الجامعة التكنولوجية

قسم علوم الحاسوب

فرع الوسائط المتعددة

المرحلة الثالثة

الكورس الأول

مادة طرق البحث الذكية

2024 - 2023

أ.م.د. حسنين سمير عبدالله

الجامعة التكنولوجية / قسم علوم الحاسوب

فرع الوسائط المتعددة – المرحلة الثالثة – الكورس الأول

مادة طرق بحث ذكية – أ.م.د. حسنين سمير

"Artificial Intelligence Concept and Fundamentals"

1. Principles fundamentals of A.I.

Artificial Intelligence, Artificial Evolution and Artificial Life are three distinct approaches to programming computers in order to make them behave as if they were human, more primitive animals, or other living species.

There are two fundamentally major approaches in the field of AI. One is often termed traditional symbolic AI, which has been historically dominant. It is characterized by a high level of abstraction. Knowledge engineering systems and logic programming fall in this category. Symbolic

AI covers areas such as knowledge base systems, logical reasoning, symbolic machine learning, search techniques, and natural language processing. The second approach is based on low level, microscopic biological models, similar to the emphasis of physiology orgenetics. Neural networks, genetic algorithms and DNA computing are the prime

examples of this latter approach. These biological models do not necessarily resemble their original biological counterparts. However, they are evolving areas from which many people expect significant practical applications in the future.

2. What Means by A.I.

Artificial intelligence (AI) maybe defined as the branch of computer science that is concerned with the automation of intelligent behavior. This definition is particularly appropriate to this book in that it emphasizes our conviction that AI is apart of computer science and, as such, must be based on sound theoretical and applied principles of that field. These principles include the data structures used in knowledge representation, the algorithms needed to apply that knowledge, and the languages and programming techniques used in their implementation.

For any computing system it is very important to achieve an acceptable level of software quality. The basic goal of software quality is the prevention of software faults or, at least, the lowering of software fault rates.

One way to combine higher quality with higher efficiency is to use supporting quality devices, these devices, based as they are on the accumulated knowledge and experience of the organization's development and maintenance professional, contribute to meeting software goals by:

- Saving the time required to run the application of the software under hand.

- Reviewing register the memory area that handles the knowledge base and the operations.
- Defining the type of controlled needed and search technique.

3. General Problem Solving Approaches

There exist quite a large number of problem solving techniques in AI that rely on search. The simplest among them is the generate-and-test method. The algorithm for the generate-and-test method can be fom1ally stated in the figure (1) follow. It is clear from the above algorithm that the algorithm continues the possibility of exploring a new state in each iteration of the repeat-until loop and exits only when the current state is equal to the goal. Most important part in the algorithm is to generate a new state. This is not an easy task.

If generation of new states is not feasible, the algorithm should be terminated. In our simple algorithm, we, however, did not include this intentionally to keep it simplified. But how does one generate the states of a problem? To formalize this, we define a four tuple, called state space, denoted by {nodes, arc, goal, current },

```
Procedure Generate & Test
Begin
    Repeat
     Generate a new state and call it current-state;
    Until current-state = Goal;
End.
```

Figure (1) Generate and Test Algorithm

Where:

nodes represent the set of existing states in the search space;

an arc denotes an operator applied to an existing state to cause transition to another state; goal denotes the desired state to be identified in the nodes; and current represents the state, now generated for matching with the goal. The state space for most of the search problems takes the form of a tree or a graph. Graph may contain more than one path between two distinct nodes, while for a tree it has maximum value of one.

To build a system to solve a particular problem, we need to do four things:

1. Define the problem precisely. This definition must include precise specifications of what the initial situation(s) will be as well as what final situations constitute acceptable solutions to the problem.

2. Analyze the problem. A few very important features can have an immense impact on the appropriateness of various possible techniques for solving the problem.
3. Isolate and represent the task knowledge that is necessary to solve the problem.
4. Choose the best problem-solving technique(s) and apply it (them) to the particular problem.

Measuring problem-solving performance is an essential matter in term of any problem solving approach. The output of a problem-solving algorithm is either failure or a solution. (Some algorithm 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?
- **Time complexity:** How long does it take to find a soluϴon?1
- **Space complexity:** How much memory is needed to perform the search ?

"Knowledge Representation"

1. What are Knowledge Representation Schemes ?

In Al, there are four basic categories of representational schemes: logical, procedural, network and structured representation schemes.

1. Logical representation uses expressions in formal logic to represent its knowledge base. Predicate Calculus is the most widely used representation scheme.

2. Procedural representation represents knowledge as a set of instructions for solving a problem. These are usually if-then rules we use in rule-based systems.

3. Network representation captures knowledge as a graph in which the nodes represent objects or concepts in the problem domain and the arcs -represent relations or associations between them.

4. Structured representation extends network representation schemes by allowing each node to have complex data structures named slots with attached values.

2.The Propositional Calculus

2.1 Symbols and Sentences

The prepositional calculus and, in the next subsection, the predicate calculus are first of all languages. Using their words, phrases, and sentences, we can represent and reason about properties and relationships in the world. The first step in describing a language is to introduce the pieces that make it up: set of symbols.

DEFINTION

PROPOSITIONAL CALCULUS SYMBOLS

The symbols of prepositional Calculus are, the prepositional symbols:
P. Q. R,S,T....
truth symbols

true, false
and connectives:
¬ ∃ ∀ ∧ ∨

Prepositional symbols denote propositions of statements about the world that may be either true or raise, such as "the car is red" or" water is wet." Propositions are denoted by uppercase letters near the end of the English alphabet. Sentences in the propositional calculus are formed from these atomic symbols according to the following rules:

DEFINITION

PROPOSIONAL CALCULUS SENTENCES

Every prepositional symbol and truth symbol is a sentence.

For example: true, P, Q, and R are sentences.

The negation of a sentence is a sentence

for example: ¬P and ¬false are sentences

The conjunction and of two sentences : is a sentence

For example: P ∧ ¬P is a sentence.

The disjunction or of two sentences is a sentence.

For example P ∨ ¬P. is a sentence,

The implication of one sentence for another is a sentence.

For example P →Q is sentence.

The equivalence of two sentences Is a sentence.

For example: P ∨ Q = R is a sentence.

3. Legal sentences are also called well-formed formulas 01 WFFs.

IN expressions of the form P ∧ Q. P and Q are called the conjuncts In P ∨ Q and Q are referred to as disjuncts in an implication P → Q, P is the premise or antecedent and Q, the conclusion or consequent .

In propositional calculus sentences. the symbols( ) and [ ] are used to group symbols into sub expressions and so control their order of evaluation and meaning. for example (P ∧ Q)= R is quit different from P ∨ (Q=R).

An expression is a sentence, or well-formed formula, of the prepositional calculus if and only if it can be formed of legal symbols through some sequence of these rules. For example :

$P \wedge Q \rightarrow R = \neg P \vee \neg Q \vee R$

is a well-formed sentence in the in the propositions calculus because:

P, Q, and R are-proposition and .thus sentences.

$P \wedge Q$, the conjunction of two sentences, is a sentence.

$(P \wedge Q) \rightarrow R$, the implication of a sentence for another, is a sentence.

$\neg P$ and $\neg Q$, the negations of sentences, are sentences.

$\neg P \vee \neg Q$, the disjunction of two sentences, is a sentence.

$\neg P \vee \neg Q \vee R$, the disjunction of two sentences, is a sentence.

$((P \wedge Q) \rightarrow R) = \neg P \vee \neg Q \vee R$, the equivalence of two sentences, is a sentence.

This is our original sentence, which has been Constructed through a series of applications of legal roles and is therefore well formed.

DEFINITION

PROPOSITIONAL CALCULUS SEMANTICS

An interpretation of a set of propositional is the assignment of a truth value , either T or F , to each propositional symbol .

The symbol true is always assigned T, and the symbol false is assigned F.

P	Q	P∧Q
T	T	T
T	F	F
F	T	F
F	F	F

Figure (1), Truth table for the operator AND

P	Q	¬P	¬pvQ	P⇒Q	(¬pvQ)=(P⇒Q)
T.	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

Figure (2), Truth table demonstrating the equivalence formula.

4. The Predicate Calculus (Predicate Logic)

In prepositional calculus, each atomic symbol (P, Q, etc.) denotes a proposition of some complexity.

There is no way to access the components of an individual assertion. Predicate calculus provides this ability.

For example, instead pf letting a single prepositional symbol, P, denote: 1he entire sentence "it rained on Tuesday," we can create a predicate weather that describes a relationship between a date and the

weather. weather (Tuesday, rain) through inference rules we can manipulate predicate calculus expression accessing their individual components and inferring new sentences.

Predicate calculus also allows expressions^ contain variables. Variables let us create general assertions about classes of entities. For example, we could state that for :

all values, of X, where X is a day of the week , the statement weather (X, rain) is true ; I,e., it rains it rains everyday. As with propositional calculus, we will first define the syntax of the language and then discuss its semantics.

Examples of English sentences represented in Predicate Logic:

1- If it doesn't rain tomorrow, Tom will go to the mountains.

¬ weather (rain, tomorrow)→go(tom, mountains).

2- Emma is a Doberman pinscher and a good dog.

