Heuristic Optimisation Lecture Notes

Sándor Zoltán Németh
School of Mathematics, University of Birmingham
Watson Building, Room 324
Email: s.nemeth@bham.ac.uk

Overview

- Module structure
- Topics
- Difficult problems

Module structure

- There will be two lectures per week and one example class each fortnight.
- **Office hours:** Will be posted on CANVAS.
- **Online material:** All material is on CANVAS.

Topics

1. Basic difficulties in problem solving. The need for heuristic optimization
2. Basic concepts: representation, optima, neighbourhood
3. Exhaustive search and local search
4. Classification of Algorithms
5. Greedy algorithm
6. Divide and Conquer
7. A* search.
8. Simulated annealing
9. Tabu search
10. Evolutionary algorithms
11. ...

Recommended books

- Z. Michalewicz, D.B. Fogel: How to solve it: Modern heuristics, Springer, Corrected Third Printing 2002, ISBN 3-540-66061-5. (core text)
- S. Russel, P. Norvig: Artificial Intelligence: A modern approach, Prentice Hall, Second edition, 2002, ISBN 0137903952. (10%)
- Other literature, recommended in the notes

Contents

1 Introduction

1.1 What this course is and is not about

- It is about traditional and modern heuristic optimization methods for problem solving.
- It is not about telling what the ultimate method to use is!

In general when we solve a problem, the most difficult step is how to start the solution. During your studies you mostly encounter problems which are at the end of a chapter in a book or lecture notes and you already know which tools to use to solve them. But real life problems are not at the end of a chapter and usually nobody tells you the necessary tools to handle them. Let us assume for example that these problems will be given to you by your future manager, who does not have the time or expertise to guide you. He probably hired you because he is relying on your expertise (at least this is often the case), that is, on your skill to find the necessary tools to solve a problem for him. A problem is much more difficult if you don't know what are the tools to be used for solving it.

Consider for example the following puzzle:

Example Puzzle. There is a well, open at the top, with diameter $D = 3m$. We throw two sticks of lengths $4m$ and $5m$ into the well. The sticks stay in a plane and cross each other. The problem is to find the distance h from the bottom of the well to the point where they cross.

This problem has been solved in one hour or less by one percent of the people tested by Z. Michalewicz and D. B. Fogel, all of whom had an undergraduate degree either in mathematics, engineering or computer science. Can you do better? Can you find the necessary tools to solve this problem?

This course provides you with several traditional and modern heuristic methods to solve difficult problems, so that when you encounter a real life problem you can make a good choice. I hope you all agree that some problems (actually many problems) are difficult to solve and therefore, let us try to summarise what are the main difficulties when you try to solve them.

Why are some problems difficult to solve?

- The size of the search space is **huge**.
- The problem is very complicated, the solutions to a simplified model are useless.
- The evaluation function varies with time, so a set of solutions is required.
- There are heavy constraints, it is hard to even construct a feasible solution.
- The person solving the problem is not prepared properly.

Next let us consider these difficulties in some more details.

1.2 The size of the search space

1.2.1 The Boolean Satisfiability Problem (SAT)

The task is to make a compound statement of Boolean variables, i.e., a *Boolean function*

$$F : \{0,1\}^n \to \{0,1\}$$

evaluate to TRUE. The operations to construct such a function are NOT (denoted by an overline and called **negation**), AND (denoted by $\wedge$ and called **conjunction**) and OR (denoted by $\vee$ and called **disjunction**). The operation tables for these operations are:

x	0	1
$\overline{x}$	1	0

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

with the most usual way of representing the truth values, that is, FALSE= 0, TRUE= 1 and x a variable. Then, the NOT operation for this representation can be thought as

$$\overline{x} = 1 - x;$$

the AND as

$$F \wedge G = \min(F, G) = FG;$$

and the OR as

$$F \vee G = \max(F, G);$$

where F, G are Boolean functions, or in particular Boolean variables. Let $F_1, \ldots, F_n$ be Boolean functions, or in particular Boolean variables. Then, the Boolean function

$$F = F_1 \vee F_2 \vee \cdots \vee F_n$$

is TRUE if **at least one** of $F_1, \ldots, F_n$ is TRUE and its is FALSE if all $F_1, \ldots, F_n$ are FALSE. The Boolean function

$$F = F_1 \wedge F_2 \wedge \cdots \wedge F_n$$

is FALSE if **at least one** of $F_1, \ldots, F_n$ is FALSE and its is TRUE if all $F_1, \ldots, F_n$ are TRUE. You can easily remember the above operation table and rules if you think about how the max and min functions take their values.

The function

$$F(x_1, x_2, \ldots, x_6) = \left(x_1 \bigvee \overline{x}_3 \bigvee x_5\right) \bigwedge \left(\overline{x}_2 \bigvee x_4 \bigvee x_6\right)$$

is an example of a Boolean function. If $x_1 = x_2 = x_3 = x_4 = x_5 = 0$, then $F(x_1, x_2, \ldots, x_6) = 1$. However, if $x_1 = x_4 = x_5 = x_6 = 0$ and $x_2 = x_3 = 1$, then $F(x_1, x_2, \ldots, x_6) = 0$. Can you tell what happens if all variables are equal to one (that is, TRUE)?

For an arbitrary $n > 0$ positive integer the size of the search space is 2^n, because the variables take the values TRUE or FALSE completely independently.

For example, for $n = 100$ the size of the search space is $2^{100} \approx 10^{30}$ (which is a huge number)!

1.2.2 The Traveling Salesperson Problem (TSP) (also known as Travelling Salesman Problem)

The travelling salesperson must visit every city in his territory exactly once and return to his hometown covering the lowest cost (shortest distance / shortest travelling time / using the least fuel).

Given the cost of travel between all pairs of cities, the itinerary for the minimum cost tour is sought.

A TSP is called symmetric if the cost of getting from any city A to any city B is assumed to be the same as the cost of getting from the city B to city A (for example this holds if the "cost" is the "distance").

For a symmetric TSP, the size of the search space for n cities is $n!/(2n) = (n-1)!/2$. Indeed, you can have $n!$ itineraries (tours). However the cost of your tours are the same regardless of from which city do you start and which direction you are going.

For example:

n=6	$Size(\text{SAT}) = 64$	$Size(\text{TSP}) = 60$
n=7	$Size(\text{SAT}) = 128$	$Size(\text{TSP}) = 360$
n=10	$Size(\text{SAT}) = 1\,024$	$Size(\text{TSP}) = 181\,440$
n=20	$Size(\text{SAT}) = 1\,048\,576$	$Size(\text{TSP}) = 60\,822\,550\,204\,416\,000$

1.2.3 The Nonlinear Programming Problem (NLP)

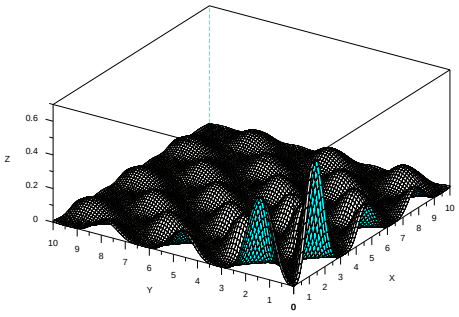
It is the problem of maximising an objective function subject to several constraint functions, such that at least one of these functions is not linear.

For example consider the following test problem:

Maximize the function

$$G_2(x) = \left| \frac{\sum_{i=1}^{n} cos^4(x_i) - 2\prod_{i=1}^{n} cos^2(x_i)}{\sqrt{\sum_{i=1}^{n} i x_i^2}} \right|$$

subject to
$\prod_{i=1}^{n} x_i \geq 0.75$,
$\sum_{i=1}^{n} x_i \leq 7.5n$
with bounds $0 \leq x_i \leq 10$
for $1 \leq i \leq n$.



(The test function G_2 is denoted by $G2$ in the literature, we slightly changed the notation to avoid any ambiguity.) The size of this problem is of course ∞, but a computer cannot work with arbitrary precision. Hence, if each variable has six digit precision the size of the search space (considering only the bounds for the variables and ignoring the other constraints, this is also called infeasible search space) is 10^{7n}, because the variables take the values independently and the digits of each variable are independent (one digit before the decimal point and 6 after).

1.3 Choosing the model for a problem

Solving a real world problem consists of two separate steps (1) creating a model for the problem and (2) using that model to generate a solution:

$$\text{Problem} \Rightarrow \text{Model} \Rightarrow \text{Solution}.$$

There are two separate modelling approaches:

Simplify the model and use a traditional optimizer:

$$\text{Problem} \Rightarrow \text{Model}_{\text{approx}} \Rightarrow \text{Solution}_{\text{prec}}(\text{Model}_{\text{approx}}).$$

This approach has the advantage that it can be solved exactly with a traditional optimization method, but the solution of a simplified model could be far from a solution of the original problem and become useless in case of over-simplification.

Keep the exact model and use a non-traditional optimizer to find a near-optimal solution:

$$Problem \Rightarrow Model_{prec} \Rightarrow Solution_{approx}(Model_{prec}).$$

This approach has the advantage that it keeps the original complexity of the problem and even if the solutions are not perfect they can be close to a perfect (global) solution. There are however unfortunate circumstances when the found solution is far from a global solution, but these circumstances are rare. This latter approach is used in Heuristic Optimisation.

1.4 Constraints

The NLP problem:

We need the optimum within the **feasible** region. If there are complex relationships between the variables (that is, highly nonlinear constraints), then even finding a feasible solution is difficult, not to speak about solving the problem itself.

Moreover, the difficulty of the problem can be further increased by classifying constraints into *hard constraints* and *soft constraints*. **Hard constraints must** be satisfied, but **soft constraints** are only **desirable**.
For example consider **timetabling** of modules:

Let us list some hard constraints for this problem:

- A student cannot have two lectures at the same time.
- A lecturer cannot have two lectures at the same time.
- There cannot be two different lectures in the same room.
- The lecture room size cannot be larger than the number of the students in the class.
- ...

Let us also list some soft constraints:

- If there are 3 lectures/week for a module, they would be preferred on Monday, Wednesday and Friday.
- The number of students should not be much smaller than the number of seats in the lecture room.
- Students should not have too many lectures in the same day.
- Lectures time should not be too late in the day.
- ....

1.5 Change in time

Consider the TSP problem and let the cost be the travelling time between cities.

The time of possible routes may change due to

- rush hours,
- weather conditions,
- deteriorating road conditions,
- accidents,
- traffic lights,
- a train crossing your path,
- etc.

For example a travel from city A to city B normally takes 40 minutes, but can be 60 minutes in the worst case.

Shall we use the expected time of 50 minutes in our calculations? Probably not, because if you use such averages along each edge on your tour the compound error could make your estimated time of the tour very far from its actual time.

Therefore, rather than dealing with one solution you should deal with a set of solutions.

1.6 Heuristic optimization

The original Greek word means "I found it".

We **use** some **information** available about the problem to

- **reduce** the region(s) of the **search space** to be checked
- or to **speed up the search**
- at the cost of **not guaranteeing the optimum**.

Summary

- Complex problems have a huge number of possible solutions
- We often need to simplify the problem, but then we might obtain useless solutions
- The problem's conditions can change in time.
- Real-world problems may have constraints, so generating **feasible** solutions is hard already!

Recommended reading

Z. Michalewicz & D.B. Fogel, How to Solve It: Modern Heuristics, Chapter 1. Why are some problems difficult to solve?

2 Basic concepts

Overview

- Representation
- Objective
- Evaluation function
- Optimisation problem definition
- Neighbourhoods and local optima
- The hill-climbing method

2.1 Introduction

Consider the two stage modelling approach for solving a problem:

$$\text{Problem} \Rightarrow \text{Model} \Rightarrow \text{Solution}.$$

Regardless of what kind of algorithm we use to solve a problem the basic concepts are the same:

- The representation - encodes the alternative solutions for further manipulation
- The objective - the purpose to be achieved (may or may not be minimising one function)

- The evaluation function - tells us how good a solution (given the representation) is (by assigning to it a "goodness" number), or at least how good is compared to the other solution (in this case there is no explicit number assigned to the solution).

2.2 Representation

Consider the three model problems SAT, TSP and NLP for examining possible representations.

The Boolean Satisfiability Problem (SAT)

The most usual representation or encoding of a possible solution is a string formed from binary variables 0 or 1 (or simply bit string) of length n. We can think about this as assigning 0 and 1 to each of the n variables independently. Therefore, the search space has size 2^n, an each point in it corresponds to a feasible solution.

The Travelling Salesperson Problem (TSP)

A possible solution can be encoded as a permutation of natural numbers $1, 2, \ldots, n$, with each number corresponding to one city. Although there are $n!$ possible tours, if we assume that the TSP is symmetric than this number needs to be divided first by n, because the cost of the tour is the same regardless from which point it is started and then by 2, because TSP being symmetric it doesn't matter in which direction (clockwise or anticlockwise) the salesperson is travelling along the tour. Hence, the search size for a symmetric TSP is $n!/2n = (n-1)!/2$.

The Nonlinear Programming Problem (NLP)

The search space is formed by all real numbers in n dimensions, thus the size of the search space is infinite. However, computers can work with a finite digit precision. Therefore, consider for example an approximate representation of your variables with six digit precision. Assuming that your n variables take values from the interval $[0, 10]$ independently and the 7 digits of your variables (6 after the decimal point and one before) take integer values from 0 to 9 independently, the size of the search space is $(10^7)^n = 10^{7n}$.

The Size of the Search Space

The size of the search space is determined by the representation (that is, the possible encoding of your solutions) and its corresponding interpretation. The problem itself does not determine the search space size. Choosing the **right** representation is of utmost importance! Otherwise, a wrong choice of the domain of search can either lead to several unfeasible solutions or duplicate solutions (which slows down your search), or prevent you to from being able to find the right answer at all.

For example, consider the following puzzle:

Example Puzzle

Construct four equilateral triangles from six matches of equal size, such that the length of the sides of the triangles are equal to the length of a match.

If you start putting your matches on the table, then it is easy to form two equilateral triangles, by using five matches. Now, how on earth would you use the sixth match to form two more equilateral triangles.

So, what's wrong?

Who told you that the matches need to be on the same table? Maybe your search domain is wrong. Can you help me by using a different search domain? The solution is in the bracket written from right to left if you are really stuck (niamod hcraes lanoisnemid eerht a ni nordehartet raluger a si noitulos eht).

2.3 The Objective

Once you encoded your possible solutions, that is, you defined your search space, you have to decide what you want to achieve. The objective is the mathematical statement of the task. It is an expression rather than a function.

The Boolean Satisfiability Problem (SAT)

For this problem the objective is to **find the bit vector for which the Boolean statement evaluates to TRUE.**

The Travelling Salesperson Problem (TSP)

For this problem the objective is to **minimise the total distance, when visiting each city exactly once and returning to the starting city** which can be shortly written as the following mathematical expression:

$$\min \sum dist(x, y),$$

where the sum is taken along a tour ((x, y) are the edges of the tour) and the minimum is considered with respect to each possible tour.

The Nonlinear Programming Problem

For this problem the objective is to **minimise or maximise a nonlinear function** of feasible solutions.

2.4 The Evaluation Function

It is different from the objective. In general is a function from the set of possible encoded solutions (that is, feasible solutions under the given representation) to a set of numbers (for example reals). The numeric value indicates the quality of the solution. the reals). However, sometimes it is sufficient to know whether one solution is better than the other, without knowing how much better or the absolute value on some scale. For example this may be the case when you are choosing candidates for a job interview.

The Travelling Salesperson Problem (TSP)

For this problem there is an obvious exact (not comparative) evaluation function: **the sum of distances along the route**.

The Nonlinear Programming Problem (NLP)

For this problem the evaluation function is **the value of the function of feasible solutions to be minimised or maximised** which is exact.

How to choose the evaluation function?

In a real-world problem you need to choose what the evaluation function is. The evaluation function is **not** given with the problem, only the objective is. Often the objective suggests how to choose the evaluation function, but sometimes is not clear how to assign an evaluation function for a given objective. We could see above that TSP and NLP suggest the evaluation function, but things are not clear at all for SAT. In this case the objective is to make a Boolean function TRUE. However, the value of the Boolean function is not a good evaluation function, because it does not indicate how to improve your solutions. If improvement means solely getting from FALSE to TRUE, then you will not be able to design a good evaluation function. You have to be more clever than that. We will see later on this course possibilities for evaluation functions of SAT. Regardless of how you choose your evaluation function, the following remarks are useful when you design your evaluation functions:

- A solution that meets the objective should also have the best evaluation.
- A solution that fails to meet the objective cannot be evaluated to be better than a solution that meets the objective.
- The objective may suggest the evaluation function (TSP, NLP) or may not (SAT).
- The feasible region of the search space must be considered.

2.5 Defining the optimisation problem

For us the "search problem" and "optimisation problem" mean the same thing. Even if the "optimisation nature" of the problem is not clear (for example SAT), we are still aiming to solve the problem by using a "search algorithm" which is minimising or maximising something.
By an **optimisation problem** we mean the following problem:
Given a search space S together with its feasible part $F \subseteq S$, find $x \in F$ such that

$eval(x) \leq eval(y)$, $\forall y \in F$ (minimisation)

Such an x is called a **global solution** (with respect to the evaluation function $eval$).

2.6 Reasons for using heuristics

We use heuristics because:

- We can avoid combinatorial explosion,
- We rarely need the optimal solution, good approximations are usually sufficient,
- The approximations may not be very good in the worst case, but in reality worst cases occur very rarely.
- Trying to understand why a heuristic works/does not work leads to a better understanding of the problem

2.7 Neighbourhoods

For an optimisation problem in one hand we want to get arbitrarily "close" (i.e., converge) to a "good" solution and on the other hand we want to limit the "cost" (that is, number of steps, time etc.) of our search. Roughly speaking a "neighbourhood" contains solutions which are "close" to a given solution, where closedness can be defined by using a distance or a mapping which shows the "degree of similarity" of two solutions. Let us formulate the above ideas more precisely, in mathematical terms.
Let S be our search space. Define a distance function $dist : S \times S \to \mathbb{R}$. This distance function is usually nonnegative and satisfies some axioms such as

- $d(x, x) = 0$, that is the distance of a point from itself is zero.
- $d(x, y) = d(y, x)$, this property is called **symmetry**. Please note that this property does not hold for non-symmetric TSP.
- $d(x, z) \leq d(x, y) + d(y, z)$, this property is called **triangle inequality**. If the distance means a "cost" rather then a physical distance, then this property may not hold. For example the distance can mean the *amount of fuel* or *time* used by going from a city to another one by a car.

In any case, having defined a distance function d, one can define the neighbourhood for a given $\varepsilon > 0$ such as

$$N(x) = \{y \in S : dist(x, y) \leq \varepsilon\}.$$

Some examples:

- NLP: Euclidean distance: $dist(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$.
- NLP: Manhattan distance: $dist(x, y) = \sum_{i=1}^{n} |x_i - y_i|$, where $|\cdot|$ denotes the modulus or absolute value function (this distance will be frequently used later for other type of problems too, such as the Maze Problem and the 8 Puzzle Problem).
- SAT: Hamming distance (number of bit positions with different truth assignments).

The Neighbourhood as a Mapping

Alternatively the neighbourhood can be defined as a mapping $m : S \to 2^S$, where 2^S denotes the set of subsets of S. The neigbourhood of x is given by $m(x)$.

Some examples:

- TSP: The **2-swap** mapping generates for any potential solution x the set of potential solutions obtained by swapping two cities in x.
- TSP: The **2-interchange** mapping generates for any potential solution x the set of potential solutions obtained by replacing two non-adjacent edges AB and CD with AC and BD.
- SAT: The **1-flip** mapping flips exactly one bit of a potential solution x (with the representation of x as a Boolean string with string entries 0 or 1).

2.8 Local optima

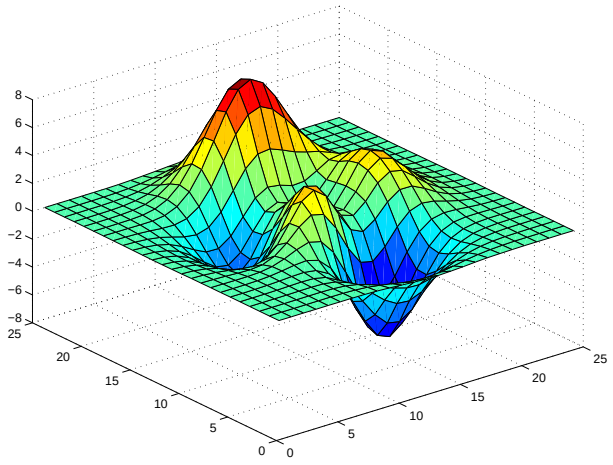
A potential solution $x \in F$ is a **local optimum** with respect to neighbourhood $N(x)$ if

$$eval(x) \leq eval(y), \ \forall y \in N(x).$$

Local search methods operate on neighbourhoods:

They try to modify the current solution x into a better one within its neighbourhood $N(x)$. We will consider local search in more details later.

2.9 Hill-climbing



Hill-climbing analogy

Consider the following analogy between **hill-climbing** and the problem of **orientation and moving on a surface** :

- **Height**: the quality of a node (the evaluation function).
- **Peaks**: optimal solutions.
- **Orientation**: evaluating neighbouring positions.

2.9.1 Basic hill-climbing algorithm

1. Evaluate the initial point x. If the objective is met, return x. Otherwise set x as the **current** point.
2. Loop until a solution is found:
   (a) Generate all the neighbours of x. Select the best one (or randomly one of the best ones) y.
   (b) If x is at least as good as y, then return x,

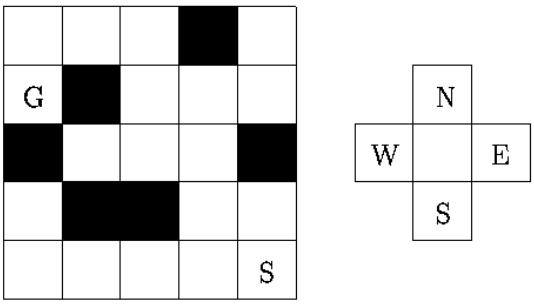
   else set y as the current point.

For better efficiency, the algorithm can be restarted a predefined number of times (iterative hill-climbing).

2.9.2 Examples of hill-climbing

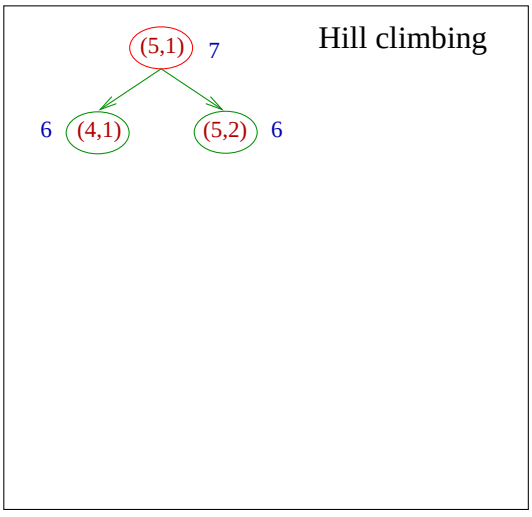
Example of hill-climbing for the maze problem

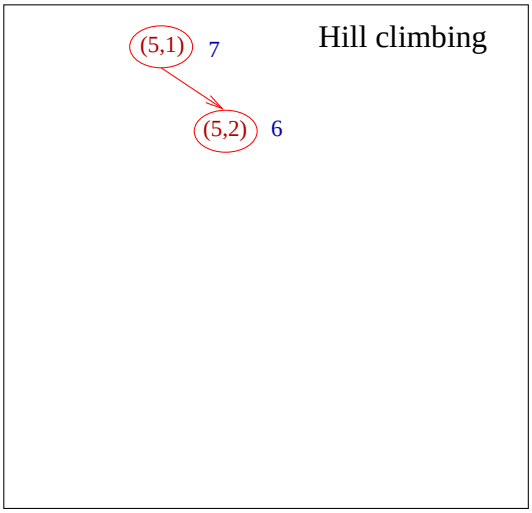
Consider the following maze:

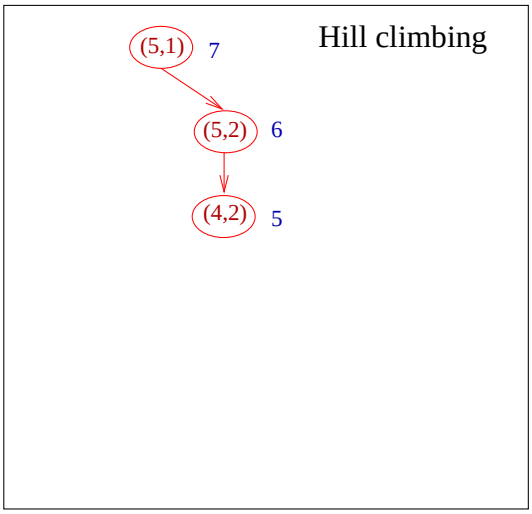


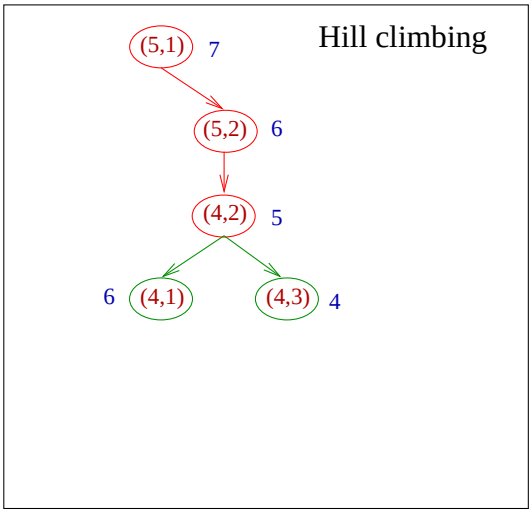
The problem is to get from the start node S to the goal node G, by moving horizontally and vertically and avoiding the black obstacles in the above maze. For this problem there is a way to get from S to G. Please note, that for a different maze problem there may not be a way to get from S to G. In this case the problem will be to get as close as possible to G with respect to the Manhattan distance. If we formulate the Maze Problem in this way, then we don't care about the length of our path from S to G, just about getting to G, or getting as close as possible to G. Later graph search approaches, based on the greedy algorithm or A* will have a completely different goal, namely, finding the shortest path from S to G.

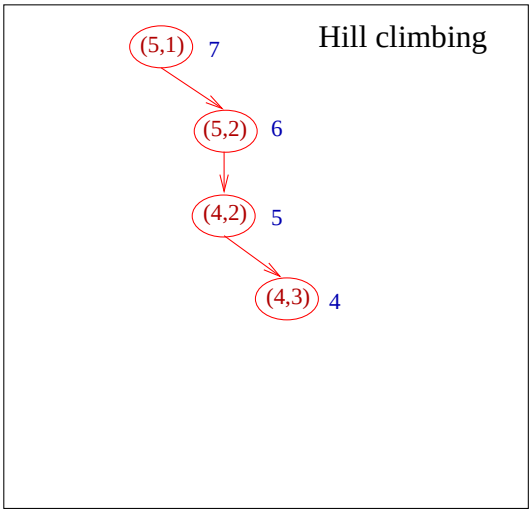
We will solve this problem by using the Basic hill-climbing algorithm. In the below figures describing the steps of the algorithm each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate). Recall that the Manhattan distance between the points (x, y) and (z, t) is given by $d((x, y), (z, t)) = |x - z| + |y - t|$, where $|\cdot|$ denotes the absolute value (or modulus) function. At each step the algorithm examines the neighbours of the current point x and chooses (randomly) one with the lowest Manhattan distance. If the chosen point is better than x, then it becomes the current point and the algorithm continues. If not, then the algoritms returns x. The steps of the algorithm are described below:

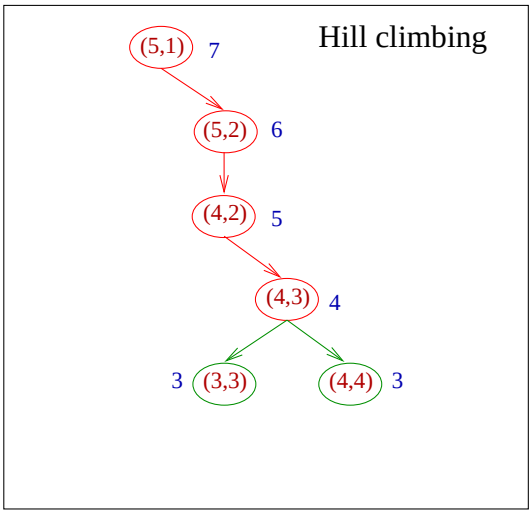


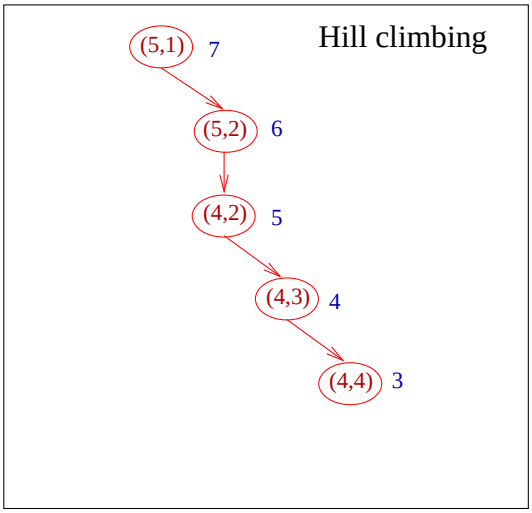

Hill climbing
(5,1)
7
(5,2)
6

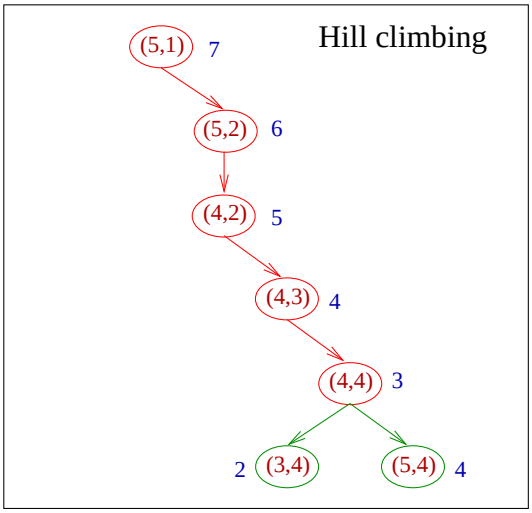

Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5

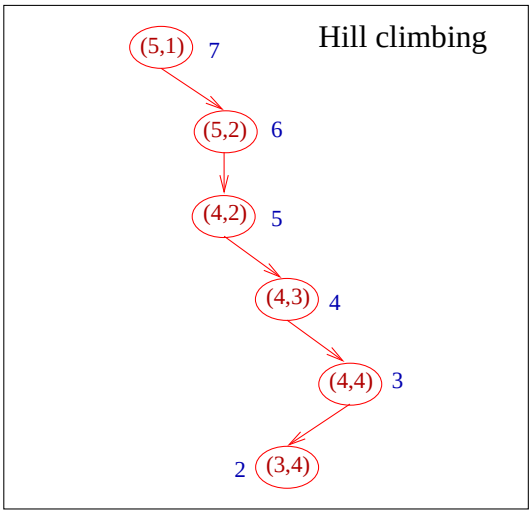

Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5
6
(4,1)
(4,3)
4


Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5
(4,3)
4


Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5
(4,3)
4
3
(3,3)
(4,4)
3


Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5
(4,3)
4
(4,4)
3





The algorithm is stuck in (3,4) because all neigbours of (3,4) are worse than (3,4) (i.e., they have a higher Manhattan distance).

Example of hill-climbing for finding the minimum of a quadratic function

Problem: Consider the function $f(x) = x^2+3\times x+5$ defined on integer numbers in the interval $[-20, 20]$. Use the hill-climbing algorithm to find the function's minimum value on this interval.

First define the search space, then the neighbourhood. Try to find the minimum using starting point $x = 0$. Then try using starting point $x = -6$.

Solution:

The search space is the set $\{-20, -19, ..., -1, 0, 1, 2, ..., 19, 20\}$.

If we consider $\varepsilon = 1$, the neighbourhood for $x = 0$ is $\{-1, 0, 1\}$, for -1, $\{-2, -1, 0\}$ etc. In general for an x, the neighbourhood is $\{x - 1, x, x + 1\}$.

$f(0) = 5$, $f(-1) = 3$, $f(1) = 9$, as we are looking for the minimum, hill climbing will select $x = -1$

$f(-1) = 3$, $f(-2) = 3$, $f(0) = 5$, so the algorithm will stop and give the answer $x = -1$, with the minimum value $f(-1) = 3$

Starting from $x = -6$, $f(-6) = 23$, $f(-7) = 33$, $f(-5) = 15$, so select $x = -5$.

$f(-5) = 15$, $f(-6) = 23$, $f(-4) = 9$, so $x = -4$ is selected.

$f(-4) = 9$, $f(-5) = 15$, $f(-3) = 5$, so $x = -3$ is selected.

$f(-3) = 5$, $f(-4) = 9$, $f(-2) = 3$, so $x = -2$ is selected.

$f(-2) = 3$, $f(-3) = 5$, $f(-1) = 3$, so $x = -2$ is returned as solution.

2.9.3 Properties of hill-climbing

Hill-climbing can be used for both maximisation and minimisation problems. Suppose that we have a maximisation problem. Then, hill-climbing has the following properties:

- It can get stuck in **local maxima**. In this case you need to backtrack (implicitly this means a violation of the hasic hill climbing approach, but your ultimate goal is to find the solution, so you need to be creative, without rigidly following a given approach).
- **Plateaus** (areas of search space where the evaluation function is flat, that is, it has almost the same values) are a problem. Then you need to make big jumps (another violation, same comments hold as above).
- There is no information about how far the found local optimum from the global optimum is.
- The solution depends on the initial configuration (i.e., starting point).
- Finding an upper bound for the computational time is in general not possible.

Summary

- Choosing the right representation is essential for success.
- The evaluation function must also be carefully chosen.
- **Effective** search balances **exploitation** of the best solutions so far and **exploration** of the search space.
- **Hill-climbing** is an easy local search method that is good at exploitation but neglects exploration.
- **Random search** is good at exploration, but does no exploitation.

Recommended reading

Z. Michalewicz & D.B. Fogel, How to Solve It: Modern Heuristics, Chapter 2. Basic concepts

3 Classification of classic algorithms

Overview

- Classification of classic algorithms
- Algorithms on complete solutions
- Algorithms on partial solutions

There is a large number of classical algorithms to solve a problem. It is natural to ask, why do we need so many algorithms? Why not choose a good one to solve the problem. Unfortunately, things are more complicated, because none of these algorithms is robust. If you change the problem, then the previously good algorithm may become unsuitable to handle your problem, that is, you need to choose another one. This new algorithm, which may not have been good previously, but it became good after modifying the problem. Summarising: Classic optimisation methods can be very effective if appropriate to a specific task.

Having in mind the large number of algorithms it is natural to try to classify them. This classification may suggest which one to use. Classical algorithms fall into two classes:

- Algorithms that only evaluate complete solutions
- Algorithms that evaluate partially constructed or approximate solutions

3.1 Algorithms on complete solutions

Algorithms on complete solutions can be stopped at any time to get a solution which you can try. The solution may not be good, but at least you have something, a potential solution which can be easily compared to another potential solution by using an evaluation function.

A **complete solution** means that all decision variables are specified. An **algorithm on complete solutions** is an algorithm which manipulates only complete solutions.

Please note that there is a difference between a "complete solution" and an "algorithm on complete solutions". The latter is an optimisation method which manipulates only "complete solutions". The next examples are examples for "complete solutions" for given problems and NOT "algorithms on complete solutions" for solving these problems.

Examples of complete solutions:

- SAT: a binary string of length n
- TSP: a permutation of n cities
- NLP: a vector of n real numbers

Algorithms on complete solutions manipulate one complete solution at a time and when a better solution is found the previous solution is replaced.
The algorithmic description of this idea is the following:

1. Initialise **best** solution.
2. Generate a solution x according to the specifics of the algorithm.
3. If x is better than **best**, replace **best** by x.
4. Repeat steps 2-3.

The next examples are examples for "algorithms on complete solutions" (unrelated to specific problems) and NOT examples of "complete solutions" for specific problems.

Examples of algorithms on complete solutions:

- exhaustive search (considered later)

- local search (already mentioned but it will be considered in more details later)
- hill-climbing
- gradient-based optimisation methods
- simulated annealing (considered later)
- tabu search (considered later)
- genetic algorithms (considered later)

3.2 Algorithms on partial solutions

The algorithms on partial solutions are of two types:

- Algorithms which manipulate **incomplete** solutions to the **original** problem.

  An incomplete solution to the original problem is a subset of the original problem's search space with a particular property hopefully shared by the real solution.

- Algorithms which manipulate **complete** solutions to a **reduced** problem, in the sense that:

  1. They decompose the original problem into smaller & simpler problems,
  2. They solve these problems,
  3. They try to combine the partial solutions into a solution to the original problem.

The next examples are examples of "algorithms on partial solutions" (not related to a specific problem) and not examples of "incomplete solutions" for specific problems.

Examples of algorithms on partial solutions:

- greedy search (considered later) – manipulating incomplete solutions of the original problem
- divide and conquer (considered later) – manipulating complete solutions to a reduced problem
- A* algorithm (considered later) – manipulating incomplete solutions of the original problem

The next examples are examples of "incomplete solutions" for specific problems and NOT examples of "algorithms manipulating incomplete solutions of the original problem".

