Advanced and Distributed Algorithms

Academic Year 2023/2024

Nicola Aggio 880008

Index

1 Introduction

1.1 Course content

The course provides basic techniques for the development and analysis of advanced algorithms. The contents of the course are the following:

- Revision of NP-completeness concepts;
- Approximation algorithms;
- Local search techniques;
- Randomized algorithms;
- Distributed algorithms;
- Examples of applications in security and robotics.

1.2 Easy and hard problems

In this section we provide a very informal definition of *easy* and *hard* problems. A problem is defined as **easy** if a computer is able to solve it in a very fast way; on the other hand, a problem is **hard** when even the fastest computers cannot solve it in a *smart way*, i.e. they have to check all the possible solutions.
In general, when we approach a new problem, it is always very important to classify it as an *easy* or *hard* one: we will see two examples of similar problems, one of which is easy, and the other one is hard.

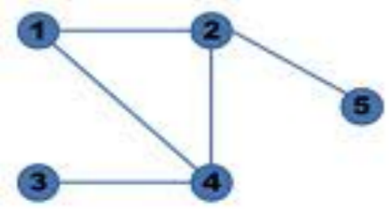
1.2.1 Easy problem

An example of easy problem is the problem of the **Könisberg** (Prussian city) **bridges**, introduced by Euler, which states the following: starting from an area, is it possible to come back to the same area by visiting all bridges exactly one time?
Before addressing more specifically this problem, we need to introduce some basic definitions.

Definition (Path). *Given a graph and two nodes x and y of such graph, a **path** from x to y is a sequence of adjacent nodes that connects node x to node y.*

Definition (Closed path). *A **closed path** is a path that starts from a node and goes back to the same node.*

Example. *In the image, if we choose $x = 1$ and $y = 5$, then {1,2,5} is one of the paths from 1*



to 5, while {1,4,2,1} is a closed path for $x = 1$.

Definition (Simple path). *A **simple path** is a path without repeated nodes.*

Definition (Circuit). *A **circuit** is a simple closed path.*

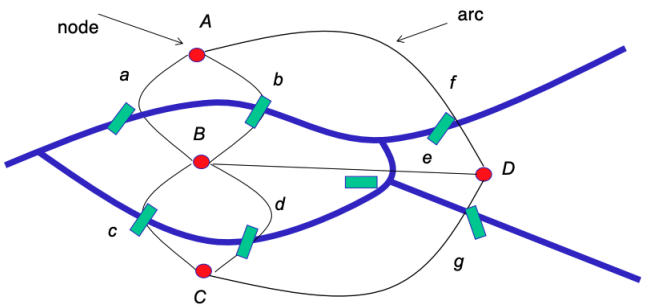
Example. *In the previous image, {1,2,5} is a simple path, {1,2,4,1,2,5} is not a simple path and {1,2,4,1} is a circuit.*

Definition (Tree). *A **tree** is an undirected simple graph G that satisfies any of the following equivalent conditions:*

- *An acyclic graph with $|E| = |V| - 1$;*
- *Connected and acyclic graph;*
- *Acyclic and connected graph, and by adding an edge it becomes cyclic.*

A rooted tree is a tree with a node called root. A degree of a vertex of an undirected graph is the number of in-degree and out-degree edges. In each tree nodes with degree ≥ 2 are called internal nodes, the ones with degree 1 are called leaves.

Coming back to the problem of the Könisberg bridges, it is clear that such problems can be formulated using graphs, where we have a node for each area, and an arc for each bridge, as showed in the image.



Intuitively, it easy easy to notice that in order to provide a solution to the problem, we require each node to have an even number of arcs (if we go away from a node, we need another arc to come back), but mathematically the previous problem can be re-formulated as the problem of searching for an Eulerian circuit in the graph.

Definition (Eulerian circuit). *A **Eulerian circuit** is a circuit that visits each **arc** of the graph **exactly once**.*

In order to solve this problem, we must rely on the following theorem.

Theorem. *A graph is Eulerian (i.e. it has a Eulerian circuit) if and only if*

1. *Is is connected (i.e. there exists a path between every pair of nodes);*
2. *All nodes have an even degree.*

Exploiting this theorem, we can easily solve the previous problem: *If a graph does not satisfy the previous properties, then it is not Eulerian, otherwise we need an algorithm for finding it.*
The algorithm that finds an Eulerian circuit in a graph is outlined as follows.
Note that the **complexity** of the algorithm is $O(m)$ steps, where $m = |E|$ is the number of arcs.

Algorithm 1: Find the Eulerian circuit of a graph

Input: Vertex v_1
Output: ..
if v_1 *does not have outgoing arcs* **then**
    return v_1;
else
    Create a closed path $C = [v_1, v_2, .., v_k, v_1]$ from v_1 visiting all the arcs only once;
    Erase all the arcs of path C;
    return (*Eulero*(v_1), .., *Eulero*(v_k), v_1);

1.2.2 Hard problem

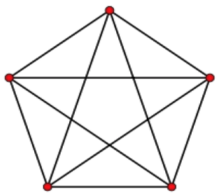
An example of *hard problem* is the one of finding an Hamiltonian circuit in the graph.

Definition (Hamiltonian circuit). *An **Hamiltonian circuit** is a circuit that visits all the **nodes** of the graph **exactly once**.*

This is a *hard* problem, since there exists no efficient algorithm that solves it in general (for any possible graph). The only solution is to find all paths.
As we can see, the two problems we've seen are very **similar** (in one we have to visit all the arcs exactly once, in the other all the nodes), but one is *easy* to solve, the other is *hard*.
Another hard problem similar to the search for the Hamiltonian circuit is the **travelling salesman problem**: given a weighted (with weight on arcs) complete (with all possible arcs) graph, find a Hamiltonian circuit of minimum weight. The salesman has to start from a city, visit all other cities, go back to the starting point travelling the minimal number of kilometers.
In general, computer science studies real problems, builds mathematical models and searches for efficient algorithms that solve the problems. It is useful to CLASSIFY the problems, i.e., to evaluate if a problem is "easy" (thus solvable by an efficient algorithm) or hard. In the second case we will search for: approximate solutions, or solutions which are correct with high probability, etc..

1.3 Exercises

1. Execute the algorithm for finding Eulerian circuit in the following graph.



2 Basics of algorithm analysis

Before providing the formal definitions, let us introduce an intuitive overview of what easy, hard/intractable and undecidable problems are:

- **Easy** problem: it is possible to solve it using an efficient algorithm (with at most polynomial time complexity);
- **Hard** or **intractable** problem: to find a solution we have to explore all the possible ones;
- **Undecidable** problem: it does not have algorithmic solution.

In general, **analyzing algorithms** involves thinking about how their **resource requirements** (the amount of time and space they use) will **scale** with **increasing input size**.

2.1 Worst-case running time and brute-force search

To begin with, we will focus on analyzing the **worst-case running time**: we will look for a **bound** on the **largest** possible **running time** the algorithm could have over all inputs of a given size N, and see how this scales with N.
While in general the **worst-case analysis** of an algorithm has been found to do a reasonable job of capturing its efficiency in practice, **average-case analysis** (the obvious appealing alternative, in which one studies the performance of an algorithm averaged over "random" instances) can sometimes provide considerable **insight**, but very often it can also become a **quagmire**. As we observed earlier, it's very **hard** to express the **full range of input instances** that arise in practice, and so attempts to study an algorithm's performance on "random" input instances can quickly devolve into debates over how a random input should be generated: the same algorithm can perform very well on one class of random inputs and very poorly on another. After all, real inputs to an algorithm are generally not being produced from a random distribution, and so average-case analysis risks telling us more about the means by which the random inputs were generated than about the algorithm itself.
But what is a reasonable analytical benchmark that can tell us whether a running-time bound is impressive or weak? A first simple guide is by comparison with brute-force search over the search space of possible solutions.

Definition (Efficient algorithm (1)). *An algorithm is* ***efficient*** *if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.*

2.2 Polynomial time as a definition of efficiency

When people first began analyzing discrete algorithms mathematically (a thread of research that began gathering momentum through the 1960s) a consensus began to emerge on how to quantify the notion of a **"reasonable" running time**. Search spaces for natural combinatorial problems tend to grow exponentially in the size N of the input; if the input size increases by one, the number of possibilities increases multiplicatively. We'd like a good algorithm for such a problem to have a **better scaling property**: when the input size increases by a constant factor (say, a factor of 2) the algorithm should only slow down by some constant factor C.
Arithmetically, we can formulate this scaling behavior as follows.

Definition (Polynomial-time algorithm). *Suppose an algorithm has the following property: there are absolute constants $c > 0$ and $d > 0$ so that on every input instance of size N, its running time is bounded by cN^d steps (in other words, its running time is at most proportional*

*to N^d). If this running-time bound holds, for some c and d, then we say that the algorithm has a **polynomial running time**, or that it is a **polynomial-time algorithm**.*

Note that any polynomial-time bound has the scaling property we're looking for. If the input size increases from N to $2N$, the bound on the running time increases from cN^d to $c(2N)^d = c2^dN^d$, which is a slow-down by a factor of $2d$. Since d is a constant, so is 2^d; of course, as one might expect, lower-degree polynomials exhibit better scaling behavior than higher-degree polynomials. From this notion, we can derive another definition of efficiency.

Definition (Efficient algorithm (2)). *An algorithm is **efficient** if it has a **polynomial running time**.*

The justification for this definition relies on the fact that it really works: problems for which polynomial-time algorithms exist almost invariably turn out to have algorithms with running times proportional to very moderately growing polynomials like n, $n \log n$, n^2, or n^3. Conversely, problems for which no polynomial-time algorithm is known tend to be very difficult in practice. There are certainly **exceptions** to this principle in both directions: there are cases, for example, in which an algorithm with **exponential worst-case** behavior generally runs **well** on the kinds of instances that arise in practice; and there are also cases where the best **polynomial-time** algorithm for a problem is completely **impractical** due to large constants or a high exponent on the polynomial bound.
One further reason why the mathematical formalism and the empirical evidence seem to line up well in the case of polynomial-time solvability is that the gulf between the growth rates of polynomial and exponential functions is enormous, as shown in the image below.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Figure 1: *The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.*

2.3 Analysis and design of algorithms

In order to evaluate an algorithm A for a problem P, we need to consider the following aspects:

- **Correctness:** *Does A solve the problem P? Does A terminate?*;
- **Complexity** (i.e. analysis of the performance of A): compute the complexity of A in the worst/average case;
- Find a **lower bound** to the problem: try to compute the number of operations that are "necessary" to solve the problem, no matter which solution is provided;
- **Compare** the complexity of A with the **lower bound**: is A a "good" algorithm?

Example. *Given a sequence of numbers* $32, 1, 25, 9, 2, 23, 34, 0, 77$*, find the maximum value. A possible algorithm could be the following: initialize the maximum value with the first element of the sequence, and then update it as we scan the sequence. In this case:*

- *Correctness? Does the algorithm solve the problem? Yes, simple argument;*
- *Complexity? How many operations are required?* $n-1$ *comparisons in worst/average case (no matter which is the input), so the complexity is* $T(n) = n-1$ *(linear time complexity);*
- *Lower bound:* $n-1$ *comparisons (intuitive idea:* $n-1$ *is with* $< n-1$ *comparisons, as the maximum could be excluded. Think it as a tournament, at each match the biggest wins, each element has to loose but the biggest, so at least* $n-1$ *"loosers", i.e., comparisons. We could also think about* $n-1$ *internal nodes of a binary tree with* n *leaves);*
- *Optimality: the complexity of the algorithm is* $n-1$ *and it mathces the lower bound, thus the algorithm is optimal.*

2.4 Complexity of problems

We can now provide more formal definitions of easy, hard/intractable and undecidable problems:

- **Easy problems:** it is possible to solve them using an **efficient algorithm** (with at most a polynomial complexity, $O(n^k)$ where n is the size of the input and k is a constant);
- **Hard or intractable problems:** to find the solution we have to explore **all the possible solutions** ($O(k^n)$, where n is the size of the input and k is a constant, no polynomial solution is known);
- **Undecidable problem**: they do **not** have any **algorithmic solution**.

2.4.1 The first undecidable problem

The first example of undecidable problem was the **Halting problem** (Alan Turing, 1936). In this case:

- Input: An arbitrary algorithm A and its input data D;
- Output: Decide in finite time if computing A on D halts (finishes execution) or not.

We could try to solve this problem by simply applying A to D and wait the result.., and what if $A(D)$ does not terminate? Turing proved that the existence of such an algorithm A would produce a paradox.

2.4.2 Hard or intractable problems

Some examples of hard/intractable problems (no polynomial solution is presently known) are the problem of finding an **Hamiltonian circuit** in a graph, i.e. finding a closed path that visits all the nodes of the graph exactly once, or the **travelling salesman problem** (TSP), where we want to find an Hamiltonian circuit of minimum weight cost.
Usually it is not easy to find the solution, we have to search for ALL the possible paths, so we have to label the vertices as $1, 2, .., n$. What is the number of possible paths? If the labels of the vertices are $1, 2, .., n$ we check all the $n!$ permutations of all vertices to see if there is an Hamiltonian path ($n!$ is not a polynomial number in n). If we are luck we find it right away, otherwise at the end (notice that we could only check from one node, thus having $(n-1)!$): note that if the graph is bipartite with n odd no solution exists.

This problem is **hard**, and problems like these are called **NP-complete**. We now define the related **decisional problem**, e.g., *"Do we have a Hamiltonian cycle or not, no matter which the solution (sequence of nodes) is?"*. The answer of such problems is YES or NO.

2.5 NP and computation intractability

We first define what a decision problem is.

Definition (Decision problem). *A decision problem is a* ***problem*** $P : I \to S$ *where:*

- I *is a set of instances;*
- S *are the solutions, i.e.* $S = \{YES, NO\}$.

Example. *Given a graph G, does an Hamiltonian cycle exist?*

Example. *Given a graph G, a source s and a destination d, does a path between s and d exist, having length at most k?*

To define NP-complete problems we will refer to decision problems. Clearly, there's a difference between **optimization** and **decision problems**:

- Decision problem: Given a graph G, a source s and a destination d, does a path between s and d exist, having length at most k? Given a complete undirected weighted graph G, does it have an Hamiltonian cycle of cost at most k?;
- Optimization problem: Given a graph G, a source s and a destination d, find a shortest path (a path of minimal length, i.e., with a minimal number of nodes) from s to d. Given a complete undirected weighted graph G find an Hamiltonian cycle of minimal cost.

Note: if we show that the decisional problem is NP-complete, then also the optimization problem in NP-complete.

2.6 P problems

Definition (P problems). *P problems are* ***decision problems*** *for which there is a* ***polytime algorithm****. The algorithm has* ***complexity*** $O(n^k)$ *where n is the input and k a constant. Usually, k is* ***small****.*

Some examples of P problems are showed in the image below.

Problem	Description	Algorithm	Yes	No
MULTIPLE	Is x a multiple of y?	Grade school division	51, 17	51, 16
RELPRIME	Are x and y relatively prime?	Euclid (300 BCE)	34, 39	34, 51
PRIMES	Is x prime?	AKS (2002)	53	51
EDIT-DISTANCE	Is the edit distance between x and y less than 5?	Dynamic programming	niether neither	acgggt ttttta

2.7 NP problems

Definition (NP problems (1)). *NP problems are **decision problems** which can be **verified** in **poly-time**.*

Intuitively, NP problems are the ones for which, given their solution, we can check the correctness in polynomial time.