Good dog (emma) ^ isa (emma, Doberman)

3- All basketball players are tall.

∀ X (basketball _ player(X)→ tall (X))

4- Some people like anchovies.

∃X (person(X) ^ likes(X, anchovies)),

5- If wishes were horses, beggars would ride.

equal(wishes, horses) → ride(beggars).

6- Nobody likes taxes

¬ ∃ X likes(X,taxes).

Example: Convert the following sentences into Predicate logic form:

"All people who are not poor and are smart are hippy. Those people who read are not stupid. John can read and is wealthy. Happy people have exciting lives. Can anyone be found with an exciting life?"

Solution:

$\forall X (\neg poor(X) \wedge smart(X) \rightarrow happy(X))$.

$\forall Y (read(Y) \rightarrow \neg stupid(Y))$. == $\forall Y (read(Y) \rightarrow smart(Y))$.

$read(john) \wedge wealthy(john)$. == $read(john) \wedge \neg poor(john)$.

$\forall Z (Happy(Z) \rightarrow exciting(Z))$.

$\exists W (exciting(W))$.

Homework: Convert the following sentences into Predicate logic form:

Anyone passing his history exams and winning the lottery is happy. But anyone who studies or is lucky can pass all his exams. John did not study but he is lucky. Anyone who is lucky wins the lottery. Is John happy?

5. Conceptual Graphs: a Network Language

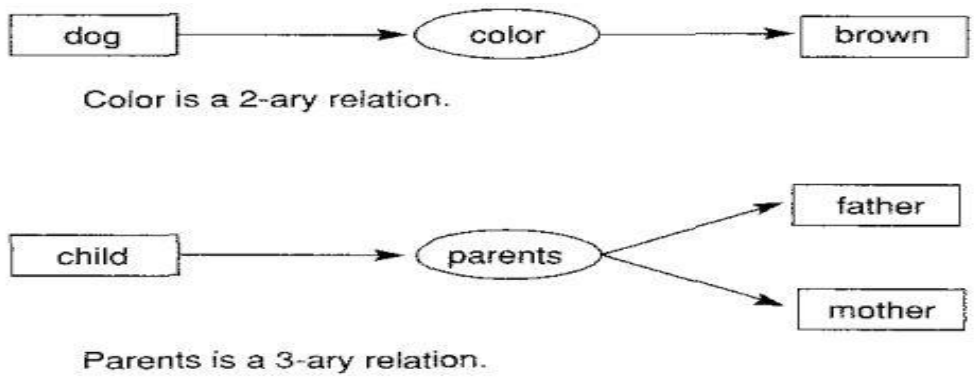


Figure (3), Conceptual relations of different arities.

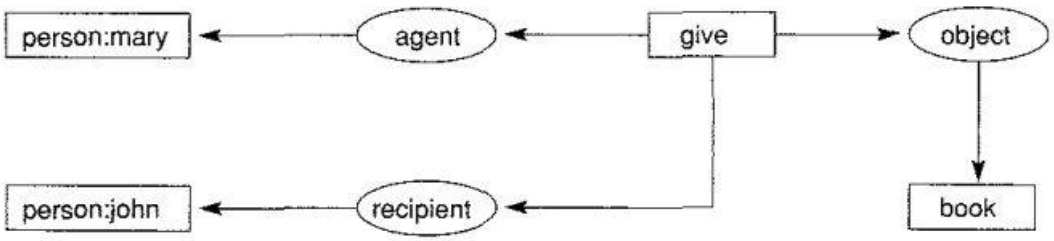


Figure (4), Graph of "Mary gave John the book."

Figure (5), Conceptual graph indicating that the dog named emma is brown.

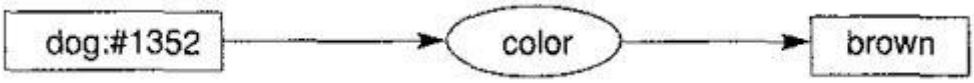


Figure (6), Conceptual graph indicating that a particular (but unnamed) dog is brown.

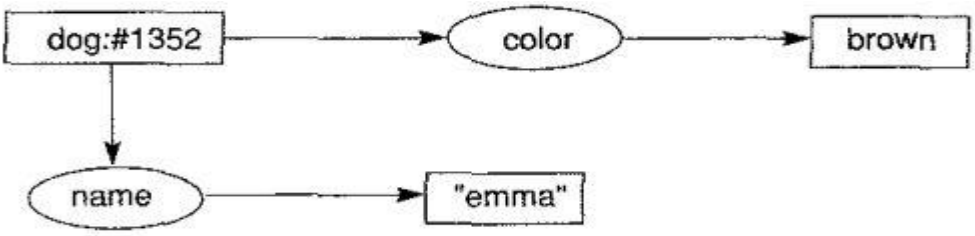


Figure (7), Conceptual graph indicating that a dog named emma is brown.

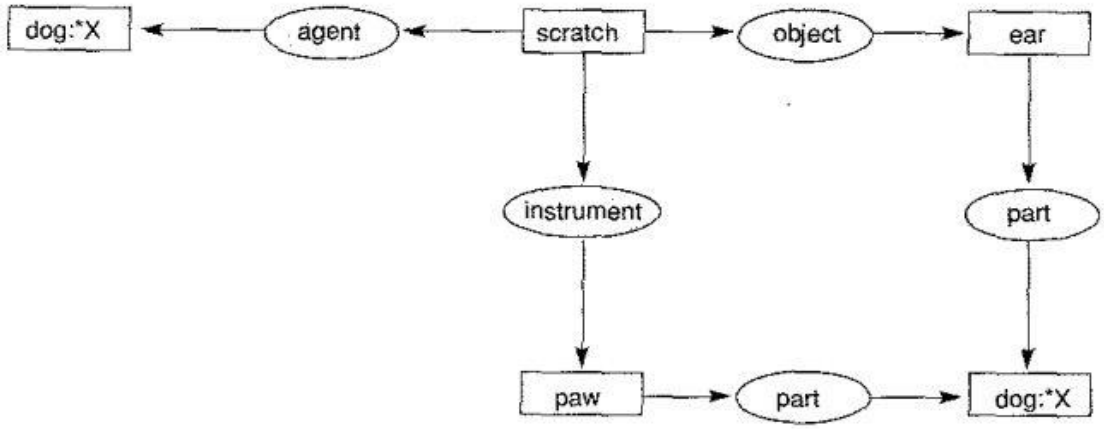


Figure (8), Conceptual graph of the sentence "The dog scratches its ear with its paw."

Showing the use of a propositional concept. Figure 9 and 10

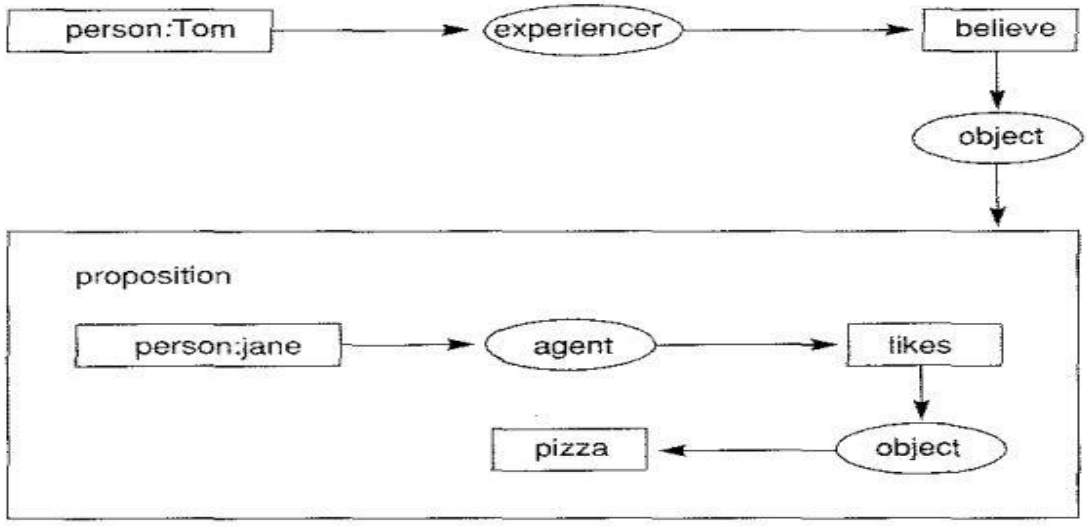


Figure (9), Conceptual graph of the statement "Tom believes that Jane likes pizza".