Examples of incomplete solutions to the original problem:

- SAT: all binary strings of length n that start with 11
- TSP: all permutations that start with $7 - 5 - 11$
- NLP: all solutions having $x_2 = 3.4567$

A Boolean Satisfiability Problem (SAT) F is in **conjunctive normal form** if $F = F_1 \wedge F_2 \wedge \cdots \wedge F_n$, where $F_1, \ldots, F_n$ are formed by using negations and/or disjunctions. $F_1, \ldots, F_n$ are called **clauses** or **conjuncts**.

Examples of complete solutions to a reduced problem:

- SAT: if the formula is in conjunctive normal form, try to solve the conjuncts (i.e., make them TRUE).
- TSP: consider only $k < n$ cities and try to find the shortest route $i \to j$ (i.e., from i to j) passing through all k cities.
- NLP: limit the domain of some variables and search for a solution in the restricted domain.

3.2.1 Problems with partial solutions

(1) It is difficult to devise a way for organising the subspaces of the search space to assure effective search.

(2) It is difficult to devise a new evaluation function that can assess the quality of partial solutions.

For task (1) you may think of organising the search space into a tree, where complete solutions are on the leaves of the tree.

Task (2) is very challenging for real-world problems. For example it is difficult to assess how good a remote control is if you have only partial information about it. You may know the brand of the remote control, but having a famous brand compared to a noname one does not necessarily mean that your remote control is better, because you don't know if it is a remote control for a blue ray player or a TV set or your hi-fi music system etc.

Going back to task (1), organising the search space depends on the representation of the problem and there are different ways of encoding your potential solutions. We have already seen the most common representations for SAT, TSP and NLP: binary string, permutations, vectors of decimal numbers, respectively. Let us, see some other representations:

SAT:

Represent the potential solutions as vectors of real numbers in the range $[-1, 1]$.

Encode the truth values of your variables as follows:

$x_i \geq 0$ corresponds to TRUE,

$x_i < 0$ corresponds to FALSE.

NLP:

Represent the potential solutions as binary strings of length kn, where each variable $x_i \in [l_i, u_i]$ is encoded as a binary string of length k.

You can convert a binary string $\langle b_{k-1} \dots b_0 \rangle$ into a decimal value x_i from some interval $[l_i, u_i]$ as follows:

1. First convert the binary string $\langle b_{k-1} \dots b_0 \rangle$ from base 2 to base 10:

$$(\langle b_{k-1} \dots b_0 \rangle)_2 = (\sum_{j=0}^{k-1} b_j \cdot 2^j)_{10} = x'.$$

2. Next find a corresponding decimal value in the required interval:

$$x_i = l_i + x' \cdot \frac{u_i - l_i}{2^k - 1}.$$

For example if the interval is $[-2, 3]$ and the string is $\langle 1001010001 \rangle$, then the string converts in base 10 into

$$2^9 + 2^6 + 2^4 + 2^0 = 593$$

and therefore the mapping is

$$0.89833822 = -2 + 593 \cdot \frac{5}{1023}.$$

Recommended reading

Z. Michalewicz & D.B. Fogel, How to Solve It: Modern Heuristics, Chapter 3. Traditional methods - Part 1, pp. 55-58

4 Exhaustive Search

Overview

- Exhaustive search
- Examples: SAT, TSP, NLP

Description

Looks at every possible solution in the search space until a global optimum is found.

If the value of the global optimum is not known, then **all** points in the search space must be checked. This can be really **exhausting**. For example a symmetric TSP with 50 cities has search space size 10^{62}, which is a huge number. However, exhaustive search is still interesting for the following reasons:

- The algorithm is very simple.
- In order to use exhaustive search we have to generate every possible solution to the problem in a systematic way. This enumeration may provide a deeper understanding of the structure of your representation.
- There are ways of reducing the amount of work such as **backtracking**.

4.0.1 Backtracking

Backtracking reduces the amount of work:
If after assigning values to some variables it turns out that there is no solution that has those values, we do not continue assigning values to the remaining variables but we go back one step and try a different option.

4.1 Enumerating the SAT

We have to generate all binary strings of length n, from $< 00\ldots0 >$ to $< 11\ldots1 >$.

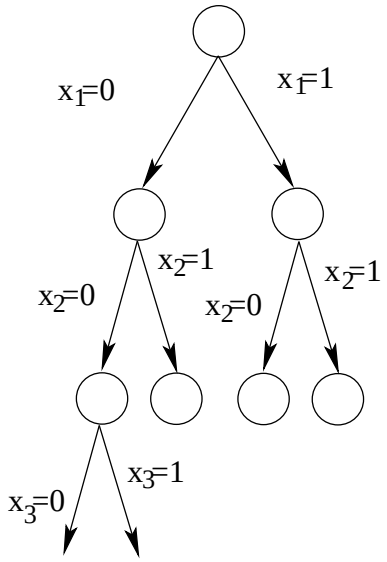
This is an easy task, we just have to generate all the integers from 0 to $2^n - 1$ and convert them into the matching binary string.

In order to see if you have got a solution or not use the following **evaluation function**:

- 1 if the Boolean statement is satisfied,
- 0 otherwise.

4.1.1 Partitioning the search space for SAT

Partition the search space for SAT into a tree:



Use depth-first search, which defines itself recursively, as follows:

1. Visit **current node**,
2. For each child of **current node** perform depth-first search.

You may end up in a dead end when there is no point to continue further. For example, supposet that your SAT formula, with at least 5 variables contain a clause

$$\ldots \bigwedge (x_1 \bigvee x_3) \bigwedge \ldots,$$

then if you assigned the values

$$x_1 = 0,\ x_2 = 0,\ x_3 = 0,$$

or

$$x_1 = 0,\ x_2 = 1,\ x_3 = 0.$$

Then, there is no point of continuing assigning values to your variables $x_4, x_5, \ldots$ because your problem will evaluate to FALSE regardless of what you assign to these variables. Therefore, you go back one step and assign $x_3 = 1$ and continue with another branch.

Example. Consider the following 6 dimensional Boolean Satisfiability problem:

$$F(x_1, x_2, x_3, x_4, x_5, x_6) = (\overline{x}_1 \bigvee x_2 \bigvee \overline{x}_3) \bigwedge (x_2 \bigvee x_6) \bigwedge (\overline{x}_3 \bigvee \overline{x}_6)).$$

Find all solutions of this problem by using exhaustive search.

Solution: The search space has size $2^6 = 64$. The values for x_4 and x_5 are not used for calculating $F(x_1, x_2, x_3, x_4, x_5, x_6)$, so we only need to look at the values of the remaining variables.
The truth table for the four variables (the X stands for "don't care" values):

x_1	x_2	x_3	x_6	$\overline{x}_1 \bigvee x_2 \bigvee \overline{x}_3$	$x_2 \bigvee x_6$	$\overline{x}_3 \bigvee \overline{x}_6$	F
0	*0*	*0*	*0*	*X*	*0*	*X*	*0*
0	*0*	*0*	*1*	*1*	*1*	*1*	*1*
0	*0*	*1*	*0*	*X*	*0*	*X*	*0*
0	*0*	*1*	*1*	*X*	*X*	*0*	*0*
0	*1*	*0*	*0*	*1*	*1*	*1*	*1*
0	*1*	*0*	*1*	*1*	*1*	*1*	*1*
0	*1*	*1*	*0*	*1*	*1*	*1*	*1*
0	*1*	*1*	*1*	*X*	*X*	*0*	*0*
1	*0*	*0*	*0*	*X*	*0*	*X*	*0*
1	*0*	*0*	*1*	*1*	*1*	*1*	*1*
1	*0*	*1*	*0*	*X*	*0*	*X*	*0*
1	*0*	*1*	*1*	*X*	*X*	*0*	*0*
1	*1*	*0*	*0*	*1*	*1*	*1*	*1*
1	*1*	*0*	*1*	*1*	*1*	*1*	*1*
1	*1*	*1*	*0*	*1*	*1*	*1*	*1*
1	*1*	*1*	*1*	*X*	*X*	*0*	*0*

Note that for each solution shown in the truth table (the lines where $F = 1$) we have 4 different solutions which all have the given values for variables x_1, x_2, x_3, x_6 and have the 4 different combinations for x_4 and x_5. Please also note that once you found a clause with truth value 0, then you automatically know that the value of F is false. Therefore, you don't need to check the values of the other clauses, that is, they have the "don't care value" X. Of course as humans we can observe if a clause is false, but computers need to check the clauses in a particular order, so for them the part of the table to be completed with zeros or ones will be larger.

4.2 Enumerating the TSP

Enumerating the potential solutions of TSP is not entirely straightforward. The main question is: How to generate all permutations of $1, 2, \ldots, n$? A possible solution is described below:

1. Fix 1 in the first position.
2. Generate all permutations of numbers $2, \ldots, n$ for positions $2, \ldots, n$.
3. Fix 1 in the second position.
4. Now list the permutations found in step 2. in positions $1, 3, 4, \ldots, n$.
5. Repeat with 1 fixed in all positions.

However, if you keep in mind what is the evaluation function for TSP, a more suitable enumeration for TSP is:

Enumerate by successively exchanging adjacent elements. Then the cost of evaluating the next solution is reduced to calculating the effect of the change on the previous solution.

But how can we deal with the situation when not all pairs of cities are connected?

The answer to this question is left to the reader.

4.3 Enumerating the NLP

Theoretically speaking, in case of NLP there is an infinite number of alternatives and of course you cannot enumerate an infinite set in a finite time (moreover, in this case the number of solutions is uncountable, so you cannot enumerate them at all!). However, you can use the following approximating scheme:

We can divide the continuous domain for each variable into a finite number of intervals.

In this way the search space becomes a set of cells

We can evaluate each cell as one point (a corner or the center).

With the best cell found, the whole procedure can be repeated (by dividing the found cell in a similar way).

Problems with enumerating NLP

- If you choose a **fine granularity** then you get a too large number of small cells. Your algorithm becomes **impractical** (i.e., too slow both in number of steps and time).
- If you choose a **coarse granularity**, then you have a small number of large cells.

  Therefore, the possibility of missing the best solution increases.

Recommended reading

Z. Michalewicz & D.B. Fogel, How to Solve It: Modern Heuristics, Chapter 3. Traditional methods - Part 1, pp. 58-64

5 Local search

Roughly speaking, *local search* and *exhaustive search* are at the "opposite ends of the spectrum", in the sense that:

- Exhaustive search enumerates all the possible solutions.
- Local search focuses on searching only a neighbourhood of some particular solution.

Next we will give a general description of the algoritm of local search.

1. Pick a solution from the search space. Define this as the **current** solution.
2. Apply a transformation to the current solution to obtain a **new** solution.
3. If the new solution is better than the current one, replace current solution by the new solution.
4. Repeat steps 2.-3. until no improvement is possible.

The transformation in step 2 can range from returning a **random** solution from the whole search space to returning the **current** solution. Returning a random solution is essentially enumerative. Moreover, it can be even worse than simple enumeration of the search space, because you might resample some previously considered solutions. On the other hand returning the current solution leads you nowhere, because no change is made which can improve your solution.

From a practical point of view the size of the neighbourhood should lie somewhere between the sizes of the neighbourhoods generated by the above extremal transformations. In other words, the size of the neigbourhood should be a tradeoff between a

- Too **Small** neighbourhood, when your algorithm is quick, but it might get stuck in a local optima (smaller the neighbourhood higher chances you have to get stuck in a local optima) and a
- Too **Large** neighbourhood, when the chances of getting stuck in a local optima, but the search might be too long (larger the neigbourhood more time it takes to search).

5.1 Local search for SAT

Consider the most common representation of SAT where potential solutions are Boolean strings (i.e., strings whose entries are 0 or 1).
In order to be able to compare two solutions for a SAT problem F one needs to define an appropriate evaluation function. Simply saying that $eval(F) = 0$ if F is TRUE and $eval(F) = 1$ if F is false is not good enough, because it does not indicate how to improve your solutions. Suppose that your SAT formula is given in conjunctive normal form, that is, it is a conjunction of clauses: $F = F_1 \wedge F_2 \wedge \cdots \wedge F_n$ where each clause F_i; $i = 1, \ldots, n$ is given by using disjunctions (i.e., OR) and negations (i.e., NOT) only (recall that this was the definition of a clause). By the previously given examples we could observe that satisfying the clauses is essential for our success, because they all need to be TRUE in order to get a solutions. This leads to the idea to define $eval(F)$ as the number of unsatisfied clauses. The idea is that, we need to decrease this number until no clause remains unsatisfied and we get a solution. Having defined the evaluation function, next we need to define a neigbourhood of a solution x. We define the neighbourhood of x as the set of Boolean strings defined by the 1-flip mapping, that is, the Boolean strings which differ from x in exactly one bit position. Now, we have all "ingredients" to define the GSAT local searh algorithm for SAT.

5.1.1 The GSAT algorithm:

1. Repeat steps 2.-3. MAX_TRIES times:
2. Assign values to $X =< x_1, \ldots, x_n >$ randomly
3. Repeat MAX_FLIPS times:

   If the formula is satisfied, return X

   else **make a flip**
4. Return "No solution found"

Make a flip

GSAT has two parameters MAX_TRIES which is the maximum number of searches (trials) and MAX_FLIPS which is the maximum number of flips (moves) per each search.

The SAT formula must be given in conjunctive normal form (a conjunction of clauses), for example:

$$\left(x_1 \bigvee x_4 \bigvee x_5\right) \bigwedge \left(\overline{x}_1 \bigvee x_2 \bigvee \overline{x}_6\right)$$

For each possible bit flip (from the current solution) the decrease in the number of unsatisfied (false) clauses is calculated.

The flip with the **largest** decrease (which can actually be an increase, in this case the largest decrease means -1 times the smallest increase) is made. (You can of course consider as evaluation function the the number of satisfied clauses and make the flip with the largest increase in the number of satisfied clauses (which can actually be a decrease, in this case the largest increase means -1 times the smallest decrease). If you prefer to choose the flip with the largest number of satisfied clauses, rather than counting decrease in the number of unsatisfied clauses, that's fine as well. However, in each case it is very important to explain your notations when you solve a problem!).

Example for GSAT

$F = F_1 \wedge F_2 \wedge F_3 \wedge F_4$

$F_1 = x_1 \bigvee x_2$, $F_2 = x_2$, $F_3 = \overline{x}_1 \bigvee \overline{x}_2$, $F_4 = x_3$

START: $(x_1, x_2, x_3) = (0, 0, 0)$

	F_1	F_2	F_3	F_4	
flip	0	0	1	0	decrease
x_1	1	0	1	0	$+1$
x_2	1	1	1	0	$+2$
x_3	0	0	1	1	$+1$

FLIP x_2

$(x_1, x_2, x_3) = (0, 1, 0)$

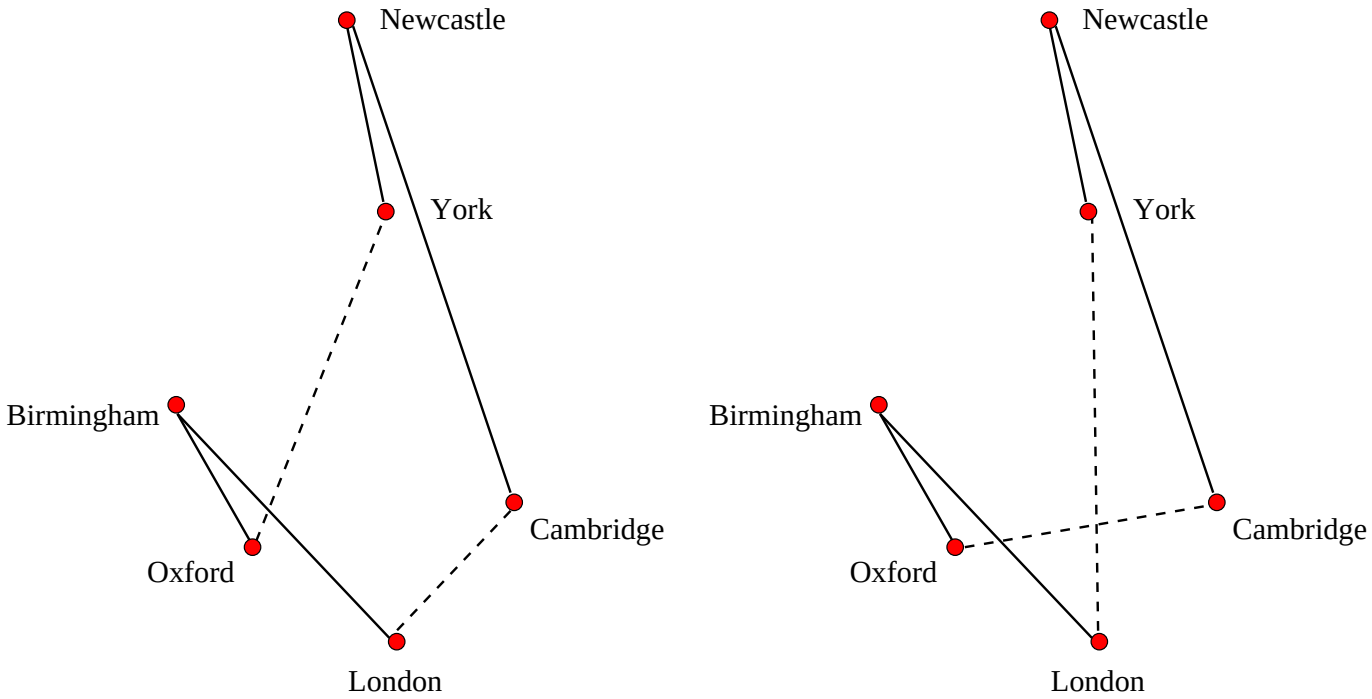
	F_1	F_2	F_3	F_4	
flip	1	1	1	0	decrease
x_1	1	1	0	0	-1
x_2	0	0	1	0	-2
x_3	1	1	1	1	$+1$

FLIP x_3

SOLUTION: $(x_1, x_2, x_3) = (0, 1, 1)$

5.2 Algorithm 2-opt for TSP

The neigbourhood is given by the 2-interchange transformation consisting of changing two non-adjacent edges in the tour, as shown in the next figure.



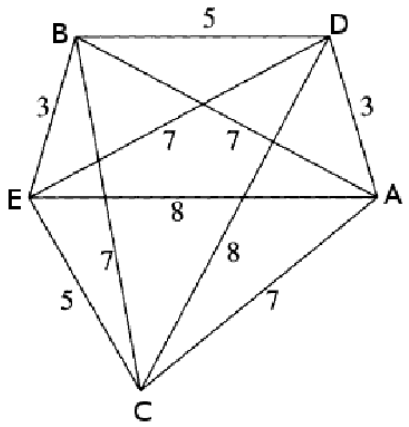
Recall the more precise definition of the 2-interchange operation. If AB and CD are two non-adjacent edges in a tour, by 2-interchange we mean replacing AB and CD with AC and BD and leaving all other edges unchanged.

The algorithm should be restarted several times for better results!

Local Search Example for TSP

Consider a symmetric travelling salesperson problem with 5 cities A, B, C, D, E. The distances between cities are as follows:

City	City	Distance
A	B	7
A	C	7
A	D	3
A	E	8
B	C	7
B	D	5
B	E	3
C	D	8
C	E	5
D	E	7



Solve the problem by using the 2-opt algorithm starting from the tour A-B-C-D-E-A.

Solution:

The cost of the starting tour is

$$\text{AB} + \text{BC} + \text{CD} + \text{DE} + \text{EA} = 7 + 7 + 8 + 7 + 8 = 37.$$

Replace the edges AB and CD by the edges DB and CA. The new tour will be D-B-C-A-E-D. The cost of the new tour is

$$\text{DB} + \text{BC} + \text{CA} + \text{AE} + \text{ED} = 5 + 7 + 7 + 8 + 7 = 34.$$

The new tour is better then the current starting solution (because it has lower cost). Replace the current solution by the new tour. The current solution becomes D-B-C-A-E-D. Its cost is 34. In the current solution replace the edges BC and ED by the edges EB and DC. The new tour will be D-C-A-E-B-D. The cost of the new tour is

$$\mathrm{DC} + \mathrm{CA} + \mathrm{AE} + \mathrm{EB} + \mathrm{BD} = 8 + 7 + 8 + 3 + 5 = 31.$$

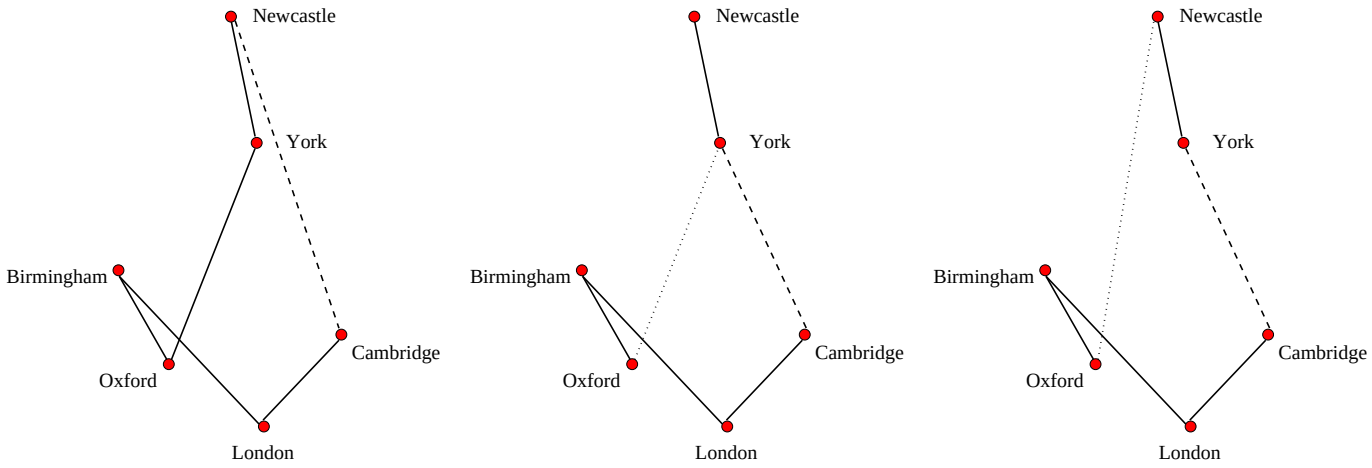
The new tour is better then the current solution (because it has lower cost). Replace the current solution by the new tour. The current solution becomes D-C-A-E-B-D. Its cost is 31. Replace the edges DC and AE by the edges AD and EC. The new tour will be A-D-B-E-C-A. The cost of the new tour is

$$\mathrm{AD} + \mathrm{DB} + \mathrm{BE} + \mathrm{EC} + \mathrm{CA} = 3 + 5 + 3 + 5 + 7 = 23.$$

No further improvements are possible. The tour A-D-B-E-C-A is optimal. Its cost 23 is minimal.

5.3 A δ-path for TSP

In a δ-path all nodes appear exactly once with the exception of the last one which appears twice. An example of δ-path is the middle graph in the next figure:



By replacing one edge a δ path can be

- repaired to a legal tour (see the left and right graphs in the figure above), or
- modified to another δ-path, called **switch** (for example, if you replace the Oxford-York edge with Oxford-Cambridge, in the middle graph of the figure above, then you obtain another δ-path).

5.3.1 The Lin-Kernighan algorithm for TSP

The Lin-Kernighan algorithm is an extension of the local search ide, where the algorithm is allowed to manipulate "illegal tours", more precisely δ-path as well. The steps of the algorithms are presented below:

1. Generate a tour T and store it as **best**.
2. For each node k on the tour T and for each edge (i, k):

   If there is a node j with $cost(i, j) \leq cost(i, k)$ then create the δ-path p where edge (i, j) replaces (i, k).

(a) Repair p into a tour T_1, store it, if better than **best**.

(b) If there is a switch from p to p_1 with $cost(p_1) \leq cost(T_1)$, replace p with p_1.

(c) Repeat from (a).

3. Return **best**.

It is the best algorithm for large fully-connected symmetric TSP problems, as it is fast and can find near-optimal solutions.

Since local search for NLP has been considered in great details in the NLP course in the Autumn term, and since we don't have time to present it in too much detail for students who did not attend NLP, we only sketch some important ideas about it in the next section.

5.3.2 Local search for NLP

The majority of numerical optimisation algorithms for the NLP are based on local search.

Consider the **NLP problem:**

Given $f : \mathbb{R}^n \to \mathbb{R}$, optimise $f(\mathbf{x})$, $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, subject to $x \in F \subset S \subset \mathbb{R}^n$.

The **search space** of this problem $S \subset \mathbb{R}^n$ is given by:

$$l_i \leq x_i \leq u_i,\ 1 \leq i \leq n$$

The **feasible region** of this problem $F \subset S$ is given by:

$$g_j(\mathbf{x}) \leq 0,\ 1 \leq j \leq q$$

$$h_j(\mathbf{x}) = 0,\ q < j \leq m$$

At least one of the functions above should not be linear.

The gradient method of minimisation

- We use the gradient of the evaluation function at a particular candidate solution.
- We use this information to direct the search toward **improved** solutions (in a local sense).
- It is essential to find the right step size to guarantee the best rate of convergence over the iterations.

The maximum decrease occurs in the direction of the negative gradient $-\nabla f(\mathbf{x})$

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right]_{\mathbf{x}}$$

Steepest descent:

1. Start with an initial candidate solution $\mathbf{x_1}$. Let $k = 1$.

2. Generate a new solution

   $\mathbf{x_{k+1}} = \mathbf{x_k} - \alpha_k \nabla f(\mathbf{x_k})$

3. If $\|\mathbf{x_{k+1}} - \mathbf{x_k}\| > \varepsilon$ let $k = k + 1$ and go to step 2.

An evaluation (or objective) function f of a maximisation (minimisation) problem is called **unimodal** if it has a unique local maximum (unique local minimum). For smooth, unimodal evaluation function gradient methods perform very well.

Newton's method

$\mathbf{x}_{k+1} = \mathbf{x}_k - (H(f(\mathbf{x_k})))^{-1} \nabla f(\mathbf{x_k})$

The Hessian matrix of f:

$$H(f(\mathbf{x})) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Calculating the inverse of the Hessian matrix can be very time-consuming!

Summary

- The algorithms that work on complete solutions can be stopped at any time, they give a potential solution, provided the chosen algorithm matches the problem.
- For a smooth, unimodal evaluation function the gradient method is great.
- For a combinatorial problem (TSP) there are many local operators that take advantage of characteristics of the problem domain.
- But what if you don't know much about the problem or it does not fit within the problem class of any of these algorithms (i.e., algorithms on complete solutions)? Then it reasonable to start thinking of using algorithms on partial solutions which will be described in the next section.

Recommended reading

Z. Michalewicz & D.B. Fogel, How to Solve It: Modern Heuristics, Section 3.2. Local search

6 Greedy algorithms and Divide and Conquer

Overview

- Algorithms working on partial solutions
- Greedy algorithm
- Examples of greedy algorithm

- Divide and conquer
- Examples of divide and conquer

Algorithms working on partial solutions

Recall that for algorithms working on partial solutions one solution is constructed at a time, either an:

- **Incomplete** solution to the **original** problem, i.e.,

  A subset of the original problem's search space with a particular property hopefully shared by the real solution, or a

- **Complete** solution to a **reduced** problem. In this case the construction means that

  1. We decompose the original problem into smaller & simpler problems,
  2. We solve these problems,
  3. We try to combine the partial solutions into a solution to the original problem.

6.1 Greedy algorithms

Greedy algorithms are popular because they are very simple and fast. They construct the solution step by step. At each step the value for one decision variable is assigned by making the best available decision.

A **heuristic** is needed for making the decision at each step, that is, to decide what is the best now?

The best "profit" is chosen at every step, therefore the algorithm is called ***greedy***!

By taking the best available decision at each step, it does not mean that the "series of decisions" are the best for solving the problem. A best available decision at a given a step may not be an "overall best". A bad decision (or bad decisions) taken early cannot be corrected later, i.e., the greedy algorithm is "shortsighted". Therefore, we **cannot** expect the greedy algorithm to obtain the **overall** optimum. However, the greedy algorithm works well in cases when we know that there are many local optima and a global optimum is not necessary.

From now on (unless otherwise stated) we assume that we use the most common representation of SAT is to encode potential solution as a string of formed from zeros and ones. Therefore, from now on when we say Boolean string, we assume that we have such a string (unless otherwise stated).

Next, I will try to convince you that regardless of what greedy approach you use for SAT, there is no guarantee for success (i.e., finding a Boolean string which makes the Boolean function evaluate to TRUE).

6.1.1 Greedy algorithm and SAT

First we fix an order of considering the variables (usually the order is $x_1, x_2, \ldots, x_n$) and assign the truth value to the variables one at a time.

Next we choose the value 0 or 1 depending on which satisfies a greater number of presently unsatisfied clauses.

Please note that inside a clause you have either the unary operation NOT (negation) or the binary operation $\bigvee$ (disjunction). Suppose that the variable x_i, $i \in \{1, \ldots, n\}$ belongs to a clause of the form $\ldots \bigvee x_i \bigvee \ldots$. Thus, if $x_i = 1$, then it will satisfy the clause regardless of the values of the other variables. Now, suppose

that the variable x_i, $i \in \{1, \ldots, n\}$ belongs to a clause of the form $\ldots \bigvee \overline{x}_i \bigvee \ldots$. Thus, if $x_i = 0$, then it will satisfy the clause regardless of the values of the other variables. In conclusion, the assignments of further variables will not change the already satisfied clauses. This is the reason for ONLY considering the presently unsatisfied clauses at each assignment and choosing the value 0 or 1 depending on which satisfies a greater number of these clauses.

Unfortunately there are Boolean problems for which this greedy approach does not work. Below such a problem is presented.

Example of greedy for SAT:

Consider the Boolean function

$$\overline{x}_1 \bigwedge \left(x_1 \bigvee x_2\right) \bigwedge \left(x_1 \bigvee x_3\right) \bigwedge \left(x_1 \bigvee x_4\right).$$

Assume that the order of considering the variables is the usual one, i.e., first x_1, then x_2, then x_3, and finally x_4. At the beginning all clauses are unsatisfied. The assignment $x_1 = 0$ satisfies only the first clause. Thus, the assignment $x_1 = 0$ satisfies in total one clause. The assignment $x_1 = 1$ satisfies the second, third and fourth clauses. Thus, the assignment $x_1 = 1$ satisfies in total three clauses, more than the assignment $x_1 = 0$ which satisfies only one clause. We are "greedy" and thereore we choose the assignment $x_1 = 1$. However, with this assignment the first clause will become FALSE and the whole problem will evaluate to FALSE, regardless of the later assignments of values to x_2, x_3 and x_4.

Our first approach is too greedy!

But don't give up yet!

You may say that we could not find the solution because of the wrong order of considering the variables. We should have left x_1 to be assigned a value after assigning values to x_2, x_3 and x_4 because it appears in many clauses (in fact in all) and therefore there is a higher chance to break a clause by assigning a value to x_1 before assigning values to the other variables. Indeed, any order we consider, where x_1 is the last assignment would lead to a solution. We will describe this improved greedy algorithm next

Improved greedy algorithm for SAT

Sort all variables according to frequency of appearance, from smallest to largest.

For each variable, taken in the above order, assign the value that satisfies the greatest number of unsatisfied clauses.

For the example considered before this approach works indeed. Unfortunately, it doesn't work for all problems. The next example will show a SAT problem when this improved greedy approach does not work.

Example of improved greedy for SAT

Consider the Boolean function

$$\left(x_1 \bigvee x_2\right) \bigwedge \left(x_1 \bigvee x_3\right) \bigwedge \left(\overline{x}_1 \bigvee x_4\right) \bigwedge \left(\overline{x}_2 \bigvee \overline{x}_4\right) \bigwedge (x_2 \bigvee x_5) \bigwedge (x_2 \bigvee x_6) \bigwedge Formula(x_3, x_4, x_5, x_6),$$

with at least four occurences of each of the variables x_3, x_4, x_5 and x_6 in the Boolean function $Formula(x_3, x_4, x_5, x_6)$ of conjunctive normal form (which does not depend on x_1 and x_2!).

Can you satisfy both the third and fourth clauses in the same time?

The answer is simple: Even if you consider the improved version of greedy which takes into account the frequency of variable too, you cannot. Indeed, let us count the frequencies of variables. Variable x_1 occurs three times, variable x_2 four times and all the other variables at least five times. Therefore, we start assigning values to x_1 first. At the beginning all clauses are unsatisfied. The assignment $x_1 = 0$ satisfies the third clause only (satisfies one presently unsatisfied clauses). The assignment $x_1 = 1$ satisfies the first and second clause (satisfies two presently unsatisfied clauses). Therefore, we choose the assignment $x_1 = 1$, because it satisfies more presently unsatisfied clauses. Next, you continue by assigning values to x_2. The first two clauses have been already satisfied, so the presently unsatisfied clauses are the third, fourth, fifth and sixth. The assignment $x_2 = 0$ satisfies only one of these clauses, the fourth clause (satisfies one presently unsatisfied clause). The assignment $x_2 = 1$ satisfies the fifth and sixth clause (satisfies two presently unsatisfied clauses). Therefore, we choose the assignment $x_2 = 1$, because it satisfies more presently unsatisfied clauses. Hence, our Boolean problem is now

$$\cdots \bigwedge \left(0 \bigvee x_4\right) \bigwedge \left(0 \bigvee \overline{x}_4\right) \bigwedge \cdots,$$

where we ignored all clauses except the third and fourth, and this clauses we inserted the already assigned values $x_1 = 1$ and $x_2 = 1$. Now, if we assign the value $x_4 = 0$, then the third clause is FALSE (i.e., 0) and if we assign the value $x_4 = 1$, then the fourth clause if FALSE (i.e., 0). Hence, regardless of what we assign to x_4 our Boolean problem will evaluate to false. In conclusion, the improved version of greedy cannot find the solution of this problem.

But don't give up yet!

You can consider other ideas for an even better greedy for SAT!

Further ideas for improvement

- Forbid any assignment that makes any clause $FALSE$.
- At any time consider the frequency of variables for the ***remaining*** unsatisfied clauses.

  Indeed, these are the clauses which really matter, because the satisfied ones will remain satisfied regardless of the following assignments. So, why should we bother to consider the frequencies by including the satisfied clauses as well?
- We could take into account the ***length*** of the clause: the value of a variable in a short clause has more impact than the value of a variable in a long clause.

  Indeed, if a long clause is presently unsatisfied you will have higher chances for satisfying it by the following assignments than a short one, because it contains more variables.

Unfortunately, NONE of these can guarantee a solution to SAT! Regardless of what greedy approach you follow, you can always find a Boolean problem for which greedy will not find the solution. In conclusion:

There is NO GREEDY algorithm FOR SAT which WORKS!

Next we will consider greedy algorithms for TSP.

6.2 Greedy algorithm and TSP

We only consider the symmetric TSP. Let us first start by presenting the greedy algorithm based on the nearest neighbour heuristic.

Greedy based on **nearest neighbour heuristic**:

1. Select an arbitrary starting city.
2. As next city, select the city from the cities not yet visited that is closest to the current city. Go to this city.
3. Repeat step 2 until all cities have been visited.

