UNIT-2
PROBLEM SOLVING
HEURISTIC SEARCH STRATEGIES:

One of the core methods AI systems use to navigate problem-solving is through heuristic search techniques. These techniques are essential for tasks that involve finding the best path from a starting point to a goal state, such as in navigation systems, game playing, and optimization problems.

Uninformed Search Algorithms

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

1. Breadth-first Search
2. Depth-first Search
3. Depth-limited Search
4. Iterative deepening depth-first search
5. Uniform cost search
6. Bidirectional Search

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.

- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

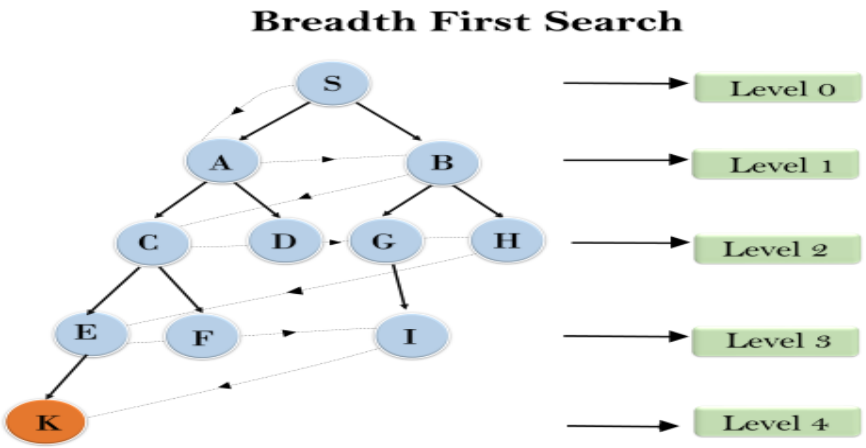
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

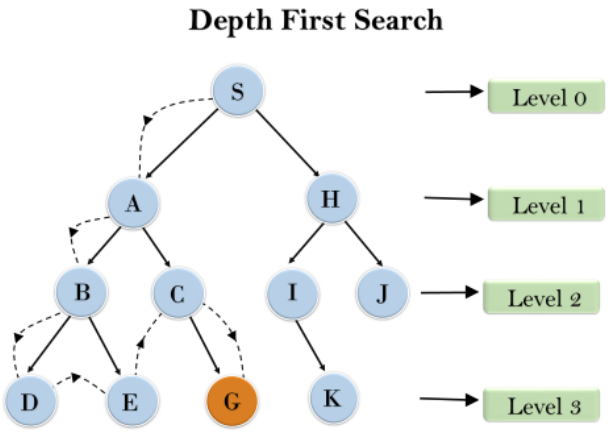
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

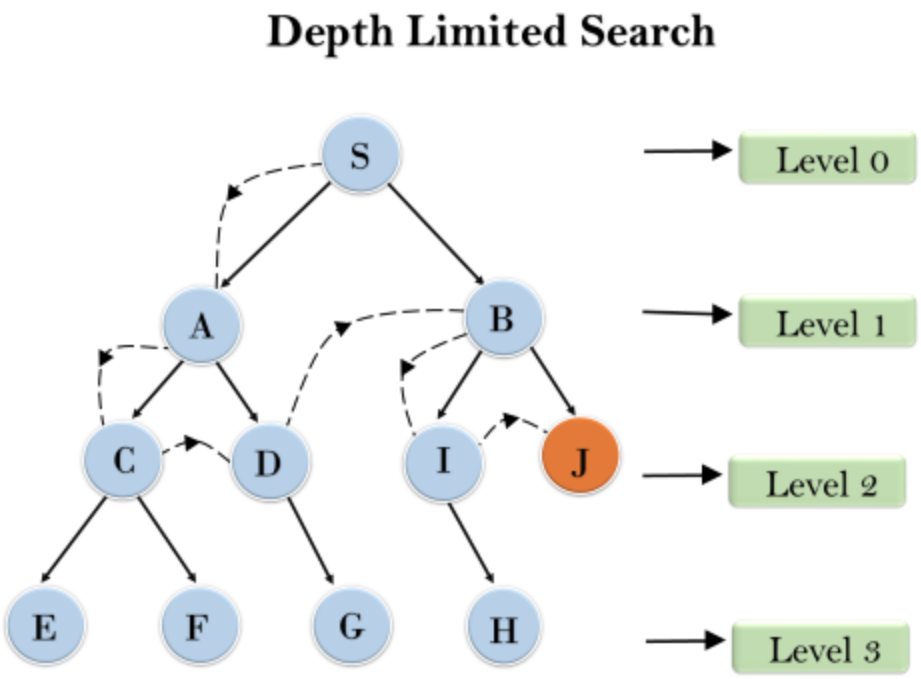
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is $O(b^{\ell})$.

