University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

بحـــث ذكـي متقدم

Prof. Dr. Alia Karim Abdul Hassan

ا.دعلياء كريم عبدالحسن

cs.uotechnology.edu.iq

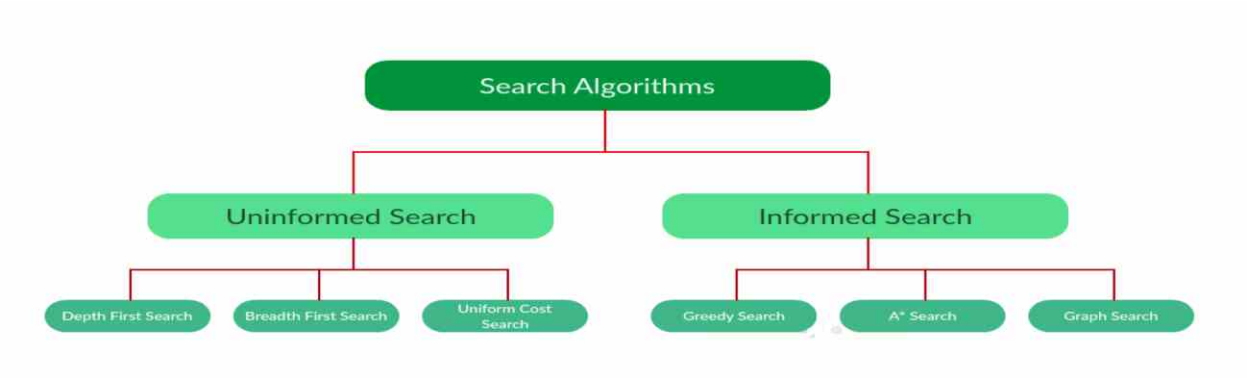
1.Introduction

We have a problem and want to find a solution. In **computer science** and in the part of artificial intelligence that deals with algorithms, problem solving encompasses a number of techniques known as algorithms, heuristics, root cause analysis, etc. *Artificial Intelligence* is the study of building agents that act rationally. most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

A search problem consists of:

- State Space. Set of all possible states where you can be.
- Start State. The state from where the search begins.
- Goal State. A function that looks at the current state returns whether or not it is the goal state.

The Solution to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state.This plan is achieved through search algorithms. Types of search algorithms: There are far too many powerful search algorithms.



2. HEURISTICS

AI applications often rely on the application of heuristics. **heuristic** :is a set of guidelines that *often* works to solve a problem. Contrast a heuristic with an **algorithm**, which is a prescribed set of rules to solve a problem and whose output is entirely predictable. The reader is undoubtedly familiar with many algorithms used in computer programs, such as those for sorting, including bubble sort and quicksort, and for searching, including sequential search and binary search. With a heuristic, a

favorable outcome is likely, but is not guaranteed. Heuristic methods were especially popular in the early days of AI, a period including the 1950s and into the 1960s.

3. The Water Jug Problem: Working Backward

You are provided with two jugs of sizes *m* and *n* respectively; and you are re- quired to measure *r* quarts of water where *m*, *n,* and *r* are all different quantities. An instance of this problem is: How can you measure exactly twelve quarts of water from a tap or a well when you have only an eight-quart jug and an eighteen- quart jug? See Figure 1.11. One way to solve the problem is to use trial and error and hope for the best. Instead, Polya suggests the heuristic of starting with the goal state and working backward. See Figure 1.12.

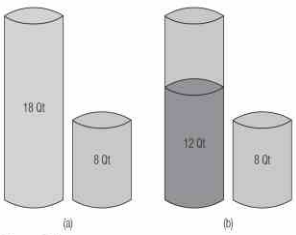


Figure 1.11
Water Jug Problem. (a) Initial state and (b) Final state.

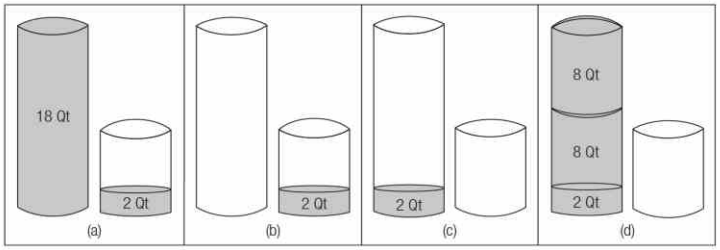


Figure 1.12
Starting with the goal state and working backward. (Drawing by authors.)

(a) the eighteen-quart jug has been filled up and there are two quarts of water in the eight-quart jug.

This state is just one step away from the goal state, where you pour an additional six quarts of water into the eight-quart jug; where twelve quarts of water remains in the eighteen-quart jug. Parts (b) through (d) of the figure provide the requisite steps to reach this penultimate state in part (a). You should turn your attention to part (d) and work your way back to portion (b) to see all the states that precede the state depicted in part (a).

Working backward to solve the Water Jug Problem and measure 12 quarts of water using only an 18-quart pail and an eight-quart pail, path (a), (b), (c), (d) shows how to go from the desired goal state back to the initial state.

To actually solve this problem, you reverse the order of the states. First fill the 18-quart pail (state d). Then fill and empty the eight-quart pail twice by transferring water from the 18-quart pail. This leaves you with two quarts in the 18-quart pail (state c). Pour the last two quarts into the eight-quart pail (state b). Fill the 18-quart pail again from the tap or well, and pour water from the larger container to fill the eight-quart pail, which removes six quarts from the 18, leaving 12 quarts in the larger pail (state a).

4.IDENTIFYING PROBLEMS SUITABLE FOR AI

There are three characteristics that are common to most AI problems:
1. AI problems tend to be large.
2. They are computationally complex and cannot be solved by straightforward algorithms.
3. AI problems and their domains tend to embody a large amount of human expertise, especially if tackled by strong AI methods.

Some types of problems are better solved using AI, whereas others are more suitable for traditional computer science approaches involving simple decision-making or exact computations to produce solutions. Let us consider a few examples:
• Medical diagnosis
• Shopping using a cash register with barcode scanning
• ATMs :
• Two person games such as chess and checkers

Medical diagnosis is a field of science that has for many years employed and welcomed contributions from AI, particularly through the development of **expert systems**. Expert systems are typically built in domains where there is considerable human expertise and where there exist many rules (rules of the form: if condition, then action; for example: if you have a headache, then take two aspirins and call me in the morning.) more rules than any human can or wishes to hold in his/her head. Expert systems are among the most successful AI techniques for producing results that are comprehensive and effective.

HW: write simple introduction about MYCIN.

ATMs : a general financial advisor, however, keeping track of a person's spending, as well as the categories and frequencies of items purchased. The machine could interpret spending for entertainment, necessities, travel, and other categories and offer advice on how spending patterns might be beneficially altered. ("Do you really need to spend so much on fancy restaurants?") An ATM as described here would be considered an intelligent system.

Another example of an intelligent system is one that plays chess. Although the rules of chess are easy to learn, playing this game at an expert level is no easy matter. It is generally accepted that chess has some 10^{42} possible reasonable games (whereby "reasonable" games are distinguished from the number of "possible" games earlier given as 10^{120}). This is such a large number that, even if the entire world's fastest computers worked together to solve the game of chess (i.e., develop a program to

play perfect chess, one which always makes the best move), they wouldn't finish for 50 years.

5. Search Algorithms and Puzzles

The 15-puzzle and related search puzzles, such as the **8-puzzle** and the **3-puzzle**, serve as helpful examples of search algorithms, problem-solving techniques, and the application of heuristics. **The blank can move in one of four directions:**
• Up (↑) • Down (↓) • Right (→) • Left (←)

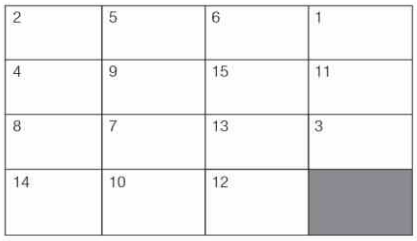


Figure 1.13
Setting up a 15-puzzle.

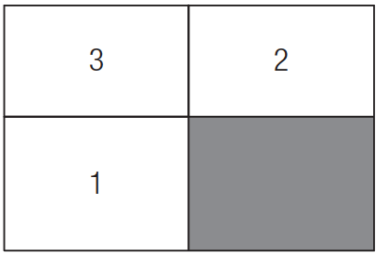


Figure 1.14
Using a 3-puzzle.

Notice that the 3 is free to move down, while the 12 is free to move to the right. Smaller instances of this puzzle are more convenient to work with, including the 8-puzzle and the 3-puzzle. For example, consider the 3-puzzle, shown in Figure 1.14. In these puzzles, it is naturally the numbered tiles that slide; however, it is more convenient to consider the blank square to be moving. The objective of this puzzle is to get from the start state to the goal state. In some instances, a solution with the minimum number of moves is desired. The structure that corresponds to all possible states of a given problem is called the **state-space graph**.

The graph consists of all possible states of a problem, denoted by nodes, with arcs representing all legal transitions between states (legal moves in a puzzle). The **space tree**, which is generally a proper subset of the state-space graph, is a tree whose root is the start state, and one or more of its leaves is a goal state. One search methodology you can use to traverse state-space graphs is called a **blind search**. It presumes no knowledge of the search space for a problem. There are two classic blind search algorithms that are often explored in courses on data structures and algorithms; they are **depth first search (dfs)** and **breadth first search (bfs)**. In dfs, you plunge as deeply into the search tree as possible. That is, when you have a choice of moves, you usually (but not always) move left. With bfs, you first visit all nodes close to the root, level by level, usually moving from left to right. A dfs traversal of the tree, shown in Figure 1.16, would inspect nodes in the order A, B, D, E, C, F, G.

Meanwhile, a bfs traversal of this tree would visit the nodes in the order A, B, C, D, E, F, G.

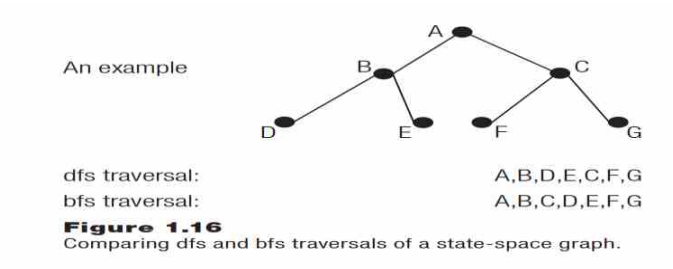


Figure 1.16
Comparing dfs and bfs traversals of a state-space graph.

HW: solve 8- puzzle with dfs and bfs

6. Combinatorial explosion: means that the number of possible states of the puzzle is too high to be practical. Solving problems of a reasonable size can involve search spaces that grow too rapidly to allow blind search methods to succeed. (This will remain true regardless of how fast computers become in the future.) For example, the state-space graph for the 15-puzzle might contain more than $16! \leq (2.09228 \times 10^{13})$ states. Because of combinatorial explosion, success with AI problems depends more upon the successful application of heuristics than the design of faster machines. Whenever two or more alternative paths appear, these algorithms pursue the path or paths closest to the goal. The algorithms can use heuristic estimates of remaining distance, however.

Reference:

Stephen Lucci and Danny Kopec. *ARTIFICIAL INTELLIGENCE IN THE 21ST CENTURY: A Living Introduction 2/E* . Copyright ©2016 by MERCURY LEARNING AND INFORMATION.

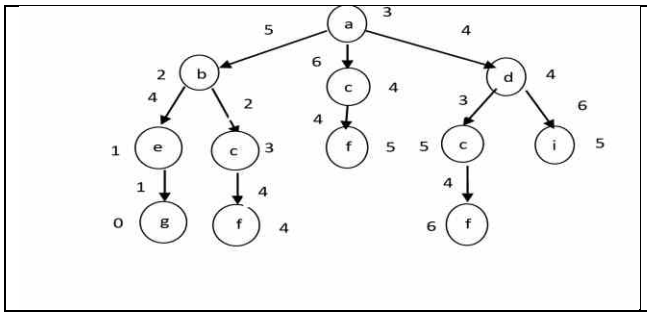
1.A* and D* Algorithms

A* algorithm is simply define as a best first search plus specific function. This specific function represents the actual distance (levels) between the current state and the goal state and is denoted by h(n). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: f(n) = g(n) + h(n).