Example of greedy based on the nearest neighbour heuristic for TSP:

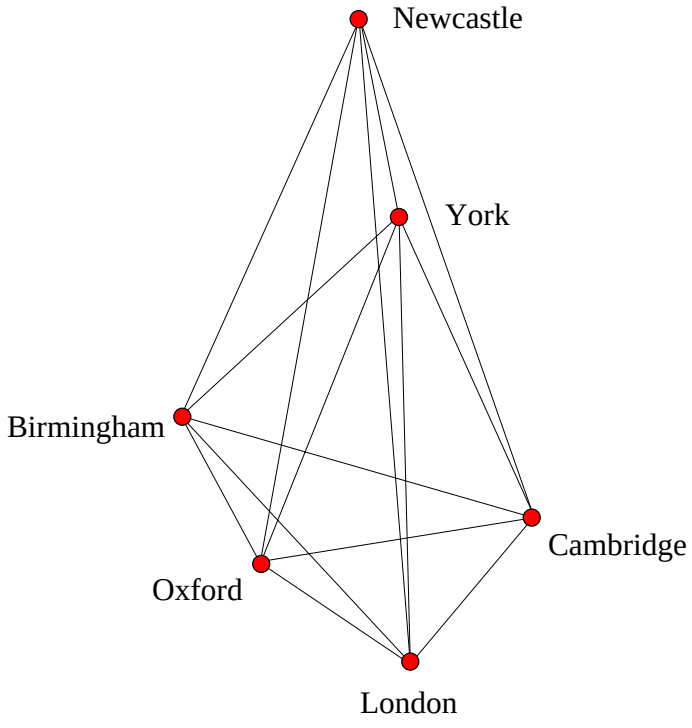
The travelling salesperson has to start the journey in Birmingham and visit the cities Cambridge, London, Newcastle, Oxford, York exactly once before going back to Birmingham and find the shortest route. We will show the solution (not necessarily best!) found by greedy based on the nearest neighbour heuristic. The pairwise distances by car were found by using the

`http://distancecalculator.globefeed.com/UK_Distance_Calculator.asp`

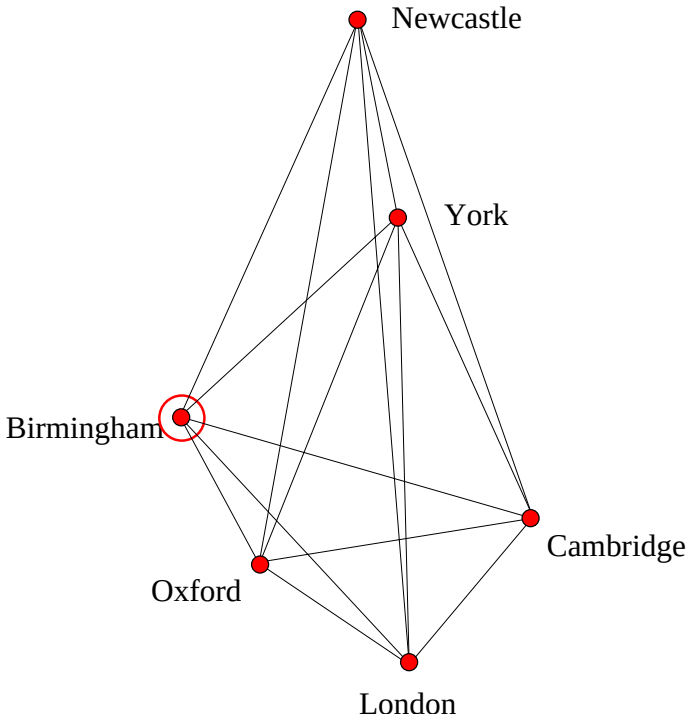
online distance calculator and are given in the following table (everything was rounded to the nearest integer):

	Oxford	London	Cambridge	York	Newcastle
Birmingham	79 miles	126 miles	100 miles	132 miles	206 miles
Oxford		60 miles	85 miles	194 miles	267 miles
London			64 miles	215 miles	289 miles
Cambridge				156 miles	229 miles
York					85 miles

The next figure shows a complete graph containing all possible edges between these cities:



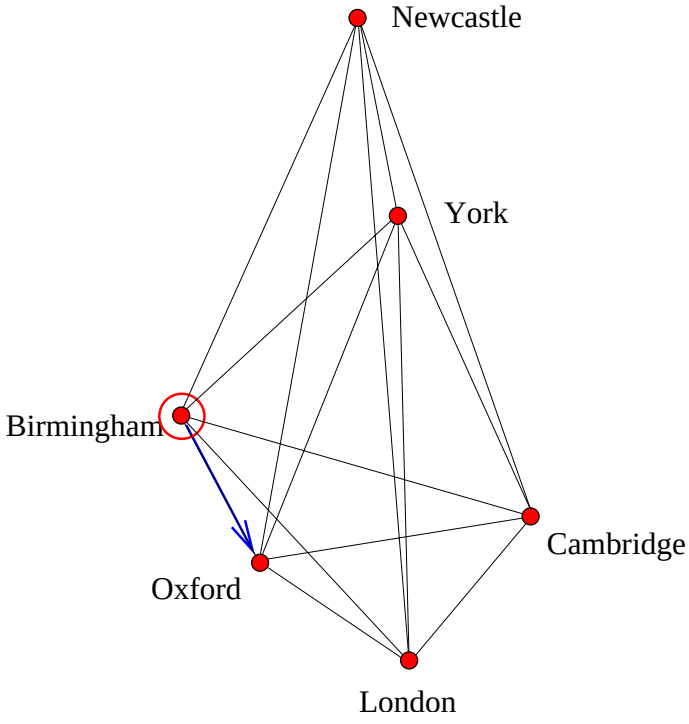
The starting point is Birmingham:



We can choose between the following possibilities:

Birmingham-Oxford	79 miles
Birmingham-London	126 miles
Birmingham-Cambridge	100 miles
Birmingham-York	132 miles
Birmingham-Newcastle	206 miles

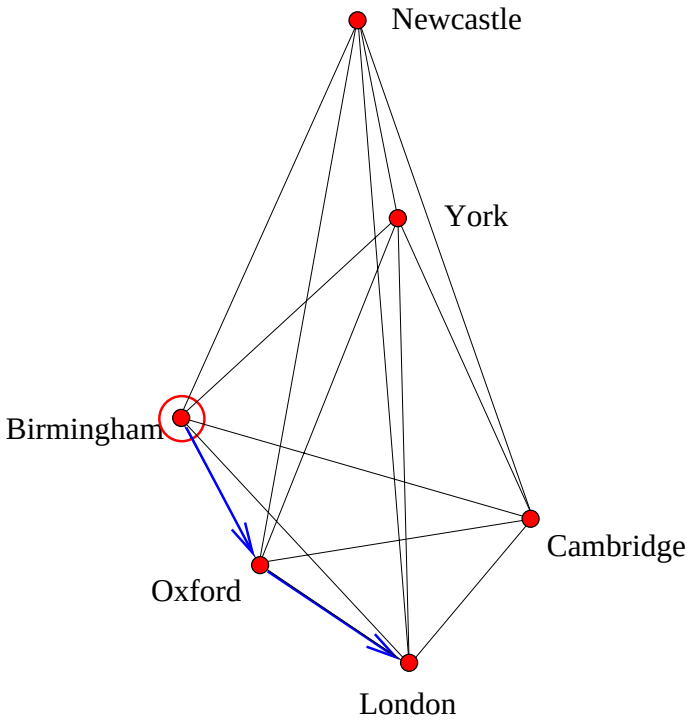
The shortest of these is Birmingham-Oxford, so we select Oxford:



The currently visited city is Oxford and the unvisited cities are London, Cambridge, York and Newcastle. Hence, we can choose between the following possibilities:

Oxford-London	60 miles
Oxford-Cambridge	85 miles
Oxford-York	194 miles
Oxford-Newcastle	267 miles

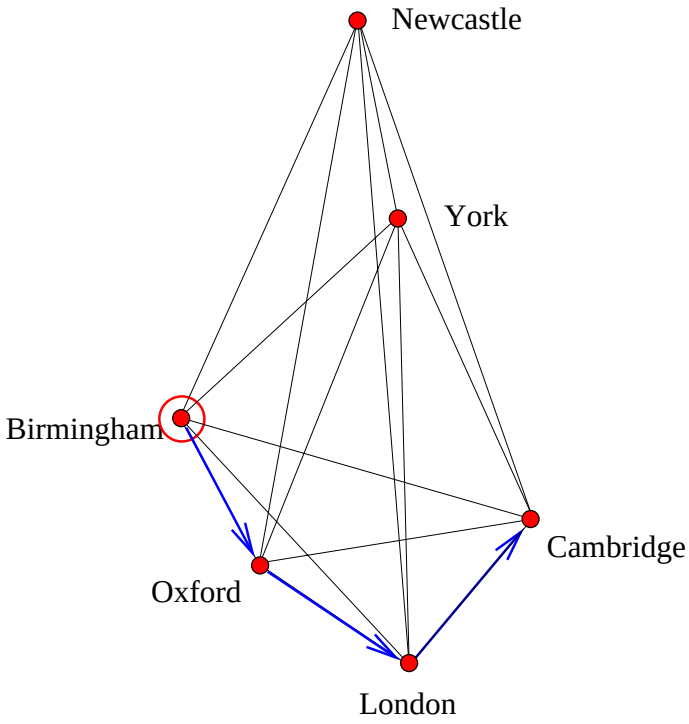
The shortest of these is Oxford-London, so we select London:



The currently visited city is London and the unvisited cities are Cambridge, York and Newcastle. Hence, we can choose between the following possibilities:

London-Cambridge	64 miles
London-York	215 miles
London-Newcastle	289 miles

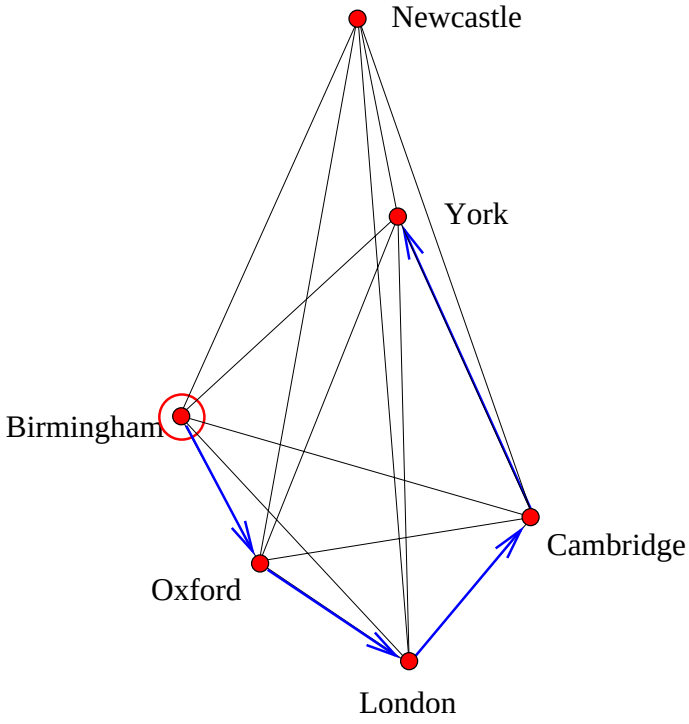
The shortest of these is London-Cambridge, so we select Cambridge:



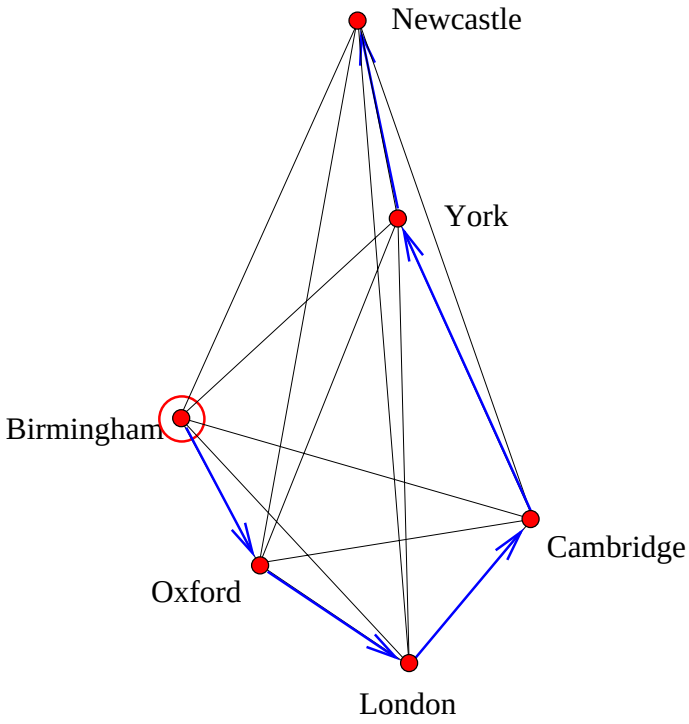
The currently visited city is Cambridge and the unvisited cities are York and Newcastle. Hence, we can choose between the following possibilities:

Cambridge-York	156 miles
Cambridge-Newcastle	229 miles

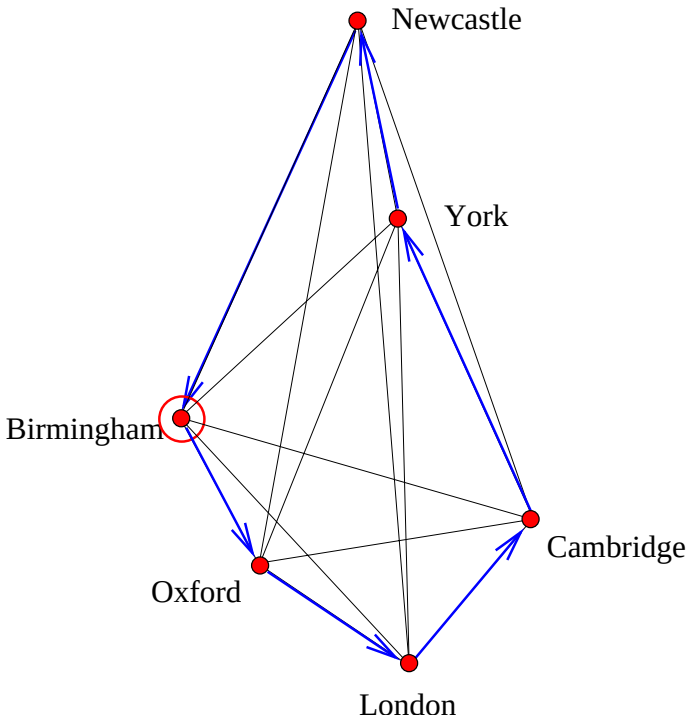
The shortest of these is Cambridge-York, so we select York:



Only one unselected city remains which is Newcastle, so we select Newcastle:



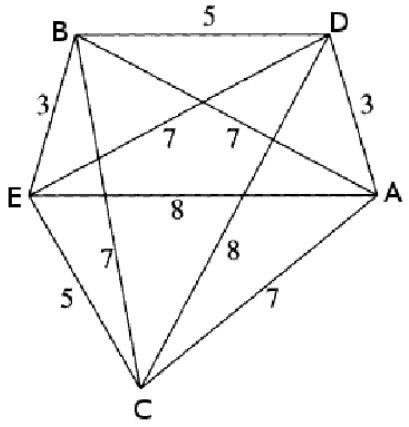
Finally, we complete our tour:



Another example of greedy based on the nearest neighbour heuristic for TSP:

Consider a symmetric travelling salesman problem with 5 cities A, B, C, D, E. The distances between cities are as follows:

City	City	Distance
A	B	7
A	C	7
A	D	3
A	E	8
B	C	7
B	D	5
B	E	3
C	D	8
C	E	5
D	E	7



Solve the problem by using the nearest neighbour heuristic starting from A.

Solution:

At A we can go to B, C, D, E. The nearest of these to A is D. Hence, we go to D. At D we can go to B, E, C. The nearest of these to D is B. Hence, we go to B. At B we can go to E, C. The nearest of these to B is E. At E the only choice is to go C. Hence, we go to C. Finally, we complete our tour (A,D,B,E,C,A). Therefore, the solution found by the nearest neighbour heuristic is (A,D,B,E,C,A).

What is the solution if you start from a different city? Will you get the same solution? Construct a TSP example for which you can find two different starting points with different solutions for the nearest neighbour heuristic!

Next we present another greedy algorithm for TSP.

Algorithm based on another greedy heuristic for TSP

1. From all possible edges select the shortest one and add it to the tour.
2. Reduce the set of possible edges which would lead to illegal tours, i.e., the ones which would lead to one of the following situations:
   (a) A node which belongs to three edges.
   (b) A cycle which does not contain all cities.
3. Repeat steps 1-2 until the tour is complete.

In practice you would order the pairwise distances and pick edges in the order the shortest one, second shortest one, third shortest one etc. If one of the edges leads to an illegal tour, then skip it.

To illustrate this idea consider the TSP problem described in the previous example. It is easy to see that in this case the algorithm starts by choosing in the first two steps AD and BE. In the next two steps the algorithm is choosing BD and EC and finally the algorithm must choose CA to complete the tour (A,D,B,E,C,A). However, things are not always that easy. We describe a more complicated situation in the next example.

Example of greedy based on another heuristic for TSP

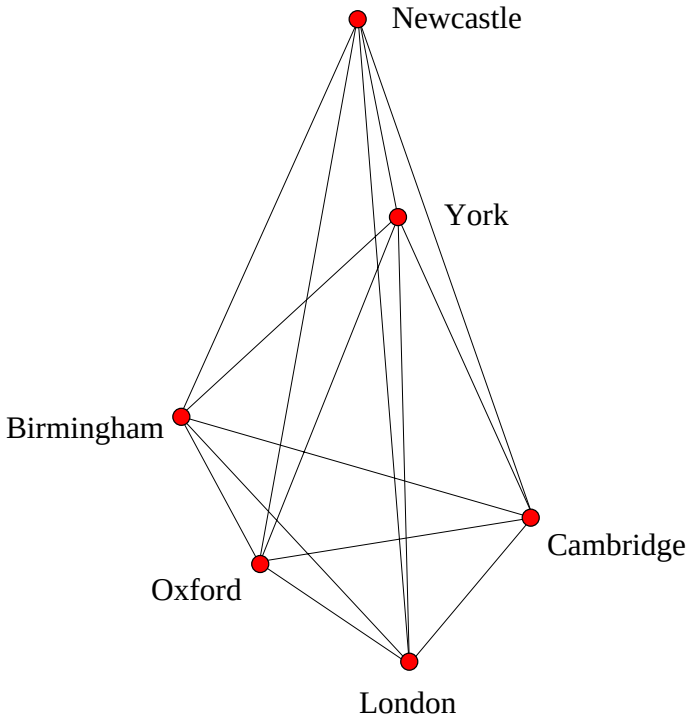
Consider the same problem as before, but now we have pairwise times between different cities (subject to traffic) and instead of giving them explicitly we rank them as follows:

1 = shortest time, 2 = second shortest time, 3 = third shortest time, etc. If you have times given between cities, certainly it should not be a problem for you to order them from the shortest time to the longest one. For simplicity, assume that we have the situation given in the below table:

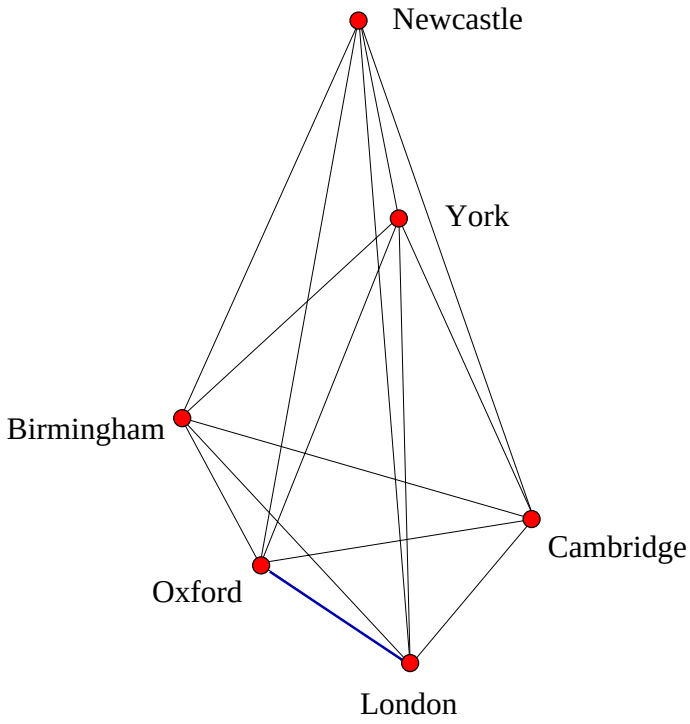
	Oxford	London	Cambridge	York	Newcastle
Birmingham	3	7	6	8	11
Oxford		1	5	10	14
London			4	12	15
Cambridge				9	13
York					2

Please note, that by finding a tour based on the order of distances only, you don't have its evaluation, but suppose that we first had the times and we ordered them only afterwards. In this way, after finding a tour we can check whether we can find a shorter one or not (which is in general a nontrivial task).

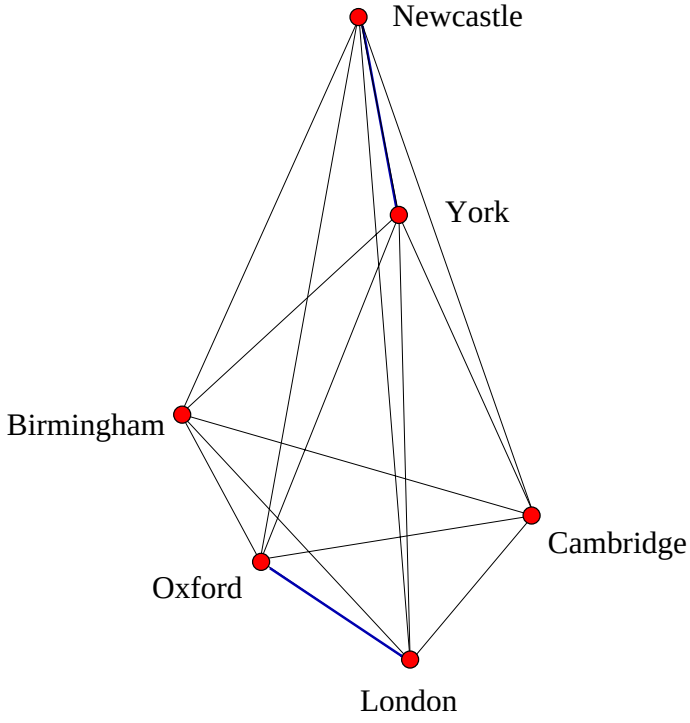
The next figure shows a complete graph containing all possible edges between these cities:



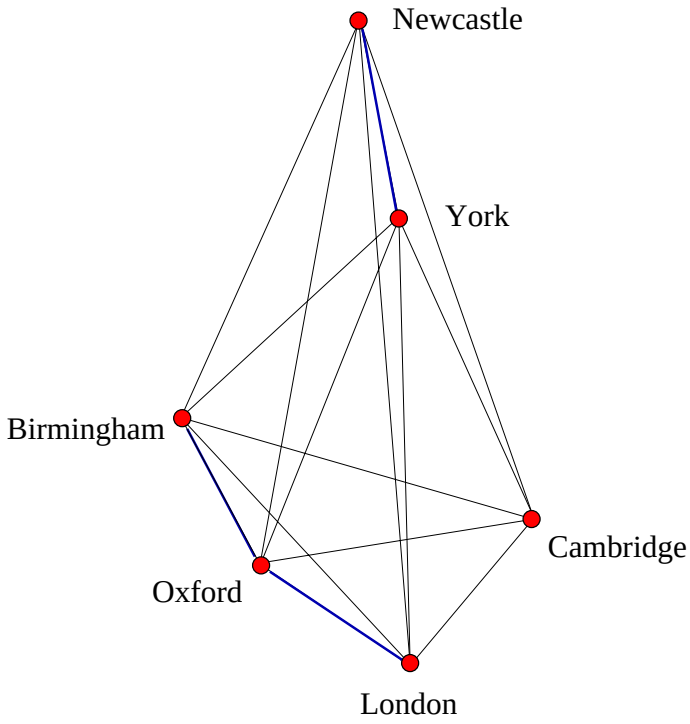
We start by choosing the edge with the shortest time: Oxford-London.



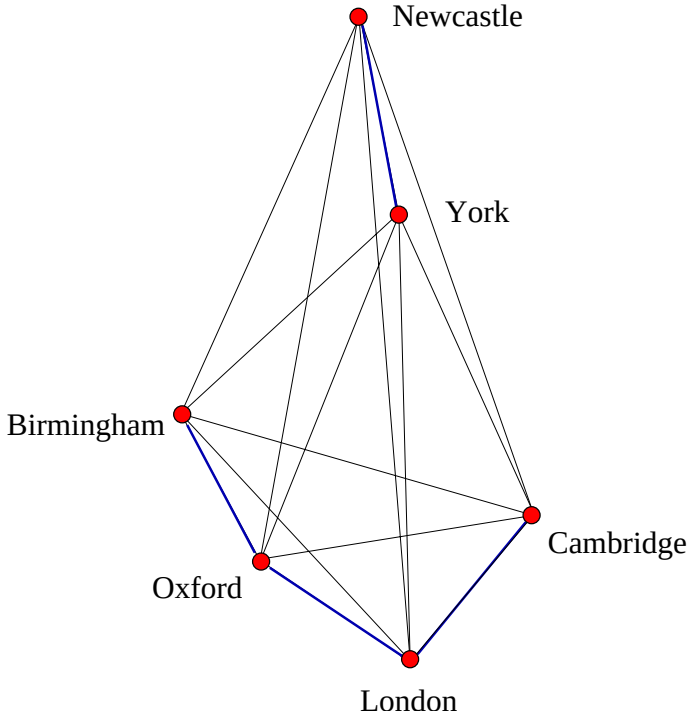
We continue, by choosing the edge with the second shortest time: York-Newcastle.



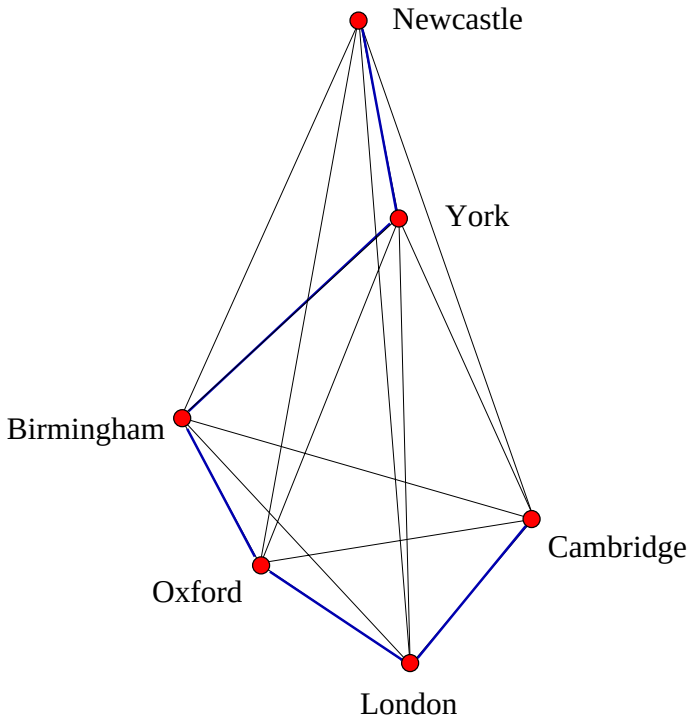
We continue, by choosing the edge with the third shortest time: Birmingham-Oxford.



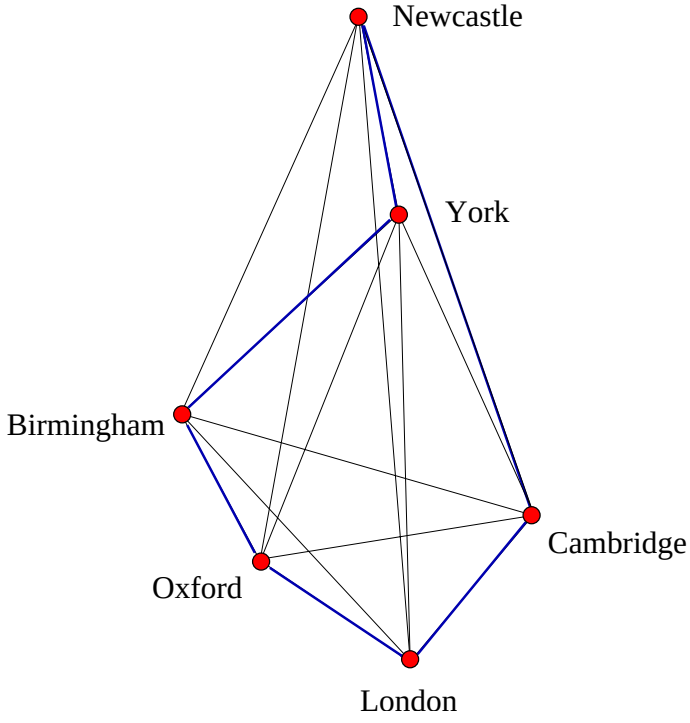
We continue, by choosing the edge with the fourth shortest time: London-Cambridge.



The next edge would be edge with the fifth shortest time: Oxford-Cambridge. But choosing this edge would lead to an illegal tour (because of the cycle Oxford-London-Cambridge-Oxford, which does not contain all cities or the city Oxford belonging to three edges). We skip this edge. The next edge would be edge with the sixth shortest time: Birmingham-Cambridge. But choosing this edge would lead to an illegal tour (because of the cycle Birmingham-Oxford-London-Cambridge-Birmingham, which does not contain all cities). We skip this edge. The next edge would be edge with the seventh shortest time: Birmingham-London. But choosing this edge would lead to an illegal tour (because of the cycle Birmingham-Oxford-London-Birmingham, which does not contain all cities). We skip this edge. The next edge would be edge with the eighth shortest time: Birmingham-York. We choose this edge.



Without presenting more details, it is clear that any choice different from Newcastle-Cambridge would lead to an illegal tour. Therefore, we complete our tour by choosing Newcastle-Cambridge.



Example of greedy for the knapsack problem

Consider the 0/1 knapsack problem:

A set of N items, each having a value v_i and a weight w_i are given. There is a bag of finite capacity K (maximum weight it can hold). The problem is to fill the bag with items (without exceeding its capacity),

so that the total value of items in the bag is maximised. You can either put an item in the bag as a whole or leave it out, it is not possible to use parts of an item.

(a) Describe a greedy algorithm for this problem (Hint: first try to solve the example from (b) by hand)

(b) For the particular case:

N=5

$v_1 = 25$, $w_1 = 21$

$v_i = 10$, $w_i = 10$ for $i = 2, 3, 4, 5$

$K = 40$

would the greedy algorithm find the optimal solution?

Solution:

(a) The greedy step can be selecting the item with the *"best value for money"*, that is the item with maximum $\frac{v_i}{w_i}$ and putting it in the bag if the bag can hold it. The stopping criterion: there are no more items that can fit in the bag, in the sense that if the current item does not fit in the bag it is allowed to consider the next item in the decreasing order of v_i/w_i and put it in the bag if it fits.

(b) For this case the optimum is having items $2, 3, 4, 5$ in the bag. The total value is $V = \sum_{i=2}^{5} v_i = 40$, the total weight is $W = \sum_{i=2}^{5} w_i = 40 \leq K$.

The the solution found by the greedy algorithm would be items $1, 2$ in the bag. The total value is $V = v_1 + v_2 = 35$ and the total weight is $W = w_1 + w_2 = 31 \leq 40$. This is not the global optimum, because putting items $2, 3, 4, 5$ in the bag has a total weight of $W = w_2 + w_3 + w_4 + w_5 = 40 \leq 40$ and a higher total value $V = v_2 + v_3 + v_4 + v_5 = 40$.

6.3 Divide and conquer

Divide and conquer is an algorithm which brakes a problem P in several subproblems, solve these problems and assemble these sub-solutions into a solution of the original problem P. If these problems are too large, then continue by breaking these subproblems into sub-subproblems and use the idea described before replacing the original problem P with the subproblems. If the sub-subproblems are still too large, then further decompose them into sub-sub-subproblems etc., until this recursive process leads to small enough problems which can be combined in a recursive fashion to find the solution of the original problem. This process is outlined by the below algorithm:

Divide&conquer(P)

1. Split problem P into subproblems $P_1, P_2, \ldots, P_k$.
2. For i taking all the values from 1 to k

   get the solution S_i to problem P_i
3. Combine $S_1, S_2, \ldots, S_k$ into the solution S for problem P.
4. Return the solution S.

The recursive step 2

Get the solution S_i to problem P_i:

if $size(P_i) < \rho$ then
    solve P_i and get its solution S_i
else
    apply **Divide&conquer(P_i)** and
    assign the result to S_i.

The complete algorithm

Divide&conquer(P)

1. Split problem P into subproblems $P_1, P_2, \ldots, P_k$.
2. For i taking all the values from 1 to k do
   if $size(P_i) < \rho$ then
       solve P_i and get its solution S_i
   else
       apply **Divide&conquer(P_i)** and
       assign the result to S_i.
3. Combine $S_1, S_2, \ldots, S_k$ into the solution S for problem P.
4. Return the solution S.

Is divide and conquer effective?

Time and effort is needed for

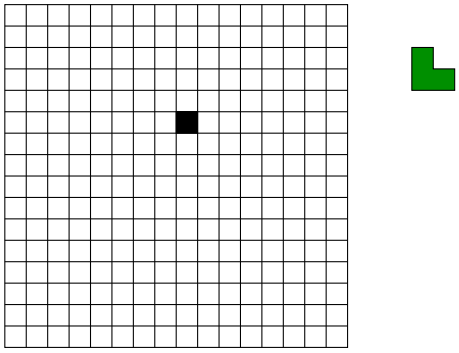
- decomposing the problem into subproblems,
- solving the subproblems,
- assembling the solutions to the subproblems into the solution to the initial problem.

The algorithm is cost effective **only if** its cost is **less than** the cost of solving the original problem.

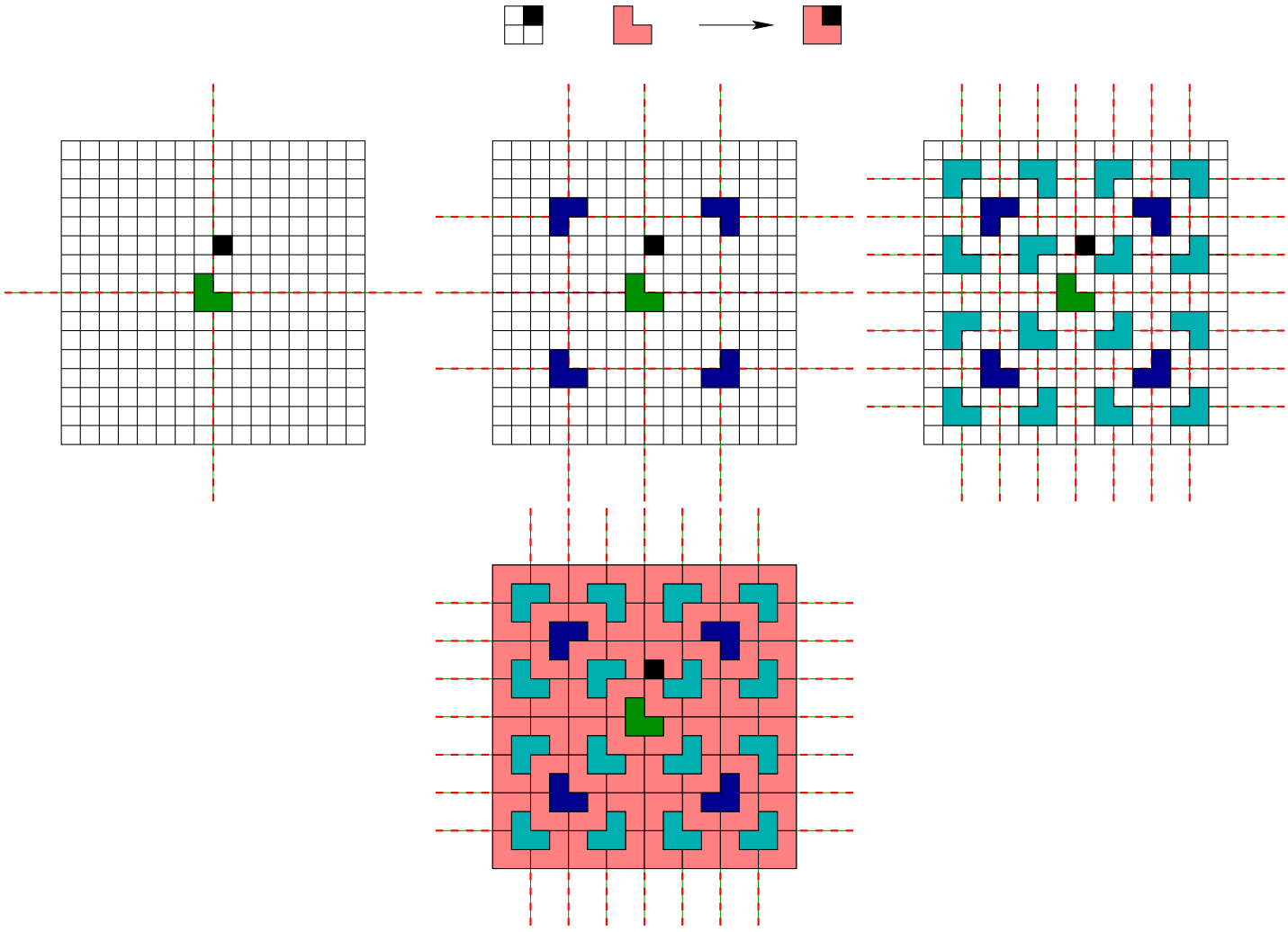
Example of divide and conquer

A chessboard of size 16×16 with a hole is given (one arbitrary 1×1 square is removed from the board).

A sufficient number of L-shaped tiles is given and the task is to cover the board with these tiles such that every 1×1 square of the board will be covered by exactly one L-shaped tile and none of the tiles will have points outside the board (that is, the board will be fully covered by non-overlapping tiles which have their entire surface on the chessboard). Let us use divide and conquer to solve this puzzle.



First of all, we can observe that a similar problem of size 2×2 can be easily solved. Hence, by a recursive division we would like to reduce our problem to 2×2 problems. First divide the board into four boards of size 8×8, by cutting the original board at the middle both horizontally and vertically. In order to reduce the problem to four similar problems of size 8×8 put a tile at the middle of our original board, so that it covers exactly one 1×1 square for three of the four 8×8 boards which did not miss any 1×1 square. Now all four 8×8 boards have a missing 1×1 square (the covered 1×1 squares can also be considered missing, because the tiles covering the boards should be non-overlapping). We can repeat the process by dividing the 8×8 problems into similar 4×4 boards with a missing square (each 8×8 board into four 4×4 boards) by using the same trick of putting a tile at the middle of the 4×4 boards. If we do one more step in a similar fashion we arrive to similar 2×2 problems which need to be solved. But solving this problem is trivial. The whole process is shown in the figure below:



6.4 Divide and conquer for SAT?

It seems easy to divide the SAT problem:

$$F(x_1, x_2, x_3, x_4) = \left(x_1 \bigvee x_2\right) \bigwedge \left(x_1 \bigvee x_3\right) \bigwedge \left(\overline{x}_1 \bigvee x_4\right) \bigwedge \left(\overline{x}_2 \bigvee \overline{x}_4\right).$$

$$\begin{aligned} F_1(x_1, x_2, x_3, x_4) &= x_1 \bigvee x_2 \\ F_2(x_1, x_2, x_3, x_4) &= x_1 \bigvee x_3 \\ F_3(x_1, x_2, x_3, x_4) &= \overline{x}_1 \bigvee x_4 \\ F_4(x_1, x_2, x_3, x_4) &= \overline{x}_2 \bigvee \overline{x}_4 \\ F(x) &= F_1(x) \bigwedge F_2(x) \bigwedge F_3(x) \bigwedge F_4(x) \end{aligned}$$

However, the subproblems F_1, F_2, F_3 and F_4 are **not independent**:

F_1, F_2, F_3 all depend on x_1.
F_1, F_4 depend on x_2.
F_3, F_4 depend on x_4.