Example. *Given an undirected weighted graph G of 4 nodes, verify if a given sequence of 4 nodes is a Hamiltonian cycle of length at most 20. We have to do 4 (in general n) additions. The cost of verifying the solution is polynomial in n (while finding the optimal solution is not easy ..).*

We can define NP problems using a **certification** intuition. A certifier views things from "managerial" viewpoint, and does not determine a solution of the problem P, rather, it checks a proposed solution for P.

Definition (NP problems (2)). *Given a problem P, an instance x of P is true if and only if there exists a **certificate** for x of length limited by a polynomial $p(.)$ in x, such that given to P will say YES in **polynomial time**.*

Definition (NP problems (3)). ***Decision problems** for which there exists a **poly-time certifier**.*

Example. *We consider the COMPOSITES problem: given an integer s, is s composite (i.e. not prime)? In this case a **certificate** is a nontrivial factor t of s, and such a certificate exists if and only if s is composite. Moreover $|t| \leq |s|$. The **certifier** is showed in the following image. For these reasons, COMPOSITES is in NP.*

```
boolean C(s,t) {
   if (t ≤ 1 or t ≥ s)
      return false
   else if (s is a
multiple of t)
      return true
   else
      return false
}
```

Example. *We consider the HAM-CYCLE problem: given an undirected graph $G = (V, E)$, does a simple cycle C that visits every node exist? In this case a **certificate** is a permutation (a list) of the n nodes, while the **certifier** checks if the permutation contains each node in V exactly once, and if there is an edge between each pair of adjacent nodes in the permutation (e.g., look at the adjacency list of nodes). Thus, HAM-CYCLE is in NP.*
Notice that if I send a sequence of nodes which is not a solution, the certifier only returns NO, but this does not mean that another solution does not exist.

Example. *We consider the SAT problem: given a CNF (Conjunctive Normal Form) formula F, is there a satisfying assignment (i.e., an assignment that makes the formula true)?*
We use $\wedge$ and $\vee$ or $\neg$ not, the x_i are Boolean variables which are combined into clauses in CNF. A literal is either a variable (in which case it is called a positive literal) or the negation of a

variable (called a negative literal). In this case the ***certificate*** *is an assignment of truth values to the n boolean variables, and the* ***certifier*** *checks that each clause in F has at least one true literal.*

instance s $\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$

certificate t $x_1 = true,\ x_2 = true,\ x_3 = false,\ x_4 = false$

Theorem. $P \subseteq NP$

Proof. Consider any problem p in P: by definition, there exists a poly-time algorithm A that solves p. In we consider a certificate $t = e$ (empty), then the certifier is $C(p, t) = A(p)$. (i.e., the certifier is the algorithm itself). ■

2.8 EXP problems

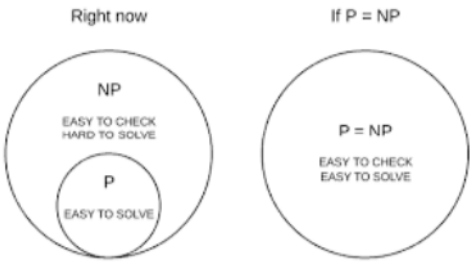
Definition (EXP problems). *Decision problems for which there is an* ***exponential algorithm*** *that runs in $O(2^{p(n)})$ time, where $p(n)$ is a polynomial function of n.*

Intuitively, problems in EXP can be solved in exponential time, but cannot be checked in polynomial time.

Theorem. $NP \subseteq EXP$

2.9 P vs NP

A big question is clearly whether $P = NP$ (Millennium Prize problem), i.e. to establish if the decision problems are as easy as the certification problems. If yes, then we have efficient algorithms for 3-COLOR, TSP, FACTOR, SAT; if no, there are no efficient algorithms possible for 3-COLOR, TSP etc..



The consensus opinion is that $P \neq NP$.

2.10 NP-Complete problems

In the absence of progress on the $P = NP$ question, people have turned to a related but more approachable question: What are the hardest problems in NP? Before analyzing NP-complete problems, let's introduce some crucial concepts.

Definition (Polynomial transformation). *We say that a decision problem P_1* ***polynomial transforms*** *to a decision problem P_2 if given any input x to P_1, we can construct an input y, such that x is a yes instance of P_1 if and only if y is a yes instance of P_2. We denote this concept as $P_1 \leq_p P2$*

Intuitively a problem P_1 can be reduced to a problem P_2 if any instance of P_1 can be "easily rephrased" as an instance of P_2.

Example.

- P_1 = *solution of linear equations* $ax + b = 0$*;*
- P_2 = *solution of quadratic equations* $a'x^2 + b'x + c' = 0$*;*

P_1 reduces to $P2$: given an instance $ax + b = 0$ of P_1, we transform it into an instance of P_2 as $0x^2 + ax + b = 0$. A solution of $0x^2 + ax + b = 0$ provides a solution to $ax + b = 0$ and vice versa.

Thus, P_1 can be reduced to P, i.e., intuitively P_1 is "not harder to solve" than P_2.

Example.

- P_1 = *Does a Hamiltonian cycle exist in a graph?;*
- P_2 = *Does a Hamiltonian cycle of cost at most k exist in a graph?*

P_1 reduces to P_2: we have to write an instance of P_1 into an instance of P_2. We take an undirected weighted G of k nodes, give cost 1 to the existing arcs, transform it into a complete graph G' and give cost 2 to the new arcs.

- *Now, we look for a solution of P_2 of G' of cost k, if such a solution exists, it contains only arcs in G, thus G is Hamiltonian (if it does not exist, we have taken at least a new arc of cost 2);*
- *Now we look for a solution of P_1, a Hamiltonian cycle in G, if it exists it has cost k and is also a solution to P_2*

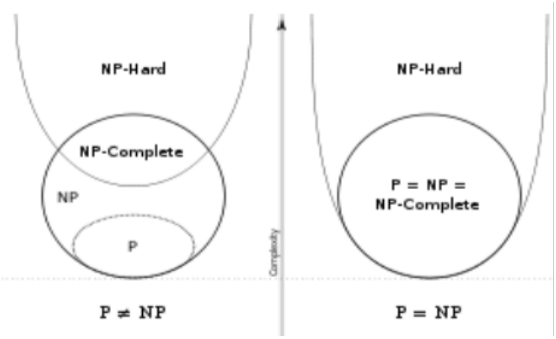
Definition (NP-complete problems). *A problem P_2 is* ***NP-complete****, i.e. it is in NPC if:*

- *P_2 is in NP;*
- *Every problem P_1 in NP, $P_1 \leq_p P_2$, i.e., all the known problems in NP reduce to P_2.*

If only the second relation holds, we say that the problem is ***NP-hard****.*

Theorem. *Suppose P_1 is an NP-complete problem. Then P_1 is solvable in polynomial time if and only if $P = NP$.*

Proof.



- If $P = NP$, then P_1 can be solved in poly-time since P_1 is in NP and thus in P;
- Suppose P_1 can be solved in poly-time. Let X be any problem in NP. Since $X \leq_p P_1$, we can solve X in poly-time. This implies $NP \subseteq P$. We already know $P \subseteq NP$. Thus $P = NP$. ■

Example. *We consider the CIRCUIT-SAT problem: given a combinational circuit built out of AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1? Intuitively, this problem is in NP, because if we have a solution, i.e. a set of inputs for which the output is 1, then we can verify it in polynomial time.*

Theorem. *CIRCUIT-SAT is NP-complete.*

Note:

- We use this as a base to prove that all the other problems are NP-complete;
- We do not need to prove that all NP-complete problems reduce to a new problem P_1, but we can pick just one and show the reduction.

Theorem. *3-SAT is NP-complete.*

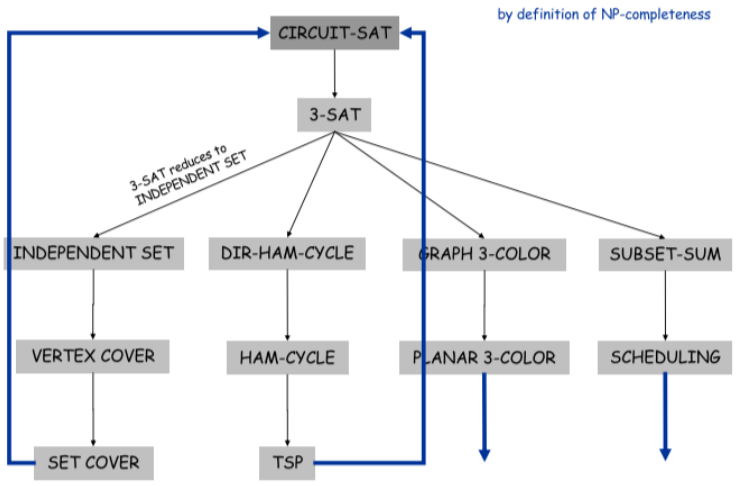
Proof. It is sufficient to show that $CIRCUIT - SAT \leq_p 3 - SAT$, since 3-SAT is in NP (omitted). ■
We recall that the 3-SAT problem is the SAT problem where each clause is limited to at most three literals.

2.10.1 Establishing NP-completeness

Once we establish first "natural" NP-complete problem, others fall like dominoes: the recipe to establish NP-completeness of problem P_1 is the following:

1. Show that P_1 is in NP;
2. Choose an NP-complete problem X;
3. Prove that $X \leq_p P_1$: if X is an NP-complete problem, and P_1 is a problem in NP with the property that $X \leq_p P_1$ then P_1 is NP-complete.

All problems below are NP-complete and polynomial reduce to one another!



2.10.2 NP-completeness proofs

We now consider the proof of a very famous NP-complete problem.

Definition (Clique). *A clique in an undirected graph $G = (V, E)$, is a subset $V' \subseteq V$ of vertices such that each vertex $u, v \subseteq V'$ is connected by an edge $(u, v) \in E$. In other words, a clique is a* ***complete subgraph*** *of G*

We can have two possible problems:

- The optimization **CLIQUE** problem: find a clique of maximum size in a graph;
- The Decision Clique Problem (**DCLIQUE**): given an undirected graph G and an integer k, determine whether G has a clique with k vertices.

Theorem. *The DCLIQUE problem in NP-complete.*

Proof. We need to show two things:

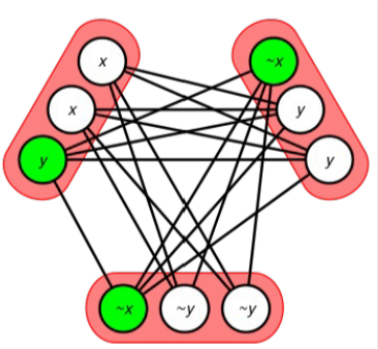
1. That $DCLIQUE \in NP$;
2. That there is some $P_1 \in NP$-complete such that $P_1 \leq_p DCLIQUE$.

1) We prove that $DCLIQUE \in NP$. A certificate will be a set of vertices $V' \subseteq V$, $|V'| = k$, that is a possible clique. To check that V' is a clique all that is needed is to check that all edges (u, v) with $u \neq v$, $u, v \in V'$ are in E. This can be done in time $O(|V|^2)$ if the edges are kept in an adjacency matrix;
2) We prove that there is some $P_1 \in NP$-complete such that $P_1 \leq_p DCLIQUE$. We choose $P_1 = 3{-}SAT$ which is NP-complete. Given a CNF formula consisting of k clauses $C_1 \wedge C_2 \wedge .. \wedge C_k$, where each C_i is composed of 3 elementary literals l_1^i, l_2^i, l_3^i, the corresponding graph consists of a vertex for each literal, and an edge between each two non-contradicting literals from different clauses. The graph has a k-clique if and only if the formula is satisfiable. So:

- If I have a true assignment, each clause has to be true, at least one element is set to true. Since two groups of vertices are connected if the literals are non-contradicting, I have a clique;

- If V has a clique V' of size k, and since vertices of the same group are not connected, I cannot pick two of them to form the clique but they have to belong to different groups. I then set true to the corresponding literal, and there are no contradicting literals in distinct groups, thus I satisfy the formula. ■

Example. *The 3-SAT instance* $(x \vee x \vee y) \wedge (\neg x \vee \neg y \vee \neg y) \wedge (\neg x \vee y \vee y)$ *reduced to a clique problem. The green vertices form a 3-clique and correspond to the satisfying assignment* x*=FALSE,* y*=TRUE.*



2.11 Exercises

1. Define the decisional version of the vertex cover problem.

   Describe the simple 2-approximation algorithm seen in class for the solution of this problem. First describe the algorithm (in code, or pseudocode or words) in a complete and precise way, then show how the algorithm works on a small example with $n = 5$ nodes.

   Solution on slide 30-31 of L10.

2. Give the formal definition of polynomial time transformation and describe a simple example. Solution on slide 20 of L15.

3 Approximation algorithms

Following our encounter with NP-completeness and the idea of computational intractability in general, we've been dealing with a fundamental question: How should we design algorithms for problems where polynomial time is probably an unattainable goal? In this chapter, we focus on a new theme related to this question: approximation algorithms, which run in polynomial time and find solutions that are guaranteed to be close to optimal.
There are two key words to notice in this definition: close and guaranteed. We will not be seeking the optimal solution, and as a result, it becomes feasible to aim for a polynomial running time. At the same time, we will be interested in proving that our algorithms find solutions that are guaranteed to be close to the optimum (more specifically, a ρ-approximation of the optimal solution). There is something inherently tricky in trying to do this: in order to prove an approximation guarantee, we need to compare our solution with (and hence reason about) an optimal solution that is computationally very hard to find. This difficulty will be a recurring issue in the analysis of the algorithms in this chapter.

Definition (ρ-approximated algorithm). *Assume we search for the* ***minimum*** *or* ***maximum*** *of a cost function. Assume C^* is the* ***cost*** *of an* ***optimal solution*** *and C is the* ***cost*** *of the* ***approximation algorithm****, we have a ρ-approximated algorithm if and only if $max(C/C^*, C^*/C) \leq \rho$ (depending whether we're considering a min/max problem). An approximation algorithm has as input a problem instance and a value r. We are looking for polynomial time approximation algorithms.*

Consider two problems, P_1 and P_2, both of which are NP-complete (decisional version). If I find an approximation algorithm for the optimization version of one, can I apply it to the other? In general, a good approximation algorithm for one problem does not provide a good approximation to the other unless the reduction is approximation-preserving.

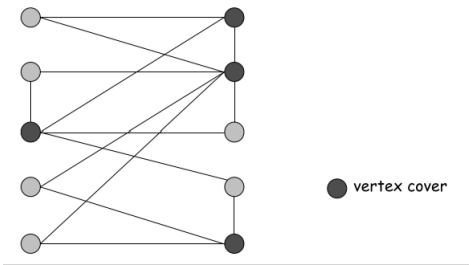
Definition (Approximation-preserving reduction). *An approximation-preserving reduction is an* ***algorithm*** *for transforming one optimization problem into another problem, such that the* ***distance*** *of solutions from optimal is* ***preserved*** *to some degree.*

3.1 Vertex cover

Definition (Vertex cover problem). *Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $V' \subseteq V$ such that $|V'| \leq k$, and for each edge, at least one of its endpoints is in V'?*

This basically reduces to the problem of **finding** a **subset of vertices** that **covers all the edges** of the graph.

