MODULE 2

PROBLEM SOLVING AGENTS

2.1 Problem Solving Agents
2.2 Illustration of the problem solving process by agents
2.3 Searching for solutions
2.4 Uninformed search strategies
- BFS
- Uniform-cost search
- DFS
- Depth limited search
- Iterative deepening depth-first search.

2.5 Informed search strategies
- Best First search
- A* Search

2.6 Heuristics

2.1 PROBLEM SOLVING AGENTS

Problem-solving agents is one kind of goal-based agent it uses **atomic** representations that is, states of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms.

Goal-based agents that use more advanced **factored** or **structured** representations are usually called **planning agents.** Goals help organize behavior by limiting the objectives that the agent is trying to achieve and hence the actions it needs to consider.

Steps Require to Solve a Problem

Goal Formulation:

- It organizes finite steps to formulate a target/goals which require some actionto achieve the goal.
- based on AI agents.

Problem formulation:

- decides what action should be taken to achieve the formulated goal.

Goal formulation

- It is based on the current situation and the agent's performance measure.
- The goal is formulatedas a set of world states, in which the goal is satisfied. Reaching from initial state to goal state some actions are required.
- *Actions* are the operators causing transitions between world states.
- **Actions** should be abstract enough at a certain degree, instead of very detailed.
- E.g., turn left VS turn left 30 degrees, etc. With such high level of detail there is too much uncertainty in the world and there would be too many steps in a solution for agent to find a solution

Problem formulation

- It is the process of deciding what actions and states to consider.
- E.g., driving Ernakulam to Chennai in-between states and actions defined.
  - States: Some places in Ernakulam and Chennai.

  Actions: Turn left, Turn right, go straight, accelerate & brake, etc.
- Agent will consider actions at the level of driving from one major town to another.
- Each state therefore corresponds to being in a particular town.

Properties of the Environment

The properties of the environment are

- **Observable:** agent always knows the current state
- **Discrete:** at any given state there are only finitely many actions to choose from
- **Known:** agent knows which states are reached by each action.
- **Deterministic:** each action has exactly one outcome

Under these assumptions, the solution to any problem is a fixed sequence of actions

Search

- The process of looking for a sequence of actions that reaches the goal is called **search**.
- A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence.
- Once a solution is found, the actions it recommends can be carried out. This is called the **execution** phase.

Agent has a "formulate, search, execute" design.

Open-loop system

- While the agent is executing the solution sequence it *ignores its percepts* when choosing an action because it knows in advance what they will be.
-
- An agent that carries out its plans with its eyes closed, so to speak, must be quite certain of what is going on is an open loop.
- Ignoring the percepts breaks the loop between agent and environment.

Well-defined problems and solutions

A **problem** can be defined formally by following components:

1. The **initial state** that the agent starts in
2. A description of the possible **actions** available to the agent.
   - Given a particular state s, ACTIONS(s) returns the set of actions that can beexecuted in s. We say that each of these actions is **applicable** in s.
   - For example, from the state In(Ernakulam), the applicable actions are

{Go(Thrissur), Go(Palakkad), Go(Kozhikod)}.

3. **Transition model:** description of what each action does, specified by a functionRESULT(s, a) that returns the state that results from doing action a in state s

   Successor: any state reachable from a given state by a single action

   - RESULT(In(Ernakulam),Go(Thrissur)) = In(Thrissur) .

   state space:

   the set of all states reachable from the initial state by any sequence of actions. forms a directed network or **graph** in which the nodes are states and the links between nodes are actions.

   A **path** in the state space is a sequence of states connected by a sequence of actions

4. The **goal test**, which determines whether a given state is a goal state {In(Chennai)}
5. A **path cost** function that assigns a numeric cost to each path cost of a path can be describedas the *sum* of the costs of the individual actions along the path.

The step cost of taking action a in state s to reach state s is denoted by c(s, a, s). A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution qualityis measured by the path cost function, and an optimal solution has the lowest path cost amongall solutions

Formulating problems

Anything else besides the four components for problem formulation

Abstraction

- the process to take out the irrelevant information
- leave the most essential parts to the description of the states (Remove detail fromrepresentation)
- **Conclusion**: Only the most important parts *that are contributing to searching* areused

The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out. Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped bythe real world.

Evaluation Criteria

- formulation of a problem as search task
- basic search strategies
- important properties of search strategies
- selection of search strategies for specific tasks (The ordering of the nodes in FRINGEdefines the search strategy)

2.2 Illustration of the problem solving process by agents

There are two well known distinguished type of problems.

1. Toy Problem
2. Real world Problem

Toy Problem is intended to illustrate or exercise various problem-solving methods. E.g., puzzle, chess, etc.

A real-world problem is one whose solutions people actually care about. E.g., Design,

planning, etc.