Therefore, the solutions to these subproblems cannot be combined into a solution to the original problem, because they are not independent.

Another possibility for SAT?

Try to group the disjunctions according to the variables they depend on.

Disjunctions that have at least one variable in common, should be in the same group.

For the given example, all disjunctions would belong to the same group.
$\longrightarrow$ **NO** such division is possible!
However, if we consider the problem

$$F(\mathbf{x}) = \left(x_1 \bigvee \overline{x}_2\right) \bigwedge \left(x_2 \bigvee \overline{x}_3 \bigvee x_4\right) \bigwedge \left(x_5 \bigvee \overline{x}_6\right) \bigwedge \left(x_6 \bigvee \overline{x}_7 \bigvee x_8\right) \bigwedge \left(x_9 \bigvee \overline{x}_{10}\right) \bigwedge \left(x_{10} \bigvee \overline{x}_{11} \bigvee x_{12}\right),$$

where $x = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12})$, we can observe that we can divide the problem into 3 similar problems of equal size $F_k(x_{4k-3}, x_{4k-2}, x_{4k-1}, x_{4k}) = (x_{4k-3} \bigvee \overline{x}_{4k-2}) \bigwedge (x_{4k-2} \bigvee \overline{x}_{4k-1} \bigvee x_{4k})$; $k = 1, 2, 3$. Solving these problems is similar. Combining the solutions of these problems we find a solution of the original problem. Hence, divide and conquer can be used in this case.

Summary

- The **greedy algorithm** constructs a solution to the problem step by step.
- The **greedy algorithm** is recommended in cases when we know that there are many local optima and a global optimum is not necessary.
- **Divide and conquer** solves subproblems.
- It is not always clear how to divide a problem into subproblems or it might not be possible.

Recommended reading

Z. Michalewicz & D.B. Fogel, How to Solve It: Modern Heuristics, Chapter 4: Traditional methods -Part 2, Sections 4.1-4.2

7 The A* search algorithm

Overview

- Best-first search
- Minimizing the total cost ⇒ A*
- Examples of A*
- The algorithm of A*
- Observations

Next we will consider graph search algorithms where the goal is to find the shortest (or more generally lowest "cost", where "cost" is additive along any path and is not necessarily a physical distance, it can be for example "time" or "fuel quantity" etc) path from a START node to a GOAL node in a graph. Although this algorithm is usually used for minimisation it is possible to use it for maximisation too. We will restrict ourselves to minimisation problems only. Graph search algorithms are algorithms on incomplete solutions, because if you stop the algorithm earlier than reaching the goal, then you don't have a path from START to GOAL, hence your solution is incomplete.

Let us start with explaining the terminology used in graph search algorithms. A graph search algorithm chooses a node at each step and **expands** it, by generating its **successors**. These successors will be called the **generated nodes** and if they have not been expanded before, then they are added to a list called OPEN list (some of them might have already been there, in which case no addition is necessary). Any expanded node go to a list called CLOSED list. At each step the algorithm chooses between the nodes in the OPEN list. Please note that a node from the OPEN list may go to the CLOSED list, but a node from the CLOSED list never goes back to the OPEN list (that's why the terminology CLOSED). Please also note the following remarks: (1) Any CLOSED node has been expanded before, so it has been also generated before; (2) Any CLOSED node has been OPEN before; (3) Any OPEN node has been generated before; (4) Hence, the nodes which are neither in the OPEN nor in the closed list are exactly the ones which have not been generated before.

Let us first consider a purely greedy approach.

7.1 Best-first search

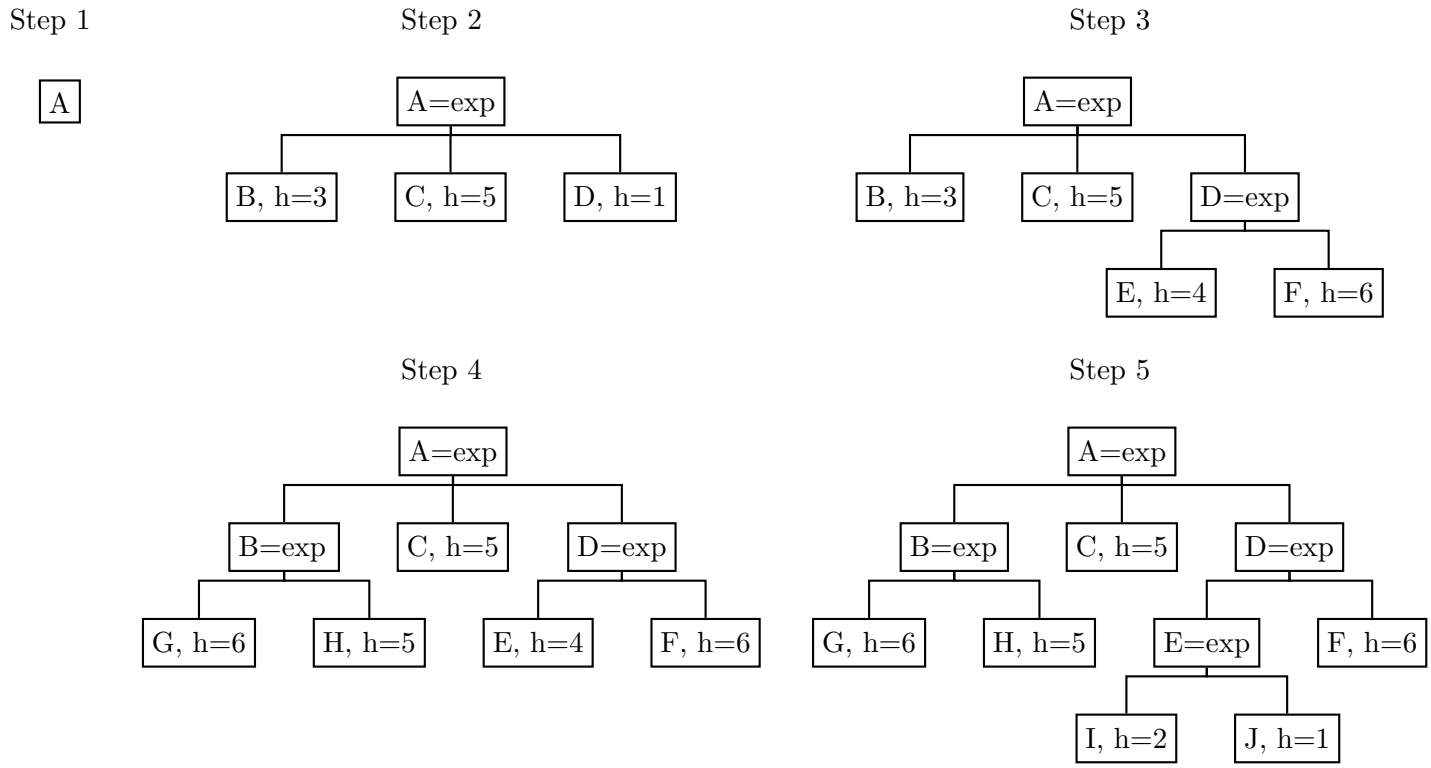
This algorithm is also called **greedy best-first** and is described as follows:

- We follow only one path at a time, but when a competing path looks more promising, we switch to that path.
- We apply a heuristic function to each of the generated nodes.
- At each step we choose (randomly) one of the generated (but not yet expanded) nodes (at the present step and at all previous ones) which has the lowest heuristic value.

Therefore, the greedy algorithm chooses at each step the best node from the OPEN list (i.e., with lowest evaluation value, which is the heuristic in the case of greedy best-first) expands it, adds it to the CLOSED list (if an expanded node has a duplicate, then all of its duplicates will be added to the CLOSED list) while adding its not yet expanded successors to the OPEN list. In case of the examples we will present for greedy search we will try to keep in mind the nodes which have been generated and the nodes which have been expanded (called also examined in the

literature) and make our choices accordingly, without writing the OPEN and CLOSED list explicitly during the algorithm. In case of more complicated graph search problems (especially for A*) we will write the OPEN and CLOSED lists explicitly at each step. The next example illustrates a few steps of a possible greedy best-first algorithm. The heuristic (h) values of the OPEN nodes are given next to each node. The nodes which have been expanded (i.e., the nodes in the CLOSED list) are denoted by "=exp". At each step the algorithm chooses the node in the OPEN list with the lowest h value.

Example:



At Step 2 we expand node A, put it in the closed list and put its successors B, C, D in the open list. The closed list becomes CLOSED=(A) and the open list OPEN=(B,C,D). We have to choose the node in the open list with the lowest h value, that is, D.

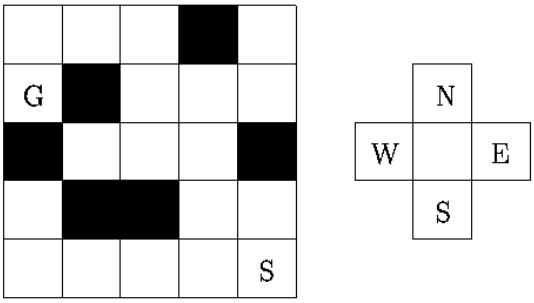
At Step 3 we expand node D, put it in the closed list and put its successors E, F in the open list. The CLOSED list becomes CLOSED=(A,D) and the open list OPEN=(B,C,E,F). We have to choose the node in the open list with the lowest h value, that is, B.

At Step 4 we expand node B, put it in the closed list and put its successors in the open list. The CLOSED list becomes CLOSED=(A,B,D) and the open list OPEN=(C,E,F,G,H). Observe that we switched to a more promising competing path. We have to choose the node in the open list with the lowest h value, that is, E.

At Step 5 we expand node E, put it in the closed list and put its successors in the open list. The CLOSED list becomes CLOSED=(A,B,D,E) and the open list OPEN=(C,F,G,H,I,J). Observe that we switched to a more promising competing path. We have to choose the node in the open list with the lowest h value, that is, J.

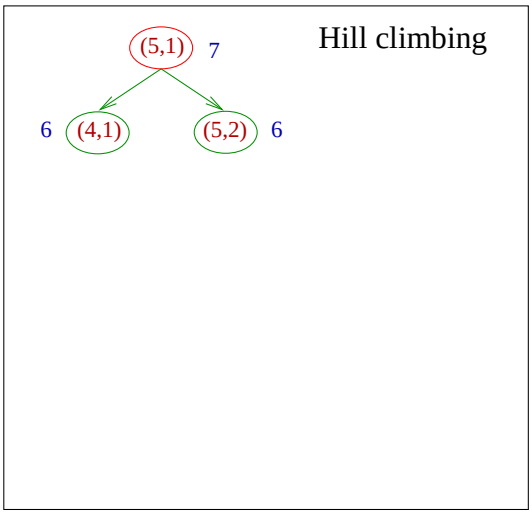
7.2 Comparison of Best-First and Hill-Climbing for the Maze problem

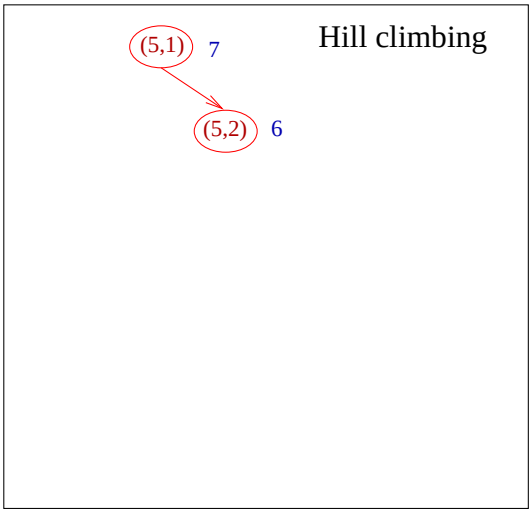
Consider the following maze:

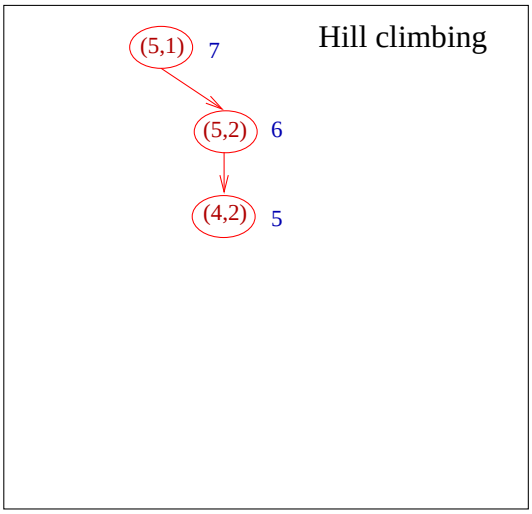


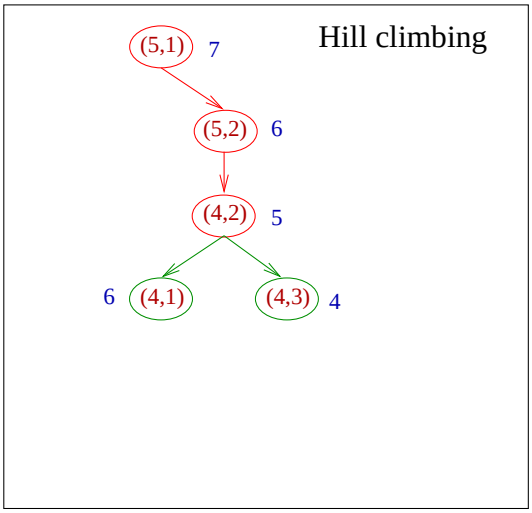
The problem is to get from the start node S to the goal node G, by moving horizontally and vertically and avoiding the black obstacles in the above maze. For this problem there is a way to get from S to G. Please note, that for a different maze problem there may not be a way to get from S to G. In this case the problem will be to get as close as possible to G with respect to the Manhattan distance. If we formulate the Maze Problem in this way, then we don't care about the length of our path from S to G, just about getting to G, or getting as close as possible to G. However, the greedy best-first algorithm will have a completely different goal, namely, finding the shortest path from S to G.

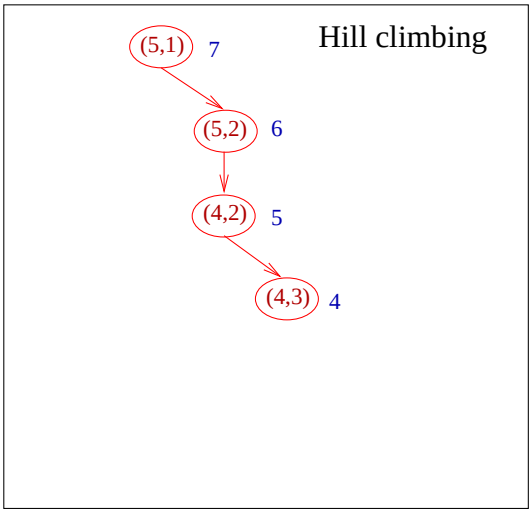
Let us first recall how the basic hill-climbing algorithm solves this problem. In the below figures describing the steps of the algorithm each position will be denoted by a pair of numbers which are the horizontal and vertical coordinates (that is, for (x, y), x denotes the horizontal coordinate and y denotes the vertical coordinate). Recall that the Manhattan distance between the points (x, y) and (z, t) is given by $d((x, y), (z, t)) = |x - z| + |y - t|$, where $|\cdot|$ denotes the absolute value (or modulus) function. At each step the algorithm examines the neighbours of the current point x and chooses (randomly) one with the lowest Manhattan distance. If the chosen point is better than x, then it becomes the current point and the algorithm continues. If not, then the algoritms returns x. The steps of the basic hill-climbing algorithm are described below:

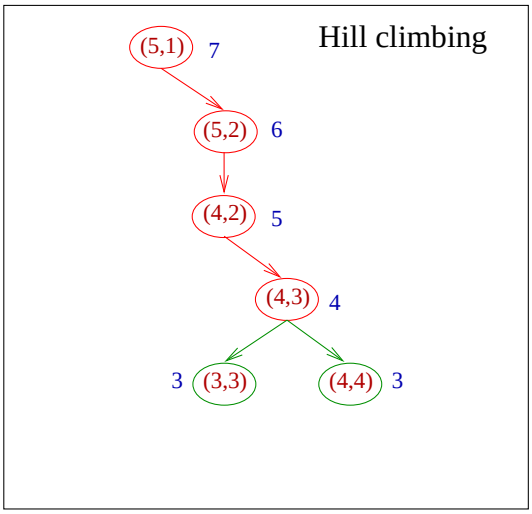


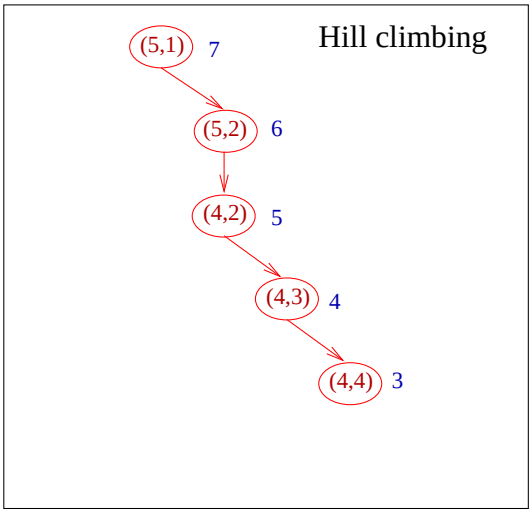

Hill climbing
(5,1)
7
(5,2)
6

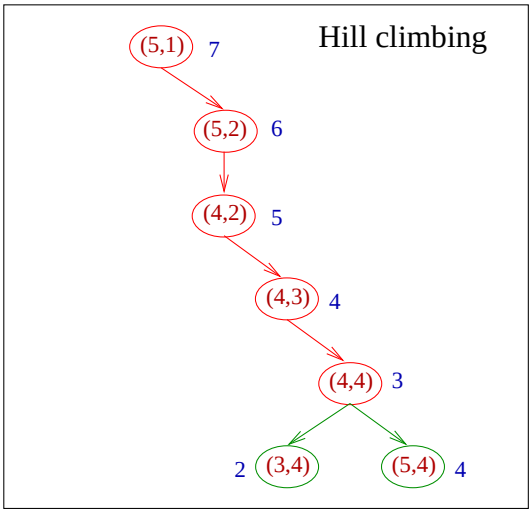

Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5

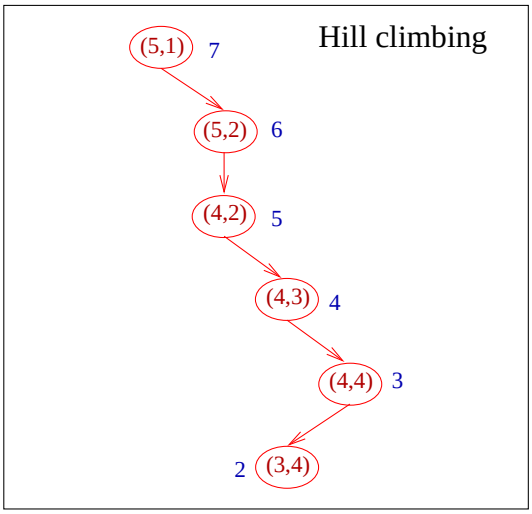

Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5
6
(4,1)
(4,3)
4


Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5
(4,3)
4


Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5
(4,3)
4
3
(3,3)
(4,4)
3

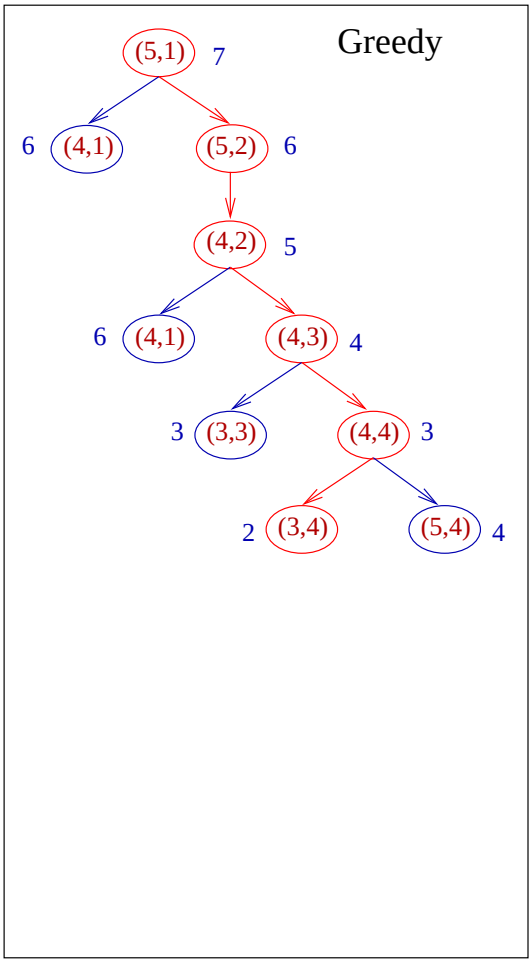

Hill climbing
(5,1)
7
(5,2)
6
(4,2)
5
(4,3)
4
(4,4)
3



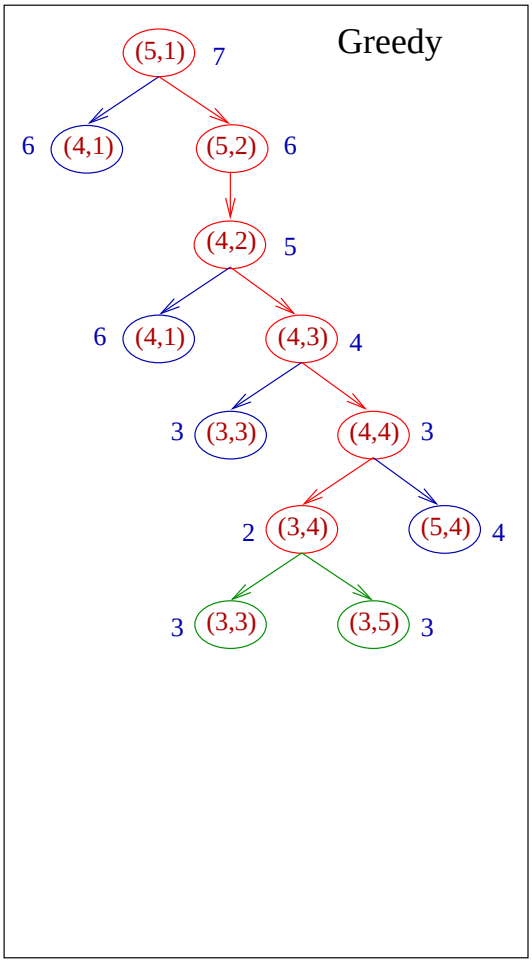


The hill-climbing algorithm is stuck in (3,4) because all neigbours of (3,4) are worse than (3,4) (i.e., they have a higher Manhattan distance).

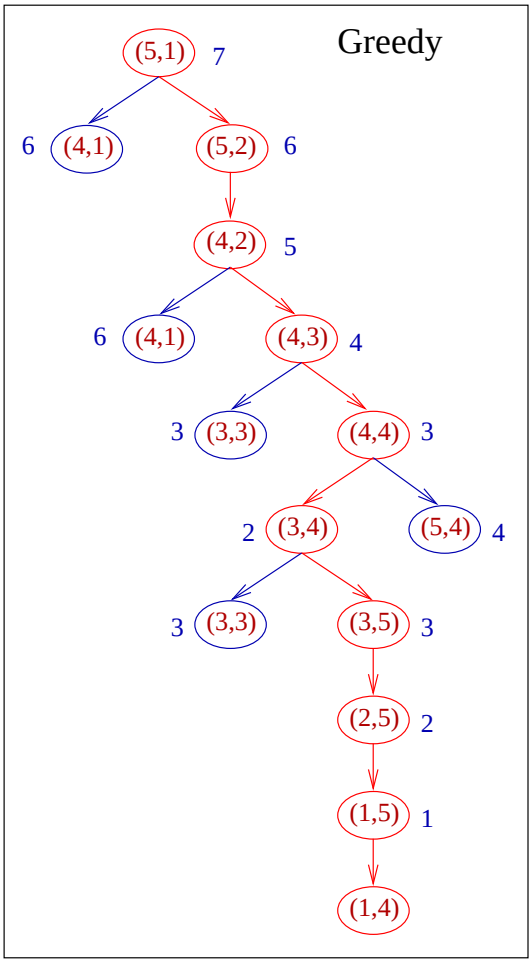
Now, let us see if the greedy algorithm can do better than hill-climbing. The heuristic value of a node is the Manhattan distance of the the node from the goal node G. Please recall that at each step greedy considers the nodes which has been generated (at the current step and all previous steps) but not examined (i.e., expanded) yet. Among these nodes chooses the one with lowest Manhattan distance and expands its by generating its successors. By doing so, greedy will reach (i.e., choose at some step) the same node (3,4). At this stage the generated, but not yet examined nodes are "two copies of (4,1)", (3,3) and (5,4).



By expanding (3,4) the algorithm generates another copy of (3,3) and (3,5) too. Next, the algorithm chooses between the generated, but not yet examined nodes, which are: "two copies of (4,1)", "two copies of (3,3)", (5,4) and (3,5). Among these the ones with lowest heuristic value are the "two copies of (3,3)" and (3,5). The algorithm chooses one of these nodes randomly. Let us assume that the chosen node is (3,5) (Hint: you are not a computer, hence when you solve a problem always try to make the simplest choice when you are at a "draw", i.e., which makes your solution better and if possible shorter.).



From the chosen node (3,5) the algorithm will go directly to the goal node.



7.3 The underlying compound heuristic function of A*

The A* graph search is similar to the greedy best-first. The algorithm has exactly the same steps, except that the quality of a node n is measured by a different evaluation function $f(n)$, which contains both the estimated value of n (a guess of how close n is to the GOAL node) and the cost of getting from the START node to n, which is an exact value. More precisely:

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

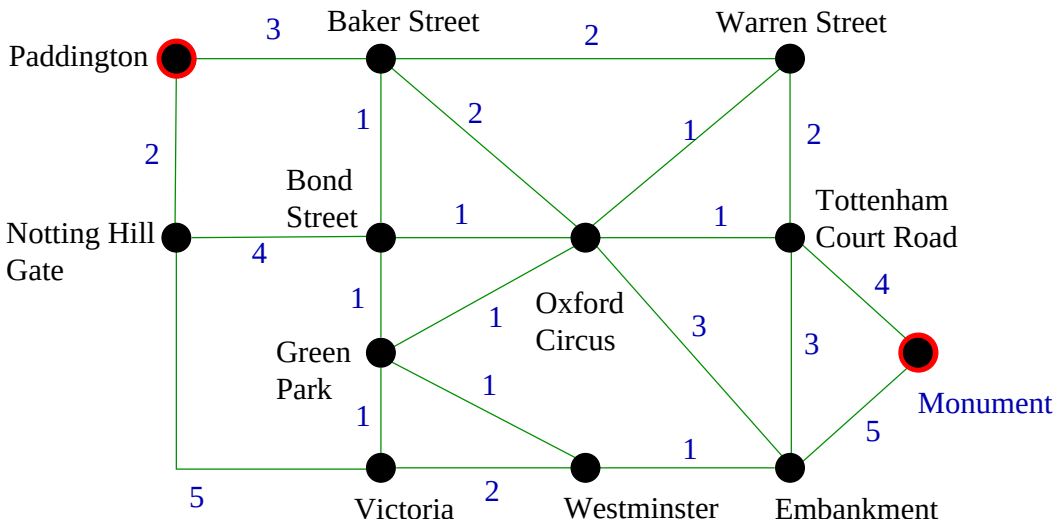
where $g(n)$ is a measure of the cost of getting from the **initial** state to the **current state** n and

$h(n)$ is an **estimate** of the cost of getting from the **current** state n to the **goal state**.

One question which arises naturally (and causes confusion for some students) is what is is a good value of h, large or low? Although A* minimises f, and h is a component of f, the goodness of h has a different meaning in A*. As we will see later an essential requirement for the success of A* is that $h(n)$ is not overestimating the cost to get from n to GOAL. However, under this restriction, we would like h to be as large as possible, that is, as close as possible to the real distance, because the better h estimates the real distance, the closer A* is to the **"direct"** path, i.e., the more efficient is the algorithm. We will see more details about these ideas later. For the moment, let us note that the large values of h are the good ones.

Example 1 – Shortest path for the London underground

Consider a part of the London underground. The goal is to find the shortest path from Paddington to Monument. The distances between the stations are given on the edges of the below graph:



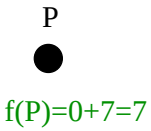
The estimated distances from Monument (i.e., the h values) are:

P	BaS	WaS	NHG	BoS	OC	TCR	GP	V	We	E	M
7	5	4	7	5	5	3	5	5	4	3	0

with the obvious abbreviation for the corresponding stations.

Example 1 – A* solution

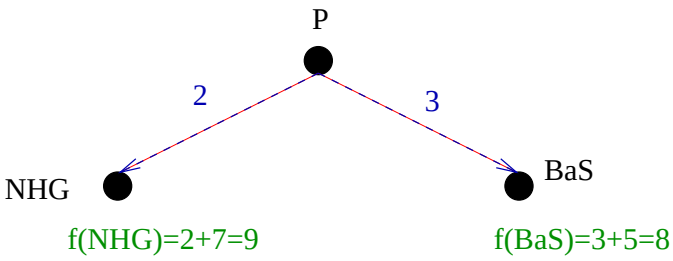
At the beginning we are in P where g(P)=0 and h(P)=7, hence f(P)=g(P)+h(P)=0+7=7. The OPEN list contains only P and the CLOSED list is empty. See the below figure:



OPEN=[P]

CLOSED=[]

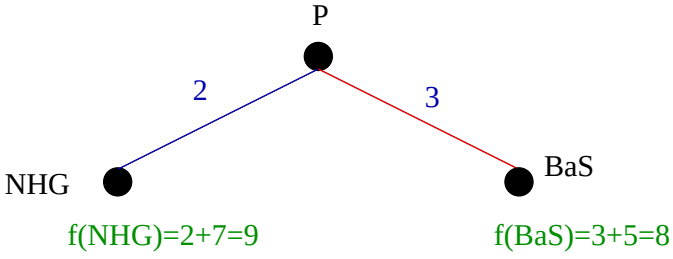
Next we expand P, add it to the CLOSED list, add its (not yet expanded) successors BaS, NHG to the OPEN list and calculate their f values. See the following figure:



OPEN=[BaS,NHG]

CLOSED=[P]

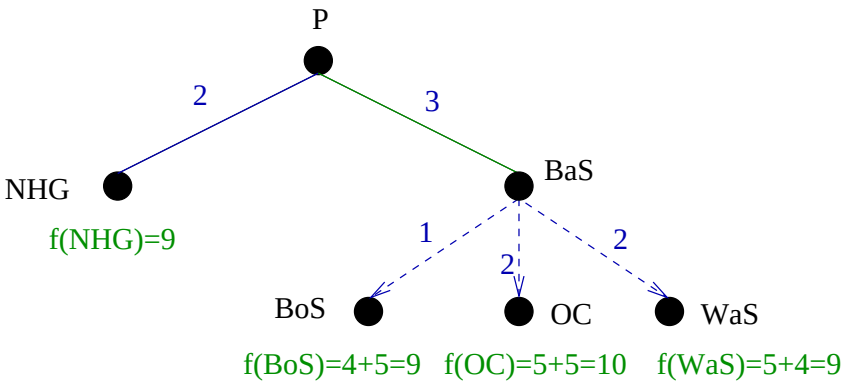
Next we consider the f values of NHG and BaS (our OPEN list) and choose the lowest one, that is, BaS. See the following figure:



OPEN=[BaS,NHG]

CLOSED=[P]

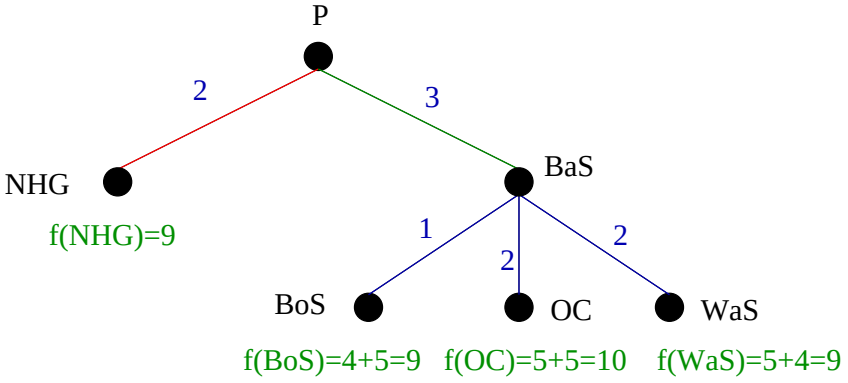
Next we expand BaS add it to the CLOSED list, and add its (not yet expanded) successors BoS, OC, WaS to the OPEN lise. We also calculate the f values of BoS, OC an WaS. See the following figure:



OPEN=[NHG, BoS,WaS,OC]

CLOSED=[P,BaS]

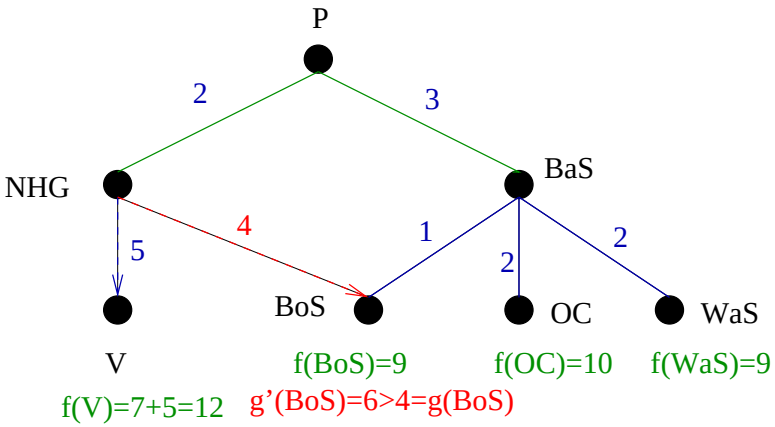
Next we consider the f values of NHG, BoS, OC an WaS (our OPEN list) and choose one of the lowest one, for example NHG. See the following figure:



OPEN=[NHG, BoS,WaS,OC]

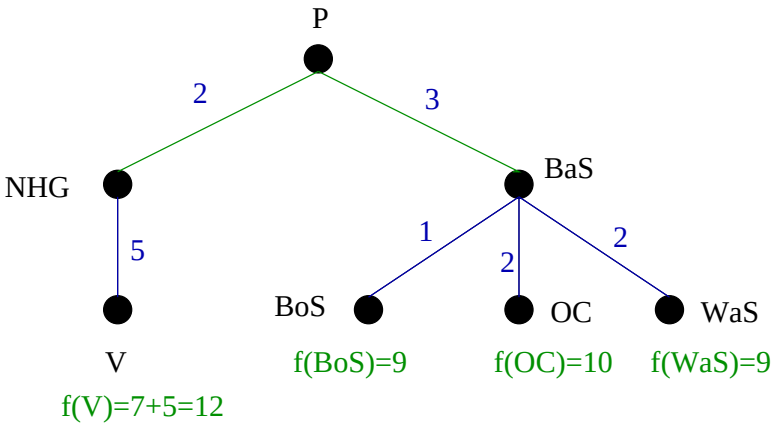
CLOSED=[P,BaS]

Next we expand NHG, add it to the closed list, add its successors which have neither been expanded nor generated (i.e., V) to the open list and calculate their f value (i.e, calculate f(V)). For the successors which have already been generated (but not expanded before), calculate the new cost g', and if it is lower than the previous g, then change the parent and modify accordingly the cost for the depending nodes. There is only one such node BoS, but g'(BoS)>g(BoS), so no modification is needed. See the following figure:



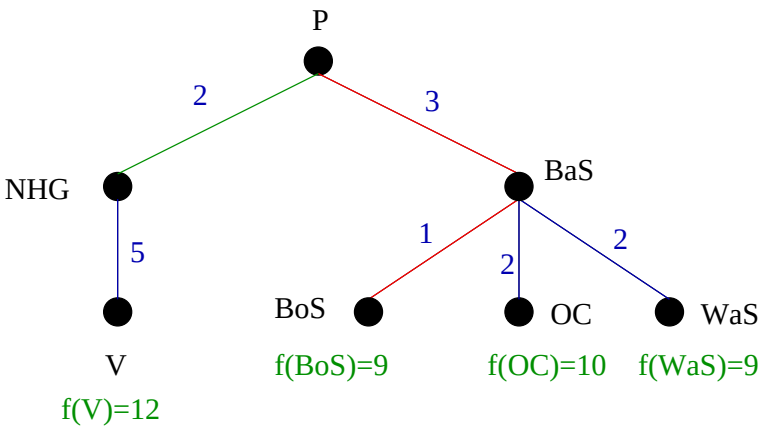
OPEN=[BoS,WaS,OC,V]
CLOSED=[P,BaS,NHG]

The next figure shows that there was no modification because of the alternative path to BoS:



OPEN=[BoS,WaS,OC,V]
CLOSED=[P,BaS,NHG]