Space Complexity: Space complexity of DLS algorithm is $O(b \times \ell)$.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

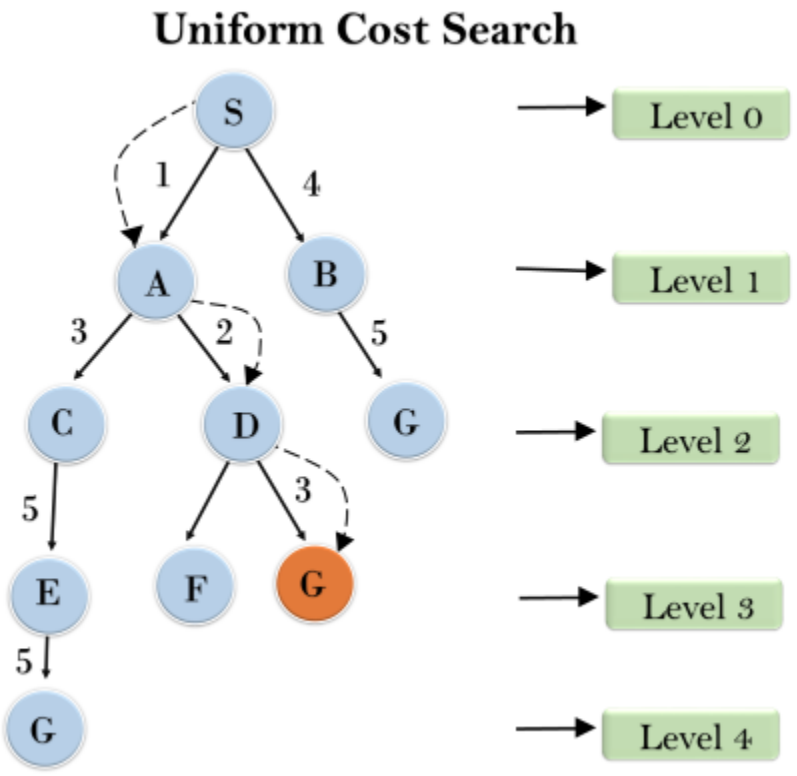
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Time Complexity:

Let C* is Cost of the optimal solution, and ε is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is$O(b^{1 + [C^*/\varepsilon]})$/.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^{1 + [C^*/\varepsilon]})$.

5. Iterative deepening depth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

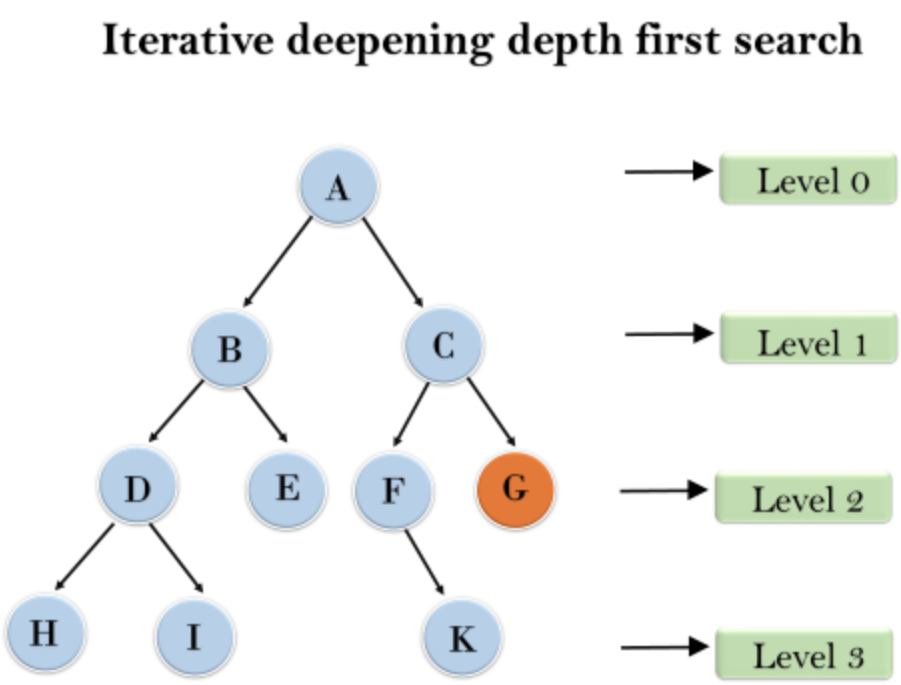
- Itcombines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $O(b^d)$.