Example. *Is there a vertex cover of size ≤ 4? Yes. Is there a vertex cover of size ≤ 3? No.*



Obviously, we can have:

- *Optimization version*: In this case the input is a graph G, and the output is the vertex cover V' of minimum-size;
- *Decision version*: In this case the input is a graph G, and an integer k, and we want to answer the following question: does G have vertex cover of size $\leq k$?

The decision version of the problem is NP-complete ($3SAT \leq_p VC$), and the optimization version is at least as hard.
Here is a trivial **2-approximation algorithm**, that is, the solution contains at most twice the number of vertices of the optimal solution.

Algorithm 2: Approximated algorithm for finding the vertex cover of a graph

Input: Graph $G = (V, E)$
Output: ..

```
1 V' ← ∅;
2 E' ← E;
3 while E' ≠ ∅ do
4 |   Let (u, v) be an arbitrary edge of E';
5 |   Remove from E' all the edges incident on either u or v;
6 |   V' = V' ∪ {u, v}
7 return V';
```

We now discuss the proposed algorithm.

3.1.1 Correctness

Theorem. *The algorithm is a poly-time 2-approximation algorithm.*

Proof. The running time is trivially bounded by $O(V * E)$ (at most $|E|$ iterations, each of complexity at most $O(V)$. Notice that if $E = n^2$, then we have a complexity of $O(V * E) = O(n^3)$).
Correctness: V' clearly is a vertex cover and the algorithm terminates since at each step at least one edge is removed.
Let's now consider the size of the resulting cover: let A denote set of edges that are picked.

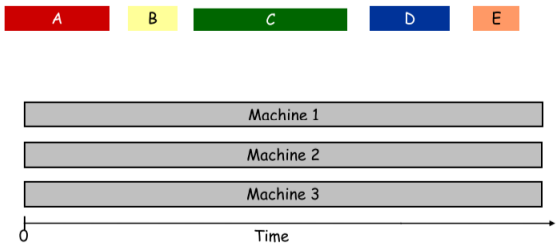
1. In order to cover edges in A, any vertex cover, in particular an optimal cover VC^*, must include at least one endpoint of each edge in A. By construction of the algorithm, no two edges in A share an endpoint (once edge is picked, all edges incident on either endpoint are removed). Therefore, no two edges in A are covered by the same vertex in VC^*, and $|VC^*| \geq |A|$;
2. When an edge is picked, neither endpoint is already in VC, thus $|VC| = 2|A|$.

Combining (1) and (2) yields $|VC| = 2|A| \leq 2|VC^*|$, from which we get $\frac{VC}{VC^*} \leq 2$, so the algorithm is 2-approximated. ■

3.2 Load balancing

Now we consider the Load Balancing problem. We have in input m identical machines and n jobs:

- Job j has processing time t_j;
- Job j must run contiguously on one machine;
- A machine can process at most one job at a time.



Definition (Load). *Let $J(i)$ be the subset of jobs assigned to machine i. The **load** of machine i is $L_i = \sum_{j \in J(i)} t_j$.*

Definition (Makespan). *The **makespan** is the **maximum load** on any machine $L = \max_i L_i$.*

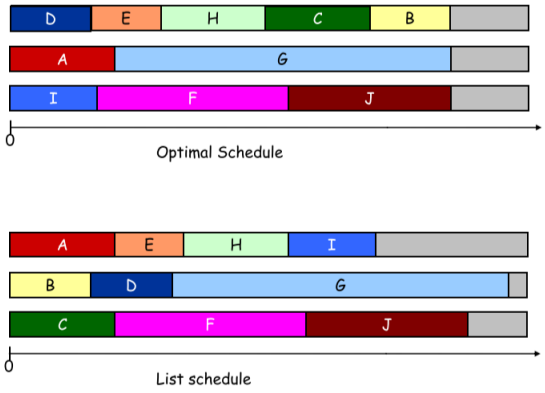
The goal of the Load Balancing problem is to assign each job to a machine to minimize makespan. This problem in **NP-hard**.

3.2.1 List scheduling algorithm

In this case the approach is the following:

1. Consider n jobs in some fixed order;
2. Assign job j to machine whose load is smallest so far.

An example of execution of the algorithm is showed in the following image.



Formally, the algorithm is the following:

```
List-Scheduling(m, n, t1,t2,...,tn) {
   for i = 1 to m {
      Li ← 0        ← load on machine i
      J(i) ← φ      ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = minimum mink Lk ← machine i has smallest load
      J(i) ← J(i) ∪ {j}   ← assign job j to machine i
      Li ← Li + tj        ← update load of machine i
   }
   return J(1), …, J(m)
}
```

The **complexity** is $O(n \log m)$ if we exploit a priority queue.

Lemma 1. *The **optimal makespan** is* $L^* \geq \max_j t_j$

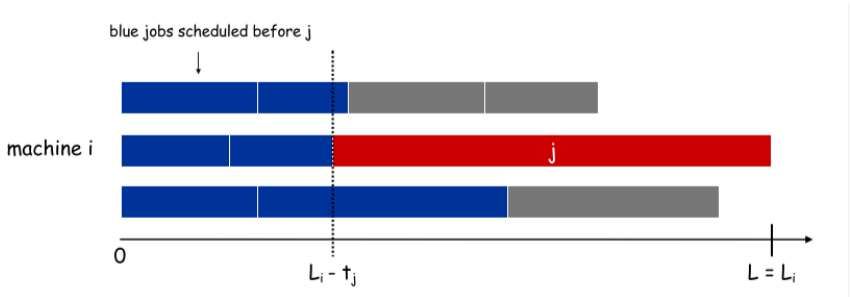
Proof. Some machine must process the most time-consuming job.

Lemma 2. *The **optimal makespan** is* $L^* \geq \frac{1}{m} \sum_j t_j$

Proof. The total processing time is $\sum_j t_j$, and one of the m machines must do at least a $1/m$ fraction of total work.

Theorem. *The greedy algorithm is a 2-approximation (i.e. the makespan is at most twice as the optimal one).*

Proof. Consider load L_i of bottleneck machine i, and let j be last job scheduled on machine i. When job j is assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \implies L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.



We sum all the inequalities over all k and we divide by m:

$$\sum_k L_k \geq m(L_i - t_j)$$

from which we get that

$$L_i - t_j \leq \frac{1}{m} \sum_k L_k$$

, but the total load is equal to the total time ($\sum_k L_k = \sum_k t_k$), so

$$L_i - t_j \leq \frac{1}{m} \sum_k t_k$$

From Lemma 2 we know that $L_i - t_j \leq L^*$, so

$$L_i = (L_i - t_j) + t_j$$

, but:

- $(L_i - t_j) \leq L^*$;
- $t_j \leq L^*$ from Lemma 1, since $L^* \geq \max_j t_j$,

so we conclude that

$$L_i \leq 2L^*$$

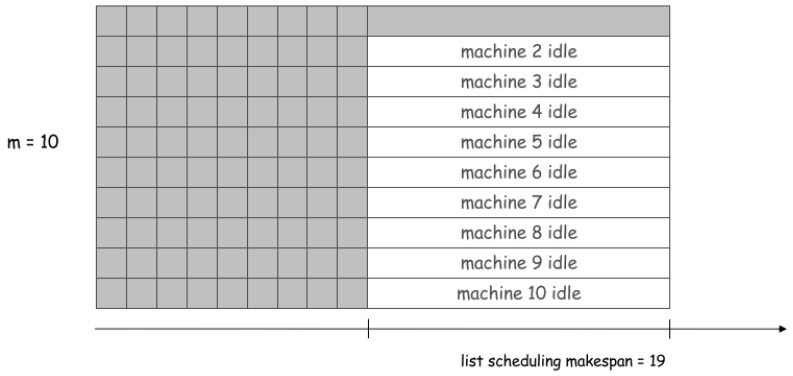
, that is, the greedy algorithm is a 2-approximation. ■

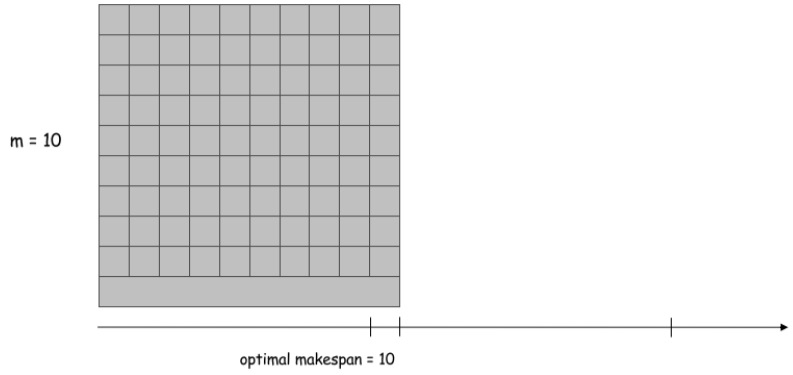
3.2.2 LPT Rule

Can we do better than this? Consider the following case where we have:

- m machines;
- $m(m-1)$ jobs of length 1;
- one job of length m.

In this case, the makespan of the list scheduling algorithm is clearly worse than the optimal one, as we can see in the following image.





The idea of this second approach is the following: **sort** the n jobs in **descending** order of processing time, and then run list scheduling algorithm (LPT, Long Processing Time).
The new algorithm is the following.

```
LPT-List-Scheduling(m, n, t1,t2,...,tn) {
   Sort jobs so that t1 ≥ t2 ≥ ... ≥ tn

   for i = 1 to m {
      Li ← 0        ← load on machine i
      J(i) ← φ      ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = minimum mink Lk ← machine i has smallest load
      J(i) ← J(i) ∪ {j}   ← assign job j to machine i
      Li ← Li + tj        ← update load of machine i
   }
   return J(1), …, J(m)
}
```

As we can see, the complexity does not change, so it is again $O(n \log m)$. Moreover, we can notice that if we have at most m jobs, then list-scheduling is optimal, since each job put on its own machine. A more complex discussion arises if we have more than m jobs.

Lemma 3. *If there are more than m jobs,* $L^* \geq 2t_{m+1}$.

Proof. Consider the first $m+1$ jobs $t_1, .., t_{m+1}$. Since the t_i's are in descending order, each takes at least t_{m+1} time. There are $m+1$ jobs and m machines, so by pigeonhole principle, at least one machine gets two jobs.

Theorem. *LPT rule is a 3/2-approximation algorithm.*

Proof. We follow the same basic approach as for list scheduling:

$$L_i = (L_i - t_j) + t_j$$

but:

- $(L_i - t_j) \leq L^*$;
- $t_j \leq 1/2L^*$ from Lemma 3 (by observation, we can assume that the number of jobs $> m$)

so we conclude that:

$$L_i \leq \frac{3}{2} L^* \qquad \blacksquare$$

An even better approximation was discovered in 1969, when Graham proved that LPT rule is a 4/3-approximation (lower-bound).

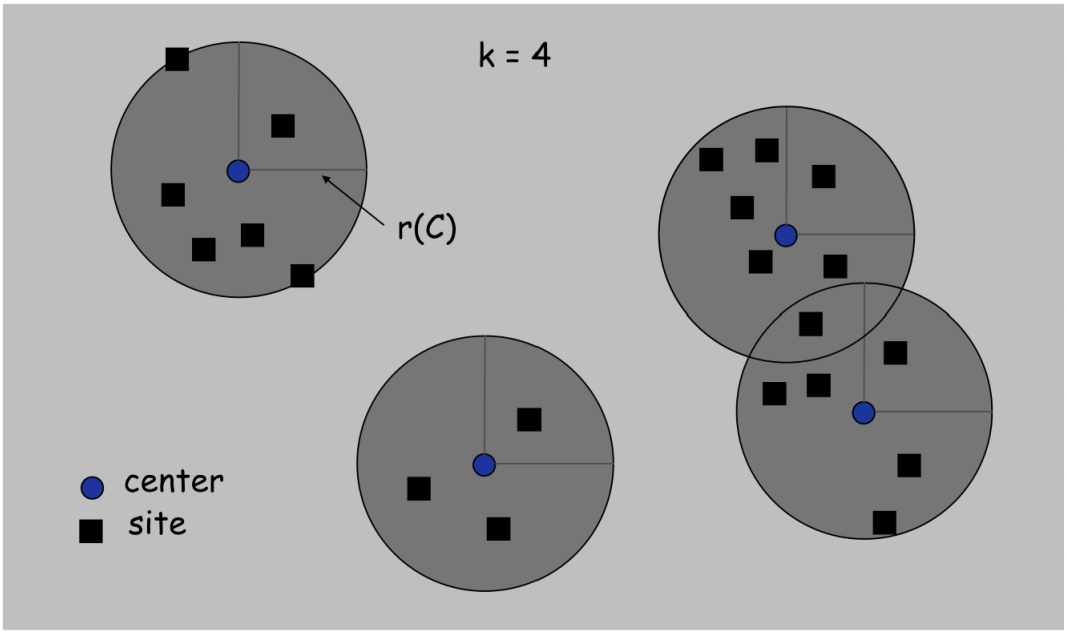
3.3 Center selection

Like the problem in the previous section, the Center Selection Problem, which we consider here, also relates to the general task of allocating work across multiple servers. The **issue** at the heart of Center Selection is **where best to place the servers**; in order to keep the formulation clean

and simple, we will not incorporate the notion of load balancing into the problem. The Center Selection Problem also provides an example of a case in which the most natural greedy algorithm can result in an arbitrarily bad solution, but a slightly different greedy method is guaranteed to always result in a near-optimal solution.

3.3.1 The problem

Consider the following scenario. We have a set S of n sites: we want to select $k > 0$ centers so that the **maximum distance** from a site to the nearest center is **minimized**.



Let us start by defining the input to our problem more formally. We are given:

- An integer k;
- A set S of n sites;
- A distance function.

When we consider instances where the sites are points in the plane, the distance function will be the standard **Euclidean distance** between points, and any point in the plane is an option for placing a center. The algorithm we develop, however, can be applied to more **general notions of distance**: we will allow any distance function that satisfies the following natural properties.

- **Identity**: $dist(s, s) = 0$ for all $s \in S$;
- **Symmetry**: $dist(s, z) = dist(z, s)$ for all sites $s, z \in S$;
- **Triangle inequality**: $dist(s, z) + dist(z, h) \geq dist(s, h)$.