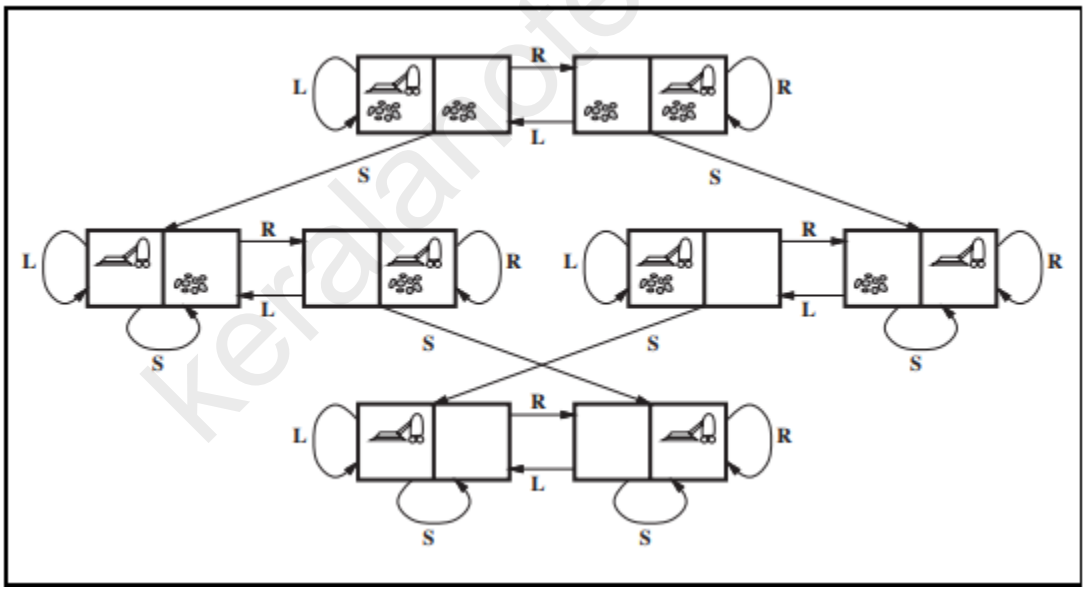
TOY PROBLEM

Vacuum World

8-puzzle
8-queens problem

Vacuum World

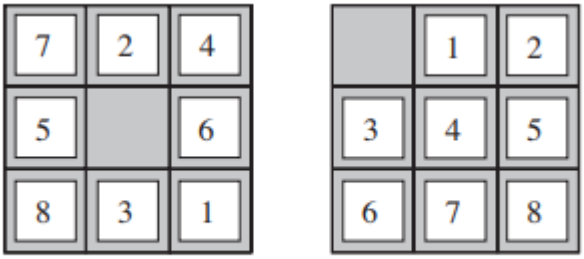
- **States**: The state is determined by both the agent location and the dirt locations. The agent is in one of two locations, each of which might or might not contain dirt. Thus, there are 2 × 22 = 8 possible world states. A larger environment with n locations has n · 2n states.
- **Initial state**: Any state can be designated as the initial state.
- **Actions**: In this simple environment, each state has just three actions: Left, Right, and Suck. Larger environments might also include Up and Down.
- **Transition model**: The actions have their expected effects, except that moving Left in the leftmost square, moving Right in the rightmost square, and Sucking in a clean square have no effect. The complete state space is shown in Figure.
- **Goal test**: This checks whether all the squares are clean.
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path.



Compared with the real world, this toy problem has discrete locations, discrete dirt, reliable cleaning, and it never gets any dirtier

8-puzzle

The 8-puzzle, an instance of which is shown in Figure, consists of a 3×3 board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state, such as the one shown on the right of the figure.



Start State Goal State

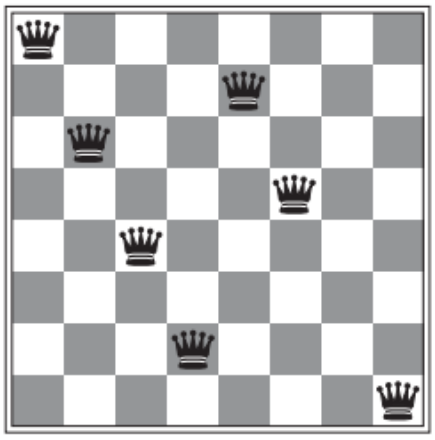
The standard formulation is as follows:

- States: A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- Initial state: Any state can be designated as the initial state. Note that any given goal can be reached from exactly half of the possible initial states.
- Actions: The simplest formulation defines the actions as movements of the blank space Left, Right, Up, or Down. Different subsets of these are possible depending on where the blank is.
- Transition model: Given a state and action, this returns the resulting state; for example, if we apply Left to the start state in Figure, the resulting state has the 5 and the blank switched.
- Goal test: This checks whether the state matches the goal configuration shown in Figure. (Other goal configurations are possible.)
- Path cost: Each step costs 1, so the path cost is the number of steps in the path.

The 8-puzzle has 9!/2 = 181, 440 reachable states and is easily solved. The 15-puzzle (on a 4×4 board) has around 1.3 trillion states, and random instances can be solved optimally in a few milliseconds by the best search algorithms. The 24-puzzle (on a 5 × 5 board) has around 10^{25} states, and random instances take several hours to solve optimally

8-queens problem

The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.) Figure 3.5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left



There are two main kinds of formulation.
An **incremental formulation** involves operators that augment the state description, starting with an empty state; for the 8-queens problem, this means that each action adds a queen to the state.

A **complete-state formulation** starts with all 8 queens on the board and moves them around.
In either case, the path cost is of no interest because only the final state counts.