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f (n) = estimated cost of the cheapest solution through n .

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.

Example:



Open	Closed
[A3]	[]
[B7,D8,C10]	[A3]
[D8,E10,C10]	[A3,B7]
[E10,C10,I15]	[A3,B7,D8]
[G10,C10,I15]	[A3,B7,D8,E10]
	[A3,B7,D8,E10,G10]

The goal is found &the resulted path is:

A0 → B5 → D4 → E4 → G1 =14

1.2 A* Algorithm Properties:-

1) Admissibility

Admissibility means that h(n) is less than or equal to the cost of the minimal path from n to the goal

2) Consistency

Consistency means that the difference between the heuristic of a state and the heuristic of its descendent is less than or equal the cost between them, and the heuristic of the goal equal zero. In other words, **1) h(ni)-h(nj) ≤ cost(ni,nj). 2) h(goal)=0.**

3) Informedness

For two A* heuristics h1 and h2 , if $h1(n) \leq h2(n)$, for all states n in the search space , heuristics h2 is said to be more informed than h1.

Example: Example of A* Search Algorithm Informedness:

Consider the problem of finding the shortest path through a maze from a starting point (S) to an ending point (E). Let's compare two heuristic functions to illustrate the concept of informedness:

Heuristic 1: Manhattan Distance

This heuristic estimates the remaining distance to the goal by calculating the straight-line distance between the current node and the goal node, ignoring obstacles in the maze.

Heuristic 2: Number of Walls Left

This heuristic estimates the remaining distance by counting the number of walls that need to be traversed to reach the goal.

Informedness comparison:

• Heuristic 1 is less informed because it does not consider the actual maze layout, potentially guiding the search towards paths with more turns even if they are straight-line shorter.

• Heuristic 2 is more informed because it takes into account the actual obstacles in the maze, guiding the search towards paths with fewer turns, even if they might be slightly longer in straight-line distance.

1.3 The A* Search

The last incantation of branch and bound search is the A* search. This approach employs branch and bound with both estimates of remaining distance and dynamic programming. The A* algorithm is shown below:

```
//A* Search
A* Search (Root_Node, Goal)
{
Create Queue Q
Insert Root_Node into Q
```

```
While (Q_Is_Not_Empty)
        {
        G = Remove from Q Mark G visited
        If (G= goal) Return the path from Root_Node to G;
        Else Add each child node's estimated distance to current distance.
        Insert the children of G which have not been previously visited into the Q
        Sort Q by path length
        } // end while
Return failure
}// end of A* function.
```

1.4 EXAMPLE : the 3-PUZZLE to illustrate A* search

The solution to this puzzle via the A* search is shown in Figure 3.25.

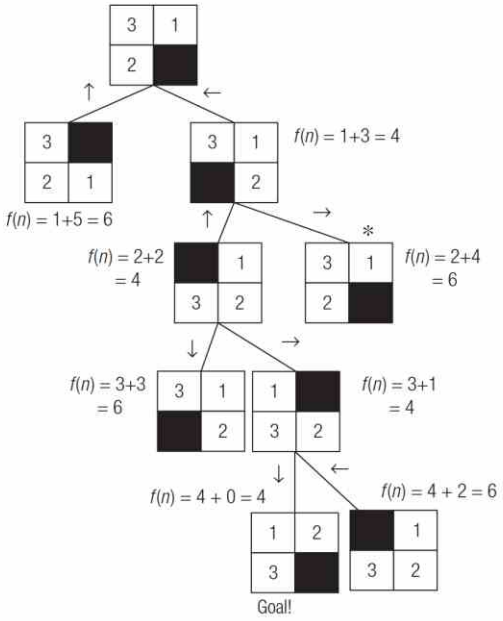


Figure 3.25
A* search that employs total Manhattan distance as the heuristic estimate. Refer to the node marked with a * in level 3 of this tree. Tile 1 must travel one square left; tile 2 must move one square east and one square north (or equivalently one square north and then one square east) and tile 3 needs to travel one square south. Hence the sum of the Manhattan distances, h(n) = 1 + 2 + 1 = 4 for this node.

The term Manhattan distance is employed because a tile must travel north, south, east, and west similar to how a taxi cab would maneuver through the streets of Manhattan.

Observe that the A* search in the above example employs Manhattan distance as a heuristic is *more informed* and used a number of tiles out of place as a heuristic estimate of remaining distance.

2. D-Star (*D)**, short for dynamic A* is a sensor based algorithm that deals with dynamic obstacles by real time changing its edge's weights. It always computes a shortest path from its current cell to the start cell under the assumption that cells with unknown blockage status are traversable. D* maintains a list of nodes which is used to propagate information about changes of the cost function.

2.1 Algorithm

The algorithm works by iteratively selecting a node from the OPEN list and evaluating it. D* begins by searching backwards from the goal node towards the start node. Each expanded node has a back pointer which refers to the next node leading to the target, and each node knows the exact cost to the target.

Example1: D* algorithm is the dynamic A*
A* algorithm equation:
f(n) = g(n) + h(n)
D* algorithm equation:
f(n) =h(n)
It could be propagated cost changes to its neighbors as shown in figure (1).
N(x1,x2)=1
N(x1,x3)=1.414
N(x1,x4)=100000, if x4 has an obstacle and x1 is a free cell.
N(x1,x5)=100000.4, if x5 has an obstacle and x1 is a free cell
The arc cost of the initial node = (0.0) Where the arc cost of vertical and horizontal nodes are calculated as:
From x1 to x2 =$\sqrt{}$ $(\mathbf{00}-\mathbf{00})\mathbf{22}+(\mathbf{00}-\mathbf{11})\mathbf{22}=\mathbf{1}$
From x1 to x4 = $\sqrt{}$ $(\mathbf{00}-\mathbf{11})\mathbf{22}+(\mathbf{00}-\mathbf{00})\mathbf{22}=\mathbf{1}$
But the arc cost of diagonal nodes can be calculated as:
From x1 to x3 = $\sqrt{}$ $(\mathbf{00}-\mathbf{11})\mathbf{22}+(\mathbf{00}-\mathbf{11})\mathbf{22}=\sqrt{}\mathbf{22}=\mathbf{1.414}$
And so on.

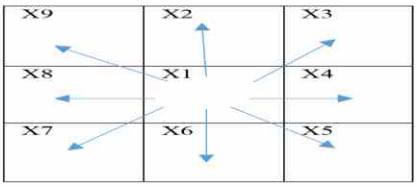


Figure 1: Node expansion

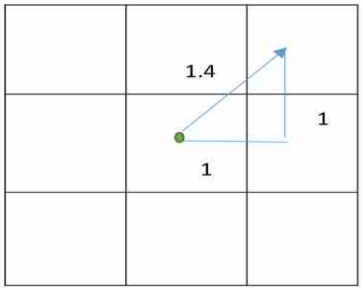


Figure 2: Calculation of node expansion

A* and D* are both pathfinding algorithms, but they have different approaches and applications. A* is a heuristic search algorithm that guarantees the shortest path from a start node to a goal node. It uses heuristics to guide the search and is suitable for static environments where the entire map is known in advance. D*, on the other hand, is an incremental search algorithm designed to efficiently update the path when the environment changes. It is suitable for dynamic environments where the map is not entirely known in advance and may change over time. For example, in a scenario where a robot needs to navigate through a changing environment with dynamic obstacles, D* would be more suitable due to its ability to adapt the path as the environment changes. In contrast, A* would be more appropriate for finding the shortest path in a known, static environment.

2.3 Example:

For more detailed explanation of the Trace D* search algorithm for 4x4 environment with the given start, goal, and obstacle positions:

- Initialization:

• Open List: [(1, 1), f(1, 1) = 0 (h(1, 1) = estimated distance to goal + g(1, 1) = 0)]
• Closed List: Empty

Iteration 1:

1- Remove (1, 1) from the Open List (current cell).
2- Expand (1, 1):
◦ Up: (2, 1), tentative g(2, 1) = 1, h(2, 1) = 3, f(2, 1) = 4.
◦ Right: (1, 2), tentative g(1, 2) = 1, h(1, 2) = 2, f(1, 2) = 3.
◦ Down: (2, 2) (ignoring for now due to obstacle at (2, 3)).
3 Add (2, 1) and (1, 2) to the Open List.
4 Closed List remains empty.

Iteration 2:

1- Remove (1, 2) from the Open List (current cell).
2- Expand (1, 2):
◦ Up: (2, 2) (ignoring for now due to obstacle at (2, 3)).
◦ Right: (1, 3), tentative g(1, 3) = 2, h(1, 3) = 1, f(1, 3) = 3.
◦ Down: (2, 3) (ignoring due to obstacle).
3- Add (1, 3) to the Open List.
4- Closed List: [(1, 2), f(1, 2) = 3].
Iteration 3:
1 -Remove (2, 1) from the Open List (current cell).
2 - Expand (2, 1):
◦ Right: (2, 2) (already in Closed List with f(2, 2) = 4, tentative g(2, 2) = 2 is better, update to f(2, 2) = 3).
◦ Down: (3, 1), tentative g(3, 1) = 2, h(3, 1) = 3, f(3, 1) = 5.
3- Update (2, 2) in Closed List and add (3, 1) to the Open List.
4- Closed List: [(1, 2), f(1, 2) = 3], [(2, 2), f(2, 2) = 3].
Iteration 4: 1 Remove (1, 3) from the Open List (current cell).
2- Expand (1, 3):
◦ Up: (2, 3) (ignoring due to obstacle).
◦ Right: (1, 4), goal reached! f(1, 4) = 3 (g(1, 4) = 3, h(1, 4) = 0).
3- Stop searching, goal found.
Trace back the path:
• (1, 4) -> (1, 3) -> (1, 2) -> (1, 1) (following back pointers in the Closed List).
Note: This is just one possible path, and the order of cell expansions may differ depending on tie-breaking rules and how you handle the obstacle. The key takeaway is the incremental nature of Trace D*, where the search continuously updates based on new information and changes in the environment.

Reference:

1) Stephen Lucci and Danny Kopec. *ARTIFICIAL INTELLIGENCE IN THE 21ST CENTURY: A Living Introduction 2/E* . Copyright ©2016 by MERCURY LEARNING AND INFORMATION.
2) Path Planning Algorithm using D* Heuristic Method Based on PSO in Dynamic Environment Firas A. Raheema*, Umniah I. Hameedb, *American Scientific Research Journal for Engineering, Technology, and Sciences (ASRJETS) (2018) Volume 49, No 1, pp 257-271*

3) https://dibyendu-biswas.medium.com/d-d-lite-lpa-e7483779a7ca

1 .Advanced Intelligent Search

Search is an essential component of any intelligent system. “Informed Search,” demonstrated the ways heuristics enable you to search through the most auspicious portions of search trees. The inspiration for these heuristics derives from our insights into a problem, for example: How many tiles must be moved to solve an instance of the 8-Puzzle? Inspiration will be provided by natural systems both living and nonliving. This insight that the physical properties of a substance depend not only on its composition but also upon the arrangement of its molecules, and that this arrangement can be modified is the impetus behind annealing. Inside a computer program we can perform “artificial evolution.” Genetic Algorithms form. All wish to possess software that (magically) **writes itself to solve** the problems we are confronted with?

1.1 Common Concepts

- **Optimization**: occurs in the minimization of time, cost, and risk or the maximization of profit, quality, and efficiency. For instance, there are many possible ways to design a network to optimize the cost and the quality of service; there are many ways to schedule a production to optimize the time; there are many ways to predict a 3D structure of a protein to optimize the potential energy, and so on.
- **optimization problems** :A large number of real-life optimization problems in science, engineering, economics, and business are complex and difficult to solve. They cannot be solved in an exact manner within a reasonable amount of time. Using approximate algorithms is the main alternative to solve this class of problems.
- **Approximate algorithms** :
  Approximate algorithms can further be decomposed into two classes: specific heuristics and metaheuristics. Specific heuristics are problem dependent; they are designed and applicable to a particular problem.
- **Metaheuristics** :Metaheuristics solve instances of problems that are believed to be hard in general, by exploring the usually large solution search space of these instances. These algorithms achieve this by reducing the effective size of the space and by exploring that space efficiently. Metaheuristics serve three main purposes: solving problems faster, solving large problems, and obtaining robust algorithms. Moreover, they are simple to design and implement, and are very flexible.

