LECTURE NOTES– MODULE II

Problem Solving by Search and Exploration

ECS302

ARTIFICIAL INTELLIGENCE

Module II Lecture Notes

Syllabus

Problem Solving by Search and Exploration:

Solving Problems by Searching: *Problem solving agents, example problems, Searching for solutions, Uninformed search strategies, avoiding repeated states, searching with partial information.*

Informed Search and Exploration: *Informed (heuristic) search strategies, heuristic functions, local search algorithms and optimization problems, local search in continuous spaces.*

1. What is Search?

An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value, and then choosing the best sequence. The process of looking for sequences actions from the current state to reach the goal state is called **search.**

The simple reflex agents don't specifically search for best possible solution, as they are programmed to perform a particular action for a particular state. On the contrary, ***the artificially intelligent agents that work towards a goal, identify the action or series of actions that lead to the goal. The series of actions that lead to the goal becomes the solution for the given problem***. Here, the agent has to consider the impact of the action on the future states. Such agents search through all the possible solutions to find the best possible solution for the given problem.

Search is the systematic examination of **states** to find path from the **start/root state** to the **Goal State.** The set of possible states, together with ***operators*** defining their connectivity constitute the ***search space***. The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

1.1 Search Space

"*Unlike state space, which is a physical configuration, the search space is an abstract configuration represented by a search tree or graph of possible solutions*."

A search tree is used to model the sequence of actions. It is constructed with initial state as the root. The actions taken make the branches and the nodes are results of those actions.

A node has depth, path cost and associated state in the state space.

The search space is divided into 3 regions, namely

- **Explored**
- **Frontier**
- **Unexplored**

- ✓ Search involves moving the nodes from unexplored region to the explored region.
- ✓ Strategical order of these moves performs a better search. The moves are also known as ***node expansion.***
- ✓ Picking the order of node expansion has provided us with different search strategies that are suited for different kind of problems.

Different search strategies are evaluated along completeness, time complexity, space complexity and optimality. The time and space complexity are measured in terms of:

- ✓ **b:** maximum branching factor of the search tree (actions per state).
- ✓ **d**: depth of the solution
- ✓ **m**: maximum depth of the state space (may be ∞) (also noted sometimes D).

1.2 Types of Search

There are 2 kinds of search, based on whether they use information about the goal.

Uninformed Search

This type of search does not use any domain knowledge. This means that it does not use any information that helps it reach the goal, like closeness or location of the goal. The strategies or algorithms, using this form of search, ignore where they are going until they find a goal and report success.

The basic uninformed search strategies are:

- ✓ BFS (Breadth First Search): It expands the shallowest node (node having lowest depth) first.
- ✓ DFS (Depth First Search): It expands deepest node first.
- ✓ DLS (Depth Limited Search): It is DFS with a limit on depth.
- ✓ IDS (Iterative Deepening Search): It is DFS with increasing limit
- ✓ UCS (Uniform Cost Search): It expands the node with least cost (Cost for expanding the node).

Informed Search

This type of search uses domain knowledge. It generally uses a heuristic function that estimates how close a state is to the goal. This heuristic need not be perfect. This function is used to estimate the cost from a state to the closest goal.

The basic informed search strategies are:

- ✓ Greedy search (best first search) : It expands the node that appears to be closest to goal

- ✓ A* search: Minimize the total estimated solution cost that includes cost of reaching a state and cost of reaching goal from that state.

Search Agents are just one kind of algorithms in Artificial Intelligence. Here, an AI has to choose from a large solution space, given that it has a large action space on a large state space. Selecting the right search strategy for your Artificial Intelligence, can greatly amplify the quality of results. This involves formulating the problem that your AI is going to solve, in the right way.

Although, we are just scratching the surface, this article provides you with an outline of what kind of algorithms drive an AI. Depending on the problem, an Artificial Intelligence can use many other algorithms involving Machine Learning, Bayesian networks, Markov models, etc.

Solving Problems by Searching

1.3 Problem Solving by Search

An important aspect of intelligence is ***goal-based*** problem solving. The **solution** of many **problems** can be described by finding a **sequence of actions** that lead to a desirable **goal.** Each action changes the ***state*** and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.
A well-defined problem can be described by:

- **Initial state**
- **Operator or successor function** - for any state **x** returns **s(x)**, the set of states reachable from **x** with one action.
- **State space** - all states reachable from initial by any sequence of actions.
- **Path** - sequence through state space.
- **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path.
- **Goal test** - test to determine if at goal state.

2. Problem Solving Agent

A Problem solving agent is a **goal-based** agent. It decides what to do by finding sequence of actions that lead to desirable states. The agent can adopt a goal and aim at satisfying it. To illustrate the agent's behavior, let us take an example where our agent is in the city of Arad, which is in Romania. The agent has to adopt a **goal** of getting to Bucharest.

Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving. The agent's task is to find out which sequence of actions will get to a goal state.

Problem formulation is the process of deciding what actions and states to consider given a goal.

Goal Formation of Route Finding Problem

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

Problem formulation of Route Finding Problem

A **problem** is defined by four items:

The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as In(Arad) e.g., "at Arad"

successor function S(x) = set of action-state pairs

e.g., S (Arad) = {[Arad -> Zerind; Zerind]…}

goal test, can be

explicit, e.g., x = at Bucharest" ; implicit, e.g., NoDirt(x)

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.

Figure 2.1: Goal formulation and problem formulation

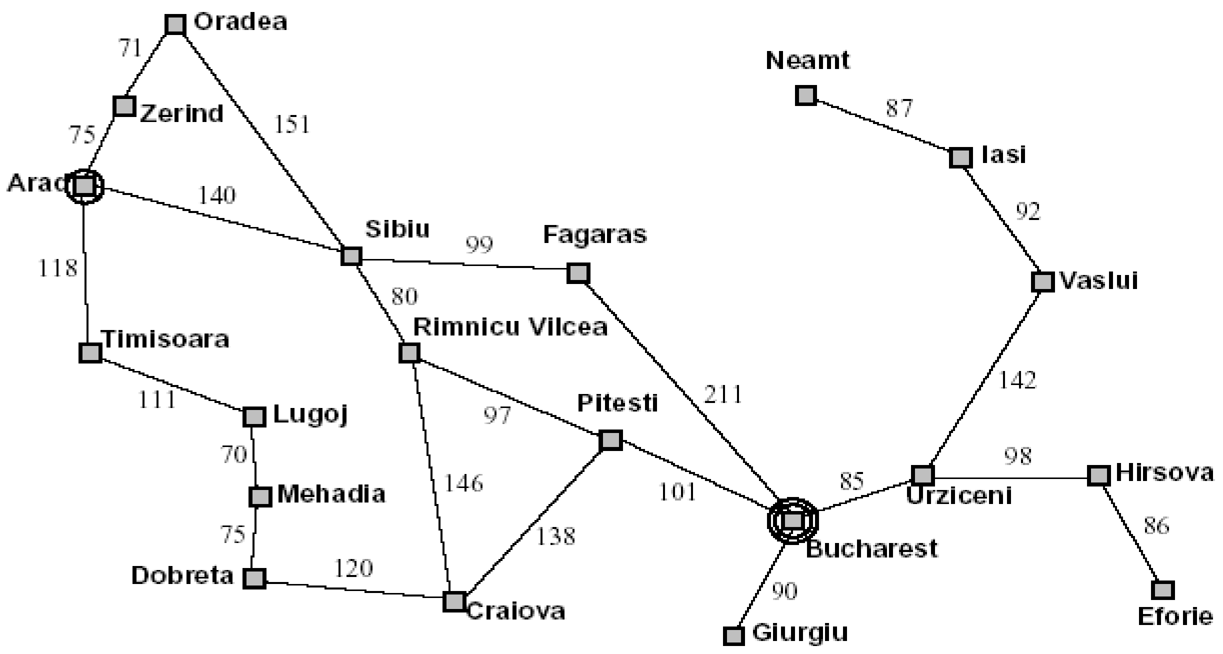


Figure 2.2: A simplified Road Map of part of Romania

The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.** Once a solution is found, the **execution phase** consists of carrying out the recommended action.

function SIMPLE-PROBLEM-SOLVING-AGENT(*percept*) **returns** an action

inputs : *percept*, a percept

```
static: seq, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation
state ← UPDATE-STATE(state, percept)
if seq is empty then do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH( problem)
  action ← FIRST(seq);
  seq ← REST(seq)
return action
```

Figure 2.3: A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over. Note that when it is executing the sequence it ignores its percepts: it assumes that the solution it has found will always work.

2.1 Well-defined problems and solutions

A problem can be defined formally by four components:

- INITIAL STATE The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as In (Arad).

- A description of the possible **actions** available to the agent. The most common for- SUCCESSORFUNCTION mulation3 uses a **successor function.** Given a particular state x, SUCCESSOR-FN(x) returns a set of (action, successor) ordered pairs, where each action is one of the legal actions in state x and each successor is a state that can be reached from x by applying the action. For example, from the state In(Arad), the successor function for the Romania problem would return

{(Go (Sibiu), In (Sibiu)), (Go (Timisoara), In (Tzmisoara)), (Go (Zerind), In (Zerind))}

Together, the initial state and successor function implicitly define the **state space** of the problem-the set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions. (The map of Romania shown in Figure 2.2 can be interpreted as a state space graph if we view PATH each road as standing for two driving actions, one in each direction.) A **path** in the state space is a sequence of states connected by a sequence of actions.

- GOAL TEST The **goal test,** which determines whether a given state is a goal state. Sometimes there is an explicit set of possible goal states, and the test simply checks whether the given state is one of them. The agent's goal in Romania is the singleton set {In (Bucharest)). Sometimes the goal is specified by an abstract property rather than an explicitly enumerated set of states. For example, in chess, the goal is to reach a state called "checkmate," where the opponent's king is under attack and can't escape.

- A PATH COST function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers. In this chapter, we assume that the cost of a path can be described as the STEP COST sum of the costs of the individual actions along the path. The **step cost** of taking action ***a*** to go from state x to state y is denoted by *c(x, a,* y). The step costs for Romania are shown in Figure 2.2 as route distances. We will assume that step costs are nonnegative.

The preceding elements define a problem and can be gathered together into a single data structure that is given as input to a problem-solving algorithm. A **solution** to a problem is a path from the initial state to a goal state. Solution quality is measured by the path cost OPTIMALSOLUTION function, and an **optimal solution** has the lowest path cost among all solutions.

Compare the sirnple state description we have chosen, *In (Arad),* to an actual cross-country trip, where the state of the world includes so many things: the traveling companions, what is on the radio, the scenery out of the window, whether there are any law enforcement officers nearby, how far lit is to the next rest stop, the condition of the road, the weather, and so on. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a representation is called **abstraction.**

3. Example Problems

The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between ***toy*** and ***real-world*** problems.

- A **toy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.
- A **real world problem** is one whose solutions people actually care about.

3.1 Toy Problems:

3.1.1 The first example we will examine is the vacuum world. This can be formulated as a problem as follows:

Problem Formation of the Vacuum World

States: The agent is in one of two locations, each of which might or might not contain dirt. Thus there are 2 x 2^2 = 8 possible world states.

