CST401 ARTIFICIAL INTELLIGENCE

MODULE 2

Course Outcomes: After the completion of the course the student will be able to

CO#	CO
CO1	Explain the fundamental concepts of intelligent systems and their architecture. **(Cognitive Knowledge Level: Understanding)**
CO2	Illustrate uninformed and informed search techniques for problem solving in intelligent systems. **(Cognitive Knowledge Level: Understanding)**
CO3	Solve Constraint Satisfaction Problems using search techniques. **(Cognitive Knowledge Level: Apply**)
CO4	Represent AI domain knowledge using logic systems and use inference techniques for reasoning in intelligent systems. (**Cognitive Knowledge Level: Apply**)
CO5	Illustrate different types of learning techniques used in intelligent systems **(Cognitive Knowledge Level: Understand)**

Mapping of course outcomes with program outcomes

	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	✓											
CO2	✓	✓										✓
CO3	✓	✓	✓	✓								✓
CO4	✓	✓	✓	✓								✓
CO5	✓	✓			✓							✓

Abstract POs defined by National Board of Accreditation			
PO#	**Broad PO**	**PO#**	**Broad PO**
PO1	Engineering Knowledge	**PO7**	Environment and Sustainability
PO2	Problem Analysis	**PO8**	Ethics
PO3	Design/Development of solutions	**PO9**	Individual and team work
PO4	Conduct investigations of complex problems	**PO10**	Communication
PO5	Modern tool usage	**PO11**	Project Management and Finance
PO6	The Engineer and Society	**PO12**	Life long learning

****

SYLLABUS- MODULE 2

Solving Problems by searching-Problem solving Agents, Example problems, Searching for solutions, Uninformed search strategies, Informed search strategies, Heuristic functions.
****

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 action to 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 formulated as 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.

Components to formulate the associated problem

- **Initial State:** This state requires an initial state for the problem which starts the AI agent towards a specified goal. In this state new methods also initialize problem domain solving by a specific class.
- **Action:** This stage of problem formulation works with function with a specific class taken from the initial state and all possible actions done in this stage.
- **Transition:** This stage of problem formulation integrates the actual action done by the previous action stage and collects the final stage to forward it to their next stage.
- **Goal test:** This stage determines that the specified goal achieved by the integrated transition model or not, whenever the goal achieves stop the action and forward into the next stage to determines the cost to achieve the goal.

- **Path costing:** This component of problem-solving numerical assigned what will be the cost to achieve the goal. It requires all hardware software and human working cost.

An agent with several immediate options of unknown value can decide what to do by first examining future actions that eventually lead to states of known value.

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**

Searching Process

1. Formulate a goal and a problem to solve,
2. the agent calls a search procedure to solve it
3. Agent uses the solution to guide its actions,
4. do whatever the solution recommends
5. remove that step from the sequence.
6. Once the solution has been executed, the agent will formulate a new goal.

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 be executed 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 function RESULT(s, a) that returns the state that results from doing action a in state s
4. **Successor:** any state reachable from a given state by a single action
    - RESULT(In(Ernakulam),Go(Thrissur)) = In(Thrissur) .
5. **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.
6. A **path** in the state space is a sequence of states connected by a sequence of actions

7. The **goal test**, which determines whether a given state is a goal state {In(Chennai)}
8. A **path cost** function that assigns a numeric cost to each path cost of a path can be described as 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 quality is measured by the path cost function, and an optimal solution has the lowest path cost among all 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 from representation)
- ◦ **Conclusion**: Only the most important parts *that are contributing to searching* are used

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 by the 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 FRINGE defines the search strategy)

Problem-Solving Agents

agents whose task is to solve a particular problem (steps)

- goal formulation
  - ◦ what is the goal state
  - ◦ what are important characteristics of the goal state
  - ◦ how does the agent know that it has reached the goal
  - ◦ are there several possible goal states
    - ◦ are they equal or are some more preferable
- problem formulation
  - ◦ what are the possible states of the world relevant for solving the problem
  - ◦ what information is accessible to the agent
  - ◦ how can the agent progress from state to state

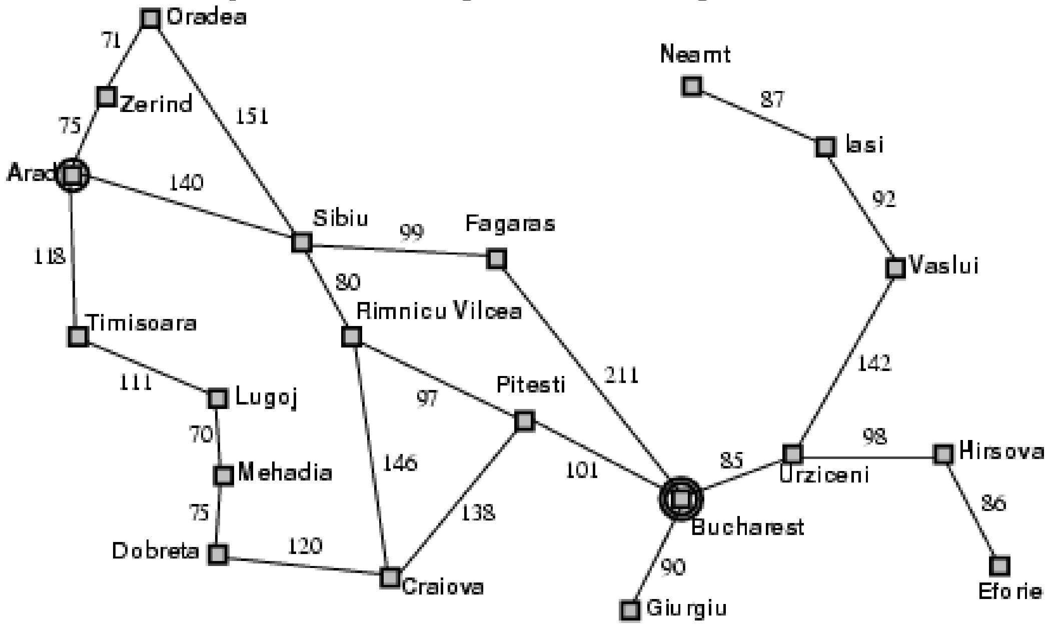
Example: Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest

Formulate goal:
- ◦ be in Bucharest

Formulate problem:
- ◦ states: various cities
- ◦ actions: drive between cities
- ◦ Find solution:

- sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest



1. **Problem : To Go from** Arad **to** Bucharest
2. **Initial State :** Arad
3. **Operator : Go from One City To another .**
4. **State Space : {**Sibiu, Fagaras, Timisora**,….}**
5. **Goal Test : are the agent in** Bucharest**.**
6. **Path Cost Function : Get The Cost From The Map.**
7. **Solution :{** {Ar → sib → Fr→Bu} , {Ar →Ti → Lu → Me → Cr,→Pi→Bu} …..**}**
8. **State Set Space :** {Arad → Sibiu → Fagaras → Bucharest}

Single-state problem formulation
A problem is defined by four items:
1. . initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad)* = {*<Arad → Zerind, Zerind>, …* }
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

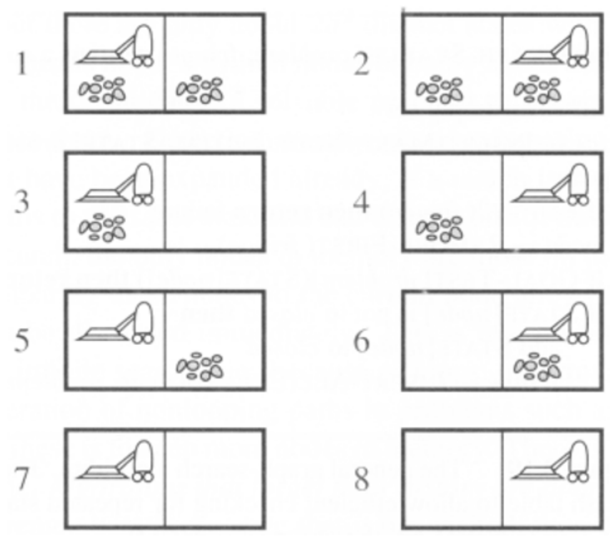
   A solution is a sequence of actions leading from the initial state to a goal state

Example Problems
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.

