complete.
- DFS is not admissible (because it is not complete).
- Expected case time complexity is the same for both BFS and DFS.

3.3 Iterative Deepening Search

Question:
Can we design a search algorithm that has the same time complexity of BFS and DFS, is guaranteed to find the goal node at depth d (if one exists), and has linear space requirements?

Solution: Iterative Deepening Search (IDS)

1. start at depth cutoff k set to 0.
2. While $k \leq d$ do
   • Conduct DFS to depth k.
   • If goal is not found increment k.
   End while

Space Complexity of IDS $= O(bd). = O(d)$ for constant b
Time Complexity of IDS $= O(b^d/2)$.

Theorem: IDS is an optimal blind search algorithm.

Proof:
Time Complexity — We know that the goal node is at depth d and we must search on an average, $(1 + b^d)/2$ or roughly half the nodes at depth d before we encounter the goal node. So we can do no better with a blind search algorithm under the assumptions made in our analysis.

Space Complexity — Any algorithm that needs T steps to execute needs to be able to at least count up to T (to keep track of which step it is at). This needs $logT$ bits of storage. In our case, the minimum storage needed to count up to b^d is $logb^d = d(logb)$. And we can do no better. (This also shows that the space complexity is linear in d (for a constant b)).
Since we can do no better than $O(b^d)$ time and $O(d)$ space, and IDS does as well as the best that we can expect, it is an *optimal* blind search algorithm in this sense. To do better, we need to consider *informed* search algorithms.

Observations:

- IDS is complete.
- IDS is admissible under the same conditions as BFS.

In many instances, we are interested not only in finding a solution quickly, but also finding a solution of good "quality". We can model the cost of a solution by non-negative costs associated with each operator application. The task is to find an optimal (often the cheapest) solution to the search problem.

3.4 Branch and Bound Search

In looking for the shortest path to a goal node, the simplest method that comes to mind is to enumerate all possible paths and then choose the shortest of those paths that reaches a goal node. However, this is prohibitively costly in terms of computational resources (space, time, or both) needed except in the most trivial of problems. Furthermore, it may be impossible even in principle if the search space being dealt with is infinite. Branch and bound search offers a somewhat better alternative.

The Branch and Bound search (BBS) algorithm maintains a list of paths sorted in ascending order of path costs and it always picks the path at the head of this list, which is the shortest known path. If it terminates in a goal, the path is returned, otherwise it extends the path by one step. The following example shows the list of partial paths that is maintained at each step by the Branch and Bound algorithm.

```
Solution:  The format is (Partial_Path Cost)

  (S 0)
```

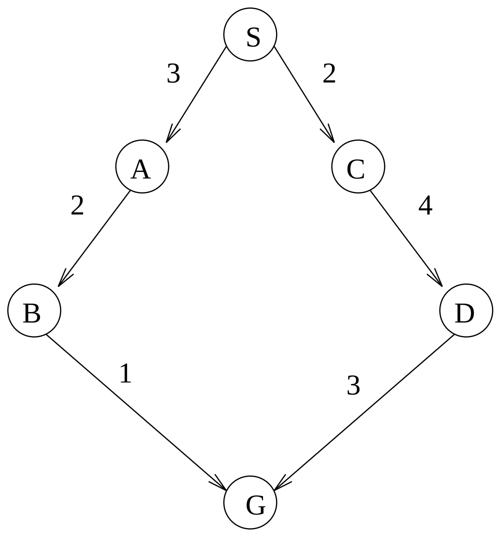

Figure 5: Graph for Branch and Bound Search Example

```
(SC 2) (SA 3)
(SA 3) (SCD 6)
(SAB 5) (SCD 6)
(SABG 6) (SCD 6)  -->  Done
```

Consider another example:

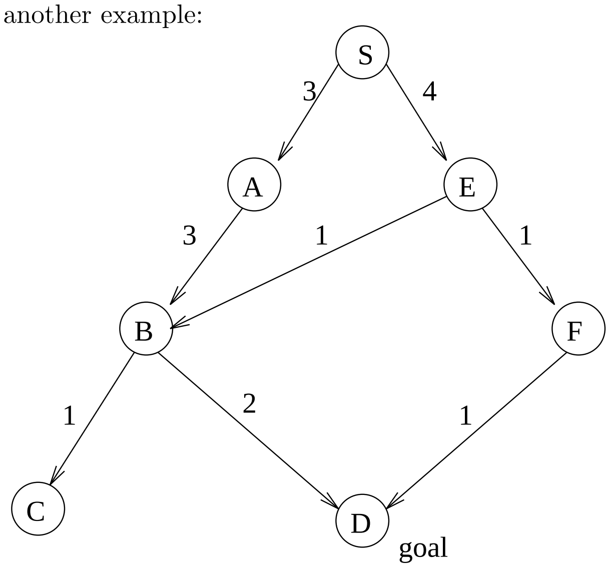

Figure 6: Another Graph for Branch and Bound Search Example

For the graph in Figure 6, write down the state of the list maintained by a branch and bound search assuming the links are unidirectional.

Answer:

```
(S 0)
(SA 3) (SE 4)
(SE 4) (SAB 6)
(SEF 5) (SEB 5) [note: SAB deleted because SEB is a shorter path to B]
(SEB 5) (SEFD 6)
(SEFD 6) (SEBC 6) (SEBD 7) --> Done
```

Formally, BBS uses what is termed as the *Dynamic Programming Principle*. When a shorter path to a given node is found, there is no longer any need to keep track of any longer paths to that node. As an exercise, write down the pseudocode for the branch and bound algorithm that uses dynamic programming along with the detection and elimination of cycles (see the Winston text for a solution).

Definition: Dynamic Programming Principle

$min\_cost(s, g) = min\_cost(s, i) + min\_cost(i, g)$

It is worth pointing out that dynamic programming principle (DPP), although generally quite useful whenever it is applicable, does not always apply. Examples of problems where DPP may not be applicable include sight seeing trips (your decision as to where to go next may depend on the places you have visited along the path to current location) and problems that involve accumulation as well as depletion of resources (e.g., the need for fuel along the way might influence you to choose a longer path just so that you can fill up your tank for the rest of the trip). Another such problem in which the dynamic programming principle is not applicable is in a case when there are negative cost edges. We will assume that the cost associated with each edge is non-negative.

Theorem: Branch and Bound search is admissible.

Note that Branch and Bound search is guaranteed to find an optimal path to a goal (if a path exists) even if the search space is infinite (so long as the branching factor at each node is finite).

Observations:

- BBS is complete.
- BBS is admissible.

BBS is still a blind search algorithm (in other words, it makes use of no additional information besides what it has accumulated in terms of partial path costs at any step during the search). Can we do better? In order to answer this question, we have to consider informed search algorithms. In particular, we will examine heuristic search.

4 Implementation of State-Space Search

```
State-Space Search Algorithm

function general-search(problem, QUEUEING-FUNCTION)
  ;; problem describes the start state, operators, goal test, and
  ;;   operator costs
  ;; queueing-function is a comparator function that ranks two states
  ;; general-search returns either a goal node or "failure"

  nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
  loop
     if EMPTY(nodes) then return "failure"
     node = REMOVE-FRONT(nodes)
     if problem.GOAL-TEST(node.STATE) succeeds
        then return node
     nodes = QUEUEING-FUNCTION(nodes, EXPAND(node, problem.OPERATORS))
     ;; Note: The goal test is NOT done when nodes are generated
     ;; Note: This algorithm does not detect loops
  end
```

Key Procedures to be Defined are:

- EXPAND: Generate all successor nodes of a given node
- GOAL-TEST: Test if state satisfies all goal conditions
- QUEUING-FUNCTION: Used to maintain a ranked list of nodes that are candidates for expansion

Some details that need to be attended to include:

- Search process constructs a search tree, where root is the initial state and all of the leaf nodes are nodes that have not yet been expanded (i.e., they are in the list "nodes") or are nodes that have no successors (i.e., they're "dead-ends" because no operators were applicable and yet they are not goals)
- Search tree may be infinite because of loops even if state space is small
- Return a path or a node depending on problem. E.g., in cryptarithmetic return a node; in 8-puzzle return a path
- Changing definition of the QUEUING-FUNCTION leads to different search strategies

Since blind search can be prohibitively costly in terms of computational requirements, much work in AI has focused on principles (general as well as problem-specific) that help constrain the search without sacrificing completeness, admissibility, optimality whenever possible or by compromising on completeness, admissibility, and optimality in precisely quantifiable ways. Such heuristic search techniques are discussed in the next chapter.