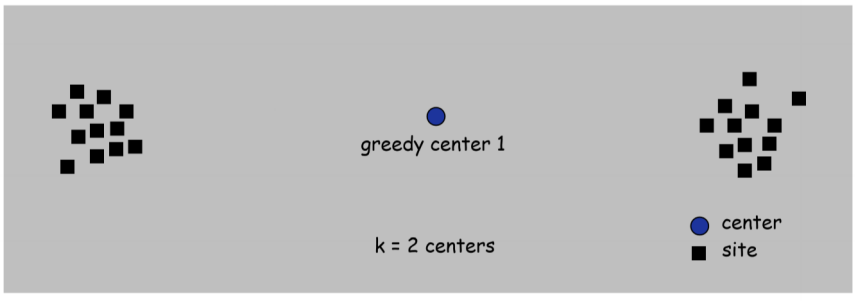
The **first** and **third** of these properties tend to be satisfied by essentially all natural notions of distance. Although there are applications with asymmetric distances, most cases of interest also satisfy the **second** property. Our greedy algorithm will apply to any distance function that satisfies these three properties, and it will depend on all three.
Next we have to clarify what we mean by the goal of wanting the centers to be **"central"**. Let C be a set of centers, then:

- The distance of a site s to the centers is defined as $dist(s, C) = \min_{c \in C} dist(s, c)$, i.e. the distance between s and the closest center;
- We say that C forms an r-cover if each site is within distance at most r from one of the centers, that is, if $dist(s, C) \leq r$ for all sites $s \in S$. The minimum r for which C is an r-cover will be called the covering radius of C and will be denoted by $r(C)$. More formally, $r(C) = \max_i dist(s_i, C)$, i.e. the smallest covering radius.

In other words, the **covering radius** of a set of centers C is the **farthest** that anyone needs to travel to get to his or her **nearest center**. Our goal will be to select a set C of k centers for which $r(C)$ is as **small** as possible.

3.3.2 Greedy algorithm

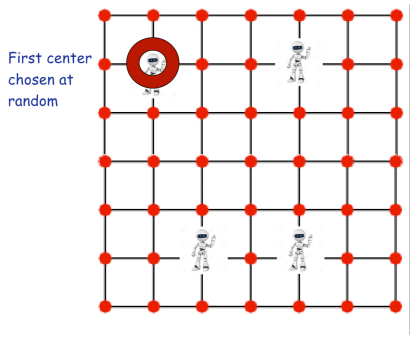
We now discuss greedy algorithms for this problem. As before, the meaning of "greedy" here is necessarily a little fuzzy; essentially, we consider algorithms that select sites one by one in a myopic fashion (that is, choosing each without explicitly considering where the remaining sites will go).
Probably the simplest greedy algorithm would work as follows. It would put the **first center** at the **best possible location** for a single center, then keep **adding centers** so as to reduce the covering radius, each time, by as much as possible. It turns out that this approach is a bit **too simplistic** to be effective: there are cases where it can lead to very bad solutions.
To see that this simple greedy approach can be really bad, consider an example with only two sites s and z, and $k = 2$. Assume that s and z are located in the plane, with distance equal to the standard Euclidean distance in the plane, and that any point in the plane is an option for placing a center. Let d be the distance between s and z. Then the best location for a single center c_1 is halfway between s and z, and the covering radius of this one center is $r(\{c_1\}) = d/2$. The greedy algorithm would start with c_1 as the first center. No matter where we add a second center, at least one of s or z will have the center c_1 as closest, and so the covering radius of the set of two centers will still be $d/2$. Note that the optimum solution with $k = 2$ is to select s and z themselves as the centers. This will lead to a covering radius of 0.
A more complex example illustrating the same problem can be obtained by having two dense "clusters" of sites, one around s and one around z. Here our proposed greedy algorithm would start by opening a center halfway between the clusters, while the optimum solution would open a separate center for each cluster.

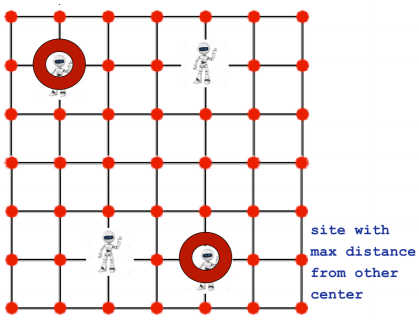


For implementing this algorithm, we can simply select the site s that is farthest away from all previously selected centers: If there is any site at least $2r$ away from all previously chosen centers, then this farthest site s must be one of them. Here is the resulting algorithm.
Note: upon termination, all centers in C are pairwise at least $r(C)$ apart, by construction of the algorithm.

```
Assume k ≤ |S| (else define C = S)
Select any site s and let C = {s}
While |C| < k
   Select a site s ∈ S that maximizes dist(s, C)
   Add site s to C
EndWhile
Return C as the selected set of sites
```

Example. *Apply the greedy algorithm on a grid with 4 sites and $k = 2$ centers.*





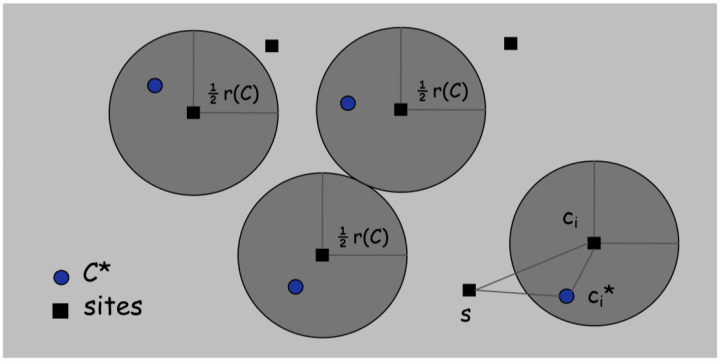
Theorem. *Greedy algorithm is a 2-approximation for center selection problem.*

Proof. Let C^* be an optimal set of centers, then we want to prove that $r(C) \leq 2r(C^*)$. Let $r(C^*)$ denote the minimum possible radius of a set of k centers. For the proof, we assume that we obtain a set of k centers C with $r(C^*) < \frac{1}{2}r(C)$, and from this we derive a contradiction.

- For each site $c_i \in C$, consider the ball of radius $\frac{1}{2}r(C)$ around it;
- We have exactly one c_i^* (i.e. center of the optimal solution), and let c_i be the site paired with c_i^*;
- Consider any site s and its closest center c_i^* in C^*:

$$dist(s, C) \leq dist(s, c_i) \leq dist(s, c_i^*) + dist(c_i^*, c_i) \leq 2r(C^*)$$

The first inequality comes from the fact that C is the set of all centers, the second from the triangular inequality, while the last one comes from the fact that both $dist(s, c_i^*)$ and $dist(c_i^*, c_i)$ are $\leq r(C^*)$ since c_i^* is the closest center. ■



Notice that the greedy algorithm always places centers at sites, but is still within a **factor of 2** of best solution that is allowed to place centers anywhere. Is there a hope of a 3/2-approximation, or a 4/3?

Theorem. *Unless $P = NP$, there is no ρ-approximation for center-selection problem for $\rho < 2$.*

3.4 Metric Traveling Salesman Problem (Metric TSP)

In the traveling-salesman problem we are given a complete undirected graph $G = (V, E)$ that has a non-negative integer cost $c(u, v)$ associated with each edge $u, v \in E$, and we must find a **hamiltonian cycle** (a tour) of G with **minimum cost**.

Theorem. *There is no constant factor approximation algorithm for TSP, unless $P = NP$.*

In many practical situations, the least costly way to go from a place u to a place w is to go directly, with no intermediate steps. Put another way, cutting out an intermediate stop never increases the cost. We formalize this notion by saying that the cost function c satisfies the **triangle inequality** if, for all vertices $u, v, w \in V$:

$$c(u, w) \leq c(u, v) + c(v, w)$$

The triangle inequality seems as though it should naturally hold, and it is **automatically satisfied in several applications**. For example, if the vertices of the graph are points in the plane and the cost of traveling between two vertices is the ordinary euclidean distance between them, then the triangle inequality is satisfied. Furthermore, many cost functions other than euclidean distance satisfy the triangle inequality.

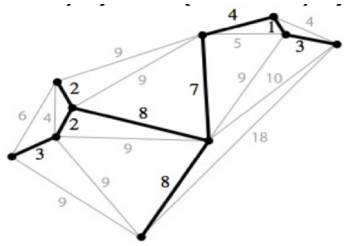
Definition (Metric TSP problem). *Given a complete graph with edge costs satisfying triangle inequalities, find a **minimum cost cycle** visiting **every vertex** exactly once.*

This problem is a special case of the standard TSP problem, is **NP-hard**, and it is characterized by a 2-approximation and a 3/2-approximation. In the next section we will discuss such approximations.

3.4.1 Approximation algorithms for metric TSP

2-approximation The algorithm is the following:

1. **Find** the minimum spanning tree of G: $MST(G)$;



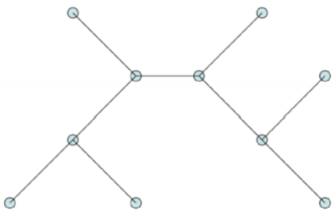
   We recall that a MST is defined as follows.

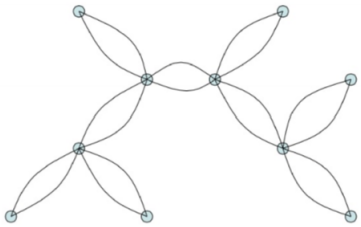
   Definition (Spanning tree). *Given a connected, undirected graph, a* ***spanning tree*** *of that graph is a* ***subgraph*** *that is a* ***tree*** *and* ***connects all the vertices*** *together.*

   Definition (Minimum spanning tree). *A minimum spanning tree (****MST****) or minimum weight spanning tree is then a* ***spanning tree*** *with* ***minimum weight****.*

   In order to find an MST of a graph, we can use the Prim's algorithm, whose complexity is $O(m \log n)$ or $O(m + n \log n)$, depending on the data structure;

2. Take $MST(G)$ and **double** the edges: $T = 2 * MST(G)$;



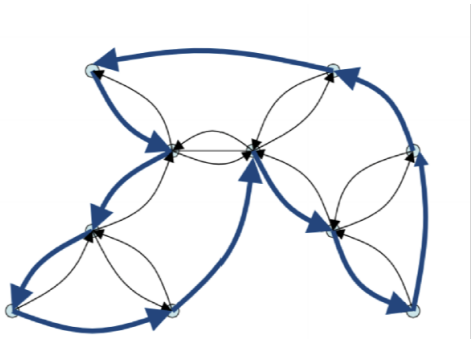


3. The graph T is **Eulerian** (we have an even degree for each node, and the graph is connected), so we can traverse it, visiting each edge exactly once;
4. Create **shortcuts** in the Eulerian tour, to create a tour.

   Start at an arbitrary point and follow the tour, but mark vertices when you visit them as already visited. Later, when encountering a vertex previously visited, just skip over it and go directly to the next vertex on the tour. If that vertex has also already been seen, go on to the next, etc. Keep going until encountering the start vertex again, i.e., create shortcuts in the Euler tour, to create a tour.

   Note: the initial graph is complete: by triangular inequalities, the shortcut tour is not longer than the Eulerian tour.

Theorem. *The above algorithm gives a 2-approximation for the TSP problem (in a metric graph with the triangle inequality).*



Proof. Let us define:

- OPT as the optimal solution (i.e. set of edges) for TSP;
- A as the set of edges chosen by the algorithm;
- EC as the set of edges in the Eulerian cycle.

We have that:

$$cost(T) \leq cost(OPT)$$

since OPT is a cycle, remove any edge and obtain a spanning tree, MST is a lower bound of TSP.
Then,

$$cost(EC) = 2cost(T)$$

because every edge appears twice.
Thus,

$$cost(EC) = 2cost(T) \leq 2cost(OPT)$$

Since $cost(A) \leq cost(EC)$ (because of triangle inequalities, shortcutting) we finally state that

$$cost(A) \leq 2cost(OPT) \qquad \blacksquare$$

3/2-approximation Before analyzing the algorithm, let us define some concepts.

Definition (Graph matching). *A **matching** M in a graph $G(V, E)$ is a **subset of the edges** of G such that **no two edges** in M are **incident** to a common vertex. The weight of M is the sum of its edge weights.*

Definition (Perfect matching). *A **perfect matching** is a matching in which **every vertex is matched**.*

The matching problem can be solved in **polynomial time**.
The 3/2-**approximation algorithm** is the following:

1. **Find** the minimum spanning tree of G: $MST(G)$;
2. **Locate** the **odd degree vertices** in the MST (any graph has an even number of odd degree nodes);

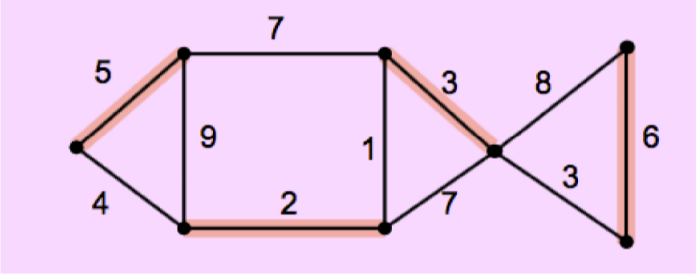
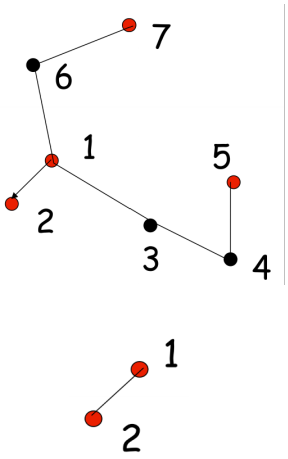
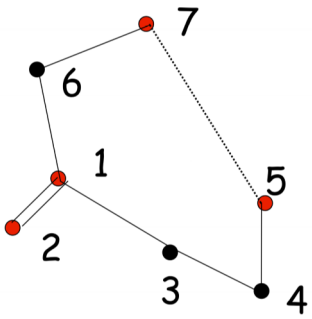


Figure 2: *Example of perfect matching with weight* 16.



3. Let M be the minimum weight perfect matching on the odd degree nodes located at the previous step;
4. **Merge** the perfect edges with MST ($E = M + MST$): E is Eulerian, so find a Eulerian walk of E;



5. **Bypass repeated nodes** on the Eulerian walk to get a TSP tour.

Theorem. *The above algorithm gives a* 3/2*-approximation for the TSP problem (in a metric graph with the triangle inequality).*

Proof. We have:

$$cost(T) \leq cost(OPT)$$

since OPT is a cycle, remove any edge and obtain a spanning tree, MST is a lower bound of TSP.
Then,

$$cost(M) \leq 0.5 * cost(OPT)$$

, since a tour contains 2 matchings, thus the cost of a minimum weight perfect matching is at most $0.5 * C(OPT)$.
Moreover,

$$cost(EC) = cost(T) + cost(M) \leq 1.5 * cost(OPT)$$

and

$$cost(A) \leq cost(EC) \leq 1.5 * cost(OPT)$$

since there are shortcuttings.
Thus,

$$cost(A) \leq 1.5 * cost(OPT) \qquad \blacksquare$$

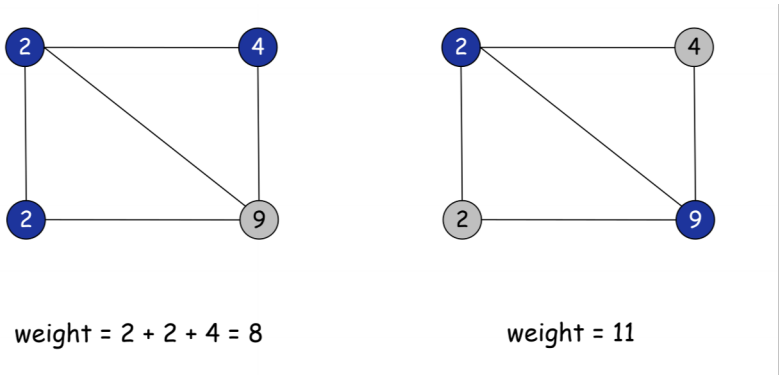
3.5 The Pricing Method: Vertex Cover

We now turn to our second general technique for designing approximation algorithms, the pricing method, and we will introduce this technique by considering a version of the Vertex Cover Problem.