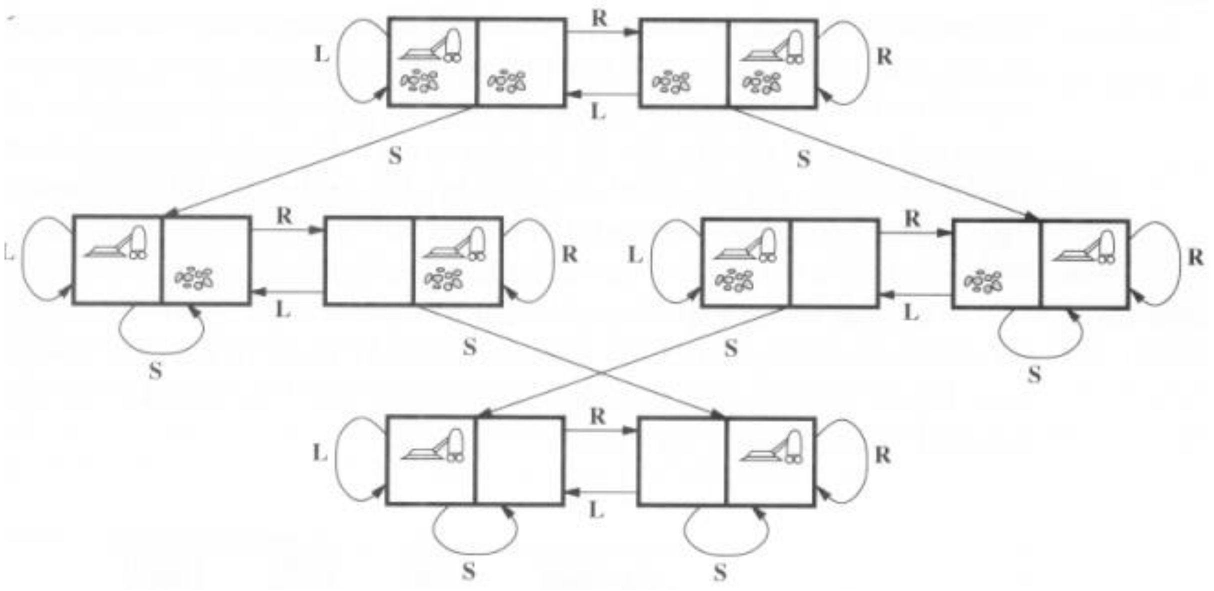
Toy problems
1. Vacuum World

- **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 $2 \times 2^2 = 8$ possible world states. A larger environment with n locations has $n \cdot 2^n$ 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 *Suck*ing in a clean square have no effect. The complete state space is shown in Figure 3.3.
- **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.

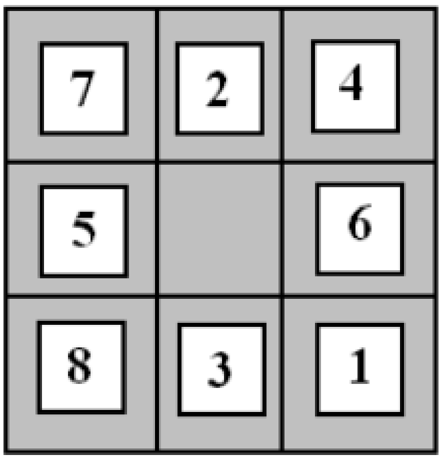


- Number of states: 8
- Initial state: Any
- Number of actions: 4
  - *left, right, suck, noOp*
- Goal: clean up all dirt
  - Goal states: {7, 8}
- Path Cost:
  - Each step costs 1

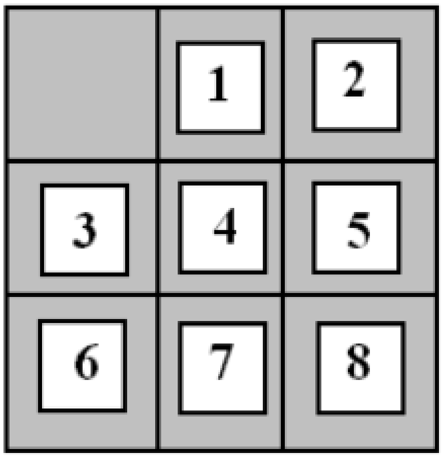


2. 8-puzzle

A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state



Start state



Goal state

- **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 (Exercise 3.4).
- **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 3.4, the resulting state has the 5 and the blank switched.
- **Goal test**: This checks whether the state matches the goal configuration shown in Figure 3.4. (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 1025 states, and random instances take several hours to solve optimally.

3. 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.

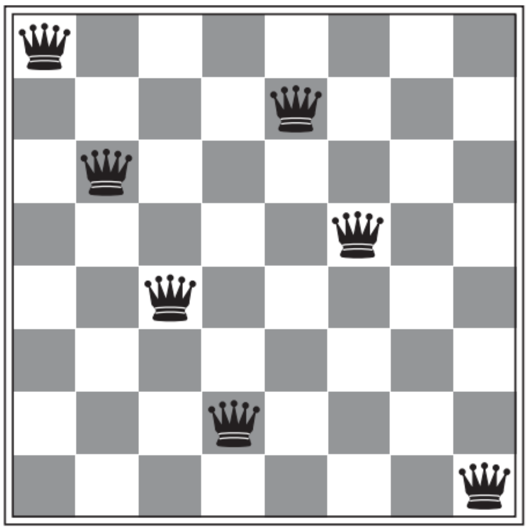


Figure 3.5 Almost a solution to the 8-queens problem. (Solution is left as an exercise.)

There are two main kinds of formulation

- An **incremental formulation**
  - ❖ involves operators that augment the state description starting from an empty state
  - ❖ Each action adds a queen to the state
  - ❖ States:
    - ✓ any arrangement of 0 to 8 queens on board
  - ❖ Successor function:
  - ❖ add a queen to any empty square
- A **complete-state formulation**
  - ❖ starts with all 8 queens on the board
  - ❖ move the queens individually around
  - ❖ States:
    - ✓ any arrangement of 8 queens, one per column in the leftmost columns
  - ❖ Operators: move an attacked queen to a row, not attacked by any other
- the right formulation makes a big difference to the size of the search space

Incremental formulation

- **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. On the other hand, for 100 queens the reduction is from roughly 10400 states to about 1052 states (Exercise 3.5)—a big improvement, but not enough to make the problem tractable

Real-world problems- Route-Finding problem

- ❑ Route-finding problems
- ❑ Touring problems
- ❑ Traveling Salesman problem
- ❑ VLSI layout problem
- ❑ Robot navigation
- ❑ Automatic assembly sequencing
- ❑ Internet searching

1. Airline travel problems
   - **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.

   A really good system should include contingency plans such as backup reservations on alternate flights to the extent that these are justified by the cost and likelihood of failure of the original plan.

2. Touring problems

"Visit every city in at least once, starting and ending in Bucharest.". Actions correspond to trips between adjacent cities. 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}), goal test would check whether the agent is in Bucharest and all 20 cities have been visited.

3. Traveling salesperson problem

The **traveling salesperson problem** (TSP) is a touring problem in which each city must be visited exactly once. The aim is to find the *shortest* tour. The problem is known to be NP-hard, but an enormous amount of effort has been expended to improve the capabilities of TSP algorithms. In addition to planning trips for traveling salespersons, these algorithms have been used for tasks such as planning movements of automatic circuit-board drills and of stocking machines on shop floors

4. **VLSI layout**

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**.

Cell layout

the primitive components of the circuit are grouped into cells, each of which performs some recognized function.
Each cell has a fixed footprint and requires a certain number of connections to each of the other cells.
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.

5. **Robot navigation**

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 an infinite set of possible actions and states. For a circular robot moving on a flat surface, the space is essentially two-dimensional. 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.

6. Automatic assembly sequencing

Aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking a step in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. **Protein design** is an automatic assembly problem in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

Searching for solution

A solution is an action sequence, so 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.