Initial state: Any state can be designated as the initial state.

Successor function: This generates the legal states that result from trying the three actions (**Left**, ***Right*, **and ***Suck*)**. The complete state space is shown in Figure 3.1.1.

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.

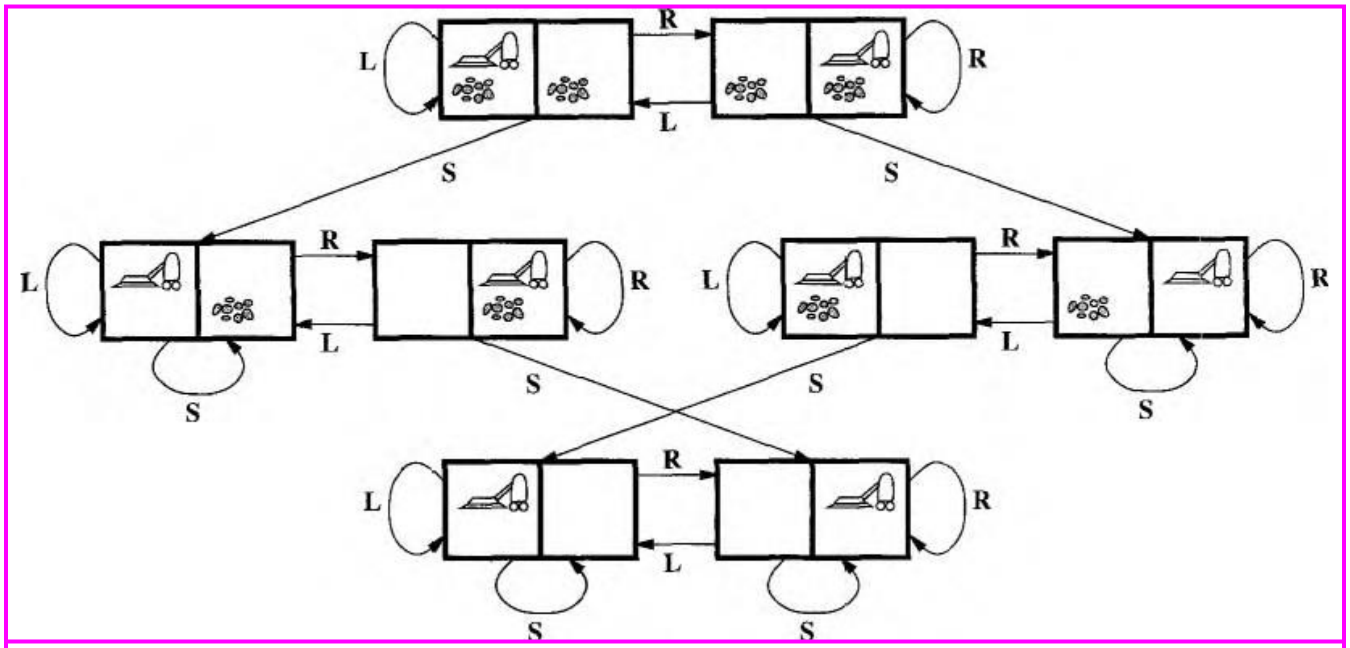


Figure 3.1.1: The state space for the vacuum world. Arcs denote actions: L = ***Left***, R = ***Right,*** S = **Suck**.

3.1.2 The second example we will examine the 8 –Puzzle problem. This can be formulated as a problem as follows:

Problem Formation of the 8 Puzzle Problem

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. It can be noted that any given goal can be reached from exactly half of the possible initial states.

Successor function: This generates the legal states that result from trying the four actions (blank moves **Left**, **Right**, **Up** or **down**).

Goal Test: This checks whether the state matches the goal configuration shown in Figure 3.1.2. (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 belongs to the family of **sliding-block puzzles,** which are often used as test problems for new search algorithms in AI. This general class is known to be NP-complete, so one does not expect to find methods significantly better in the worst case than the search algorithms

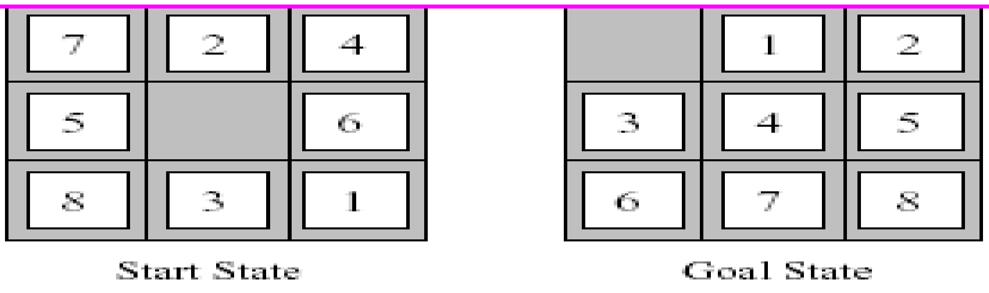


Figure 3.1.2: A typical instance of 8-puzzle.

3.1.3 The third example we will examine the Eight Queen's problem. This can be formulated as a problem as follows:

Problem Formation of the Eight Queen's Problem

Incremental Formulation

States: Any arrangement of 0 to 8 queens on board is a state.

Initial state: No queen on the board.

Successor function: Add a queen to any empty square.

Goal Test: 8 queens are on the board, none attacked.

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

Complete State Formation

States: Arrangements of n queens ($0 <= n < = 8$), one per column in the left most columns, with no queen attacking another are states.

Successor function: Add a queen to any square in the left most empty column such that it is not attacked by any other queen.

Although efficient special-purpose algorithms exist for this problem and the whole n-queens family, it remains an interesting test problem for search algorithms. 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 queue 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.

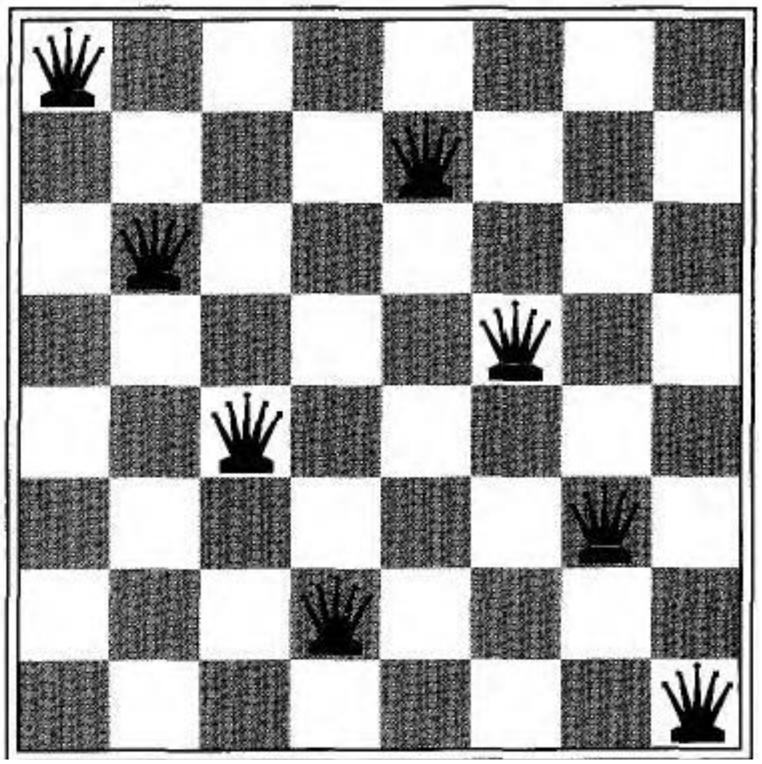


Figure 3.1.3: Almost a solution to the 8-queens problem. **(Solution is left as an exercise.)**

3.2 Real World Problems

Some of the real world examples, where they are being used are:

- **Route finding problem**: 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, such as routing in computer networks, military operations planning, and air line travel planning systems. Examples of applications include tools for driving directions in websites, in-car systems, etc.

Formulation of the airline travel problem
States: Each is represented by a location (e.g., an airport) and the current time. **Initial state:** This is specified by the problem. **Successor function:** This returns the states resulting from taking any scheduled flight (further specified by seat class and location), leaving later than the current time plus the within-airport transit time, from the current airport to another. **Goal Test:** Are we at the destination by some prespecified time? **Path cost:** This depends upon the monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of date, type of air plane, frequent-flyer mileage awards, and so on.

- **Travelling salesman problem**: Find the shortest tour to visit each city exactly once. 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**. Enormous efforts have been expended to improve the capabilities of TSP algorithms. These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.

- **VLSI Layout**: position million of components and connections on a chip to minimize area, shorten delays. 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 is split into two parts: **cell layout** and **channel routing**.

- **Robot Navigation**: Special case of route finding for robots with no specific routes or connections, where state space and action space are potentialy infinite. **Robot Navigation** is a generalization of the route-finding problem. Rather than 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 also must be controlled, the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

- **Automatic assembly sequencing**: find an order in which to assemble parts of an object which is a difficult and expensive geometric search. The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is chosen, there will be no way to add some part later without undoing some work already done.

- **Protein design**: find a sequence of amino acids that will fold into a 3D protein with the right properties to cure some disease.
- **Internet Searching:** In recent years there has been increased demand for software robots that perform Internet searching. , looking for answers to questions, for related information, or for shopping deals. The searching techniques consider internet as a graph of nodes (pages) connected by links.

4. Searching for solutions

Having formulated some problems, we now need to solve them. This is done by a search through the state space. A **search tree** is generated by the **initial state** and the **successor function** that together define the **state space**. In general, we may have a ***search graph*** rather than a *search tree*, when the same state can be reached from multiple paths. The choice of which state to expand is determined by the **search strategy.**

There are many ways to represent nodes, but we will assume that a node is a data structure with five Components:

- **STATE**: the state in the state space to which the node corresponds;
- **PARENT-NODE**: the node in the search tree that generated this node;
- **ACTION**: the action that was applied to the parent to generate the node;
- **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; and
- **DEPTH**: the number of steps along the path from the initial state.

We also need to represent the collection of nodes that have been generated but not yet expanded-this collection is called the **fringe.** Each element of the fringe is a **leaf node,** that is, a node with no successors in the tree.