- *heuristic :*The word *heuristic* has its origin in the old Greek word *heuriskein*, which means the art of discovering new strategies (rules) to solve problems. The suffix *meta*, also a Greek word, means "upper level methodology."

- *Metaheuristic:* The term *metaheuristic* was introduced by F. Glover in the paper. Metaheuristic search methods can be defined as upper level general methodologies (templates) that can be used as guiding strategies in designing underlying heuristics to solve specific optimization problems.

2. OPTIMIZATION MODELS

As scientists, engineers, and managers, we always have to take decisions. Decision making is everywhere. As the world becomes more and more complex and competitive, decision making must be tackled in a rational and optimal way. Decision making consists in the following steps (Fig. 1.1):

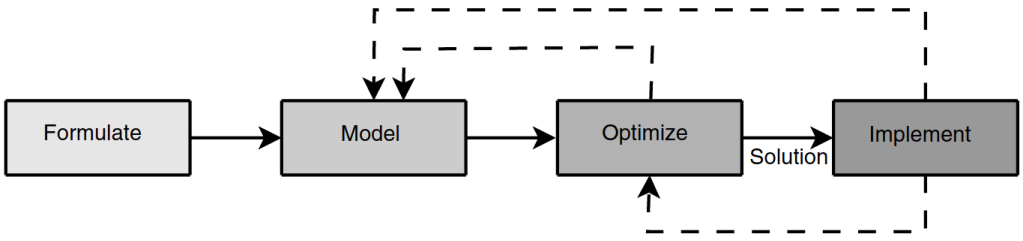


FIGURE 1.1 The classical process in decision making: formulate, model, solve, and implement. In practice, this process may be iterated to improve the optimization model or algorithm until an acceptable solution is found. Like life cycles in software engineering, the life cycle of optimization models and algorithms may be linear, spiral, or cascade.

• **Formulate the problem:** In this first step, a decision problem is identified. Then, an initial statement of the problem is made. This formulation may be imprecise. The internal and external factors and the objective(s) of the problem are outlined. Many decision makers may be involved in formulating the problem.

• **Model the problem:** In this important step, an abstract mathematical model is built for the problem.

• **Optimize the problem:** Once the problem is modeled, the solving procedure generates a "good" solution for the problem. The solution may be optimal or suboptimal. The algorithm designer can reuse state-of-the-art algorithms on similar problems or integrate the knowledge of this specific application into the algorithm.

• **Implement a solution:** The obtained solution is tested practically by the decision maker and is implemented if it is "acceptable." . If the solution is unacceptable, the model and/or the optimization algorithm has to be improved and the decision-making process is repeated.

- **exploration and exploitation**

metaheuristics allow to tackle large-size problem instances by delivering satisfactory solutions in a reasonable time.
There is no guarantee to find global optimal solutions or even bounded solutions.

In designing a metaheuristic, two contradictory criteria must be taken into account: *exploration* of the search space (*diversification*) and *exploitation* of the best solutions found (*intensification*) (Fig. 1.9). Promising regions are determined by the obtained "good" solutions.
In intensification, the promising regions are explored more thoroughly in the hope to find better solutions.
In diversification, non explored regions must be visited to be sure that all regions of the search space are evenly explored and that the search is not confined to only a reduced number of regions.

In this design space, the extreme search algorithms in terms of the exploration (resp. exploitation) are random search (resp. iterative improvement local search). In random search, at each iteration, one generates a random solution in the search space. No search memory is used. In the basic steepest local search algorithm, at each iteration one selects the best neighboring solution that improves the current solution.

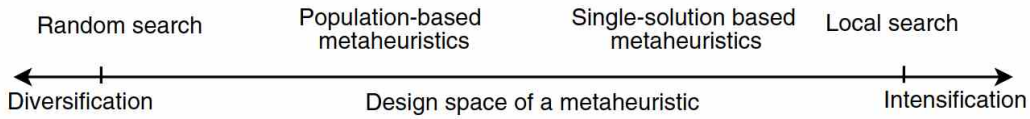


FIGURE 1.9 Two conflicting criteria in designing a metaheuristic: exploration (diversification) versus exploitation (intensification). In general, basic single-solution based metaheuristics are more exploitation oriented whereas basic population-based metaheuristics are more exploration oriented.

3. Metaheuristics methods classification

Many classification criteria may be used for metaheuristics:

• **Nature inspired versus non nature 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 be obtained 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.

• **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.

4. Main Common Concepts for Metaheuristics

There are two common design questions related to all iterative metaheuristics: the representation of solutions handled by algorithms and the definition of the objective function that will guide the search.

1) Representation

Designing any iterative metaheuristic needs an encoding (representation) of a solution. It is a fundamental design question in the development of metaheuristics. The encoding plays a major role in the efficiency and effectiveness of any metaheuristic and constitutes an essential step in designing a metaheuristic. The encoding must be suitable and relevant to the tackled optimization problem. Moreover, the efficiency of a representation is also related to the search operators applied on this representation (neighborhood, recombination, etc.). In fact, when defining a representation, one has to bear in mind how the solution will be evaluated and how the search operators will operate.

Many alternative representations may exist for a given problem. A representation must have the following characteristics:

Many straightforward encodings may be applied for some traditional families of optimization problems (Fig. 1.16). There are some classical representations that

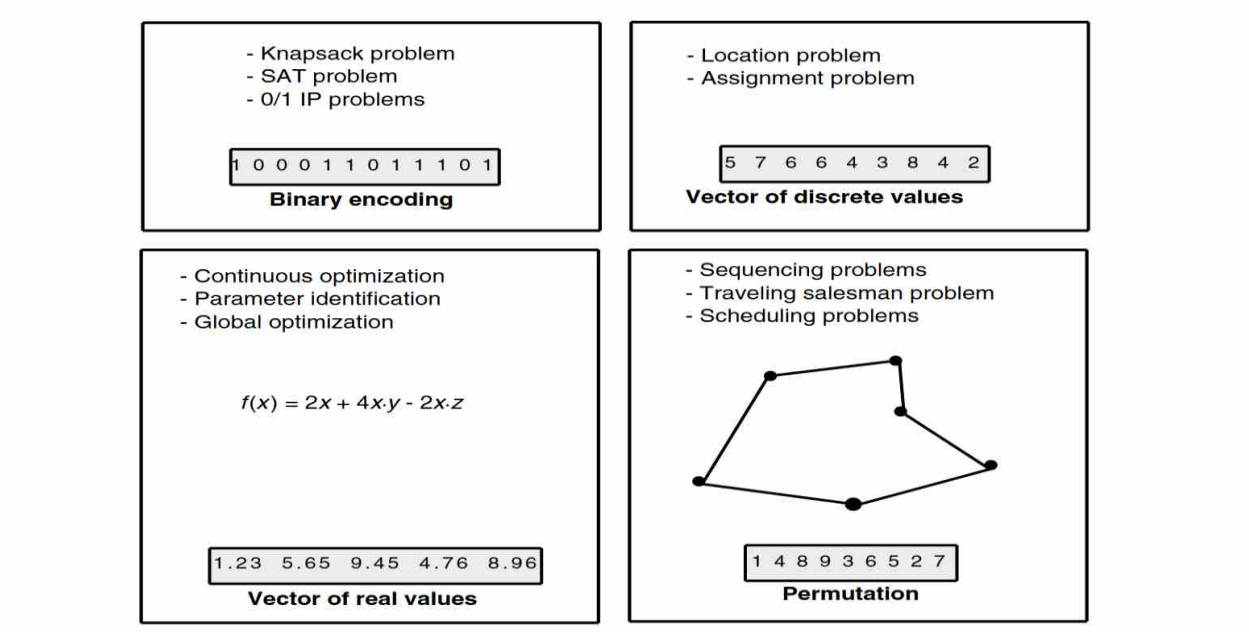


FIGURE 1.16 Some classical encodings: vector of binary values, vector of discrete values, vector of real values, and permutation.

2) Objective Function

The objective function formulates the goal to achieve. It associates with each solution of the search space a real value that describes the quality or the fitness of the solution, f : S → R. Then, it represents an absolute value and allows a complete ordering of all solutions of the search space.
The objective function is an important element in designing a metaheuristic. It will guide the search toward "good" solutions of the search space. If the objective function is improperly defined, it can lead to nonacceptable solutions whatever metaheuristic is used.

3) Constraint handling:

Dealing with constraints in optimization problems is another important aspect of the efficient design of metaheuristics. Indeed, many continuous and discrete optimization problems are constrained, and it is not trivial to deal with those constraints. Most of the constraint handling strategies act on the representation of solutions or the objective function (e.g., reject, penalizing, repairing, decoding, and preserving strategies).

Example :Encoding

Let us consider an encoding for real numbers based on binary vectors. Let us consider two consecutive integers, 15 and 16. Their binary representation is, respectively, 01111 and 10000.

Example objective function:

given a function F of the propositional calculus in a conjunctive normal form (CNF). The function F is composed of m clauses Ci of k Boolean variables, where each clause Ci is a disjunction. The objective of the problem is to find an assignment of the k Boolean variables such as the value of the function F is true. Hence, all clauses must be satisfied.

$$F = (x_1 \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\overline{x_1} \vee x_2) \wedge (x_1 \vee x_2 \vee x_4)$$
$$\wedge (x_2 \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3})$$

A solution for the problem may be represented by a vector of k binary variables.
A straightforward objective function is to use the original F function:
f = {0 if is F false
1 otherwise

If one considers the two solutions

s1 = (1, 0, 1, 1) and
s2 = (1, 1, 1, 1), they will have the same objective function, that is, the 0 value, given that the function F is equal to false. The drawback of this objective function is that it has a poor differentiation between solutions.

A more interesting objective function to solve the problem will be to count the number of satisfied clauses. Hence, the objective will be to maximize the number of satisfied clauses. This function is better in terms of guiding the search toward the optimal solution.
In this case, the solution s1 (resp. s2) will have a value of 5 (resp. 6).

Reference:

Talbi, El-Ghazali, " Metaheuristics : from design to implementation ", Copyright ©2009 by John Wiley & Sons, Inc. Published by John Wiley & Sons, Inc., Hoboken, New Jersey Published simultaneously in Canada.

1.Single Solution Metaheuristics

While solving optimization problems, single-solution based metaheuristics (S-metaheuristics) improve a single solution. They could be viewed as "walks" through neighborhoods or search trajectories through the search space of the problem at hand. The walks (or trajectories) are performed by iterative procedures that move from the current solution to another one in the search space. S-metaheuristics show their efficiency in tackling various optimization problems in different domains.

1.1 Common Concepts

S-metaheuristics iteratively apply the generation and replacement procedures from the current single solution.

generation phase, a set of candidate solutions are generated from the current solution s. This set C(s) is generally obtained by local transformations of the solution.

replacement phase, a selection is performed from the candidate solution set C(s) to replace the current solution; that is, a solution $s' \in C(s)$ is selected to be the new solution. This process iterates until a given stopping criteria.

The generation and the replacement phases may be *memoryless*. In this case, the two procedures are based only on the current solution. Otherwise, some history of the search stored in a memory can be used in the generation of the candidate list of solutions and the selection of the new solution. Algorithm 2.1 illustrates the high-level template of S-metaheuristics.

Algorithm 2.1 High-level template of S-metaheuristics.

Input: Initial solution s_0.
$t = 0$;
Repeat
  /* Generate candidate solutions (partial or complete neighborhood) from s_t */
  Generate($C(s_t)$) ;
  /* Select a solution from $C(s)$ to replace the current solution s_t */
  s_{t+1} = Select($C(s_t)$) ;
  $t = t + 1$;
Until Stopping criteria satisfied
Output: Best solution found.

a) Initial Solution

Two main strategies are used to generate the initial solution:
a *random approach* and
a *greedy* approach.