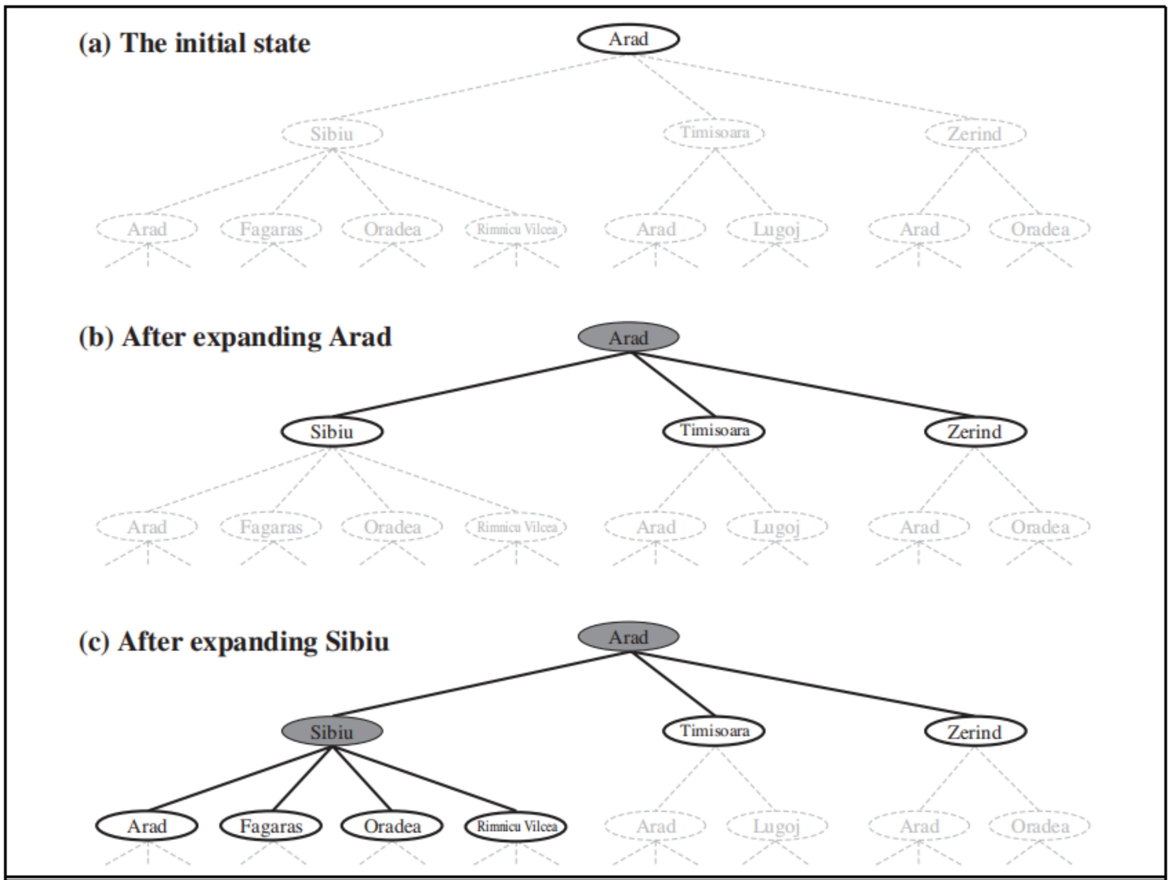


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

Frontier

We reach a state when we identify a path from the start state to it. But, we say that we expanded it if we had followed all its outward edges and reached all its children. So, we can also think of a search as a sequence of expansions, and we first have to reach a state before expanding it. Frontier is the reached but unexpanded states **because we can expand only them**

The root node of the tree corresponds to the initial state, *In(Arad)*. The first step is to test whether this is a goal state. Then we need to consider taking various actions. We do this by expanding the current state; applying each legal action to the current state, thereby generating a new set of states.

In this case, we add three branches from the parent node *In(Arad)* leading to three new child nodes: *In(Sibiu), In(Timisoara),* and *In(Zerind)*.

Now we must choose which of these three possibilities to consider further. The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand. Search strategy here is how they choose which state to expand next

loopy path: path from Arad to Sibiu and back to Arad again! We say that *In(Arad)* is a **repeated state** in the search tree, generated in this case by a **loopy path**

Considering such loopy paths means that the complete search tree for Romania is *infinite* because there is no limit to how often one can traverse a loop. Loops can cause certain algorithms to fail, making otherwise solvable problems unsolvable. There is no need to consider loopy paths. We can rely on more than intuition for this: because path costs are

additive and step costs are nonnegative, a loopy path to any given state is never better than the same path with the loop removed.
redundant paths: exist whenever there is more than one way to get from one state to another eg, the paths Arad–Sibiu (140 km long) and Arad–Zerind–Oradea–Sibiu (297 km long).

A search strategy has two components:

- rule(s) to decide whether or not to place the node in the frontier
- rule(s) to choose the next frontier node for expansion

TREE-SEARCH algorithm
With a data structure called the **explored set** (also known as the **closed list**), which **remembers every expanded node**. Newly generated nodes that match previously generated nodes ones in the explored set or the frontier can be discarded instead of being added to the frontier.

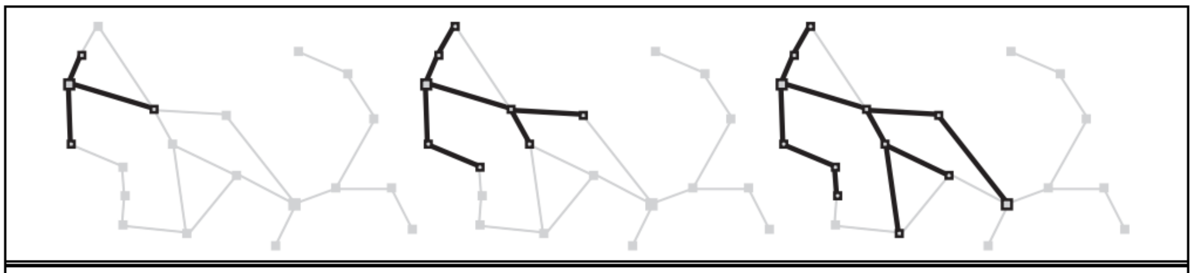


Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier
```

GRAPH-SEARCH algorithm

Each state appears in the graph only once. But, it may appear in the tree multiple times contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph,
Don't add a node if its state has already been expanded or a node pointing to the same state is already in the frontier.

so that every path from the initial state to an unexplored state has to pass through a state in the frontier.
As every step moves a state from the frontier into the explored region while moving some states from the unexplored region into the frontier,
we see that the algorithm is systematically examining the states in the state space, one by one, until it finds a solution.

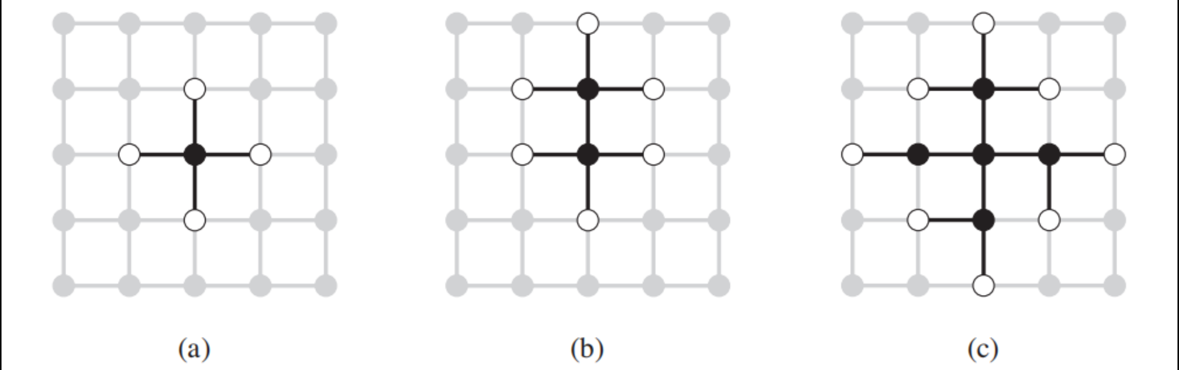


Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

