Search Methods

Michiel Blommaert

December 2019

CHAPTER 1

Introduction to Search Methods

1.1 PROBLEM FORMULATION

A **problem-solving process** consists of three steps:

- *Formulation* of the problem. This means we have to define the problem, identify the possible solution alternatives (or search space, S) and define the value criteria.
- *Analysis* of the problem.
- *Interpretation*: We evaluate the possible solutions and make our decision.

The **problem** formulation is the first step in the problem-solving process. We will illustrate this using a shortest-path problem. We search the shortest path between two Romanian cities: Timisoara and Bucharest (Fig. 1.1). This problem (and any other problem) can be defined by four items: the *(initial) state*, some *actions*, a *goal test* and a *path cost*.

In this particular problem, we start in Timisoara. So, the initial state x is $Timisoara$. Each state x has a set of action–state pairs $S(x)$. For example: $S(Timisoara) = \{< Timisoara \rightarrow Arad, Arad >, ...\}$. Our destination is Bucharest, so we can formulate the goal test explicitly: $x = Bucharest$. A goal test can also be implicit. The path cost $g(x)$ is the sum of distances in this case.

A **solution** is a sequence of actions leading from the initial state to a goal state. The *optimal solution* is the solution with the smallest path cost. To visualize the problem, we create a **search tree**. Each **node** constitutes part of a search tree and includes a state, a parent node, an action, the path cost and the depth (e.g. Lugoj, Fig. 1.2). Using a search tree, we can explore the state space by generating successors of already-explored states (also called *expanding states*).

The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` function of the problem to create the corresponding states. The order in which we select nodes for expansion is called the **search strategy or method**.

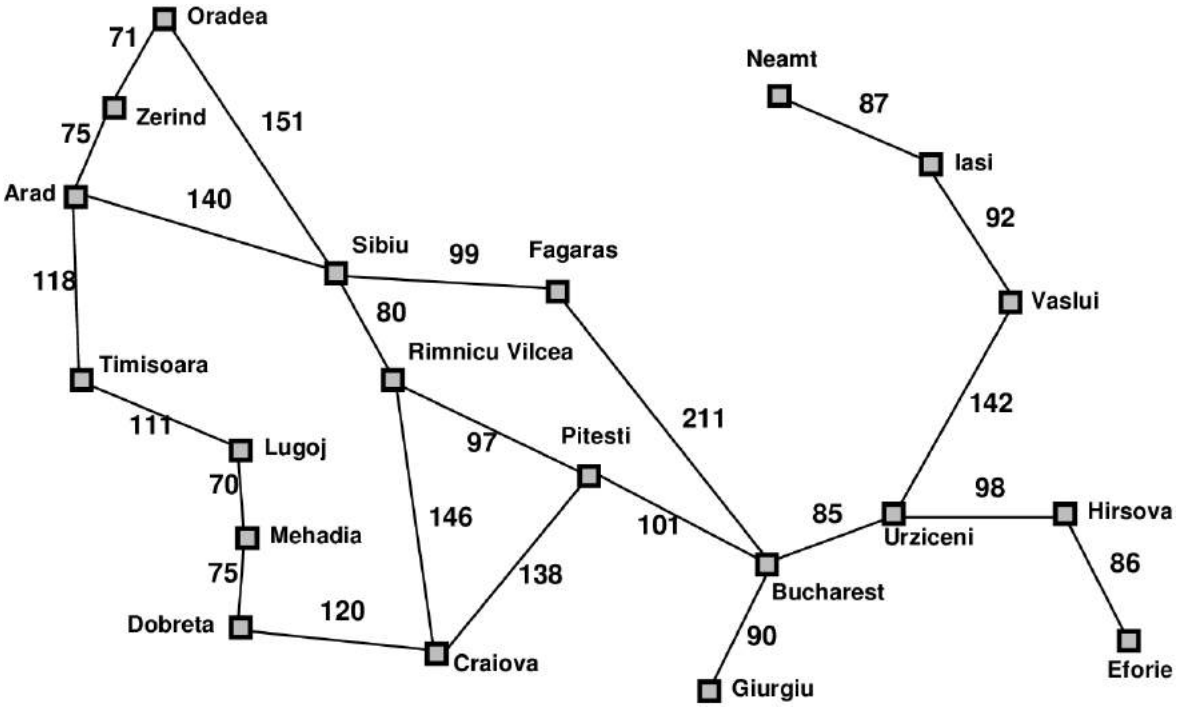


Figure 1.1: A shortest path problem in Romania

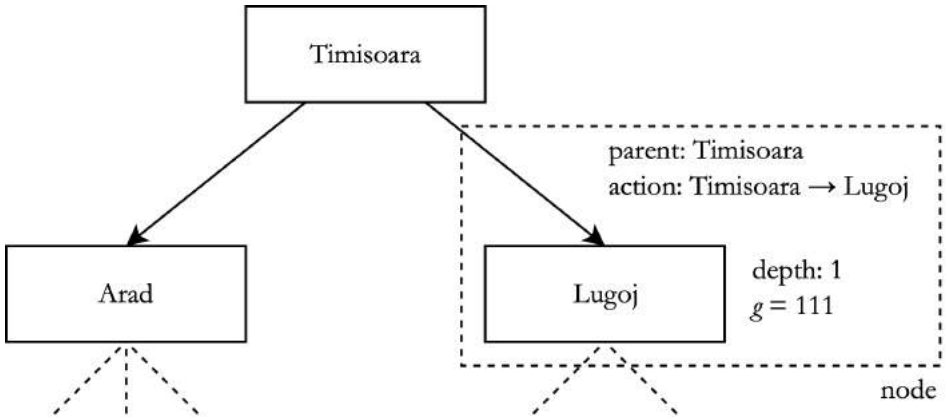


Figure 1.2: An example of a node in a search tree

Strategies are evaluated along the following dimensions:

- *Completeness.* Does it always find a solution if one exists?
- *Time complexity* or number of nodes generated.
- *Space complexity* or maximum number of nodes in memory.
- *Optimality.* Does it always find a least-cost (or optimal) solution?

Time and space complexity are measured in terms of (i) maximum branching factor of the search tree b (i.e. the number of successors generated by a given node), (ii) depth of the least-cost solution d and (iii) maximum depth of the state space m (may be ∞).

1.2 CLASSIFICATION OF SEARCH METHODS

We can classify search methods along two different dimensions (Fig. 1.4).

- *Blind vs. heuristic search methods.* Blind search is also called uninformed or brute force search. It is totally brute in nature because it doesn't have any domain specific knowledge. The search process remembers all the unwanted nodes which are no use for the search process. Therefore, large memory is required. By the use of domain specific knowledge, the search process can be reduced. This procedure is called heuristic search. On the basis of experience or judgement, it offers a reasonable solution to a problem, but the mathematically optimal solution is not guaranteed. No time is wasted in this type of search and no large memory is required.

  EXAMPLE 1.1 In a traveling salesman problem, a salesman has to plan a tour of cities that is of minimal length. Because of its simplicity, the nearest neighbor heuristic is a great method to solve this TSP (traveling salesman problem). In this heuristic, the

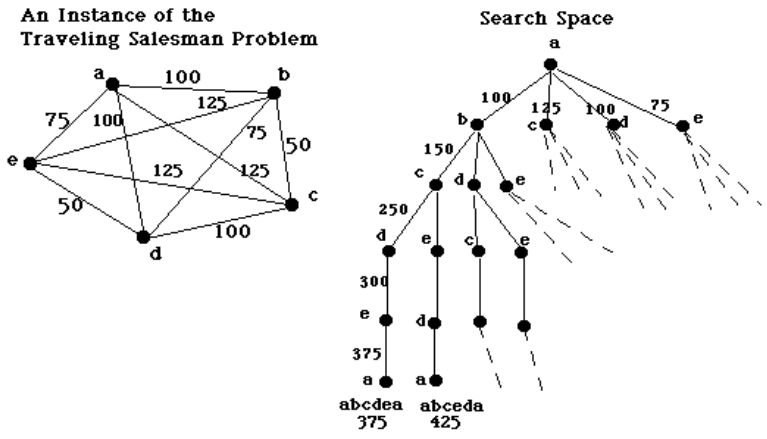


Figure 1.3: The search tree of a traveling salesman problem

  salesman starts at some city and then visits the city nearest to the starting city, and so on, only taking care not to visit a city twice. This method is fast, but doesn't guarantee optimality.