Generating a random initial solution is a quick operation, but the metaheuristic may take much larger number of iterations to converge. To speed up the search, a greedy heuristic may be used. Indeed, in most of the cases, greedy algorithms have a reduced polynomial-time complexity.
Using greedy heuristics often leads to better quality local optima. Hence, the S-metaheuristic will require, in general, less iterations to converge toward a local optimum. Some approximation greedy algorithms may also be used to obtain a bound guarantee for the final solution. However, it does not mean that using better solutions as initial solutions will always lead to better local optima.

b) Incremental Evaluation of the Neighborhood

A naive exploration of the neighborhood of a solution s is a *complete* evaluation of the objective function for every candidate neighbor s′ of N(s).
A more efficient way to evaluate the set of candidates is the *evaluation* $\Delta(s, m)$ of the objective function when it is possible to compute, where s is the current solution and m is the applied move.
This is an important issue in terms of efficiency and must be taken into account in the design of an S-metaheuristic. It consists in evaluating only the transformation $\Delta(s, m)$ applied to a solution s rather than the complete evaluation of the neighbor solution $f(s') = f(s \oplus m)$. The definition of such an incremental evaluation and its complexity depends on the neighborhood used over the target optimization problem. It is a straightforward task for some problems and neighborhoods but may be very difficult for other problems and/or neighborhood structures.

c) Fitness Landscape Analysis

The main point of interest in the domain of optimization must not be the design of the best algorithm for *all* optimization problems but the search for the most adapted algorithm to a given class of problems and/or instances. No metaheuristic can be uniformly better than any other metaheuristic. The question of superiority of a given algorithm has a sense only in solving a given class of problems and/or instances.

Definition Search space. *The search space is defined by a directed graph* G = (S, E)*, where the set of vertices* S *corresponds to the solutions of the problem that are defined by the representation (encoding) used to solve the problem, and the set of*

edges E *corresponds to the move operators used to generate new solutions (neighborhood in S-metaheuristics).*

Definition Fitness landscape. *The fitness landscape may be defined by the tuple* (G, f)*, where the graph* G *represents the search space and* f *represents the objective function that guides the search.*

d) stopping criteria may be used: time to obtain a given target solution, time to obtain a solution within a given percentage from a given solution (e.g., global optimal, lower bound, best known), number of iterations, and so on.

2. Single solution Metaheuristic Basic Methods

In addition to the representation, the objective function and constraint handling that are common search concepts to all metaheuristics, the common concepts for single-solution based metaheuristics are:
• **Initial solution:** An initial solution may be specified randomly or by a given heuristic.
• **Neighborhood:** The main concept of S-metaheuristics is the definition of the neighborhood. The neighborhood has an important impact on the performances of this class of metaheuristics. The interdependency between representation and neighborhood must not be neglected. The main design question in S-metaheuristics is the trade-off between the efficiency of the representation/neighborhood and its effectiveness (e.g., small versus large neighborhoods).
• **Incremental evaluation of the neighborhood:** This is an important issue for the efficiency aspect of an S-metaheuristic.
• **Stopping criteria.**

Hence, most of the search components will be reused by different single- solution based metaheuristics (Fig. 2.43). Moreover, an incremental design and implementation of different S-metaheuristics can be carried out. In addition to the common search concepts of S-metaheuristics, the following main search components have to be defined for designing the following S-metaheuristics:
• **Local search:** Neighbor selection strategy.
• **Tabu search:** Tabu list, aspiration criteria, medium- and long-term memories.
• **Simulated annealing, threshold accepting:** Annealing schedule.
• **Iterated local search:** Perturbation method, acceptance criteria.
• **Variable neighborhood search:** Neighborhoods for shaking and neighborhoods for local search.

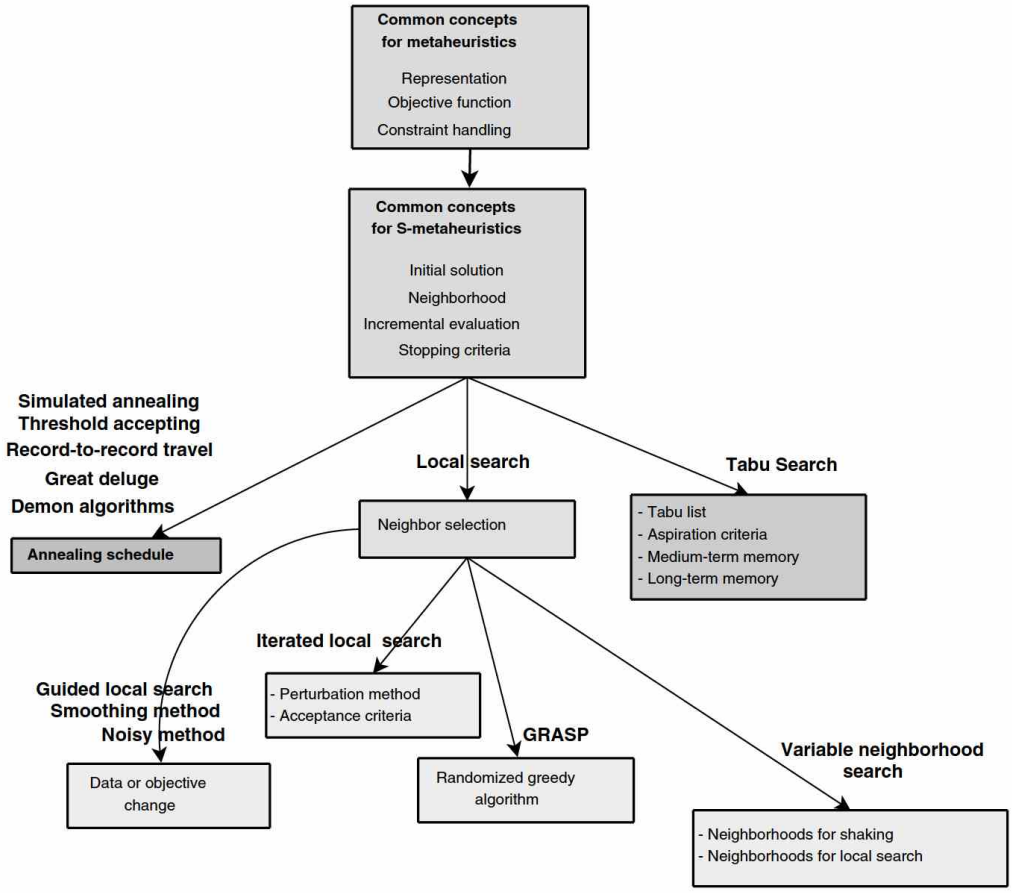


FIGURE 2.43 Common concepts and relationships in S-metaheuristics.

Reference:

Talbi, El-Ghazali, " Metaheuristics : from design to implementation ", Copyright ©2009 by John Wiley & Sons, Inc. Published by John Wiley & Sons, Inc., Hoboken, New Jersey Published simultaneously in Canada.

1. Local search

Local search is likely the oldest and simplest metaheuristic method. It starts at a given initial solution. At each iteration, the heuristic replaces the current solution by a neighbor that improves the objective function.
The search stops when all candidate neighbors are worse than the current solution, meaning a local optimum is reached.

A Local Search Algorithm.

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' e N(s) */
End while
Output Final solution found (local optima).

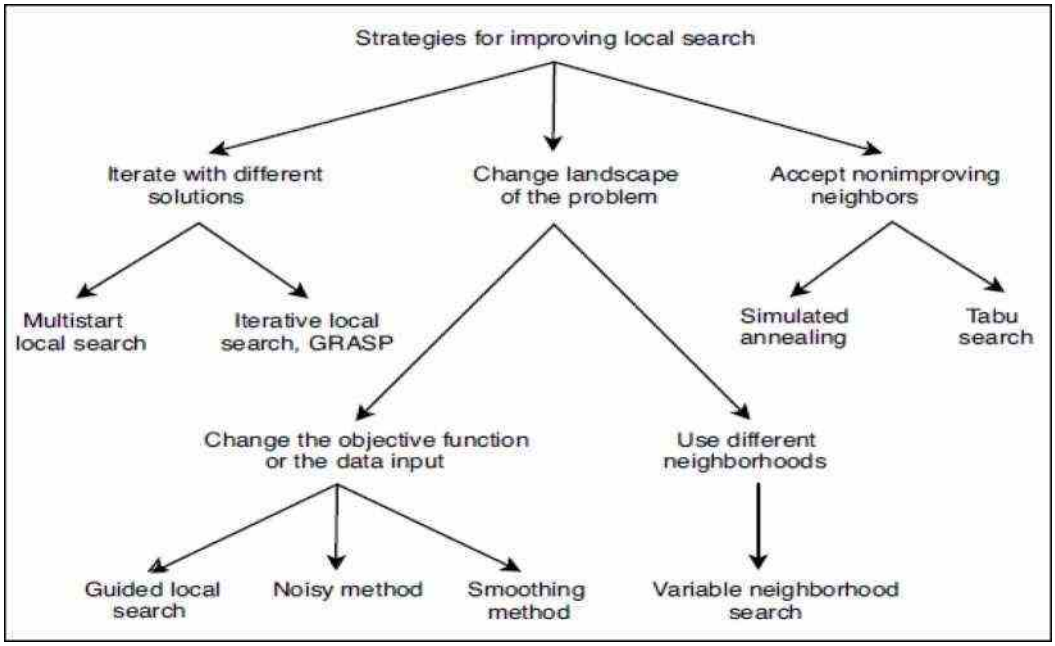
Selection of the Neighbor

Many strategies can be applied in the selection of a better neighbor:

- **Best improvement (steepest descent):** In this strategy, the best neighbor (i.e., neighbor that improves the most the cost function) is selected. The neighborhood is evaluated in a fully deterministic manner. Hence, the exploration of the neighborhood is *exhaustive*, and all possible moves are tried for a solution to select the best neighboring solution.

- **First improvement:** This strategy consists in choosing the first improving neighbor that is better than the current solution. Then, an improving neighbor is immediately selected to replace the current solution. This strategy involves a partial evaluation of the neighborhood.

- **Random selection:** In this strategy, a random selection is applied to those neighbors improving the current solution.



Local search family of algorithms for the improvement of basic local search and escaping from local optima.

2 TABU SEARCH

Tabu Search (TS) was developed by Fred Glover in the 1970s 23 and employs two types of lists: **tabu lists** and **aspiration lists**. TS also allows backward jumps; tabu lists are present to prevent the search from revisiting previous points in the searched as Figure 12.29.

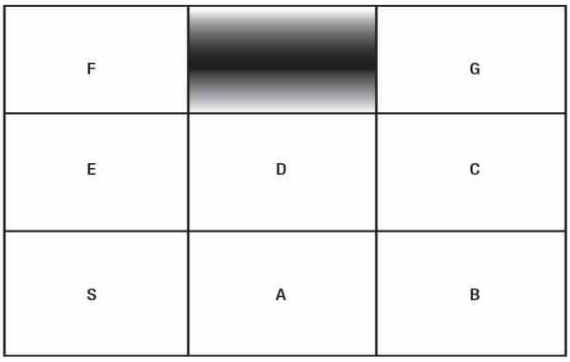


Figure 12.29
A robot must get from square S (Start) to G (Goal).

solution to this problem by a
sequence of size =four, over the alphabet ={“N,” “S,” “W,” “E”}
where the alphabet symbols represent a move of one square in the directions:

N= north,
S=south,
W= west, or
E= east.
start the search with a random feasible solution , $x0$ = ENWS.
fitness function: $f(xi) = 4 -$ Manhattan distance to the goal after the move contained in sample point *xi* is executed.

TS terminology refers to this function $f()$ as an objective function, and solutions in TS need not necessarily be strings; hence, we use *xi* rather than *si*. In our example, $x0$ takes the robot one square east, then one square north, then west one and south one, leaving it where it started, in square S.
The objective function $f(x0)$ therefore equals $4 - 4 = 0$.