3.5.1 The problem

Recall that a vertex cover in a graph $G = (V, E)$ is a set $S \in V$ so that each edge has at least one end in S. In the version of the problem we consider here, each vertex $i \in V$ has a weight $w_i \geq 0$, with the weight of a set S of vertices denoted $w(S) = \sum_{i \in S} w_i$. We would like to find a **vertex cover** S for which $w(S)$ is **minimum**.

Example. *In this example, we have two different vertex covers, but the first one has minimum weight.*



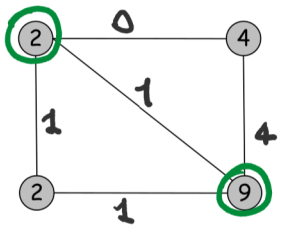
When all weights are equal to 1, deciding if there is a vertex cover of weight at most k is the standard decision version of Vertex Cover.

3.5.2 The Pricing Method

The pricing method (also known as the primal-dual method) is motivated by an economic perspective. For the case of the Vertex Cover Problem, we will think of the **weights** on the nodes as **costs**, and we will think of each **edge** as having to pay for its "share" of the cost of the vertex

cover we find. We will think of the weight w_i of the vertex i as the cost for using i in the cover. We will think of each edge e as a separate "agent" who is willing to "pay" something to the node that covers it. The **algorithm** will not only find a **vertex cover** S, but also determine **prices** $p_e \geq 0$ for each edge $e \in E$, so that if each edge $e \in E$ pays the price p_e, this will in total approximately **cover** the **cost** of S.
First of all, selecting a vertex i covers all edges incident to i, so it would be "unfair" to charge these incident edges in total more than the cost of vertex i.

Definition (Fairness). *We call prices p_e **fair** if, for each vertex i, the edges adjacent to i do not have to pay more than the cost of the vertex, i.e. $\sum_{e=(i,j)} p_e \leq w_i$.*



A useful fact about fair prices is that they provide a lower bound on the cost of any solution.

Lemma (Fairness Lemma). *For any vertex cover S and any fair price p_e, $\sum_e p_e \leq w(S)$.*
Proof. We have that:

$$\sum_e p_e \leq \sum_{i \in S} \sum_{e=(i,j)} p_e \leq \sum_{i \in S} w_i = w(S)$$

Note:

- The fist inequality comes from the fact that each edge e is covered by at least one node in S;
- The second inequality comes from the fact that we sum fairness inequalities for each node in S (recall, from fairness definition, that $\sum_{e=(i,j)} p_e \leq w_i$)

3.5.3 The algorithm

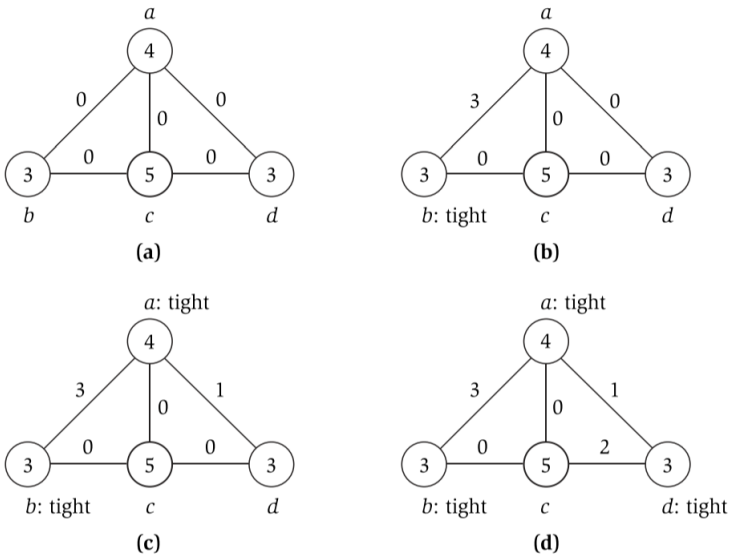
The **goal** of the approximation algorithm will be to find a **vertex cover** and to **set prices** at the **same time**. We can think of the algorithm as being **greedy** in how it sets the prices. It then uses these prices to drive the way it selects nodes for the vertex cover.

Definition (Tight node). *We define vertex i **tight** if $\sum_{e=(i,j)} p_e = w_i$*

The algorithm is the following.

Example. *Consider the execution of the algorithm on the following graph. Initially, no node is tight; the algorithm decides to select the edge (a, b). It can raise the price paid by (a, b) up to 3, at which point the node b becomes tight and it stops. The algorithm then selects the edge (a, d).It can only raise this price up to 1, since at this point the node a becomes tight (due to the fact that the weight of a is 4, and it is already incident to an edge that is paying 3). Finally, the algorithm selects the edge (c, d). It can raise the price paid by (c, d) up to 2, at which point d becomes tight. We now have a situation where all edges have at least one tight end, so the algorithm terminates.*

```
Vertex-Cover-Approx(G, w):
  Set p_e = 0 for all e ∈ E
  While there is an edge e = (i, j) such that neither i nor j is tight
     Select such an edge e
     Increase p_e without violating fairness
  EndWhile
  Let S be the set of all tight nodes
  Return S
```



The tight nodes are a, b and d; so this is the resulting vertex cover. (Note that this is not the minimum-weight vertex cover; that would be obtained by selecting a and c.)

3.5.4 Analyzing the algorithm

Termination The algorithm **terminates** since at least one new node becomes tight after each iteration of while loop and the edges are finite.

Correctness Let S be the set of all tight nodes upon termination of algorithm. S is a **vertex cover**: if some edge $i - j$ is uncovered, then neither i nor j is tight. But then while loop would not terminate.

Theorem. *The previous algorithm gives a 2-approximation for the Vertex Cover problem.*

Proof. Let S^* be **optimal** vertex cover. Then, $w(S) \leq 2w(S^*)$. We have that:

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e \leq \sum_{i \in V} \sum_{e=(i,j)} p_e = 2 \sum_{e \in E} p_e \leq 2w(S^*)$$

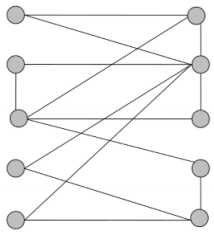
Note:

- $w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e$ because all nodes in S are tight;

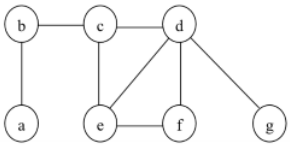
- $\sum_{i \in S} \sum_{e=(i,j)} p_e \leq \sum_{i \in V} \sum_{e=(i,j)} p_e$ because $S \subseteq V$ and all the prices are ≥ 0;
- $\sum_{i \in V} \sum_{e=(i,j)} p_e = 2 \sum_{e \in E} p_e$ because each edge is counted twice;
- Finally, the last inequality derives from the **fairness lemma**, which states that for any S, $\sum_e p_e \leq w(S)$, so it also holds that $\sum_e p_e \leq w(S^*)$, which is the optimal one.

3.6 Exercises

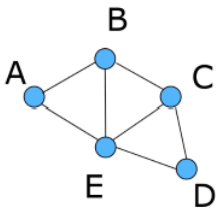
1. Try to run the 2-approximation algorithm for the VERTEX COVER problem on this graph;



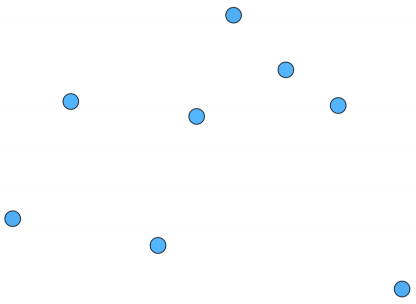
2. Try to run the 2-approximation algorithm for the VERTEX COVER problem on this graph. Solution on slide 8 of L4;



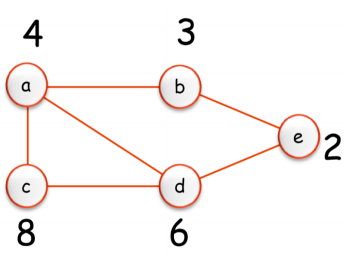
3. Try to run the 2-approximation algorithm for the VERTEX COVER problem on this graph;



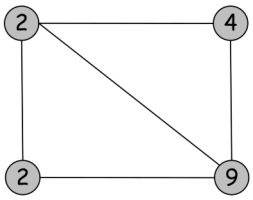
4. Consider the center selection problem of the example. Compute $r(C)$, find the optimal solution, the related centers and $r(C^*)$: check the relation $r(C) \leq 2r(C^*)$ for this case. Solution on slide 16 of L5;
5. Try to run the center selection algorithm with 3 centers. What is $r(C)$, what is $r(C^*)$? Solution on slide 24 of L5;



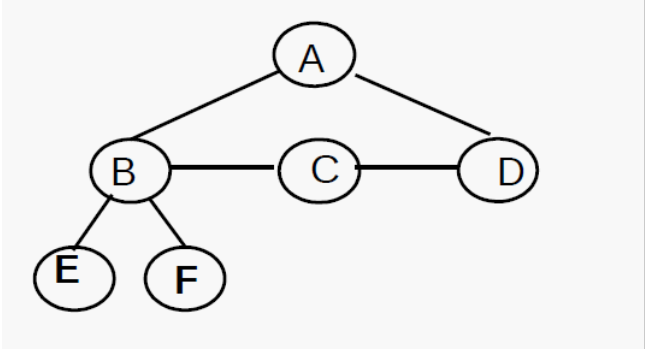
6. Define the metric TSP problem, describe a 2-approximation algorithm and show an example. Solution on slide 23-27 of L11;
7. Run metric TSP algorithms on a complete graph of 5 nodes. Find a good assignment to the edges so that the triangle inequality holds. Solution on slides 4-5 of L6;
8. Apply the pricing method for finding the vertex cover of minimum weight of the following graph. Solution on slides 15-20 of L6;



9. Apply the pricing method for finding the vertex cover of minimum weight of the following graph;

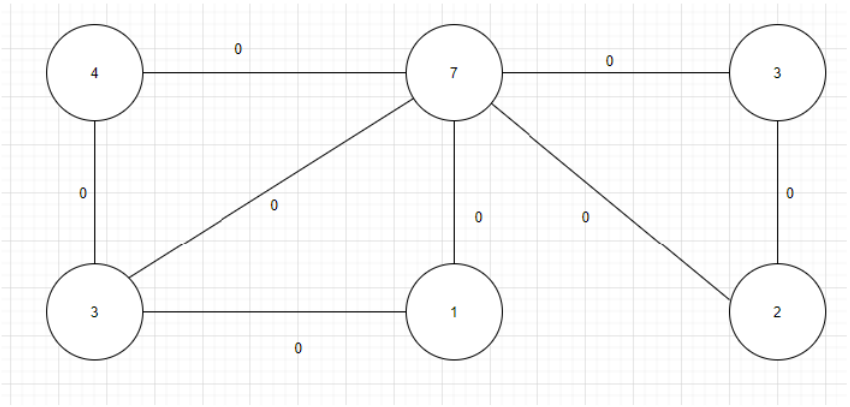


10. Define the load balancing problem, the load of a machine and the makespan. Does the problem belong to the class P? If so provide a polynomial time exact algorithm, otherwise provide a simple 2-approximation algorithm. First describe the algorithm (in code, or pseudocode or words) in a complete and precise way, then show how the algorithm works on a small example. Can we improve the algorithm? Describe an improved variation of the algorithm and show a small example. Solution on slide 7-13 of L11.5;
11. What is the vertex cover problem? Describe a trivial 2-approximation algorithm (with code, pseudocode, words, as long as it is complete, precise and clear). Provide an execution example on this graph.



12. Define the vertex cover of minimum weight. Describe in a complete and precise way (in code, or pseudocode or words) the algorithm seen in class to solve the weighted vertex

cover problem. Show how the algorithm works in the following example. Solution on slide 20-24 of L11.5



4 Local Search

In the previous two chapters, we have considered techniques for dealing with computationally intractable problems: in Chapter 2, by identifying structured special cases of NP-hard problems, and in Chapter 3, by designing polynomial-time approximation algorithms. We now develop a third and final topic related to this theme: the design of local search algorithms.
Local search is a very general technique; it describes any algorithm that "explores" the space of possible solutions in a sequential fashion, moving in one step from a current solution to a "nearby" one. The generality and flexibility of this notion has the advantage that it is not difficult to design a local search approach to almost any computationally hard problem; the counterbalancing disadvantage is that it is often very difficult to say anything precise or provable about the quality of the solutions that a local search algorithm finds, and consequently very hard to tell whether one is using a good local search algorithm or a poor one. Our discussion of local search in this chapter will reflect these trade-offs.
Local search algorithms are generally heuristics designed to find good, but not necessarily optimal, solutions to computational problems, and we begin by talking about what the search for such solutions looks like at a global level. A useful intuitive basis for this perspective comes from connections with energy minimization principles in physics, and we explore this issue first.

4.1 Landscape of an optimization problem

Much of the core of local search was developed by people thinking in terms of analogies with **physics**. Looking at the wide range of hard computational problems that require the minimization of some quantity, they reasoned as follows. Physical systems are performing **minimization** all the time, when they seek to minimize their potential energy. What can we learn from the ways in which nature performs minimization? Does it suggest new kinds of algorithms?

4.1.1 Potential energy

If the world really looked the way a freshman mechanics class suggests, it seems that it would consist entirely of hockey pucks sliding on ice and balls rolling down inclined surfaces. Hockey pucks usually slide because you push them; but why do balls roll downhill?

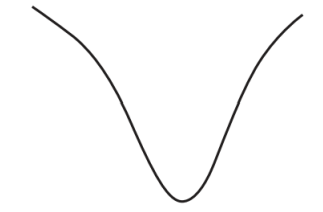


Figure 3: *When the potential energy landscape has the structure of a simple funnel, it is easy to find the lowest point.*

One perspective that we learn from Newtonian mechanics is that the ball is trying to minimize its potential energy: so, if we release a ball from the top of the funnel-shaped landscape in Figure 4.1.1, its potential energy will be **minimized** at the lowest point.
If we make the **landscape** a little more **complicated**, some extra issues creep in. Consider the **"double funnel"** in Figure 4.1.1.
Point A is lower than point B, and so is a more desirable place for the ball to come to rest. But if we start the ball rolling from point C, it will not be able to get over the barrier between the two funnels, and it will end up at B. We say that the ball has become **trapped** in a local minimum:

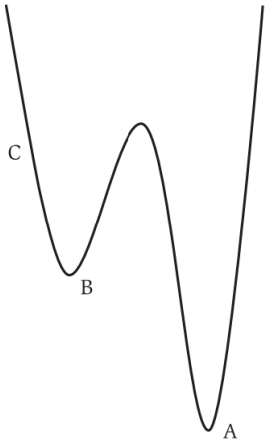


Figure 4: *Most landscapes are more complicated than simple funnels; for example, in this "double funnel," there's a deep global minimum and a shallower local minimum*

It is at the lowest point if one looks in the neighborhood of its current location; but stepping back and looking at the whole landscape, we see that it has missed the global minimum.