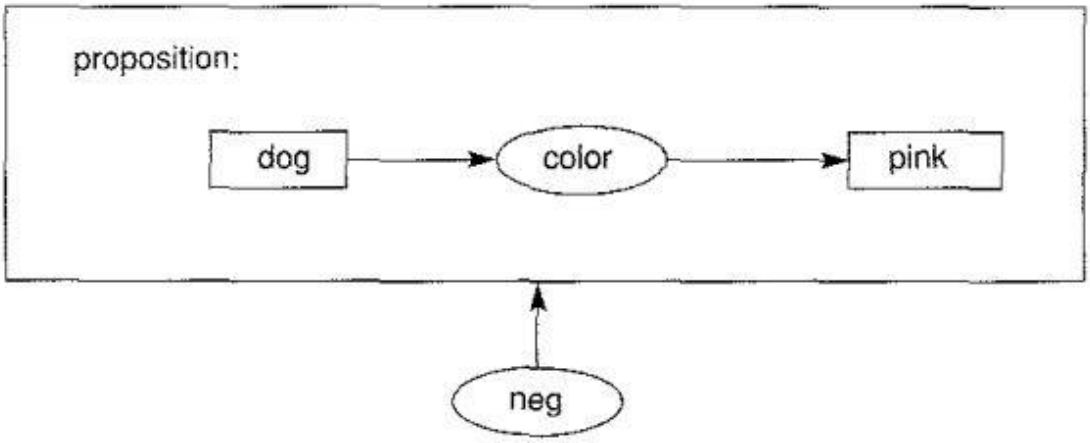


Figure (10), Conceptual graph of the proposition "There are no pink dogs".

Homework:

Represent the following sentences using Conceptual Graph method:

- John likes small cars.
- Mary gave Tom red book.
- John thinks that Mary gave the book to Tom.

John thinks that Mary gave the book to Tom in the classroom.

1- Intelligent Search Methods and Strategies

Search is inherent to the problem and methods of artificial intelligence (AI). This is because AI problems are intrinsically complex. Efforts to solve problems with computers which human can routinely innate cognitive abilities, pattern recognition, perception and experience, invariably must turn to considerations of search. All search methods essentially fall into one of two categories, exhaustive (blind) methods and heuristic or informed methods.

2 -State Space Search

The state space search is a collection of several states with appropriate connections (links) between them. Any problem can be represented as such space search to be solved by applying some rules with technical strategy according to suitable intelligent search algorithm.

What we have just said, in order to provide a formal description of a problem, we must do the following:

1- Define a state space that contains all the possible configurations of the relevant objects (and perhaps some impossible ones). It is, of course, possible to define this space without explicitly enumerating all of the states it contains.

2- Specify one or more states within that space that describe possible situations from which the problem-solving process may start. These states are called the initial states.

3- Specify one or more states that would be acceptable as solutions to the problem. These states are called goal states.

4- Specify a set of rules that describe the actions (operators) available. Doing this will require giving thought to the following issues:

• What unstated assumptions are present in the informal problem description?

• How general should the rules be?

• How much of the work required to solve the problem should be precomputed and represented in the rules?

The problem can then be solved by using rules, in combination with an appropriate control strategy, to move through the problem space until a path from an initial state to a goal state is found. Thus the process of search is fundamental to the problem-solving process. The fact that search provides the basis for the process of problem solving does not, however, mean that other, more direct approaches cannot also be exploited. Whenever possible, they can be included as steps in the search by encoding them rules. Of course, for complex problems, more sophisticated computations will be needed. Search is a general mechanism that can be used when no more direct methods is known. At the same time, it provide the framework into which more direct methods for solving subparts of a problem can be embedded.

To successfully design and implement search algorithms, a programmer must be able to analyze and predict their behavior. Questions that need to be answered include:

• Is the problem solver guaranteed to find a solution?

• Will the problem solver always terminate, or can it become caught in an infinite loop?

• When a solution is found, is it guaranteed to be optimal?

• What is the complexity of the search process in terms of time usage? Memory usage?

• How can the interpreter most effectively reduce search complexity?

• How can an interpreter be designed to most effectively utilize a representation language?

To get a suitable answer for these questions search can be structured into three parts. A first part presents a set of definitions and concepts that lay the foundations for the search procedure into which induction is mapped. The second part presents an alternative approaches that have been taken to induction as a search procedure and finally the third part present the version space as a general methodology to implement induction as a search procedure. If the search procedure contains the principles of the above three

requirement parts, then the search algorithm can give a guarantee to get an optimal solution for the current problem.

3 -General Problem Solving Approaches

There exist quite a large number of problem solving techniques in AI that rely on search. The simplest among them is the generate-and-test method. The algorithm for the generate-and-test method can be fom1ally stated in figure (1). It is clear from the above algorithm that the algorithm continues the possibility of exploring a new state in each iteration of the repeat-until loop and exits only when the current state is equal to the goal. Most important part in the algorithm is to generate a new state. This is not an easy task.

```
Procedure Generate & Test
Begin
    Repeat
     Generate a new state and call it current-state;
    Until current-state = Goal;
End.
```

Figure (1), Generate and Test Algorithm

If generation of new states is not feasible, the algorithm should be terminated. In simple algorithm, we, however, did not include this intentionally to keep it simplified. But how does one generate the states of a problem? To formalize this, we define a four tuple, called state space, denoted by
{nodes, arc, goal, current },

where

Nodes represent the set of existing states in the search space;

an arc denotes an operator applied to an existing state to cause transition to another state;

Goal denotes the desired state to be identified in the nodes;

and current represents the state, now generated for matching with the goal.

The state space for most of the search problems takes the form of a tree or a graph. Graph may contain more than one path between two distinct nodes, while for a tree it has maximum value of one.

To build a system to solve a particular problem, we need to do four things:

1. Define the problem precisely. This definition must include precise specifications of what the initial situation(s) will be as well as what final situations constitute acceptable solutions to the problem.

2. Analyze the problem. A few very important features can have an immense impact on the appropriateness of various possible techniques for solving the problem.

3. Isolate and represent the task knowledge that is necessary to solve the problem.

4. Choose the best problem-solving technique(s) and apply it (them) to the particular problem.

Measuring problem-solving performance is an essential matter in term of any problem solving approach. The output of a problem-solving algorithm is either failure or a solution. (Some algorithm 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?

• **Time complexity:** How long does it take to find a solution?

• **Space complexity:** How much memory is needed to perform the search?

4 - Search Technique

Having formulated some problems, we now need to solve them. This is done by a search through the state space. The root of the search tree is a search node corresponding to the initial state. The first step is to test whether this is a goal state. Because this is not a goal state, we need to consider some other states. This is done by expanding the current state; that is, applying the successor function to the current state, thereby generating a new set of states. Now we must choose which of these possibilities to consider further. We continue choosing, testing and expanding either a solution is found or there are no more states to be expanded. The choice of which state to expand is determined by the search strategy. It is important to distinguish between the state space and the search tree. For the route finding problem, there are only N states in the state space, one for each city. But there are an infinite number of nodes.

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.

As usual, we differentiate between two main families of search strategies: systematic search and local search. Systematic search visits each state that could be a solution, or skips only states that are shown to be dominated by others, so it is always able to find an optimal solution.

Local search does not guarantee this behavior. When it terminates, after having exhausted resources (such as time available or a limit number of iterations), it reports the best solution found so far, but there is no guarantee

that it is an optimal solution. To prove optimality, systematic algorithms are required, at the extra cost of longer running times with respect to local search. Systematic search algorithms often scale worse with problem size than local search algorithms.

Blind Search

There many search strategies that come under the heading of blind search (also called uniformed search). The term means that they have 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.

Thus blind search strategies have not any previous information about the goal nor the simple paths lead to it. However blind search is not bad, since more problems or applications need it to be solved; in other words there are some problems give good solutions if they are solved by using depth or breadth first search.

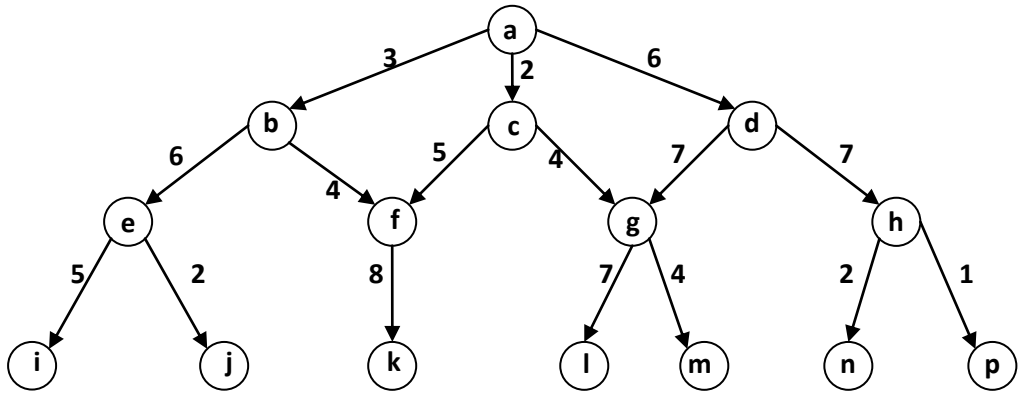
Breadth first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. Breadth first search can be implemented by calling TREE-SEARCH with any

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) result 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.