```
**function** GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  ***initialize the explored set to be empty***
  **loop do**
     **if** the frontier is empty **then return** failure
     choose a leaf node and remove it from the frontier
     **if** the node contains a goal state **then return** the corresponding solution
     ***add the node to the explored set***
     expand the chosen node, adding the resulting nodes to the frontier
       ***only if not in the frontier or explored set***
```

Infrastructure for search algorithms
Search algorithms require a data structure to keep track of the search tree that is being constructed. For each node n of the tree, we have a structure that contains four components:

- n.STATE: the state in the state space to which the node corresponds;
- n.PARENT: the node in the search tree that generated this node;
- n.ACTION: the action that was applied to the parent to generate the node;
- n.PATH-COST: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

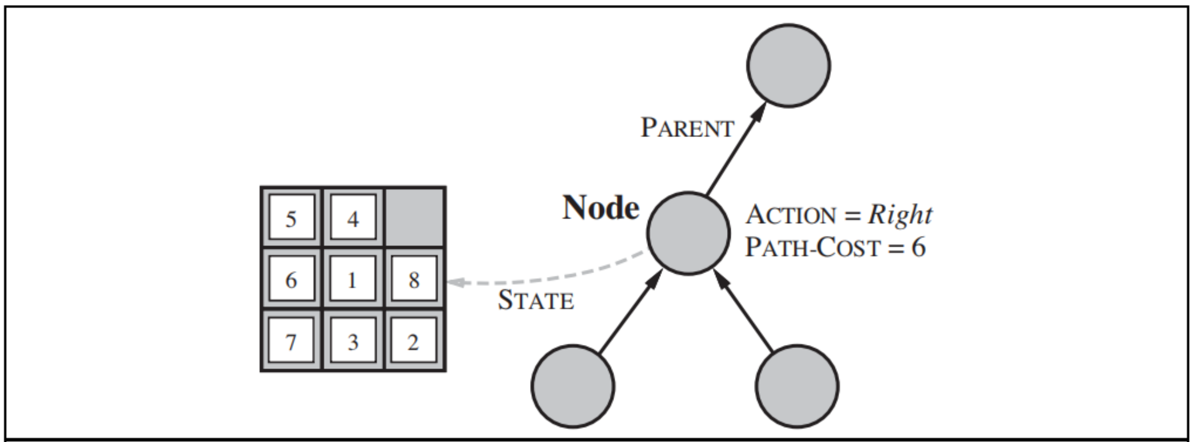


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

PARENT pointers string the nodes together into a tree structure. These pointers also allow the solution path to be extracted when a goal node is found; we use the SOLUTION function to return the sequence of actions obtained by following parent pointers back to the root.
A node is a bookkeeping data structure used to represent the search tree. A state corresponds to a configuration of the world nodes are on particular paths, as defined by PARENT pointers, whereas states are not. Two different nodes can contain the same world state if that state is generated via two different search paths.

Given the components for a parent node, it is easy to see how to compute the necessary components for a child node. The function CHILD-NODE takes a parent node and an action and returns the resulting child node:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
     STATE = problem.RESULT(parent.STATE, action),
     PARENT = parent, ACTION = action,
     PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Search Algorithm using Queue Data Structure
The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy
The operations on a queue are as follows:
• EMPTY?(queue) returns true only if there are no more elements in the queue.
• POP(queue) removes the first element of the queue and returns it.
• INSERT(element, queue) inserts an element and returns the resulting queue
Three common variants of queue are

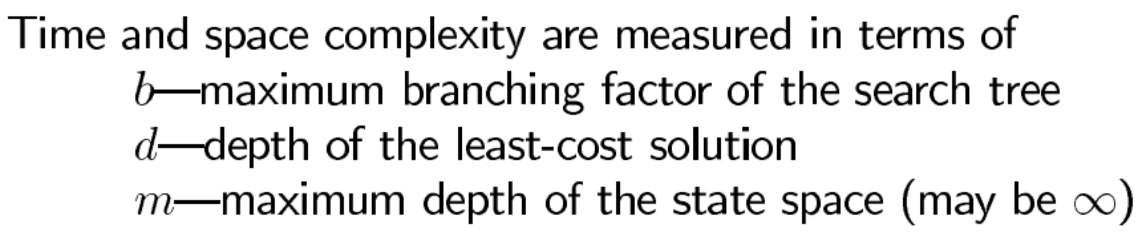
- first-in, first-out or **FIFO queue**, which pops the oldest element of the queue;
- last-in, first-out or **LIFO queue** (also known as a **stack**), which pops the *newest* element of the queue
- **priority queue**, which pops the element of the queue with the highest priority according to some ordering function.

The explored set can be implemented with a hash table to allow efficient checking for repeated states. With a good implementation, insertion and lookup can be done in roughly constant time no matter how many states are stored. One must take care to implement the hash table with the right notion of equality between states.

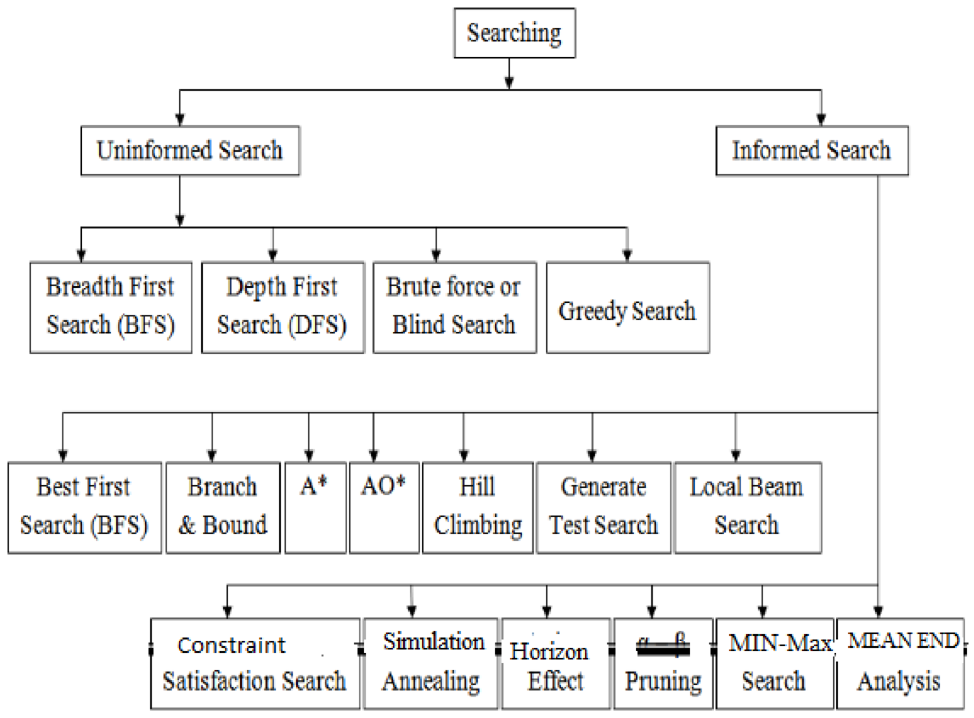
Measuring problem-solving performance
We can evaluate an algorithm's performance in four ways:

- **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
- **Optimality**: Does the strategy find the optimal solution?
- **Time complexity**: How long does it take to find a solution?
- **Space complexity**: How much memory is needed to perform the search?

Complexity is expressed in terms of three quantities: b, the **branching factor** or maximum number of successors of any node; d, the **depth** of the shallowest goal node and m, the maximum length of any path in the state space.


Time and space complexity are measured in terms of
b—maximum branching factor of the search tree
d—depth of the least-cost solution
m—maximum depth of the state space (may be ∞)

DIFFERENT TYPES OF SEARCHING



Uninformed Search Strategies
In this there is no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. They that do not take into account the location of the goal. These algorithms ignore where they are going until they find a goal and report success.

1. Breadth-First search

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 before any 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. This is achieved very simply by using a FIFO queue for the frontier new nodes go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first the goal test is 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.