The first incremental formulation one might try is the following:

- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

In this formulation, we have $64 \cdot 63 \cdots 57 \approx 1.8 \times 1014$ possible sequences to investigate.

A better formulation would prohibit placing a queen in any square that is already attacked:

- **States**: All possible arrangements of n queens ($0 \leq n \leq 8$), one per column in the leftmost n columns, with no queen attacking another.
- **Actions**: Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.

This formulation reduces the 8-queens state space from 1.8×1014 to just 2,057, and solutions are easy to find.

REAL WORLD PROBLEMS

Route-finding problem is defined in terms of specified locations and transitions along links between them. Route-finding algorithms are used in a variety of applications.

Consider the airline travel problems that must be solved by a travel-planning Web site:

- **States**: Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.
- **Initial state**: This is specified by the user's query.
- **Actions**: Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **Transition model**: The state resulting from taking a flight will have the flight's destination as the current location and the flight's arrival time as the current time.
- **Goal test**: Are we at the final destination specified by the user?
- **Path cost**: This depends on monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on

Touring problems are closely related to route-finding problems, but with an important

difference. Consider, for example, the problem "Visit every city at least once, starting and ending in Bucharest." As with route finding, the actions correspond to trips between adjacent cities. The state space, however, is quite different. Each state must include not just the current location but also the set of cities the agent has visited. So the initial state would be In(Bucharest), Visited({Bucharest}), a typical intermediate state would be In(Vaslui), Visited({Bucharest, Urziceni, Vaslui}), and the goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**. In cell layout, the aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells.

Robot navigation is a generalization of the route-finding problem. Rather than following a discrete set of routes, a robot can move in a continuous space with (in principle) an infinite set of possible actions and states. When the robot has arms and legs or wheels that must also be controlled, the search space becomes many-dimensional. Advanced techniques are required just to make the search space finite

2.3 SEARCHING FOR SOLUTIONS

A **solution** is an action sequence, and **search** algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem

Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
Search Space: Search space represents a set of possible solutions, which a system may have.
Start State: It is a state from where agent begins the search.
Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.

Search tree
A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.

Actions
It gives the description of all the available actions to the agent.

Transition model
A description of what each action do, can be represented as a transition model.

Path Cost
It is a function which assigns a numeric cost to each path.

Solution
It is an action sequence which leads from the start node to the goal node.

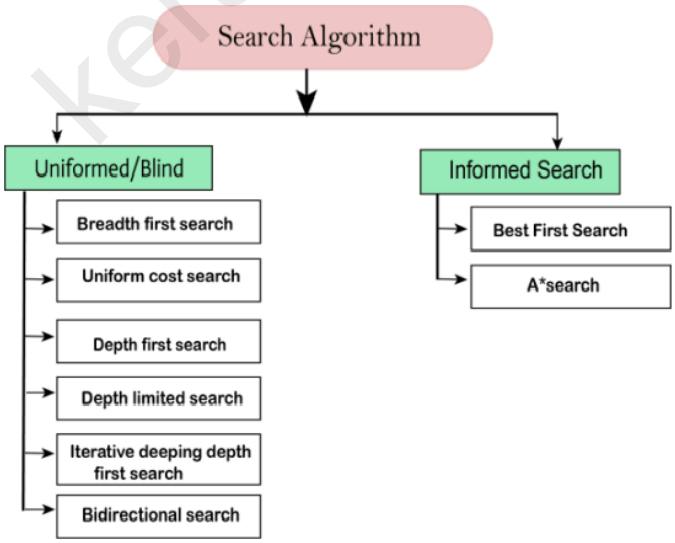
Optimal Solution
If a solution has the lowest cost among all solutions.

Measuring problem-solving performance
We can evaluate an algorithm's performance in four ways:

- **Completeness**: A search algorithm is said to be complete if it guarantees to return a solution if at least any solution exists for any random input.
- **Optimality**: If a solution found for an algorithm is guaranteed to be the best solution (lowest path cost) among all other solutions, then such a solution for is said to be an optimal solution.
- **Time Complexity**: Time complexity is a measure of time for an algorithm to complete its task.
- **Space Complexity**: It is the maximum storage space required at any point during the search, as the complexity of the problem.

TYPES OF SEARCH ALGORITHMS

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.



2.4 UNINFORMED/BLIND SEARCH

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes.

Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is also called blind search. It examines each node of the tree until it achieves the goal node.

It can be divided into five main types:

- Breadth-first search
- Uniform cost search
- Depth-first search
- Iterative deepening depth-first search
- Bidirectional Search

Breadth-first Search

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadth wise 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 solution 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. o BFS needs lots of time if the solution is far away from the root node.