Function Breadth-First Search

```
Begin
  Open: = [Initial state];                          %initialize
  Closed: = [ ];
  While open <> [ ] do                              %state remain
   Begin
    Remove leftmost state from open, call it X;
    If X is a goal then return SUCCESS              %goal found
     Else
     Begin
      Generate children of X;
      Put X on closed;
      Discard children of X if already on open or closed;   %loop check
      Put remaining children on right end of open           %queue
     End;
   End;
  Return FAIL                                       %no states left
End.
```

Consider the following problem state space:



Find the path from **a** to **j** using breadth first search algorithm

Open	**Closed**
[a]	[]
[b, c, d]	[a]
[c, d, e, f]	[a, b]
[d, e, f, g]	[a, b, c]
[e, f, g, h]	[a, b, c, d]
[f, g, h, i, j]	[a, b, c, d, e]
[g, h, i, j, k]	[a, b, c, d, e, f]
[h, i, j, k, l, m]	[a, b, c, d, e, f, g]
[i, j, k, l, m, n, p]	[a, b, c, d, e, f, g, h]
[j, k, l, m, n, p]	[a, b, c, d, e, f, g, h, i]
Stop the goal (j) is found	

Depth first search always expands the deepest node in the current fringe 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

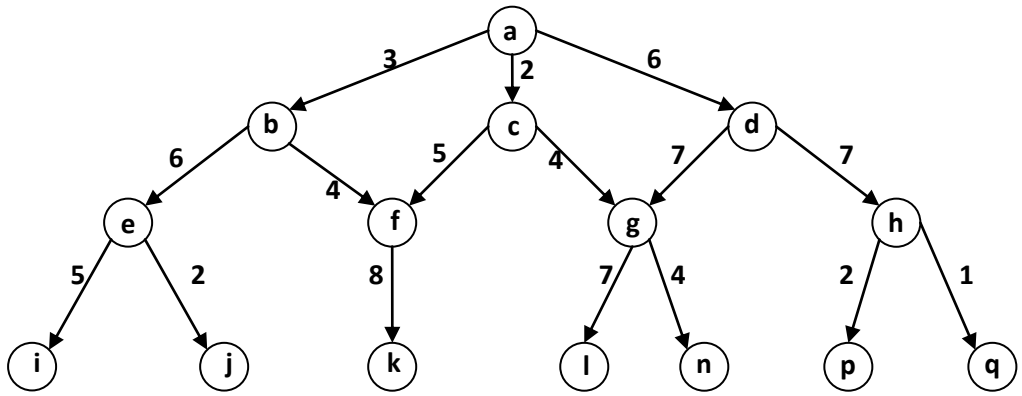
expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors. This strategy can be implemented by TREE-SEARCH with a last-in first-out (LIFO) queue, also known as a stack.

As an alternative to the TREE-SEARCH implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

Function Depth-First Search

```
Begin
  Open: = [Initial state];                          %initialize
  Closed: = [ ];
  While open <> [ ] do                              %state remain
   Begin
    Remove leftmost state from open, call it X;
    If X is a goal then return SUCCESS              %goal found
     Else
     Begin
      Generate children of X;
      Put X on closed;
      Discard children of X if already on open or closed;   %loop check
      Put remaining children on left end of open            %stack
     End;
   End;
  Return FAIL                                       %no states left
End.
```

Consider the following problem state space:



Find the path from **a** to **k** using depth first search algorithm

Open	**Closed**
[a]	[]
[b, c, d]	[a]
[e, f, c, d]	[a, b]
[i, j, f, c, d]	[a, b, e]
[j, f, c, d]	[a, b, e, i]
[f, c, d]	[a, b, e, i, j]
[k, c, d]	[a, b, e, i, j, f]

Stop the goal (k) is found

Heuristic Search Algorithms

In this section, we can saw that many of the problems that fall within the purview of artificial intelligence are too complex to be solved by direct techniques; rather they must be attacked by appropriate search methods armed

with whatever direct techniques are available to guide the search. These methods are all varieties of heuristic search.

They can be described independently any particular task or problem domain. But when applied to Particular problems, their efficacy is highly dependent on the way they exploit domain-specific knowledge since in and of themselves they are unable to overcome the combinatorial explosion to which search processes are so vulnerable. For this reason, these techniques are often called weak methods. Although a realization of the limited effectiveness of these weak methods to solve hard problems by themselves has been an important result that emerged from the last decades of AI research, these techniques continue to provide the framework into which domain-specific knowledge can be placed, either by hand or as a result of automatic learning.

Hill climbing is a variant of generate-and-test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space. In a pure generate-and-test procedure, the test function responds with only a yes or no. but if the test function is augmented with a heuristic function that provide an estimate of how close a given is to a goal state. This is particularly nice because often the computation of the heuristic function can be done at almost no cost at the same time that the test for a solution is being performed. Hill climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available. For example, suppose you are in an unfamiliar city without a map and you want to get downtown. You simply aim for the tall buildings. The heuristic function is just distance between the current location and the location of the tall buildings and the desirable states are those in which this distance is minimized.

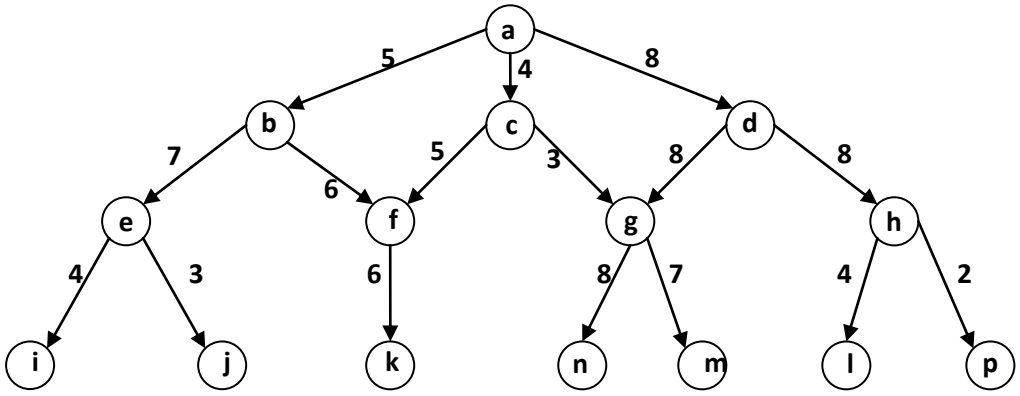
For each state f(n) = h(n) where h(n) is the heuristic function that computes the heuristic value for each state n.

Function Hill Climbing Search

```
Begin
 Open: = [Initial state];                          %initialize
 Closed: = [ ];
 CS= initial state;
 Path= [initial state];
 Stop= FALSE;
 While open <> [ ] do                              %states remain
  Begin
   If CS=goal then return path
   Generate all children of CS and put them into open;
    If open= [ ] then
    Stop= TRUE
     Else
      Begin
       X= CS;
       For each state Y in open do
        Begin
         Compute the heuristic value of y (h(Y));
         If Y is better than X then
         X=Y
        End;
       If X is better than CS then
       CS=X
```

```
            Else
            Stop= TRUE;
        End;
    End;
    Return (FAIL);                    %open is empty
End.
```

Consider the following problem state space then:



Find the path from **a** to **m** using Hill Climbing search algorithm.

Open	**Closed**
[a]	[]
[b5, c4, d8]	[a]
[c4, b5, d8]	[a]
[f5, g3, b5, d8]	[a, c4]
[g3, f5, b5, d8]	[a, c4]
[n8, m7, f5, b5, d8]	[a, c4, g3]
[m7, n8, f5, b5, d8]	[a, c4, g3]

Stop the goal (m) is found

Now let us discuss a new heuristic method called "Best First Search", which is a way of combining the advantages of both depth-first and breadth-first search into a single method.

The actual operation of the algorithm is very simple. It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state. At each step, it picks the most promising of the nodes that have so far been generated but not expanded. It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before. By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successors. Then the next step begins.

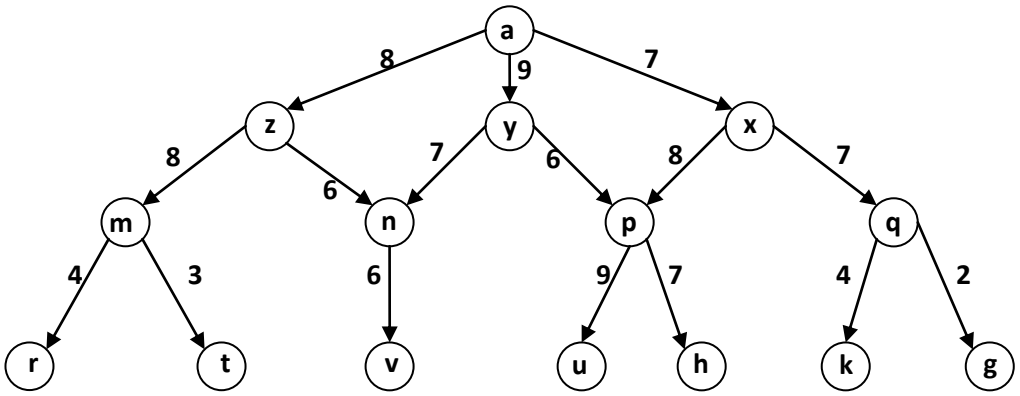
For each state f(n) = h(n) where h(n) is the heuristic function that computes the heuristic value for each state n.