TS uses both **short-term memory** and **long-term memory**. Short-term memory is incorporated into the search in terms of a recency based tabu list. States in the state space that have been recently visited cannot be revisited for a period of time referred to as the **tabu tenure**. Actually, it is the moves m that transform one point xi into another *xj* (where $xi + m = xj$) that are tabulated. This strategy encourages exploration. Long term memory is reflected in the use of aspiration criteria.
We mentioned earlier that one aspiration criteria is to visit x* even if forbidden by the tabu list, if $f(x^*)$ is superior to any previously visited point *xi*.

Other aspiration criteria include the following:

• *Aspiration by default*; if all moves are tabu, then select the oldest move.
• *Aspiration by direction* favors moves that have led to improved values of $f(x)$ in the past. This heuristic fosters exploitation.
• *Aspiration by influence* favors moves that lead to unexplored regions of the state space. This heuristic favors exploration.

Long-term memory also includes a frequency-based tabu list; this list monitors how often each move has been used since the search began.

We stated that $x0$ = ENWS and that $f(x0) = 0$. We let a move correspond to the alteration of a single step. When selecting moves, we need to ensure that a path exists from $x0$ to an optimal solution.

There is no concern with this simple problem, but this latter proviso cannot be ignored when more realistic problems are encountered. We observe that there are 44, or 256, points in the state space of this problem; many of these points correspond to infeasible solutions (take the robot off the grid).

The neighborhood of a sample point *xj* : N(*xj*), corresponds to all points reachable from *xj* via one move. More accurately, we should refer to the neighborhood of *xj* at time k or N(*xj*, *k*), because the neighborhood changes as the search progresses (and various tabu and aspiration criteria are modified). It is no surprise that memory usage can become a concern for TS on moderate to large problems. The neighborhood of *x*0 at time 0 (the search just beginning), N(*x*0, 0) contains 12 additional sample points (i.e., in addition to *x*0 itself). To see this, just observe that any one of the four directional steps can be changed to any of the three remaining directional steps. We comment that some of these 12 sample points are not feasible, for example, ENNS attempts to enter the barrier between squares F and G. Any move that is made is reflected in a recency-based tabu list (RTL).

Initially, this list will have the following format:

1	2	3	4	RTL
0	0	0	0	

RTL(*i*) = *j* indicates that step *i* of a sample point was last modified at time *j*. Observe that the list is initialized to all zeroes as no moves have yet been made.

Suppose that at time 1 we choose x1, which belongs to N(ENWS, 0) equal to ENWN. Note that *f* (*x*1) = *f* (ENWN) = 2 as ENWN leaves the robot in square D, which is a Manhattan distance of two from G. The recency-based tabu list now equals:

1	2	3	4	
0	0	0	1	RTL

The “1” in RTL (4) reflects that this step was last modified at time 1. Any move that takes place will remain tabu for k time units, in other words, this move cannot be made again until sufficient time has elapsed. This quantity *k* is referred to as the tabu tenure and must be specified. We shall let *k* = 3, hence step 4 cannot be modified

again until three time periods have elapsed, in other words, until time 4. What can occur if tabu tenure is set too high, say $k = 4$ or 5, in this example?
At time 2, we modify $x1$ = ENWN to $x2$ = EEWN as no other move brings us closer to square G.

We have modified the second step from N to E. Observe that EEWN also brings the robot to square D, hence $f(x2)$ still equals 2. RTL appears as:

1	2	3	4	RTL
0	2	0	1	

At time 3, we observe that steps 2 and 4 cannot yet be modified (they are tabu). By converting step 3 from W to N we obtain $x3$ = EENN. Our final tabu list equals:

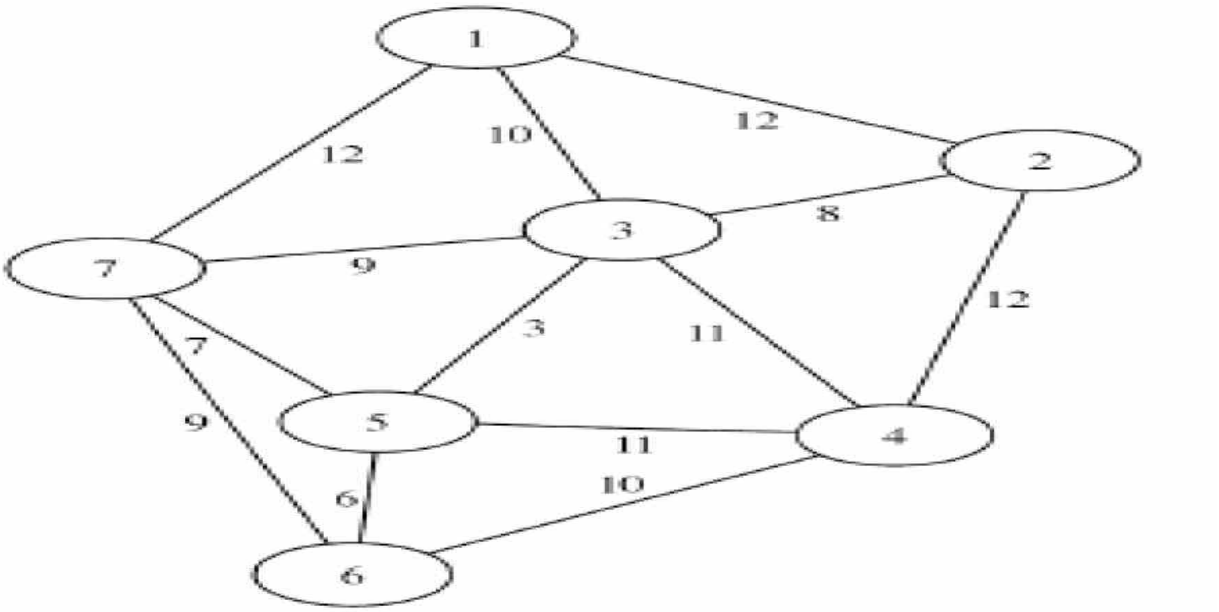
1	2	3	4	RTL
0	2	3	1	

More importantly, however, the fitness of this proposed solution, $f(x3) = 4 - 0 = 4$, and therefore the problem has been solved as EENN sends the robot to square G. It is difficult to construct a toy problem for TS that uses frequency-based tabu lists and aspiration.

Algorithm tabu search

1. Randomly choose an initial solution x0. // A Greedy method can also sometimes be used to get started.
2. Calculate f(x0) // Objective function.
3. Initialize tabu list // Fill in RTL with all 0's.
4. Count = 0
5. while **Count** < **maxcount** and **progress being made** and **ideal solution not found.**
6. Count = Count + 1
7. Choose xt in N(x, t) - (tabu elements) // Observe that the neighborhood changes with time
8. Calculate f(xt)
9. Update the tabu list RTL
10. // end while /*Output the last solution xt and indicate whether this represents an ideal or approximate solution. */

Using the Tabu search algorithm to solve the Travelling Salesman problem



Let initial trial solution = 1-2-3-4-5-6-7-1 Distance = 69 Tabu list : Blank at this point
Iteration 1: reverse 3-4
Delete Links: 2-3 and 4-5 Added links: 2-4 and 3-5 Tabu list : Links 2-4 and 3-5
New trial solution: 1-2-4-3-5-6-7-1 Distance = 65

Iteration 2
Reverse 3-5-6
Delete links: 4-3 and 6-7
Added links: 4-5 and 3-7
Tabu list: links 2-4, 3-5, 4-6 and 3-7
New trial solution: 1-2-4-6-5-3-7-1 Distance = 64
The tabu search algorithm now escapes from this local optimum by moving next to the best immediate neighbor of the current trial solution even though its distance is longer. Considering the limited availability of links between pairs of cities in figure, the current trial solution has only the two immediate neighbours listed below.
Reverse 6-5-3: 1-2-4-3-5-6-7-1 Distance = 65
Reverse 3-7: 1-2-4-6-5-7-3-1 Distance = 66
Reversing 2-4-6-5-3-7 to obtain 1-7-3-5-6-4-2-1 is ruled out since it is simply the same tour in the opposite direction. However the of these immediate neighbours must be ruled out because it would require deleting links 4-6 and 3-7, which is tabu since both of these links are on the tabu list. This move could still be allowed if it would improve upon the best trial solution found so far but it does not.

The tabu search algorithm now escapes from this local optimum by moving ne be ruled out because it would require deleting links 4-6 and 3-7, which is tabu since both of these links are on the tabu list. This move could still be allowed if it would improve upon the best trial solution found so far but it does not. Ruling out this immediate neighbor does not allow cycling back to the preceding trial solution. Therefore by default, the second of these immediate neighbours is chosen to be the next trial solution as summarized below.

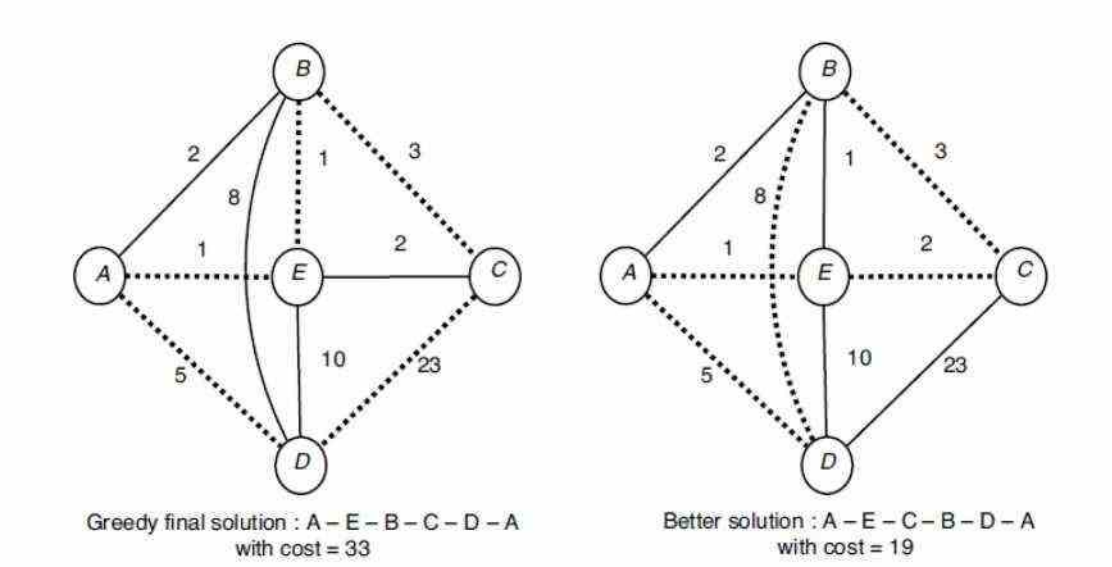
teration 3
Reverse 3-7
Delete links: 5-3 and 7-1
Add links: 5-7 and 3-1
Tabu List: 4-5, 3-7, 5-7 and 3-1
New trial solution: 1-2-4-6-5-7-3-1 Distance = 66

The new trial solution has the four immediate neighbours listed below.
Reverse 2-4-6-5-6: 1-7-5-6-4-2-3-1 Distance = 65
Reverse 6-5: 1-2-4-5-6-7-3-1 Distance = 69
Reverse 5-7: 1-2-4-6-5-7-3-1 Distance = 63
Reverse 7-3: 1-2-4-6-5-3-7-1 Distance

Both of the deleted links 4-6 and 5-7 are on the tabu list. The second of these immediate neighbours is therefore tabu. The fourth immediate neighbor is also tabu.Thus, there are only two options, the first and the third immediate neighbours. The third immediate neighbor is chosen since it has shorter distance.
Iteration 4
Reverse 5-7
Delete links: 6-5 and 7-3 Add links: 6-7 and 5-3
Tabu list: 5-7, 3-1, 6-7 and 5-3
(4-6 and 3-7 are now deleted from the list)
New trial solution: 1-2-4-6-7-5-3-1 Distance = 63

The only immediate neighbor of the current trial solution would require deleting links 6-7 and 5-3, both of which are on the tabu list so cycling back to the preceding trial solution is prevented. Since no other immediate neighbours are available, the stopping rule terminates the algorithm at this point with 1-2-4-6-7-5-3-1 as the final solution with Distance = 63.