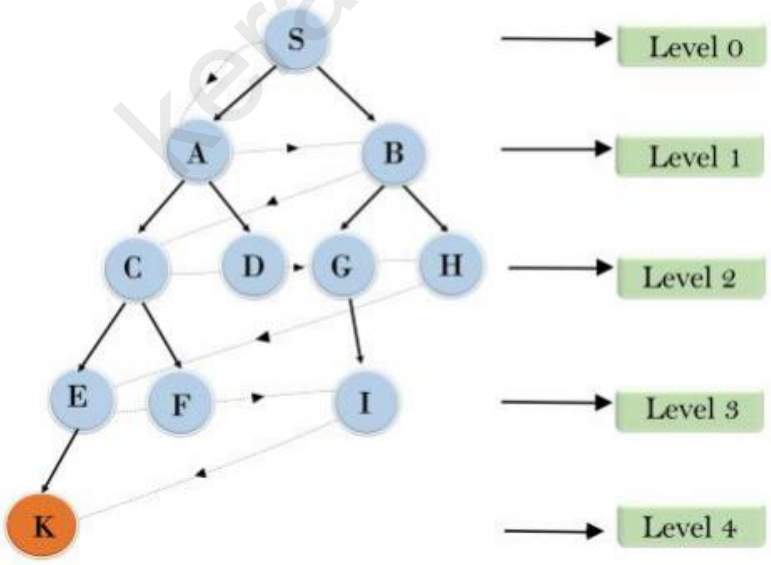
Algorithm

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

The root node is expanded first, then all the successors of the root node are expanded next, then*their* successors, and so on. All the nodes are expanded at a given depth in the search tree beforeany nodes at the next level are expanded.

Breadth-first search is an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion. Thisis achieved very simply by using a FIFO queue for the frontier new nodes go to the back of thequeue, and old nodes, which are shallower than the new nodes, get expanded first the goal testis applied to each node when it is *generated* rather than when it is selected for expansion breadth-first search always has the shallowest path to every node on the frontier.

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:

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

complete—if the shallowest goal node is at some finite depth d, breadth-first search willeventually find it after generating all shallower nodes

not optimal one: breadth-first search is optimal if the path cost is a non decreasing function ofthe depth of the node. The most common such scenario is that all actions have the same cost.

Time Complexity: The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b2 at the second level. Each of *these* generates b morenodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d. In the worst case, it is the last node generated at that level. Then the total number of nodesgenerated is
$b + b^2 + b^3 + \cdots + b^d = O(b^d)$

Space Complexity: every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier so the space complexity is $O(b^d)$, exponential complexity

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.

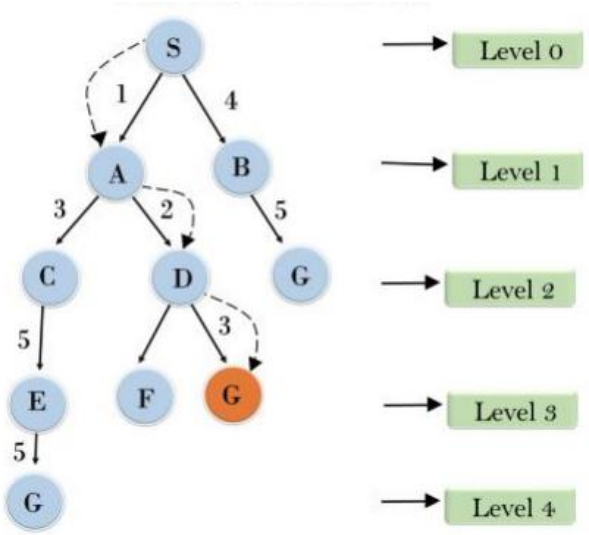
- Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

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



Complete: guaranteed provided the cost of every step exceeds some small positive constant ε

Optimal: optimal path to that node has been found. because step costs are nonnegative, paths never get shorter as nodes are added. uniform-cost search expands nodes in orderof their optimal path cost.

Time Complexity: # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution

Space Complexity: # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$

Uniform-cost search is guided by path costs rather than depths, so its complexity is not easily characterized in terms of b and d. Let C* be the cost of the optimal solution and that every action costs at least ε. Then the algorithm's worst-case time and space complexity iswhich can be much greater than b^d. This is because uniform cost search can explore large treesof small steps before exploring paths involving large and perhaps useful steps. When all step costs are equal $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^{d+1}. When all step costs are the same,uniform-cost search is similar to breadth-first search, except that bfs stops as soon as it generates a goal, whereas **uniform-cost search examines all the nodes at the goal's depth tosee if one has a lower cost** thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

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:

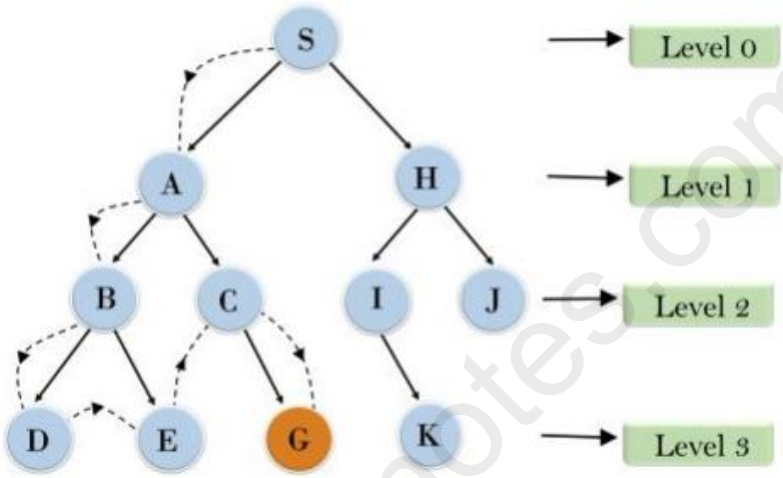
- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

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. 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.



it will follow the order as:

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

1. **Completeness:**
   - depth-first search is implemented with a recursive function that calls itself on eachof its children in turn.
   - The properties of depth-first search depend strongly on whether the graph-search or
   - tree-search version is used.
   - The graph-search version, which avoids repeated states and redundant paths, iscomplete in finite state spaces because it will eventually expand every node.
   - The tree-search version, on the other hand, is *not* complete
   - Depth-first tree search can be modified at no extra memory cost so that it checksnew states against those on the path from the root to the current node;
   - this avoids infinite loops in finite state spaces but does not avoid the proliferationof redundant paths.