4.1.2 The connection to optimization

This perspective on energy minimization has really been based on the following core ingredients: The physical system can be in one of a large number of possible states; its energy is a function of its current state; and from a given state, a small perturbation leads to a "neighboring" state. The way in which these neighboring states are linked together, along with the structure of the energy function on them, defines the underlying energy landscape.
It's from this perspective that we again start to think about computational minimization problems. In a typical such problem, we have:

- A large (typically exponential-size) set C of possible solutions;
- A cost function $c(.)$ that measures the quality of each solution: for a solution $S \in C$, we write its cost as $c(S)$.

The goal is to find a **solution** $S^* \in C$ for which $c(S^*)$ is as **small** as possible.
So far this is just the way we've thought about such problems all along. We now add to this the notion of a **neighbor relation** on solutions, to capture the idea that one solution S' can be obtained by a **small modification** of another solution S. We write $S \sim S'$ to denote that S' is a neighboring solution of S, and we use $N(S)$ to denote the neighborhood of S, the set $\{S' : S \sim S'\}$.
We will primarily be considering **symmetric** neighbor relations here, though the basic points we discuss will apply to asymmetric neighbor relations as well. A crucial point is that, while the set C of possible solutions and the cost function $c(.)$ are provided by the specification of the problem, we have the freedom to make up any neighbor relation that we want.
A *local search algorithm* takes this setup, including a neighbor relation, and works according to the following high-level scheme. At all times, it maintains a **current solution** $S \in C$. In a given step, it chooses a **neighbor** S' of S, declares S' to be the **new current solution**, and iterates. Throughout the execution of the algorithm, it remembers the **minimum-cost solution** that it has seen thus far; so, as it runs, it gradually finds **better** and better **solutions**. Note that there's **no guarantee** that we'll end up at the **best solution**. The crux of a local search

algorithm is in the **choice** of the **neighbor relation**, and in the design of the rule for **choosing** a **neighboring solution** at each step.
Thus one can think of a neighbor relation as defining a (generally undirected) graph on the set of all possible solutions, with edges joining neighboring pairs of solutions. A **local search algorithm** can then be viewed as performing a **walk on this graph**, trying to move toward a good solution.

An application to the vertex cover problem This is still all somewhat vague without a concrete problem to think about; so we'll use the Vertex Cover Problem as a running example here. It's important to keep in mind that, while Vertex Cover makes for a good example, there are many other optimization problems that would work just as well for this illustration.
Thus we are given a graph $G = (V, E)$, and the goal is to find a **subset of nodes** S of **minimal cardinality** such that each edge in E has at least one end in S; the set C of possible solutions consists of all subsets S of V that form vertex covers. Hence, for example, we always have $V \in C$. The cost $c(S)$ of a vertex cover S will simply be its size; in this way, **minimizing** the **cost of a vertex cover** is the same as finding one of minimum size. Finally, we will focus our examples on local search algorithms that use a particularly simple neighbor relation: we say that $S \sim S'$ if S' can be obtained from S by adding or deleting a single node. Thus our local search algorithms will be **walking** through the **space** of possible **vertex covers**, **adding** or **deleting** a **node** to their current solution in each step, and trying to find as small a vertex cover as possible.
One useful fact about this neighbor relation is the following: each vertex cover S has at most n neighboring solutions. The reason is simply that each neighboring solution of S is obtained by adding or deleting a distinct node. A consequence of this property is that we can efficiently examine all possible neighboring solutions of S in the process of choosing which to select.
Let's think first about a very **simple** local search algorithm, which we'll term **gradient descent**. Gradient descent starts with the **full vertex set** V and uses the following **rule** for **choosing** a **neighboring solution**.
Let S denote the current solution. If there is a neighbor S' of S with strictly lower cost, then choose the neighbor whose cost is as small as possible. Otherwise terminate the algorithm.
So gradient descent **moves** strictly **"downhill"** as long as it can; once this is no longer possible, it stops.
We can see that gradient descent **terminates** precisely at solutions that are **local minima**: solutions S such that, for all neighboring S', we have $c(S) \leq c(S')$. This definition corresponds very naturally to our notion of local minima in energy landscapes: They are points from which no one-step perturbation will improve the cost function.
How can we **visualize** the behavior of a local search algorithm in terms of the kinds of energy landscapes we illustrated earlier? Let's think first about gradient descent.

- The **easiest** instance of Vertex Cover is surely an n-node graph with **no edges**. The **empty set** is the **optimal solution** (since there are no edges to cover), and gradient descent does exceptionally well at finding this solution: It starts with the full vertex set V, and keeps deleting nodes until there are none left. Indeed, the set of vertex covers for this edge-less graph corresponds naturally to the funnel we drew in Figure 4.1.1: The unique local minimum is the global minimum, and there is a downhill path to it from any point;
- The hardest instance corresponds to the **"star graph"** G, consisting of nodes $x_1, y_1, y_2, .., y_{n-1}$, with an edge from x_1 to each y_i. The minimum vertex cover for G is the singleton set $\{x_1\}$, and gradient descent can reach this solution by successively deleting $y_1, .., y_{n-1}$ in any order. But, if gradient descent deletes the node x_1 first, then it is immediately stuck: No

node y_i can be deleted without destroying the vertex cover property, so the only neighboring solution is the full node set V, which has higher cost. Thus the algorithm has become trapped in the local minimum $\{y_1, y_2, .., y_{n-1}\}$, which has very high cost relative to the global minimum. Pictorially, we see that we're in a situation corresponding to the double funnel of Figure 4.1.1;

- What kind of graph might yield a Vertex Cover instance with a landscape like the jagged funnel in Figure 4.1.2?

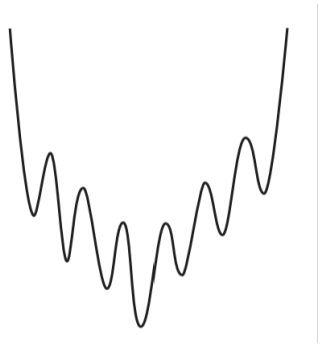


Figure 5: *In a general energy landscape, there may be a very large number of local minima that make it hard to find the global minimum.*

One such graph is simply an n-node path, where n is an odd number, with nodes labeled $v_1, v_2, .., v_n$ in order. The unique minimum vertex cover S^* consists of all nodes v_i where i is even. But there are many local optima. For example, consider the vertex cover $v_2, v_3, v_5, v_6, v_8, v_9, ..$ in which every third node is omitted. This is a vertex cover that is significantly larger than S^*; but there's no way to delete any node from it while still covering all edges.

Another example of local search algorithm is called **Hill-climbing search**: this is the opposite of what we've seen so far, since here we have to maximize our result.

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

4.2 The Metropolis algorithm

The first idea for an improved local search algorithm comes from the work of Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953). This represents an attempt to **improve** the simplified **idea** of **gradient descend** by **simulating** the behavior of a **physical system** according to principles of statistical mechanics. The **intuition** behind the behaviour of this algorithm is

that it is globally **biased** toward **"downhill" steps**, but occasionally makes **"uphill" steps** to **break out** of **local minima** (and to correct wrong choices).
A **basic model** from this field asserts that the probability of finding a physical system in a state with energy E is proportional to the Gibbs-Boltzmann function $e^{-E/(kT)}$, where $T > 0$ is the **temperature** and $k > 0$ is a **constant**.
Let's look at this function.

- For any temperature T, the function is **monotone decreasing** in the energy E, so this states that a physical **system** is **more likely** to be in a **lower energy state** than in a **high energy state**;
- Now let's consider the effect of the **temperature** T:
  - When T is **small**, the **probability** for a **low-energy state** is significantly **larger** than the **probability** for a **high-energy state**;
  - However, if the temperature is **large**, then the **difference** between these two **probabilities** is very **small**, and the system is almost equally likely to be in any state.

Metropolis algorithm Metropolis et al. proposed the following method for performing step-by-step simulation of a system at a **fixed temperature** T. At all times, the simulation maintains a **current state** S of the system and tries to produce a **new state** $S' \in N(S)$ by applying a **perturbation** to this state (we'll assume that we're only interested in states of the system that are "reachable" from some fixed initial state by a sequence of small perturbations, and we'll assume that there is only a finite set C of such states.)

1. In a single step, we first generate a **small random perturbation** to the current state S of the system, resulting in a **new state** S'. Let $E(S)$ and $E(S')$ denote the **energies** of S and S', respectively;
2. If $E(S') \leq E(S)$, then we **update** the current state to be S'. Otherwise let $\Delta E = E(S') - E(S) > 0$;
3. We update the current state to be S' with **probability** $e^{-\Delta E/(kT)}$, and otherwise leave the current state at S.

Let's see some examples:

- On the Vertex Cover instance consisting of the star graph we see that the Metropolis Algorithm will quickly bounce out of the local minimum that arises when x_1 is deleted: The neighboring solution in which x_1 is put back in will be generated and will be accepted with positive probability. On more complex graphs as well, the Metropolis Algorithm is able, to some extent, to correct the wrong choices it makes as it proceeds;
- In the case of a graph with no edges, the gradient descent solves this instance with no trouble, deleting nodes in sequence until none are left. But, while the Metropolis Algorithm will start out this way, it begins to go astray as it nears the global optimum. Consider the situation in which the current solution contains only c nodes, where c is much smaller than the total number of nodes, n. With very high probability, the neighboring solution generated by the Metropolis Algorithm will have size $c + 1$, rather than $c - 1$, and with reasonable probability this uphill move will be accepted. Hence, the algorithm in this case has a tendency to step away from good solutions.

4.3 Hopfield Neural Networks

Thus far we have been discussing local search as a method for trying to find the global optimum in a computational problem. There are some cases, however, in which, by examining the specification of the problem carefully, we discover that it is really just an arbitrary local optimum that is required.We now consider a problem that illustrates this phenomenon.

4.3.1 The problem

The problem we consider here is that of finding stable configurations in *Hopfield neural networks.* **Hopfield networks** have been proposed as a simple **model** of an **associative memory**, in which a large collection of **units** are **connected** by an **underlying network**, and neighboring units try to correlate their states.
Concretely, a Hopfield network can be viewed as an **undirected graph** $G = (V, E)$, with an integer-valued **weight** w_e on each edge e; each weight may be positive or negative. A **configuration** S of the network is an assignment of the value -1 or $+1$ to each node u; we will refer to this value as the **state** s_u of the node u. The meaning of a configuration is that each node u, representing a unit of the neural network, is trying to choose between one of two possible states ("on" or "off"; "yes" or "no"); and its choice is influenced by those of its neighbors as follows. Each **edge** of the network imposes a **requirement** on its endpoints:

- If u is joined to v by an edge of **negative** weight, then u and v want to have the **same state**;
- If u is joined to v by an edge of **positive** weight, then u and v want to have **opposite states**.

The absolute value $|w_e|$ will indicate the **strength** of this requirement, and we will refer to $|w_e|$ as the **absolute weight** of edge e.
Unfortunately, there may be **no configuration** that respects the **requirements** imposed by all the edges. For example, consider three nodes a, b, c all mutually connected to one another by edges of weight 1. Then, no matter what configuration we choose, two of these nodes will have the same state and thus will be violating the requirement that they have opposite states.

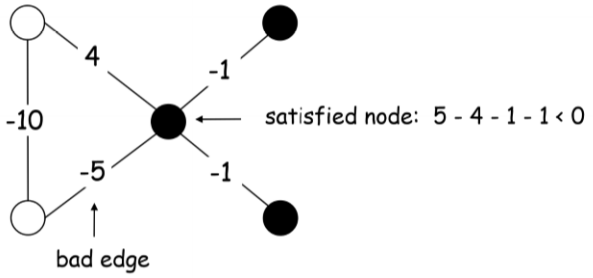
Definition (Good edge). *With respect to a given configuration, we say that an edge $e = (u, v)$ is **good** if the requirement it imposes is satisfied by the states of its two endpoints: either $w_e < 0$ and $s_u = s_v$, or $w_e > 0$ and $s_u \neq s_v$. Otherwise we say e is **bad**. Note that we can express the condition that e is good very compactly, as follows: $w_e s_u s_v < 0$.*

Definition (Node satisfaction). *We say that a node u is **satisfied** in a given configuration if the total **absolute weight** of all **good** edges incident to u is at least **as large** as the total **absolute weight** of all **bad** edges incident to u. We can write this as*

$$\sum_{v:e=(u,v)} w_e s_u s_v \leq 0$$

Definition (Stable configuration). *A configuration is **stable** if all nodes are **satisfied***

Why do we use the term **stable** for such configurations? This is based on viewing the network from the perspective of an individual node u. On its own, the only choice u has is whether to take the state -1 or $+1$; and like all nodes, it wants to respect as many edge requirements as



possible (as measured in absolute weight). Suppose u asks: Should I flip my current state? We see that if u does flip its state (while all other nodes keep their states the same), then all the good edges incident to u become bad, and all the bad edges incident to u become good. So, to maximize the amount of good edge weight under its direct control, u should flip its state if and only if it is not satisfied. In other words, a stable configuration is one in which no individual node has an incentive to flip its current state.
A basic question now arises: Does a Hopfield network always have a **stable configuration**, and if so, how can we find one?

4.3.2 State-flipping algorithm

The intuition behind the **algorithm** is to repeatedly **flip** the **state** of an **unsatisfied node**.

```
While the current configuration is not stable
  There must be an unsatisfied node
  Choose an unsatisfied node u
  Flip the state of u
Endwhile
```

Figure 6: *State-flipping algorithm*

An **example** of the execution of this algorithm is depicted in Figure 4.3.2, ending in a stable configuration.