```
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)
```

Figure 3.11 Breadth-first search on a graph.

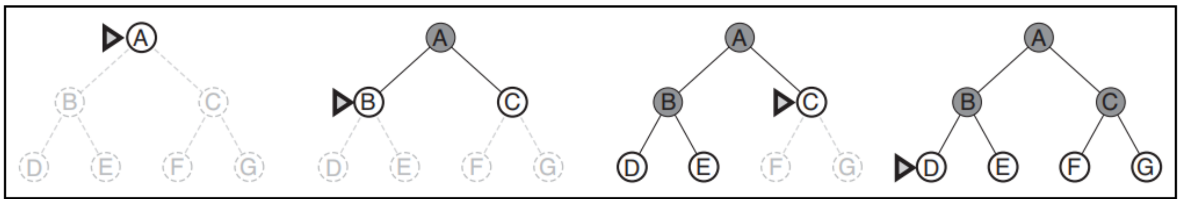


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Problem solving performance-BFS

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

not optimal one: breadth-first search is optimal if the path cost is a nondecreasing function of the 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 more nodes, 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 nodes generated 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

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

The memory requirements are a bigger problem for breadth-first search than is the execution time. One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take. If your problem has a solution at depth 16, then it will take about 350 years for breadth-first search to find it. In general, *exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.*

2. Uniform-cost search

Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the *lowest path cost* g(n). This is done by storing the frontier as a priority queue ordered by g.
the goal test is applied to a node when it is *selected for expansion because* the first goal node that is *generated* may be on a suboptimal path a test is added in case a better path is found to a node currently on the frontier.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        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
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

Here the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97 = 177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99 + 211 = 310.
Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+ 97+ 101 = 278. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

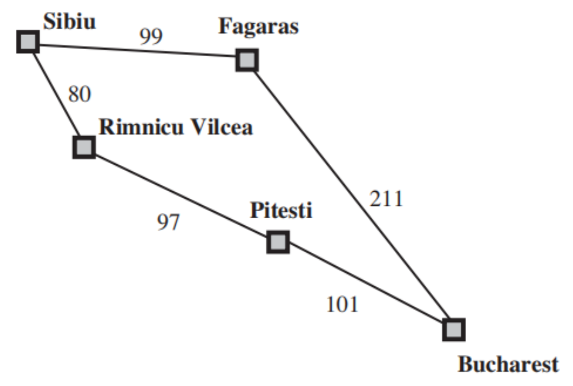


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

Problem solving performance-UCS

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 order of 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 is which can be much greater than b^d. This is because uniform cost search can explore large trees of 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 to see 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 always expands the *deepest* node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.

As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors. Depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent which, in turn, was the deepest unexpanded node when it was selected

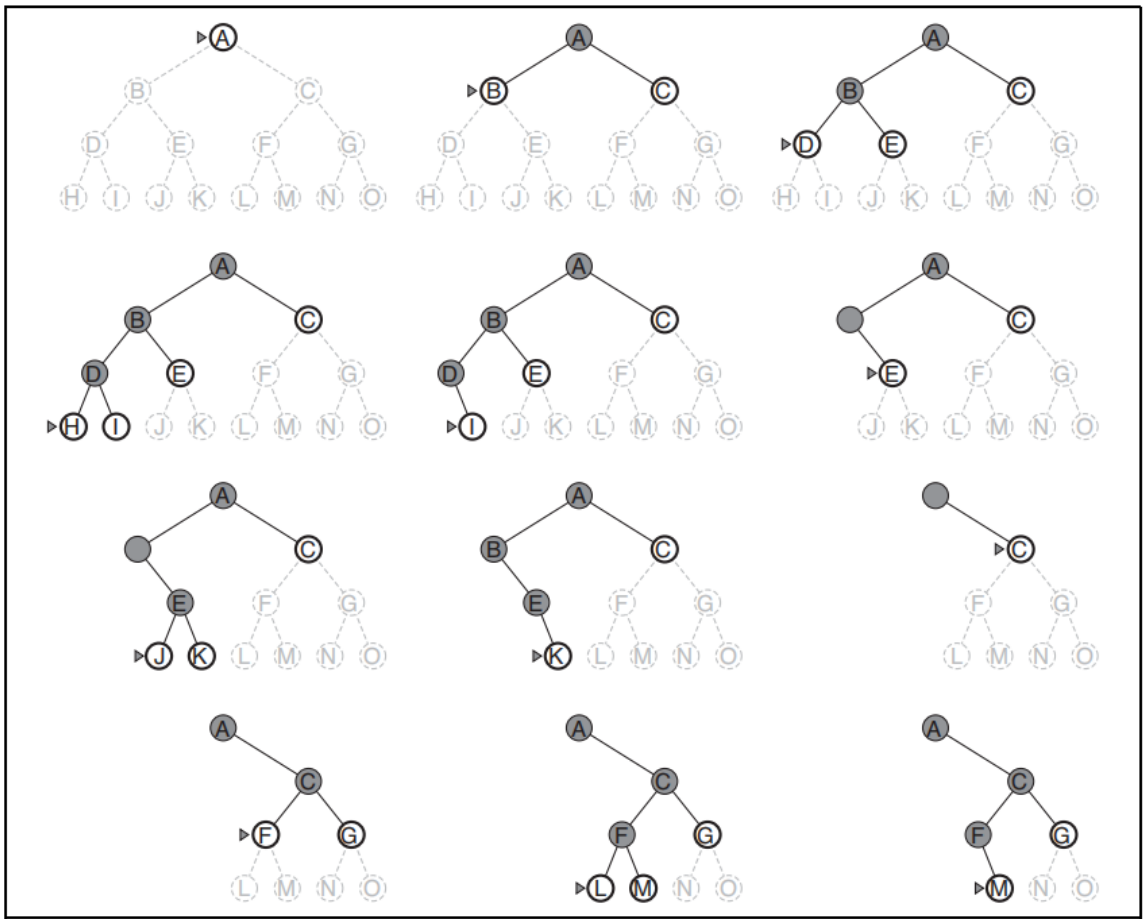


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and *M* is the only goal node.

Problem solving performance-DFS

1. **Completeness:**
   - depth-first search is implemented with a recursive function that calls itself on each of 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, is complete 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 checks new 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 proliferation of 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**.

Backtracking search

A variant of depth-first search called **backtracking search** uses still less memory only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. Only O(m) memory is needed rather than O(bm). Backtracking search facilitates the idea of generating a successor by *modifying* the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and O(m) actions. For this to work, we must be able to undo each modification when we go back to generate the next successor

Depth-limited search

Depth-limited search is depth-first search with a predetermined depth limit l nodes at depth l are treated as if they have no successors. The depth limit solves the infinite-path problem. It also introduces an additional source of incompleteness if we choose $l<d$, that is, the shallowest goal is beyond the depth limit. Depth-limited search will also be non optimal if we choose $l >d$. Its time complexity is $O(b^l)$ and its space complexity is O(bl). Depth-first search can be viewed as a special case of depth-limited search with $l=\infty$.
It is depth-first search with a **predefined** maximum depth. However, it is usually not easy to define the suitable maximum depth if it is too small then no solution can be found, if it is too large then the same problems are suffered from. Anyway the search is complete but still not optimal

```
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.

Iterative deepening depth-first search
Iterative deepening search is often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit—first 0, then 1, then 2, and so on until a goal is found. This will occur when the depth limit reaches d, the depth of the shallowest goal node.

