SEARCH TECHNIQUES

Defining the Problem

- Much of the point of the AI research was to understand “how” to solve the problem, not just to get a solution.
- How do you define the problem with enough precision so that you can figure out how to represent it?

- we are interested in how to represent our problem so that we can solve it using search techniques,
- and which search techniques to use.
- we explore one of the primary problem representations, the state-space approach.

State Space

- The combination of the initial state and the set of operators make up the *state space* of the problem.
- The sequence of states produced by the valid application of operators from the initial state is called the *path* in the state space.
- In many search problems, we are not only interested in reaching a goal state, we would like to reach it with the lowest possible cost (or the maximum profit).

- An algorithm is *optimal* if it will find the best solution from among several possible solutions.
- A strategy is *complete* if it guarantees that it will find a solution if one exists.

The complexity of the algorithm,

-*time complexity* (how long it takes to find a solution)

-*space complexity* (how much memory it requires), is a major practical consideration.

The Blind Techniques:

If we have a systematic search strategy that does not use information about the problem to help direct the search, it is called *brute-force, uninformed*, or *blind* search.

- The Breadth-First Search
- The Depth-First Search
- Improving Depth-First Search

Heuristic search methods

Search algorithms which use information about the problem, such as the cost or distance to the goal state, are called heuristic, informed, or directed search.

- Best-First Search
- Hill climbing Technique
- Greedy Search
- A* Search

The Breadth-*First* Search

The breadth-first search algorithm searches a state-space by constructing a hierarchical tree structure consisting of a set of nodes and links. The algorithm defines a way to move through the tree structure, examining the values at nodes in a controlled and systematic way so that we can find a node which offers a solution to the problem we have represented using the tree-structure.

- 1. create a queue and add the first search node to it
- 2. Loop:
  - If the queue is empty; quit.
  - Remove the first Search Node from the queue.
  - If the Search Node contains the goal state, then exit with the Search Node as the solution.
  - For each child of the current Search Node: Add the new state to the back of the queue.

The breadth-first algorithm

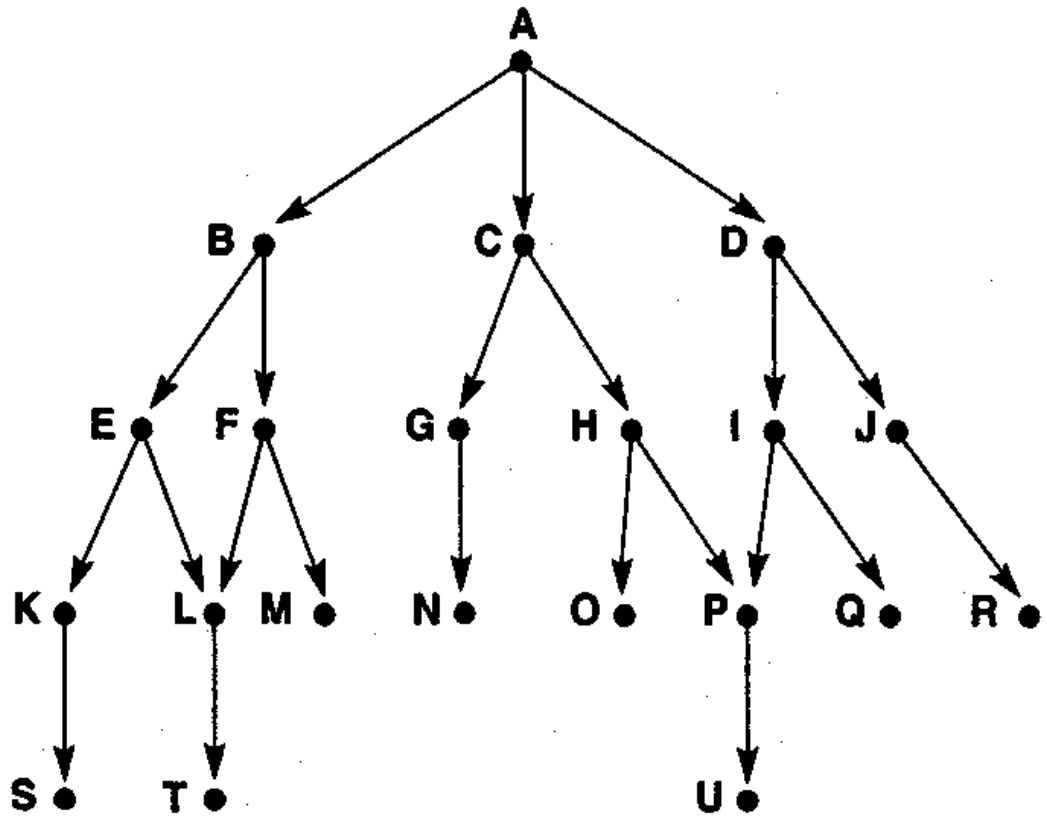
The breadth-first algorithm spreads out in a uniform manner from the start node. From the start, it looks at each node one edge away. Then it moves out from those nodes to all nodes two edges away from the Start. This continues until either the goal node is found or the entire tree is searched, Breadth-first search is *complete*; it will find a solution if one exists. But it is *neither optimal* in the general case (it won't find the best solution, just the first one that matches the goal state). *nor does it have good time or space complexity* (it grows exponentially in time and memory consumption).