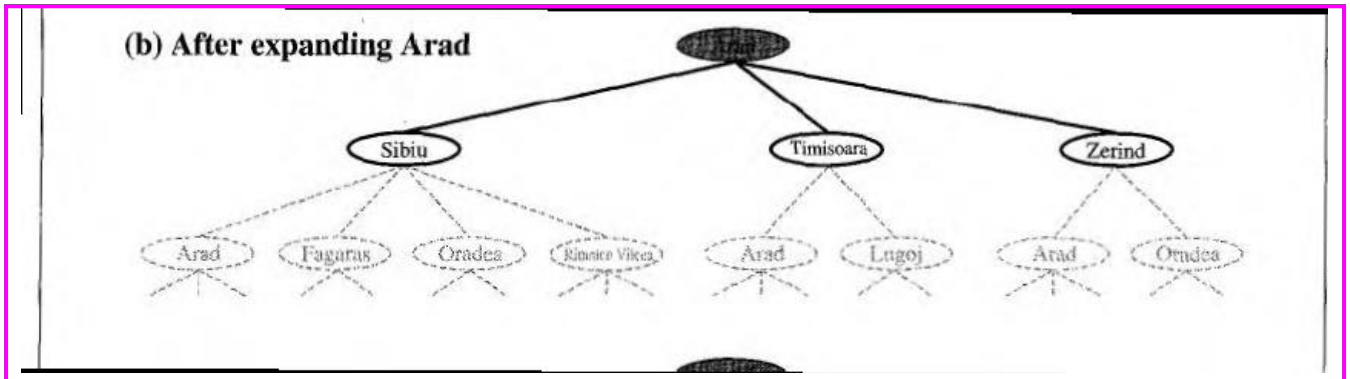


Figure 4: Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

```
function T R E E - S E A R C H (problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

Figure 4.1: An informal description of the general tree-search algorithm.

The fringe of each tree consists of those nodes with bold outlines. The simplest representation of the fringe would be a set of nodes. The search strategy then would be a function that selects the next node to be expanded from this set. Although this is conceptually straightforward, it could be computationally expensive, because the strategy function might have to look at every element of the set to choose the best one. Therefore, we will assume that the collection of nodes is implemented as a **queue.** The operations on a queue are as follows:

- **MAKE-QUEUE (element…)** creates a queue with the given element(s).
- **EMPTY? (queue)** returns true only if there are no more elements in the queue.
- **FIRST (queue)** returns FIRST (queue) and removes it from the queue.

- **INSERT (element, queue)** inserts an element into the queue and returns the resulting queue.
- **INSERT-ALL(elements, queue)** inserts a set of elements into the queue and returns the resulting queue.

With these definitions, we can write the more formal version of the general tree-search algorithm

```
function T R E E - S E A R C H (problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL- STATE[problem]), fringe)
loop do
    if EMPTY?( fringe) then return failure
     node  ←REMOVE-FIRST (fringe)
    if GOAL-TEST [problem] applied to STATE[node] succeeds
        then return SOLUTION (node )
   fringe ←INSERT-ALL ( EXPAND(problem,) , fringe)
```

```
function EXPAND ( problem, fringe ) returns a set of nodes
   successors t the empty set
  for each (action, result) in SUCCESSOR-FN [ problem ] ( STATE [ node ] ) do
       s ←a new NODE
      STATE[s]←result
      PARENT-NODE[s]  ←node
      ACTION[s] ←action
     PATH-COST[s] ← PATH-COST [node] + STEP-COST(STATE[node], action, result)
     DEPTH[s] ← DEPTH[node]+ 1
     add s to successors
  return successors
```

Figure 4.2: The general tree-search algorithm. (Note that the *fringe* argument must be an empty queue, and the type of the queue will affect the order of the search.) The *SOLUTION* function returns the sequence of actions obtained by following parent pointers back to the root.

4.1 Measuring problem-solving performance

The output of a problem-solving algorithm is either *failure* or a solution. (Some algorithms might get stuck in an infinite loop and never return an output.) We will 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, as defined?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

In AI, where the graph is represented implicitly by the initial state and successor function and is frequently infinite, 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.

5. Uninformed search strategies

Uninformed Search [Blind Search] Strategies have no additional information about states beyond that provided in the **problem definition**. **Strategies** that know whether one non goal state is "more promising" than another are called **informed search or heuristic search** strategies.

There are five uninformed search strategies as given below.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

5.1 Breadth First Search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, 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 can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue, assuring that the nodes that are visited first will be expanded first. In other words, calling **TREE-SEARCH (**problem, **FIFO-QUEUE** ()) results in a breadth-first search. The FIFO queue puts all newly generated successors at the end of the queue, which means that shallow nodes are expanded before deeper nodes.

Now suppose that the solution is at depth d. In the worst case, we would expand all but the last node at level d (since the goal itself is not expanded), generating $b^{d+1} - b$ nodes at level d + 1. Then the total number of nodes generated is

$$b + b^2 + b^3 + \cdots + b^d + (b^{d+1} - b) = O(b^{d+1}) .$$

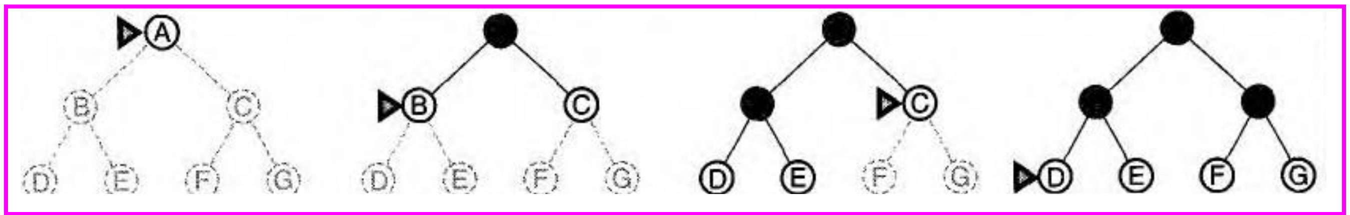


Figure 5.1: Breadth First Search on a Simple Binary Tree. At each Stage, the node to be expanded next is indicated by a marker.

Evaluation Criterion of Breadth First Search

- **Complete**: **YES [if *b* is finite]**
- **Time Complexity**: **O (b^{d+1})**
- **Space Complexity**: **O (b^{d+1}) [Keeps every node in Memory]**
- **Optimal**: **YES [optimal if step costs are all identical]**

Drawbacks of BFS

- The memory requirements are a bigger problem for breadth First search than is the execution time
- Exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.

5.2 Uniform Cost Search

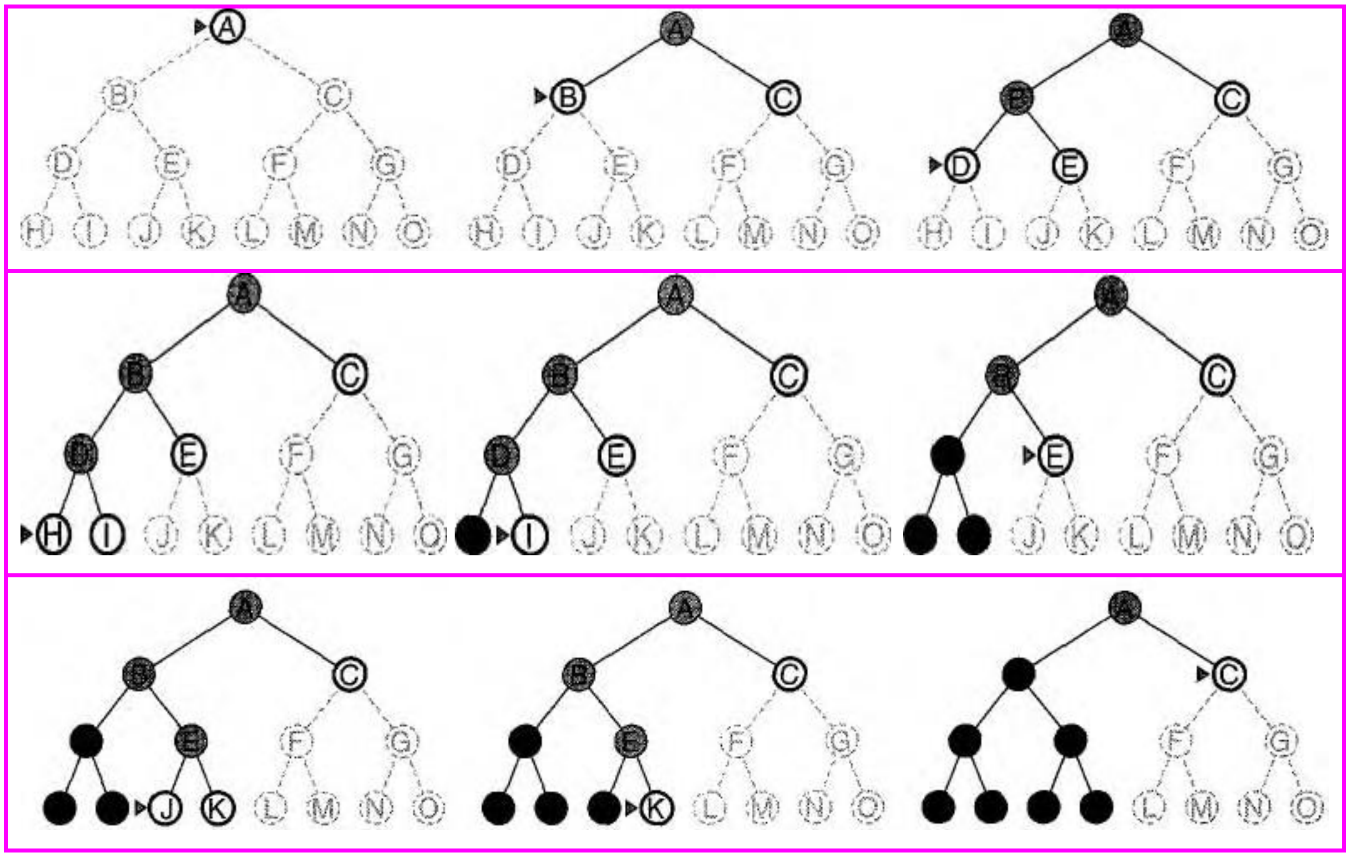
Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost. Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Note that if all step costs are equal, this is identical to breadth-first search.

Evaluation Criterion of Uniform Cost Search

- **Complete**: **YES [if step cost ≥ Small Positive constant '€' and *b* is finite]**
- **Time Complexity**: **O ($b^{1+[C^*/€]}$) [C* is the optimal solution]**
- **Space Complexity**: **O ($b^{1+[C^*/€]}$) [Keeps every node in Memory]**
- **Optimal**: **YES[nodes expand in the increasing order of g(n)]**

5.3 Depth First Search

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in figure 5.3. 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 fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors. This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.



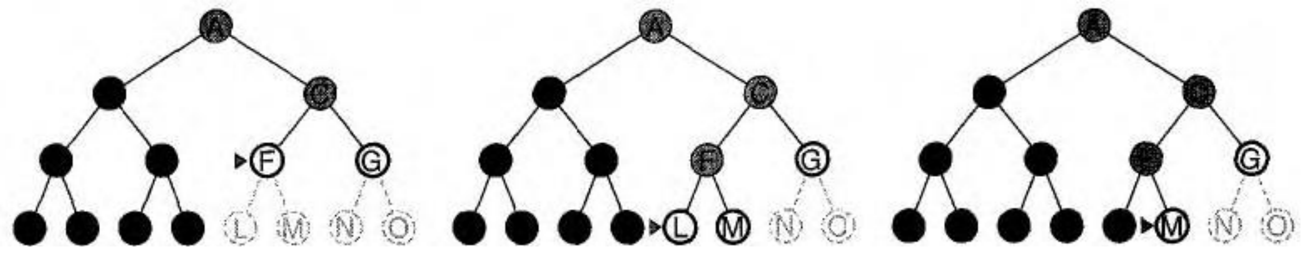


Figure 5.3: Depth-first search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

Evaluation Criterion of DFS

- **Complete**: **NO**
- **Time Complexity**: **O (b^m)** **[where *m* is the maximum depth of any node**]
- **Space Complexity**: **O (bm)**
- **Optimal**: **NO [Goes in to infinite path]**

Drawbacks of DFS

- It can make a wrong choice and get stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree.
- In the worst case, depth-first search will generate all of the O (bm) nodes in the search tree, where m is the maximum depth of any node. Note that m can be much larger than d (the depth of the shallowest solution), and is infinite if the tree is unbounded.

5.4 Depth Limited Search

The problem of unbounded trees can be alleviated by supplying depth-first search with a predetermined depth limit ***l***. That is, nodes at depth ***l*** are treated as if they have no successors. This approach is called **depth-limited search.** The depth limit solves the infinite-path problem.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution or failurelcutoff
    return RECURSIVE-DLS (MAKE-NODE(INITIAL- STATE[problem]), problem, limit)
```

```
function RECURSIVE-DLS (node, problem, limit) returns a solution, or failurelcutoff
    cutoff-occurred? ← false
    if GOAL-TEST [problem] (STATE[node] ) then return SOLUTION (node )
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND (node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result # failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Figure 5.4: A recursive implementation of depth-limited search.

Evaluation Criterion of DLS

- **Complete: NO**
- **Time Complexity: O (b^l)** [nodes at depth ***l***]
- **Space Complexity: O (bl)**
- **Optimal: NO**

Drawbacks of DLS

- Unfortunately, it also introduces an additional source of incompleteness if we choose **l < d**, that is, the shallowest goal is beyond the depth limit. (This is not unlikely when **d** is unknown.) Depth-limited search will also be nonoptimal 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** = *infinity*.

5.5 Iterative Deepening Search

Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search that finds the better 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. Iterative deepening combines the benefits of depth-first and breadth-first-search Like depth-first-search, its memory requirements are modest; O (bd) to be precise. Like Breadth-first-search, it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.

In general, iterative deepening is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.

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

Figure 5.5 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.

Evaluation Criterion of IDS

- **Complete: YES [if *b* is finite]**
- **Time Complexity: O (b^d) [nodes at depth *d*]**
- **Space Complexity: O (bd)**
- **Optimal: YES [optimal if step costs are all identical]**

Drawbacks of IDS

- The idea is to use increasing path-cost limits instead of increasing depth limits. The resulting algorithm, called **iterative lengthening search**. It turns out, unfortunately, that iterative lengthening incurs substantial overhead compared to uniform-cost search.

5.6 Comparing Uninformed Search Strategies

Figure 5.6 compares search strategies in terms of the four evaluation criteria. In this we are going to compare BFS, Uniform cost, DFS, DLS, IDS and Bidirectional. We will have the comparison in the Complete, Optimal, Time and Space Complexities.

Criterion\ Search Algorithm	BFS	Uniform Cost	DFS	DLS	IDS	Bidirectional
Complete	YES[a]	YES[a,b]	NO	NO	YES[a]	YES[a,d]
Time complexity	O (b^{d+1})	O $(b^{1+[C^*/€]})$	O (b^m)	O (b^l)	O (b^d)	O $(b^{d/2})$
Space complexity	O (b^{d+1})	O $(b^{1+[C^*/€]})$	O (bm)	O (bl)	O (bd)	O $(b^{d/2})$
Optimal	YES[c]	YES	NO	NO	YES[c]	YES[c,d]

Figure 5.6 Evaluation of search strategies. ***b*** is the branching factor; **d** is the depth of the shallowest solution; **m** is the maximum depth of the search tree; 1 is the depth limit. Superscript caveats are as follows: **a** complete if ***b*** is finite; **b** complete if step costs ≥ ***€*** for positive **€; *c*** optimal if step costs are all identical; **d** if both directions use breadth-first search.

6. Avoiding Repeated States

In searching, time is wasted by expanding states that have already been encountered and expanded before. For some problems repeated states are unavoidable. The search trees for these problems are infinite. If we prune some of the repeated states, we can cut the search tree down to finite size. Considering search tree up to a fixed depth, eliminating repeated states yields an exponential reduction in search cost. Repeated states can cause a solvable problem to become unsolvable if the algorithm does not detect them. Repeated states can be the source of great inefficiency: identical sub trees will be explored many times.

If an algorithm remembers every state that it has visited, then it can be viewed as exploring the state-space graph directly. We can modify the general TREE-SEARCH algorithm CLOSED LIST to include a data structure called the closed list, which stores every expanded node. (The OPEN LIST fringe of unexpanded nodes is sometimes called the open list.) If the current node matches a node on the closed list, it is discarded instead of being expanded. The new algorithm is called GRAPH-SEARCH (**Figure 6.1**). On problems with many repeated states, GRAPH-SEARCH is much more efficient than TREE-SEARCH. Its worst-case time and space requirements are proportional to the size of the state space. This may be much smaller than ***O* (b^d).**