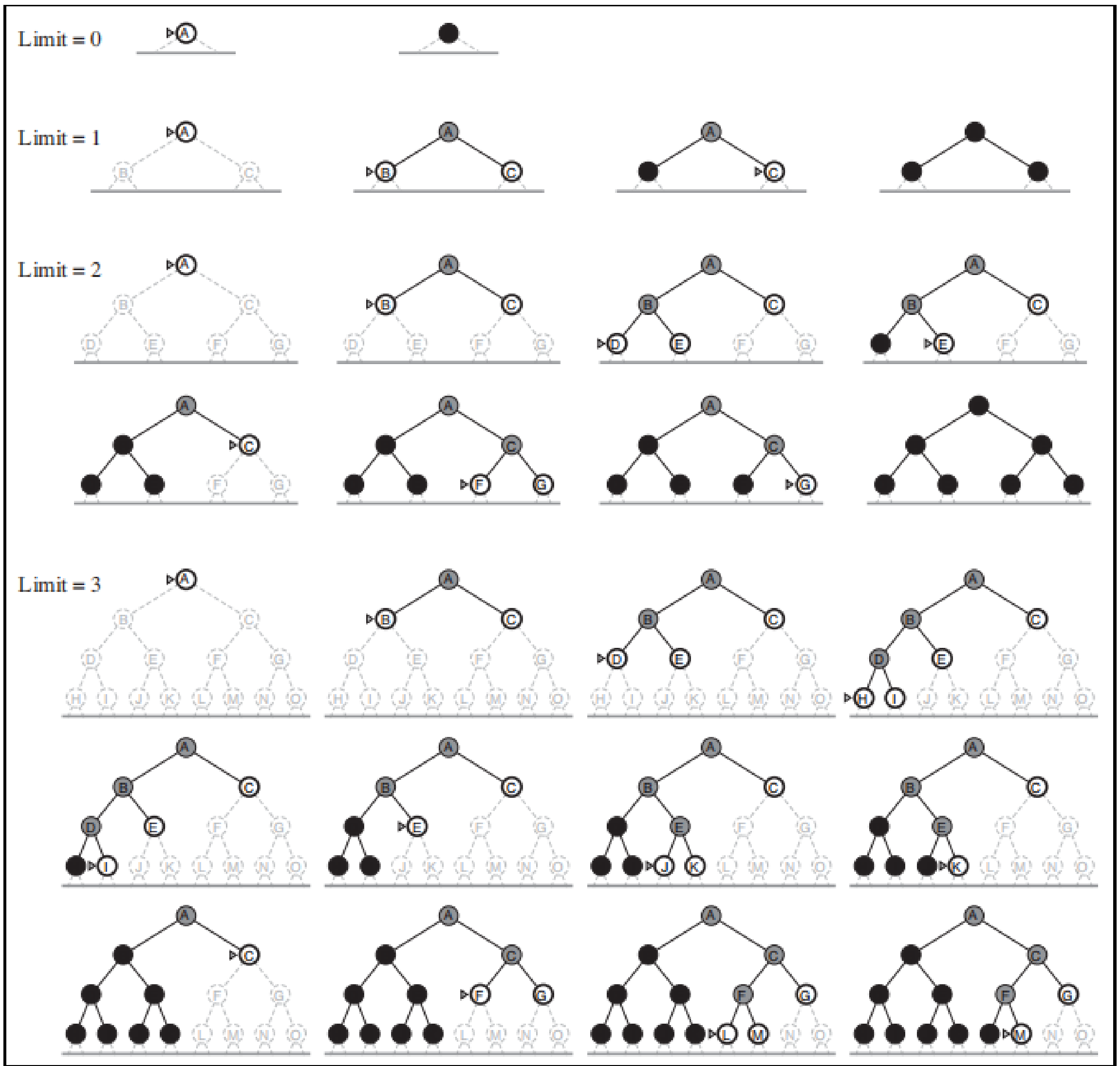


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
   for depth = 0 to ∞ do
      result ← DEPTH-LIMITED-SEARCH(problem, depth)
      if result ≠ cutoff then return result
```

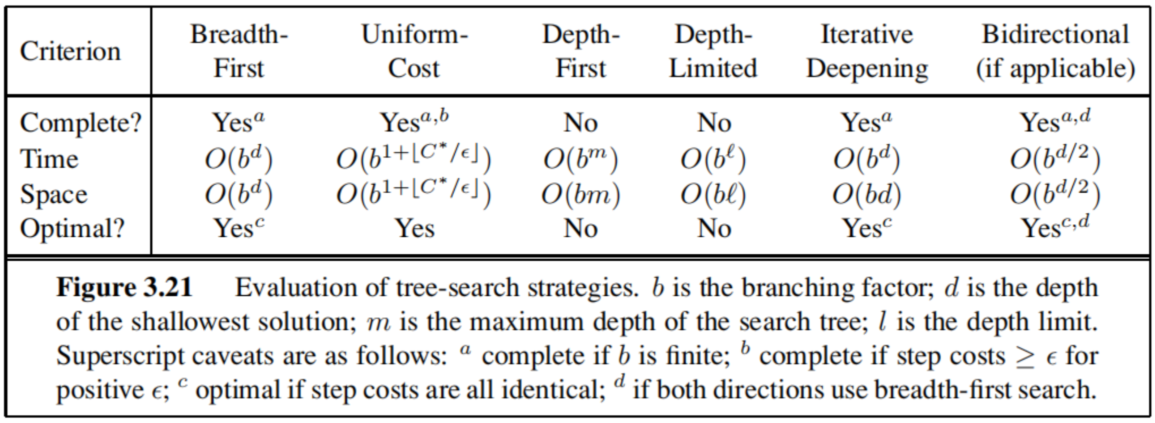
Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

Iterative deepening search may seem wasteful because states are generated multiple times but it is is not too costly. Because search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the

upper levels are generated multiple times. The nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times.
So the total number of nodes generated in the worst case is

$$N(IDS)=(d)b + (d - 1)b^2 + \cdots + (1)b^d ,$$

which gives a time complexity of $O(b^d)$ asymptotically the same as breadth-first search. Iterative deepening search is analogous to breadth-first search in that it explores a complete layer of new nodes at each iteration before going on to the next layer. *Iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known*



Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

INFORMED (HEURISTIC) SEARCH STRATEGIES

In Informed search problem-specific knowledge beyond the definition of the problem itself can find solutions more efficiently than an uninformed strategy

BEST-FIRST SEARCH

Best-first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function**, f(n). The evaluation function is construed as a cost estimate, so the node with the *lowest* evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search except for the use of *f* instead of *g* to order the priority queue. The choice of *f* determines the search strategy.
Best-first algorithms include as a component of *f* a **heuristic function**, denoted h(n): h(n) = estimated cost of the cheapest path from the state at node *n* to a goal state. If n is a goal node, then h(n)=0. For example, in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via the straight-line distance from Arad to Bucharest.

Greedy best-first search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. It evaluates nodes by using just the heuristic function; that is, f(n) = h(n).

Greedy best-first search- Route finding problems in Romania

Here the heuristic is straight line distance, H_{sld}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest. h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

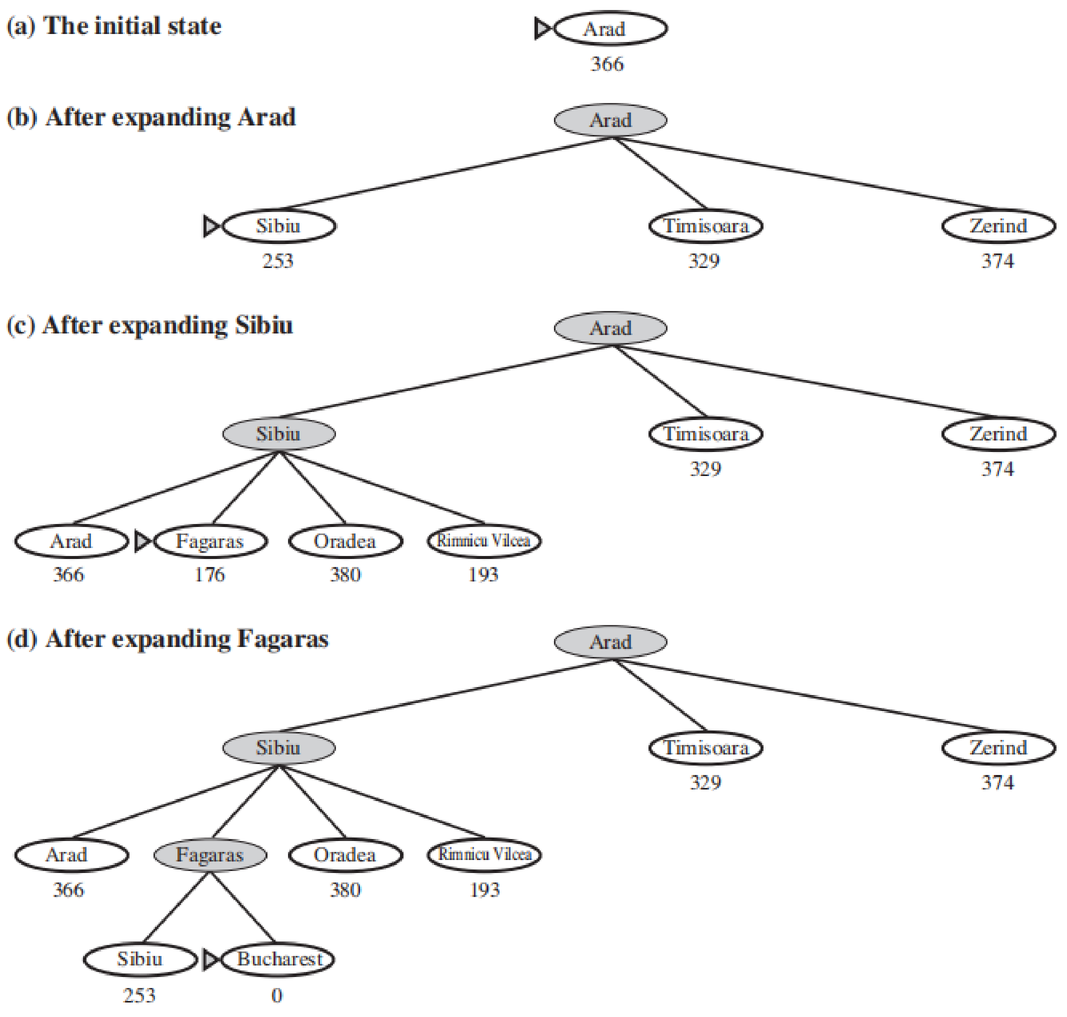
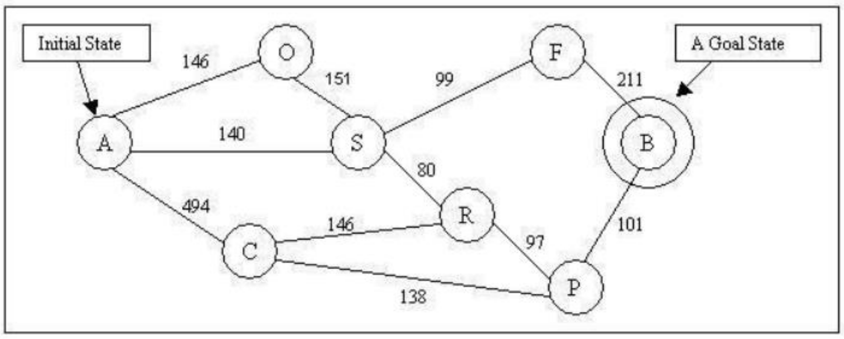
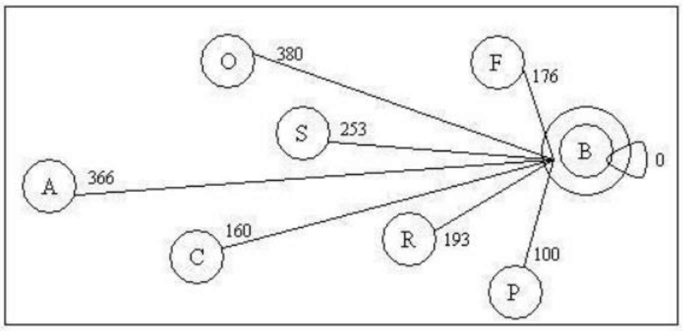


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

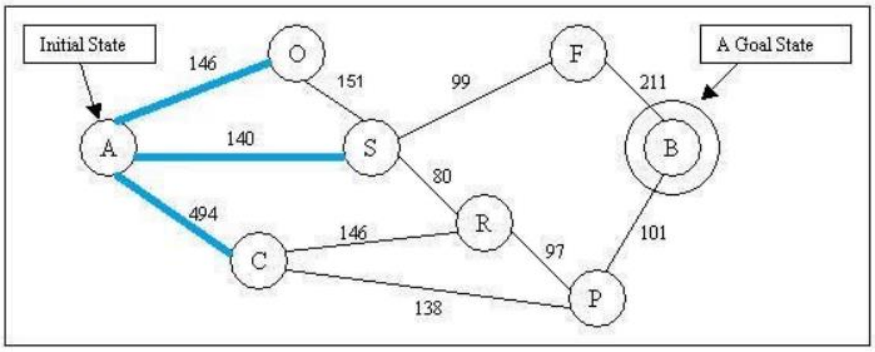
Now let's use an example to see how greedy best-first search works Below is a map that we are going to search the path on. For this example, let the valuation function f(n) simply be equal to the heuristic function h(n).



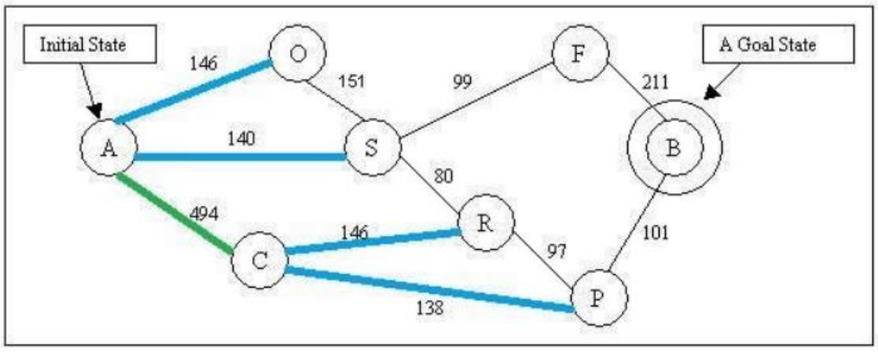
This is a map of a city. We are going to find out a path from city A to city B.



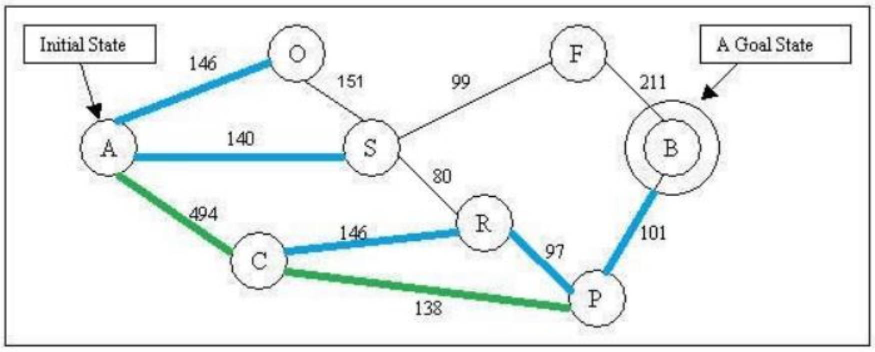
This graph shows the straight distance from each city to city B. The straight distances serve as each cities' h(n) in this example. Because $f(n) = h(n)$ our algorithm will at every stage choose to explore the node that we know is closest to our goal. Let's assume cities on the map as nodes and path between cities as edges. If you start exploring at node A, we have node O, S and C reachable now.



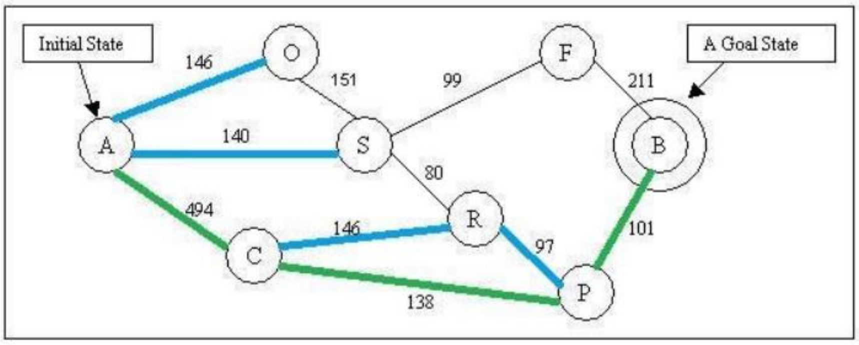