Space Complexity:

The space complexity of IDDFS will be O(bd).

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Informed Heuristic Functions and Advanced Search Algorithms

Introduction to Informed Heuristic Functions

In AI, heuristic functions play a critical role in guiding search algorithms towards efficient solutions. An informed heuristic function provides an estimate of the cost from a current state to the goal, aiding algorithms in making informed decisions about which paths to explore.

Properties of Heuristic Functions

- Admissibility: A heuristic function h(n)h(n)h(n) is admissible if it never overestimates the true cost to reach the goal from state nnn.
- Consistency (Monotonicity): For every state nnn and its successor mmm, the estimated cost h(n)h(n)h(n) should be less than or equal to the cost of reaching mmm plus the estimated cost from mmm to the goal, ensuring consistency in estimates.

Greedy Best-First Search

- Concept: Greedy Best-First Search (GBFS) is an informed search algorithm that selects the node which appears closest to the goal based solely on the heuristic function h(n)h(n)h(n).

- Strategy: At each step, GBFS expands the node with the lowest h(n)h(n)h(n) value, making it greedy in nature as it prioritizes immediate gains based on the heuristic.
- Limitations: GBFS does not guarantee optimality since it may get stuck in local optima based on the heuristic estimates alone, without considering the actual path cost.

A* Search Algorithm

- Overview: A* is a widely-used informed search algorithm that combines the actual path cost g(n)g(n)g(n) from the start node to node nnn with the heuristic function h(n)h(n)h(n).
- Optimality: A* is both complete (finds a solution if one exists) and optimal (finds the shortest path) when the heuristic function h(n)h(n)h(n) is admissible.
- F-function: The total cost function f(n)f(n)f(n) in A* is defined as f(n)=g(n)+h(n)f(n) = g(n) + h(n)f(n)=g(n)+h(n), guiding the algorithm to expand nodes based on their combined path cost and heuristic estimate.

Memory-Bounded Search

- Challenge: In scenarios with limited memory resources, traditional search algorithms like A* may face challenges due to the large number of nodes stored in memory.
- Memory-Bounded Heuristic Search: Techniques such as Memory-Bounded A* (MA*) address this by limiting the number of nodes stored in memory, focusing resources on nodes most likely to lead to the goal.
- Trade-offs: MA* trades off optimality for memory efficiency, ensuring that it operates within predefined memory constraints while attempting to find a satisfactory solution.

Learning to Search Better

- Machine Learning Integration: Recent advancements in AI have explored integrating machine learning techniques to enhance heuristic functions and search algorithms.
- Benefits: Learning-based approaches can adaptively improve heuristic estimates based on experience or training data, potentially leading to more accurate and efficient search strategies.

- Applications: Learning to search better can be applied in domains where heuristic functions are challenging to define manually, such as complex decision-making processes or dynamic environments.

HEURISTIC FUNCTIONS:

What are Heuristic Functions?

Heuristic functions are strategies or methods that guide the search process in AI algorithms by providing estimates of the most promising path to a solution. They are often used in scenarios where finding an exact solution is computationally infeasible. Instead, heuristics provide a practical approach by narrowing down the search space, leading to faster and more efficient problem-solving.