Function Best-First Search

```
Begin
 Open: = [Initial state];                          %initialize
 Closed: = [ ];
 While open <> [ ] do                              %states remain
  Begin
   Remove leftmost state from open, call it X;
   If X = goal then return the path from initial to X
   Else
   Begin
    Generate children of X;
    For each child of X do
```

```
      Case
       The child is not on open or closed;
        Begin
          Assign the child a heuristic value;
          Add the child to open
        End;
       The child is already on open;
        If the child was reached by a shorter path
        Then give the state on open the shorter path
       The child is already on closed;
        If the child was reached by a shorter path then
          Begin
            Remove the state from closed;
            Add the child to open
          End;
      End;                                          %case
      Put X on closed;
      Re-order states on open by heuristic merit (best leftmost)
   End;
Return FAIL                                         %open is empty
End.
```

Consider the following problem state space then:



Find the path from **a** to **k** using Best first Search algorithm.

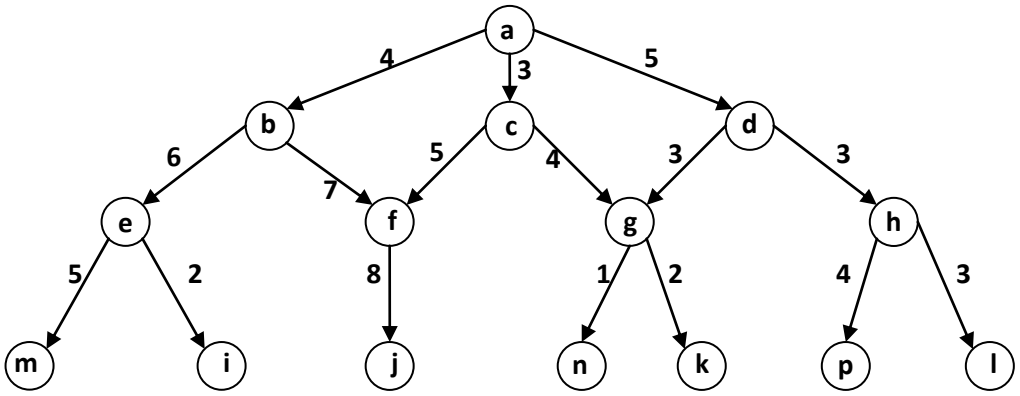
Open	**Closed**
[a]	[]
[z8, y9, x7]	[a]
[x7, z8, y9]	[a]
[p8, q7, z8, y9]	[a, x7]
[q7, p8, z8, y9]	[a, x7]
[k4, g2, p8, z8, y9]	[a, x7, q7]
[g2, k4, p8, z8, y9]	[a, x7, q7]
[k4, p8, z8, y9]	[a, x7, q7, g2]

The goal (k) is found

The first advance approach to the best first search is known as A-search algorithm. A algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the initial state and the current state and is denoted by g(n). A notice will be mentioned here that the same steps that are used in the best first search are used in an A algorithm but in addition to the g(n) as follow;

f(n) = h(n) + g(n) where h(n) is the heuristic function that computes the heuristic value for each state n, and g(n) is the generation function that computes the actual distance (levels) between initial state to current state n.

Example:



Find the path from **a** to **k** using A-search algorithm

Open	Closed
[a]	[]
[b4, c3, d5]	[a]
[b4+1, c3+1, d5+1]	[a]
[c4, b5, d6]	[a]
[f5, g4, b5, d6]	[a, c4]
[f5+2, g4+2, b5, d6]	[a, c4]
[f7, g6, b5, d6]	[a, c4]
[b5, g6, d6,, f7]	[a, c4]
[e6, f7, g6, d6, f7]	[a, c4, b5]
[e6+2, f7+2, g6, d6, f7]	[a, c4, b5]
[g6, d6, f7, e8]	[a, c4, b5]
[n1, k2, d6, f7, e8]	[a, c4, b5, g6]

[n1+3, k2+3, d6, f7, e8]	[a, c4, b5, g6]
[n4, k5, d6, f7, e8]	[a, c4, b5, g6]
[k5, d6, f7, e8]	[a, c4, b5, g6, n4]
Stop the goal (k) is found	

The second advance approach to the best first search is known as A*-search algorithm. A* algorithm is simply define as a best first search plus specific function. This specific function represent the actual distance (levels) between the current state and the goal state and is denoted by g(n).

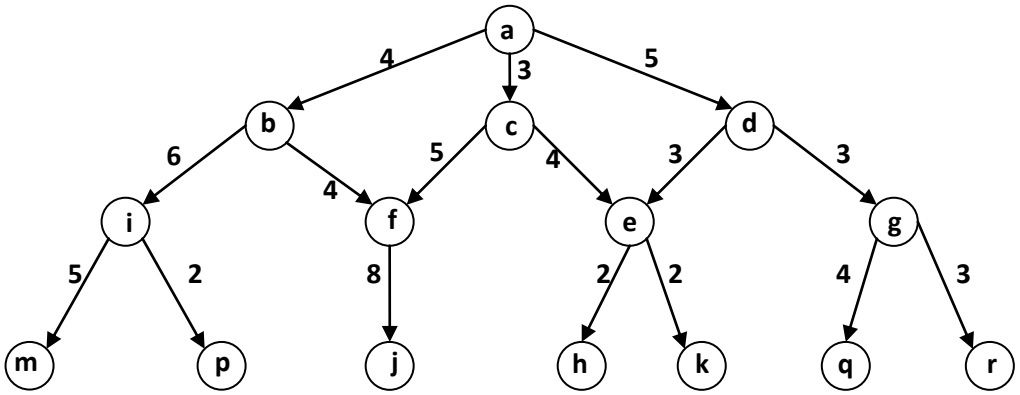
f(n) = h(n) + g(n) where h(n) is the heuristic function that computes the heuristic value for each state n, and g(n) is the generation function that computes the actual distance (levels) between current state n to goal state.

Function A* Search Algorithm

```
Begin
 Open: = [Initial state];                    %initialize
 Closed: = [ ];
 While open <> [ ] do                        %states remain
  Begin
   Remove leftmost state from open, call it X;
   If X = goal then return the path from initial to X
   Else
   Begin
    Generate children of X;
    For each child of X do
```

```
Begin
 Add the distance between current state to goal state to the heuristic
value for each child                                    %make the g(n)
Case
 The child is not on open or closed;
  Begin
   Assign the child a heuristic value;
   Add the child to open
  End;
 The child is already on open;
  If the child was reached by a shorter path
  Then give the state on open the shorter path
 The child is already on closed;
  If the child was reached by a shorter path then
   Begin
    Remove the state from closed;
    Add the child to open
   End;
End;                                                    %case
Put X on closed;
Re-order states on open by heuristic merit (best leftmost)
End;
Return FAIL                                             %open is empty
End.
```

Example:



Find the path from **a** to **k** using A*-search algorithm

Open	Closed
[a]	[ ]
[b4, c3, d5]	[a]
[b4+4, c3+2, d5+2]	[a]
[c5, d7, b8]	[a]
[f5, e4, d7, b8]	[a, c5]
[f5+3, e4+1, d7, b8]	[a, c5]
[e5, d7, f8, b8]	[a, c5]
[h2, k2, d7, f8, b8]	[a, c5, e5]
[h2+2, k2+0, d7, f8, b8]	[a, c5, e5]
[k2, h4, d7, f8, b8]	[a, c5, e5]

Stop, the goal (k) is found

Heuristic Search Methods with Heuristic Function

Hill Climbing

For each state f(n) = h(n) where h(n) is the heuristic function that computes the heuristic value for each state n.

Best First Search

For each state f(n) = h(n) where h(n) is the heuristic function that computes the heuristic value for each state n.

A-search algorithm

f(n) = h(n) + g(n) where h(n) is the heuristic function that computes the heuristic value for each state n, and g(n) is the generation function that computes the actual distance (levels) between initial state to current state n.

A*-search algorithm

f(n) = h(n) + g(n) where h(n) is the heuristic function that computes the heuristic value for each state n, and g(n) is the generation function that computes the actual distance (levels) between current state n to goal state.

Problems with Hill Climbing Search Procedure

1- Fost Hill (Local Minima)

This problem causes stopping search procedure.

The algorithm not found the goal state although it is existed in the search space, this is because of the algorithm search performance and behavior which depends on a determined strategy without backing path from dead end state which causes algorithm termination, this problem can be solved by using backtracking process in the algorithm strategy.

2- Plateau Problem

This problem causes stopping search procedure.

When the search procedure reach to a state has an equivalent heuristic values (choices), the algorithm stops searching for the goal and not get the solution path although it is existed in the search space, in other words, there is a state has two or

more children with the same heuristic value (Plateau partial search space), this problem can be solved by some kind of search procedures such as continuing search with the most left side.

3- Ridge Problem

This problem does not cause stopping search procedure.

The search procedure gets the solution path with some cost measurements which is not considered the best, since the best path is existed in dominate partial search space; this problem can be solved by applying more than one rule in each search procedure stage.

A Comparison between Heuristic Search and Blind Search

	Blind Search	**Heuristic Search**
1	In term of complexity: it is less complex.	In term of complexity: it is more complex.
2	In term of memory capacity: usually need more memory capacity.	In term of memory capacity: usually need less memory capacity.
3	In term of run time consuming: usually consumes more run time.	In term of run time consuming: usually consumes less run time.
4	Guarantee for solution.	Guarantee for solution, except Hill Climbing (not always).
5	Usually does not find the optimal solution path.	Usually finds the optimal solution path or nearly the optimal solution path.
6	It does not have a guider in search behavior.	It has a guider in search behavior (Heuristic Function).
7	It is not efficient in game playing.	It is efficient in game playing such as Minmax or Alpha-Beta procedures.