Analyzing the algorithm

Theorem. *State-flipping algorithm terminates with a stable configuration after at most* $W = \sum_e |w_e|$ *iterations.*

Clearly, **if** the State-flipping **algorithm** we have just defined **terminates**, we will have a **stable configuration**. What is not obvious is whether it must in fact terminate. Indeed, in the earlier directed example, this process will simply cycle through the three nodes, flipping their states sequentially forever.
We now **prove** that the state-flipping algorithm always **terminates**, and we give a **bound** on the **number of iterations** it takes until termination. This will provide a proof of Theorem 4.3.2. The key to proving that this process terminates is an idea we've used in several previous situations: to look for a **measure of progress**, i.e. a **quantity** that **strictly increases** with

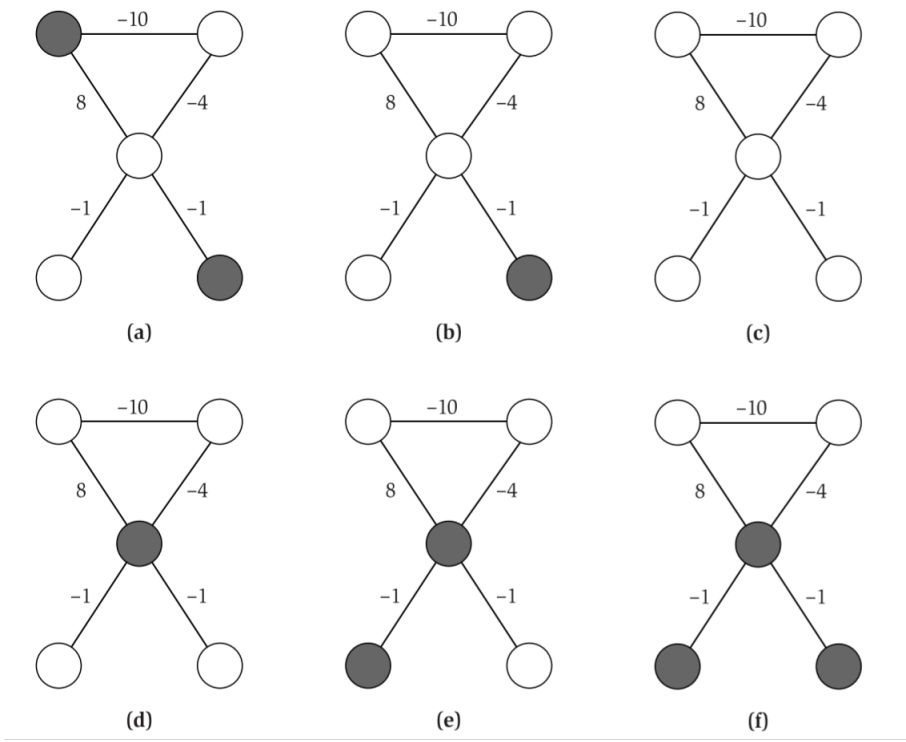


Figure 7: *Parts (a)–(f) depict the steps in an execution of the State-Flipping Algorithm for a five-node Hopfield network, ending in a stable configuration. (Nodes are colored black or white to indicate their state.)*

every **flip** and has an **absolute upper bound**. This can be used to bound the number of iterations.
Probably the most natural progress measure would be the **number of satisfied nodes**: If this increased every time we flipped an unsatisfied node, the process would run for at most n iterations before terminating with a stable configuration. Unfortunately, **this does not turn out to work**. When we flip an unsatisfied node v, it's true that it has now become satisfied, but several of its previously satisfied neighbors could now become unsatisfied, resulting in a net decrease in the number of satisfied nodes. This actually happens in one of the iterations depicted in Figure 4.3.2: when the middle node changes state, it renders both of its (formerly satisfied) lower neighbors unsatisfied.
However, there is a more subtle progress measure that does increase with each flip of an unsatisfied node. Specifically, for a given configuration S,we define $\Phi(S)$ to be the **total absolute weight of all good edges** in the network. That is,

$$\Phi(S) = \sum_{\text{good edge } e} |w_e|$$

Note:

- Clearly, for any configuration S, we have $\Phi(S) \geq 0$ (since $\Phi(S)$ is a sum of positive integers), and $\Phi(S) \leq W = \sum_e |w_e|$ (since, at most, every edge is good);
- Now suppose that, in a **non-stable configuration** S, we choose a node u that is **unsatisfied** and **flip** its state, resulting in a configuration S'. What can we say about the relationship of $\Phi(S')$ to $\Phi(S)$? When u flips its state:

- All **good** edges incident to u become **bad**;
- All **bad** edges incident to u become **good**;
- All edges that don't have u as an endpoint remain the same.

So, if we let g_u and b_u denote the **total absolute weight on good and bad edges** incident to u, respectively, then we have

$$\Phi(S') = \Phi(S) - g_u + b_u$$

But, since u was unsatisfied in S, we also know that $b_u > g_u$; and since b_u and g_u are both integers, we in fact have $b_u \geq g_u + 1$. Thus

$$\Phi(S') \geq \Phi(S) + 1$$

Hence the value of Φ begins at some non-negative integer, increases by at least 1 on every flip, and cannot exceed W. Thus our process runs for at most W **iterations**, and when it terminates, we must have a stable configuration.
Moreover, in each iteration we can identify an unsatisfied node using a number of arithmetic operations that is polynomial in n; thus a **running-time bound** that is polynomial in n and W follows as well.
It's worth noting that while our algorithm proves the **existence** of a **stable configuration**, the running time leaves something to be desired when the absolute weights are large. Specifically, the algorithm we obtain here is **polynomial** only in the actual magnitude of the weights, not in the size of their binary representation. For very large weights, this can lead to running times that are quite **infeasible**. However, no simple way around this situation is currently known. It turns out to be an open question to find an algorithm that constructs stable states in time polynomial in n and $\log W$ (rather than n and W), or in a number of primitive arithmetic operations that is polynomial in n alone, independent of the value of W.

4.4 Maximum Cut approximation via Local Search

We now discuss a case where a local search algorithm can be used to provide a provable approximation guarantee for an optimization problem. We will do this by analyzing the structure of the local optima, and bounding the quality of these locally optimal solutions relative to the global optimum. The problem we consider is the Maximum-Cut Problem, which is closely related to the problem of finding stable configurations for Hopfield networks that we saw in the previous section.

4.4.1 The problem

In the **Maximum-Cut Problem**, we are given an undirected graph $G = (V, E)$, with a positive integer weight w_e on each edge e. For a partition (A, B) of the vertex set, we use $w(A, B)$ to denote the total weight of edges with one end in A and the other in B:

$$w(A, B) = \sum_{e=(u,v) u \in A, v \in B} w_e$$

The **goal** is to find a **partition** (A, B) of the vertex set so that $w(A, B)$ is **maximized**. Maximum Cut is **NP-hard**, in the sense that, given a weighted graph G and a bound β, it is **NP-complete** to decide whether there is a partition (A, B) of the vertices of G with $w(A, B) \geq \beta$.

4.4.2 The algorithm

The State-Flipping Algorithm used for Hopfield networks provides a local search algorithm to approximate the Maximum Cut objective function $\Phi(S) = w(A, B)$. In terms of partitions, it says the following: If there exists a node u such that the total weight of edges from u to nodes in its own side of the partition exceeds the total weight of edges from u to nodes on the other side of the partition, then u itself should be moved to the other side of the partition.
We'll call this the **"single-flip" neighborhood** on partitions: Partitions (A, B) and (A', B') are **neighboring** solutions if (A', B') can be obtained from (A, B) by **moving** a **single node** from **one side** of the partition **to the other**.

```
Max-Cut-Local (G, w) {
   Pick a random node partition (A, B)

   while (∃ improving node v) {
      if (v is in A) move v to B
      else           move v to A
   }

   return (A, B)
}
```

4.4.3 Analyzing the algorithm

Let's ask two basic questions:

1. Can we say anything concrete about the **quality** of the **local optima** under the **single-flip neighborhood**?
2. Since the single-flip neighborhood is about as simple as one could imagine, what other **neighborhoods** might yield **stronger local search** algorithms for Maximum Cut?

We address the first of these questions here, and we take up the second one in the next section.

Theorem. *Let (A, B) be a partition that is a local optimum for Maximum Cut under the single-flip neighborhood. Let (A^*, B^*) be a globally optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.*

Proof. Let $W = \sum_e w_e$. We also extend our notation a little: for two nodes u and v, we use w_{uv} to denote w_e if there is an edge e joining u and v, and 0 otherwise.
For any node $u \in A$, we must have

$$\sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

, since otherwise u should be moved to the other side of the partition, and (A, B) would not be locally optimal. Suppose we add up these inequalities for all $u \in A$; any edge that has both ends in A will appear on the left-hand side of exactly two of these inequalities, while any edge that has one end in A and one end in B will appear on the right-hand side of exactly one of these inequalities.
Thus, we have

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A, B)$$

We can apply the same reasoning to the set B, obtaining

$$2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A, B)$$

If we add together the inequalities and divide by 2, we get

$$\sum_{\{u,v\} \subseteq A} w_{uv} + \sum_{\{u,v\} \subseteq B} w_{uv} \leq w(A, B)$$

The **left-hand side** of inequality accounts for a**ll edge weight** that does **not** cross from A to B; so, if we add $w(A, B)$ to both sides, the left-hand side becomes equal to W. The **right-hand side** becomes $2w(A, B)$,so we have $W \leq 2w(A, B)$, or $w(A, B) \geq \frac{1}{2}W$.
Since the globally optimal partition (A^*, B^*) clearly satisfies $w(A^*, B^*) \leq W$, we have $w(A, B) \geq \frac{1}{2}w(A^*, B^*)$.
Notice that we never really thought much about the optimal partition (A^*, B^*) in the proof of the theorem; we really showed the stronger statement that, in any l**ocally optimal solution** under the single-flip neighborhood, at least **half** the total edge weight in the graph **crosses** the **partition**.
Moreover, the theorem proves that a local optimum is a **2-approximation** to the maximum cut. This suggests that the **local optimization** may be a **good** algorithm for approximately **maximizing** the cut value. However, there is one more issue that we need to consider: the **running time**. As we saw in the previous sections, the Single-Flip Algorithm is only **pseudo-polynomial**, since we can only bound it by W and it is an open problem whether a local optimum can be found in **polynomial time**. However, in this case we can do almost as well, simply by stopping the algorithm when there are no **"big enough" improvements**.

Definition (Big improvement flip). *Let (A, B) be a partition with weight $w(A, B)$. For a fixed $\epsilon > 0$, let us say that a single node flip is a **big-improvement-flip** if it improves the cut value by at least $\frac{2\epsilon}{n}w(A, B)$ where $n = |V|$.*

Now, consider a version of the Single-Flip Algorithm when we only accept big-improvement-flips and terminate once no such flip exists, even if the current partition is not a local optimum. We claim that this will lead to almost as good an approximation and will run in polynomial time. First we can extend the previous proof to show that the resulting cut is almost as good. We simply have to add the term $\frac{2\epsilon}{n}w(A, B)$ to each inequality, as all we know is that there are no big-improvement-flips.

Claim. *Let (A, B) be a partition s.t. no big-improvement-flip is possible. Let (A^*, B^*) be a globally optimal partition. Then, $(2 + \epsilon)w(A, B) \geq w(A^*, B^*)$.*

Claim. *The version of the Single-Flip Algorithm that only accepts big-improvement-flips terminates after at most $O(\epsilon^{-1} n \log W)$ flips, assuming the weights are integral, and $W = \sum_e w_e$.*

Theorem (Sahni-Gonzales, 1976). *There exists a $\frac{1}{2}$-approximation algorithm for the maximum cut problem.*

Theorem (Goemans-Williamson, 1995). *There exists an 0.878567-approximation algorithm for the maximum cut problem.*

Theorem (Hastad, 1997). *Unless $P = NP$, no 16/17-approximation algorithm exists for the maximum cut problem.*

4.5 Choosing a neighbor relation

We began the chapter by saying that a local search algorithm is really based on two fundamental ingredients: the **choice** of the **neighbor relation**, and the rule for **choosing** a **neighboring solution** at each step. In previous sections we spent time thinking about the second of these: the Metropolis Algorithm took the neighbor relation as given and modified the way in which a neighboring solution should be chosen.
What are some of the **issues** that should go into our **choice** of the **neighbor relation**? This can turn out to be quite subtle, though at a high level the **trade-off** is a basic one.

1. The **neighborhood** of a **solution** should be **rich** enough that we do not tend to get **stuck** in **bad local optima**; but
2. the **neighborhood** of a solution should **not** be **too large**, since we want to be able to efficiently search the set of neighbors for possible local moves.

If the **first** of these points were the only concern, then it would seem that we should simply make all solutions neighbors of one another (after all, then there would be no local optima, and the global optimum would always be just one step away!). The **second point** exposes the (obvious) problem with doing this: If the **neighborhood** of the **current solution** consists of **every possible solution**, then the **local search** paradigm gives us **no leverage** whatsoever; it reduces simply to **brute-force search** of this neighborhood.

4.5.1 Local search algorithms for Graph Partitioning

In Section 4.4.2, we considered a state-flipping algorithm for the Maximum-Cut Problem, and we showed that the locally optimal solutions provide a 2-approximation. We now consider **neighbor relations** that produce **larger neighborhoods** than the single-flip rule, and consequently attempt to **reduce** the prevalence of **local optima**. Perhaps the most natural generalization is the k-flip neighborhood, for $k \geq 1$.

Definition (k-flip neighborhood). *We say that partitions (A, B) and (A', B') are neighbors under the k-flip rule if (A', B') can be obtained from (A, B) by moving at **most** k nodes from one side of the partition to the other.*

Now, clearly if (A, B) and (A', B') are neighbors under the k-flip rule, then they are also neighbors under the k'-flip rule for every $k' > k$. Thus, if (A, B) is a local optimum under the k'-flip rule, it is also a local optimum under the k-flip rule for every $k < k'$. But **reducing** the **set of local optima** by raising the value of k comes at a steep **computational price**: to examine the set of neighbors of (A, B) under the k-flip rule, we must consider all $\Theta(n^k)$ ways of moving up to k nodes to the opposite side of the partition. This becomes prohibitive even for **small values** of k.

Kernighan and Lin (1970) proposed an alternate **method** for **generating neighboring solutions**; it is **computationally** much more **efficient**, but still allows **large-scale transformations** of solutions in a single step. Their method, which we'll call the **K-L heuristic**, defines the neighbors of a partition (A, B) according to a n-phase procedure.

4.6 Nash Equilibrium

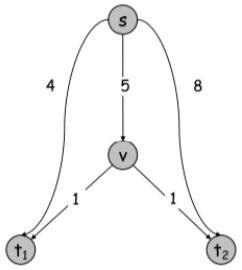
Thus far we have been considering local search as a technique for solving optimization problems with a single objective, in other words, applying local operations to a candidate solution so as to minimize its total cost. There are many settings, however, where a potentially large number of agents, each with its own goals and objectives, collectively interact so as to produce a solution to some problem. A solution that is produced under these circumstances often reflects the "tug-of-war" that led to it, with each agent trying to pull the solution in a direction that is favorable to it. We will see that these interactions can be viewed as a kind of local search procedure; analogues of local minima have a natural meaning as well, but having multiple agents and multiple objectives introduces new challenges.
The field of **game theory** provides a natural framework in which to talk about what happens in such situations, when a collection of agents interacts strategically—in other words, with each trying to optimize an individual objective function. To illustrate these issues, we consider a concrete application, motivated by the problem of routing in networks; along the way, we will introduce some notions that occupy central positions in the area of game theory more generally.

4.6.1 The problem

In a network like the Internet, one frequently encounters situations in which a number of nodes all want to establish a connection to a single source node s. For example, the source s may be generating some kind of data stream that all the given nodes want to receive, as in a style of one-to-many network communication known as **multicast**. We will model this situation by representing the underlying network as a **directed graph** $G = (V, E)$, with a **cost** $c_e \geq 0$ on each edge. There is a designated **source node** $s \in V$ and a collection of k **agents** located at distinct **terminal nodes** $t_1, t_2, .., t_k \in V$. For simplicity, we will not make a distinction between the agents and the nodes at which they reside; in other words, we will think of the agents as being $t_1, t_2, .., t_k$. Each agent t_j wants to construct a **path** P_j from s to t_j using as **little total cost** as possible.
Now, if there were **no interaction** among the agents, this would consist of k separate shortest-path problems: Each agent t_j would find an $s - t_j$ path for which the total cost of all edges is minimized, and use this as its path P_j. What makes this problem interesting is the prospect of agents being able to **share** the **costs** of edges. Suppose that after all the agents have chosen their paths, agent t_j only needs to pay its "fair share" of the cost of each edge e on its path; that is, rather than paying c_e for each e on P_i, it pays c_e divided by the number of agents whose paths contain e. In this way, there is an **incentive** for the agents to **choose paths that overlap**, since they can then benefit by splitting the costs of edges.