Heuristic functions transform complex problems into more manageable subproblems by providing estimates that guide the search process. This approach is particularly effective in AI planning, where the goal is to sequence actions that lead to a desired outcome.

Search Algorithm

Search algorithms are fundamental to AI, enabling systems to navigate through problem spaces to find solutions. These algorithms can be classified into uninformed (blind) and informed (heuristic) searches. Uninformed search algorithms, such as breadth-first and depth-first search, do not have additional information about the goal state beyond the problem definition. In contrast, informed search algorithms use heuristic functions to estimate the cost of reaching the goal, significantly improving search efficiency.

Heuristic Search Algorithm in AI

Heuristic search algorithms leverage heuristic functions to make more intelligent decisions during the search process. Some common heuristic search algorithms include:

A* Algorithm

The A* algorithm is one of the most widely used heuristic search algorithms. It uses both the actual cost from the start node to the current node (g(n)) and the estimated cost from the current node to the goal (h(n)). The total estimated cost (f(n)) is the sum of these two values:

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

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

Greedy Best-First Search

The Greedy Best-First Search algorithm selects the path that appears to be the most promising based on the heuristic function alone. It prioritizes nodes with the lowest heuristic cost (h(n)), but it does not necessarily guarantee the shortest path to the goal.

Hill-Climbing Algorithm

The Hill-Climbing algorithm is a local search algorithm that continuously moves towards the neighbor with the lowest heuristic cost. It resembles climbing uphill towards the goal but can get stuck in local optima.

Role of Heuristic Functions in AI

Heuristic functions are essential in AI for several reasons:

- Efficiency: They reduce the search space, leading to faster solution times.
- Guidance: They provide a sense of direction in large problem spaces, avoiding unnecessary exploration.
- Practicality: They offer practical solutions in situations where exact methods are computationally prohibitive.

Common Problem Types for Heuristic Functions

Heuristic functions are particularly useful in various problem types, including:

1. Pathfinding Problems: Pathfinding problems, such as navigating a maze or finding the shortest route on a map, benefit greatly from heuristic functions that estimate the distance to the goal.
2. Constraint Satisfaction Problems: In constraint satisfaction problems, such as scheduling and puzzle-solving, heuristics help in selecting the most promising variables and values to explore.
3. Optimization Problems: Optimization problems, like the traveling salesman problem, use heuristics to find near-optimal solutions within a reasonable time frame.

8 puzzle problem:
/https://www.gatevidyalay.com/tag/a-algorithm-for-8-puzzle-problem/

Local Search Techniques and Optimization in AI

1. Hill-Climbing

- Concept: Hill-climbing is a simple heuristic search algorithm that continually moves towards increasing the value of the objective function (or minimizing cost), by making small adjustments to the current solution.
- Mechanism: It evaluates the neighboring solutions (or states) and moves to the best neighboring solution that improves the objective function value, provided it is better than the current state.
- Issues: Hill-climbing can get stuck in local optima (suboptimal solutions) or plateaus (flat regions) where no better neighboring solution exists, leading to premature convergence and inability to escape local minima.

2. Gradient Methods

- Concept: Gradient-based optimization methods use the gradient (or derivative) of the objective function to guide the search towards the direction of steepest descent (minimization) or ascent (maximization).
- Mechanism: It iteratively adjusts the parameters (variables) of the solution based on the gradient of the objective function until convergence criteria are met.
- Issues: Gradient methods require the objective function to be differentiable and continuous. They can converge to local optima and struggle with non-convex or noisy objective functions.

3. Simulated Annealing

- Concept: Simulated annealing is a probabilistic technique inspired by the annealing process in metallurgy, where a material is cooled slowly to reach a low-energy state.
- Mechanism: It explores the search space by allowing moves to solutions of higher energy (worse solutions) with a probability that decreases over time, based on a temperature parameter.

- Benefits: Simulated annealing can escape local optima by occasionally accepting worse solutions, providing a better chance of finding a global optimum in complex and rugged landscapes.
- Issues: It requires tuning parameters such as temperature schedule and acceptance probability, and the method's performance heavily depends on these choices.