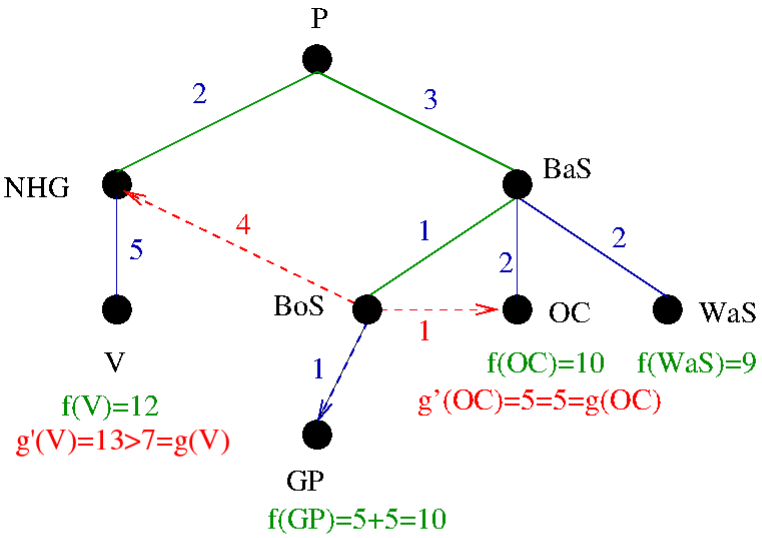
Next we consider the f values of V, BoS, OC and WaS (our OPEN list) and choose one of the lowest one, for example BoS. See the following figure:



OPEN=[BoS,WaS,OC,V]
CLOSED=[P,BaS,NHG]

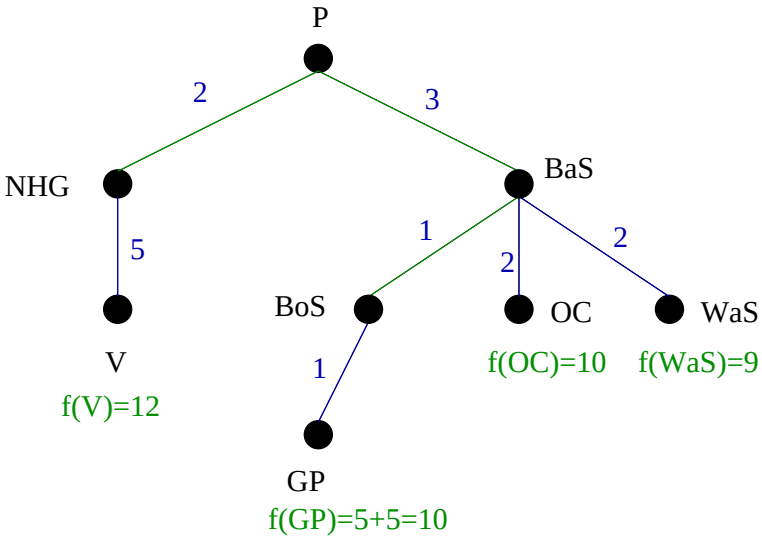
Next we expand BoS, add it to the CLOSED list, and add its successors which have neither been expanded nor generated before to the OPEN list and calculate their f values. There are two successors which have been generated

before: NHG and OC. Although NHG is in the CLOSED list, V is depending on NHG, hence we need to calculate g'(V) to see if we need to change the g value of V. Since g'(V)>g(V), no modification is needed. Similarly, no modification is needed due to an alternative path to OC because g'(OC)=g(OC). See the following figure:



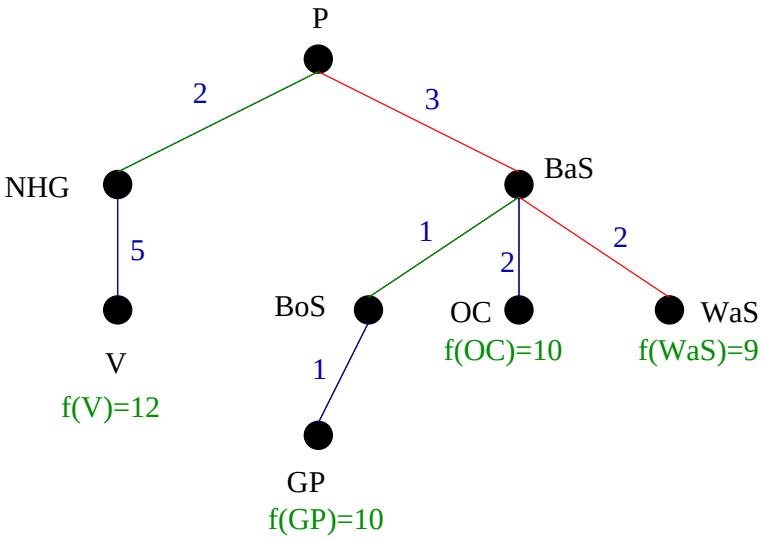
OPEN=[WaS,OC,GP,V]
CLOSED=[P,BaS,NHG,BoS]

The next figure shows that there was no modification because of the alternative path to NHG (or V) and OC:



OPEN=[WaS,OC,GP,V]
CLOSED=[P,BaS,NHG,BoS]

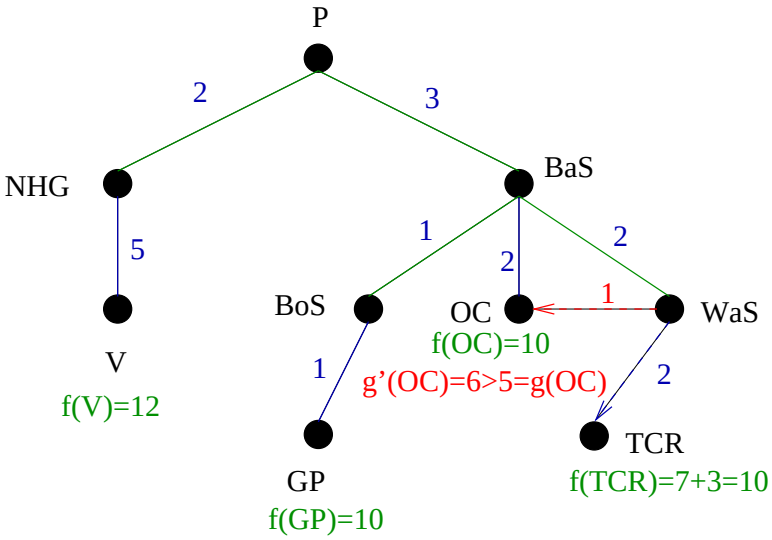
Next we consider the f values of V, GP, OC and WaS (our OPEN list) and choose the lowest one: WaS. See the following figure:



OPEN=[WaS,OC,GP,V]

CLOSED=[P,BaS,NHG,BoS]

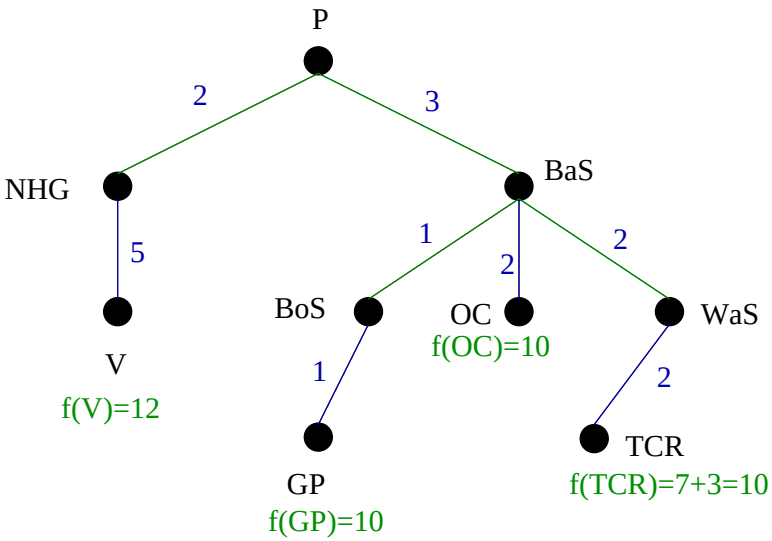
Next we expand WaS and add it to the CLOSED list. We generate its successors: OC and TCR. TCR has not been generated before, so add it to the OPEN list and calculate its f value. OC has been generated before and it is in the OPEN list (thus not yet expanded). Hence, we need to calculate g'(OC), and if it is lower than g(OC), then we need to change the parent and modify accordingly the cost for the depending nodes. But g'(OC)>g(OC). Hence, no modification is needed. See the following figure:



OPEN=[OC,GP,TCR,V]

CLOSED=[P,BaS,NHG,BoS,WaS]

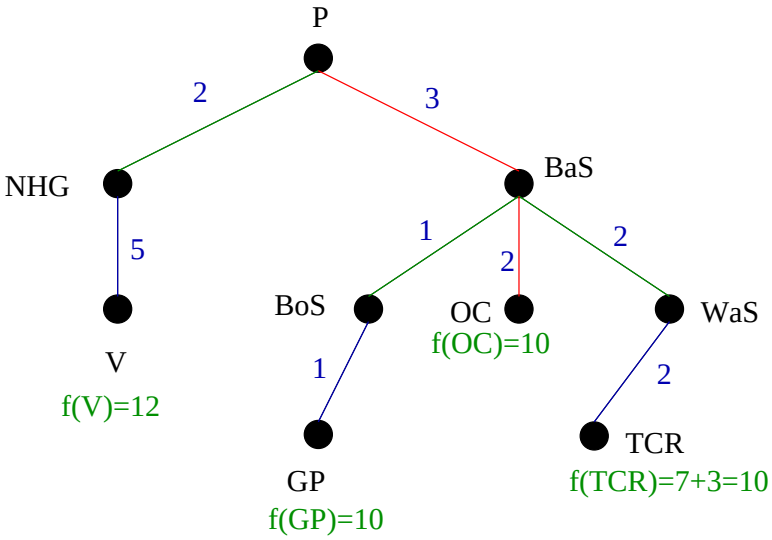
The next figure shows that there was no modification because of the alternative path to OC:



OPEN=[OC,GP,TCR,V]

CLOSED=[P,BaS,NHG,BoS,WaS]

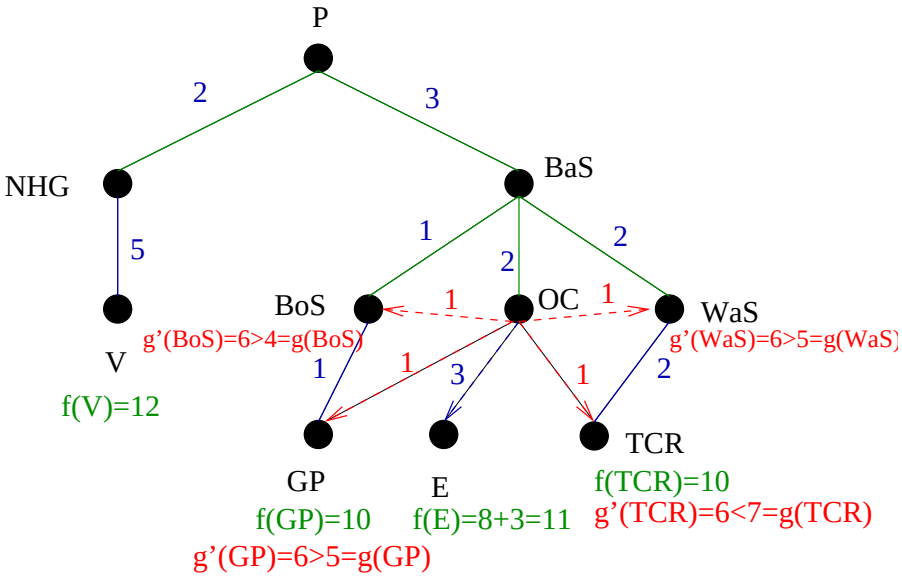
Next we consider the f values of V, GP, OC and TCR (our OPEN list) and choose one of the lowest one, for example OC. See the following figure:



OPEN=[OC,GP,TCR,V]

CLOSED=[P,BaS,NHG,BoS,WaS]

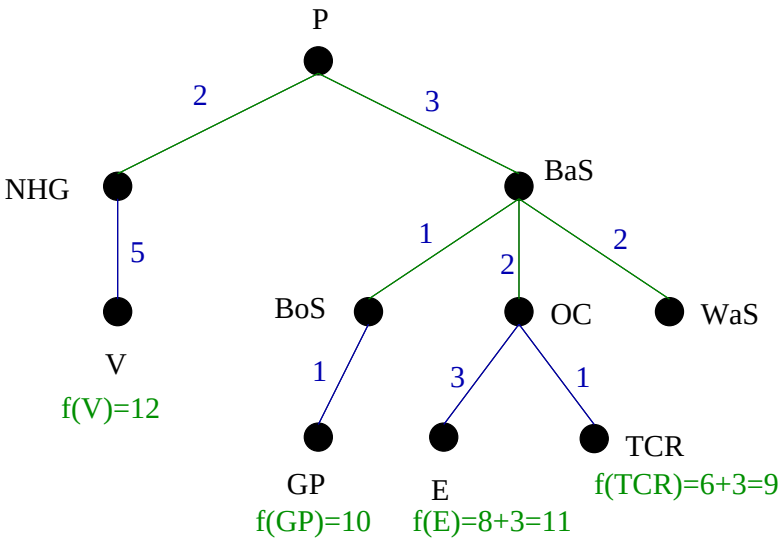
Next we expand OC and add it to the CLOSED list. We generate its successors. Among these successors only E has not been generated before. Hence add it to the OPEN list and calculate its f value. All the other successors have been generated before. The successors of OC are BoS, GP, E, TCR and WaS. In case of BoS, GP, E, and WaS we don't need to make any modifications due to the alternative path to these nodes because for all of them $g' > g$ (i.e., the new g value is larger). However, g'(TCR)<g(TCR). Hence, we need to change its parent from WaS to OC modify its f value and the cost for the depending nodes (thus the f values of these nodes will change too). See the following figure:



OPEN=[TCR,GP,V,E]

CLOSED=[P,BaS,NHG,BoS,WaS,OC]

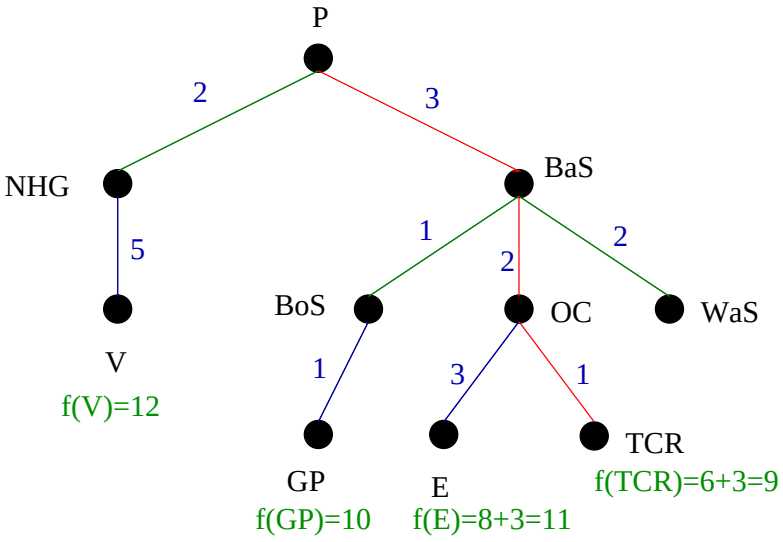
The next figure show the modified f values which are due to the alternative path to TCR:



OPEN=[TCR,GP,V,E]

CLOSED=[P,BaS,NHG,BoS,WaS,OC]

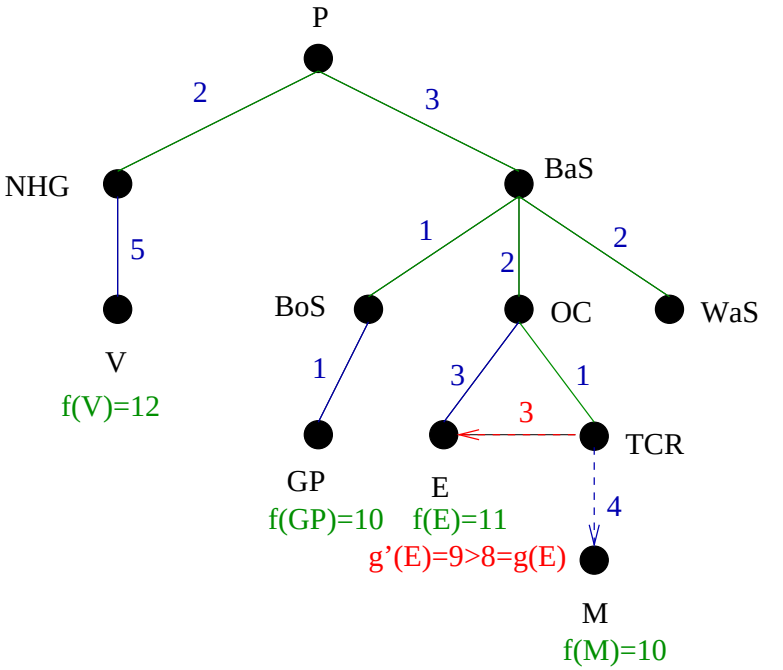
Next we consider the f values of V, GP, E and TCR (our OPEN list) and choose the lowest one: TCR. See the following figure:



OPEN=[TCR,GP,V,E]

CLOSED=[P,BaS,NHG,BoS,WaS,OC]

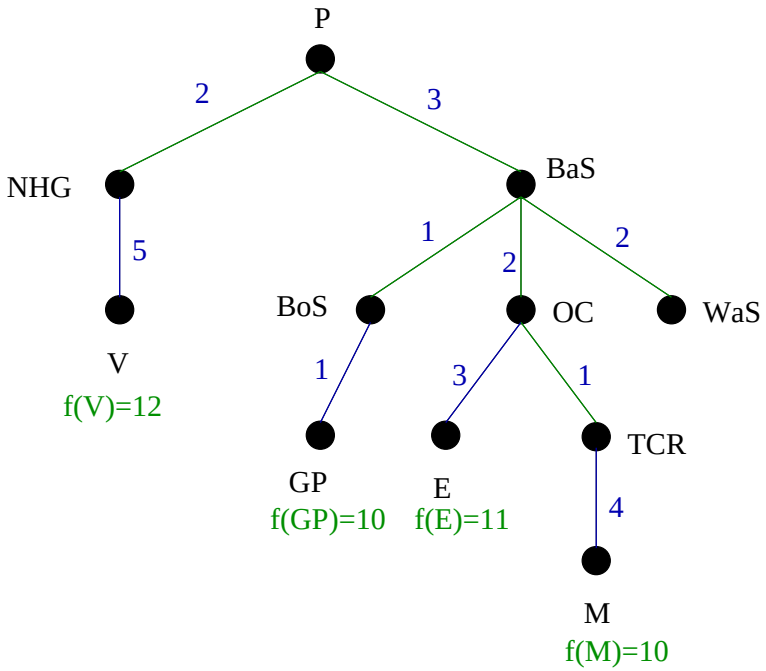
Next we expand TCR and add it to the CLOSED list. We generate its successors. Among these successors only M has not been generated before. Hence add it to the OPEN list and calculate its f value. The other successor E has been generated before. Hence, we need to calculate g'(E), and if it is lower than g(E), then we need to change the parent and modify accordingly the cost for the depending nodes. But g'(E)>g(E). Hence, no modification is needed. See the following figure:



OPEN=[M,GP,V,E]

CLOSED=[P,BaS,NHG,BoS,WaS,OC,TCR]

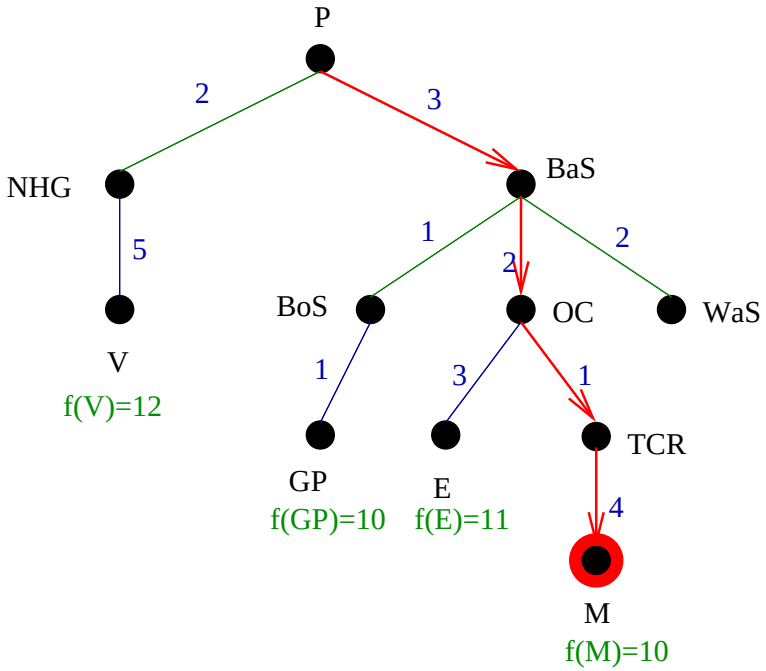
The next figure shows that there was no modification because of the alternative path to OC:



OPEN=[M,GP,V,E]

CLOSED=[P,BaS,NHG,BoS,WaS,OC,TCR]

Next we consider the f values of V, GP, E and M (our OPEN list) and choose one of the lowest one, for example M. We reached M and therefore the problem is solved. See the following figure:



Example 2: The 8-puzzle

The 8-puzzle is a sliding puzzle that consists of a frame of numbered square tiles with one tile missing. The goal of the puzzle is to get from a **Start state** to a **Goal state** by making sliding moves that use the empty space.

For example:

	2	3
1	4	5
8	7	6

Start state

1	2	3
8		4
7	6	5

Goal state

Possible estimated costs:

- The number of tiles in wrong position:

  $h(start) = 6$

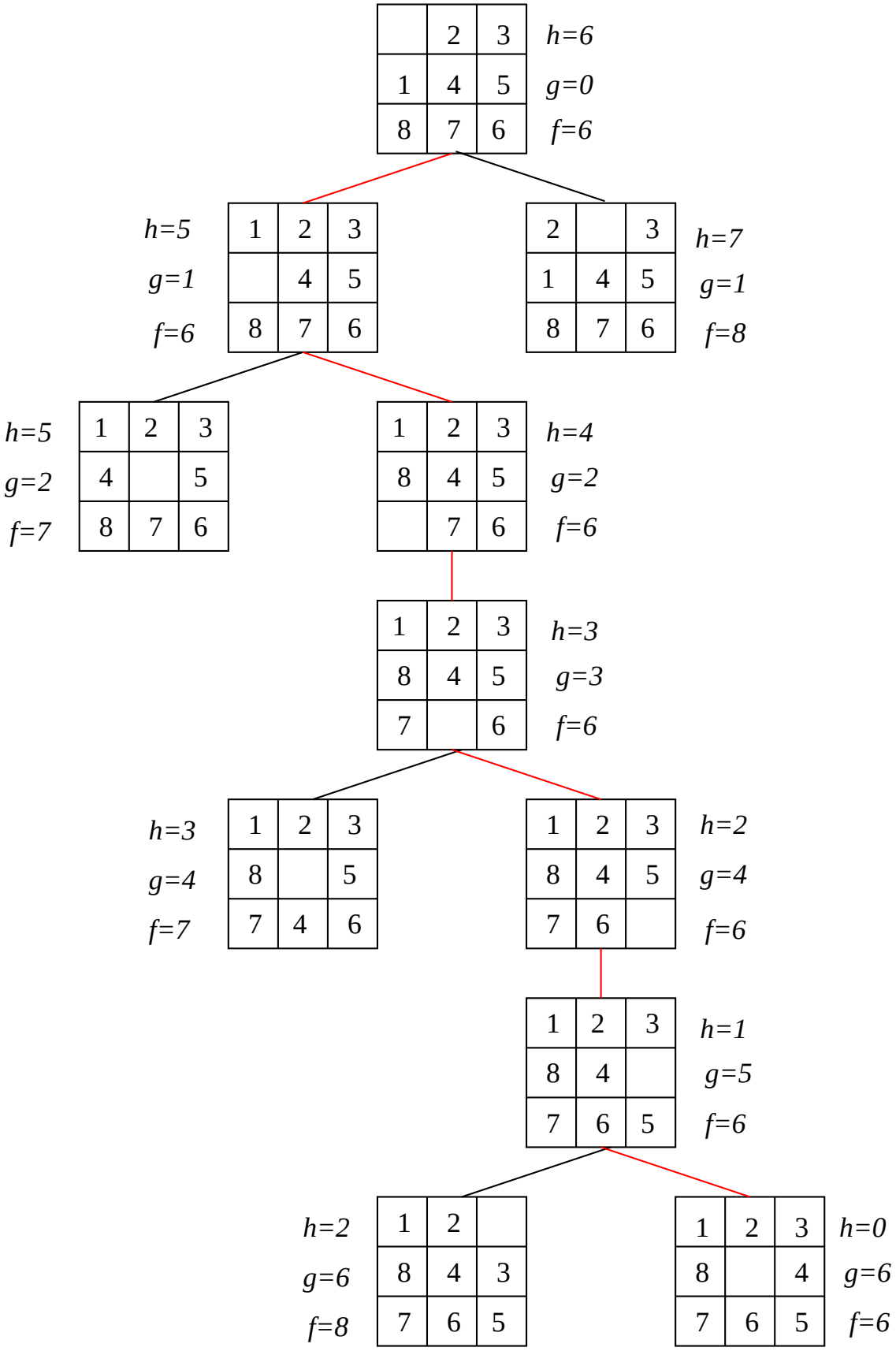
- The **Manhattan distance**:

  For each tile, calculate the Manhattan distance from its current position to its goal position and sum up these numbers. In our case:

  $h(start) = 1 + 0 + 0 + 1 + 1 + 1 + 1 + 1 = 6$

Example 2 – A* solution

Let $h =$ tiles in wrong position.



7.4 Algorithm of A*

Next we show the steps of the A* algorithm which were already presented for the London Underground Problem (Example 1):

Let OPEN be the list of generated, but not yet examined nodes.

1. Start with OPEN containing the initial state.
2. Do until a goal state is found or no nodes are left in OPEN:
   (a) Take the best node (lowest f value) in OPEN and generate its successors.
   (b) For each successor do:
      i. if it has not been generated before, **evaluate it** and add it to OPEN.
      What does evaluation mean?
      ii. otherwise change the parent, if **this path is better** and modify accordingly the costs for the depending nodes.
      When is a path better then the other?

How to perform step 2.(b)ii.

Let CLOSED be the list of generated and examined nodes.

For a node that has a copy in the OPEN list we only have to modify the parent and the g, f values.

A node that has a copy in the CLOSED list has already been examined.
So, we need to modify the g values (and the f values, too) of the nodes below this node.

We can do a depth-first traversal of the tree rooted at this node.

Each branch ends if there are no more successors or the present g value of a node is better.

Observations

$\boldsymbol{f(n)}$ is the **shortest estimated path** from the **start state** to the **goal state** that **passes** through state $\boldsymbol{n}$.

If the **cost of each step is 1**, A* will find the path consisting of **the fewest number of steps**. Indeed, the cost of any path from START to GOAL is the number of steps of the path.

Particular cases of A*

$g = 0 \Rightarrow$ **greedy best-first search**.

$h = 0$ and $g = 0 \Rightarrow$ **random search**. Indeed, all f values are 0, hence at each step the algorithm will randomly choose one node from the OPEN list.

Properties of h

- If h is a **perfect** estimator of the distance from the current node to the goal node, A* will never leave the optimal path.
- The better h estimates the real distance, the closer A* is to the **"direct"** path.
- If h never overestimates the real distance, A* is **guaranteed** to find the optimal solution.
- If h may overestimate the real distance, the optimal path can be found only if all paths in the search graph longer than the optimal solution are expanded.

The next section addresses the following question: What is better for A* a search graph or a search graph?

Search tree or search graph?

More general form ⇒ search graph.

More simple form ⇒ search tree.

In case of search graphs you need to update the g values for alternative paths. Because of the many possible paths in your graph A* for graphs is slower than A* for threes.

Search trees generate nodes faster, BUT duplicate nodes.

The question is:

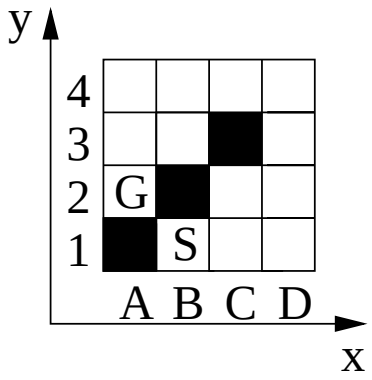
Is node duplication a problem?

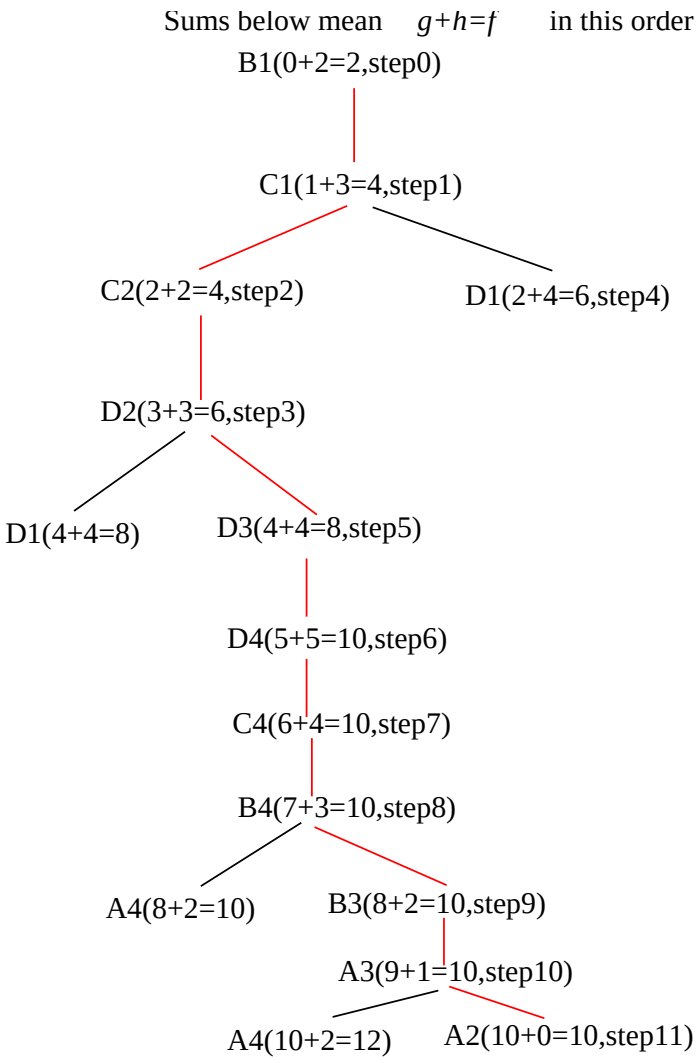
Yes, because for more complicated problems the computer may run out of memory if it needs to keep too many nodes in its memory.

Let us finish this section by presenting an example of A* for the maze problem.

Example 3: A* for the maze problem

The problem and its solution is presented in the next figure:





The solution of this problem is more complicated than the solution for the 8-puzzle problem. Please note that duplication of nodes occurs and a node and its duplicate(s) are allowed to be simultaneously in the OPEN list. All OPEN nodes are considered for choice. The algorithm chooses a node from the OPEN list having the smallest f value. It may happen that the chosen node has a duplicate (or duplicates) in the OPEN list with higher (or at least as large) f value. However, it is not allowed to have a node in the OPEN list and a duplicate in the CLOSED list, that is, if a node has been expanded then the node and all of its duplicates go to the CLOSED list.

Recommended reading

E. Rich and K. Knight: Artificial Intelligence, Section 3.3

7.5 Properties of A*

In this section we will study in more details the properties of A*. Some of these properties have been briefly mentioned before.

- Completeness
- Admissibility
- Monotonicity
- Informedness

An algorithm is called complete if it returns a solution in case one exists and reports that there is no solution if one does not exist. An algorithm is called optimal if it returns an optimal solution if one exists and reports that the there is no such solution if no optimal solution exists. Please note that a complete algorithm is NOT an algorithm working on complete solutions, it is a very different concept! In case of a graph search algorithm, the algorithm is called complete if it finds a path from the START node to the GOAL node when such a path exists and reports that there is no such path if one does not exists. In case of a graph search algorithm, the algorithm is called optimal if it finds an optimal (usually minimal cost) path from the START node to the GOAL node if such a path exists and reports that there is no such path if one does not exists.

Recall that **greedy search** minimizes the estimated cost $h(n)$ from n to the goal for the nodes in the OPEN list.

Greedy search is **neither optimal nor complete**, but it has a **low search cost**.

Recall that A* search minimizes $f(n) = g(n) + h(n)$, where $h(n)$ has the same meaning as in case of greedy search and $g(n)$ is the cost from the start node to n. We will see that A* is **optimal and complete** with a restriction on h.

Next we recall some properties of h which will be considered in more details later:

Properties of *h*

- If h is a **perfect** estimator of the distance from the current node to the goal node, A* will never leave the optimal path.
- The better h estimates the real distance, the closer A* is to the **"direct"** path.
- If h never overestimates the real distance, A* is **guaranteed** to find the optimal solution.
- If h may **overestimate** the real distance, the optimal path can be found only if all paths in the search graph longer than the optimal solution are expanded.

7.5.1 Admissibility

A heuristic h is **admissible** if it never overestimates the cost to reach the goal. Introducing this concept is important because we don't want to repeat all the time the phrase "it never overestimates the cost to reach the goal".

An admissible heuristic is **optimistic** because it "thinks" the cost of solving the problem is less than it actually is.

If h is admissible, $f(n)$ never overestimates the cost of the best solution through n. Indeed,

$$f(n) = g(n) + h(n) \leq g(n) + h_{\text{perfect}}(n) = f_{\text{perfect}}(n).$$

So A* will find the optimal solution because A* expands the nodes in order of increasing f (more details later).

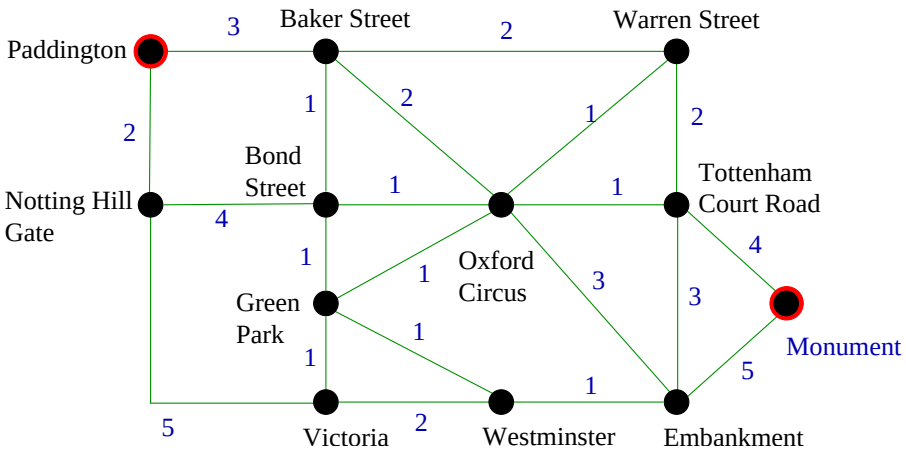
A* example

Recall the London Underground Problem where the goal is to find the shortest distance from Paddington (P) to Monument (M). The estimated distances from each node to M are given by the following table:

P	BaS	WaS	NHG	BoS	OC	TCR	GP	V	We	E	M
7	5	4	7	5	5	3	5	5	4	3	0

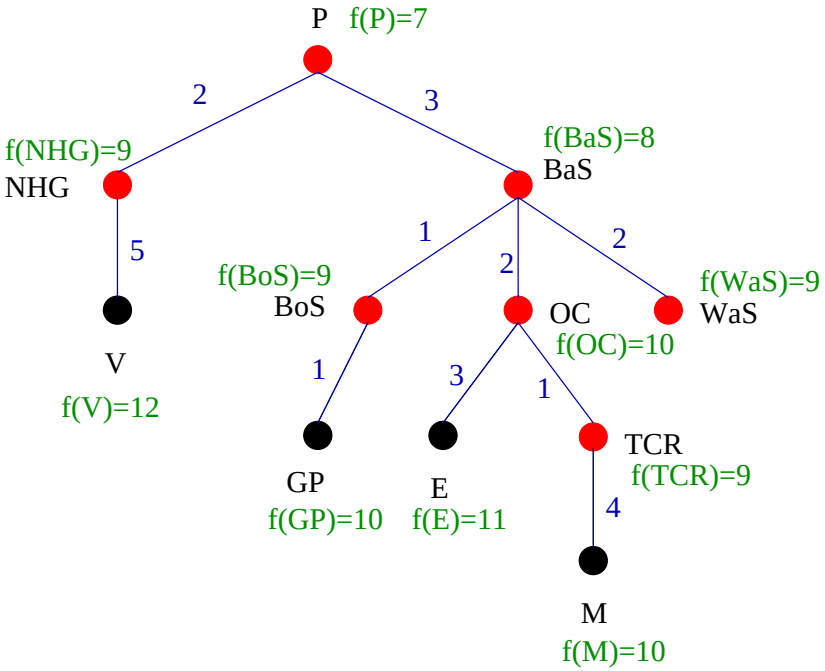
.

The problem is based on a part of the London Underground Map given by the graph



The numbers on the edges of this graph denote the pairwise distances between the corresponding stations.