- In infinite state spaces, both versions fail if an infinite non-goal path is encountered.

2. **Not optimal**
   - depth- first search will explore the entire left subtree even if node C is a goal node.
   - If node J were also a goal node, then depth-first search would return it as a solution instead of C, which would be a better solution; hence, depth-first search is not optimal.
3. **Time complexity**
   - depth-first graph search is bounded by the size of the state space
   - A depth-first tree search, on the other hand, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node; this can be much greater than the size of the state space.
   - m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded
4. **Space complexity**
   - a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
   - Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.
   - For a state space with branching factor b and maximum depth m, depth-first search requires storage of only O(bm) nodes.
   - assuming that nodes at the same depth as the goal node have no successors, we find that depth-first search would require 156 kilobytes instead of 10 exabytes at depth d = 16, a factor of **7 trillion times less space**.

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

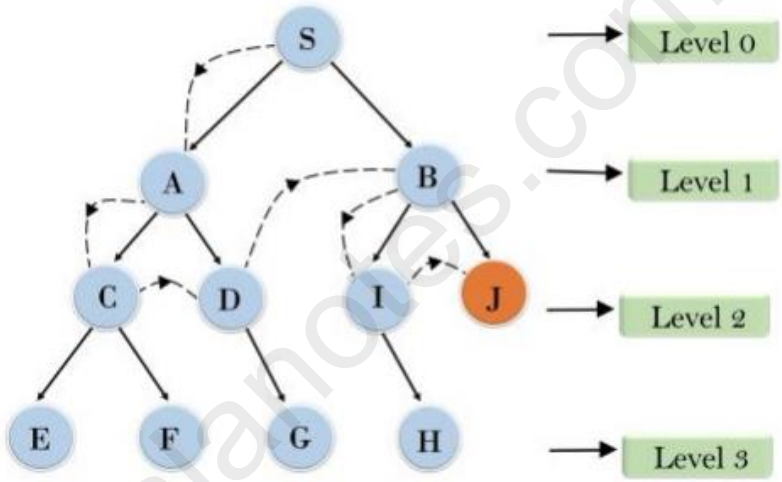
Algorithm

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

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).
- **Space Complexity**: Space complexity of DLS algorithm is O(b×ℓ).
- **Optimal**: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

Iterative deepening depth-first Search

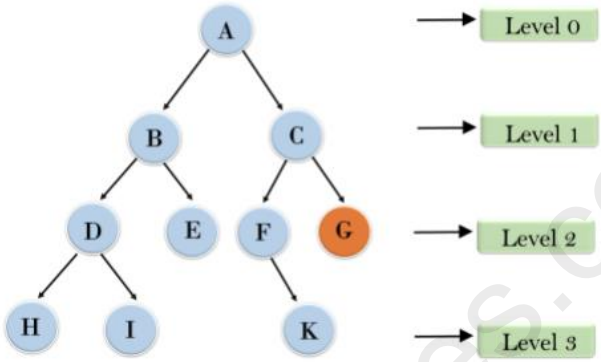
The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found. This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the 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:
It combines 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.



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(bd).
- **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.

2.5 Informed Search Algorithms

- So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space.
- But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc.
- This knowledge help agents to explore less to the search space and find more efficiently the goal node. The informed search algorithm is more useful for large search space.
- Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function

- Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states.
- The value of the heuristic function is always positive.
- **Admissibility** of the heuristic function is given as: **h(n) <= h*(n)** Here h(n) is heuristic cost, and h*(n) is the estimated cost.
- Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search

- Pure heuristic search is the simplest form of heuristic search algorithms.
- It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list.
- In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list.
- The algorithm continues unit a goal state is found.
- In the informed search we will discuss two main algorithms which are given below:
  - Best First Search Algorithm(Greedy search)
  - A* Search Algorithm

Best-first Search Algorithm (Greedy Search)

- Greedy best-first search algorithm always selects the path which appears best at that moment.
- It is the combination of depth-first search and breadth-first search algorithms.
- It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms.
- With the help of best-first search, at each step, we can choose the most promising node.
- In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e. **f(n)= g(n).**
- Were, h(n)= estimated cost from node n to the goal.
- The greedy best first algorithm is implemented by the priority queue.

Best first search algorithm:

- Step 1: Place the starting node into the OPEN list.
- Step 2: If the OPEN list is empty, Stop and return failure.
- Step 3: Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
- Step 4: Expand the node n, and generate the successors of node n.
- Step 5: Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else

proceed to Step 6.

- Step 6: For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
- Step 7: Return to Step 2.