4. Genetic Algorithms

- Concept: Genetic algorithms (GA) are inspired by the process of natural selection and genetics. They use populations of candidate solutions (individuals) and evolutionary operators to iteratively improve solutions.
- Mechanism: GA involves selection, crossover (recombination of solutions), and mutation (introducing random changes) operations to evolve the population towards fitter solutions.
- Benefits: Genetic algorithms are effective for exploring large search spaces and can handle non-linear and non-differentiable objective functions. They are robust against local optima.
- Issues: GA can be computationally expensive due to the evaluation of a large number of solutions and parameter tuning. They may converge prematurely or struggle with high-dimensional problems.

Issues with Local Search

- Local Optima: Local search techniques are prone to getting stuck in local optima, where the algorithm cannot find a better solution within its neighborhood.
- Plateaus: Flat regions in the search space where the objective function does not change significantly, leading to slow progress or stagnation in the search.
- Premature Convergence: Techniques like hill-climbing or gradient descent may converge to a suboptimal solution without exploring the entire search space adequately.
- Exploration vs. Exploitation: Balancing between exploring new regions of the search space (diversification) and exploiting known solutions (intensification) is crucial for effective optimization.

LOCAL SEARCH IN CONTINUOUS SPACE:

Local Search in Continuous Spaces

As we know that environment can be descrete or continuous, but the most real-world environment are continuous.

1) It is very difficult to handle continuous state space. The successor for real-world problem are many infinite states.

2) Origin of local search in continuous spaces lies in Newton and Leibnitz in the 17th century.

Search and Evaluation Theory

1) Search is basic problem solving technique. Basically it is always related with evolution theory.

2) Charles Darwin is father of evolutionary theory. The theory was based on the origin of species by means of natural selection (1859).

3) The variations (mutations) are well known attributes of reproduction and best features are preserved in next generation in proper propagation.

4) The qualities are inherited or modified. This fact was not associated with Darwin theory.

5) Gregor Mendel (1866) theory found the fact of inheritance. He performed artificial fertilization on peas.

6) DNA module structure was identified by Watson and Crick.

A → Adenin, G- Guanine,

T→ Thymine, C - Cytosine

7) The key difference between stochastic beam search and evolution is that successors are generated from multiple organisms (states) rather than one organism (state).

8) The theory of evolution is very much rich than genetic algorithm.

9) The process of mutations involves duplication, reversals and motion of large group of DNA.

10) Most important is the fact that the genes themselves encode the mechanisms whereby the genome is reproduced and translated into an organism. In genetic algorithms, those mechanisms are a separate program that is not represented within the strings being manipulated.

11) French naturalist Jean Lamarck (1809) proposed a theory of evolution whereby traits acquired by adaptation during an organism's lifetime would be passed on to its offspring. Such a process would be effective, but does not seem to occur in nature.

12) James Baldwin (1896) proposed a superficially similar theory : - that behavior learned during an organism's lifetime could accelerate the rate of evolution.

For example -

Suppose we want to place three new airports anywhere in India, such that the sum of squared distances from each city to its nearest airport is minimized.

i) The state space, is defined by the co-ordinates of the airports: (x_1, y_1), (x_2, y_2) and (x_3, y_3).

ii) This is a six-dimensional space: We also say that states are defined by six variables. (In general, states are defined by an n-dimensional vector of variables, x).

iii) Moving around in this space corresponds to moving one or more of the airports on the map.

iv) The objective function f $(x_1, y_1, x_2, y_2, x_3, y_3)$ is relatively easy to compute for any particular state, once we compute the closest cities, but rather tricky to write down in general.

Problems Associated with Local Search

• Local search methods suffer from local maxima, ridges and plateaux in continuous state spaces just as much as in discrete spaces. Random restarts and simulated annealing can be used and are often helpful. High-dimensional continuous spaces are, however, big places in which it is easy to get lost.

Constrained Optimization Problems

• An optimization problem is constrained if solutions must satisfy some hard constraint like sites to be inside India and on dry land (rather than in the middle of rivers). The difficulty of constrained optimization problems depends on the nature of the constraints and the objective function. The best-known category is that of linear programming problems, in which constraints must be linear inequalities forming a convex region and the objective function is also linear. Linear programming problems can be solved in time which is polynomial in the number of variables.