According to the estimate function $f(n) = h(n)$, and given by the graph of h(n) values above: Node O's f(n) = 380, Node S' f(n) = 253, Node C's f(n) = 160. The algorithm will choose to explore node C. After explored C, we have new node R and P reachable now.



According to the estimate function f(n): Node O's f(n) = 380, Node S' f(n) = 253, Node R's f(n) = 193, Node P's f(n) = 100. The algorithm will choose node P to explore. After explored C, we have new node B reachable now.



According to the estimate function f(n): Node O's f(n) = 380, Node S' f(n) = 253, Node R's f(n) = 193, Node B's f(n) = 0. The algorithm will choose node B to explore. After explored B, we have reached our goal state. The algorithm will be stopped and the path is found.



Note that in this example, the distance between the current node and next node does NOT inform our next step, only the heuristic given by the distance between potential nodes to the goal.

One can generalize the evaluation function of a target node to be a weighted sum of the heuristic function and the distance from the current node to that target, which could produce a different result.

A* search: Minimizing the total estimated solution cost

It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

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

Since g(n) gives the path cost from the start node to node n, and

h(n) is the estimated cost of the cheapest path from n to the goal, we have

f(n) = estimated cost of the cheapest solution through n

If we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of g(n) + h(n). It turns out that this strategy is more than just reasonable: provided that the heuristic function h(n) satisfies certain conditions, A∗ search is both complete and optimal. The algorithm is identical to UNIFORM-COST-SEARCH except that A∗ uses g + h instead of g.

Progress of an A∗ tree search

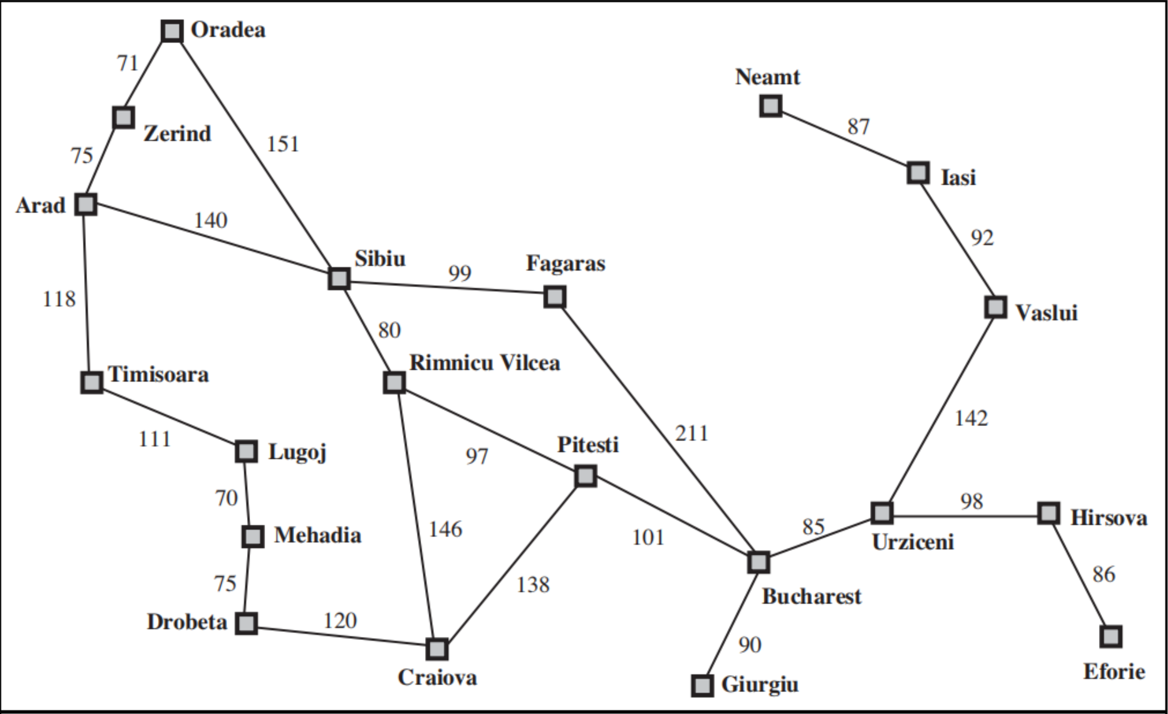


Figure 3.2 A simplified road map of part of Romania.

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figure 3.22 Values of h_{SLD}—straight-line distances to Bucharest.

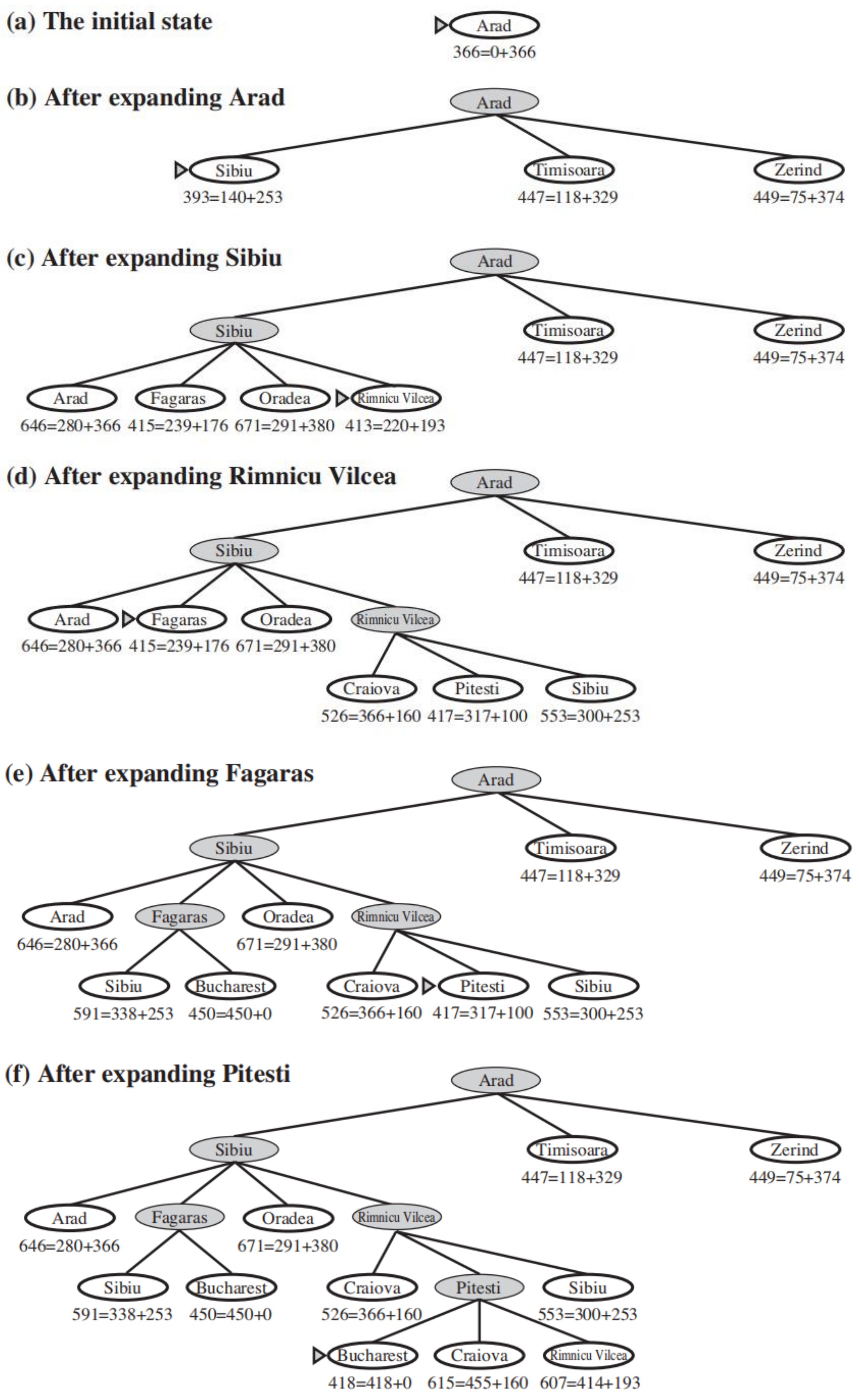


Figure 3.24 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.22.

Conditions for optimality: Admissibility

The first condition we require for optimality is that h(n) be an **admissible heuristic**. An admissible heuristic is one that *never overestimates* the cost to reach the goal. g(n) is the actual cost to reach n along the current path, and f(n) = g(n) + h(n), we have as an immediate consequence that f(n) never overestimates the true cost of a solution along the current path through n.

Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is

- Eg, straight-line distance h_{SLD} that we used in getting to Bucharest is n admissible heuristic
- because the shortest path between any two points is a straight line
- Straight line cannot be an overestimate

Conditions for optimality: Consistency

It is required only for applications of A∗ to graph search

A heuristic h(n) is consistent if, for every node n and every successor n of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n plus the estimated cost of reaching the goal from n :

$h(n) \leq c(n, a, n\ ) + h(n\ )$

This is a form of the general **triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n, n , and the goal Gn closest to n. For an admissible heuristic, the inequality makes perfect sense: if there were a route from n to Gn via n that was cheaper than h(n), that would violate the property that h(n) is a lower bound on the cost to reach Gn.