Using Heuristic in Games

The sliding-tile puzzle consists of three black tiles, three white tiles, and an empty space in the configuration shown in Figure (1). The puzzle has two legal moves with associated costs:

- A tile may move into an adjacent empty location. This has a cost of 1.
- A tile can hop over one or two other tiles into the empty' position, this has a cost equal to the number of tiles jumped over.

The goal is to have all the white tiles to the left of all the black tiles. The position of the blank is not important.

a. Analyze the state space with respect to complexity and looping.

b. Propose a heuristic for solving this problem and analyze it.

Figure (5), the sliding block puzzle

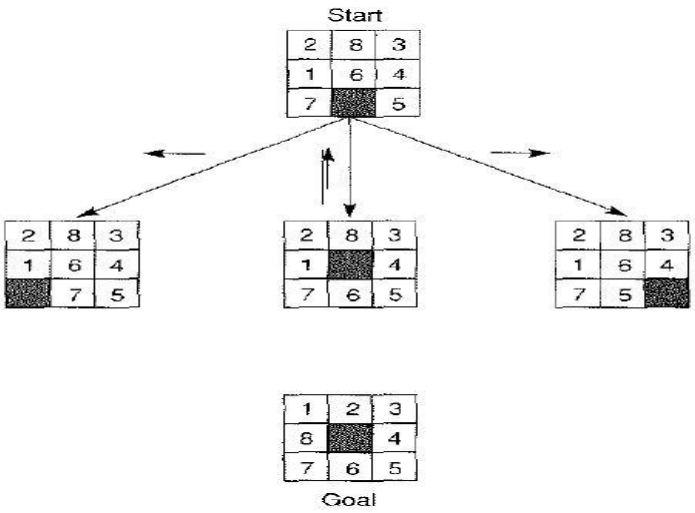
The 8-puzzle Problem

We now evaluate the performance of several different heuristics for solving the 8-puzzle. Figure (6), shows a start and goal state for the 8-puzzle, along with the first three states generated in the search.

The simplest heuristic counts the tiles out of place in each state when it is compared with the goal. This is intuitively appealing, because it would seem that, all else being equal; the state that had fewest tiles out of place is probably closer to the desired goal and would be the best to examine next.

However, this heuristic does not use all of the information available in a board configuration, because it does not take into account the distance the tiles must be moved.

A "better" heuristic would sum all the distances by which the tiles are out of place, one for each square a tile must be moved to reach its position in the goal state. Both of these heuristics can be criticized for failing to acknowledge the difficulty of tile reversals. That is, if two tiles are next to each other and the goal requires their being in apposite locations, it takes (many) more than two moves to put them back in place, as the tiles must "go around" each other (Figure 7).



Figure(6), The start state, first moves, and goal state for an example 8-puzzle.

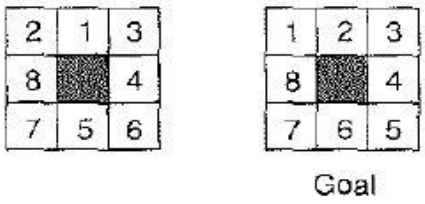


Figure (7) An 8-puzzle state with a goal and two reversals: 1 and 2, 5 and 6.

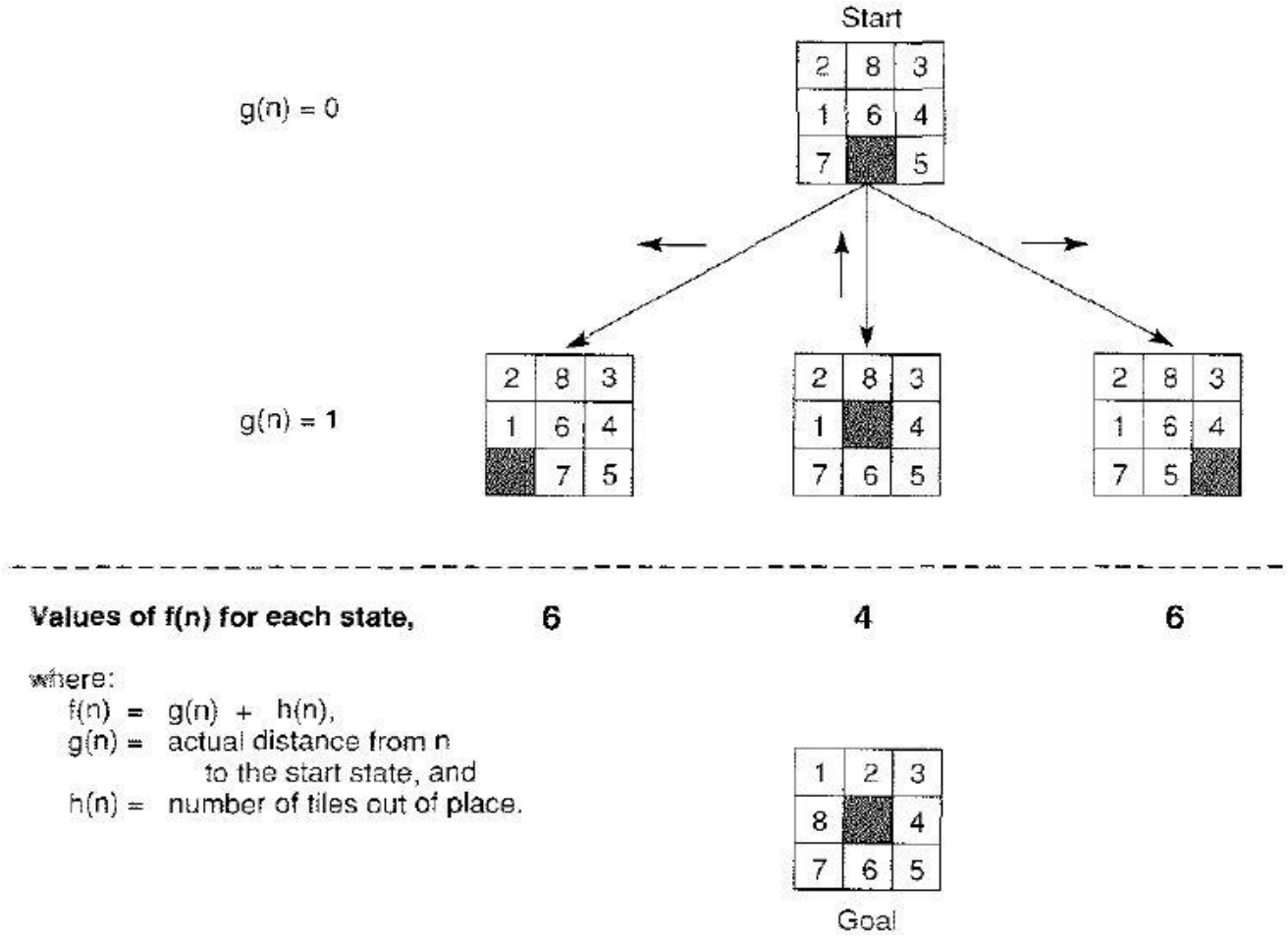


Figure (8), the 8-puzzle problem solving with heuristic values

For the 8-puzzle Grid

There is one center location.

There are four corners location.

There are four sides location.

Possible Moves

- *When the space position is in the center of the grid, possible moves = 4.*
- *When the space position is in the corner of the grid, possible moves = 2*
- *When the space position is in the side of the grid, possible moves = 3.*