Searching with nondeterministic actions

When the environment is either partially observable or nondeterministic (or both), the future percepts cannot be determined in advance, and the agent's future actions will depend on those future percepts.

Nondeterministic problems:

Transition model is defined by RESULTS function that returns a set of possible outcome states;

Solution is not a sequence but a contingency plan (strategy),

e.g.

```
[Suck, if State = 5 then [Right, Suck] else []];
```

In nondeterministic environments, agents can apply AND-OR search to generate contingent plans that reach the goal regardless of which outcomes occur during execution.

AND-OR search trees

OR nodes: In a deterministic environment, the only branching is introduced by the agent's own choices in each state, we call these nodes OR nodes.

AND nodes: In a nondeterministic environment, branching is also introduced by the environment's choice of outcome for each action, we call these nodes AND nodes.

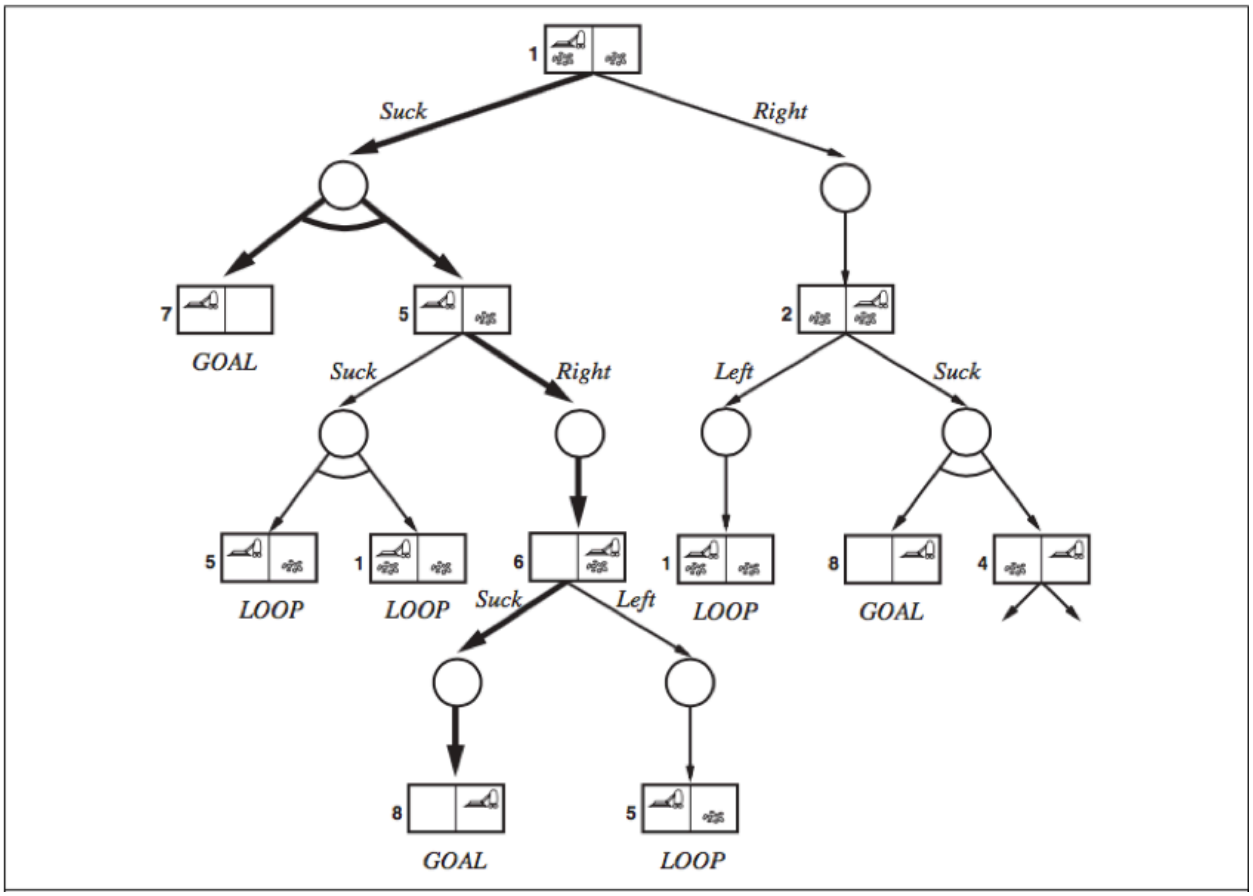
AND-OR tree: OR nodes and AND nodes alternate. States nodes are OR nodes where some action must be chosen. At the AND nodes (shown as circles), every outcome must be handled.