Advantages:

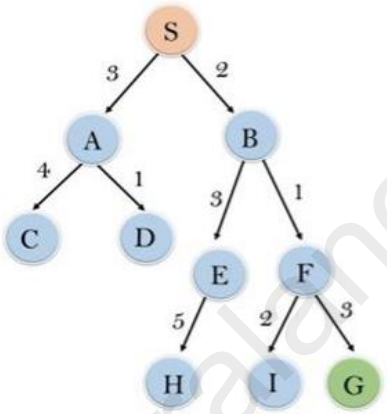
- Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
- This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

- It can behave as an unguided depth-first search in the worst case scenario.
- It can get stuck in a loop as DFS.
- This algorithm is not optimal.

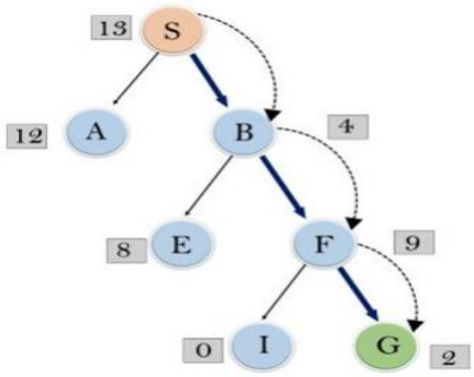
Example

- Consider the below search problem, and we will traverse it using greedy best-first search.
- At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

- In this search example, we are using two lists which are OPEN and CLOSED Lists.
- Following are the iteration for traversing the above example.



- Expand the nodes of S and put in the CLOSED list
- Initialization: Open [A, B], Closed [S]
- Iteration 1: Open [A], Closed [S, B]
- Iteration 2: Open [E, F, A], Closed [S, B] : Open [E, A], Closed [S, B, F]
- Iteration 3: Open [I, G, E, A], Closed [S, B, F] : Open [I, E, A], Closed [S, B, F, G]
- Hence the final solution path will be: S----> B----->F----> G

Time Complexity

- The worst case time complexity of Greedy best first search is O(bm).

Space Complexity

- The worst case space complexity of Greedy best first search is O(bm). Where, m is the maximum depth of the search space.

Complete

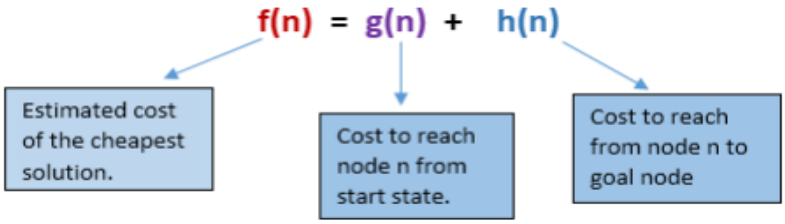
- Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal

- Greedy best first search algorithm is not optimal.

A* Search Algorithm

- A* search is the most commonly known form of best-first search.
- It uses heuristic function h(n), and cost to reach the node n from the start state g(n).
- It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently.
- A* search algorithm finds the shortest path through the search space using the heuristic function.
- This search algorithm expands less search tree and provides optimal result faster.
- A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).
- In A* search algorithm, we use search heuristic as well as the cost to reach the node.
- Hence we can combine both costs as following, and this sum is called as a fitness number.



At each point in the search space, only those node is expanded which have the lowest value of f(n), and the algorithm terminates when the goal node is found.

Algorithm of A* search

- Step1: Place the starting node in the OPEN list.
- Step 2: Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
- Step 3: Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
- Step 4: Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
- Step 5: Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
- Step 6: Return to Step 2.

Advantages

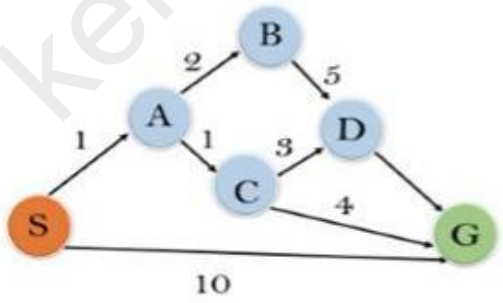
- A* search algorithm is the best algorithm than other search algorithms.
- A* search algorithm is optimal and complete. o This algorithm can solve very complex problems

Disadvantages:

- It does not always produce the shortest path as it mostly based on heuristics and approximation.
- A* search algorithm has some complexity issues.
- The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

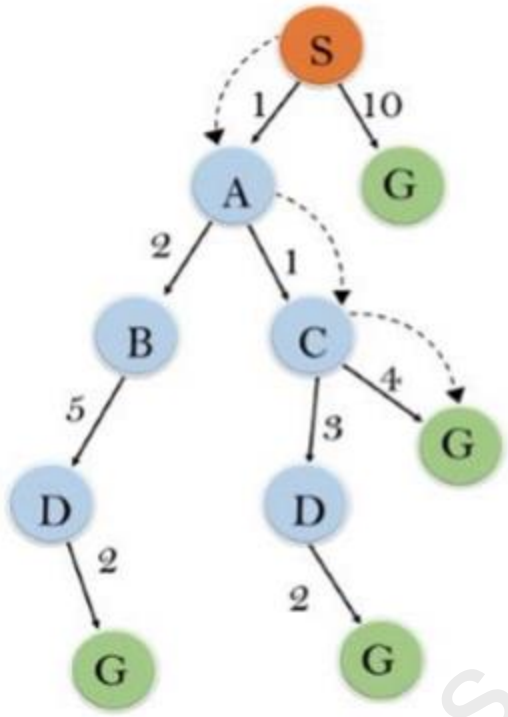
Example

- In this example, we will traverse the given graph using the A* algorithm.
- The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula **f(n)= g(n) + h(n),** where g(n) is the cost to reach any node from start state.
- Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Solution