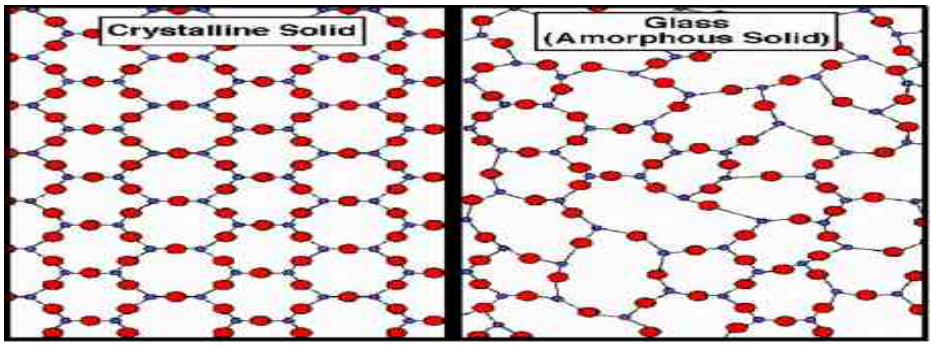
Example:



3. SIMULATED ANNEALING

Simulated annealing (SA) capitalizes on the analogy between the energy level of the molecules within a physical substance and a search algorithm in which some objective function is to be optimized.

In metallurgy, metals are often subjected to molecular realignment in a process known as annealing. The molecules in a metal are arranged in a local energy minimum. In order to rearrange these molecules at a lower energy, it is first necessary to heat the metal until it liquefies. The molten metal is then slowly cooled until it solidifies; annealed metals exhibit many desirable properties, for instance, they are stronger and often more pliable.



There are two components to any search algorithm: **exploitation** and **exploration**.

Exploitation employs the maxim that good solutions are likely to lie close to one another. Once a good solution is found, you examine its neighbors to determine if a better solution is present.

Exploration, on the other hand, relies upon the adage, "Nothing ventured, nothing gained"; in other words, better solutions can lie in unexplored regions of the state space, so do not confine your search to one small region.

An ideal search algorithm must strike the proper balance between these two conflicting strategies. Hill climbing makes advantageous use of exploitation to find *x**, the *local optimum* in Figure 12.3.

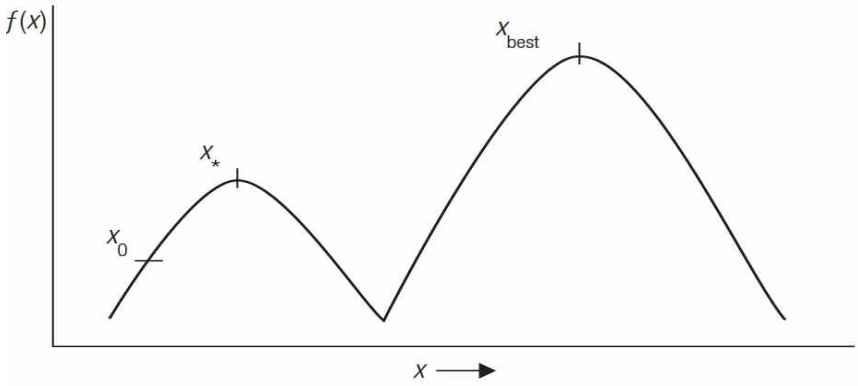


Figure 12.2
Hill climbing sometimes gets stuck in a local optimum. A search that begins at x_0 will get stuck at x*. Observe that $f(x_{best}) > f(x^*)$.

In this example, however, if the global maximum located at *x*best is to be found, then some use of exploration is required as well. Consult Figure 12.4 and assume that *x*3 is the present location.

In SA there is a global temperature parameter T. At the beginning of the simulation, T is high; as the simulation progresses, T is lowered. The manner in which T is decreased is referred to as the **cooling schedule**. Two widely used methods are **geometric cooling** and **linear cooling**.
In geometric cooling,

$T_{new} = \alpha * T_{old}$ with $\alpha < 1$,
whereas with linear cooling,
$T_{new} = T_{old} - \alpha$ with $\alpha > 0$.

Whenever $f(x_{new}) > f(x_{old})$ SA will allow this jump. However an SA also permits counter intuitive or backward jumps with a probability P, which is proportional to

$$e^{-[(f(x_{old}) - f(x_{new}))/T]}$$

Observe that when T is high, jumps that result in a lower objective function will occur with a greater probability. Consulting Figure 12.4 once again, this means that a jump from *x*3 to *x*6 is more likely to occur at the beginning of the simulation, when T is much higher, rather than later. Hence, the early stages of SA favor exploration, whereas exploitation is preferred in later stages of the

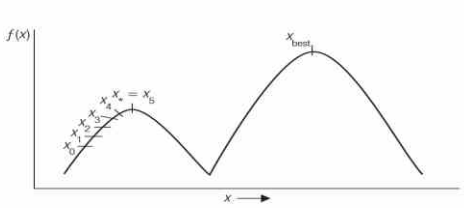


Figure 12.3
Hill climbing relies heavily upon exploitation.

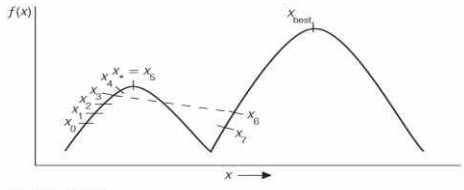


Figure 12.4
If a search is to find the global maximum at x_{best}, then it perhaps will have to use exploration as well as exploitation. Simulated annealing would allow a jump from x_3 to x_6 even though $f(x_6) < f(x_3)$. (Ignore x_7 for the present.)

search. Referring once again to the above equation, we observe that even though counterintuitive jumps are allowed, as the difference between $f(x_{old})$ and $f(x_{new})$ increases, that is, as the new value of x becomes less and less favorable, the probability of going there decreases. This last observation dictates that if each of $x6$ and $x7$ in Figure 12.4 are possible successors to $x3$, the probability of going to $x6$ is greater than to $x7$ as $f(x7)$ is less than $f(x6)$. Pseudocode for SA is provided in Figure 12.5.

Algorithm simulated annealing

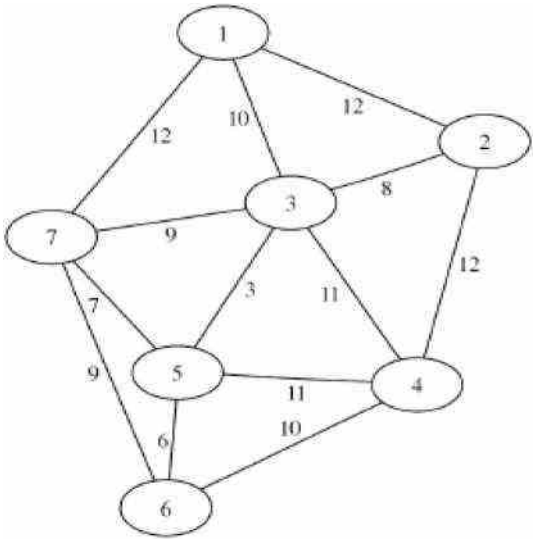
1. choose x_0 as initial solution// usually done randomly
2. calculate $f(x_0)$ // objective function
3. place in memory // solution =$[x_0,f(x_0)]$
4. $x_{old}=x_0$,
5. $f(x_{old})=f(x_0)$,
6. count=0,
7. $T=T_0$ // initial temperature t0 is high,
8. while count< **maxcount** and **progress being made** high and **ideal solution not fou**nd // number of iteration permitted,
9. count=count+1,
10. choose x_{new} from neighborhood of x_{old},
11. calculate $f(x_{new})$,
12. if $f(x_{new})=f(x_{old})$ or rand[0,1]= $e^{-[(f(x_{old})- f(x_{new}))/T]}$ then $x_{old}=x_{new}$, solution=[x_{old}, $f(x_{old})$
13. endif,
14. Tnew=cooling_schedule(count,T_{old})//geometric or linear cooling can be adaptive. greater decrease if a large improvement is made,
15. end while ,
16. print solution // best solution so far

General schema for a simulated annealing algorithm.

a. Generate a starting solution S and set the initial solution $S^* = S$.
b. Determine a starting temperature T.
c. While not yet at *equilibrium* for this temperature, do the following:
d. Choose a *random neighbor* S^* of the current solution.
e. Set $\Delta = Length(S^*) - Length(S)$.
f. If ≤ 0 (downhill move):
Set $S = S^*$.
If $Length(S) < Length(S^*)$, set $S^* = S$.

h. If length(S) < length(S*) (uphill move): Choose a random number *r* uniformly from [0, 1]. If $r < e^{-\Delta/T}$, set $S = S^*$.
i. End __While not yet at equilibrium‘‘ loop.
j Lower the temperature *T*. k. End __While not yet frozen‘‘ loop.
l. Return *S* *.

EXAMPLE :Using the simulated annealing algorithm to solve the Travelling



Taking the initial solution to be in the tour in the order :1-2-3-4-5-6-7-1
using the parameters;
$T0 = 20$ $Tk+1 = aTk$ $a = 0.5$
Stop when $T < 0.1$

First Iteration
Assuming $x0$ =1-2-3-4-5-6-7-1
d($x0$)=d(1,2)+d(2,3)+d(3,4)+d(4,5)+d(5,6)+d(6,7)+d(7,1)=69

Using the sub-tour reversal as local search to generate the new solution $x1$ =1-3-2-4-5-6-7-1
d($x1$)=d(1,3)+d(3,2)+d(2,4)+d(4,5)+d(5,6)+d(6,7)+d(7,1)=68
$d = d(x1) - d(x0) = 68-69 = -1$
Since $d < 0$, set $x0 = x1$
Updating the temperature $T1 = aT0 = 0.5(20) = 10$

Second Iteration d($x0$)=68

By the sub-tour reversal as local search to generate the new solution 1-2-3-5-4-6-7-1

$x1$ =1-2-3-5-4-6-7-1

d($x1$)=d(1,2)+d(2,3)+d(3,5)+d(5,4)+d(4,6)+d(6,7)+d(7,1)=65

$\Delta = d(x1) - d(x0)$ =65-68=-3 Since $\Delta < 0$, set $x0 = x1$

Updating the temperature , $T2$ =0.5(10)=5

Third Iteration d($x0$)=65

Using the sub-tour reversal as local search to generate the new solution 1-2-3-4-6-5-7-1

$x1$ =1-2-3-4-6-5-7-1

d($x1$)=d(1,2)+d(2,3)+d(3,4)+d(4,6)+d(6,5)+d(5,7)+d(7,1)=66

$\Delta = d(x1) - d(x0)$ =66-65=1 Since $d > 0$, then apply Boltzmann's condition $m\ e^{-\Delta / T2}$ = 0.81

A random number would be generated from a computer say q

If m>q then set $x0 = x1$ otherwise $x1 = x0$ Updating the temperature, $T3$ = 0.5(5) = 2.5

This process will continue until the final temperature and the optimal solution are obtained.

4.Threshold Accepting **(TA)** TA escapes from local optima by accepting solutions that are not worse than the current solution by more than a given threshold Q.
The threshold parameter in TA operates somewhat like the temperature in simulated annealing. The threshold Q is updated following any annealing schedule.
The below Algorithm describes the template of the TA Algorithm.

Threshold Accepting Algorithm

Input: Threshold annealing.
s = s0; /∗ Generation of the initial solution ∗/
Q = Qmax; /∗ Starting threshold ∗/
Repeat
Repeat /∗ At a fixed threshold ∗/
Generate a random neighbor s' ∈ N(s);
DE = f (s') − f (s); If DE ≤ Q Then s = s' /∗ Accept the neighbor solution ∗/
Until Equilibrium condition /∗ e.g. a given number of iterations executed at each threshold Q ∗/
Q = g(Q); /∗ Threshold update ∗/
Until Stopping criteria satisfied /∗ e.g. Q ≤ Qmin∗/
Output: Best solution found.