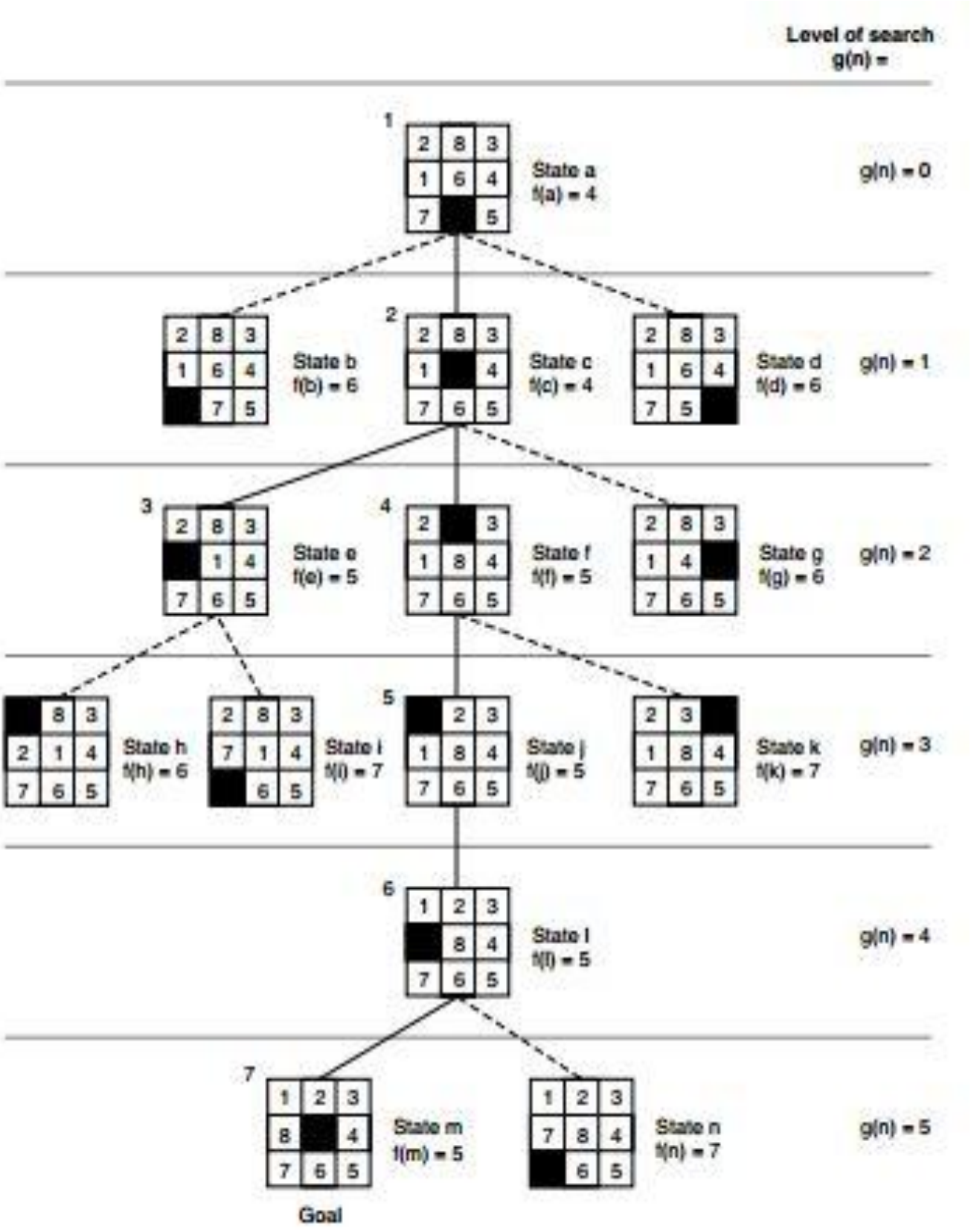
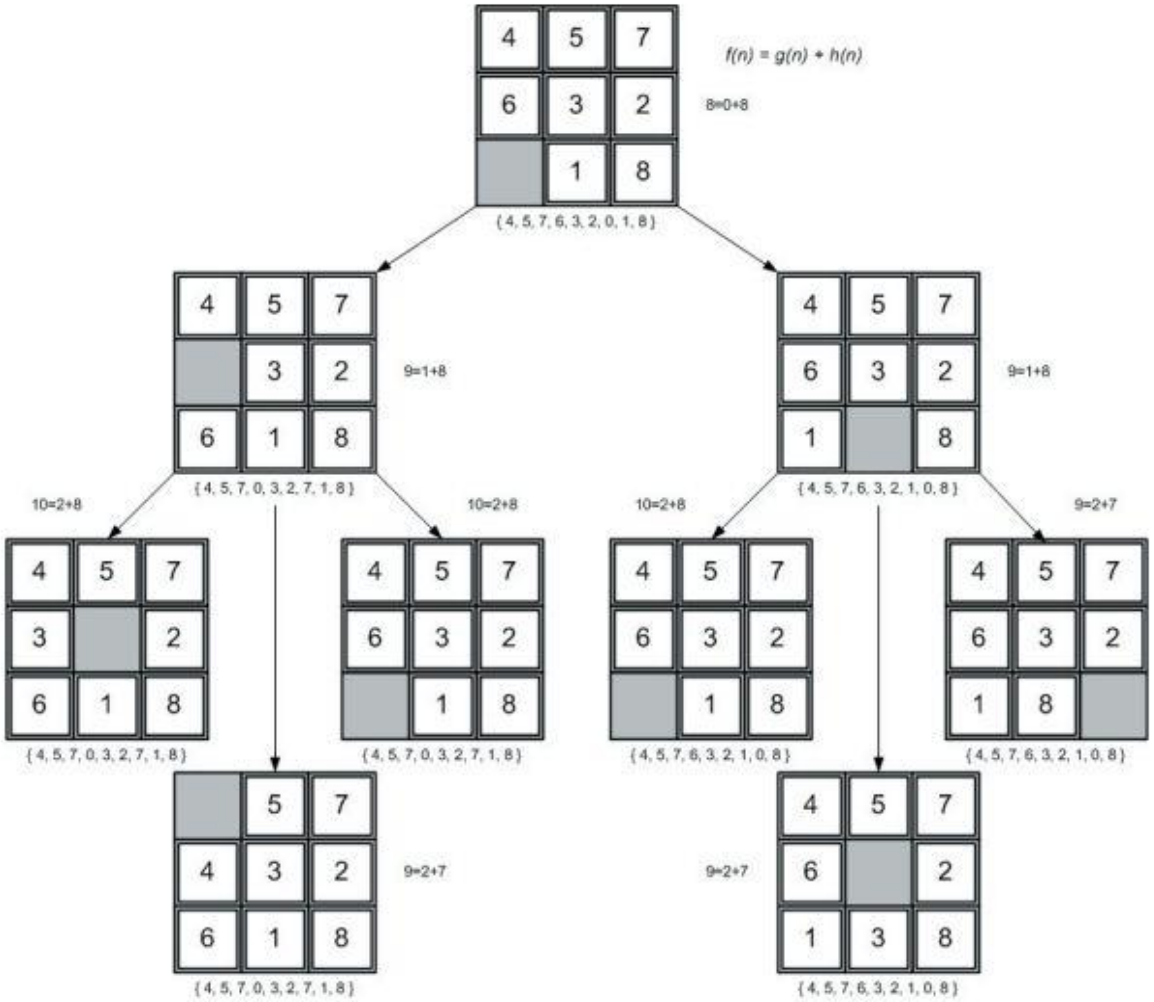


Figure (9), the 8-puzzle problem solved by A-search algorithm

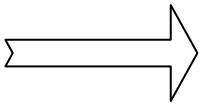
Another Examples of 8-Puzzle Problem



Consider the following **8-puzzle** problem then draw the problem state space to find the goal using A-search algorithm (or Best first or Hill climbing) then list the *solution path.*

	R	I
A	O	O
P	L	G

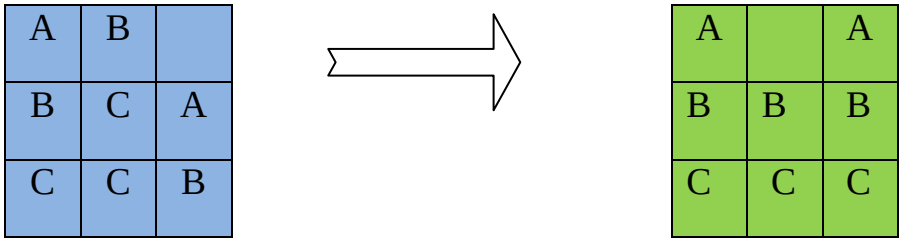
I.S



A		I
P	R	O
L	O	G

G.S

Consider the following 8-tiles problem then draw the problem state space to give the following requirements:



I.S G.S

Find the goal using Best first search (or A-algorithm or Hill climbing) algorithm

---

Single Solution Metaheuristics

Some Metaheuristic Classifications

• **Nature inspired versus nonnature inspired:** Many metaheuristics are inspired by natural processes: evolutionary algorithms and artificial immune systems from biology; ants, bees colonies, and particle swarm optimization from swarm intelligence into different species (social sciences); and simulated annealing from physics.

• **Memory usage versus memoryless methods:** Some metaheuristic algorithms are memoryless; that is, no information extracted dynamically is used during the search. Some representatives of this class are local search, GRASP, and simulated annealing. While other metaheuristics use a memory that contains some information extracted online during the search. For instance, short-term and long-term memories in tabu search.

• **Deterministic versus stochastic:** A deterministic metaheuristic solves an optimization problem by making deterministic decisions (e.g., local search, tabu search). In stochastic metaheuristics, some random rules are applied during the search (e.g., simulated annealing, evolutionary algorithms). In deterministic algorithms, using the same initial solution will lead to the same final solution, whereas in stochastic metaheuristics, different final solutions may beobtained from the same initial solution. This characteristic must be taken into account in the performance evaluation of metaheuristic algorithms.