- Initialization: {(S, 5)}
- Iteration1: {(S--> A, 4), (S-->G, 10)}
- Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}
- Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}
- Iteration 4 will give the final result, as S--->A--->C--->G it provides the optimal path with cost 6.

Points to remember

- A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
- The efficiency of A* algorithm depends on the quality of heuristic.
- A* algorithm expands all nodes which satisfy the condition f(n)<="" li="">

Complete

- A* algorithm is complete as long as:
  - Branching factor is finite. o Cost at every action is fixed.

Optimal

- A* search algorithm is optimal if it follows below two conditions:
- Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.
- Consistency: Second required condition is consistency for only A* graph-search.
- If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity

- The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So the time

complexity is **O(b^d),** where b is the branching factor.

Space Complexity

- The space complexity of A* search algorithm is **O(b^d).**

2.6 Heuristics

- Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states.
- The value of the heuristic function is always positive.
- **Admissibility** of the heuristic function is given as: **h(n) <= h*(n)** Here h(n) is heuristic cost, and h*(n) is the estimated cost.
- Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search

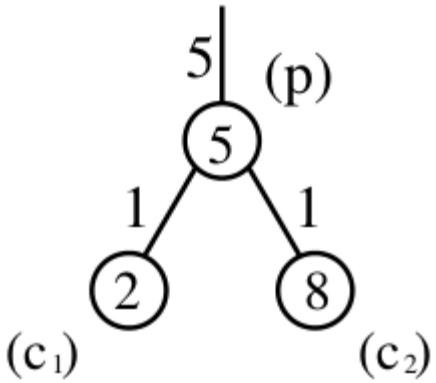
- Pure heuristic search is the simplest form of heuristic search algorithms.
- It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list.
- In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list.
- The algorithm continues unit a goal state is found.

Consistent heuristic

- For every node *N* and each successor *P* of *N*, the estimated cost of reaching the goal from *N* is no greater than the step cost of getting to *P* plus the estimated cost of reaching the goal from *P*.
- That is:

$$h(N) \leq c(N, P) + h(P) \text{ and}$$
$$h(G) = 0.$$

- Where
  - *h* is the consistent heuristic function
  - *N* is any node in the graph
  - *P* is any descendant of *N*
  - *G* is any goal node
  - c(N,P) is the cost of reaching node P from N



- This heuristic is **inconsistent at c1** because it is giving a lower (i.e. less informative) lower bound on the cost to get to the goal than its parent node is.
- The cost estimate of getting to the goal through the parent node is at least 10 (because the cost of the path to p is 5 and the heuristic estimate at p is also 5).
- The cost estimate for getting to the goal through c1, however, is just 8 (cost of parent (5), plus cost of path from parent (1), plus heuristic estimate at c1 (2)).
- Since this graph is undirected, this heuristic is also inconsistent at c2, because going from c2 to p has the same problem as above.

HEURISTIC FUNCTIONS

An 8-puzzle search space has

- typical solution length: 20 steps,
- Average branching factor: 3
- Exhaustive search: 3^{20}=3.5 x 10^9
- Bound on unique states: 9! = 362,880

Admissible Heuristics

- h1 = the number of misplaced tiles. For Figure 3.28, all of the eight tiles are out of position, so the start state would have h1 = 8. h1 is an admissible heuristic because it isclear that any tile that is out of place must be moved at least once.
- h2 = the sum of the distances of the tiles from their goal positions. Because tiles cannotmove along diagonals, the distance we will count is the sum of the horizontal and vertical distances. This is sometimes called the **city block distance** or **Manhattan distance**. h2 is also admissible because all any move can do is move one tile one step closer to the goal. Tiles 1 to 8 in the start state give a Manhattan distance of

  h2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18

Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

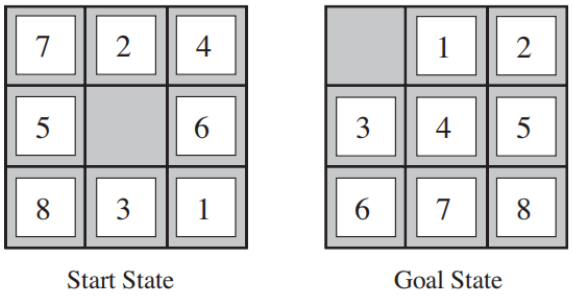


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

Heuristic Performance

Experiments on sample problems can determine the number of nodes searched and CPUtime for different strategies.

One other useful measure is effective branching factor: If a method expands N nodes to find solution of depth d, and a uniform tree of depth d would require a branching factor of b* to contain N nodes, the effective branching factor is b*

- $N = 1 + b^* + (b^*)2 + ...+ (b^*)d$

Quality of Heuristics

Since A* expands all nodes whose f value is less than that of an optimal solution, it is always better to use a heuristic with a higher value as long as it does not over-estimate.