```
function GRAPH-SEARCH (fringe) returns a solution, or failure
closed ← an empty set
fringe ←INSERT(MAKE-NODE(INITIAL-STATE [problem],fringe)
loop do
if EMPTY?(fringe) then return failure
node ← REMOVE-FIRST (fringe )
if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
if STATE(node) is not in closed then
add STATE [node] to close d
fringe ←INSERT-ALL (EXPAND (node, problem) , fringe )
```

Figure 6.1: The general graph-search algorithm. The set *closed* can be implemented with a hash table to allow efficient checking for repeated states. This algorithm assumes that the first path to a state *s* is the cheapest.

The set closed can be implemented with a hash table to allow efficient checking for repeated states.

- Do not return to the previous state.
- Do not create paths with cycles.
- Do not generate the same state twice.
- Store states in a hash table.
- Check for repeated states.
  - ✓ Using more memory in order to check repeated state
  - ✓ Algorithms that forget their history are doomed to repeat it.
  - ✓ Maintain Close-List beside Open-List(fringe)

7. Searching with Partial Information

We assumed that the environment is fully observable and deterministic and that the agent knows what the effects of each action are. What happens when knowledge of the states or actions is incomplete? We find that different types of incompleteness lead to three distinct problem types:

- **Sensorless problems** (*conformant*): If the agent has no sensors at all, then (as far as it knows) it could be in one of several possible initial states, and each action might therefore lead to one of several possible successor states.
- **Contingency problem**: If the environment is partially observable or if actions are uncertain, then the agent's percepts provide *new* information after each action. Each possible percept defines a contingency that must be planned for. A problem is called **adversarial** if the uncertainty is caused by the actions of another agent.
- **Exploration problems**: When the states and actions of the environment are unknown, the agent must act to discover them. Exploration problems can be viewed as an extreme case of contingency problems.

Partial knowledge of states and actions:

- *Sensorless or conformant problem*
  - ✓ Agent may have no idea where it is; solution (if any) is a sequence.
- *Contingency problem*
  - ✓ Percepts provide *new* information about current state; solution is a tree or policy; often interleave search and execution.
  - ✓ If uncertainty is caused by actions of another agent: *adversarial problem*
- *Exploration problem*
  - ✓ When states and actions of the environment are unknown.

Informed Search and Exploration

8. Informed (heuristic) search strategies

Informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself. It can find solutions more efficiently than uninformed strategy.

The general approach we will consider is called **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) . Traditionally, the node with the *lowest* evaluation is selected for expansion, because the evaluation measures distance to the goal. Best-first search can be implemented within our general search framework via a priority queue, a data structure that will maintain the fringe in ascending order of f -values.

A **heuristic function** or simply a **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on available information in order to make a decision which branch is to be followed during a search. Heuristic functions are the most common form in which additional knowledge is imparted to the search algorithm

The key component of Best-first search algorithm is a **heuristic function**, denoted by **h (n):**

h (n) = estimated cost of the **cheapest path** from node n to a **goal node**.

We will consider heuristic function to be arbitrary problem-specific functions, with one constraint: 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 a **straight-line distance** from Arad to Bucharest.

8.1 Greedy Best First Search

Greedy best-first search3 tries to expand the node that is closest to the goal, on the: grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function:

$$\boxed{f\ (n) = h\ (n).}$$

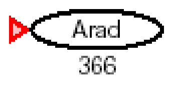
Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in **Figure 8.1**. For example, the initial state is In (Arad), and the straight line distance heuristic h_{SLD} (In (Arad)) is found to be 366. Using the **straight-line distance** heuristic h_{SLD}**,** the goal state can be reached faster.

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

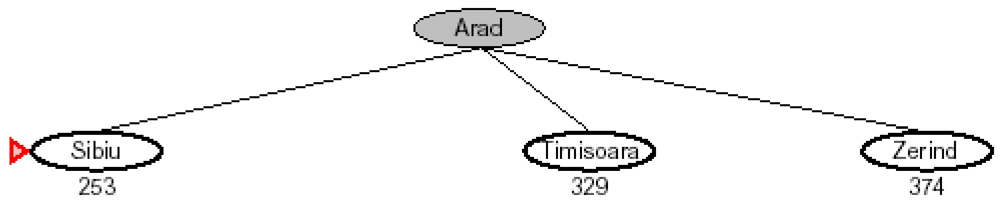