- *Systematic vs. non-systematic search methods.* Non-systematic heuristics are called metaheuristics. Metaheuristics are problem-independent techniques that can be

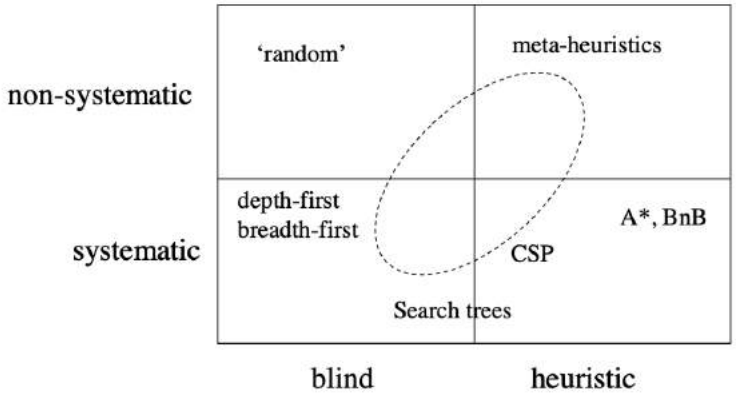


Figure 1.4: Classification of search methods

applied to a broad range of problems. (Systematic) heuristics are often problem-dependent, that is, you define a heuristic for a given problem. You could say that a heuristic exploits problem-dependent information to find a 'good enough' solution to a specific problem, while metaheuristics are, like design patterns, general algorithmic ideas that can be applied to a broad range of problems.
Metaheuristics are generally applied to problems for which there is no satisfactory problem-specific algorithm or heuristic solution method. Examples are genetic algorithms, simulated annealing and tabu search.

CHAPTER 2

Uninformed Search Methods

Uninformed or blind search strategies use only the information available in the problem definition. In this chapter, we will discuss four (systematic) blind search methods: (i) breadth-first search, (ii) depth-first search and two variants of the latter: (iii) depth-limited search and (iv) iterative deepening search.

2.1 BREADTH-FIRST SEARCH

The breadth-first algorithm starts (in most of the cases) at the tree root and explores all the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level. The nodes waiting in a queue to be explored, called *fringe*, is a first-in-first-out (FIFO) queue, i.e., new successors go at the end of the queue. Coded in Python, the breadth-first method has the following structure:

```
def BFS(graph, initial_node, goal_node):
    fringe = [start_node]
    while len(fringe) > 0:
                v = fringe[0]
                if v == goal_node:
                        return v
                for w in graph.adjacentEdges(v):
                        fringe.append(w)
```

EXAMPLE 2.1 We will apply the breadth-first algorithm to the shortest-path problem in Romania. First we check if the initial node is the goal state. Because Timisoara is not

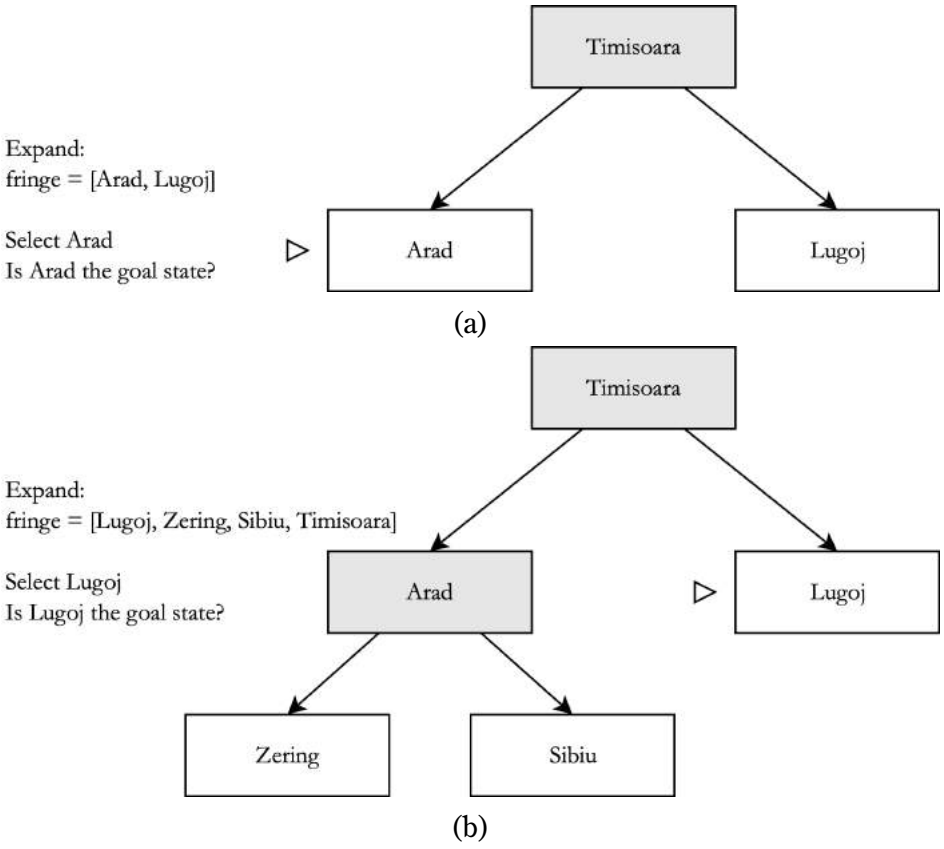


Figure 2.1: Breadth-first search method example

the goal state, we add all the adjacent nodes of Timisoara to the queue (or fringe). The adjacent nodes of Timisoara are Arad and Lugoj. Because Timisoara is explored, the first node in the fringe becomes Arad. Arad is also not the goal state. So, we add all the adjacent nodes of Arad to the queue (Zerind and Sibiu). The first node now becomes Lugoj. Notice that we explore the first depth level nodes (Arad and Lugoj) first before moving on to a deeper level. Lugoj is again not the goal state. We continue this process until the goal state is found.

We will now **evaluate** the breadth-first strategy along the four criteria:

- *Completeness.* The BFS method always finds a solution if one exists (if the branching factor b is finite).
- *Time* complexity or the number of nodes generated. With branching factor b and depth of the optimal solution d we have:

$$1 + b + b^2 + ... + b^d + (b^{d+1} - b)$$

We assumed that the optimal node is the last node we explore at the optimal solution depth. That is why we added $(b^{d+1} - b)$ to the sum.
In computer science, **big O notation** is used to classify algorithms according to how their running time or space requirements grow as the input size grows. Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards infinity. As b grows large, the b^{d+1} term will come to dominate, so that all other terms can be neglected. So the big O notation captures what remains:

$$O(b^{d+1})$$

- *Space* complexity or the maximum number of nodes in memory. This is the same as the time complexity, since the BFS method keeps every node in memory (either in fringe or on a path to fringe).
- *Optimality.* The BFS solution is optimal if we assume that deeper solutions are less optimal (step-cost equal to 1). In other cases (like the TSP in Romania), the BFS solution can be suboptimal.

Space is the biggest problem, more than time.

2.2 DEPTH-FIRST SEARCH

The depth-first search (DFS) algorithm starts at the tree root and and explores as far as possible along each branch before backtracking (i.e. abandoning a branch as soon as that branch cannot possibly contain a valid solution. The fringe is a last-in-first-out (LIFO) queue, i.e., new successors go at the front of the queue. Coded in Python, the DFS method has the following structure:

```
def DFS(graph, initial_node, goal_node):
    fringe = [start_node]
    while len(fringe) > 0:
                v = fringe[0]
                if v == goal_node:
                        return v
                for w in graph.adjacentEdges(v):
                        fringe.insert(0, w)
```

EXAMPLE 2.2 We will apply the depth-first algorithm to the shortest-path problem in Romania. First we check if the initial node is the goal state. Because Timisoara is not the goal state, we add all the adjacent nodes of Timisoara to the queue (or fringe). The adjacent nodes of Timisoara are Arad and Lugoj. Because Timisoara is explored, the first node in the fringe becomes Arad. Arad is also not the goal state. So, we add all the adjacent nodes of Arad at the front of queue (Zerind and Sibiu). The first node now becomes Zerind. Notice that we explore the Timisoara-Arad-Zering branch first before moving on to another branch. Zering is again not the goal state. We continue this process until the goal state is found.