The problem was solved by A* and the next graph show the last step of the algorithm



OPEN=[M,GP,V,E]

CLOSED=[P,BaS,NHG,BoS,WaS,OC,TCR]

The cost of the optimal path is $f(M) = 10$. Observe that all nodes with the f value less than the optimal value 10 have been expanded (the expanded nodes are exactly the ones in the CLOSED list) and OC with an optimal value of 10 has been also expanded before the algorithm selected the goal node M.

From now on denote by f^* the cost of the optimal path (called **optimal value**). The previous observations hold for a general A* algorithm with an admissible heuristic as well:

Observations

- A* expands all nodes with $f(n) < f^*$.
- A* might expand some of the nodes with $f(n) = f^*$ before selecting a goal node.
- A* expands no nodes with $f(n) > f^*$.

7.5.2 Completeness and optimality of A*

If h is admissible and there is a path with finite cost from the start node to a goal node, A* is guaranteed to terminate with a minimal-cost path to the goal.

Proof:

Let G be an optimal goal state with path cost f^*. Let G' be a suboptimal goal state (i.e., where $h = 0$; this may be a duplicate of the goal node for search trees or the goal node reached by a path with a g value which can be improved), that is,

$$\boldsymbol{g(G') = f(G') - h(G') = f(G') > f^*} \ .$$

Suppose A* selected G' from the OPEN list.

It can be shown (see "S. Russell and P. Norvig: Artificial Intelligence, A Modern Approach") that this is not possible. .

A* expands the nodes in order of increasing f, so it will eventually reach the goal state if the number of nodes with $f(n) < f^*$ is finite.
From now on assume that the heuristics of A* are admissible, that is, consider only heuristics which are admissible. We consider two further properties of these heuristics, **monotonicity** and **informedness**.

7.5.3 Monotonicity (Consistency)

If along any path in the search graph of A* the f-cost never decreases then the heuristic is said to be **monotonic** (or **consistent**).

A heuristic is monotonic (for example, the straight-line distance) if and only if it obeys the following **triangle inequality** :

$$f(A) \leq f(B) \Longleftrightarrow h(A) \leq h(B) + c(A, B),$$

for any nodes A, B and any path going through A and after that through B. Here, $c(A, B)$ denoted the cost of the sub-path from A to B.

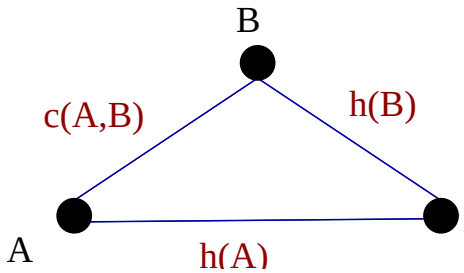
Proof: First note that $g(B) = g(A) + c(A, B)$. Hence, we have $f(A) \leq f(B)$ if and only if

$$g(A) + h(A) = f(A) \leq f(B) = g(B) + h(B),$$

which is equivalent to

$$g(A) + h(A) \leq g(B) + h(B) = g(A) + c(A, B) + h(B).$$

The latter relation is equivalent to $h(A) \leq h(B) + c(A, B)$.



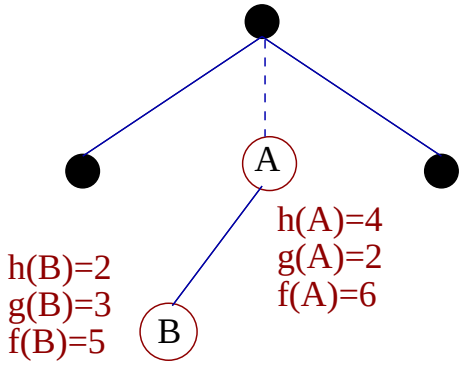
We will see that A* with a monotonic heuristics has nice "efficiency" properties. Therefore, it is natural to ask the following question arises:

Can we transform a nonmonotonic heuristic into a monotonic heuristic?

Restoring monotonicity

From now on assume that our problem is a cost minimisation one.

Let A and B two nodes in a search tree, so that A is the parent of B. Suppose $g(A) = 2$ and $h(A) = 4$, so $f(A) = g(A) + h(A) = 2 + 4 = 6$. Suppose $g(B) = 3$ and $h(B) = 2$, so $f(B) = g(B) + h(B) = 3 + 2 = 5$.



Since $f(B) < f(A)$, h is a nonmonotonic heuristic.

But any path through B passes through its parent A, too.

Replace the original f value of B (i.e., $g(B) + h(B)$) with $f(A)$ if it is lower than $f(A)$. Hence, the new f value of B is given by

$$f(B) = \max(f(A), g(B) + h(B)), \tag{1}$$

which is never decreasing along any path from the root (i.e., the start node). Do similar replacements for A* for an arbitrary cost minimisation problem.

Equation (1) is called the **pathmax equation**.

7.5.4 Efficiency of A*

The consistency property is important because it has nice "efficiency" properties described in the followings.

A* is **optimally efficient** for any consistent h function:

No other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly nodes with $f(n) = f^*$)

Monotonicity theorem:

If a monotonic heuristic h is used, when A* expands a node n, it has already found an optimal path to n.

Important: in this case, the current path to a node will never be better than some previously found path.

Consequences:

- No modifications will be made after a node is expanded. Hence, step 2.(b)ii. of the A* algorithm is not needed, that is the g (and consequently f) values do not need modifications because of the possible alternative paths.
- Searching a graph will be the same as searching a tree.

7.6 Informedness

If two versions of A*, A_1^* and A_2^*, differ only in $h_1 < h_2$ for all nodes that are not goal nodes then A_2^* **is more informed** than A_1^* (strictly speaking A_2^* is more informed than A_1^* if $h_1 \leq h_2$ for all nodes that are not goal nodes and there is a non-goal node for which strict inequality holds).

Example – the 8-puzzle

Let us compare A* based on the Manhattan distance and A* based on the number of tiles in wrong position. Which one is more informed? The Manhattan distance is more informed because each tile which is in wrong position has a Manhattan distance of at least 1. Therefore, by summing up the Manhattan distances of each tile, we get a number which is at least as large as the number of tiles in wrong position. It is easy to see that there are configurations where the Manhattan distance is larger than the number of tiles in wrong position.

The informedness property is important because more informed A* algorithms are more efficient. This is described in more details in the following theorem.

Informedness theorem:

If A_2^* is more informed than A_1^*, then at the termination of their searches on any graph having a path from the start node to a goal node, every node expanded by A_2^* is also expanded by A_1^*.

Therefore:

- A_1^* expands at least as many nodes as A_2^*.
- A more informed algorithm is more efficient.
- The most efficient heuristic is the perfect estimator!

Recommended reading

- N. Nilsson: Artificial Intelligence, A New Synthesis, Section 9.2.
- S. Russell and P. Norvig: Artificial Intelligence, A Modern Approach, Section 4.1.
- E. Rich and K. Knight: Artificial Intelligence, Section 3.3.

8 Simulated Annealing (SA)

Overview

- Introduction
- Basic structure of Simulated Annealing
- Properties of Simulated Annealing
- SA applied to SAT, TSP, NLP

The algorithms on complete solutions discussed up to now are of two types: They either guarantee to find the global optimum but they are too expensive (i.e., too time consuming) or they tend to get stuck in a local optima. Since there is almost no hope of speeding up the former type of algorithms (i.e., the ones which always find the global solution), the other possibility is to devise algorithms with a higher chance of escaping from a local optima. We will continue to discuss two major types of algorithms which are modifications of local search in order to escape from a local optima: Simulated Annealing and Tabu Search. The idea behind these two approaches is similar: Allow a higher possibility of exploring the search space in order to avoid getting stuck in a local optimal solution.

While Simulated Annealing uses a dynamic parameter called temperature which sets the changing probability of moving from point-to-point in the search space, Tabu Search uses a parameter called memory which forces the algorithm to explore unvisited areas of the search space. In this section we discuss Simulated Annealing.

8.1 Introduction

In the early 80's Kirkpatrick, Gelatt, and Vecchi and independently Cerny introduced the concepts of annealing in combinatorial optimisation.

In condensed matter physics annealing is known as a thermal process for obtaining low energy states of a solid in a heat bath.

The concept of Simulated Annealing is based on a strong analogy between the physical annealing process of solids and the problem of solving large combinatorial optimisation problems.

In order to reach a low energy state:

1. The temperature of the heat bath is **increased** to a maximum value at which the solid melts. In this liquid phase the particles of the solid arrange themselves randomly.
2. The temperature of the heat bath is carefully **decreased** until the particles arrange themselves in the ground state of the solid.

The goal of the physical annealing process is to reach the ground state of the solid where the particles are arranged in a highly structured lattice and the energy of the system is **minimal**.

The ground state of the solid is obtained only if the maximum temperature is sufficiently high and the cooling is done sufficiently slowly.

Otherwise the solid will be frozen into a **meta-stable** state rather than into the ground state.

8.2 Analogy with physical annealing

Methods for simulation of the physical annealing process can be directly applied to solve optimisation problems, based on the following table of analogies (between the first and second columns of the table):

Physical system	Optimisation problem
state	feasible solution
energy	evaluation function value
ground state	optimal solution
temperature	control parameter T
careful annealing	simulated annealing

8.3 Hill-Climbing and Simulated Annealing

Since simulated annealing it is an extension of the ideas of stochastic hill-climbing which is a modification of iterative hill-climbing, let us first present the structure of the latter two search procedures.

8.3.1 Iterative Hill-Climbing

1. Choose a starting solution i_{start} and initialise *best*
2. $k := 0; i := i_{start}$
3. Repeat until $k = MAX$:
   (a) Repeat until a local optimum is found:

i. select j as the neighbour of i with **best** value of the evaluation function $F(j)$

ii. if $F(j)$ is better than $F(i)$ then $i := j$
else a local optimum is found

(b) $k := k + 1$

(c) if j is better than *best* then *best* $:= j$

(d) $i :=$new random solution

The possibility of escaping from a local optima is provided by the outer loop which starts a new search, because the inner loop always returns a local optimum.

8.3.2 Stochastic Hill-Climbing

Stochastic hill-climbing is based on a **modification of** the procedure described by **iterative hill climbing**, described as follows:

- Instead of checking **all** neighbours of a current point i, select just one point j from the neighbourhood.
- Accept the new point with som e probability depending on the relative merit of the two points i and j.

Let us now describe the Stochastic Hill-Climbing Algorithm for maximisation problems.

Stochastic Hill-Climbing for Maximisation:

1. Choose a starting solution i_{start} and evaluate it
2. $k := 0; i := i_{start}$
3. Repeat until $k = MAX$:
   (a) select j as a neighbour of i
   (b) $i := j$ with probability $\frac{1}{1+e^{\frac{F(i)-F(j)}{T}}}$
   (c) $k := k + 1$

Here F denotes the evaluation function, MAX is the number of neighbours chosen at Step 3 and $T > 0$ is a fixed parameter which is not changing during the algorithm.

Compared to iterative hill-climbing, stochastic hill-climbing does not repeat its iterations from different randomly generated starting points (i.e., has only one loop). It is possible for the algorithms to accept worse points, but the newly selected point is accepted with some probability which depends on the relative merit of i and j and the the parameter T which remains constant during the algorithm. A natural question arises: What is the influence of T on the selection of j? Please note that the presented ideas are based on maximisation problems only, for minimisation the role of the current point an its neigbour "reverses", that is, $F(i)-F(j)$ in the probability formula changes to $F(j) - F(i)$, and consequently the following arguments are "reversed" as well.

Assume for example that $F(i) = 107$, $F(j) = 120$. In this case the neighbour j of the current point i is better and their relative merit is $F(i) - F(j) = -13$.

- if $T = 1$ (i.e., T is very small) the probability of acceptance is close to 1.
- as T increases the probability of acceptance decreases.
- if $T = 10^{10}$ (i.e., T is very large) the probability of acceptance is 0.5.

In conclusion: At one extreme, for very small values of T the acceptance of j is governed only by j being better or worse than i (regardless of the actual value of their relative merit), i.e., the stochastic hill-climber reverts to an ordinary hill-climber. At the other extreme, for very large values of T the probability of j being accepted is practically the same as the probability of being rejected, that is, the search becomes random. Therefore, we need to get an appropriate trade-off between too small values of T where the algorithm reverts to an ordinary hill-climber and too large values of T where the algorithm becomes purely random. Hence, we need to get a "good" value for T (neither too small nor too large) where the relative merit of i and j has a "balanced" influence on the chance of acceptance of j.

For example choose $T = 10$ and fix $F(i) = 107$. In this case the probability of acceptance depends only on $F(j)$. This seems a reasonably balanced choice, as shown by the below values:

- If $F(j) = 80$ the probability of acceptance is 0.06.
- If $F(j) = 100$ the probability of acceptance is 0.33.
- What happens if $F(j) = F(i)$? In this case the probability of acceptances is 0.5, i.e., j has the same chance of being accepted as being rejected.
- if $F(j) = 150$ the probability of acceptance is 0.99.

If $F(j)$ is very bad (small for maximisation), then the probability of acceptance is close to 0, that is, points of very low quality have almost no chance of being accepted. As $F(j)$ is improving (increasing for maximisation) the probability of acceptance is increasing. If j is worse than i, then it has a lesser chance of being accepted than being rejected. If j is of the same quality as i, then it has the same chance of being accepted as being rejected. If j is better than i, then it has a higher chance of being accepted than being rejected. If $F(j)$ is very good, then the probability of acceptance is close to 1, that is, points of very high quality are almost always accepted.
The analogy with physical annealing suggests the basic structure of the simulatead annealing (SA) algorithm.

8.3.3 Basic structure of Simulated Annealing

The following algorithm can be applied for both maximisation and minimisation problems. For maximisisation we have $e^{\frac{F(j)-F(i)}{T_k}} > random(0,1)$, while for minimisation $e^{\frac{F(i)-F(j)}{T_k}} > random(0,1)$ in (a)ii below. For maximisation the operator *isbetterthan* becomes $>$ and for minimisation $<$.

1. Choose a starting solution i_{start}
2. Initialize T_0,M_0
3. $k := 0; i := i_{start}$
4. Repeat until halting criterion is satisfied:
   (a) Repeat M_k times:
      i. generate j as a neighbour of i
      ii. if $F(j) isbetterthan F(i)$ then $i := j$
      else if $e^{-\frac{|F(i)-F(j)|}{T_k}} > random(0,1)$ then $i := j$
   (b) $k := k + 1$
   (c) Calculate M_k
   (d) Calculate the temperature T_k

Here $random(0,1)$ denotes a random number between 0 and 1; $T_k \geq 0$ is the main parameter of SA, usually called temperature by analogy with physical annealing (for all examples of SA this parameter is assumed to be nonnegative); and M_k is a control parameter which shows how many neigbours to consider at a given temperature. The notation $i := j$ means that the current solution becomes j, that is, j is accepted as the new solution. Better solutions are always accepted, while worse solutions are accepted on the relative merit of i and j. For maximisation the inequality in step (a)ii is equivalent to $F(j) > F(i) + T_k \log(random(0,1))$ (similar ideas can be used for minimisation). Hence, if T_k is large, then large deteriorations in the quality of solution are accepted. Indeed, $\log(random(0,1)$ is a negative number and hence $F(i) + T_k \log(random(0,1))$ is much smaller than $F(i)$, and any j with $F(j)$ smaller than $F(i)$, larger than $F(i) + T_k \log(random(0,1))$, but abitrarily close to $F(i) + T_k \log(random(0,1))$ is accepted. As T_k is decreasing, the gap between $F(i)$ and $F(i) + T_k \log(random(0,1))$ is decreasing and therefore only smaller deterioration in the quality of solutions are accepted. Indeed, $F(i) + T_k \log(random(0,1))$ is close to $F(i)$ and in this case any $F(j)$ with $F(j) < F(i)$ is close to $F(i)$ because $F(i) > F(j) > F(i) + T_k \log(random(0,1))$ and $F(i)$ is close to $F(i) + T_k \log(random(0,1))$. As $T_k \to 0$ almost (i.e., for temperature values close to 0) no deterioration in the quality of solutions is accepted, that is, only better solutions are accepted. This means that the algorithm is almost a local optimiser. Let us summarise this properties of simulated annealing in the next section, by comparing it to hill climbing.

8.4 Properties of Simulated Annealing

- In contrast to hill-climbing, simulated annealing accepts some **deterioration** in the quality of solutions. This helps avoiding local optima.
- Initially, at high temperatures, **large** deteriorations are accepted.
- As temperature decreases, only **smaller** deteriorations are accepted.
- As temperature approaches 0, SA behaves as **local optimisation**.
- Simulated annealing is a generalisation of local search.

8.5 Difficulties of SA

Simulated Annealing requires answers to two types of questions as outlined below:

- As in case of any search algorithm SA needs to answer the following **problem specific** questions:
  - What is a solution?
  - What are the neighbours of a solution?
  - What is the value of a solution?
  - How do we determine the initial solution?
- However, SA also needs answers to the following additional questions related to adjusting the **control parameters**:
  - How do we initialise T_k and M_k?
  - How do we determine cooling (get next value for T_k)?
  - How do we calculate M_k in each step?
  - What should be the halting criterion?

8.6 Simulated annealing for the Boolean Satisfiability Problem (SA-SAT)

The following algorithm is obtained from the GSAT algorithm by modifying it according to the general SA idea.

SA-SAT:

1. Repeat steps 2.-4. MAX_TRIES times:
2. Assign values to $X =< x_1, \ldots, x_n >$ randomly
3. $k := 0$
4. Repeat until $T_k < T_{min}$

   If the formula is satisfied, return X

   else

   $T_k := T_{max} \times e^{-kr}$

   compute the increase in the number of satisfied clauses δ, if x_i was flipped

   flip x_i with probability $\left(1 + e^{-\frac{\delta}{T_k}}\right)^{-1}$

   $k := k + 1$
5. Return "No solution found"

MAX_TRIES is the number of times the algorithms is restarted (i.e., maximum number of tries').

The major difference between GSAT and SA-SAT is that while the former one can make only one backward move (i.e., decrease in the number of satisfied clauses) because one backward move implies that in the next step we have a neighbour string which satisfies a larger number of clauses, the latter one can make an arbitrarily number of backwards moves.

The parameters of SA-SAT have a similar interpretation as in the case of other implementations of SA: T_{max} is the initial maximal temperature, T_{min} is the final minimal temperature, and r is the rate of decrease of the temperature, that is, how fast is decaying the temperature form T_{max} to T_{min}. The drop in the temperature is caused by the increasing values of k because e^{-kr} is a decreasing function.

Next, let us see if this algorithm is "really a simulated annealing algorithm", that is, it "corresponds" to the general idea of annealing (of course a slight adaptation of the basic idea of SA is needed here). The statement "better solutions are always accepted" corresponds to the line in the algorithm "If the formula is satisfied, return X" (note that better here is understood with respect to the evaluation function which assigns the value 0 to the Boolean function if it evaluates to FALSE and the value 1 if it evaluates to TRUE).

At very large temperatures (regardless of the value of δ) flipping (i.e., choosing the neighbour) has practically the same probability 0.5 as rejecting it. This can be interpreted as follows: We don't care about the quality of the solution (i.e., better or worse with respect to the evaluation function "number of satisfied clauses") we always accept it with the same probability 0.5. Hence, the algorithm behaves randomly (as the particles at very large temperatures of a melted solid in physical annealing). As T_k is decreasing, the probability of choosing a worse solution ($\delta < 0$) with respect to the evaluation function "number of satisfied clauses" is decreasing (because the probability function is an increasing function of T_k for $\delta < 0$). If T_k is very small, then the probability of choosing a worse solution (i.e., for which $\delta < 0$) is close to zero, while the probability of choosing a better solution (i.e., for which $\delta > 0$) is close to one. Therefore at very small temperatures the algorithm behaves as a local minimiser.

Also note that better the solution is (with respect to "number of satisfied clauses") higher its chance to being accepted (because the probability function is an increasing function of δ) as one would expect. From now on assume that a "better" solution means that it satisfies a larger number of clauses. At $\delta = 0$ (solution of same quality) the probability is 0.5, at $\delta > 0$ (better solution) the probability is larger than 0.5, and at $\delta < 0$ the probability is less than 0.5; as one would expect.

We can also use the following interpretation. Let p be a fixed probability of accepting a worse solution (i.e., for which $\delta < 0$). Then from the probability formula of the algorithm we get

$$-\delta = T_k \log(p^{-1} - 1), \tag{2}$$

where $-\delta$ is the decrease of satisfied clauses (since $\delta < 0$, we have $p < \frac{1}{2}$ and therefore $T_k \log(p^{-1} - 1) > 0$). Here the "decrease of satisfied clauses" means the "deterioration of solutions". Hence, from (2) it follows that (for a given probability p) initially at high temperatures large deteriorations in the quality of solutions are accepted; as the temperature decreases smaller deteriorations are accepted; and as $T \to 0$ practically no deteriorations in the quality are accepted (i.e, the algorithm behave as a local optimiser).

8.7 Simulated Annealing for the Travelling Salesperson Problem

For TSP the basic SA algorithm can be used.

There are only differences between implementations:

- the methods of generating the initial solution,
- the definition of a neighbourhood for a given tour,
- the selection of a neighbour,
- the methods for decreasing temperature,
- the halting condition,
- possible postprocessing.

For example: The starting point can be a random solution or the output of a local search, M_k should be proportional to the neigbourhood size, the postprocessing can be a local search algorithm which outputs a local optimum (as SA cannot guarantee this for all cooling procedures) etc.

8.8 Simulated Annealing for the Nonlinear Programming Problem

As the variables of NLP are continuos, the neighbourhood is often defined by using a Gaussian distribution for each variable:

$$\begin{aligned} \mathbf{x} &= (x_1, \ldots, x_n),\ l_i \le x_i \le u_i \\ x_i' &\leftarrow x_i + N(0, \sigma_i), \end{aligned}$$

where $N(0, \sigma_i)$ is and independent random Gaussian number with mean 0 and standard deviation $\sigma_i = \frac{u_i - l_i}{6}$.

For the maximisation of G_2:

$\mathbf{x}'$ is definitely accepted if $G_2(\mathbf{x}') > G_2(\mathbf{x})$,

otherwise with probability $e^{\frac{G_2(\mathbf{x}') - G_2(\mathbf{x})}{T}}$.

If the probability of acceptance is P, then $G_2(\mathbf{x}') = G_2(\mathbf{x}) + T \log P$.

Let us fix P and assume that x' is an accepted solution which is worse than x (i.e., $G_2(x') < G_2(x)$). The deterioration in the quality of solution is $G_2(x) - G_2(x') = -T \log P > 0$. Initially for large T higher deteriorations

in the quality of solutions are accepted (i.e., $G_2(x) - G_2(x')$ is large). As T decreases smaller deteriorations in the quality of solutions are accepted (i.e., $G_2(x) - G_2(x')$ becomes smaller). If $T \to 0$ the deterioration in the quality of solutions (i.e., $G_2(x) - G_2(x')$) tends to 0, i.e., the algorithm behaves as a local minimiser. Please note that the probability of acceptance increases as x' becomes better (i.e., $G_2(x')$ becomes larger).

Recommended reading

Z. Michalewicz & D.B. Fogel, How to Solve It: Modern Heuristics, Chapter 5. Escaping Local Optima, Section 5.1 Simulated Annealing.

9 Tabu search

Overview

- The main idea of tabu search
- Tabu search for SAT
- Tabu search for TSP

9.1 Introduction

Tabu search was proposed independently by Glover (1986) and Hansen (1986).

Tabu search is:

"a meta-heuristic superimposed on another heuristic. The overall approach is to avoid entrapment in cycles by forbidding or penalizing moves which take the solution, in the next iteration, to points in the solution space previously visited (hence tabu)."

9.2 Basic features

- Tabu search accepts non-improving solutions **deterministically** in order to escape from local optima (where all the neighbouring solutions are non-improving) by guiding a hill-climbing algorithm.
- Tabu search uses memory in two ways:
  - **to prevent** the search from revisiting previously visited solutions,
  - **to explore** the unvisited areas of the solution space.

Tabu search is based on the following simple idea: It uses a memory to force the search to explore unvisited regions of the search space. Recently visited solutions are memorized and become "tabu" (forbidden), i.e., they cannot be visited in the "near future", that is, for a number of moves which is called **length of memory** or **horizon**.

- Tabu search uses past experiences to improve current decision making.
- The use of a memory (a "tabu list") – to prohibit certain moves – makes tabu search a global optimiser rather than a local optimiser.

In contrast to Simulated Annealing, whose main idea is probabilistic, tabu search is predominantly deterministic (although some probabilistic elements may be added to it). The main ideas of Tabu Search can be best explained by using examples. We will start by presenting Tabu Search for SAT.

9.3 Tabu search for the Boolean Satisfiability Problem

9.3.1 Building up the memory

If Tabu Search is based on GSAT, then the evaluation function for the selection is "the number of satisfied clauses". It is possible to extend the evaluation function used by GSAT. For example we can generate an evaluation function by assigning a weights to each clause inverse proportionally to its length (because shorter clauses are more difficult to satisfy hence more important) and sum of these weights. For simplicity, from now on let us consider as an evaluation function "the number of satisfied clauses" used by GSAT and suppose that the tabu search is a modification of GSAT. This means that the steps of Tabu Search for SAT are the same as the steps of GSAT, except of following the memory at each step and forbidding the flips of variables corresponding to nonzero memory entries.

Consider a SAT problem of 8 variables. Suppose that the initial assignment is $\mathbf{x} = (0, 1, 1, 1, 0, 0, 0, 1)$

The neighbourhood of $\mathbf{x}$ (obtained by flipping the variables' values, one at a time) is examined.

Suppose that the best neighbour, is the one obtained by flipping the third variable. The algorithms is selecting this neigbour.

When flipping a variable we "make a note" of it, so that the same variable is not flipped a predefined number of steps (for example 5). This predefined number of steps is called the **length of memory** or **horizon**. The memory becomes:

0	0	5	0	0	0	0	0

.

In the next step:

- some variable (other than the third) is selected to be flipped,
- and the memory is updated, i.e., all non-zero values are decremented by one.

If in the ith position of the memory vector M we have j, then the ith variable is **tabu** (i.e., it cannot be flipped) for j steps/iterations/moves. This, can be written as $M(i) = j$. When $j \neq 0$ this also means that "the ith bit was flipped $5 - j$ steps ago" (for a horizon h, $h - j$ steps ago). If the previously nonzero ith position of the memory becomes zero, then the "tabu" is forgotten and there is no further restriction on the ith variable.

9.3.2 Using the memory

Puzzle:

Suppose that the initial assignment is $\mathbf{x} = (0, 1, 1, 1, 0, 0, 0, 1)$ and after 5 iterations the memory is:

3	0	1	5	0	4	2	0

.

What is the value of $\mathbf{x}$ after 5 iterations?

Solution:

Before solving this puzzle we make some remarks for the better understanding of the concept of memories: The "available" variables (i.e., the ones which can be flipped at next step) are x_2, x_5 and x_8. The variable x_1 cannot be flipped for for 3 steps, the variable x_3 for one step, the variable x_4 for 5 steps the variable x_6 for 4 steps and the variable x_7 for 2 steps.

If we assume that the **best** neighbour is obtained by flipping x_5, then after the 6th iteration the memory becomes:

2	0	0	4	5	3	1	0

.

Now, let us solve the puzzle. Since $M(1) = 3$, the variable x_1 was flipped $5 - 3 = 2$ steps ago, that is, at step 3. Since $M(3) = 1$, the variable x_3 was flipped $5 - 1 = 4$ steps ago, that is, at step 1. Since $M(4) = 5$, the variable x_4 was flipped $5 - 5 = 0$ steps ago, that is, at step 5. Since $M(6) = 4$, the variable x_6 was flipped $5 - 4 = 1$ steps ago, that is, at step 4. Since $M(7) = 2$, the variable x_7 was flipped $5 - 2 = 3$ steps ago, that is, at step 2. During the last 5 steps these variables were flipped exactly once and the other variables were not flipped at all (because 5 steps are needed to get from a memory value of 5 to 1). In conclusion, during the last 5 steps the variables corresponding to the zero memory entries were not flipped and the ones corresponding to the nonzero memory entries were flipped exactly once. Since the initial assignment is $\mathbf{x} = (0, 1, 1, 1, 0, 0, 0, 1)$, this means that we need to flip variable x_1, x_3, x_4, x_6, x_7 and leave variables x_2, x_5, x_8 unchanged. Therefore, after 5 iterations $\mathbf{x} = (1, 1, 0, 0, 0, 1, 1, 1)$.

9.3.3 Example

Consider the SAT problem

$$F = (x_1 \bigvee x_2 \bigvee x_3) \bigwedge (\overline{x}_2 \bigvee \overline{x}_3 \bigvee x_4) \bigwedge x_2 \bigwedge x_3 \bigwedge (\overline{x}_4 \bigvee x_5).$$

Let us solve this problem by using tabu search with a memory length 3. The solution is provided in the next table. The first column of the table gives the memory; columns 2-6 the values of the variables; columns 7-11 the values of the 1st, 2nd, 3rd, 4th, 5th clauses, respectively; column 12 the number of unsatisfied clauses; and column 13 the decrease in the number of unsatisfied clauses. At each step the algorithm checks the decrease in the number of unsatisfied clauses by flipping each variable and flips a variable (randomly if there is a draw) with the minimal decrease. If x_i is the currently flipped variable, then the ith position of the memory becomes 3. If the ith memory entry is j, then variable x_i cannot be flipped for the next j steps. If $j > 0$, then in the next step the memory value becomes $j - 1$. The steps of the algorithm should be self-explanatory:

Memory	x_1	x_2	x_3	x_4	x_5	F_1	F_2	F_3	F_4	F_5	#	decrease
00000	0	0	0	0	0	0	1	0	0	1	3	–
	1	0	0	0	0	1	1	0	0	1	2	1
	0	1	0	0	0	1	1	1	0	1	1	2←
	0	0	1	0	0	1	1	0	1	1	1	2
	0	0	0	1	0	0	1	0	0	0	4	-1
	0	0	0	0	1	0	1	0	0	1	3	0
03000	0	1	0	0	0	1	1	1	0	1	1	–
	1	1	0	0	0	1	1	1	0	1	1	0←
	0	1	1	0	0	1	0	1	1	1	1	0
	0	1	0	1	0	1	1	1	0	0	2	-1
	0	1	0	0	1	1	1	1	0	1	1	0
32000	1	1	0	0	0	1	1	1	0	1	1	–
	1	1	1	0	0	1	0	1	1	1	1	0←
	1	1	0	1	0	1	1	1	0	0	2	-1
	1	1	0	0	1	1	1	1	0	1	1	0
21300	1	1	1	0	0	1	0	1	1	1	1	–
	1	1	1	1	0	1	1	1	1	0	1	0←
	1	1	1	0	1	1	0	1	1	1	1	0
10230	1	1	1	1	0	1	1	1	1	0	1	–
	1	0	1	1	0	1	1	0	1	0	2	-1
	1	1	1	1	1	1	1	1	1	1	0	1←

9.3.4 Extensions

Aspiration criterion:

In specific circumstances an "outstanding" tabu, that is, a tabu neighbour with a much better evaluation value than any previously considered solution, can be accepted as the next point to be visited. In normal circumstances tabu search selects the best (non-tabu) neighbour even if it has a worse value than the current solution. To make the search more flexible, if an exceptional prohibited solution is found, that is, an "aspiration criterion is met, then it is selected as the next current solution overriding its tabu classification.

There are further ways for increasing the flexibility of tabu search:

- A probabilisic element can be also included in the algorithm, that is, better solutions to have a higher chance of being selected.
- Change the memory horizon during the search. For example if the algorithm reaches a promising area of the search space, then the memory length could be decreased.
- It is natural to relate the memory length to the number of variables. For example if there are n variables one may wish to choose the horizon to be $\sqrt{n}$. The number n here might have a different meaning, for example it can incorporate somehow the number of clauses too, that is, it might be some kind of "measure" of the "size" of the problem.

There is an even more interesting way for further diversifying the search, by considering **frequency-based** (or **long term**) memories. The memory discussed so far was **recency-based** (or **short term**) memory. Let is see, how the frequency-based memory can be defined:

The **frequency-based** memory H is defined as follows:

$$H(i) = j$$

means that "during the last h iterations of the algorithm variable i was flipped j times". The **memory length** (or **horizon**) h in this case is quite large compared to the recency-based memory.

For example after 100 iterations the long term memory may be

| 1 | 9 | 10 | 5 | 8 | 7 | 4 | 6 |. (3)

This means that if h is the horizon, then during the last h iterations the variables x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8 were flipped 1, 9, 10, 5, 8, 7, 4, 6 times, respectively. The horizon is the total number of flips, that is, $50 = 1 + 9 + 10 + 5 + 8 + 7 + 4 + 6$.

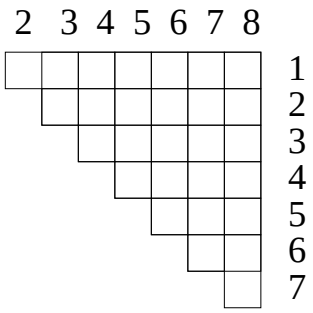
The frequency-based memory provides information about the frequency of flipping the variables and it is usually used in special circumstances only. For example, if all non-tabu moves lead to a worse solution, then the frequency based memory can be used to make the more frequent moves less attractive and make a decision accordingly about which option to choose. For example we can adapt the evaluation function for a solution $\mathbf{y}$ in the neighbourhood of $\mathbf{x}$ obtained by flipping the ith variable according to the formula $eval_p(y) = eval(y) - penalty(y)$, where $penalty(y)$ is an increasing function of $H(i)$, so that, more frequent flips have a higher penalty (recall that more frequent flips should be less attractive). A convenient choice for the penalty is $penalty(y) = \alpha H(i)$, where $\alpha > 0$ is a penalty factor. Consider the long term memory described in (3). Assume for example that we have a Boolean function with dozens of clauses and the non-tabu flips are x_1, x_3 and x_7. Suppose that the number of satisfied clauses by flipping x_1, x_3, x_7 are 27, 31, 29, respectively and $\alpha = 0.8$. The penalised evaluation function values for x_1, x_3, x_7 are $27 - 0.8 \cdot 1 = 26.2$, $31 - 0.8 \cdot 10 = 23$, $29 - 0.8 \cdot 4 = 25.8$. The variable with the highest $eval_p$, is selected for flipping. Hence, the next current solution is obtained from x by flipping x_1. Note that this happens despite of the fact that without penalizing the evaluation function the neigbour obtained by flipping this variable would have had the worst score (among the non-tabu neighbours).

9.4 Tabu search for ther Travelling Salesperson Problem

Let us now consider tabu search for TSP. Neighbourhood can be defined as swapping two **cities** in a tour.

For example in case of a TSP problem with 8 cities, a tour $(2, 4, 7, 5, 1, 8, 3, 6)$ will have $28 = (8 \cdot 7)/2$ neighbours (equal to the number of pairs of different cities).

The structure of the recency-based memory is based on the table:



where each cell of the table should be completed with the number of steps for which swap of the corresponding cities is prohibited (i.e., is a tabu). The largest cell value will be the memory lenghth (i.e., horizon).

The structure of the frequency based memory is based on the same table, but in this case each cell should be completed with the number of swaps of the corresponding cities during the last h steps, called **horizon** or **memory length**. The sum of the cell values of the table is equal to the horizon h.

9.4.1 TSP example

Assume for example that the current tour after 500 iterations is $(7, 3, 5, 6, 1, 2, 4, 8)$.

The recency-based memory is:

2	3	4	5	6	7	8	
0	0	1	0	0	0	0	1
	0	0	0	5	0	0	2
		0	0	0	4	0	3
			3	0	0	0	4
				0	0	2	5
					0	0	6
						0	7

Let us interpret these numbers. By the jth column we will mean the column below j (note that there is no 1st column). The largest value 5 is the horizon and corresponds to the 2nd row and 6th column (i.e., it is the value of the cell at the intersecion of the 2nd row with the 6th column; the meanings of the later correspondences are similar). It means that the most recent swap was the swap of cities 2 and 6 and these cities cannot be swapped in the next 5 steps. In general, if $M(i, j) = k$ (memory value of the cell (i, j) at the intersection of the ith row with the jth column, then cities i and j cannot be swapped for k steps and if $k \neq 0$ they were swapped $h - k$ steps ago, where h is the horizon; in this case $h = 5$. The second largest value is 4 and corresponds to the 3rd row and 7th column. It means that the cities 3 and 7 were swapped $1 = 5 - 4$ steps ago and cannot be swapped in the next 4 steps. The third largest values is 3 and corresponds to the 4th row and 5th column. It means that the cities 4 and 5 were swapped $2 = 5 - 3$ steps ago and cannot be swapped in the next 3 steps. The fourth largest values is 2 and corresponds to the 5th row and 8th column. It means that the cities 5 and 8 were swapped $3 = 5 - 2$ steps ago and cannot be swapped in the next 2 steps. The fifth largest values is 1 and corresponds to the 1st row and 4th column. It means that the cities 1 and 4 were swapped $4 = 5 - 1$ steps ago and cannot be swapped in the next 4 steps.