The Depth-First Technique

- **Depth-First Search**
  Depth-first search is another way to systematically traverse a tree structure to find a goal or solution node. Instead of completely searching each level of the tree before going deeper, the depth-first algorithm follows a single branch of the tree down as many levels as possible until we either reach a solution or a dead end. The algorithm follows:
- 1. create a queue and add the first search node to it
- 2. Loop:
  - If the queue is empty; quit.
  - Remove the first Search Node from the queue.
  - If the Search Node contains the goal state, then exit with the Search Node as the solution.
  - For each child of the current Search Node: Add the new state to the front of the queue.

The Depth-First Technique

- Notice that this algorithm is identical to the breadth-first search with the exception of step 2d. The depth-first algorithm searches from the start or root node all the way down to a leaf node. If it does not find the goal node, it backtracks up the tree and searches down the next untested path until it reaches the next leaf. If you imagine a large tree, the depth-first algorithm may spend a large amount of time searching the paths on the lower left when the answer is really in the lower right. But since depth-first search is a brute-force method, it will blindly follow this search pattern until it comes across a node containing the goal state, or it searches the entire tree Depth first search has lower memory requirements than breadth first search, but it is neither complete nor optimal.


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U

Improving Depth-First Search

- **Improving Depth-First Search**
  One easy way to get the best characteristics of the depth-first search algorithm along with the advantages of the breadth-first search is to use a technique called *iterative-deepening search*. In this approach.

Improving Depth-First Search

This insight leads to a search algorithm that remedies many of the drawbacks of both depth-first and breadth-first search. *Depth-first iterative deepening* (Korf 1987) performs a depth-first search of the space with a depth bound of 1. If it fails to find a goal, it performs another depth-first search with a depth bound of 2. This continues, increasing the depth bound by one at each iteration. At each iteration, the algorithm performs a complete depth-first search to the current depth bound. No information about the state space is retained between iterations.

Because the algorithm searches the space in a level-by-level fashion, it is guaranteed

Improving Depth-First Search

- This algorithm, like standard breadth-first search, is a complete search and will find an optimal solution, but it has much lower memory requirements, Like the depth-first algorithm- Although we are retracing ground when we increase our depth of search, this approach is still more efficient than pure breadth-fast or pure unlimited depth-first search for large search spaces .

Heuristic search methods

- Heuristic search methods are characterized by this sense that we have *limited time* and *space* in which to find an answer to *complex problems* and so we are willing to accept a *good solution*
- Heuristic search methods use objective functions called (surprise!) *heuristic functions* to *try to gauge the value of a particular node in the search tree and to estimate the value of any of the paths from the node.* In the next sections we describe four types of heuristic search algorithms.

Generate and Test

- Generate and Test
- The generate and test algorithm is the most basic heuristic search function. The steps are:
  1. Generate a possible solution, either a new state or a path through the problem space.
  2. Test to see if the new state or path is a solution by comparing it to a set of goal states.
  3. if a solution has been found, return success: else return to step1.