Optimality of A*

The tree-search version of A∗ *is optimal if* h(n) *is admissible, while the graph-search version is optimal if* h(n) *is consistent.* A∗ expands no nodes with f(n) > C∗—for example, Timisoara is not expanded in even though it is a child of the root. The subtree below Timisoara is **pruned**; because hSLD is admissible, the algorithm can safely ignore this subtree while still guaranteeing optimality. Pruning eliminates possibilities from consideration without having to examine them. A∗ is **optimally efficient** for any given consistent heuristic. That is, no other optimal algorithm is guaranteed to expand fewer nodes than A∗. This is because any algorithm that *does not* expand all nodes with f(n) < C∗ runs the risk of missing the optimal solution.

If h(n) is consistent, then the values of f(n) along any path are nondecreasing. The proof follows directly from the definition of consistency. Suppose n is a successor of n; then g(n) = g(n) + c(n, a, n) for some action a, and we have f(n) = g(n) + h(n) = g(n) + c(n, a, n) + h(n) ≥ g(n) + h(n) = f(n) .

Whenever A∗ selects a node n for expansion, the optimal path to that node has been found. Were this not the case, there would have to be another frontier node n on the optimal path from the start node to n. Because f is nondecreasing along any path, n would have lower f-cost than n and would have been selected first

The sequence of nodes expanded by A∗ using GRAPH-SEARCH is in nondecreasing order of f(n). Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have h = 0) and all later goal nodes will be at least as expensive.

Contours in the state space
Inside the contour labeled 400, all nodes have f(n) less than or equal to 400, and so on. Then, because A∗ expands the frontier node of lowest f-cost, we can see that an A∗ search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

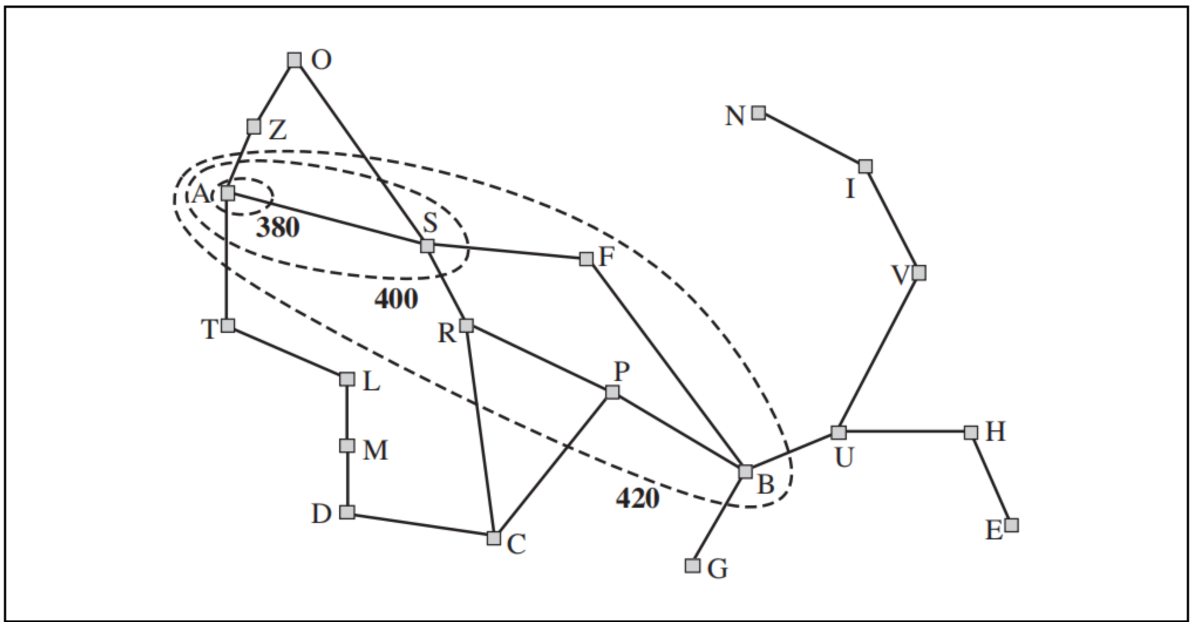


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

If C∗ is the cost of the optimal solution path, then we can say the following:

- A∗ expands all nodes with f(n) < C∗.
- A∗ might then expand some of the nodes right on the “goal contour” (where f(n) = C∗) before selecting a goal node.

Disadvantages of A* Algorithm
The number of states within the goal contour search space is still exponential in the length of the solution.

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

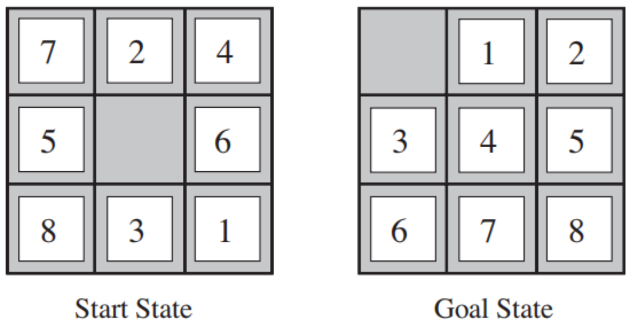


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

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 is clear 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 cannot move 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

Heuristic Performance

Experiments on sample problems can determine the number of nodes searched and CPU time 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

Experimental Results on 8-puzzle problems

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Figure 3.29 Comparison of the search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A* algorithms with h_1, h_2. Data are averaged over 100 instances of the 8-puzzle for each of various solution lengths 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 from square A to B.

If there are a number of features that indicate a promising or unpromising state, a weighted sum of these features can be useful. Learning methods can be used to set weights.