5.Variable Neighborhood Search (VNS)

The basic idea of VNS is to successively explore a set of predefined neighborhoods to provide a better solution. It explores either at random or
systematically a set of neighborhoods to get different local optima and to escape from local optima. VNS exploits the fact that using various neighborhoods in local search may generate different local optima and that the global optima is a local optima for a given neighborhood. Indeed, different neighborhoods generate different landscapes.

- **Variable Neighborhood Descent (VND)**

The VNS algorithm is based on the variable neighborhood descent, which is a deterministic version of VNS. VND uses successive neighborhoods in descent to a local optimum. First, one has to define a set of neighborhood structures Nl (l =1, ... , lmax). Let N1 be the first neighborhood to use and x the initial solution. If an improvement of the solution x in its current neighborhood Nl(x) is not possible, the neighborhood structure is changed from Nl to Nl+1. If an improvement of the current solution x is found, the neighborhood structure returns to the first one N1(x) to restart the search. The below Algorithm shows the VND algorithm.

Variable Neighborhood Descent Algorithm

Input: a set of neighborhood structures Nl for l = 1, . . . , lmax. x = x0 ; /* Generate the initial solution */ l = 1 ;
While l < lmax **Do**
Find the best neighbor x‘ of x in $N_l(x)$;
If f (x‘) < f (x) **Then** x = x‘;
Otherwise l = l + 1 ;
Output: Best found solution.

- **General Variable Neighborhood Search**

VNS is a stochastic algorithm in which, first, a set of neighborhood structures Nk (k = 1, . . . , n) are defined. Then, each iteration of the algorithm is composed of three steps: **shaking**, **local search**, and **move**. At each iteration, an initial solution is shaked from the current neighborhood N_k. For instance, a solution x’ is generated randomly in the current neighborhood Nk(x). A local search procedure is applied to the solution x‘ to generate the solution x’’. The current solution is replaced by the new local optima x’’ if and only if a better solution has been found (i.e., f(x’’) < f(x)). The same search procedure is thus restarted from the solution x’’ in the first neighborhood N1. If no better solution is found (i.e., f (x’’) > f (x)), the algorithm moves to the next neighborhood N_{k+1}, randomly generates a new solution in this neighborhood, and attempts to improve it. Let us notice that cycling is possible (i.e., x’’ = x). The below Algorithm shows the VNS algorithm.

Variable Neighborhood Search Algorithm

Input: a set of neighborhood structures Nk for k = 1, . . . , kmax for shaking. a set of neighborhood structures Nl for l = 1, . . . , lmax for local search.
x = x0; /* Generate the initial solution */
Repeat
For k=1 **To** kmax **Do**
Shaking: pick a random solution x’ from the k^{th} neighborhood Nk(x) of x;
Local search by VND;
For l=1 **To** lmax **Do**
Find the best neighbor x’’ of x’ in Nl(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 N1 (k = 1);
Otherwise k=k+1;

Until Stopping criteria
Output: Best found solution.

Example:
A traveling salesman needs to visit 5 cities (A, B, C, D, E) with the shortest possible route and return to the starting point. Distance matrix:

CITY	A	B	C	D	E
A	-	10	15	20	25
B	10	-	35	25	30
C	15	35	-	30	20
D	20	25	30	-	15
E	25	30	20	15	-

Iteration 1:
• Initial Tour: A-B-C-D-E (Distance: 125)
• Shaking: Swap B and D (Tour: A-D-C-B-E)
• Local Search (2-opt): No improvement (Tour remains A-D-C-B-E)
• Acceptance: Accepted due to initial temperature
• Neighbourhood Change: No change
Iteration 2:
• Shaking: Swap C and E (Tour: A-D-C-E-B)
• Local Search (2-opt): Swaps B and C (Tour: A-D-E-C-B)
• Acceptance: Accepted due to improved distance (115)
• Neighbourhood Change: No change
Iteration 3:
• Shaking: Swap A and E (Tour: E-D-C-B-A)
• Local Search (2-opt): No improvement (Tour remains E-D-C-B-A)
• Acceptance: Rejected due to worse distance
• Neighbourhood Change: Change to insertion neighbourhood
Iteration 4:
• Insertion: Insert C after A (Tour: A-C-D-B-E)
• Local Search (2-opt): Swaps C and D (Tour: A-C-E-B-D)
• Acceptance: Accepted due to improved distance (110)
• Neighbourhood Change: No change
...
(Iterations continue with shaking, local search, acceptance, and neighbourhood changes depending on improvement and temperature until a stopping criterion is

met.)
Final Solution: After several iterations, the algorithm might converge to a final solution like: A-C-E-D-B (Distance: 100).

6. Greedy Randomized Adaptive Search Procedure (GRASP)

The GRASP metaheuristic is an iterative greedy heuristic to solve an optimization problem. Each iteration of the GRASP algorithm contains two steps: construction and local search. In the construction step, a feasible solution is built using a randomized greedy algorithm, while in the next step a local search heuristic is applied from the constructed solution. The greedy algorithm must be randomized to be able to generate various solutions. Otherwise, the local search procedure can be applied only once. This schema is repeated until a given number of iterations and the best found solution are kept as the final result. So there is no search memory. The below Algorithm resumes the template for the GRASP algorithm. The seed is used as the initial seed for the pseudorandom number generator.

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.

The main design questions for GRASP are the greedy construction and the local search procedures:

- **Greedy construction:** at each iteration the elements that can be included in the partial solution are ordered in the list using the local heuristic. From this list, a subset is generated that represents the *restricted candidate list (RCL)*. The RCL list is made of the *p* best elements in terms of the incremental cost, where the parameter *p* represents the maximum number of elements in the list.

  At each iteration, a random element is picked from the list RCL. Once an element is incorporated in the partial solution, the RCL list is updated. To update the RCL list, the incremental costs c'(e) of the elements e composing the RCL list must be reevaluated. The below algorithm shows the template of the randomized part of the GRASP metaheuristic.
- **Local search**: the solutions found by the construction procedure are not guaranteed to be local optima, it is beneficial to carry out a local search step

in which the constructed solution is improved.

The Greedy Randomized Algorithm

```
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 */
ei = Random-Selection(RCL) ;
If s U ei e F Then /* Test the feasibility of the solution */ s = s U ei ;
Reevaluate the incremental costs of candidate elements ;
Until Complete solution found.
```

Example

Iteration 1:
- Start: Pick a random city (e.g., A).
- Construction Phase:
  - Randomly select a subset (e.g., B, C).
  - Choose next city using cost-to-benefit ratio (e.g., C due to closer distance).
  - Continue until all visited (A-C-D-B-E).
- Local Search (2-opt): Swaps D and E (A-C-E-D-B).
- Adaptive Phase: Update cost-to-benefit based on current solution.

Iteration 2:
- Repeat steps above with different random starting city and selections.

...

)Iterations continue with construction, local search, and adaptation phases, updating the best solution found so far(.

Final Solution: After a fixed number of iterations, the algorithm chooses the best solution found across all iterations (e.g., A-C-E-D-B with distance 100).

Note:

This is a simplified example, and specific details like cost-to-benefit ratio calculation and adaptive strategies can vary depending on GRASP implementation. The actual trace will reflect the chosen parameters and random selections.

7. Smoothing Methods

Other single based solution metaheuristics. some existing S-metaheuristics use other strategies to escape from local optima.

Search space smoothing and noisy methods are based on the transformation of the landscape of the problem by changing the input data associated with the problem altering the objective function.

Search space smoothing consists in modifying the landscape of the target optimization problem. The smoothing of the landscape associated with the problem

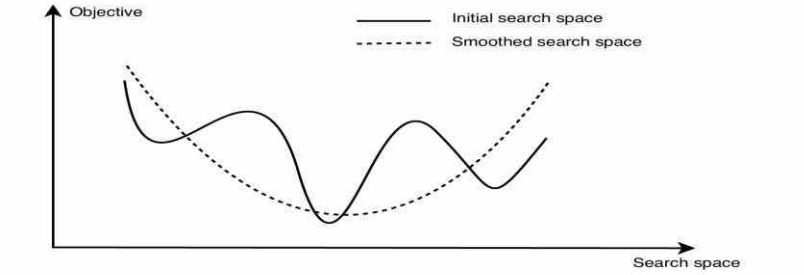


FIGURE 2.33 Search space smoothing that results in an easiest problem to solve. The smoothed landscape has less local optima than the original one.

reduces the number of local optima and the depth of the basins of attraction without changing the location region of the global optimum of the original optimization problem (Fig. 2.33).

The search space associated with the landscape remains unchanged, and only the objective function is modified. Once the landscape is smoothed by "hiding" some local optima, any S-metaheuristic (or even a P-metaheuristic) can be used in conjunction with the smoothing technique.

The main idea of the smoothing approach is the following:

- given a problem instance in a parameter space,
- the approach will transform the problem into a sequence of successive problem instances with different associated landscapes.
- Initially, the most simplified smoothed instance of the problem is solved. A local search is then applied.

The probability to be trapped by a local optima is minimized. In the ideal case, there is only one local optimum that corresponds to the global optimum (Fig. 2.34). The less the number of local optima, the more efficient a S-metaheuristic.

Then, a more complicated problem instance with a rougher landscape is generated. It takes the solution of the previously solved problem as an initial solution and further improves that solution. The solutions of smoothed landscapes are used to

guide the search in more rugged landscapes. Any S-metaheuristic can be used in conjunction with the smoothing operation. The last step of the approach consists in solving the original problem.

The main design question concerns the smoothing operation. There are many strategies to smooth a landscape. The smoothing factor α is used to characterize the strength of a smoothing operation. Using different levels of strength will generate various degrees of smoothness. When $\alpha = 1$, there is no smoothing operation, and the landscape is the same as the original one. A smoothing operation is carried out if $\alpha > 1$. The larger the smoothing factor ($\alpha >> 1$), the stronger a smoothing operation and more flat a landscape.

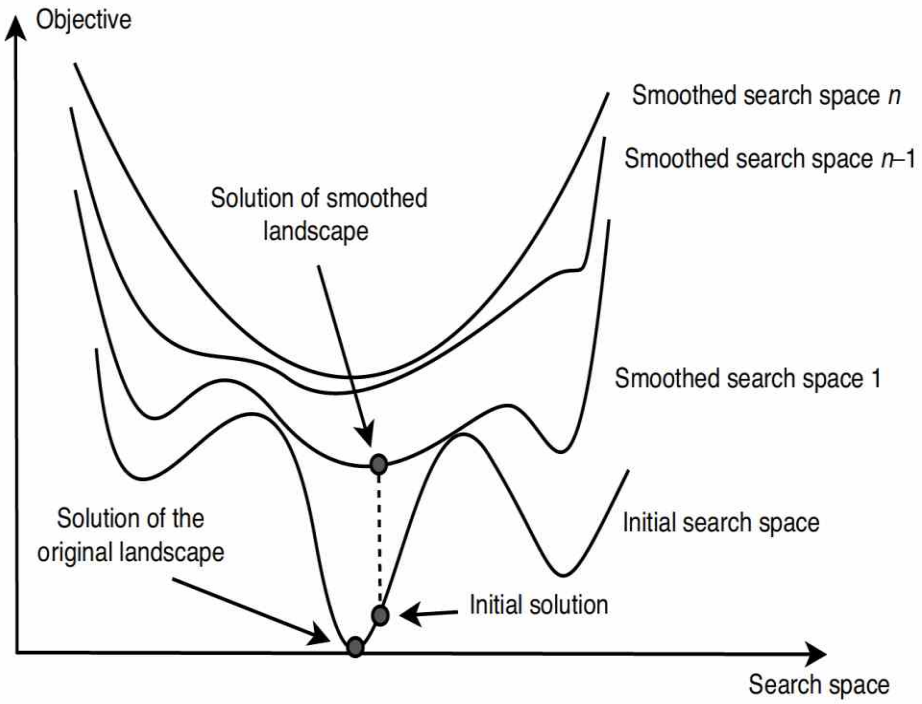


FIGURE 2.34 Successive smoothing of the landscape. The solution found at the step i will guide the search at the iteration $i + 1$ in a more rugged landscape.

The original idea of the algorithm relies on the reduced complexity of solving smoothing instances of the original problem and the effectiveness of using

intermediate local optima solutions to guide the search toward increasingly complex instances.