Generate and Test

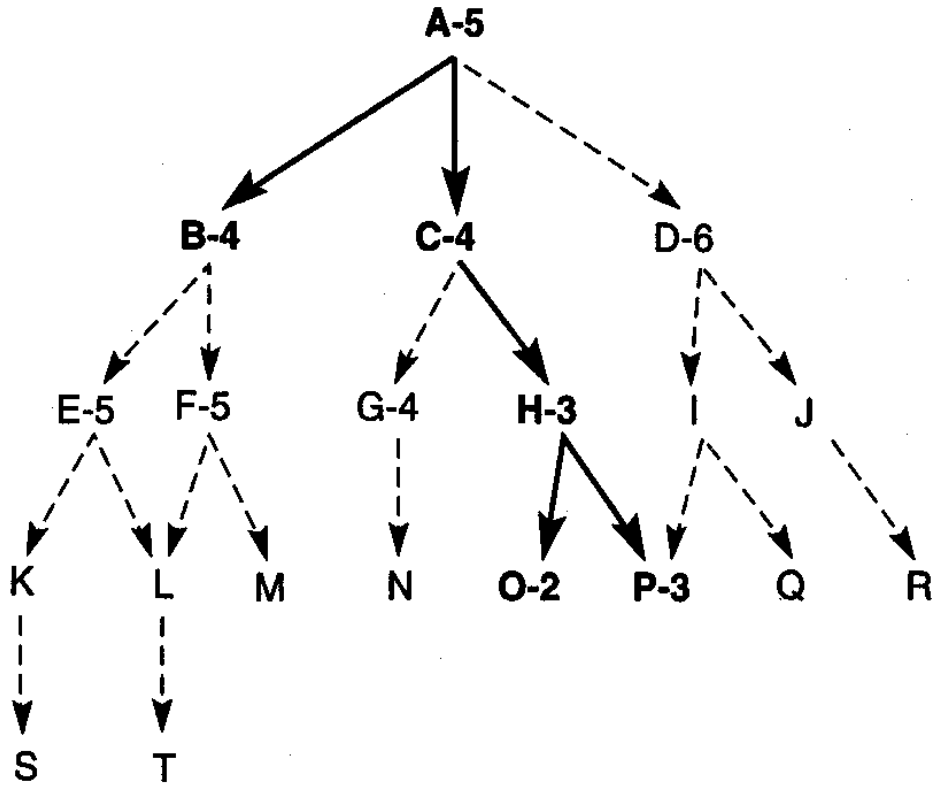
- This is a depth-first search procedure which performs an exhaustive search of the state space. If a solution is possible, the generate and test algorithm will find it, However, it may take an extremely long time. For small problems. generate and test can be an effective algorithm, but for large problems, *the undirected search strategy leads to lengthy run times and is impractical*. The major weakness of generate and test is that we get no feedback on which direction to search. We can greatly improve this algorithm by providing feedback through the use of heuristic functions.

Hill climbing Technique

- Hill climbing is an improved generate-and-test algorithm, where feedback from the tests are used to help *direct the generation (and evaluation)* of new candidate states, When a node state is evaluated by the goal test function, a measure or estimate of the distance to the goal state is also computed. One problem with hill climbing search in general is that the algorithm can get caught in *local minima or maxima.*

Best-First Search

- Best-first search is a systematic control strategy combining the strengths of breadth-first and depth-first search into one algorithm. The main difference between best-first search and the blind search techniques is that we make use of an evaluation or heuristic function to order on the queue. In this way we choose the SearchNode that appears to be best, before any others, regardless of their position in the tree or graph .


A-5
B-4
C-4
D-6
E-5
F-5
G-4
H-3
I
J
K
L
M
N
O-2
P-3
Q
R
S
T

Greedy Search

- Greedy search is a best-first strategy where we try to minimize estimated cost to reach the goal (certainly an intuitive approach!). Since we are greedy always expand the node that is estimated to be closest to the goal state, Unfortunately the exact cost of reaching the goal state usually can't be computed, but we can estimate it by using a cost estimate or heuristic function *h().* When we are examining node n, then h() gives us the estimated cost of the cheapest path from n's state to the goal state. Of course the better an estimate *h()* gives, the better and faster we will find a solution to our problem Greedy search has similar behavior to depth-first search. Its advantages are delivered via the use of a quality heuristic function to direct the search.

A* Search

- .
  A* Search One of the most famous search algorithms used in Al is the A* search algorithm, which combines the greedy search algorithm for efficiency with the uniform cost search for optimality and completeness. In A* the evaluation function is computed by the two heuristic measures; the h(n) cost estimate of traversing from n to the goal state g(n) which is the known path cost from the start node to n into a function called f(n) This combination of strategies turns out to provide A* with both completeness and optimality

A* search

mining the properties of admissible heuristics, we define an evaluation function f*:

$$f^*(n) = g^*(n) + h^*(n)$$

where $g^*(n)$ is the cost of the *shortest* path from the start node to node n and h^* returns the *actual* cost of the shortest path from n to the goal. It follows that $f^*(n)$ is the actual cost of the optimal path from a start node to a goal node that passes through node n.

Means-Ends Analysis

- Means-ends analysis is a process for problem solving which is based on detecting differences between states and then trying to reduce those differences. First used in the General Problem Solver . means-ends analysis uses both forward and backward reasoning and a recursive algorithm to systematically minimize the differences between the initial and goal states.

Means-Ends Analysis

- Means-Ends analysis (Current-State, Goal-State)
  1. Compare the current-state to the goal-state. If states are identical then return success.
  2. Select the most important difference and reduce it by performing the following steps until success or failure
  a. Select an operator that is applicable to the current difference. If there are no operators which can be applied, then return failure.
  b. Attempt to apply the operator to the current state by generating two temporary states, one where the operator's preconditions are true (prestate). and one that would be the result if the operator were applied to the current state (poststate).
  c. Divide the problem into two parts, a FIRST part, from the current-state to the prestate, and a LAST part, from the poststate to the goal state. Call means-ends analysis to solve both pieces. If both are true, then return success, with the solution consisting of the FIRST part, the selected operator, and the LAST part.