Giurgiu	77	**Timisoara**	329

Figure 8.1: Values of h_{SLD}-straight-line distances to B u c h a r e s t.

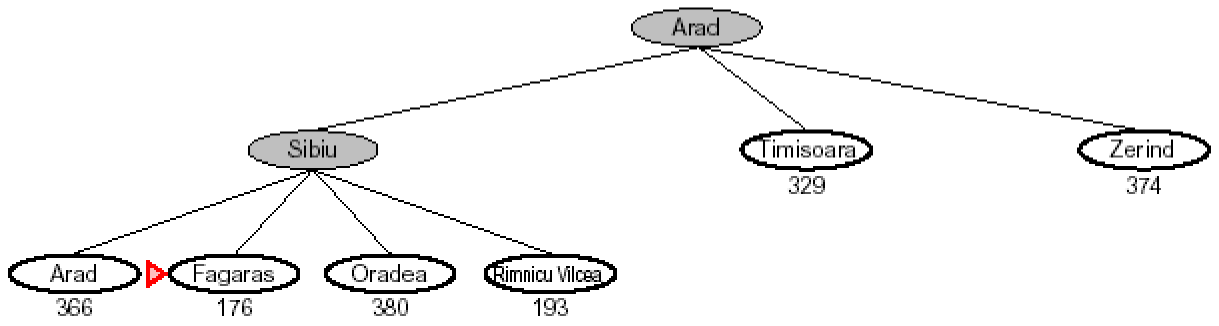
The Initial State



After Expanding Arad



After Expanding Sibiu



After Expanding Fagaras

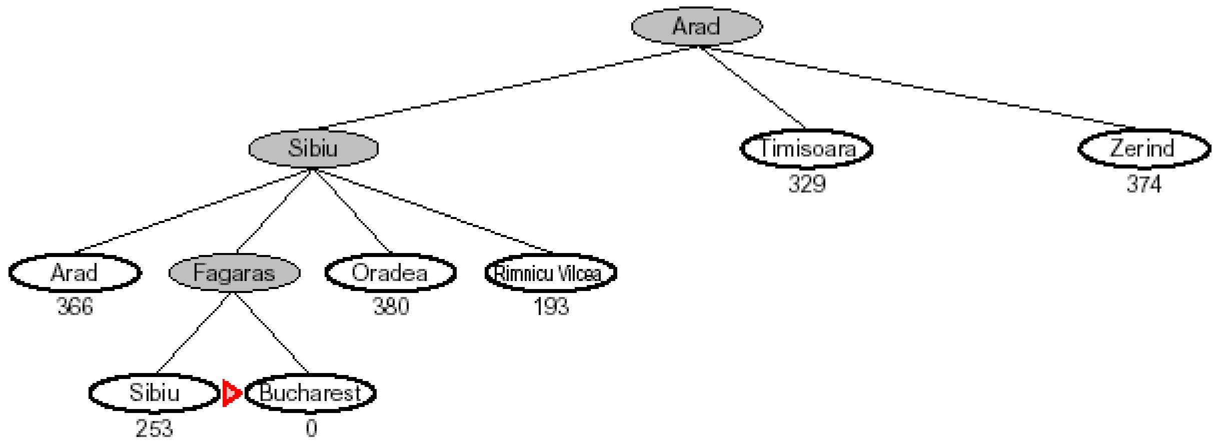


Figure 8.2: Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Figure 8.2 shows the progress of greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest. Fagaras in turn generates Bucharest, which is the goal.

Evaluation Criterion of Greedy Search

- **Complete**: **NO** **[can get stuck in loops, e.g., Complete in finite space with repeated-state checking**]
- **Time Complexity**: **O (bm)** **[but a good heuristic can give dramatic improvement**]
- **Space Complexity**: **O (bm)** **[keeps all nodes in memory**]
- **Optimal**: **NO**

Greedy best-first search is not optimal, and it is incomplete. The worst-case time and space complexity is **$O(b^m)$,** where m is the maximum depth of the search space.

8.2 A* Search

The most widely-known form of best-first search is called **A*** search .It evaluates nodes by combining ***g (n)***, t he cost to reach the node, and ***h (n.),*** the cost to get from the node to the goal:

$$\boxed{f(n) = g(n) + h(n)}$$

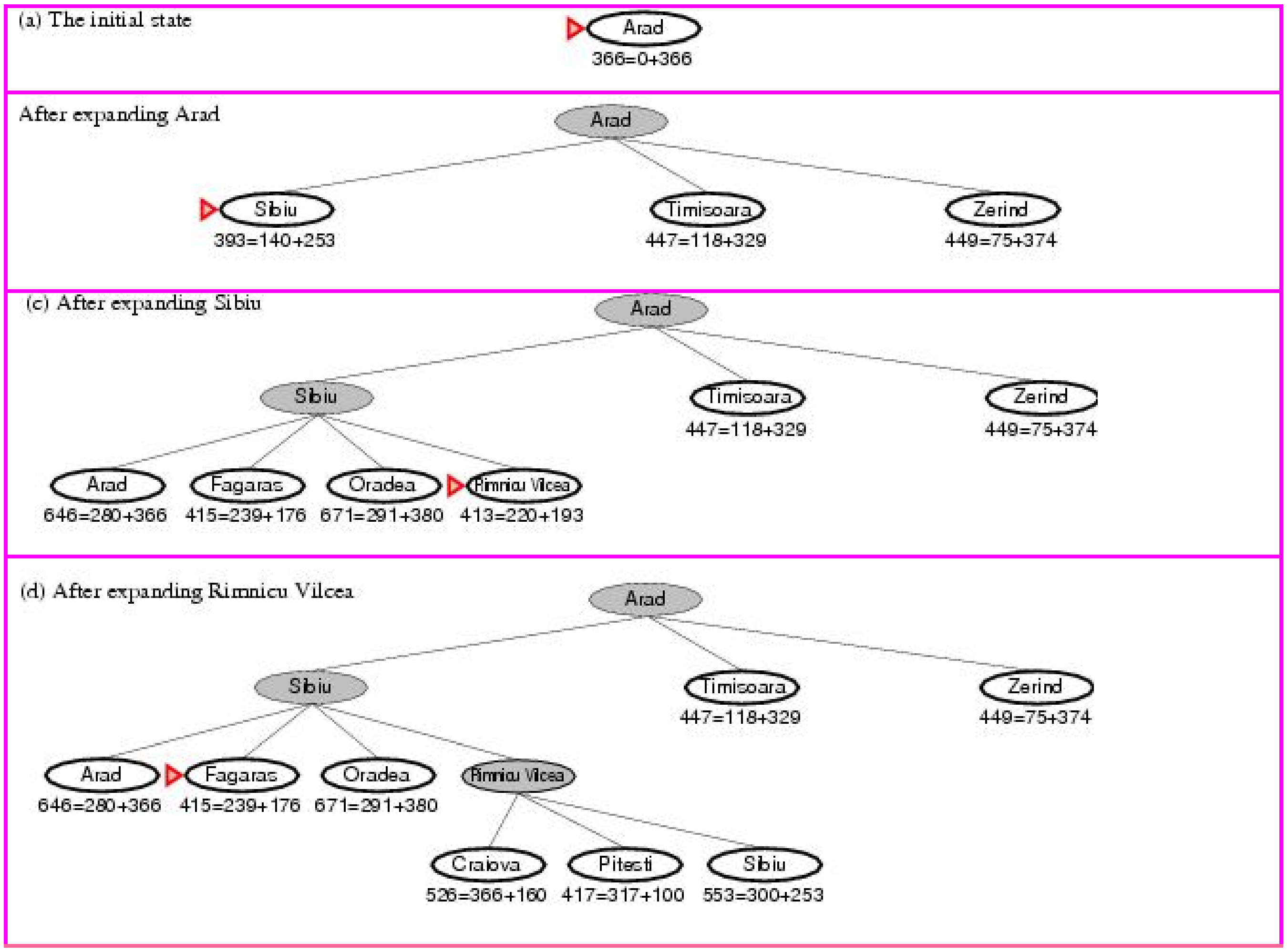
Since ***g (n)*** gives the path cost from the start node to node ***n***, and ***h (n)*** is the estirnated cost of the cheapest path from n to the goal, we have

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

Thus, 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 optimality of A* is straightforward to analyze if it is used with TREE-SEARCH. In this case, A* is optimal if ***h(n)*** is an admissible heuristic-that is, provided that ***h(n)*** ***never overestimates*** the cost to reach the goal. Admissible heuristics are by nature optimistic, because they think the cost of solving the problem is less than it actually is. Since ***g(n)*** is the exact cost to reach ***n***, we have as immediate consequence that *f (n)* never overestimates the true cost of a solution through n.

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



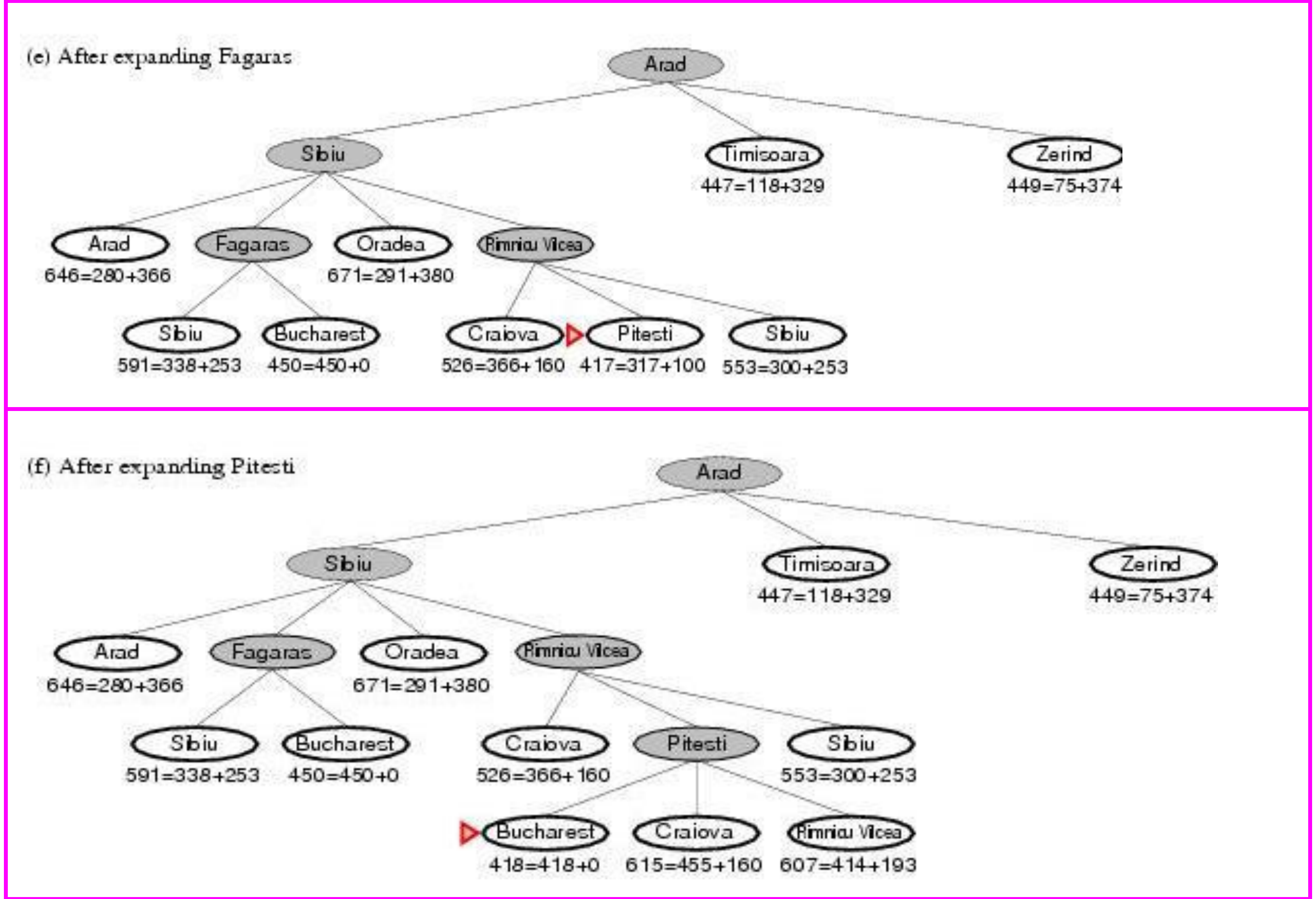


Figure 8.3: Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are the straight-line distances to Bucharest taken from **Figure 8.1**

All the admissible heuristics we discuss in this chapter are also consistent. Consider, for example, h_{SLD}. We know that the general triangle inequality is satisfied when each side is measured by the straight-line distance, and that the straight-line distance between ***n*** and ***n'*** is no greater than

c(n, a, n').

Hence, h_{SLD} is a consistent heuristic. Another important consequence of consistency is the following: If ***h(n)*** *is consistent, then the values* off *(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 ***a***, and we have

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n) = g(n) + h(n) = f(n).$$

8.3 Memory-bounded heuristic search

The simples1 way to reduce memory requirements for A" is to adapt the idea of iterative deepening to the heuristic search context, resulting in the iterative-deepening A" (IDA*) algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the ***f*** -cost (g + h) rather than the depth; at each iteration, the cutoff value is the smallest *f* -cost of any node that exceeded the cutoff on the previous iteration. IDA* is practical for many problems with unit step costs and avoids the substantial overhead associated with keeping a sorted queue of nodes.

8.3.1 Recursive best-first search (RBFS)

RBFS is **a** simple recursive algorithm that attempts to mimic the operation of standard best-first search, but using only linear space. The algorithm is shown in Figure 8.4. Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it keeps track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem )returns a solution or failure
RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞ )

function RBFS(problem, node, f-limit) returns a solution, or failure and a new f -cost limit
      if GOAL-TEST[problem](STATE[node]) then return node
      successors ← E X P A N D ( node , problem )
     if successors is empty then return failure, ∞
    for each s in successors do
        f [s] ←max(g(s) + h(s), f [node])
    repeat
       best ← the lowest f -value node in successors
       iff [best] ← f-limit then return failure, f [best]
      alternative ← the second-lowest f -value among successors
      result, f [best] ←RBFS(problem, best, min( f-limit, alternative))
      if result # failure then return result
```