• **Population-based search versus single-solution based search:** Single-solution based algorithms (e.g., local search, simulated annealing) manipulate and transform a single solution during the search while in population-based

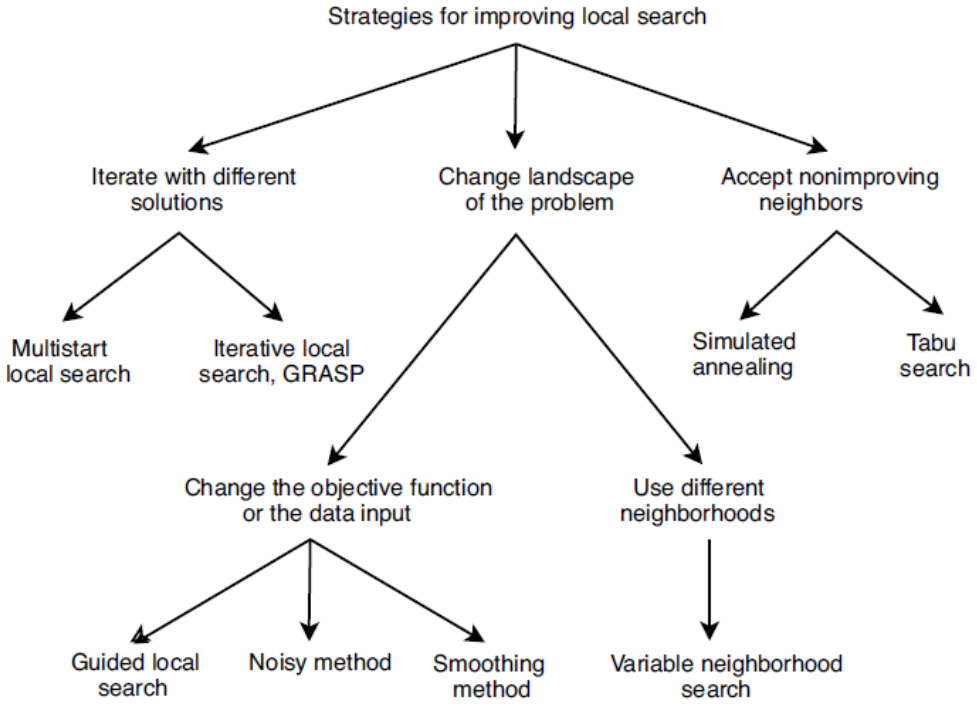
algorithms (e.g., particle swarm, evolutionary algorithms) a whole population of solutions is evolved. These two families have complementary characteristics: single-solution based metaheuristics are exploitation oriented; they have the power to intensify the search in local regions. Population-based metaheuristics are exploration oriented; they allow a better diversification in the whole search space. In the next chapters of this book, we have mainly used this classification. In fact, the algorithms belonging to each family of metaheuristics share many search mechanisms.

• **Iterative versus greedy:** In iterative algorithms, we start with a complete solution (or population of solutions) and transform it at each iteration using some search operators. Greedy algorithms start from an empty solution, and at each step a decision variable of the problem is assigned until a complete solution is obtained. Most of the metaheuristics are iterative algorithms.

Local Search

$s = s_0$; /* Generate an initial solution s_0 */
While not Termination_Criterion **Do**
  Generate ($N(s)$) ; /* Generation of candidate neighbors */
  If there is no better neighbor **Then** Stop ;
  $s = s'$; /* Select a better neighbor $s' \in N(s)$ */
Endwhile
Output Final solution found (local optima).

Others…

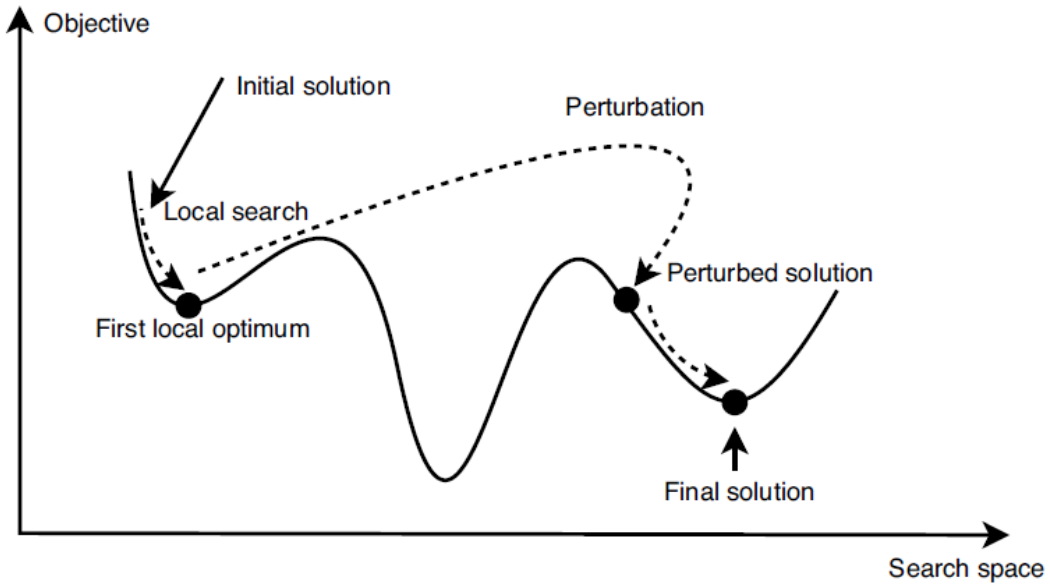


Simulated Annealing

Input: Cooling schedule.
$s = s_0$; /* Generation of the initial solution */
$T = T_{max}$; /* Starting temperature */
Repeat
  Repeat /* At a fixed temperature */
    Generate a random neighbor s' ;
    $\Delta E = f(s') - f(s)$;
    If $\Delta E \leq 0$ **Then** $s = s'$ /* Accept the neighbor solution */
    Else Accept s' with a probability $e^{\frac{-\Delta E}{T}}$;
  Until Equilibrium condition
  /* e.g. a given number of iterations executed at each temperature T */
  $T = g(T)$; /* Temperature update */
Until Stopping criteria satisfied /* e.g. $T < T_{min}$ */
Output: Best solution found.

Tabu Search

Initialize
Identify initial *Solution,* Create empty *TabuList,* Set *BestSolution=Solution*
Define *TerminationConditions*
Done=FALSE
Repeat
if value of *Solution* > value of *BestSolution* **then** *BestSolution*D*Solution*
if no *TerminationConditions* have been met **then**
begin
add *Solution* to *TabuList*
if *TabuList* is full **then** delete oldest entry from *TabuList*
find *NewSolution* by some transformation on *Solution*
if no *NewSolution* was found **or**
if no improved *NewSolution* was found for a long time **then**
generate NewSolution at random
if *NewSolution* not on *TabuList* **then**
Solution D *NewSolution*
end
else
Done=TRUE
until done=TRUE



GRASP

Greedy Randomized Adaptive Search Procedure

Input: Number of iterations.
Repeat
s = Random-Greedy(seed) ; /* apply a randomized greedy heuristic */
$s' = Local - Search(s)$; /* apply a local search algorithm to the solution */
Until Stopping criteria /* e.g. a given number of iterations */
Output: Best solution found.

$s = \{\}$; /* Initial solution (null) */
Evaluate the incremental costs of all candidate elements ;
Repeat
Build the restricted candidate list RCL ;
/* select a random element from the list RCL */
e_i = Random-Selection(RCL) ;
If $s \cup e_i \in F$ **Then** /* Test the feasibility of the solution */
$s = s \cup e_i$;
Reevaluate the incremental costs of candidate elements ;
Until Complete solution found.

Guided Local Search

Input: S-metaheuristic LS, λ, Features I, Costs c.
$s = s_0$ /* Generation of the initial solution */
$p_i = 0$ /* Penalties initialization */
Repeat
Apply a S-metaheuristic LS ; /* Let s^* the final solution obtained */
For each feature i of s^* **Do**
$u_i = \frac{c_i}{1+p_i}$; /* Compute its utility */
$u_j = max_{i=1,..m}(u_i)$; /* Compute the maximum utilities */
$p_j = p_j + 1$; /* Change the objective function by penalizing the feature j */
Until Stopping criteria /* e.g. max number of iterations or time limit */
Output: Best solution found.

Variable Neighborhood Search (VNS)

Input: a set of neighborhood structures N_k for $k = 1, \ldots, k_{max}$ for shaking.
a set of neighborhood structures N_l for $k = 1, \ldots, l_{max}$ for local search.
$x = x_0$; /* Generate the initial solution */
Repeat
For k=1 **To** k_{max} **Do**
Shaking: pick a random solution x' from the k^{th} neighborhood $N_k(x)$ of x ;
Local search by VND ;
For l=1 **To** l_{max} **Do**
Find the best neighbor x'' of x' in $N_l(x')$;
If $f(x'') < f(x')$ **Then** $x' = x''$; l=1 ;
Otherwise l=l+1 ;
Move or not:
If local optimum is better than x **Then**
$x = x''$;
Continue to search with N_1 $(k = 1)$;
Otherwise k=k+1 ;
Until Stopping criteria
Output: Best found solution.

References:

1. George F. Luger, "***Artificial Intelligence Structures and Strategies for Complex Problem Solving"***, Pearson Education Asia (Singapore), Sixth edition.
2. Stuart J. Russell and Peter Norvig, ***"Artificial Intelligence A Modern Approach"***, Second Edition, Prentice Hall.
3. Amit Konar, ***"Artificial Intelligence and Soft Computing"***, Behavior and Cognitive Modeling of the Human Brian, CRC Press.
4. El-Ghazali Talbi, ***"METAHEURISTICS FROM DESIGN TO IMPLEMENTATION"***, University of Lille – CNRS – INRIA, WILEY, A JOHN WILEY & SONS, INC, PUBLICATION.