The **properties** of the depth-first strategy are not that good:

- *Completeness.* The BFS method fails to find a solution in infinite-depth spaces ($m = \infty$). In some cases, this can be solved by avoiding repeated states along a path (e.g. returning from Arad to Timisoara is not a option).
- *Time* complexity or the number of nodes generated. With branching factor b and maximum depth of the state space m we have:

$$1 + b + b^2 + ... + b^m$$

This is of course again a worst-case representation: we assume that the solution

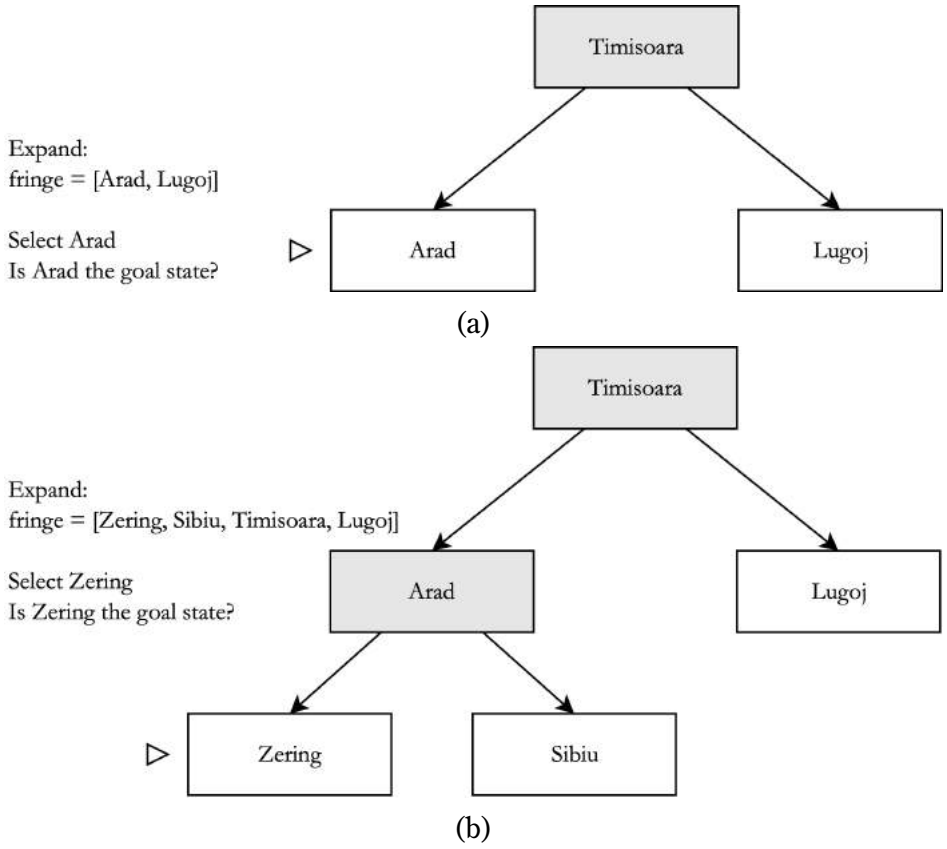


Figure 2.2: Depth-first search method example

is found on the last branch at maximum depth. Written in big O notation we have:

$$O(b^m)$$

We say that the algorithm has order of b^m time complexity.

- *Space* complexity or the maximum number of nodes in memory. We only need to remember a single path and the expanded unexplored nodes. We have a linear space:

$$bm + 1 = O(bm)$$

- *Optimality*. No, it may find a non-optimal goal first.

2.3 DEPTH-LIMITED AND ITERATIVE DEEPENING SEARCH

To avoid the infinite depth problem of the DFS algorithm, we can decide to only search until depth l, i.e. we don't expand beyond depth l. This is what we call **depth-limited search**. The time and space complexity is better (respectively $O(b^l)$ and $O(bl)$). But what if the solution is deeper than l? There will be no solution. We can solve this problem by increasing l iteratively. This is what we call **iterative deepening search**. The IDS algorithm always finds a solution.

The number of nodes generated in an iterative deepening search to depth d with branching factor b is:

$$(d+1)b^0 + db^1 + ... + 2b^{d-1} + 1b^d = O(b^d)$$

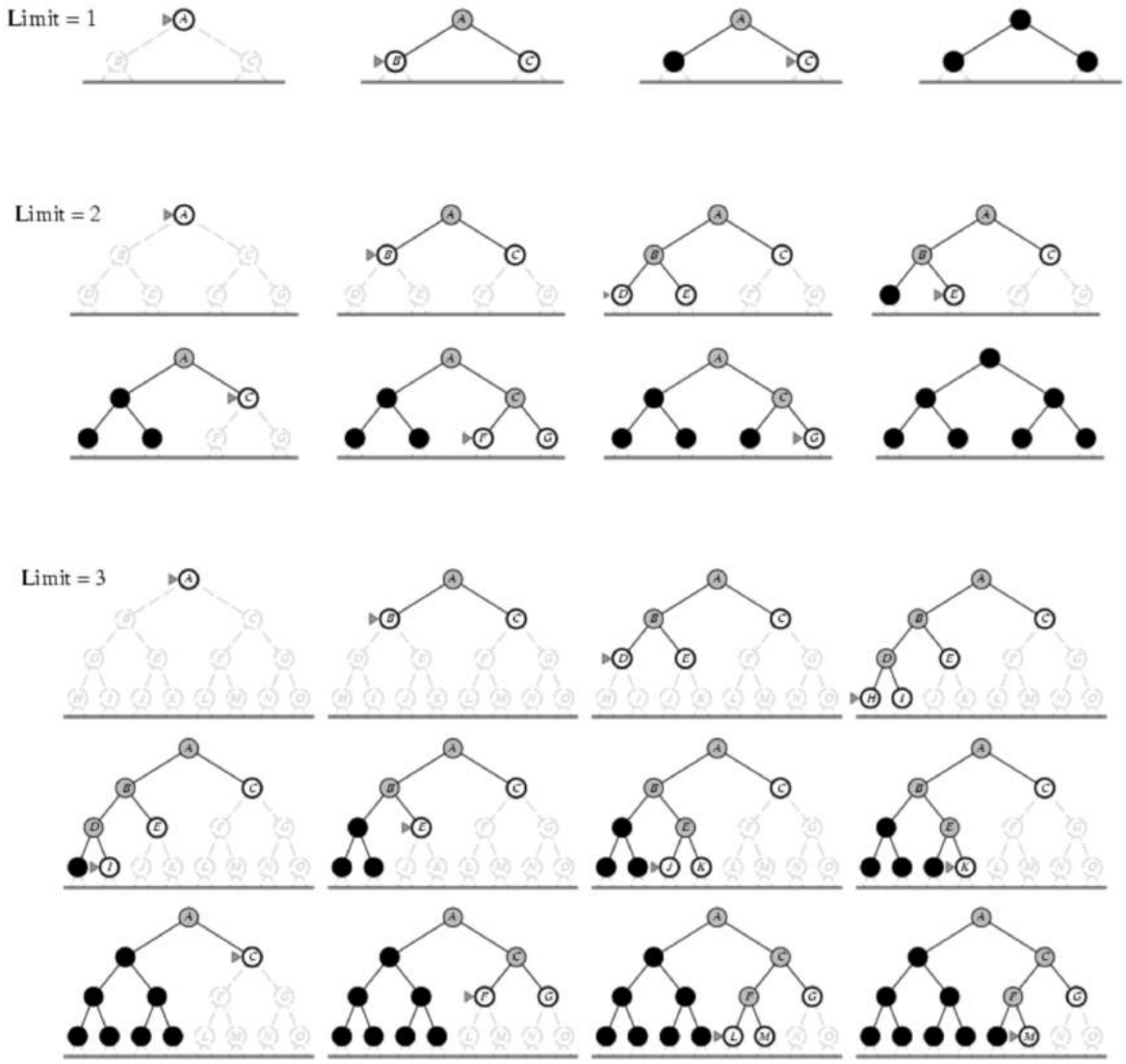


Figure 2.3: Iterative deepening search example

Criterion	Breadth-First	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal	Yes	No	No	Yes

Figure 2.4: Summary of uninformed search methods

The maximum number of nodes in memory is:

$$bd + 1 = O(bd)$$

We can conclude that IDS inherits the memory advantage of depth-first search (linear), and is better in terms of time complexity than breadth first search.

CHAPTER 3

Informed Search Methods

In uninformed search methods (like BFS and DFS), when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. An informed search method uses an evaluation function $f(n)$ to decide which adjacent node is the most promising and then explores it. So, the nodes in fringe are ordered in decreasing order of *desirability* and the desirability of a node is estimated by the evaluation function. This is also known as **best-first search**.

In this chapter, we will discuss two algorithms, each having a different evaluation function (or heuristic): *greedy best-first search* and *A* search*.