Figure 8.4: The algorithm for recursive best-first search.

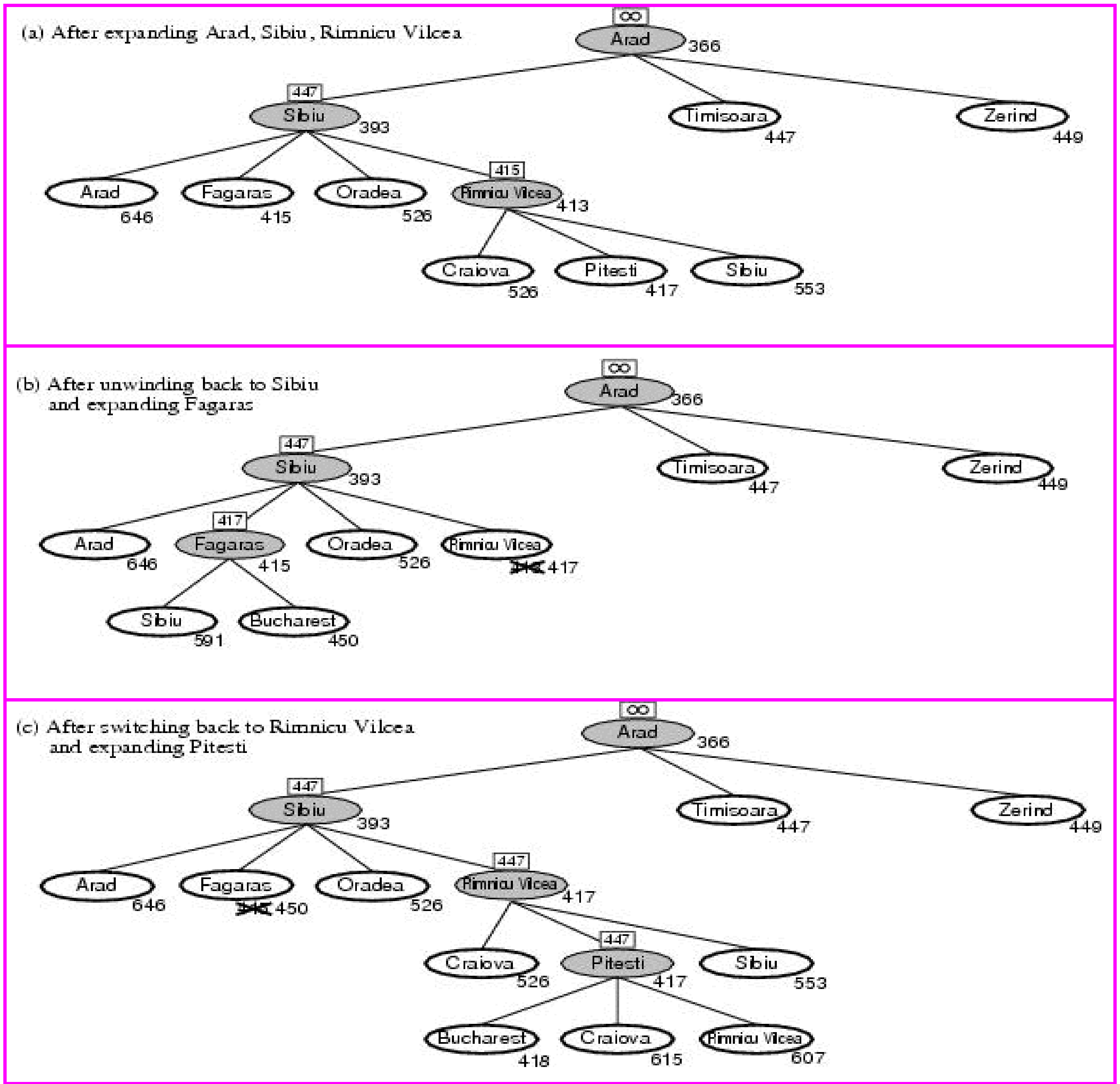


Figure 8.5: Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rirnnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the

expansion continues to Bucharest.

Evaluation Criteria of RBFS:

- **Complete: YES**
- **Time Complexity: Difficult to characterize [Depends on accuracy if h(n) and how often best path changes]**
- **Space Complexity: O (bd) [keeps all nodes in memory**]
- **Optimal: YES[if *h(n)* is admissible**]

8.3.2 Meta Level State Space and Object level State Space

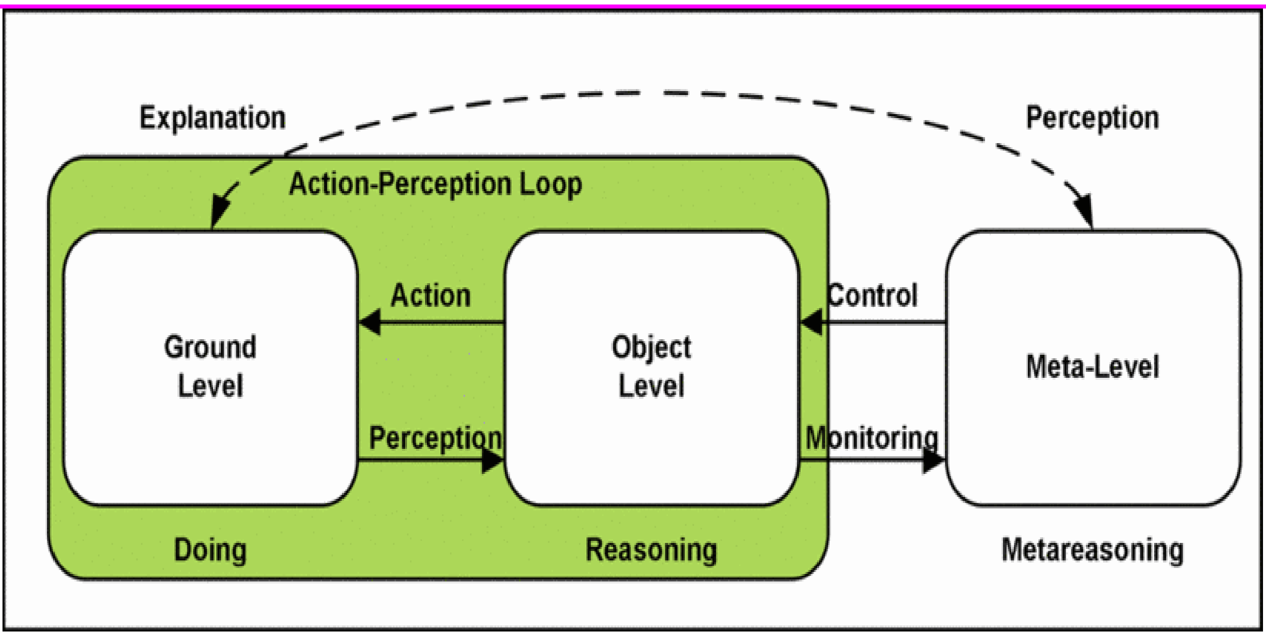


Figure 8.6 : Metareasoning concept

Traditionally, reasoning is understood as a decision making process within an Action-Perception loop. This means that an agent decides based on perceived information what action to perform. This corresponds to the context adaptation activities of a smart space. The smart space perceives its state, users, their activities and triggers corresponding service compositions, configuration and deployment to available resources to meet users' needs.

We suggest placing the reasoning about adaptation activities the smart space performs to a separate level. This activity is called Metareasoning. Metareasoning is analysis of how well the actions progress the tasks the services are performing and how well these tasks support users. In classical model, meta-level operates only with object level. However, to support the multi-user and dynamic nature of a smart space we modify this model slightly. Some ground level actions have to be perceived by meta-level in order to control the object level. The dotted line of Figure 8.6 emphasizes the feedback loop of the smart space. This means that meta-level would be able to collect feedback to estimate and understand details of the events at the ground level. Alternative strategies can be applied or even solutions can be created to achieve the goals in dynamic settings. Moreover, meta-level provides means to inform the users about system execution, for example, decision justification and failure explanation. Metareasoning consists of both the meta-level control of computational activities and the introspective monitoring of reasoning.

- The object level state space is defined for specific problem
- Then Meta level state space gathers the internal states of a computational program exact related to the Object level state space. Example: A * is internal states which consists of current search tree.
- An agent called Meta level state space performs actions that change the internal states. Example: The leaf node is expanded and successors are added to it, the above task is performed in each computational step.
- Metalevel Learning: for harder problem, sometimes there may be misstep and thus meta level learning can learn from history. Thus, it avoids expanding unpromising subtrees. It is special technique used in learning the basic objective of effective learning is to lower the total cost of problem solving trading off computational expense the path cost.