The frequency-based memory is:

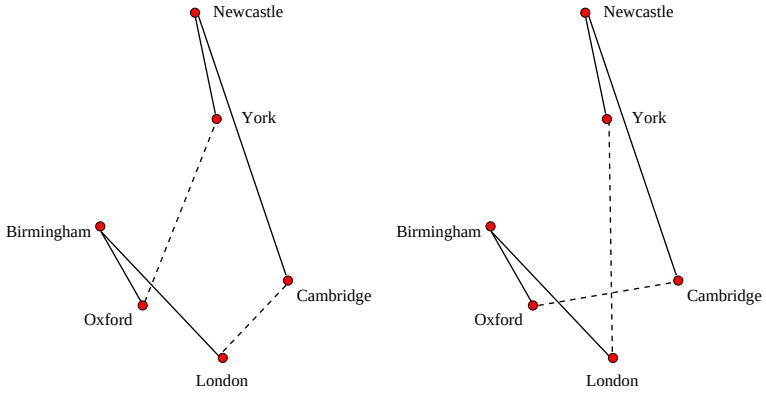
2	3	4	5	6	7	8	
0	2	3	3	0	1	1	1
	2	1	3	1	1	0	2
		2	3	3	4	0	3
			1	1	2	1	4
				4	2	1	5
					3	1	6
						6	7

Let us interpret these numbers. By cell (i, j) we will mean the cell at the intersection of the ith row with the jth column. The memory value $H(i, j) = k$ (memory value of the cell (i, j)) in this table means that during the last h steps the cities i and j were swapped k steps, where h is the horizon. The horizon is the sum of the cell values which is 52. Hence, for example, in the last 52 steps cities 3 and 7 were swapped 4 times (because the value of cell $(3, 7)$ is 4), while cities 7 and 8 were swapped 6 times. If you want to know how many times a city i was swapped with the other cities in the last $h = 52$ steps, then you should sum up all numbers in the ith row and ith column. For example city 3 was swapped $16 = (2 + 3 + 3 + 4 + 0) + (2 + 2)$ times with the other cities in the last 52 steps.

However, swapping two cities (i.e., the neigbourhood based on the 2-swap operation) is not the best neighbourhood definition (neither for tabu search nor for simulated annealing). Many researchers considered larger neigbourhoods, for example the one based on the 2-interchange operation (i.e., the neigbours of the current tour are all tours obtained by using 2-interchange operations).

Neighbourhood based on 2-interchange

Recall the next figure which describes this 2-interchange operation on which a better neighbourhood for tabu search is based.



For a general TSP if the current tour contains edges AB and CD, then the 2-interchange operations means replacing these edges with AC and BD.

Tabu search based on 2-interchange

Next we only sketch the tabu search algorithm based on the 2-interchange operation.

Repeat MAX-TRIES times:

1. Generate a tour
2. Repeat ITER times
   (a) identify a set of **2-interchange** moves
   (b) select & make a **2-interchange**
   (c) update tabu list
   (d) update local best tour info
3. Update global best tour info

Here ITER means the number of neighbours (i.e., 2-interchanges) to be considered and the algorithm is restarted MAX-TRIES times. There are many alternative ways of implementing the tabu list and aspiration criteria can also be added which override the tabu status.

9.5 Advantages and drawbacks of tabu search

Advantages:

Tabus search generally finds good solutions for optimisation problems.

Drawbacks:

Tabu list construction is problem specific.

There is no guarantee of finding a global optimum.

9.6 Comparison of tabu search and simulated annealing

- Both were designed to escape local optima.
- Both work on complete solutions.
- Tabu search only selects worse moves if it is stuck in a local optimum, whereas simulated annealing can do that all the time.
- Simulated annealing is stochastic.
- Tabu search is deterministic.
- The parameters must be carefully chosen for both.

Recommended reading

Z. Michalewicz & D.B. Fogel, Section 5.2 Tabu search

10 Genetic Algorithm Basics

Overview

1. Introduction
2. The terminology borrowed from Nature
3. Representation, selection, crossover, mutation
4. Evaluation
5. Constraints

10.1 Introduction

Traditional optimisation methods fail when

- there are **complex**, nonlinear relationships between the parameters and the value to be optimized;
- the goal function has many **local** extrema;
- resources are **limited**.

Modern heuristic optimisation methods are employed in such cases.

10.2 Evolutionary Algorithms (EAs)

EAs transpose the notions of natural evolution to the world of computers and imitate natural evolution.

EAs **evolve** solutions to a problem by maintaining a **population** of potential solutions.

EAs are based on the following principle:

Survival of the fittest: Fit individuals live to reproduce, weak individuals die off.

Examples of EAs are: Genetic Algorithms, Evolutionary Programming, Genetic Programming, Evolution Strategies

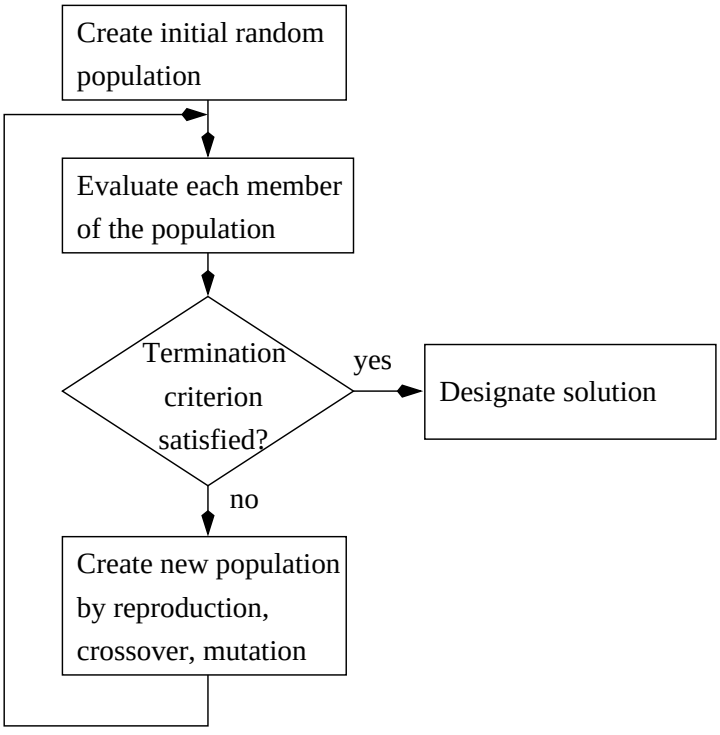
EAs are based on an analoy with the principles of nature, described in the next section.

10.2.1 Nature – Evolutionary algorithms

Nature	Evolutionary algorithms
Individual	Solution to a problem
Population	Collection of solutions
Fitness	Quality of a solution
Chromosome	Representation of a solution
Gene	Part of representation of a solution
Crossover	Binary search operator
Mutation	Unary search operator
Reproduction	Reuse of solutions
Selection	Keeping good subsolutions

The population of a Genetic Algorithm is maintained by using unary search operators (called **mutation**) and binary search operators (called **crossover**), and **reproduction and selection** through several **generations** in the hope of finding good enough solutions.

10.3 Genetic Algorithm (GA)



Previously evolved good parts of solutions (**schemata**) can be transferred to subsequent generations through crossover.

10.3.1 Representation

The representation is the first step of designing a GA.

The **representation** together with the **genetic operators** bound the exploration of the search space.

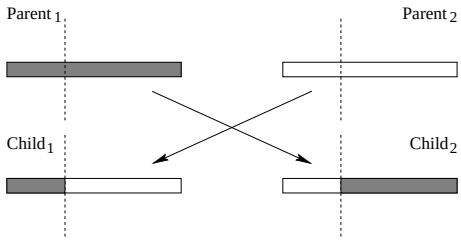
The **basic representation** of GA is to represent the **individuals** (i.e., candidate solutions) as fixed length bit strings. For example: 1011011001

Incorporating **domain knowledge** into the representation helps guiding the evolutionary process toward good solutions.

10.3.2 Crossover and Mutation

The most common crossover is the one-point crossover. For example one-point crossover is used for the basic representation. It is described by the following figure.

One-point crossover:



Example:

Consider the bit strings $Parent_1 = 0010110111$ and $Parent_2 = 1011100011$. Apply a one-point crossover between the fifth and sixth bit. What are the children (or offsprings) of this crossover?

Solution: $Child_1$ is obtained by taking the first fifth bits from $Parent_1$ and the next fifth bits from $Parent_2$. $Child_2$ is obtained by taking the first fifth bits from $Parent_2$ and the next fifth bits from $Parent_1$. Since

$$Parent_1 = \mathbf{00101\,10111}$$

and

$$Parent_2 = 10111\,00011,$$

we get

$$Child_1 = \mathbf{00101}\,00011$$

and

$$Child_2 = 10111\,\mathbf{10111}.$$

Mutation consists of applying minor changes to one individual, for example flipping a bit.

Example: A mutation of 1011011001 is 1010011001, obtained by flipping the fourth bit.

10.3.3 Evaluation: Fitness Assignment

In case of GAs the evaluation value of an individual is also called **fitness**.

Possibilities for fitness assignment are:

- We define a **fitness function** and incorporate it into the genetic algorithm.
- Fitness evaluation is performed by separate **dedicated** analysis software.
- There is no explicit fitness function, but a **human** evaluator assigns a fitness value to the solutions presented to him.
- Fitness can be assigned by **comparing** the individuals in the current population.

10.3.4 Selection

Only selected individuals of a population are allowed to have offspring.

Selection is based on **fitness.** Next we describe the possible ways of selection.

Selection schemes:

- Fitness proportional selection
  - Individuals are selected based on the value of their fitness. Individuals with a higher fitness will be more likely to be selected. If f_i is the fitness of individual i in the population, its probability of being selected is
    $$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j},$$
    where N is the number of individuals in the population.
- Ranked selection
  - The fitness proportional selection is not good in the cases when the fitnesses are too different. For example if individual i has a probability of 90% for being selected (i.e., $p_i = 0.9$), then the other individuals will have very limited chance to be selected (less than 10% because $\sum_{i=1}^{n} p_i = 1$);
  - In this case a **ranked** selection is needed. The fitness of the individuals are ranked and they are selected with a probability which is proportional to their rank. The drawback of this selection method is that it is slower because the best individuals are not too different from the other ones.
- Tournament selection
  - A few individuals are chosen randomly (i.e., tournaments are organised) and the best one is selected for crossover. The process is repeated several times (i.e., several tournaments are organised and the winner of each one is selected for crossover). In order to avoid selecting weak individuals tournament size can be increased. For diversifying the search (to increase exploration and have a higher chance to find good individuals) chances can be given for individuals weaker then the best one in each tournament. For example the best individual in a tournament can be selected with a high probability p (ex. $p = 0.9$), the second best with the probability $p(1-p)$, the third best one with the probablity $p(1-p)^2$, the fourth best one with the probablity $p(1-p)^3$, and in general the kth best one with a probability $p(1-p)^{k-1}$.

10.3.5 Constraints

In the simplest case, constraints occur as well-defined intervals for design parameters.

Methods for handling constraints in GAs:

- Reject individuals that violate constraints (i.e., infeasible individuals).
- Repair infeasible individuals.
- Penalize infeasible individuals.
- Incorporate constraints in the representation.

These methods will be described in more details later.

10.3.6 Advanced Issues (optional, not examinable)

- **Multiobjective GAs** – optimise a vector function
  - Example: good performance at low cost.
- **Parallel GAs** The main idea of parallel algorithms is to distribute the tasks between different processors and execute the tasks in the same time.
  - **Master-slave** model In the master-slave processor there is only one population, but the evaluation of fitness is shared by several processors.
    - The master processor stores the population, executes the genetic operations (crossover and mutation) and distributes individuals to the slave processors.
    - The only task of the slaves is to evaluate the fitness of the individuals.
  - **Multiple subpopulations** with migration – coarse or fine grained parallelism. There is one subpopulation at each processor, where a process takes place.
    - The fine grained parallelism are appropriate for massively parallel computers and corresponds to a spatially well structured population (i.e., grid). Crossover and selection is restricted to small neighbourhoods, but neighbourhoods' overlapping allows some interaction between all individuals. In the ideal case there is only one individual at each processor.
    - Multiple population GA's consist on several subpopulations which exchange individuals occasionally. The process of exchanging individuals is called migration. Each process is a simple GA for a subpopulation with (infrequent) communication between the subpopulations.
- **Diversity**
  - Premature convergence to a local optimum is a major problem.
  - Solutions: niching, speciation, parallelism
    - Niching methods maintain the population's diversity by investigating several peaks simultaneously and therefore preventing the GA from getting stuck in a local optimum.

* Niching methods work by using an analogy with natural ecosystems. Niches are subspaces supporting different types of life. A species is a group of individuals having similar biological characteristics that can interbreed among themselves, but they are unable to breed with individuals not belonging to their group. Each niche has limited resources which must be shared among its population. Species can be described as similar individuals with respect to similarity measures. Fitness sharing reduces the fitness in densely populated regions. The shared fitness f_i' of an individual i is reduced by dividing the original fitness f_i of the individual i with the number m_i of individuals with whom the fitness is shared, called **niche count**. This idea is expressed by the formula

$$f_i' = \frac{f_i}{m_i}.$$

The niche count is given by

$$m_i = \sum_{j=1}^{N} sh(d_{ij}),$$

where N is the population size, d_{ij} is the distance between the individual i and j, while the sharing function 'sh' measures the similarity level between two individuals. We have $sh(d_{ij}) = 1$ if $d_{ij} = 0$, $sh(d_{ij}) = 0$ if d_{ij} is higher then a given threshold of dissimilarity $\rho > 0$ and has an intermediate value at an intermediate level of dissimilarity.

10.3.7 GAs in Engineering Design (optional, not examinable)

Design is an engineering activity for creating **new** technical structures characterized by **new** parameters, aimed at satisfying predefined technical requirements.

A **design specification** is an essential qualitative and quantitative characteristic that set criteria (such as performance requirements, dimensions, weight, reliability, ruggedness) to be satisfied in designing a component, device, product, or system.

A **design description** is a description of a system in some formal way by using various graphs, diagrams engineering drawings and formulas.
Engineering design can be seen as the transformation of **design specifications** into **design descriptions**.

Modelling design helps building computer programs that assist (if not yet automatise) human design.

Design can be seen as the **search** for a suitable or optimal construction.

10.3.8 Shape Optimisation (optional, not examinable)

Values of the shape variables have to be determined, which result in an **optimal** value of some target parameter.

Shapes can be described by a structured set of shape parameters; scalars, vectors, or discrete representations such as pixels.

A **general** representation might lead to poor results.

One could use a **pixel-based** representation, when specific genetic operators need to be developed.

10.3.9 Remarks

GAs are considered **science** by some, **craft** by others, and **art** by some others.

The basic notions are very easy to understand.

BUT note that the performance of GAs depends **A LOT** on the chosen representation, evaluation, genetic operators.

The more domain knowledge is incorporated, the more likely the GA's success is.

11 Genetic algorithm for the Travelling Salesperson Problem

Overview

- Heuristic optimisation methods for solving TSP
- GA representations and operators:
  - Adjacency representation
  - Ordinal representation
  - Path representation
  - Matrix representation
- Incorporating local search methods
- A fast GA for TSP

11.1 Heuristic optimisation for the the Travelling Salesperson Problem

No algorithm is known which solves TSP in polynomial time, that is, there is no algorithm whose time of solving the problem is a polynomial function of the size of the problem (or equivalently the number of cities).

Therefore we are seeking for methods which will not find a perfect solution, but generally find a very close to (global) optimal solution. There may be exceptionally unfortunate situations where the solutions is far from the perfect solution.

For example an optimal solution for a "large" problem may take **5 hours** (or several hours) on the best computer, while

Finding a very close to optimal solution might take only a few **seconds** on a personal computer.

TSP is related to other problems as well, such as scheduling and partitioning. Therefore, it is an important problem to consider. Besides, it is a good problem to show the importance of incorporating domain knowledge into the solution of the problem. There are many algorithms which provide approximate solutions for TSP found in the last few decades. Let us list some of them.

Algorithms that generate approximate solutions:

- nearest neighbour, other greedy
- Local optimisation type algorithms:
  - 2-opt, Lin-Kernighan

- Evolutionary algorithms:
  - Several approaches were proposed and new methods are still coming up.
  - However, no perfect algorithm has been found.
  - Often, the publications don't provide compuational time for solving the TSP, only the number of used function evaluations. This makes difficult to compare the methods because of the different time complexities of these evaluations.

11.2 Performance of Genetic Algorithms for the Travelling Salesperson Problem

Approaches differ in the used **representation** and **variation operators**.

The main problem is that the GA is too slow to solve really big problems (tens of thousands of cities).

TSP with hundreds of cities can be solved by global optimisation relatively quickly (within hours) and therefore heuristic approaches should concentrate on larger problems.

Comparisons are done on publicly available test problems with documented optimal or best-known solution (using the library TSPLIB).

11.3 Variation operators of Genetic Algorithms for the Travelling Salesperson Problem

Although the evaluation function for TSP is very simple: the length of a tour, the ways of encoding tours and the variation operators corresponding to these representations are less obvious.

The most natural representation of a tour is a permutation, that is, to list the cities in order of their appearance in the tour. However, during the early development of evolutionary algorithms the binary representation and the corresponding one-point-crossover was also considered as a possible alternative. More precisely, each city is represented by a decimal number, which is transformed into a bit string (consisting of zeros and/or ones) and the strings of the cities in the tour are concatenated in the order of their occurence in the tour, hence a longer bit string is obtained which represents the tour. Thus, the one-point-crossover and the one-flip mutation operators (i.e., flip one bit randomly) for the basic representation of GA may be used. However, this approach could easily lead to illegal tours (i.e., infeasible ones) and it is difficult (i.e., takes lot of time and effort) to design special repair operators which would transform illegal tours into legal (i.e., feasible) ones. A variation (or genetic) operator (i.e., crossover or mutation) which has a high probability of generating illegal tours might not be well suited for its task. Therefore it is time to look for other better representations and genetic operators (note that the terms "genetic operator"/"variation operator"/"evolutionary operator" have the same meaning, usually standing for a crossover or a mutation, although more complicated operators are also possible):

11.4 Adjacency representation

This representation is given by the following equivalence:

City j is in position $i \Leftrightarrow$ the tour contains (i, j).

Example:

position	1	2	3	4	5	6	7	8	9
representation	2	4	8	3	9	7	1	5	6
tour	1	2	4	3	8	5	9	6	7

In this table and from now on, when we write "tour" we mean the cities listed in order of their appearance in the tour (corresponding to the "path representation" discussed later). It is easy to generate the adjacency representation from the tour. Indeed, if the tour is $\dots i\ j \dots$ or $j\ \dots\ i$ (if you think of these numbers written on a circle in clockwise order, then these two cases are the same) in the representation we have j in position i. For example, in our case we have 3 in position 4 of the representation, because the tour is of the form $\dots 4\ 3 \dots$, and 1 in position 7, because the tour is of the form $1\ \dots\ 7$.

Each tour has only one representation as a list of all cities, but some lists may belong to illegal tours (i.e., illegal tours can be represented). For example the representation given by the next table corresponds to a partial tour with a premature cycle:

position	1	2	3	4	5	6	7	8	9
representation	2	4	8	1	9	3	5	7	6
tour	1	2	4	1					

One-point-crossover is not suitable for this representation because it often leads to illegal tours that need repair procedures which are too costly. Therefore we present different crossovers for this representation in the next section.

11.4.1 Crossover for adjacency representation

Alternating edges crossover

An offspring is obtained by chosing an edge (i, j) from one of the parents p_1 (a computer makes the choice randomly, but we will start with the edge corresponding to the first position), then the next edge (j, k) from the other parent p_2, then the next edge (k, ℓ) form p_1 and so on, picking the edges alternatively from the parents. This means that in the adjacency representation of the offspring we have j, k, ℓ in the positions i, j, k, respectively. If a choice of an edge would lead to an illegal tour (in this case we have a premature cycle), then the algorithm selects randomly an edge from the available ones (i.e., the ones which would no lead to an illegal tour). Starting with p_2 instead of p_1 (i.e., swapping the roles of parents) another offspring can be constructed in a similar way.

Example:

position	1	2	3	4	5	6	7	8	9
parent 1	2	3	8	7	9	1	5	4	6
parent 2	7	5	1	6	9	2	8	4	3
offspring	2	5	8	7	9	1	6*	4	3

Let us explain this table. The algorithm first selects the edge $(1, 2)$ from p_1. Hence, 2 is in position 1 of the adjacency representation of the offspring. Next the algorithm selects the edge $(2, 5)$ from p_2. Hence, 5 is in position 2 of the adjacency representation of the offspring. Next the algorithm selects the edge $(5, 9)$ from p_1. Hence, 9 is in position 5 of the adjacency representation of the offspring. Next the algorithm selects the edge $(9, 3)$ from p_2. Hence, 3 is in position 9 of the adjacency representation of the offspring. Next the algorithm selects the edge $(3, 8)$ from p_1. Hence, 8 is in position 3 of the adjacency representation of the offspring. Next the algorithm selects the edge $(8, 4)$ from p_2. Hence, 4 is in position 8 of the adjacency representation of the offspring. Next the algorithm selects the edge $(4, 7)$ from p_1. Hence, 7 is in position 4 of the adjacency representation of the offspring. Next the algorithm would select the edge $(7, 8)$ from p_2. But a legal tour cannot contain both oriented edges $(3, 8)$ (selected earlier) and $(7, 8)$. We can also identify the premature cycle $8 - 4 - 7 - 8$ containing the edges $(8, 4)$, $(4, 7)$, $(7, 8)$ which is another reason for our tour being illegal. Hence, the algorithm selects randomly another available edge $(7, x)$ (with x a city) from the available edges (the ones which would not lead to an illegal tour, i.e., with cities x not belonging to any edge selected before). However, the only available edge is $(7, 6)$ which is selected. Since in the previous step an edge from p_2 was replaced by an available edge, the algorithm is continuing by choosing the edge $(6, 1)$ from p_1 (note that even if a random modification is made to correct the

tour the algorithm still considers the alternation of cities as if no such modification were made). We are back to our starting city 1 by visiting cities $2, 3, 4, 5, 6, 7, 8, 9$ exactly once, hence our tour is complete.

Alternating subtour-chunks crossover:

In this crossover an offspring is constructed by taking random length subtours alternatively from the two parents (i.e, first from parent p_1, then p_2, then p_1, ...).

A possible advantage of the adjacency representation is that we may look for templates corresponding to good solutions (schemata). For example, the template

* * * 3 * 7 * * *

contains all tours with edges $(4, 3)$ and $(6, 7)$.

The alternating subtour-chunks crossover is better then the alternating edges crossover, because its degree of disrupting good templates is lower.

The disadvantage of these crossovers is that they may often lead to illegal tours which need to be repaired. Moreover, the results obtained with these crossovers are poor, probably because they are blind, i.e., they don't care about the length of edges in the alternating process. Better results were obtained with the **heuristic** crossover which at each step chooses the shorter edge from the two possible edges corresponding to the two parents (we may also think about extending this crossover by considering subtours instead of edges). However, even the results obtained by using the heuristic crossover are not outstanding. Let us consider other representations and crossover possibilities for them.

11.5 Ordinal representation

In the following description of this representation consider the elements of the representation in order from the first to the last (i.e, 1st, 2nd, 3rd, ... , last):

If the ith element of the representation is j, then in the ith position of the tour is the jth city from the **remaining** cities, unvisited so far.

From this description it also follows that we must have $j \in \{1, 2, \ldots, n - i + 1\}$.

Example:

position	1	2	3	4	5	6	7	8	9
representation	1	1	2	1	4	1	3	1	1
tour	1	2	4	3	8	5	9	6	7

Let us explain this table. The 1st element of the representation is 1, so take the 1st city from the list of remaining cities, unvisited so far

$$(1, 2, 3, 4, 5, 6, 7, 8, 9),$$

that is, 1. Hence, the 1st city of the tour is 1, that is the tour is

$$(1,x,x,x,x,x,x,x,x),$$

where x denotes unknown values.

City 1 has been visited, so remove it from the list of remaining (unvisited) cities. The list of remaining (unvisited) cities becomes

$$(2, 3, 4, 5, 6, 7, 8, 9).$$

The 2nd element of the representation is 1, so take the 1st city from the list $(2, 3, 4, 5, 6, 7, 8, 9)$ of remaining cities, unvisited so far, that is, 2. Hence, the 2nd city of the tour is 2, that is the tour is

(1,2,x,x,x,x,x,x,x),

where x denotes unknown values.

City 2 has been visited, so remove it from the list of remaining (unvisited) cities. The list of remaining (unvisited) cities becomes

$$(3, 4, 5, 6, 7, 8, 9).$$

The 3rd element of the representation is 2, so take the 2nd from the list $(3, 4, 5, 6, 7, 8, 9)$ of remaining cities, unvisited so far, that is, 4. Hence, the 3rd city of the tour is 4, that is the tour is

(1,2,4,x,x,x,x,x,x),

where x denotes unknown values.

City 4 has been visited, so remove it from the list of remaining (unvisited) cities. The list of remaining (unvisited) cities becomes

$$(3, 5, 6, 7, 8, 9).$$

The 4th element of the representation is 1, so take the 1st city from the list $(3, 5, 6, 7, 8, 9)$ of remaining cities, unvisited so far, that is, 3. Hence, the 4th city of the tour is 3, that is the tour is

(1,2,4,3,x,x,x,x,x),

where x denotes unknown values.

City 3 has been visited, so remove it from the list of remaining (unvisited) cities. The list of remaining (unvisited) cities becomes

$$(5, 6, 7, 8, 9).$$

The 5th element of the representation is 4, so take the 4th city from the list $(5, 6, 7, 8, 9)$ of remaining cities, unvisited so far, that is, 8. Hence, the 5th city of the tour is 8, that is the tour is

(1,2,4,3,8,x,x,x,x),

where x denotes unknown values.

City 8 has been visited, so remove it from the list of remaining (unvisited) cities. The list of remaining (unvisited) cities becomes

$$(5, 6, 7, 9).$$

The 6th element of the representation is 1, so take the 1st city from the list $(5, 6, 7, 9)$ of remaining cities, unvisited so far, that is, 5. Hence, the 6th city of the tour is 5, that is the tour is

(1,2,4,3,8,5,x,x,x),

where x denotes unknown values.

City 5 has been visited, so remove it from the list of remaining (unvisited) cities. The list of remaining (unvisited) cities becomes

$$(6, 7, 9).$$

The 7th element of the representation is 3, so take the 3rd city from the list $(6, 7, 9)$ of remaining cities, unvisited so far, that is, 9. Hence, the 7th city of the tour is 9, that is the tour is

(1,2,4,3,8,5,9,x,x),

where x denotes unknown values.

City 9 has been visited, so remove it from the list of remaining (unvisited) cities. The list of remaining (unvisited) cities becomes

$$(6, 7).$$

The 8th element of the representation is 1, so take the 1st city from the list $(6, 7)$ of remaining cities, unvisited so far, that is, 6. Hence, the 8th city of the tour is 6, that is the tour is

$$(1,2,4,3,8,5,9,6,x),$$

where x denotes unknown values.

City 6 has been visited, so remove it from the list of remaining (unvisited) cities. The list of remaining (unvisited) cities become

$$(7).$$

The 9th element of the representation is 1, so take the 1st city from the list (7) of remaining cities, unvisited so far, that is, 7. Hence, the 9th city of the tour is 7, that is the tour is

$$(1, 2, 4, 3, 8, 5, 9, 6, 7).$$

We can reconstruct in the opposite way the representation from the tour. As before, denote by x the unknown values. From your tour (1,2,4,3,8,5,9,6,7) you can recover the representation as follows: The 1st city of your tour is 1. This is the 1st city of the remaining list

$$(1, 2, 3, 4, 5, 6, 7, 8, 9)$$

of unvisited cities. Therefore, your representation is

$$(1,x,x,x,x,x,x,x,x).$$

Remove 1 from the list of unvisited cities. The list of unvisited cities becomes

$$(2, 3, 4, 5, 6, 7, 8, 9).$$

The 2nd city of your tour is 2. This is the 1st city of the remaining list (2,3,4,5,6,7,8,9), of unvisited cities. Therefore, our representation is

$$(1,1,x,x,x,x,x,x,x).$$

Remove 2 from the list of unvisited cities. The list of unvisited cities becomes

$$(3, 4, 5, 6, 7, 8, 9).$$

The 3rd city of your tour is 4. This is the 2nd city of the remaining list (3,4,5,6,7,8,9), of unvisited cities. Therefore your representation is

$$(1,1,2,x,x,x,x,x,x).$$

Remove 4 from the list of unvisited cities. The list of unvisited cities becomes

$$(3, 5, 6, 7, 8, 9).$$

The 4th city of your tour is 3. This is the 1st city of the remaining list (3,4,5,6,7,8,9), of unvisited cities. Therefore your representation is

$$(1,1,2,1,x,x,x,x,x).$$

Remove 3 from the list of unvisited cities. The list of unvisited cities becomes

$$(5, 6, 7, 8, 9).$$

The 5th city of your tour is 8. This is the 4th city of the remaining list (5,6,7,8,9), of unvisited cities. Therefore your representation is

(1,1,2,1,4,x,x,x,x).

Remove 8 from the list of unvisited cities. The list of unvisited cities becomes

$$(5, 6, 7, 9).$$

The 6th city of your tour is 5. This is the 1st city of the remaining list (5,6,7,9), of unvisited cities. Therefore your representation is

(1,1,2,1,4,1,x,x,x).

Remove 5 from the list of unvisited cities. The list of unvisited cities becomes

$$(6, 7, 9).$$

The 7th city of your tour is 9. This is the 3rd city of the remaining list (6,7,9), of unvisited cities. Therefore your representation is

(1,1,2,1,4,1,3,x,x).

Remove 9 from the list of unvisited cities. The list of unvisited cities becomes

$$(6, 7).$$

The 8th city of your tour is 6. This is the 1st city of the remaining list (6,7), of unvisited cities. Therefore your representation is

(1,1,2,1,4,1,3,1,x).

Remove 6 from the list of unvisited cities. The list of unvisited cities becomes

$$(7).$$

The 9th city of your tour is 7. This is the 1st city of the remaining list (7), of unvisited cities. Therefore your representation is

$$(1, 1, 2, 1, 4, 1, 3, 1, 1).$$

The advantage of this representation is that classical one-point crossover generates legal tours.

11.5.1 Crossover for ordinal representation

Consider the one-point-crossover between the 4th and 5th position in the tour for the following example:

position	1	2	3	4	5	6	7	8	9
p_1 representation	1	1	2	1	4	1	3	1	1
p_2 representation	5	1	5	5	5	3	3	2	1
o_1 representation	1	1	2	1	5	3	3	2	1
o_2 representation	5	1	5	5	4	1	3	1	1
p_1 tour	1	2	4	3	8	5	9	6	7
p_2 tour	5	1	7	8	9	4	6	3	2
o_1 tour	1	2	4	3	9	7	8	6	5
o_2 tour	5	1	7	8	6	2	9	3	4

In this example we start with the parent tours $p_1 = (1, 2, 4, 3, 8, 5, 9, 6, 7)$ and $p_2 = (5, 1, 7, 8, 9, 4, 6, 3, 2)$. We represent them and make the crossover for the representations of p_1 and p_2. In this way we obtain the representation of the offsprings. For the o_1 representation we take the first 4 elements from p_1 and the remaining elements from

p_2. For the o_2 representation we take the first 4 elements from p_2 and the remaining elements from p_1. Finally, we recover the tours corresponding to the representation of the offsprings.

The partial tours to the left of the crossover point remains unchanged. However, partial tours to the right of the crossover point are disrupted in an almost random fashion. Although the representation generates legal tours, this is not enough. One would expect an inheritance from parent to offspring which is better than a blind random search. Not surprinsingly, this crossover produces poor results.

11.6 Path representation

It is the most natural representation. The representation and the tour are the same, that is, j is in the ith position of the representation if and only if j is the ith city in the tour.

Example:

position	1	2	3	4	5	6	7	8	9
representation	5	1	7	8	6	2	9	3	4
tour	5	1	7	8	6	2	9	3	4

Next we present the best known crossovers for the path representation.

11.6.1 Partially-mapped crossover (PMX)

The description of this crossover is through an example as follows:

Consider two cut points.

$$\begin{aligned} p_1 &= (123|4567|89) \\ p_2 &= (452|1876|93) \\ o_1 &= (\text{xxx}|1876|\text{xx}) \\ o_2 &= (\text{xxx}|4567|\text{xx}) \end{aligned}$$

For the o_1 offspring consider the middle part from the p_2 parent, while for the o_2 offspring consider the middle part from the p_1 parent. Here x denotes the positions which haven't been filled in yet.

Map the corresponding elements from the middle parts of the parents to each other: $1 \leftrightarrow 4, 8 \leftrightarrow 5, 7 \leftrightarrow 6, 6 \leftrightarrow 7$

For the o_1 offspring fill in the no-conflict unfilled positions x from p_1 (i.e., the ones would not lead to premature cycles), while for the o_2 offspring fill in the no-conflict unfilled positions x from p_2:

$$\begin{aligned} o_1 &= (\text{x23}|1876|\text{x9}) \\ o_2 &= (\text{xx2}|4567|93) \end{aligned}$$

Use the mappings for the remaining xs:

$$
\begin{aligned}
o_1 &= (423|1876|59) \\
o_2 &= (182|4567|93)
\end{aligned}
$$

11.6.2 Order crossover (OX)

The description of order crossover is through an example as follows:

Consider two cut points.

For offspring o_1 consider the middle part from parent p_1 and for offspring o_2 consider the middle part from parent p_2:

$$
\begin{aligned}
p_1 &= (123|4567|89) \\
p_2 &= (452|1876|93) \\
o_1 &= (xxx|4567|xx) \\
o_2 &= (xxx|1876|xx)
\end{aligned}
$$

Here x denotes the positions which haven't been filled in yet.

Starting from the second cut point the cities from the other parent (i.e., p_2 for o_1 and p_1 for o_2) are copied in the same order, omitting the ones already present:

$$
\begin{aligned}
p_2 : 934521876 &\rightarrow 93218 \\
p_1 : 891234567 &\rightarrow 92345 \\
o_1 &= (218|4567|93) \\
o_2 &= (345|1876|92)
\end{aligned}
$$

11.6.3 Cycle crossover (CX)

This crossover is described by an example

CX preserves the absolute position of elements in the parent sequence, that is, each city and its position comes from one of the parents:

$$
\begin{aligned}
p_1 &= (123456789) \\
p_2 &= (412876935) \\
o_1 &= (1xxxxxxxx) \\
o_1 &= (1xx4xxxxx) \\
o_1 &= (1xx4xxx8x) \\
o_1 &= (1234xxx8x) \\
o_1 &= (123476985)
\end{aligned}
$$

Here x denotes the unfilled positions.

Let us explain the generation of the offspring o_1 in more details. Start by taking the 1st city from parent p_1, i.e., 1:

$$o_1 = (1\text{xxxxxxxx})$$

City 4 is the the first city of p_2. However the 1st position of o_1 is already occupied by 1 from p_1. Hence, 4 and its position cannot come from p_2, it must come from p_1:

$$o_1 = (1\text{xx4xxxxx})$$

City 8 is the the 4th city of p_2. However the 4th position of o_1 is already occupied by 4 from p_1. Hence, 8 and its position cannot come from p_2, it must come from p_1:

$$o_1 = (1\text{xx4xxx8x})$$

City 3 is the the 8th city of p_2. However the 8th position of o_1 is already occupied by 8 from p_1. Hence, 3 and its position cannot come from p_2, it must come from p_1:

$$o_1 = (1\text{x34xxx8x})$$

City 2 is the the 3rd city of p_2. However the 3th position of o_1 is already occupied by 3 from p_1. Hence, 2 and its position cannot come from p_2, it must come from p_1:

$$o_1 = (1234\text{xxx8x})$$

Thus, we have completed a cycle. The remaining cities together with their positions are filled in from the other parent p_2:

$$o_1 = (123476985)$$

11.7 Edge recombination operators

Most operators we discussed so far take into consideration **cities** and not links between cities, i.e. **edges**.

But the linkage of a city with other cities is more important than its position in a tour (the distances and thus the length ot the tour is associated to the edges not the positions in the tour)!

An edge recombination operator would transfer the edges from parent to offspring most of the time

An edge list is constructed for each city in the two parents. Various ideas for edge recombination operators exist. Although based on path representation, these ideas suggests that the path representation is not enough to represent important properties of a tour. Therefore we consider next the matrix representation of a tour.

11.7.1 Matrix representation

A tour is represented by its precedence binary matrix M defined as:

$$m_{ij} = 1 \Leftrightarrow \text{city } i \text{ is before city } j \text{ in the tour.}$$

	1	2	3	4	5	6	7	8	9
1	0	1	0	1	1	1	1	1	1
2	0	0	0	1	1	1	1	1	1
3	1	1	0	1	1	1	1	1	1
4	0	0	0	0	1	1	0	0	1
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	1	0	0	0	1
7	0	0	0	1	1	1	0	0	1
8	0	0	0	1	1	1	1	0	1
9	0	0	0	0	1	0	0	0	0

The corresponding tour is $(3, 1, 2, 8, 7, 4, 6, 9, 5)$.

11.7.2 Properties of the matrix representation

Suppose we have n cities.

1. The number of ones is $\frac{n(n-1)}{2}$.
   - Indeed, the first city of the tour is before $n-1$ cities of the tour, the second city of the tour is in front of $n-2$ cities, ... , the $(n-1)$th city is in front of 1 city of the tour and summing up these numbers we obtain $(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$.
2. $m_{ii} = 0$ for all $1 \leq i \leq n$.
   - Indeed, no city is before itself in the tour.
3. If $m_{ij} = 1$ and $m_{jk} = 1$ then $m_{ik} = 1$.
   - Indeed, if city i is before city j in the tour and city j is before city k in the tour, then city i is before city k in the tour.

These properties characterise a precedence binary matrix and they will be called **conditions** of such a matrix.

New operators can be obtained from the intersection and union of ones of the precedence binary matrices of the two parents, called **intersection operator** and **union operator**, respectively. Let us describe this operators in more detail.