3.1 GREEDY BEST-FIRST SEARCH

For a greedy best-first search the evaluation function $f(n)$ is a heuristic, i.e. an estimation of the cost from node n to the goal. Greedy best-first search expands the node that *appears* to be the closest to the goal.

EXAMPLE 3.1 For example, if we apply greedy best-first search to the shortest path problem in Romania, we can use the straight-line distance from a node n to Bucharest as the heuristic to evaluate the desirability of this node. So,

$$f(n) = h_{SLD}(n)$$

with $h_{SLD}(n)$ the straight-line distance. Using this heuristic we can search for the shortest path between Arad and Bucharest (Fig. 3.2).

- Arad has three adjacent nodes: Sibiu, Timisoara and Zerind which have a straight-line distance to Bucharest of respectively 253, 329 and 374 kilometers (see table in Fig. 3.1). Sibiu is the most desirable node, because it has the shortest SLD to Bucharest. So, we will expand the Sibiu node.

- Sibiu has four adjacent nodes from which Fagaras has the shortest SLD (176 km). This is also shorter than the unexplored nodes at the first depth level (Timisoara and Zerind). Fagaras is thus the most desirable node and we will expand it.
- Fagaras has two adjacent node. One of these is Bucharest, our goal node. The solution is Arad–Sibiu–Fagaras–Bucharest.

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Figure 3.1: Straight-line distance to Bucharest

Below are the **properties** of the greedy best-first strategy:

- *Completeness.* The greedy best-first method does not always find a solution. It can get stuck in loops (e.g. Leuven–Brussels–Leuven–Brussels ... in Fig. 3.3)
- *Time* complexity is $O(b^m)$, but a good heuristic can give dramatic improvement.
- *Space* complexity is the same as time complexity, since all the nodes are kept in memory.
- *Optimality.* No, it may find a non-optimal path. For example, the path found in the SPP in Romania is not the optimal path (Fig. 3.4).

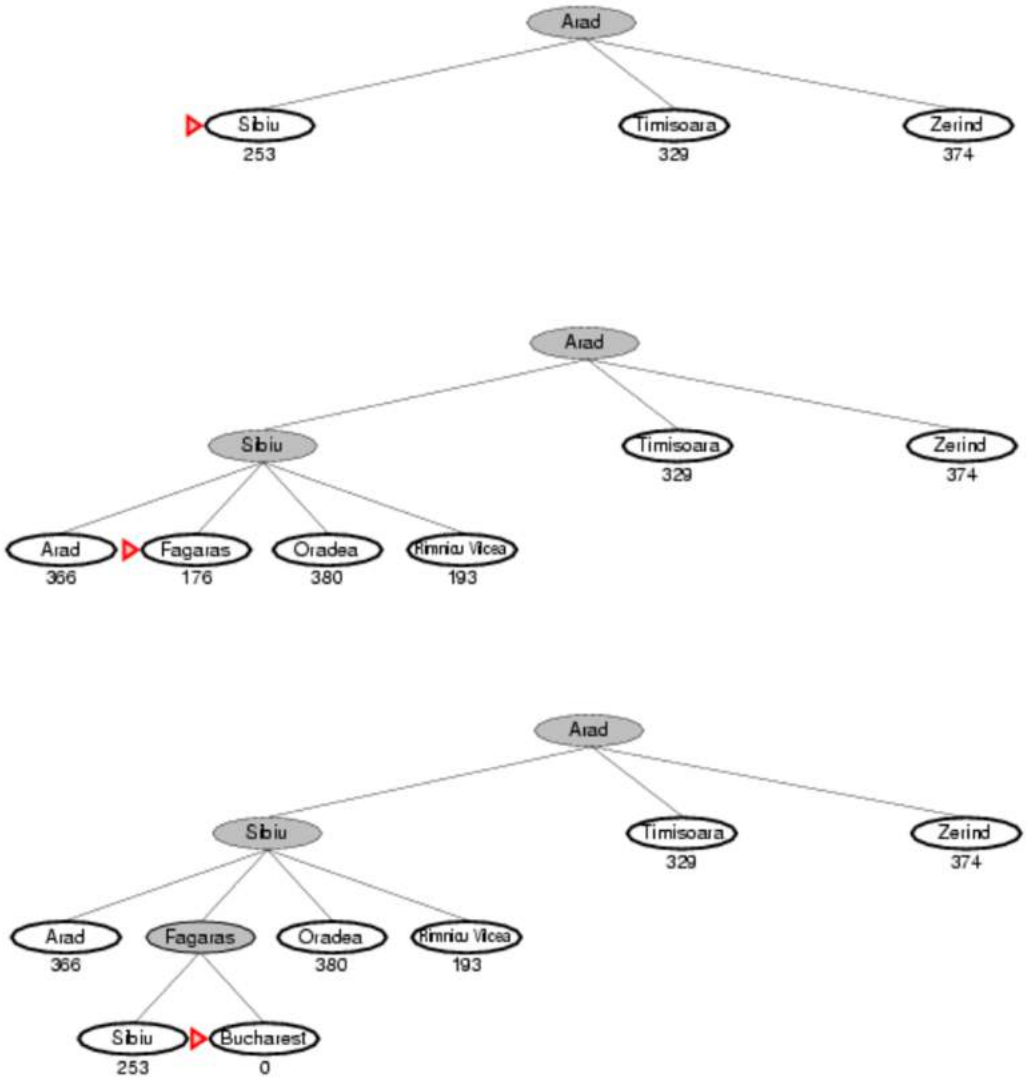


Figure 3.2: Example greedy best-first search algorithm

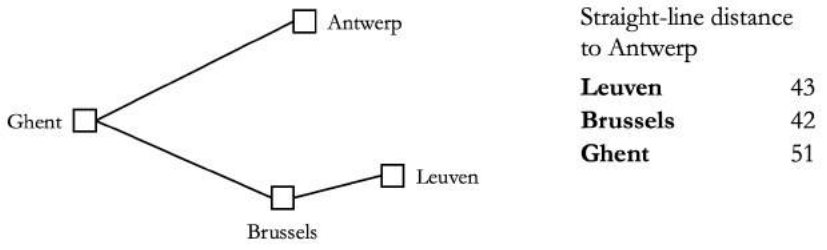


Figure 3.3: Shortest-path problem from Leuven to Antwerp

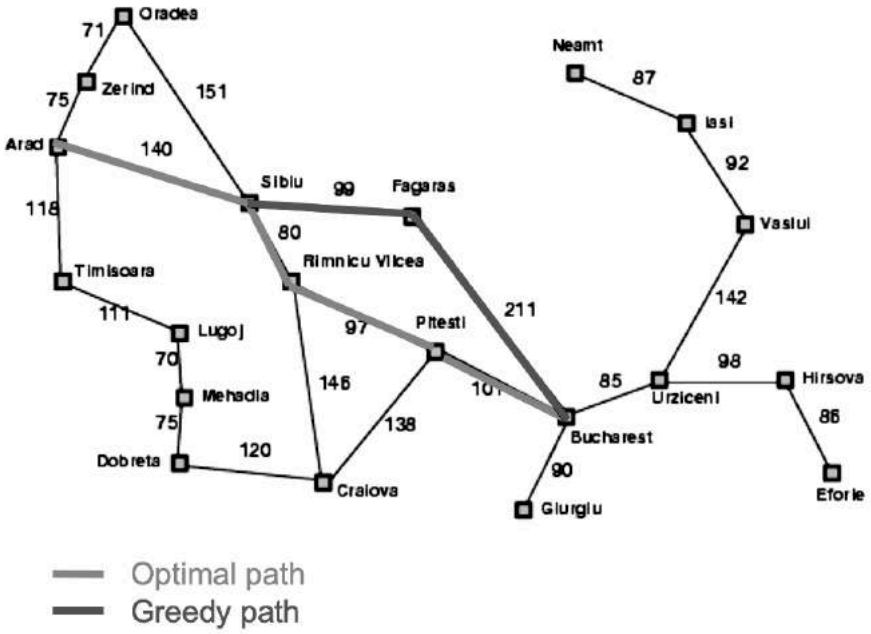


Figure 3.4: The greedy path versus the optimal path (shortest-path example)

3.2 A* SEARCH

The idea of A* search is avoiding paths that are already expensive. The evaluation function is the sum of the cost so far to reach the node $g(n)$ and the estimated cost from the node to the goal $h(n)$. Brief, the evaluation function is the estimated total cost of the path through n to the goal:

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

A* search is actually a form of (implicit) backtracking.