9. Heuristic Functions

The 8-puzzle was one of the earliest heuristic search problems. As mentioned in earlier, the object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration. [**Figure 9.1**]

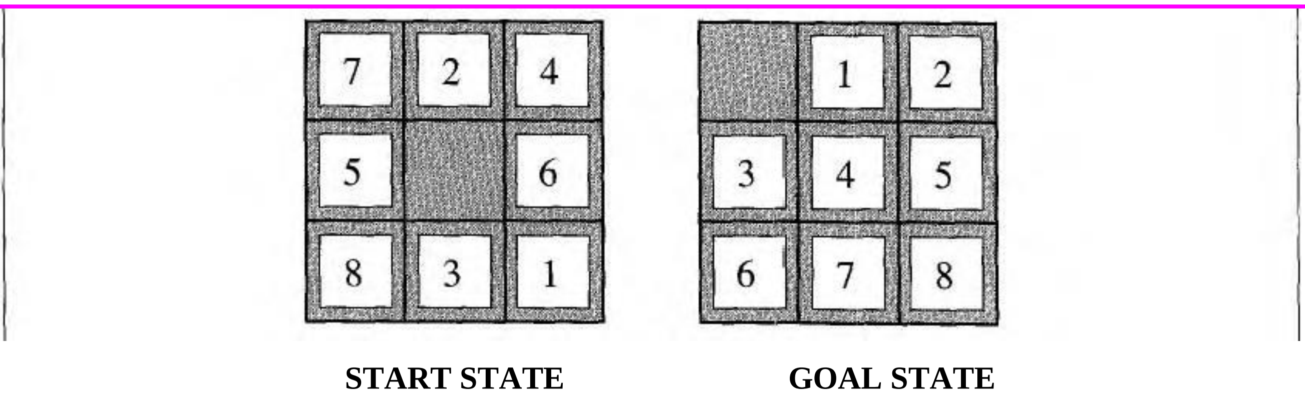


Figure 9.1: A typical instance of the 8-puzzle. The solution is 26 steps long.

The average solution cost for a randomly generated %-puzzle instance is about 22 steps. The branching factor is about 3. By keeping track of repeated states, we could cut this down by a factor of about 170,000, because there are only 9! /2 = 181,440 distinct states that are reachable. If we want to find the shortest solutions by using **A***, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 15-puzzle; here are two commonly-used candidates:

- ***h1*** = the number of misplaced tiles. For **Figure 9.1**, 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

$$h2 = 3+l+2+2+2+3+3+2 = 18.$$

As we would hope, neither of these overestimates the true solution cost, which is 26.

9.1 The effect of heuristic accuracy on performance

One way to characterize the quality of a heuristic is the effective branching factor ***b*.*** If the total number of nodes generated by ***A**** for a particular problem is N, and the solution depth is d, then ***b**** is the branching factor that a uniform tree of depth d would have to have in order to contain N + *1* nodes. Thus,

$$\boxed{N + 1 = 1 + \mathbf{b^*} + \mathbf{(b^*)2} + \ldots + (b^*)d.}$$

For example, if ***A**** finds a solution at depth *5* using 52 nodes, then the effective branching factor is 1.92. The effective branching factor can vary across problem instances, but usually it is fairly constant for sufficiently hard problems. Therefore, experimental measurements of ***b**** on a small set of problems can provide a good guide to the heuristic's overall usefulness. ***A*** well-designed heuristic would have a value of ***b**** close to ***1,*** allowing fairly large problems to be solved.

9.2 Inventing admissible heuristic functions

A problem with fewer restrictions on the actions is called a ***relaxed problem*****.** *The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.* The heuristic is admissible because the optimal solution in the original problem is, by definition, also a solution in the relaxed problem and therefore must be at least as expensive as the optimal solution in the relaxed problem. 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 formal1 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.

From **(a)**, we can derive *h1* (Manhattan distance). The reasoning is that *h2* would be the proper score if we moved each tile in turn to its destination. The heuristic derived from **(b)** is discussed later. From

(c), we can derive *hl* (misplaced tiles), because it would be the proper score if tiles could move to their intended destination in one step.

10. LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

In many problems, however, the path to the goal is irrelevant. For example, in the 8- queen's problem, what matters is the final configuration of queens, not the order in which they are added. This class of problems includes many important applications such as integrated-circuit design, factory-floor layout, job-shop scheduling, automatic programming, telecommunications network optimization, vehicle routing, and portfolio management.

Local search algorithms operate using a single **current state** (rather than multiple paths) and generally move only to neighbors of that state. Typically, the paths followed by the search are not retained.

Although local search algorithms are **not systematic**, they have two key advantages:

- they use very little memory-usually a constant amount; and
- they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

In addition to finding goals, local search algorithms are useful for solving pure **optimization problems**, in which the aim is to find the **best state** according to an **objective function**. To understand local search, it is better explained using **state space landscape** as shown in **Figure 10.1**. A landscape has both "**location**" (defined by the state) and "**elevation**"(defined by the value of the heuristic cost function or objective function). If elevation corresponds to **cost**, then the aim is to find the **lowest valley** – a **global minimum**; if elevation corresponds to an **objective function**, then the aim is to find the **highest peak** – a **global maximum.** Local search algorithms explore this landscape. A complete local search algorithm always finds a **goal** if one exists; an **optimal** algorithm always finds a **global minimum/maximum**.

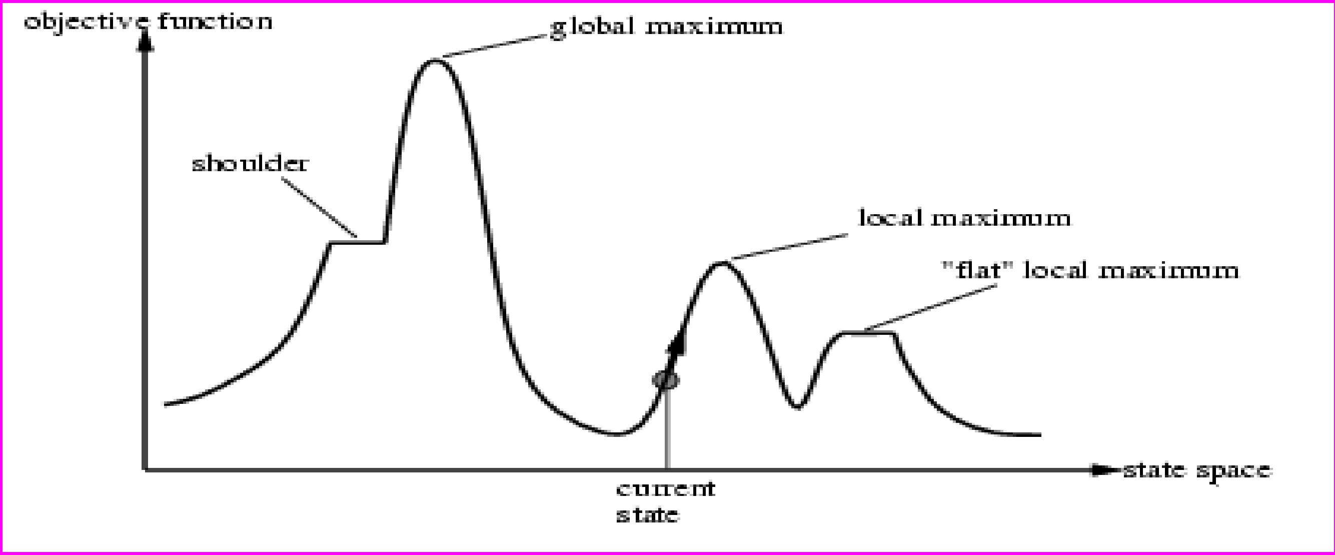


Figure 10.1: A one-dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to **try** to improve it, as shown by the arrow. The various topographic features are defined in the text.

- **Local maximum:** It is a state which is better than its neighboring state however there exists a state which is better than it (global maximum). This state is better because here value of objective function is higher than its neighbors.
- **Global maximum:** It is the best possible state in the state space diagram. This because at this state, objective function has highest value.
- **Plateau/flat local maximum:** It is a flat region of state space where neighboring states have the same value.
- **Ridge:** It is region which is higher than its neighbors but itself has a slope. It is a special kind of local maximum.
- **Current state:** The region of state space diagram where we are currently present during the search.
- **Shoulder:** It is a plateau that has an uphill edge.

10.1 Hill-climbing search

The **hill-climbing** search algorithm as shown in Figure 10.2 is simply a loop that continually moves in the direction of increasing value – that is, **uphill**. It terminates when it reaches a "**peak**" where no neighbor has a higher value. Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Greedy algorithms often perform quite well.

It is a variant of generate and test algorithm. The generate and test algorithm is as follows:

1. Generate possible solutions.

2. Test to see if this is the expected solution.

3. If the solution has been found quit else go to step 1.

Figure 10.2 : Generate and Test Algorithm format

Types of Hill Climbing

- Simple Hill climbing
- Steepest-Ascent Hill climbing
- Stochastic hill climbing

Simple Hill Climbing

It examines the neighboring nodes one by one and selects the first neighboring node which optimizes the current cost as next node. The algorithm is as follows:

Step 1: Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.

Step 2: Loop until the solution state is found or there are no new operators present which can be applied to current state.

a) Select a state that has not been yet applied to the current state and apply it to produce a new state.

b) Perform these to evaluate new state

i. If the current state is a goal state, then stop and return success.

ii. If it is better than the current state, then make it current state and proceed further.

iii. If it is not better than the current state, then continue in the loop until a solution is found.

Step 3: Exit.

Figure 10.3: Algorithm for Simple Hill climbing

Steepest-Ascent Hill climbing

It first examines all the neighboring nodes and then selects the node closest to the solution state as next node. The algorithm is as follows:

```
Step 1: Evaluate the initial state. If it is goal state then exit else make the current state as initial state
Step 2 : Repeat these steps until a solution is found or current state does not change
i. Let 'target' be a state such that any successor of the current state will be better than it;
ii. for each operator that applies to the current state
    a. apply the new operator and create a new state
    b. evaluate the new state
    c. if this state is goal state then quit else compare with 'target'
    d. if this state is better than 'target', set this state as 'target'
    e. if target is better than current state set current state to Target
Step 3 : Exit
```

Figure 10.4 : Algorithm for Steepest-Ascent Hill climbing

```
function HILL-CLIMBING( problem) return a state that is a local maximum
input: problem, a problem
local variables: current, a node.
neighbor, a node.
current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
neighbor ← a highest valued successor of current
if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
current ← neighbor
```

Figure 10.5: The hill-climbing search algorithm (**steepest ascent version**), which is the most basic local search technique. At each step the current node is replaced by the best neighbor; the neighbor with the highest VALUE. If the heuristic cost estimate h is used, we could find the neighbor with the lowest h.

Stochastic hill climbing

It does not examine all the neighboring nodes before deciding which node to select .It just selects a neighboring node at random, and decides (based on the amount of improvement in that neighbor) whether to move to that neighbor or to examine another.

Problems with hill-climbing

Hill-climbing often gets stuck for the following reasons:

- **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak, but will then be stuck with nowhere else to go
- **Ridges:** A ridge is shown in **Figure 10.6** .Ridges results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaus:** A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which it is possible to make progress.