11.7.3 Intersection and union operators

The **intersection operator** is described as follows:

The **intersection** of bits (ones) from two parents results in a matrix which can be completed to a legal tour.

Indeed, this operation means that an entry of the resulting matrix is one if the entries of same position are ones in both parents and it is x (i.e., unknown) otherwise. In this way the number of ones in the resulting matrix is at most $\frac{n(n-1)}{2}$. Let the parent p_1 have precedence binary matrix $M^1 = (m^1_{ij})$ and the parent p_2 have precedence binary matrix $M^2 = (m^2_{ij})$ and the resulting (for the moment incomplete) matrix be $M = m_{ij}$. If $m_{ij} = 1$ and $m_{jk} = 1$, then $m^\ell_{ij} = 1$ and $m^\ell_{jk} = 1$, for all $\ell \in \{1, 2\}$. Since M^ℓ are precedence binary matrices, they satisfy condition 3, that is, $m^\ell_{ik} = 1$, for all $\ell \in \{1, 2\}$. Hence, $m_{ik} = 1$ for the presently completed one values of M. Hence for these one values M satisfies condition 3 of a precedence binary matrix. It can be shown that we can increase the number of ones in M, without putting any one in its diagonal (so that condition 2 of the precedence binary relation to be satisfied) until it reaches $\frac{n(n-1)}{2}$ (i.e., condition 1 of the precedence binary relation is satisfied) and preserve condition 3 of the precedence binary matrix.

The **union operator** is described as follows:

Union of bits (ones) can be applied to disjoint subsets from the two parents with precedence binary matrices $M^1 = (m^1_{ij})$ and $M^2 = (m^2_{ij})$, by creating a binary matrix $M = (m_{ij})$. More precisely, the operator partitions the set of cities into two disjoint sets I and J.

For the subset I it copies the ones from the submatrix of M^1 corresponding to the index set I into M (into the same positions of M as the positions in M^1) and for the subset J it copies the ones from the submatrix of M^2 corresponding to the index set J into M (into the same positions of M as the positions in M^2). In this way a (for the moment incomplete) matrix M is created.

For example if $n = 9$, $I = \{1, 2, 3, 4\}$ and $J = \{5, 6, 7, 8, 9\}$, $p_1 = (1, 2, 3, 4, 5, 6, 7, 8, 9)$, $p_2 = (4, 1, 2, 8, 7, 6, 9, 3, 5)$, then we have the situation presented in the next table:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
1	0	1	1	1	x	x	x	x	x
2	0	0	1	1	x	x	x	x	x
3	0	0	0	1	x	x	x	x	x
4	0	0	0	1	x	x	x	x	x
5	x	x	x	x	0	0	0	0	0
6	x	x	x	x	1	0	0	0	1
7	x	x	x	x	1	1	0	0	1
8	x	x	x	x	1	1	1	0	1
9	x	x	x	x	1	0	0	0	0

Suppose that $m_{ij} = 1$ and $m_{jk} = 1$, which means that either $i, j, k \in I$ or $i, j, k \in J$. In the first case $m^1_{ij} = 1$ and $m^1_{jk} = 1$ and by applying condition 3 of a precedence binary matrix for M^1 we get $m^1_{ik} = 1$ and thus $m_{ik} = 1$. In the second case $m^2_{ij} = 1$ and $m^2_{jk} = 1$ and by applying condition 3 of a precedence binary matrix for M^2 we get $m^2_{ik} = 1$ and thus $m_{ik} = 1$. Hence, for the presently completed one values of M, we have that $m_{ij} = 1$ and $m_{jk} = 1$ implies $m_{ik} = 1$, that is, for these values satisfies condition 3 of a precedence binary matrix. It can be shown that we can increase the number of ones in M, without putting any one in its diagonal (so that condition 2 of the precedence binary relation to be satisfied) until it reaches $\frac{n(n-1)}{2}$ (i.e., condition 1 of the precedence binary relation is satisfied) and preserve condition 3 of the precedence binary matrix.

By using the matrix representation and the union and intersection operator the results are better than with other edge recombination operators based on the path representation.

11.7.4 Incorporating local search methods

A huge amount of effort was spent on inventing suitable **representations** and a **recombination** operator that preserves partial tours.

GA results cannot compete with the Lin-Kernighan algorithm (neither quality nor time).

Local search can be incorporated into the genetic algorithm

Before evaluation, a local optimisation is applied to each individual.

Results are much better than GA only.

11.7.5 The inver-over algorithm

Each individual competes with its offspring only.

There is only one **adaptive** variation operator called **inversion**. Without presenting details this is some kind of hybrid operation between a crossover and a mutations.

The number of times this operator is applied to an individual during a generation is variable.

This is the quickest evolutionary algorithm for TSP known so far.

It contains only three parameters:

- the population size,
- the probability of generating random inversions,
- the number of iterations in the termination criterion.

This algorithm has good precision and stability for not too large problems.

12 Constraints in Genetic Algorithms

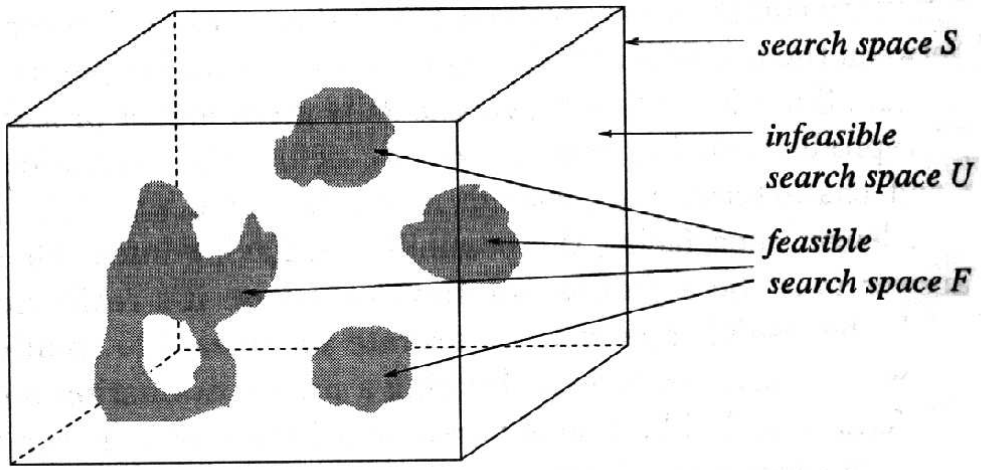
Overview

- Introduction
- Designing the evaluation function
- Rejecting infeasible individuals
- Repairing infeasible individuals
- Penalising infeasible individuals
- Using special representation, variation operators
- Using decoders

12.1 Introduction

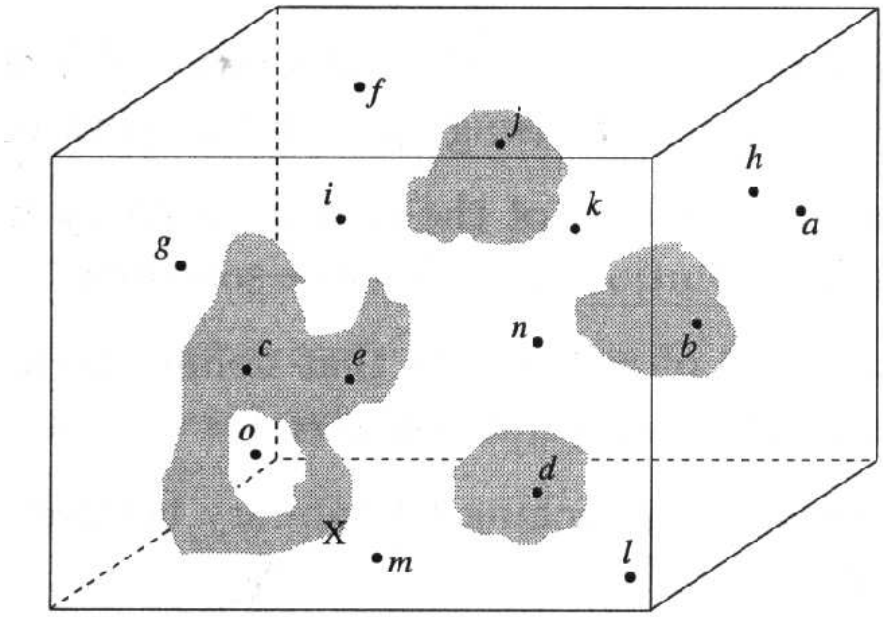
The evaluation function of a Genetic Algorithm (GA) must be carefully chosen to faithfully capture the features of the problem. In GAs better solutions have higher chance of survival. This is a difficult task if we have to deal with infeasible individuals too. However, the final solution must be feasible, otherwise it does not solve the problem. If we use GA for solving constrained optimization problems, then we have to cross the infeasible region of the search space as well, hence we must evaluate the infeasible individuals too. Indeed, it is difficult to design effective variation operators which completely avoid the infeasible region and are still effective in finding good

feasible solutions. In general a search space S can be partitioned into the disjoint union of the feasible search space F and infeasible search space U. In general we don't make any assumptions on F and S, that is, they don't have to be convex or even connected; see the particular case shown by the next figure.



Example

During our search we are processing both feasible an infeasible individuals, in order to find a feasible optimum. In particular, consider the example given by the next figure:



In this example $(a, f, g, h, i, k, l, m, n, o)$ are infeasible, while (b, c, d, e, j) are feasible, and X is the global optimum, which belongs to the feasible region.

12.2 Questions

Our seach is crossing both the F and U, therefore we should carefully design evaluation functions $eval_f$ and $eval_u$ for feasible and infeasible individuals, respectively. During this design process several questions arise:

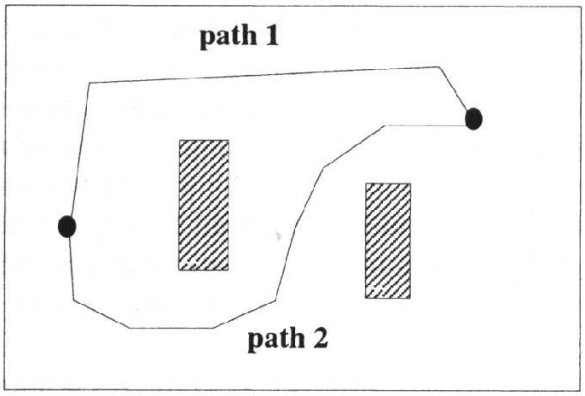
- How should we compare two feasible individuals (for example c and j in the previous figure), that is, how to design $eval_f$?
- How should we compare two infeasible individuals (for example a and n in the previous figure), that is, how to design $eval_u$?
- Should we assume that a feasible individual is always better than an infeasible one, that is, $eval_f(x)\, isbetterthan\, eval_u(y)$, where $isbetterthan$ means $>$ for maximisation and $<$ for minimisation?
- Should we eliminate infeasible individuals from the population (i.e., consider them so bad that they don't deserve any attention)?
- Should we repair infeasible individuals by replacing them with closest feasible solution (for example the global optimum X might be considered the repaired version of m)?
- If we repair an infeasible individual, then should we replace it by its repaired version, or should we use the repair process for evaluation only?
- Should we penalise infeasible individuals? If yes, then should the penalty depend on the measure of infeasibility and/or the cost that would be be needed to repair the infeasible individual?
- Should we start with a population of feasible solutions and maintain feasibility by using specialised operators?
- Should we use decoders by changing the topology of the search space and consequently translating infeasible solutions into feasible ones?
- Should we consider individuals and the constraints defining F (i.e., the feasible search space) separately?
- Should we concentrate on searching the boundary between feasible and infeasible parts of the search space, where usually good solutions reside?
- How can we find a feasible solution (this problem is called **feasibility problem** and arguably sometimes its difficulty is comparable with the difficulty of the original problem)?

12.3 Evaluating feasible solutions

For some problems the evaluation function is easily suggested by the objective. For example the evaluation function for TSP is the length of a tour. For real world problems designing a good evaluation function is often a difficult task. For example if you want to control a mobile robot by using GAs, you need to evaluate the quality of its path. Let us consider an example.

Example:

Which of the following two paths is better for a robot?:



When evaluating a path we need to consider criteria such as the total distance, clearance from obstacles and smoothness. Based on these criteria, the two paths in the previous figure are contradictory. Indeed, although path 1 is shorter and seems more clear from obstacles (it goes around one obstacle only, while path 2 goes around both obstacle), path 2 is smoother.

We have already seen that designing a good evaluation function for the Boolean Satisfiability Problem (SAT) is a tricky task. Indeed, an evaluation function which takes the value 0 if the Boolean function is false and 1 if the Boolean function is true is not too helpful, becuase it does not indicate how to improve your solutions. Therefore, we used the evaluation function "number of unsatisfied clauses" for GSAT and SA-SAT (i.e., simulated annealing for SAT), or other more sophisticated evaluation function based on the satisfied clauses. The problem with these evaluation functions is that they are discrete and prevent as from using continuous (or even better smooth) optimisation techniques to solve such problems. The following example shows how a smooth evaluation function can be designed for SAT.

Example of an evaluation function for the SAT problem:

If

$$F(\mathbf{x}) = (\overline{x}_1 \bigvee x_2 \bigvee x_3) \bigwedge (\overline{x}_1 \bigvee \overline{x}_3),$$

then let

$$eval_f(\mathbf{x}) = f(x) := x_1^2(x_2-1)^2(x_3-1)^2 + x_1^2 x_3^2.$$

Indeed, after some analysis is not too difficult to see that a boolean string (we use the standard representation for SAT) is a solution of F (i.e., F evaluates to TRUE or 1), if $f(\mathbf{x}) = 0$, that is, $\mathbf{x}$ is the global minimum of f.

Let us explain this evaluation function in more details. We put the sign + between clauses and the sign $\cdot$ (i.e., multiplication) between the following simple functions for each variable of a clause: For a variable x_i which appears in the clause without negation the function is $x_i \mapsto (x_i-1)^2$, while for a variable x_j which occurs in the clause with negation (i.e., $\overline{x}_j$ is part of the clause) the function is $x_j \mapsto x_j^2$. It is easy to see that such a function is nonnegative and the global minimum value (if it exists) of the function is attained for a solution of the corresponding SAT problem. For this observe the function is a polynomial and one of its monomials (i.e., its additive term) is zero exactly when the corresponding clause is zero. Indeed, a monomial is zero if and only if for at least one of the variables x_i of the monomial the following holds: $x_i = 1$ whenever $(x_i-1)^2$ is a multiplicative term of the monomial, while $x_i = 0$ whenever x_i^2 is a multiplicative term of the monomial. This is translated into the following statement for the corresponding clause: A clause is TRUE if and only for at least one of the variables x_i of the monomial the following holds: $x_i = 1$ whenever x_i appears in the clause without negation, while $x_i = 0$ whenever x_j occurs in the clause with negation. But the last statement obviously holds. Therefore, a monomial is zero if and only if the corresponding clause is TRUE. Therefore, the value of $f(\mathbf{x})$ is the number of unsatisfied clauses.

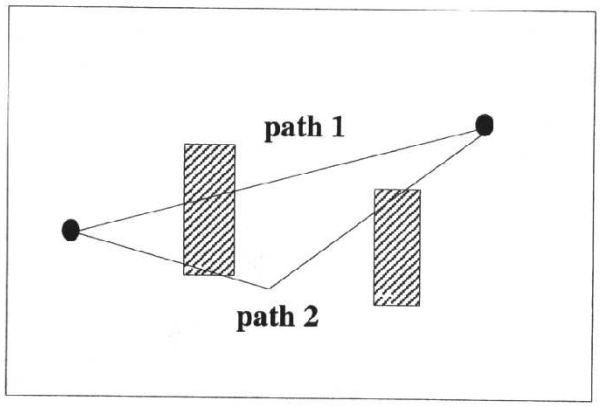
Thus, minimising the number of unsatisfied clauses is the same as minimising $f(\mathbf{x})$. Morover, we can relax the domain of variables for f, by allowing $\mathbf{x}$ to be any vector of real numbers. Minimising f on this relaxed domain of variables would still provide a solution for the corresponding SAT problem. This observation is remarkable! We transformed the highly discrete and highly combinatorial problem SAT into the smooth optimization problem $\min\{f(\mathbf{x}) : \mathbf{x} \in \mathbb{R}^n\}$!

12.4 Evaluating infeasible solutions

Evaluating infeasible is often also a difficult a task. Such a difficulty arises for evaluating infeasible paths of a robot. Let us consider an example.

Example

Which of the following two paths is better for a robot?:



The answer to this question is not clear at all, because the paths are contradictory. Although path 2 is longer than path 1, has more intersection points with the obstacles and is less smooth, path 2 is much closer to a feasible one and much easier to repair (i.e., transform into a feasible one). So should we evaluate path 2 to be the the better one? Considering as criteria only the length, the intersection points with the obstacles and the smoothness, most infeasible paths would be worse than the straight line. This would oversimplify the complexity of the problem.

A general possibility for evaluating infeasible individuals is to express $eval_u$ in terms of $eval_f$, as follows:

$$eval_u(x) = eval_f(x) \pm Q(x),$$

where x is an infeasible individual and $Q(x)$ can be a penalty (usually based on the "measure" of infeasibility of x or the cost of repairing x).

We could also use the following formula for a global evaluation function:

$$eval(x) = \begin{cases} w_1 \cdot eval_f(x) & \text{if} \quad x \in F, \\ w_2 \cdot eval_u(x) & \text{if} \quad x \in U, \end{cases}$$

where the positive weights w_1, w_2 provide the relative importance of $eval_f$ and $eval_u$.

Before continuing with the comparison of a feasible individual with an infeasible one, we present another example which shows how difficult can be to compare two infeasible solutions.

Example for comparing two infeasible solutions of the Knapsack Problem:

The **knapsack problem** can be formulated as follows:

You have a bag which has a given weight capacity and a finite number of items, each with a corresponding weight and value. You have to put items in the bag (you are not allowed to put in part of the item only, that is, you either put the item in the bag or leave it out), so as to maximise the total value of items in the bag without exceeding its weight capacity.

Suppose that the weight capacity is 99 kilograms and you have two infeasible solutions with the same total value called First and Second. The total weights of these solutions are shown below:

First: Total weight 100 kilograms

Second: Total weight 105 kilograms

Which of these solutions is better?

Solution: Without any further information we would argue that First is better than Second, because it is less infeasible. However, let us suppose that we have further information about the solutions, as follows:

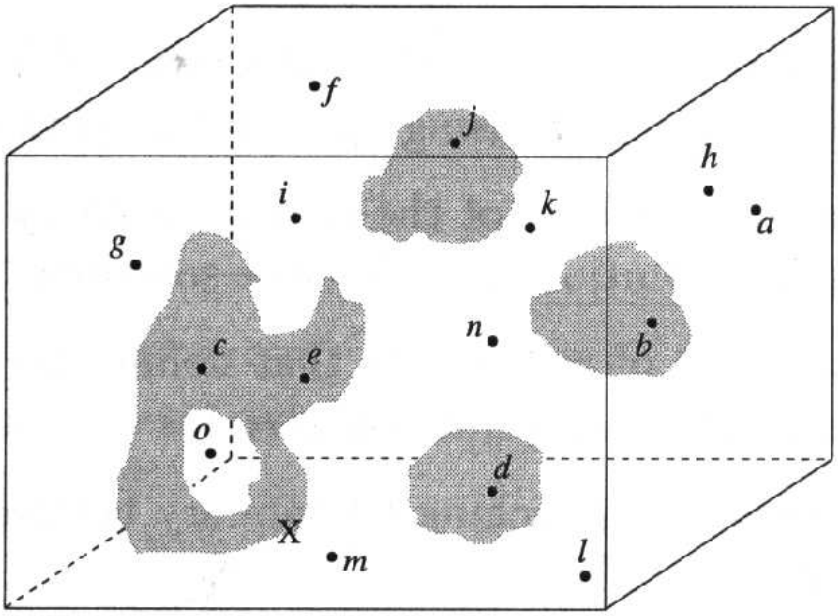
First: Contains five items of 20 kilograms each.

Second: Among other items contains an item of 6 kilograms of low profit, which is much lower in profit than the items in First.

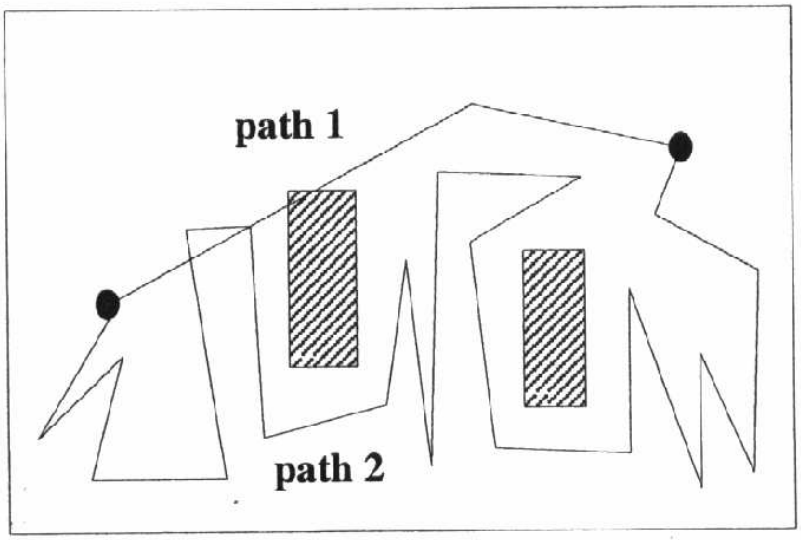
This further information completely changes the picture about the problem. Indeed, although the "measure" of infeasibility of First is lower, the only way of repairing first (into a feasible solution) is to remove an item of 20 kilograms. Hence, repairing First is more difficult, because you have to remove a heavy item of 20 kilograms, compared to repairing Second by removing a light item of 6 kilograms only. That is the "cost" (which means effort here) of repairing Second is much lower than the "cost" of repairing First. Even more importantly, the repaired version of Second is much better than the repaired version of First. Indeed, we removed an item of small value from Second, while we removed an item of much higher value from First, and originally these two solutions had the same value. Consequently, it is much more difficult to argue now that First is better than Second.

12.5 Feasible vs infeasible solutions

Is it true that a feasible individual is always better than an infeasible individual? For example should an infeasible individual close to the global optimum be always worse than a feasible one? In particular in the following figure should the infeasible solution m (which is close to the global optimum X) be worse than the feasible solution b (which is far from the global optimum X)?



To further illustrate the nontriviality of this question consider the following problem for a mobile robot:



Which path is better for the robot? Altough path 2 is feasible and path 1 is infeasible, path 1 is much shorter and smoother than path 2. Moreover, path 1 can be easily repaired (a small deviation of path 1 around the corner of the crossed obstacle is enough to make it feasible). Moreover, the repaired version of path 1 is much better than path 2.

Let us now consider the ways of dealing with infeasible individuals in detail.

12.6 Rejecting infeasible solutions

Rejecting infeasible infividuals (called **death penalty**) is popular in evolutionary search because it is simple. Death penalty rejects the infeasible individuals at each step of the search. Its main advantage is that the comparison of $eval_f$ and $eval_u$ is not needed anymore.

It may produce good results when the feasible region is convex and it is a large enough part of the search space.

Otherwise, one often encounters serious difficulties. Indeed, random sampling might give an initial population with infeasible solutions only, which would die off instantly when death penalty is applied; or a population with too few feasible solutions, which might die off after a number of generations. In such cases it is a better idea to repair the infeasible individuals rather than rejecting them outright.

Moreover variation operators might work better, if allowed to cross the infeasible region. Indeed, they might find the solution faster if they are not restricted to the feasible region only (this is especially the case if the feasible search space is nonconvex).

12.7 Repairing infeasible solutions

It is a quite popular way of dealing with infeasible solutions, especially for certain combinatorial optimisation problems, such as TSP, knapsack problem, set covering problem etc. For these problems it is relatively easy to repair infeasible individuals, that is, tranforming them into feasible ones.

Such a repaired version can be used for **evaluation only**, i.e., $eval_u(x) = eval_f(y)$, where y is the repaired feasible version of the infeasible solution x; or it can **replace** the infeasible solution (possibly with some probability only). The repair process is a combination of learning (usually means a local search for the closest feasible solution) and evolution as an interaction with the repaired version, in the sense that the fitness value of the improved version is transferred to the original (unrepaired) individual.

Replacing an individual with its repaired version assumes that an individual is improving during its lime time and the improvement is transferred into the genetic code of the individual. This is not true for natural evolution, but this flexibility of diverging from evolution in nature may be useful for improving search.

The **disadvantage** of the method of repairing infeasible individuals is its problem dependency, in the sense that different problems may require the design of different repair procedures.

Moreover, sometimes repairing an infeasible solution is just as hard as solving the original problem. This regularly happens for transportation problems, scheduling and timetabling.

12.8 Penalising infeasible individuals

It is the most common way of handling infeasibility in GAs. It is based on the following formula:

$$eval_u(x) = eval_f(x) \pm Q(x),$$

where x is an infeasible individual and $Q(x)$ is either a penalty (depending on the infeasibility "measure" of x) or the cost of repairing x.

The main question is how to design the penalty function $Q(x)$. Intuitively, one would expect that the penalty should be kept as low as possible, just above the threshold below which infeasible solutions become optimal (recall that the final solution must be feasible, therefore it is important to consider such a threshold). However this rule, called the **minimal-penalty rule**, is often difficult to implement.

We can penalise an infeasible individual based on its measure of infeasibility and/or for the cost (or effort) of repairing it (recall the knapsack example presented earlier in Section 12.4).

The penalty $Q(x)$ should depend on:

- the ratio between the size of the feasible region and the size of the search space
- the type of the evaluation function
- the number of variables

- the number of constraints
- the types of constraints
- the number of active constraints at the optimum

12.9 Different penalty factors

A natural question arises: What is the appropriate penalty function for an infeasible solution?

As an example consider the following NLP problem:

$$\begin{cases} \text{Maximize} & G_8(x) = \dfrac{\sin^3(2\pi x_1)\sin(2\pi x_2)}{x_1^3(x_1+x_2)} \\ \text{subject to} & c_1(x) = x_1^2 - x_2 + 1 \le 0 \\ & c_2(x) = 1 - x_1 + (x_2-4)^2 \le 0 \\ & 0 \le x_1 \le 10 \\ & 0 \le x_2 \le 10 \end{cases}$$

(The test function G_8 is denoted by $G8$ in the literature and the constraints c_1, c_2 by $c1$, $c2$, respectively. We slightly changed the notations to avoid any ambiguity.)

The penalised fitness (or evaluation) function

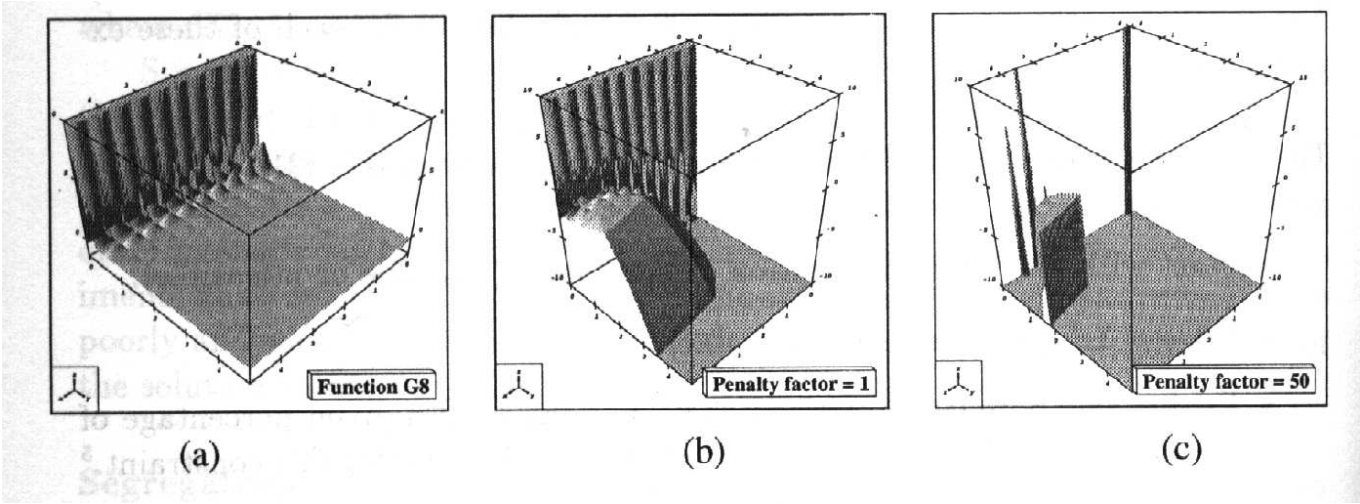
$$G_8^p(x) = G_8(x) - \alpha(c_1(x)^+ + c_2(x)^+),$$

is based on the penalty function $Q(x) = \alpha(c_1(x)^+ + c_2(x)^+)$, where $\alpha > 0$ is a penalty factor and $a^+ = \max(a, 0)$ is the **nonnegative part** of a number. This penalty function gives a measure of infeasility for x. Indeed, if x is feasible, then $Q(x) = 0$; and the larger positive value has $c_1(x)$ and/or $c_2(x)$, the larger is the penalty function $Q(x) > 0$. The following question arises: What is an appropriate value for the penalty factor $\alpha > 0$?

The function G_8 has several local optima, the highest of them reside along the x_1 axis. However, in the feasible region G_8 has two peaks (i.e., local maxima) of approximately the same fitness value, 0.1. The next figure exhibits the penalised fitness landscape, that is the values of the penalised fitness function G_8^p for different values of α.

The visible part of the penalised fitness landscape is restricted to x_3 values between -10 and 10, that is, the values of G_8^p are truncated at -10 and 10 (see (a) of the following figure). Let us distinguish the feasible region around $x_3 = 0$.

With this truncation the use of penalty function becomes a difficult task. Indeed, for small α values the really high peaks of the penalised fitness function in the feasible region remain invisible (see (b) of the following figure); while for large α values the visible penalised fitness landscape will have steep slopes (i.e., peaks which ar much higher than solutions very close to them) which are unlikely to be climbed by a GA (see (c) of the following figure).



12.10 Maintaining a feasible population

This method for handling constraints uses specialised representation and variation operators to keep within the feasible region.

The GAs for particular optimisation problems (ex. having linear constraints only) depend on unique representations and specialised variation operators. Variation operators have been developed which transform feasible individuals into other feasible ones. The advantage of this apporach is that we never need to process infeasible solutions.

This method works well for convex feasible region and linear constraints (often being more reliable then other constraint handling techniques such as for example the ones based on penalising infeasible solutions). The variation operators are often adjusted to the specific representation. Typical problems solved in this way include: Numerical Optimization, Machine Learning, Optimal Control, Cognitive Modelling, TSP, Knapsack Problems, Transportation Problems, Assignment Problems, Bin Packing, Scheduling, Engineering Design, Iterated Games, Robotics, Signal Processing etc.

12.11 Using decoders

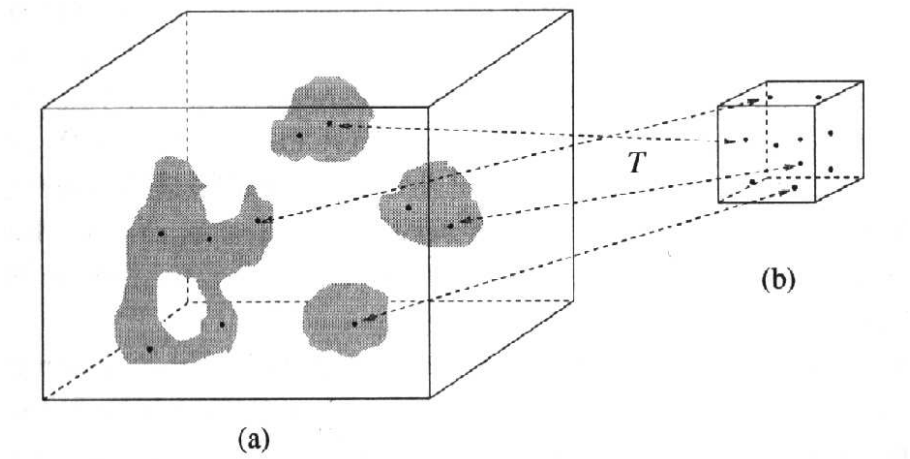
In this constraint handling method the representation (or encoding) is not for a solution, but for **building** a feasible solution.

12.11.1 Example for the Knapsack Problem

Consider the knapsack problem with items i, $i \in \{1, 2, \cdots, n\}$ (recall the description of problem from section 12.4).

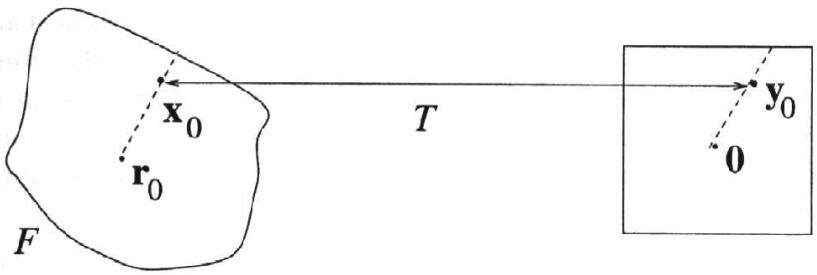
A possible solution of the knapsack problem can be encoded by a binary string of length n of zeros and ones as follows:

Put 1 in the ith position if item i is in the bag, without exceeding its capacity, and zero otherwise. Any string built in this way represents a feasible solution. This means that the representation or encoding was used for building a feasible solution. Any feasible solution can be represented by such a binary string (having 1 in the ith position for item i in the bag and zero otherwise). However, note that infeasible solutions can also be encoded by binary strings of length n. It can be seen that the basic mutation and crossover operators used to feasible solutions represented in this way generate feasible solutions (i.e., their representation which need decoding).



12.11.2 Example decoder for continuous domains

Consider a topological transformation based on the one-to-one map defined by the following figure:



Points that are close before the mapping are close after mapping and ratios of length segments in the above figure are preserved. The one-to-one mapping is between the feasible region and a hypercube H. Hence a decoder equation can be designed, as follows:

Decoder equation

$$\mathbf{y}_0 = (y_{0,1}, \ldots, y_{0,n}) \in [-1, 1]^n$$

$$\mathbf{y} = t\mathbf{y}_0, \ t \in [0, t_{\max}], \tag{4}$$

where

$$t_{\max} = \frac{1}{\max\{|y_{0,1}|, \ldots, |y_{0,n}|\}} \tag{5}$$

(4) = **line segment** from **0** to the boundary of the hypercube H. Build the feasible point x_0 corresponding to y_0 as follows:

$$\mathbf{x}_0 = \mathbf{r}_0 + \tau\mathbf{y}_0, \ \tau \in [0, \tau_{\max}], \tag{6}$$

where

$$\tau = \frac{\tau_{\max}}{t_{\max}}. \tag{7}$$

(6) = **decoder equation**: line segment from $\mathbf{r}_0$ to the boundary ∂F of F, that is, to a point $\mathbf{x}_b \in \partial F$ corresponding to $\tau_{\max}$, where $\tau_{\max}$ can be obtained numerically (e.g., using binary search).

Let us explain the above equations in more details:

$t_{\max}$ in equation (4) corresponds to a point $\mathbf{y}_b = (y_{b,1}, \ldots, y_{b,n})$ on the boundary ∂H of the hypercube H given by the equation

$$\partial H = \{\mathbf{y} = (y_1, \ldots, y_n) \in \mathbb{R}^n : \max\{|y_1|, \ldots, |y_n|\} = 1\}.$$

Thus,

$$\max\{|y_{b,1}|, \ldots, |y_{b,n}|\} = 1. \tag{8}$$

But $\mathbf{y}_b = t_{\max}\mathbf{y}_0$, i.e., $y_{b,1} = t_{\max}y_{0,1}$, ... , $y_{b,n} = t_{\max}y_{0,n}$ and therefore equation (8) implies

$$t_{\max} \cdot \max\{|y_{0,1}|, \ldots, |y_{0,n}|\} = \max\{|t_{\max}y_{0,1}|, \ldots, |t_{\max}y_{0,n}|\} = 1,$$

which implies equation (5).

Denote the Euclidean distance between two points $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$ by $d(\mathbf{u}, \mathbf{v})$ and the Euclidean length of a vector $\mathbf{z} \in \mathbb{R}^n$ by $\|\mathbf{z}\|$.

Hence, by assuming the preservation of the ratios of distances, we get

$$t_{\max} = \frac{t_{\max}\|\mathbf{y}_0\|}{\|\mathbf{y}_0\|} = \frac{\|t_{\max}\mathbf{y}_0\|}{\|\mathbf{y}_0\|} = \frac{d(\mathbf{y}_b, \mathbf{0})}{d(\mathbf{y}_0, \mathbf{0})} = \frac{d(\mathbf{x}_b, \mathbf{r}_0)}{d(\mathbf{x}_0, \mathbf{r}_0)} = \frac{\|\mathbf{x}_b - \mathbf{r}_0\|}{\|\mathbf{x}_0 - \mathbf{r}_0\|} = \frac{\|\tau_{\max}\mathbf{y}_0\|}{\|\tau y_0\|} = \frac{\tau_{\max}\|\mathbf{y}_0\|}{\tau\|y_0\|} = \frac{\tau_{\max}}{\tau},$$

which implies equation (7).

Summary

- In the majority of optimisation problems we encounter constraints.
- There are many possible methods for handling constraints.
- A careful analysis of the problem and the search space is needed before deciding which method to use.

Recommended reading

Z. Michalewicz & D.B. Fogel, How to Solve It: Modern Heuristics, Chapter 9. Constraint-handling techniques.