Algorithm Template of the smoothing algorithm.
Input: S-metaheuristic LS, α0, Instance I.
s = s0 ; /∗ Generation of the initial solution ∗/
α = α0 ; /∗ Initialization of the smoothing factor ∗/
Repeat
I = I(α) ; /∗ Smoothing operation of the instance I ∗/
s = LS(s, I) ; /∗ Search using the instance I and the initial solution s ∗/
α = g(α) ; /∗ Reduce the smoothing factor, e.g. α = α − 1 ∗/
Until α < 1 /∗ Original problem ∗/
Output: Best solution found.

Example Smoothing operation for the TSP.
The smoothing strategy has been applied successfully to many discrete optimization problems . In the case of the TSP, the smoothing operation is based on the fact that a trivial case for the TSP is the one where all the distances between cities are equal: dij = d', ∀i, j, where

$$d' = \frac{1}{n(n-1)} 1/n(n-1)\sum_{i \neq j} d\text{ij}$$

represents the average distance over all the edges. In this case, any tour represents a global optimum solution, and the landscape is flat.

The strength of a smoothing may be represented by the following equation:

$$d(\propto) = \begin{cases} d' + (d_{ij} + d')^{\propto} \; if \; d_{ij} \geq d' \\ d' - \left(d' - d_{ij}\right)^{\propto} \; if \; d_{ij} < d' \end{cases}$$

The main parameters of the smoothing algorithm are the appropriate choice of the initial value of the smoothing factor α and its controlling strategy. The larger the initial value of the smoothing factor α0, the more time consuming the algorithm.

Example simple implementation :

Let's assume:
- (α0 = 5)
- (s_0) is a solution vector ([x_1, x_2, x_3])
- (l(α)) modifies (I) by a factor of (α)
- (g(α)) reduces (α) by 1 each iteration
- (L(S, I)) improves the solution (s) based on (I)

Now, we can trace the algorithm:
1 Input: (L(S, 5)), (I)
2 Initialization: (s = [x_1, x_2, x_3]), (α = 5)
3 Iteration 1:
◦ (I) is modified by (l(5))
◦ (s) is improved to ([x'_1, x'_2, x'_3]) using (L(S([x_1, x_2, x_3], I))
◦ (α) is reduced to 4
4 Iteration 2:
◦ (I) is modified by (l(4))
◦ (s) is improved to ([x''_1, x''_2, x''_3]) using (L(S([x'_1, x'_2, x'_3], I))
◦ (α) is reduced to 3
5 Continue until (α < 1)

After the final iteration, we would output the best solution vector ([x^_1, x^_2, x^*_3]) found during the process.
This trace example demonstrates how the algorithm iteratively improves the solution by smoothing the instance and reducing the smoothing factor until the original problem conditions are met.

8.Noisy Method

The noisy method is another S-metaheuristic algorithm that is based on the landscape perturbation of the problem to solve. Instead of taking the original data into account directly, the NM considers that they are the outcomes of a series of fluctuating data converging toward the original ones.

Some random noise is added to the objective function f . At each iteration of the search, the noise is reduced. For instance, the noise is initially randomly chosen into an interval [−r, +r]. The range of the interval r decreases during the search process until a value of 0.
Different ways may be used to decrease the noise rate r.

2. Population-Based Metaheuristics (P-Metaheuristics)

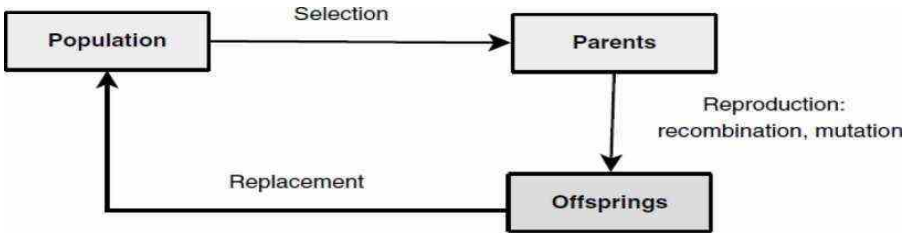
Population-based metaheuristics start from an initial population of solutions. Then, they iteratively apply the generation of a new population and the replacement of the current population. In the generation phase, a new population of solutions is created. In the replacement phase, a selection is carried out from the current and the new populations. This process iterates until a given stopping criteria.

The generation and the replacement phases may be *memoryless*. In this case, the two procedures are based only on the current population. Otherwise, some history of the search stored in a memory can be used in the generation of the new population and the replacement of the old population.

Most of the P- metaheuristics are nature-inspired algorithms. Popular examples of P- metaheuristics are evolutionary algorithms, ant colony optimization, scatter search, particle swarm optimization, bee colony, and artificial immune systems.

2.1 Evolutionary Algorithms

Evolutionary algorithms are stochastic P-metaheuristics that have been successfully applied to many real and complex problems. Initially, the population is usually generated randomly. Every individual in the population is the encoded version of a tentative solution. An objective function (a fitness value) associates with every individual indicating its suitability to the problem. At each step, individuals are selected to form the parents, following the selection paradigm in which individuals with better fitness are selected with a higher probability. Then, selected individuals are reproduced using variation operators to generate new offsprings. Finally, a replacement scheme is applied to determine which individuals of the population will survive from the offsprings and the parents. This iteration represents a generation. This process is iterated until a stopping criteria hold.



A generation in evolutionary algorithms.

The below Algorithm resumes the template for the an Evolutionary Algorithm.

Evolutionary Algorithm

```
Generate(P(0)) ;                /* Initial population */
    t = 0 ;
While not Termination Criterion(P(t)) Do
      Evaluate(P(t)) ;
      P‘(t) = Selection(P(t)) ;
      P‘(t) = Reproduction(P‘(t)); Evaluate(P‘(t)) ;
      P(t + 1) = Replace(P(t), P‘(t)) ;
      t = t + 1 ;
End While
Output Best individual or best population found.
```

Common Concepts for Evolutionary Algorithms

- ***Representation***: In the EA community, the encoded solution is referred as *chromosome* while the decision variables within a solution (chromosome) are *genes*. The possible values of variables (genes) are the *allele* and the position of an element (gene) within a chromosome is named *locus*.
- ***Population generation***: This is a common search component for all P-metaheuristics.
- ***Fitness function***: In the EA community, the term fitness refers to the objective function.
- ***Selection strategy***: The selection strategy addresses the following question: “Which parents for the next generation are chosen with a bias toward better fitness?”.
- ***Reproduction strategy***: The reproduction strategy consists in designing suitable mutation and crossover operator(s) to generate new individuals (offsprings).
- ***Replacement strategy***: The new offsprings compete with old individuals for their place in the next generation (survival of the fittest).
- ***Stopping criteria***: This is a common search component for all metaheuristics. Some stopping criteria are specific to P- metaheuristics. Stopping conditions could be:
  - The discovery of an optimal or near optimal solution.
  - Convergence on a single solution or set of similar solutions.
  - After a user-specified threshold has been reached, or
  - After a maximum number of cycles.

2.2 Memetic Algorithm

The basic principle consists in incorporating a local search algorithm during an evolutionary algorithm search. The local search improves the fitness of the population so that the next generation has "better" genes from its parents. The following steps explain the work of the Memetic Algorithm:

1. The population is initialized at random. Then, each individual makes local search to improve its fitness.
2. To form a new population for the next generation, higher quality individuals are selected. Once two parents have been selected, their chromosomes are combined and the classical operators of crossover are applied to generate new individuals.
3. The latter are enhanced using a local search techniques.
4. The role of local search in memetic algorithms is to locate the local optimum. The below Algorithm resumes the template for the memetic algorithm.

Memetic Algorithm

Initialize population *Pop*
Optimize *Pop(Local search)*
Evaluate *Pop*
Repeat
    Select *Parents* from *Pop*
    Recombine *Parents*
    Optimize *Pop(Local search)*
    Evaluate *Pop*
Until Stopping criteria
Output: the best solution in *Pop.*

Reference:

1- Ferrante Neri, Carlos Cotta (auth.), Ferrante Neri, Carlos Cotta, Pablo Moscato (eds.), Handbook of Memetic Algorithms, Springer-Verlag Berlin Heidelberg,2012.
2- *https://www.researchgate.net/publication/373514289_An_Efficient_Method_for_Solving_Traveling_Salesman_Problem"* .

Cultural Algorithms

Cultural algorithms (CAs) are special variants of evolutionary algorithms. CAs have been introduced by R. G. Reynolds in 1994. They are computational models of cultural evolution based upon principles of human social evolution. They employ a model of cultural change within an optimization problem, where culture might be symbolically represented and transmitted between successive populations.

The main principle behind this process is to preserve beliefs that are socially accepted and discard unacceptable beliefs.

Cultural algorithms contain two main elements, a *population space* at the micro evolutionary level and a *belief space* at the macroevolutionary level (see figure).

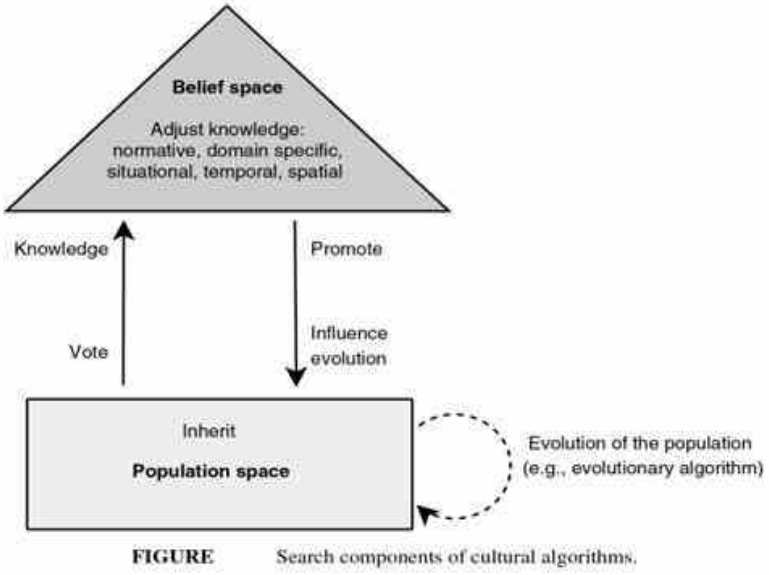


FIGURE Search components of cultural algorithms.

The term "culture" was first introduced by E. B. Tylor in his book *Primitive Culture* in 1881. He defined culture as "that complex whole which includes knowledge, belief, art, morals, customs, and any other capabilities and habits acquired by man as a member of society."

The two elements interact by means of a Vote Inherit Promote or VIP protocol. This enables the individuals to alter the belief space and allows the belief space to influence the ways in which individuals evolve.

The population space at the micro evolutionary level may be carried out by EAs. At each generation, the knowledge acquired by the search of the population (e.g., best solutions of the population) can be memorized in the belief space in many forms such as logic and rule-based models, schemata, graphical models, semantic networks, and version spaces among others to model the macro evolutionary process of a cultural algorithm.

The belief space is divided into distinct categories that represent different domains of knowledge that the population has acquired on the search space:

- normative knowledge (i.e., a collection of desirable value ranges for some decision variables of the individuals in the population),
- domain- specific knowledge (i.e., information about the domain of the problem CA is applied to),
- situational knowledge,
- temporal knowledge (i.e., information of important events about the search), and
- spatial knowledge (i.e., information about the landscape of the tackled optimization problem).

Template of the cultural algorithm is:

Algorithm
Initialize the population Pop(0) ;
Initialize the Belief BLF(0) ; t = 0 ;
Repeat
Evaluate population Pop(t) ;
Adjust(BLF(t),
Accept(POP(t))) ;
Evolve(Pop(t+1),
Influence(BLF(t))) ;
t = t + 1 ;
Until Stopping criteria
Output: Best found solution or set of solutions.

As such, cultural algorithms represent a P-metaheuristic based on hybrid evolutionary systems that integrate evolutionary search and symbolic reasoning. They are particularly useful for problems whose solutions require extensive domain knowledge (e.g., constrained optimization problems) and dynamic environments (e.g., dynamic optimization problems).