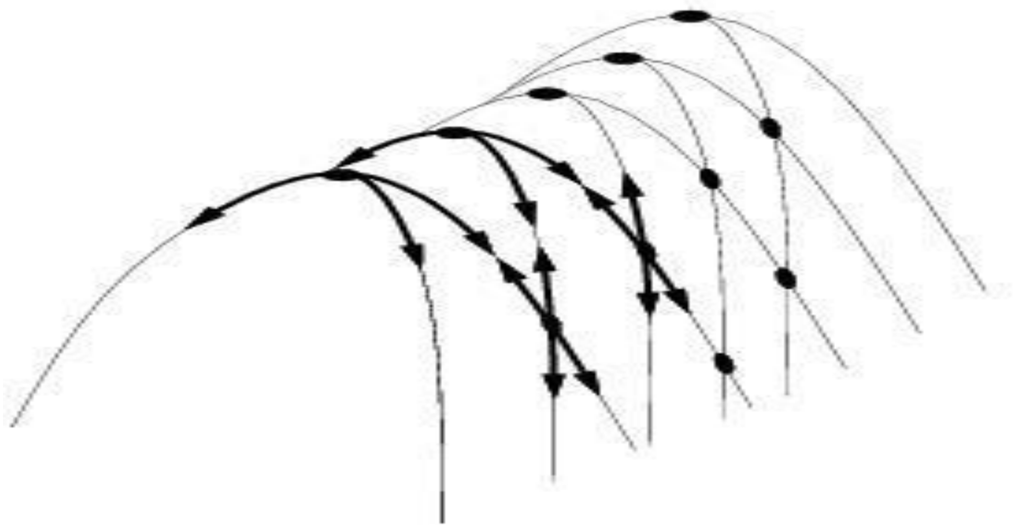


Figure 10.6: Illustration of why ridges cause difficulties for hill-climbing. The grid of states(dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available options point downhill.

Hill-climbing variations

➢ **Stochastic hill-climbing**

  ✓ Random selection among the uphill moves.

  ✓ The selection probability can vary with the steepness of the uphill move.

➢ **First-choice hill-climbing**

  ✓ Stochastic hill climbing by generating successors randomly until a better one is found.

➢ **Random-restart hill-climbing**

  ✓ Tries to avoid getting stuck in local maxima.

10.2 Simulated annealing search

A hill-climbing algorithm that never makes "downhill" moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient. Simulated annealing is an algorithm that combines hill-climbing with a random walk in some way that yields both efficiency and completeness.

Figure 10.7 shows simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move – the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad moves are more likely to be allowed at the start when temperature is high, and they become more unlikely as T decreases. One can prove that if the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems. It has been applied widely to factory scheduling and other large-scale optimization tasks.

```
function S I M U L A T E D - A N NEALING( problem, schedule) returns a solution state
inputs: problem, a problem
schedule, a mapping from time to "temperature"
local variables: current, a node
```

```
next, a node
T, a "temperature" controlling the probability of downward steps
current ← MAKE-NODE(INITIAL-STATE[problem])
for t ← l to ∞ do
T ← schedule[t]
if T = 0 then return current
next ← a randomly selected successor of current
ΔE ← VALUE[next] – VALUE[current]
if ΔE > 0 then current ← next
else current ← next only with probability e^{ΔE/T}
```

Figure 10.7: The simulated annealing search algorithm, a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in 1 the annealing schedule and then less often as time goes on. The *schedule* input determines the value of T as a function of time.

10.3 Local beam search

The **local beam search** algorithm keeps track of k states rather than just one. It begins with **k** randomly generated states. At each step, all the successors of all **k** states are generated. If anyone is a goal, the algorithm halts. Otherwise, it selects the **k** best successors from the complete list and repeats. In a local beam search, useful information is passed among the **k** parallel search threads.

In its simplest form, local beam search can suffer from a lack of diversity among the *k* states-they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called **stochastic beam search,** analogous to stochastic hill climbing, helps to alleviate this problem. Instead of choosing the best k from the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

10.4 Genetic algorithms

A genetic algorithm (or **GA)** is a variant of stochastic beam search in which successor states are generated by combining *two* parent states, rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search.

Like beam search, GAS begin with a set of k randomly generated states, called the **population.** Each state, or **individual,** is represented as a string over a finite alphabet-most commonly, a string of 0s and 1s. For example, an 8-queens state must specify the positions of 8 queens, each in a column of 8 squares, and so requires 8 *x* $\log_2 8$ = 24 bits.

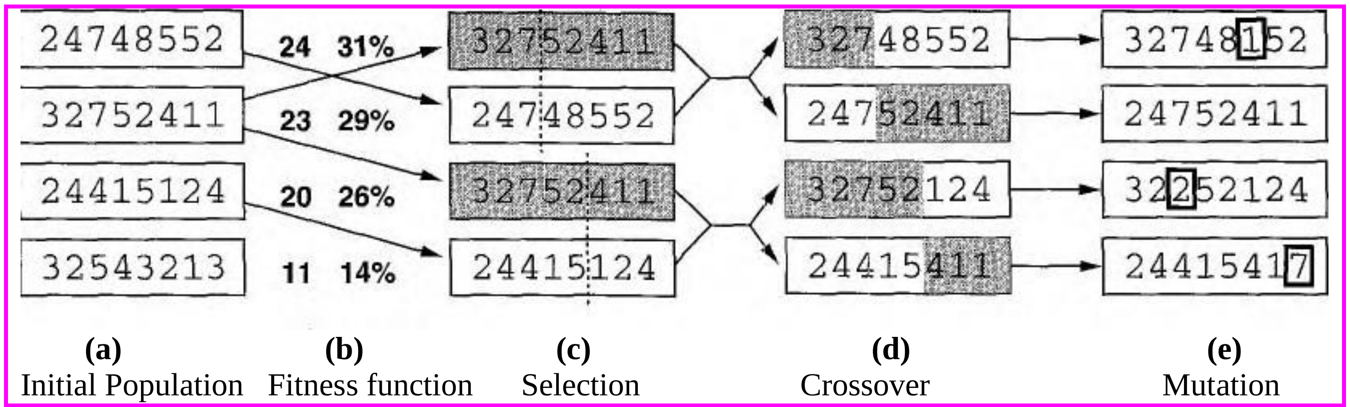


Figure 10.8: The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

The production of the next generation of states is shown in **Figure 10.8 (b)-(e)**.

In (a), Shows an Initial population of four 8-digit strings representing 8-queens states.

In (b), each state is rated by the evaluation function or (in **C; A** terminology) the **fitness function.** A fitness function should return higher values for better states, so, for the 8-queens problem we use the number of *nonattacking* pairs of queens, which has a value of 28 for a solution. The values of the four states are 24, 23, 20, and 11. In this particular variant of the genetic algorithm, the probability of being chosen for reproducing is directly proportional to the fitness score, and the percentages are shown next to the raw scores.

In (c), a random choice of two pairs is selected for reproduction, in accordance with the probabilities in (b). Notice that one individual is selected twice and one not at all. For each pair to be mated, a

crossover point is randomly chosen from the positions in the string. In **Figure 10.8** the crossover points are after the third digit in the first pair and after the fifth digit in the second pair.

In (d), the offspring themselves are created by crossing over the parent strings at the crossover point. For example, the first child of the first pair gets the first three digits from the first parent and the remaining digits from the second parent, whereas the second child gets the first three digits from the second parent and the rest from the first parent. The 8 queens states involved in this reproduction step are shown in **Figure 10.9.** The example illustrates the fact that, when two parent states are quite different, the crossover operation can produce a state that is a long way from either parent state. It is often the case that the population is quite diverse early on in the process, so crossover (like simulated annealing) frequently takes large steps in the state space early in the search process and smaller steps later on when most individuals are quite similar.

In (e), each location is subject to random **mutation** with a small independent probability. One digit was mutated in the first, third, and fourth offspring. In the 8-queens problem, this corresponds to choosing a queen at random and moving it to a random square in its column. **Figure 10.10** describes an algorithm that implements all these steps.

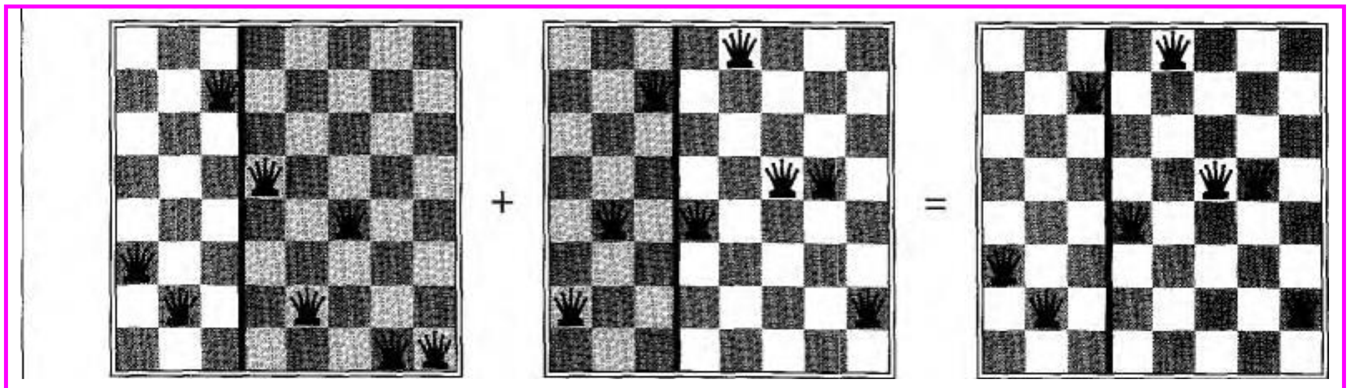


Figure 10.9: The 8-queens states corresponding to the first two parents in **Figure 10.8(c)** and the first offspring in **Figure 10.8(d).** The shaded columns are lost in the crossover step and the unshaded columns are retained.

function GENETIC_ALGORITHM(*population*, FITNESS-FN) **return an individual**
input: *population*, a set of individuals
FITNESS-FN, a function which determines the quality of the individual

```
repeat
  new_population ← empty set
   loop
    for i from 1 to SIZE(population) do
       x ←RANDOM_SELECTION(population, FITNESS_FN)
      y ←RANDOM_SELECTION(population, FITNESS_FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child )
     add child to new_population
      population ←new_population
 until some individual is fit enough or enough time has elapsed
 return the best individual
```

```
function REPRODUCE(x, y) returns an individual
 inputs: x, y, parent individuals
   n ←LENGTH(x)
   c ←random number from 1 to n
 return APPEND( SUBSTRING(x, 1, c ) ,SUBSTRING(y, c + 1, n ) )
```

Figure 10.10: A genetic algorithm. The algorithm is; the same as the one diagrammed in **Figure 10.8**, with one variation: in this more popular version, each mating of two parents produces only one offspring, not two.

10.5 LOCAL SEARCH IN CONTINUOUS SPACES

- We have considered algorithms that work only in discrete environments, but real-world environment are continuous.
- Local search amounts to maximizing a continuous objective function in a multi-dimensional vector space.
- This is hard to do in general.
- Can immediately retreat
  - ✓ Discretize the space near each state

- ✓ Apply a discrete local search strategy (e.g., stochastic hill climbing, simulated annealing)

- ➢ Often resists a closed-form solution
  - ✓ Fake up an empirical gradient
  - ✓ Amounts to greedy hill climbing in discretized state space
- ➢ Can employ Newton-Raphson Method to find maxima.
- ➢ Continuous problems have similar problems: plateaus, ridges, local maxima, etc.