EXAMPLE 3.2 For example, if we apply A* search to the shortest path problem in Romania, $h(n)$ is the straight-line distance from a node n to Bucharest like the greedy method and $g(n)$ is the distance so far to reach the node. So,

$$f(n) = g(n) + h_{SLD}(n)$$

with $h_{SLD}(n)$ the straight-line distance. Using this evaluation function we can search for the shortest path between Arad and Bucharest (Fig. 3.6). The solution is Arad–Sibiu–Rimnicu Vilcea–Pitesti-Bucharest. This is also the optimal solution (Fig. 3.4).

Does A* search always offer an optimal solution? The answer is yes, if the heuristic $h(n)$ is admissible, i.e. it never overestimates the cost to reach the goal (it is optimistic). A heuristic $h(n)$ is **admissible** if for every node n

$$h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to reach the goal state from n. For example, the straight-line distance used in the shortest-path example in Romania will always be less than the actual distance. If there are two (or more) admissible heuristics, then we say that one heuristic h_1 **dominates** another heuristic h_2 if

$$h_2 \geq h_1$$

for all n.

But how do we know if a heuristic is admissible? The cost of an optimal solution to a relaxed problem (i.e. a problem with fewer restrictions on the actions) is an admissible heuristic for the original problem.

EXAMPLE 3.3 Let's apply A* search on an 8-puzzle (Fig. 3.5). There are two possible admissible heuristics that we can use:

- The number of misplaced tiles $h_a(n)$.
- The total Manhattan distance $h_b(n)$, i.e. the sum of the number of squares from the desired location for each tile.

Both heuristics are admissible because they both give the shortest solution for a relaxed version of the puzzle. If the rules of the puzzle are relaxed so that the tile can move anywhere, then h_a gives the shortest solution. If the rules of the puzzle are relaxed so that the tile can move to any adjacent square, then h_b gives the shortest solution. Typical search costs (average number of expanded nodes) for this problem are:

$d = 12$ IDS = 3,644,035 nodes
$A^*(h_a)$ = 227 nodes
$A^*(h_a)$ = 73 nodes

$d = 24$ IDS = too many nodes
$A^*(h_a)$ = 39,135 nodes
$A^*(h_a)$ = 1,641 nodes

We see that in this case, h_b dominates h_a.

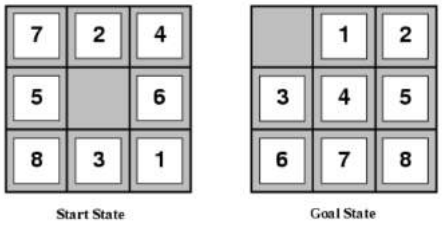


Figure 3.5: 8-puzzle problem

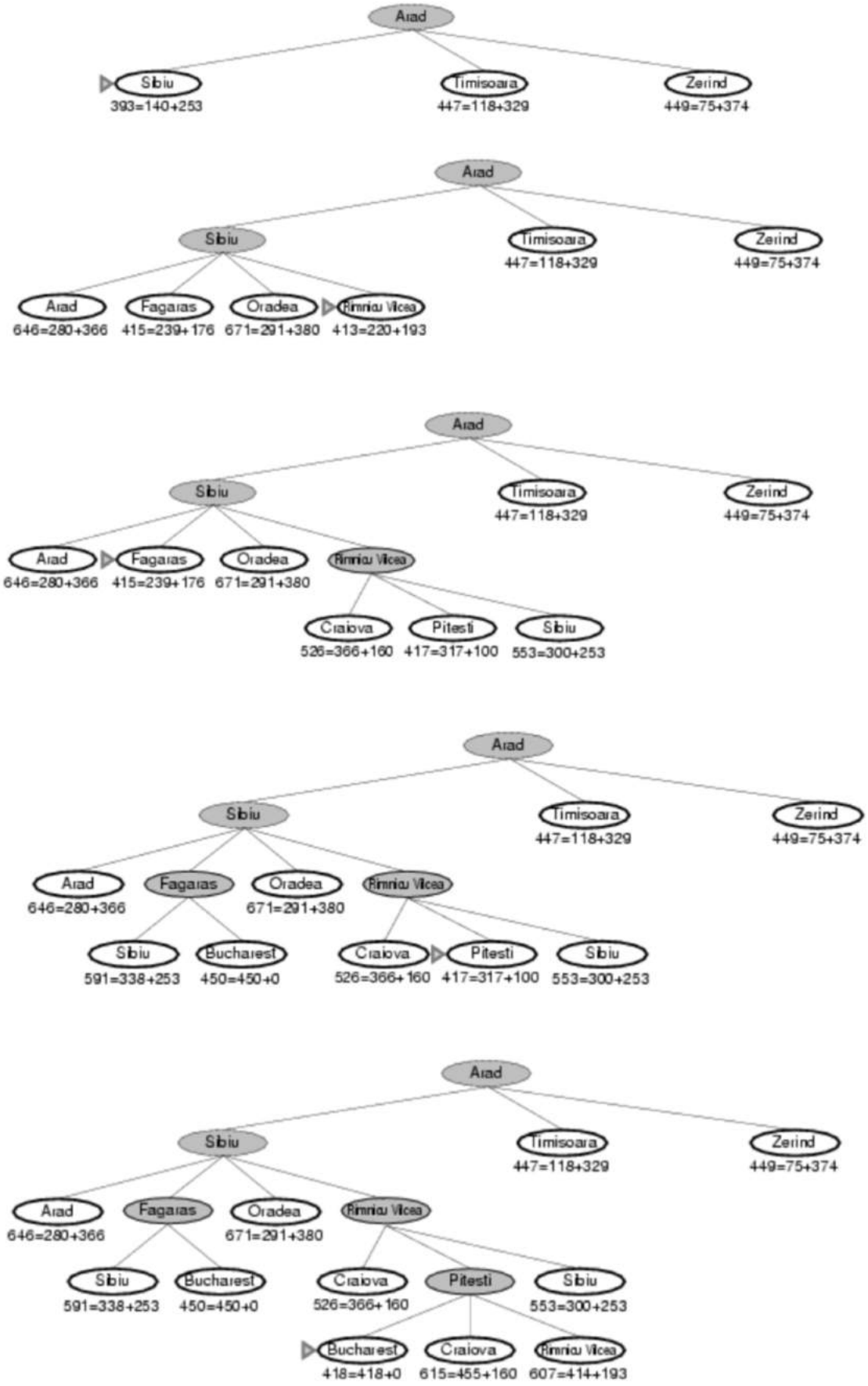


Figure 3.6: Example A* search algorithm

CHAPTER 4

Metaheuristics

Heuristics are often problem-dependent, that is, you define a heuristic for a given problem (like in Ex. 3.1 or 3.3). This kind of heuristics are called systematic heuristics. Metaheuristics are **non-systematic**. Metaheuristics are **problem-independent** techniques that can be applied to a broad range of problems. You could say that a heuristic exploits problem-dependent information to find a 'good enough' solution to a specific problem, while metaheuristics are, like design patterns, general algorithmic ideas that can be applied to a broad range of problems.

Global (or complete) search algorithms like A* will always find the correct or optimal solution if there is one, given enough time. But, in a lot of practical cases, this is inefficient.

In many optimization problems, the path to the goal is irrelevant. The goal state itself is the solution and the state space is a set of *complete* configurations. In such cases, we can use **local** search algorithms. Local search algorithms move from solution to solution in the search space by applying local changes, until a solution deemed optimal is found or a time bound is elapsed. We keep a single *current* state and try to improve it. Local search algorithms like greedy search will not always find the correct or optimal solution, if one exists. They have a tendency to become stuck in suboptimal regions or on plateaus where many solutions are equally fit (Fig. 3.3). They sacrifice completeness for greater efficiency by ordering partial solutions by some heuristic predicting how close a partial solution is to a complete one.

EXAMPLE 4.1 A great example to illustrate the limits of local search is gradient descent. The goal of gradient descent is to minimize an objective function $f(x)$. The algorithm only permits moves to neighbour solutions that improve the current objective function value and ends when no improving solutions can be found. The final x obtained by a descent method is called a local optimum, since it is at least as good as or better than all solutions in its neighborhood. The evident shortcoming of a descent method is that such a local optimum in most cases will not be a global optimum (Fig. 4.1).