A solution (shown in bold lines) for an AND-OR search problem is a subtree that

1) has a goal node at every leaf;

2) specifies one action at each of its OR nodes;

3) includes every outcome branch at each of its AND nodes.



A recursive, depth-first algorithm for AND-OR graph search

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 4.11 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation $[x \mid l]$ refers to the list formed by adding object x to the front of list l.)

Cyclic solution

Cyclic solution: keep trying until it works. We can express this solution by adding a label to denote some portion of the plan and using the label later instead of repeating the plan itself. E.g.: [Suck, L1: Right, if State = 5 then L1 else Suck]. (or “while State = 5 do Right”).

Searching with partial observatioins

Belief state: The agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point.

Standard search algorithms can be applied directly to belief-state space to solve sensorless problems, and belief-state AND-OR search can solve general partially observable problems. Incremental algorithms that construct solutions state-by-state within a belief state are often more efficient.

1. Searching with no observation

When the agent's percepts provide no information at all, we have a sensorless problem.

To solve sensorless problems, we search in the space of belief states rather than physical states. In belief-state space, the problem is fully observable, the solution is always a sequence of actions.

Belief-state problem can be defined by: (The underlying physical problem P is defined by $ACTIONS_P$, $RESULT_P$, $GOAL\text{-}TEST_P$ and $STEP\text{-}COST_P$)

·Belief states: Contains every possible set of physical states. If P has N states, the sensorless problem has up to 2^N states (although many may be unreachable from the initial state).

·Initial states: Typically the set of all states in P.

·Actions:

a. If illegal actions have no effect on the environment, take the union of all the actions in any of the physical states in the current belief b:

ACTIONS(b) = $\bigcup_{s \in b} RESULTS_P(s, a)$

b. If illegal actions are extremely dangerous, take the intersection.

·Transition model:

a. For deterministic actions,

b' = RESULT(b,a) = {s' : s' = $RESULT_P$(s,a) and s∈b}. (b' is never larger than b).

b. For nondeterministic actions,

b' = RESULT(b,a) = {s' : s' = $RESULT_P$(s,a) and s∈b} = (b' may be larger than b)

The process of generating the new belief state after the action is called the prediction step.

·Goal test: A belief state satisfies the goal only if all the physical states in it satisfy $GOAL\text{-}TEST_P$.

·Path cost

If an action sequence is a solution for a belief state b, it is also a solution for any subset of b. Hence, we can discard a path reaching the superset if the subset has already been generated. Conversely, if the superset has already been generated and found to be solvable, then any subset is guaranteed to be solvable.

Main difficulty: The size of each belief state.

Solution:

a. Represent the belief state by some more compact description;

b. Avoid the standard search algorithm, develop incremental belief state search algorithms instead.

Incremental belief-state search: Find one solution that works for all the states, typically able to detect failure quickly.

2. Searching with observations

When observations are partial, The ACTIONS, STEP-COST, and GOAL-TEST are constructed form the underlying physical problem just as for sensorless problems.

·Transition model: We can think of transitions from one belief state to the next for a particular action as occurring in 3 stages.

The prediction stage is the same as for sensorless problem, given the action a in belief state b, the predicted belief state is $\hat{b}$ = PREDICT(b,a);

The observation prediction stage determines the set of percepts o that could be observed in the predicted belief state:

$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\};$$

The update stage determines, for each possible percept, the belief state that would result from the percept. The new belief state b_o is the set of states in $\hat{b}$ that could have produced the percept:

$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\}$

(b_o can be no larger than the predicted belief state $\hat{b}$)

In conclusion:

RESULTS(b,a) = { b_o : b_o = UPDATE(PREDICT(b,a),o) and o∈POSSIBLE-PERCEPTS(PREDICT(b,a))}

Search algorithm return a conditional plan that test the belief state rather than the actual state.