Therefore h2 is uniformly better than h1, or h2 dominates h1 .

A heuristic should also be easy to compute, otherwise the overhead of computing the heuristic could outweigh the time saved by reducing search (e.g. using full breadth-first search to estimate distance wouldn't help)

Inventing Heuristics

Many good heuristics can be invented by considering relaxed versions of the problem(abstractions).

For 8-puzzle: A tile can move from square A to B if A is adjacent to B and B is blank

- (a) A tile can move from square A to B if A is adjacent to B.
- (b) A tile can move from square A to B if B is blank. (c) A tile can move fromsquare A to B.

If there are a number of features that indicate a promising or unpromising state, a weightedsum of these features can be useful.

Generating admissible heuristics from relaxed problems

- We have seen that both h1 (misplaced tiles) and h2 (Manhattan distance) are fairly good heuristics for the 8-puzzle and that h2 is better
- h1 and h2 are estimates of the remaining path length for the 8-puzzle, but they are also perfectly accurate path lengths for simplified versions of the puzzle.
- If the rules of the puzzle were changed so that a tile could move anywhere instead of just to the adjacent empty square, then h1 would give the exact number of steps in the shortest solution.

- Similarly, if a tile could move one square in any direction, even onto an occupied square, then h2 would give the exact number of steps in the shortest solution.
- A problem with fewer restrictions on the actions is called a relaxed problem.
- The state-space graph of the relaxed problem is a super graph of the original state space because the removal of restrictions creates added edges in the graph.
- Because the relaxed problem adds edges to the state space, any optimal solution in the original problem is, by definition, also a solution in the relaxed problem; but the relaxed problem may have better solutions if the added edges provide short cuts.
- Hence, the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.
- Furthermore, because the derived heuristic is an exact cost for the relaxed problem, it must obey the triangle inequality and is therefore consistent.
- If a problem definition is written down in a formal language, it is possible to construct relaxed problems automatically.
- For example, if the 8-puzzle actions are described as
- A tile can move from square A to square B
  - if A is horizontally or vertically adjacent to B and
  - B is blank,
- we can generate three relaxed problems by removing one or both of the conditions:
  - (a) A tile can move from square A to square B if A is adjacent to B.
  - (b) A tile can move from square A to square B if B is blank.
  - (c) A tile can move from square A to square B.

Generating admissible heuristics from sub problems: Pattern databases

- Admissible heuristics can also be derived from the solution cost of a subproblem of a given problem.
- For example, Figure below shows a sub problem of the 8-puzzle instance.

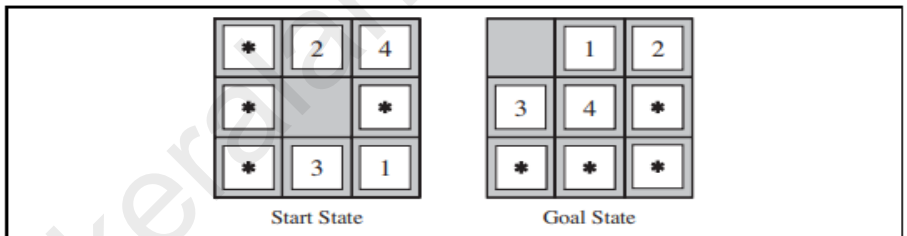


Figure 3.30 A subproblem of the 8-puzzle instance given in Figure 3.28. The task is to get tiles 1, 2, 3, and 4 into their correct positions, without worrying about what happens to the other tiles.

- The sub problem involves getting tiles 1, 2, 3, 4 into their correct positions. Clearly, the cost of the optimal solution of this sub problem is a lower bound on the cost of the complete problem.
- It turns out to be more accurate than Manhattan distance in some cases.
- The idea behind pattern databases is to store these exact solution costs for every possible sub problem instance—in our example, every possible configuration of the four tiles and the blank. (The locations of the other four tiles are irrelevant for the purposes of solving the sub problem, but moves of those tiles do count toward the cost.)
- Then we compute an admissible heuristic hDB for each complete state encountered during a search simply by looking up the corresponding sub problem configuration in the database.
- The database itself is constructed by searching back13 from the goal and recording the cost of each new pattern encountered
- The expense of this search is amortized over many subsequent problem instances.

- The choice of 1-2-3-4 is fairly arbitrary; we could also construct databases for 5-6-7-8, for 2-4-6-8, and so on.

Learning heuristics from experience

- A heuristic function h(n) is supposed to estimate the cost of a solution beginning from the state at node n. How could an agent construct such a function?
- One solution was given in the preceding sections—namely, to devise relaxed problems for which an optimal solution can be found easily.
- Another solution is to learn from experience. "Experience" here means solving lots of 8-puzzles, for instance.
- Each optimal solution to an 8-puzzle problem provides examples from which h(n) can be learned.
- Each example consists of a state from the solution path and the actual cost of the solution from that point.
- From these examples, a learning algorithm can be used to construct a function h(n) that can (with luck) predict solution costs for other states that arise during search.
- Techniques for doing just this using neural nets, decision trees, and other methods