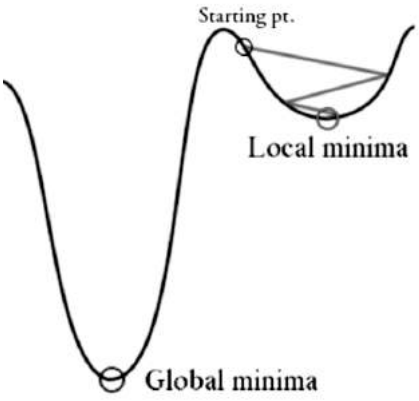


Figure 4.1: Gradient descent, stuck at local optimum

Many metaheuristics were proposed to improve local search heuristics in order to find better solutions. Such metaheuristics include *simulated annealing* and *tabu search*. These metaheuristics can both be classified as local search-based or global search metaheuristics. Other global search metaheuristic that are not local search-based are usually population-based metaheuristics, like *genetic algorithms*.

4.1 GENETIC ALGORITHMS

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals have a higher chance for being selected for reproduction in order to produce offspring of the next generation.

This notion can be applied for a search problem. We consider a set of solutions for a problem and select the set of best ones out of them. Five phases are considered in a genetic algorithm: (i) the initial population, (ii) the fitness function, (iii) selection, (iv) crossover and (v) mutation.

The process begins with a set of k randomly generated states which is called a the **population**. A state is characterized by a set of parameters (variables) known as genes (Fig. 4.2). In a genetic algorithm, the set of genes (or chromosome) of a state is represented using a string over a finite alphabet (often a string of 0s and 1s).

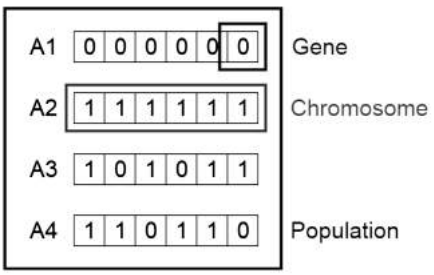


Figure 4.2: Population, chromosomes and genes

The **fitness function** (or evaluation function) determines how fit a state is. It gives a fitness score to each state. The probability that an individual will be **selected** for reproduction is based on its fitness score.

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes (with probability p_c, otherwise the children are a copy of the parents). p_c is typically in the range $[0.6, 0.9]$. Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached (Fig. 4.3). The whole population is replaced by the resulting offspring.

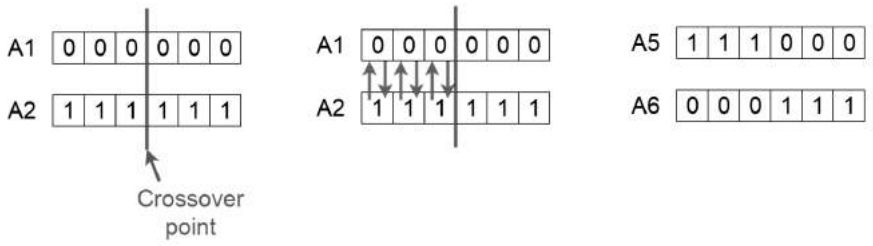


Figure 4.3: Crossover

In certain new offspring formed, some of their genes can be subjected to a **mutation** with a low random probability. This implies that some of the bits in the bit string can be flipped with probability p_m independently for each bit. The mutation rate p_m is typically in the range of

$$\frac{1}{N} \leq p_m \leq \frac{1}{L}$$

with N the population size and L the chromosome length. Mutation occurs to maintain diversity within the population and prevent premature convergence.

The algorithm terminates if the population has converged (does not produce offspring which are significantly different from the previous generation). Then it is said that the genetic algorithm has provided a set of solutions to our problem.

4.1.1 The Simple Generic Algorithm

John Holland introduced genetic algorithms in 1960. Holland's genetic algorithm is now known as the simple genetic algorithm (SGA).

The SGA has been subject of many (early) studies and is still often used as benchmark for novel genetic algorithms. But it shows many **shortcomings**:

- The representation is too restrictive.
- Mutation and crossovers are only applicable for bit-string and integer representations.
- The selection mechanism is sensitive for converging populations with close fitness values.
- The generational population model (the whole population is replaced by the resulting offspring) can be improved with explicit survivor selection.

Other genetic algorithms use different representations, mutations, crossovers and selection mechanisms.

4.1.2 Alternative Crossover Operators

The performance with **one-point-crossover** (used in SGAs) depends on the order that variables occur in the chromosomes. Genes that are near each other are more likely to be kept together. Genes from opposite ends of the string can never be kept together. This is known as *positional bias*. This can be exploited if we know about the structure of our problem, but this is not usually the case.

A (partial) solution for the bias could be choosing n random crossover points and split the strings along those points. The resulting parts are glued together, alternating between parents (Fig. 4.4). Using **n-point-crossover**, we still keep some positional bias.

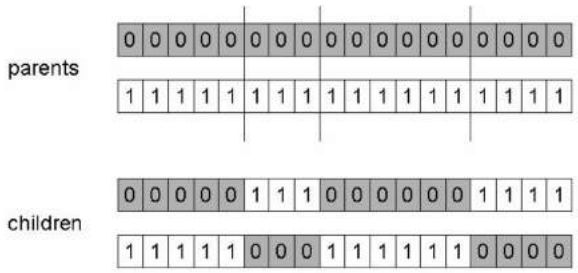


Figure 4.4: n-point-crossover

Uniform crossover is a more general version of the n-point crossover. In this scheme, at each bit position of the parent string, we toss a coin to determine whether there will be swap of the bits or not (Fig. 4.5). Inheritance is independent of the position using this method.

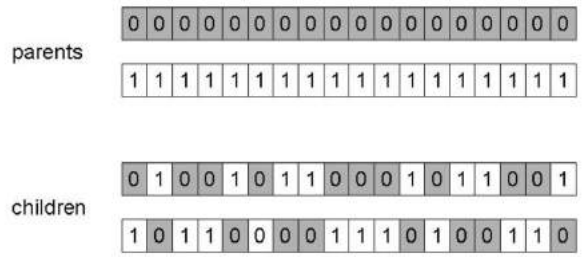


Figure 4.5: Uniform crossover

4.1.3 Crossover or Mutation?

Should we use crossover or mutation? Or both? It depends on the problem, but there is a wide agreement that, in general, it is good to have both. Both have another role:

- *Crossover* is explorative, i.e. gaining information on the problem. It makes a big jump to an area somewhere *in between* two (parent) areas.
- *Mutation* is exploitative, i.e. using the information. It creates random small diversions, thereby staying near (in the area of) the parent.

Using mutations only is possible, but crossover-only-GAs do not work. To hit the optimum you often need a *lucky* mutation.

4.1.4 Design of the Representation

In biology, an organism's genotype is the set of genes that it carries and an organism's phenotype is all of its observable characteristics. We can make the parallel with genetic algorithms: the characteristics of a state are **represented** by a string over a finite alphabet, often a string of 0s and 1s (Fig. 4.6).

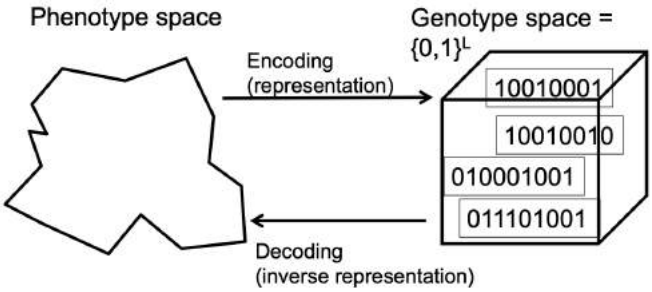


Figure 4.6: Phenotype vs. genotype

Smoother genotype-phenotype mapping, i.e. small changes in the genotype cause small changes in the phenotype, makes life easier for the genetic algorithm.

EXAMPLE 4.2 An example of smooth genotype-phenotype mapping is Gray code of integers. Gray code is an ordering of the binary numeral system such that two successive values differ in only one bit (Fig. 4.7).