Agent for partially observable environments is similar to the simple problem-solving agent (formulates a problem, calls a search algorithm, executes the solution).

Main difference:

1) The solution will be a conditional plan rather than a sequence.

2) The agent will need to maintain its belief state as it performs actions and receives percepts.

Given an initial state b, an action a, and a percept o, the new belief state is

b' = UPDATE(PREDICT(b, a), o). //recursive state estimator

Sate estimation: a.k.a. monitoring or filtering, a core function of intelligent system in partially observable environments—maintaining one's belief state.

Online search Agents

Online search is a necessary idea for unknown environments. Online search agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action.

1. Online search problem

Assume a deterministic and fully observable environment, the agent only knows:

·ACTION(s): returns a list of actions allowed in state s;

·c(s, a, s'): The step-cost function, cannot be used until the agent knows that s' is the outcome;

·GOAL-TEST(s).

·The agent cannot determine RESULT(s, a) except by actually being in s and doing a.

·The agent might have access to an admissible heuristic function h(s) that estimates the distance from the current state to a goal state.

Competitive ratio: The cost (the total path cost of the path that the agent actually travels) / the actual shortest path (the path cost of the path the agent would follow if it knew the search space in advance). The competitive ratio is expected to be as small as possible.

In some case the best achievable competitive ratio is infinite, e.g. some actions are irreversible and might reach a dead-end state. No algorithm can avoid dead ends in all state spaces.

Safely explorable: some goal state is reachable from every reachable state. E.g. state spaces with reversible actions such as mazes and 8-puzzles.

No bounded competitive ratio can be guaranteed even in safely explorable environments if there are paths of unbounded cost.

2. Online search agents

```
function ONLINE-DFS-AGENT(s′) returns an action
  inputs: s′, a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s′) then return stop
  if s′ is a new state (not in untried) then untried[s′] ← ACTIONS(s′)
  if s is not null then
      result[s, a] ← s′
      add s to the front of unbacktracked[s′]
  if untried[s′] is empty then
      if unbacktracked[s′] is empty then return stop
      else a ← an action b such that result[s′, b] = POP(unbacktracked[s′])
  else a ← POP(untried[s′])
  s ← s′
  return a
```

Figure 4.21 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action.

ONLINE-DFS-AGENT works only in state spaces where the actions are reversible.

RESULT: a table the agent stores its map, RESULT[s, a] records the state resulting from executing action a in state s.

Whenever an action from the current sate has not been explored, the agent tries that action.

When the agent has tried all the actions in a state, the agent in an online search backtracks physically (in a depth-first search, means going back to the state from which the agent most recently entered the current sate).

3. Online local search

Exploration problems arise when the agent has no idea about the satate and actions of its environment. For safely explorable environments, online search agents can build a map and find a goal if one exists. Updating heuristic estimates from experience provides an effective method to escape from local minima.

Random walk: Because online hill-climbing search cannot use restart(because the agent cannot transport itself to a new state), can use random walk instead. A random walk simply selects at random one of the available actions from the current state, preference can be given to actions that have not yet been tried.

Basic idea of online agent: Random walk will eventually find a goal or complete its exploration if the space is finite, but can be very slow. A more effective approach is to store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited. H(s) starts out being the heuristic estimate h(s) and is updated as the agent gains experience in the state space.

If the agent is stuck in a flat local minimum, the agent will follow what seems to be the best path to the goal given the current cost estimates for its neighbors. The estimate cost to reach the goal through a neighbor s' is the cost to get to s' plus the estimated cost to get to a goal from there, that is, c(s, a, s') + H(s').

LRTA*: learning real-time A*. It builds a map of the environment in the result table, update the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates.

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
              H, a table of cost estimates indexed by state, initially empty
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null
      result[s, a] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

Actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost (a.k.a.h(s)), this optimism under uncertainty encourages the agent to explore new, possible promising paths.