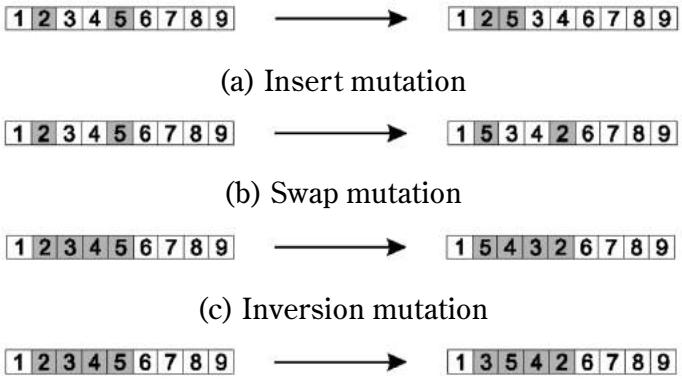
Decimal Value	Binary Code	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100

Figure 4.7: Binary versus Gray code

Nowadays it is generally accepted that it is better to encode numerical variables directly as integers or floating point variables. Some problems naturally have integer variables (e.g. image processing parameters), but others take categorical values from a fixed set (e.g. blue, green, yellow, pink) which we can represent by integers. For **integer representation** there are two principal forms of mutation:

- **Random choice**. A new value is randomly chosen from the set of permissible integer values. This method is most suitable for *categorical* variables.
- **Creep mutation**. A small (positive or negative) integer value is added to the gene value. The random value is sampled from a symmetric distribution around 0 with a higher probability of small changes. This method is most suitable for *ordinal* variables.

Ordering (or sequencing) problems form a special type. These problems are generally expressed as a permutation: if there are n variables then the representation is as a list of n integers, each of which occurs exactly once. For **permutation representation**, normal mutation operators lead to inadmissible solutions. For example, assume that a gene has value j. Changing to some other value k would mean that k occurs twice now and j no longer occurs. Therefore we must change at least two values. The mutation rate p_m now reflects the probability that some operator is applied once to the whole string, rather than individually in each position. For permutation representation there are four principal forms of mutation:



(a) Insert mutation

(b) Swap mutation

(c) Inversion mutation

(d) Scramble mutation

Figure 4.8: Mutation methods for permutations

- **Insert mutation**. Two allele values are picked at random. We move the second to follow the first, shifting the rest along to accommodate. Note that this preserves most of the order and the adjacency information.
- **Swap mutation**. Pick two alleles at random and swap their positions. This preserves most of the adjacency information (four links are broken), but disrupts the order more.
- **Inversion mutation**. Pick two alleles at random and then invert the substring between them. This preserves most of the adjacency information (only two links are broken), but disrupts the order information.
- **Scramble mutation**. Pick a subset of genes at random and randomly rearrange the alleles in those positions. Note that the subset does not have to be contiguous.

Also normal crossover operators will often lead to inadmissible solutions for permutations. Many specialised operators have been devised which focus on combining order or adjacency information from the two parents:

- **Order-1-crossover**. The idea is to preserve the relative order in which the elements occur. The procedure has the following shape: (i) choose an arbitrary part from the first parent and copy this part to the first child. (ii) Copy the numbers that are not in the first part, to the first child, using the order of the second parent starting right from the cut point of the copied part, wrapping around at the end. Do the same for the second child, with the parent roles reversed.
- **Partially mapped crossover (PMX)**. The procedure for parents p_1 and p_2 has the following shape: (i) Choose a random segment and copy it from p_1. (ii) Starting from the first crossover point look for elements in that segment of p_2 that have

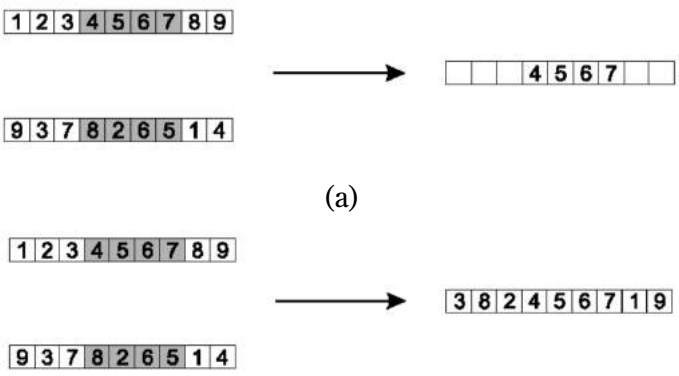


Figure 4.9: Order-1-crossover

not been copied. For each of these i, look in the offspring to see what element j has been copied in its place from p_1. Place i into the position occupied by j in p_2, since we know that we will not be putting j there (as it is already in the offspring). (iii) Having dealt with the elements from the crossover segment, the rest of the offspring can be filled from p_2. The second child is created analogously.

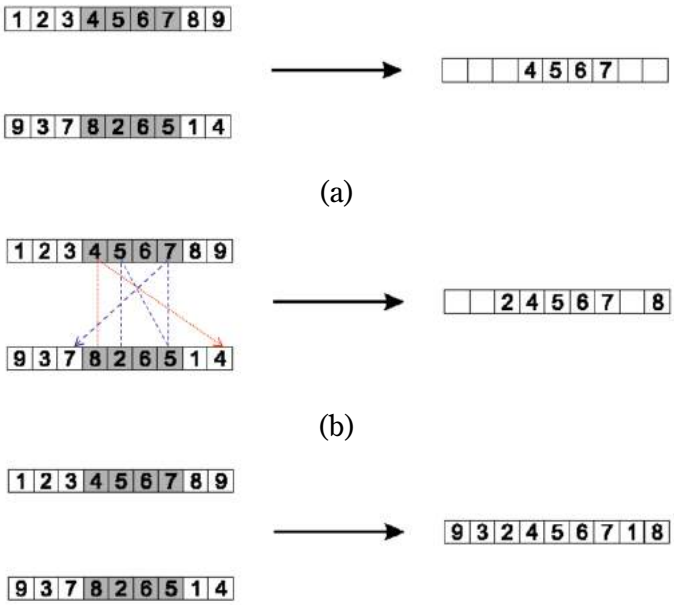


Figure 4.10: PMX

4.1.5 Design of the Selection

Simple genetic algorithms use a *generational model*, i.e. each individual survives for exactly one generation. The entire set of parents is replaced by the offspring. At the other end of the scale are *steady-state models*, i.e. one offspring is generated per generation and only one member of the population is replaced. We define the

generation gap as the proportion of the population that is replaced (1 for SGA and $1/N$ for steady-state GA).

4.2 SIMULATED ANNEALING

Simulated annealing is a metaheuristic to approximate global optimization in a large search space. It is often used when the search space is discrete (e.g. the traveling salesman problem). The name and inspiration come from annealing in metallurgy, a technique involving heating and slow cooling of a material to increase the size of its crystals and reduce their defects. This notion of slow cooling implemented in the simulated annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored. Accepting worse solutions is a fundamental property of metaheuristics because it allows for a more extensive search for the global optimal solution. The **algorithm** contains five steps:

- *Initialize.* Start with a random initial placement (initialize a very high *temperature*).
- *Move.* Perturb the placement through a defined move.
- *Calculate energy.* Calculate the change in the energy (ΔE) due to the move made. An increase in energy means that we come closer to the optimal solution.
- *Choose.* Depending on the change in the energy, we accept or reject the move. If $\Delta E > 0$, we accept the move. Otherwise, the probability of acceptance p_a depends on the current temperature T:

$$p_a = P(e^{-\frac{\Delta E}{T}} > r)$$

  with r a random number between 0 and 1. As the temperature decreases, the probability of accepting worse moves decreases. If $T = 0$, no worse moves are accepted.
- *Update and repeat.* Update the temperature value by lowering the temperature. Go back to the move-step.

The process is done until the *freezing point* is reached. Coded in Python, the simulated annealing method has the following structure:

```
def SA(intial_c, T_0, T_min):
    c = initial_c
    for T in range(T_0, T_min - 1, -1):
        E_c = energy(c)
        n = next(c)
        E_n = energy(n)
        delta_E = E_n - E_c
        if(delta_E > 0):
            c = n
        else if(math.exp(-(delta_E/T)) > random.uniform
           (0, 1)):
            c = n
    return c
```

Simulated annealing is one of the most important metaheuristics of combinatorial optimization, whose properties of convergence towards high quality solutions are well known, although with a high computational cost. Due to that, it has been produced a quite number of research works on the convergence speed of the algorithm, especially on the treatment of the temperature parameter, which is known as the **cooling schedule or strategy**:

- The *initial temperature* T_0 must be hot enough to allow moves to almost all neighbourhood state, else we are in danger of implementing steepest descent. But, it must also not be so hot that we conduct a random search for a period of time (processing time). If we know the maximum change in the `energy` function, we can use this to estimate T_0.
- The *final temperature* T_{min}. It is usual to let the temperature decrease until it reaches zero. However, this can make the algorithm run for a lot longer, especially when a geometric cooling schedule is being used. In practise, it is not necessary to let the temperature reach zero because the chances of accepting a worse move are almost the same as the temperature being equal to zero. Therefore, the stopping criteria can either be a suitably low temperature or when the system is *frozen* at the current temperature, i.e. no better or worse moves are being accepted.
- The *plateau length*. Theory states that we should allow enough iterations at each temperature so that the system stabilises at that temperature (thermal equilibrium). Unfortunately, theory also states that the number of iterations at each temperature to achieve this might be exponential to the problem size. We need to compromise: we can either do this by doing (i) a large number of iterations at a few temperatures, (ii) a small number of iterations at many temperatures or (iii) a balance between the two. An alternative is to (iv) dynamically change the number of iterations as the algorithm progresses. At lower temperatures it is important that a large number of iterations are done so that the local optimum can be fully explored. At higher temperatures, the number of iterations can be less.
- The *temperature decrement* can be either linear

$$T_{t+1} = T_t - x$$

  or geometric

$$T_{t+1} = \alpha T_t$$

  with α between 0.8 and 0.99 (from experience). Better results are found in the higher end of this range. Of course, the higher the value of α, the longer it will take to decrement the temperature to the stopping criterion.

  EXAMPLE 4.3 A cooling startegy, first suggested by Lundy in 1986, is to only do one iteration at each temperature, but to decrease the temperature very slowly. The formula used by Lundy is:

$$T_{t+1} = \frac{T_t}{1 + \beta T_t}$$

  where β is a suitably small value.

The **evaluation function** (or `energy` function) is calculated at every iteration. This is often the most expensive part of the algorithm. Therefore, we need to evaluate the energy level as efficiently as possible.

4.3 TABU SEARCH

Tabu search is a metaheuristic search method employing local search methods: it moves iteratively from one potential solution c to an improved solution n in the neighborhood $N(c)$ of c, until some stopping criterion has been satisfied (generally, an attempt limit or a score threshold).

Like we already learned in the introduction of this chapter, local search methods do not always find the optimal solution. Tabu search enhances the performance of local search by relaxing its basic rules:

- At each step *worsening moves can be accepted* if no improving move is available (like when the search is stuck at a strict local minimum). If there is no better solution in neighbourhood $N(c)$ of c then the next best solution n is chosen in $N(c)$.
- Another local search problem is that cycles can occur: c can be the best candidate solution in $N(n)$ where n is the best solution in $N(c)$. We solve this by the use of memory structures. *Recency-based memory* is the most common memory structure used in tabu search implementations. As its name suggests, this memory structure keeps track of solutions attributes that have changed during the recent past. To exploit this memory, selected attributes that occur in solutions recently visited are labeled *tabu-active*, and particular moves that contain tabu-active elements, are those that become *tabu*. The solutions admitted to the modified neighborhood, $N^*(c)$, are determined by this *short-term memory (STM)*. But, cycles are not completely avoided. Only the last L moves are stored in a tabu list. If the list is full, the last element is dropped. Sometimes it is necessary to overrule the tabu status using *aspiration level conditions*. A simple and commonly used aspiration criterion is to allow solutions which are better than the currently-known best solution. Brief, the number of admissible moves in the neighborhood of the current solution c depends on (i) the tabu activation rules, (ii) the move type, (iii) the aspiration criteria and (iv) the tabu tenure, i.e. the number of iterations an attribute remains tabu-active (Fig. 4.11).

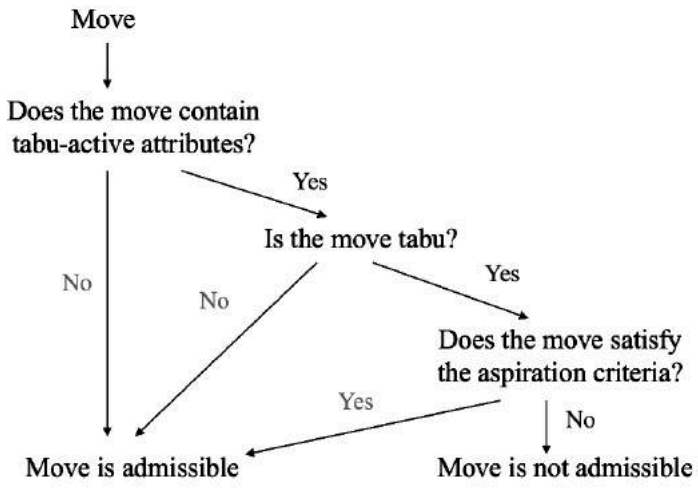


Figure 4.11: Is a move admissible?

Tabu search has several similarities with simulated annealing, as both involve possible downhill moves. In fact, simulated annealing could be viewed as a special form of tabu search where we use *graduated tenure*, that is, a move becomes tabu with a specified probability.

There are many forms in which a simple tabu search implementation can be improved by the use of *long-term memory (LTM)*. The most commonly used methods are frequency-based memory, strategic oscillation and path relinking.

4.3.1 Frequency-based Memory

Frequency-based memory provides a type of information that complements the information provided by recency-based memory, broadening the foundation for selecting preferred moves. Two examples are:

- *Transition measure*, i.e. the number of iterations where an attribute has been changed (e.g. added or deleted from a solution).
- *Residence measure*, i.e. the number of iterations where an attribute has stayed in a particular position (e.g. belonging to the current solution).

4.3.2 Strategic Oscillation

Strategic oscillation operates by orienting moves in relation to a critical level, as identified by a stage of construction or a chosen interval of functional values. Such a *critical level* or *oscillation boundary* often represents a point where the method would normally stop. Instead of stopping when this boundary is reached, however, the rules for selecting moves are modified, to permit the region defined by the critical level to be crossed. The approach then proceeds for a specified depth beyond the oscillation boundary, and turns around. The oscillation boundary again is approached and crossed, this time from the opposite direction, and the method proceeds to a new turning point (Fig. 4.12).

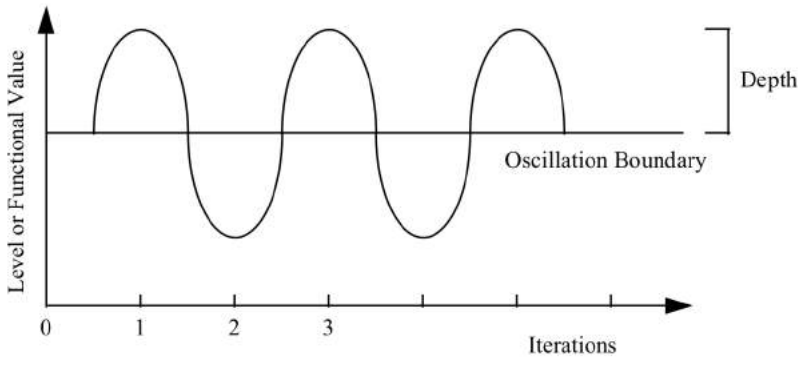


Figure 4.12: Strategic oscillation

The process of repeatedly approaching and crossing the critical level from different directions creates an oscillatory behavior, which gives the method its name. Control over this behavior is established by generating modified evaluations and rules of movement, depending on the region navigated and the direction of search. The possibility of retracing a prior trajectory is avoided by standard tabu search

mechanisms, like those established by the recency-based and frequency-based memory functions.

4.3.3 Path Relinking

Path relinking generally operates by starting from an initiating solution, selected from a subset of high quality (or elite) solutions, and generating a path in the neighborhood space that leads toward the other solutions in the subset, which are called *guiding solutions* (Fig. 4.13). This is accomplished by selecting moves that introduce attributes contained in the guiding solutions.

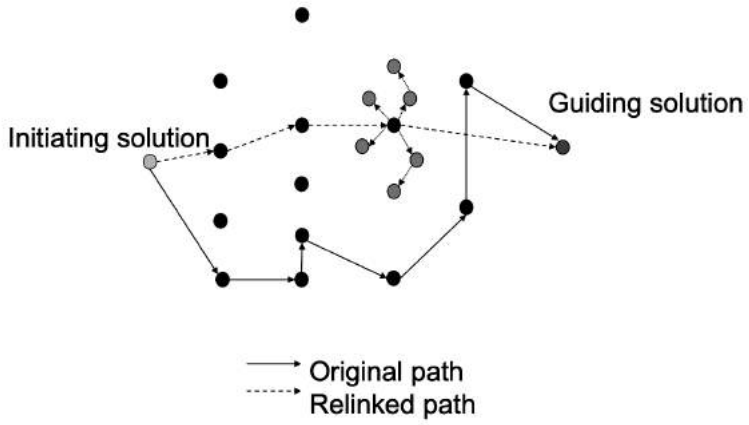


Figure 4.13: Path Relinking