Example. *In the example, suppose the two agents start out using their outer paths. Then* t_1 *sees no advantage in switching paths (since* $4 < 5 + 1$*), but* t_2 *does (since* $8 > 5 + 1$*), and so* t_2 *updates its path by moving to the middle. Once this happens, things have changed from the perspective of* t_1*: There is suddenly an advantage for* t_1 *in switching as well, since it now gets to share the cost of the middle path, and hence its cost to use the middle path becomes* $2.5 + 1 < 4$*. Thus it will switch to the middle path. Once we are in a situation where both sides are using the middle path, neither has an incentive to switch, and so this is a stable solution.*

1	2	1 pays	2 pays
outer	outer	4	8
outer	middle	4	5 + 1
middle	outer	5 + 1	8
middle	middle	5/2 + 1	5/2 + 1

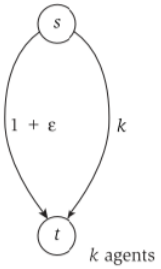


Let's discuss two **definitions** from the area of game theory that capture what's going on in this simple example. While we will continue to focus on our particular multicast routing problem, these definitions are relevant to any setting in which multiple agents, each with an individual objective, interact to produce a collective solution.

Definition (Best response dynamics). *Each **agent** is continually **prepared** to **improve** its **solution** in response to changes made by the other agent(s). In other words, we are interested in the dynamic behavior of a process in which each agent updates based on its best response to the current situation.*

Definition (Nash equilibrium). *A Nash equilibrium is a solution where **no agent** has an **incentive to switch**, i.e. it is a **stable solution**.*

Example. *This examples illustrates the possibility of multiple Nash equilibria. In this example,*



there are k agents that all reside at a common node t (that is, $t_1 = t_2 = .. = t_k = t$), and there are two parallel edges from s to t with different costs. The solution in which all agents use the left-hand edge is a Nash equilibrium in which all agents pay $(1 + \epsilon)/k$. The solution in which all agents use the right-hand edge is also a Nash equilibrium, though here the agents each pay $k/k = 1$.

The fact that this latter solution is a Nash equilibrium exposes an important point about best-response dynamics. If the agents could somehow synchronously agree to move from the right-hand edge to the left-hand one, they'd all be better off. But under best-response dynamics, each agent is only evaluating the consequences of a unilateral move by itself. In effect, an agent isn't able to make any assumptions about future actions of other agents, and so it is only willing to perform updates that lead to an immediate improvement for itself.

Definition (Social optimum). *A solution is a social optimum if it minimizes the total cost to all agents.*

Note that in both the previous examples there is a social optimum that is also a Nash equilibrium,

although in the second example there is also a second Nash equilibrium whose cost is much greater.

4.6.2 The relationship to Local Search

A set of agents following best-response dynamics are engaged in some kind of gradient descent process, exploring the "landscape" of possible solutions as they try to minimize their individual costs. The **Nash equilibria** are the natural analogues of **local minima** in this process: solutions from which no improving move is possible. And the "local" nature of the search is clear as well, since agents are only updating their solutions when it leads to an immediate improvement.
Having said all this, it's important to think a bit further and notice the crucial ways in which this **differs** from standard local search. In the beginning of this chapter, it was easy to argue that the **gradient descent** algorithm for a combinatorial problem must terminate at a **local minimum**: each update decreased the cost of the solution, and since there were only finitely many possible solutions, the sequence of updates could not go on forever. In other words, the cost function itself provided the progress measure we needed to establish termination.
In best-response dynamics, on the other hand, each agent has its own personal objective function to minimize, and so it's not clear what overall "progress" is being made when, for example, agent t_i decides to update its path from s. There's progress for t_i, of course, since its cost goes down, but this may be offset by an even larger increase in the cost to some other agent.
Consider, for example, the network in Figure 4.6.2.

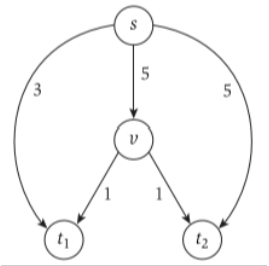


Figure 8: *A network in which the unique Nash equilibrium differs from the social equilibrium.*

If both agents start on the middle path, then t_1 will in fact have an incentive to move to the outer path; its cost drops from 3.5 to 3, but in the process the cost of t_2 increases from 3.5 to 6. (Once this happens, t_2 will also move to its outer path, and this solution (with both nodes on the outer paths) is the unique Nash equilibrium.)
There are examples, in fact, where the cost-increasing effects of best response dynamics can be much worse than this. Consider the situation in Figure 4.6.2, where we have k agents that each have the option to take a common outer path of cost $1 + \epsilon$ (for some small number $\epsilon > 0$), or to take their own alternate path. The alternate path for t_j has cost $1/j$. Now suppose we start with a solution in which all agents are sharing the outer path. Each agent pays $(1 + \epsilon)/k$, and this is the solution that minimizes the total cost to all agents. But running best-response dynamics starting from this solution causes things to unwind rapidly. First t_k switches to its alternate path, since $1/k < (1+\epsilon)/k$. As a result of this, there are now only $k-1$ agents sharing the outer path, and so t_{k-1} switches to its alternate path, since $1/(k-1) < (1+\epsilon)/(k-1)$. After this, t_{k-2} switches, then t_{k-3}, and so forth, until all k agents are using the alternate paths directly from s.
The total cost to all agents under even the most favorable Nash equilibrium solution can be worse than the total cost under the social optimum. How much worse? The total cost of the social optimum in this example is $1 + \epsilon$, while the cost of the unique Nash equilibrium is

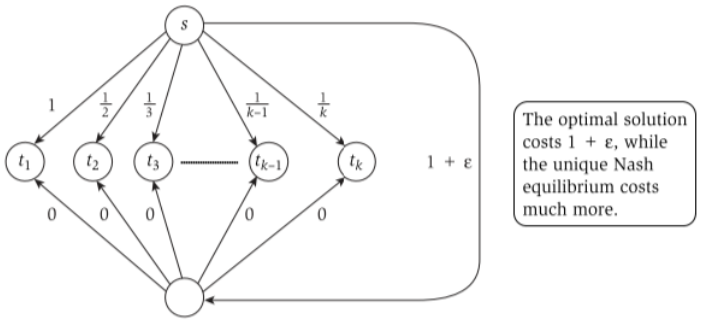


Figure 9: *A network in which the unique Nash equilibrium costs $H(k) = \Theta(\log k)$ times more than the social optimum.*

$1 + \frac{1}{2} + \frac{1}{3} + .. = \sum_{i=1}^{k} \frac{1}{i}$. This quantity is the harmonic number $H(k)$ its asymptotic value is $H(k) = \Theta(\log k)$.
These examples suggest that one can't really view the social optimum as the analogue of the global minimum in a traditional local search procedure. In standard local search, the global minimum is always a stable solution, since no improvement is possible. Here the social optimum can be an unstable solution, since it just requires one agent to have an interest in deviating.

4.6.3 Two basic questions

- The existence of a Nash equilibrium. At this point, we actually don't have a proof that there even exists a Nash equilibrium solution in every instance of our multicast routing problem. The most natural candidate for a progress measure, the total cost to all agents, does not necessarily decrease when a single agent updates its path;
- The price of stability. So far we've mainly considered Nash equilibria in the role of "observers": essentially, we turn the agents loose on the graph from an arbitrary starting point and watch what they do. But if we were viewing this as protocol designers, trying to define a procedure by which agents could construct paths from s, we might want to pursue the following approach.

  Given a set of agents, located at nodes $t_1, t_2, .., t_k$, we could propose a collection of paths, one for each agent, with two properties:

  (i) The set of paths forms a Nash equilibrium solution; and

  (ii) Subject to (i), the total cost to all agents is as small as possible.

  Of course, ideally we'd like just to have the smallest total cost, as this is the social optimum. But if we propose the social optimum and it's not a Nash equilibrium, then it won't be stable: Agents will begin deviating and constructing new paths. Thus properties (i) and (ii) together represent our protocol's attempt to optimize in the face of stability, finding the best solution from which no agent will want to deviate.

  Definition (Price of stability). *The price of stability is the ratio of the cost of the best Nash equilibrium solution to the cost of the social optimum. This quantity reflects the blow-up in cost that we incur due to the requirement that our solution must be stable in the face of the agents' self-interest.*

4.6.4 Finding a good Nash equilibrium

The following algorithm terminates with a Nash equilibrium.

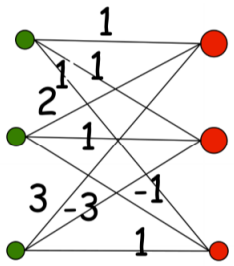
```
Best-Response-Dynamics(G, c) {
   Pick a path for each agent

   while (not a Nash equilibrium) {
      Pick an agent i who can improve by switching paths
      Switch path of agent i
   }
}
```

4.6.5 Bounding the price of stability

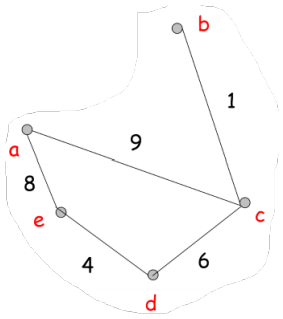
Theorem. *There is a Nash equilibrium for which the total cost to all agents exceeds that of the social optimum by at most a factor of $H(k)$, meaning that the Nash equilibrium is a quite good solution.*

4.7 Exercises

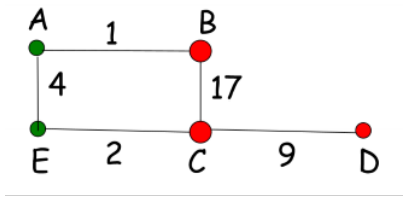
1. Execute the State-flipping algorithm on the following graph;



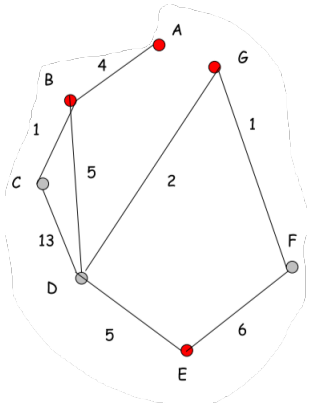
2. Execute the Maximum-Cut algorithm on the following graph. Solution on slide 25 of L7;



3. Execute the Maximum-Cut algorithm on the following graph;

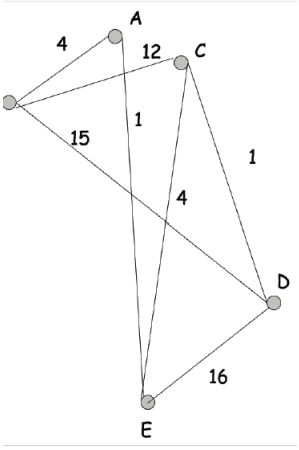


4. Given this Hopfield neural network, find a stable configuration.

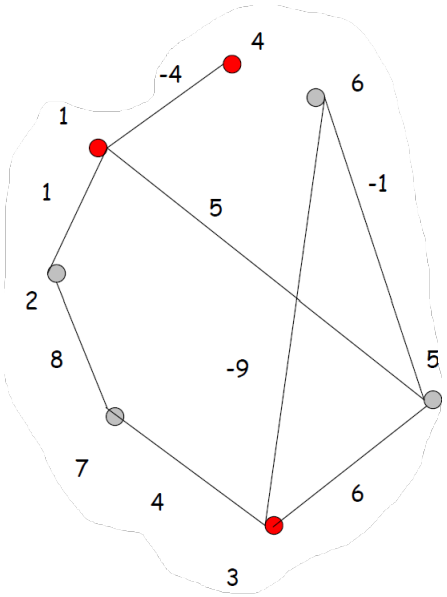


   What happens if all values are positive? Does this remind you of any known problem?

5. Execute the Maximum-Cut algorithm on the following graph, and find a stable configuration. Solution on slide 19-21 of L11;



6. What is a Hopfield Neural Network? When do we have a stable configuration? Describe the simple algorithm seen in class, that finds a stable configuration (if such a configuration exists). First describe the algorithm (in code, or pseudocode or words) in a complete and precise way, then show how the algorithm works on this example. Solution on slide 15-18 of L11.5

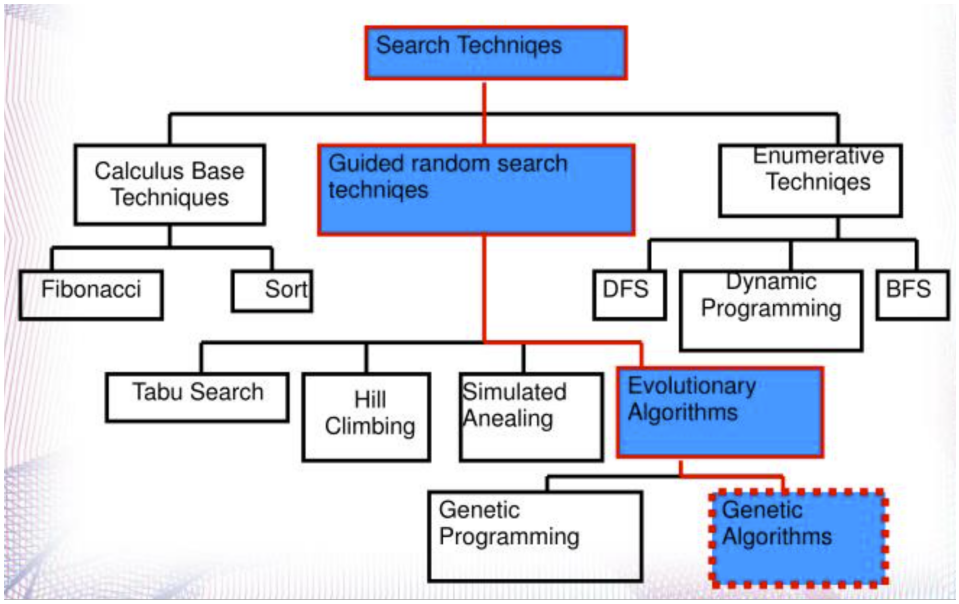


7. Describe the local search technique in general and very briefly. Solution on slide 33 of L10;

5 Genetic algorithms

5.1 Introduction

Genetic algorithms, originally developed by John Holland (1975), represent another class of iterative improvement algorithms, and they are part of the family of Evolutionary Algorithms (EA). They provide efficient and effective techniques for optimization and machine learning applications.



The idea is the following: A genetic algorithm maintains a population of candidate solutions for the problem at hand, and makes it evolve by iteratively applying a set of stochastic operators. We can see that this category of algorithms is inspired by the biological evolution process, and uses concepts of "Natural Selection" and "Genetic Inheritance" (Darwin 1859).

Structure

1. Randomly generate an initial population;
2. Evaluate the fitness population;
3. Select parents and “reproduce” the next generation;
4. Replace the old generation with the new generation;
5. Repeat step 2 though 4 till iteration N.

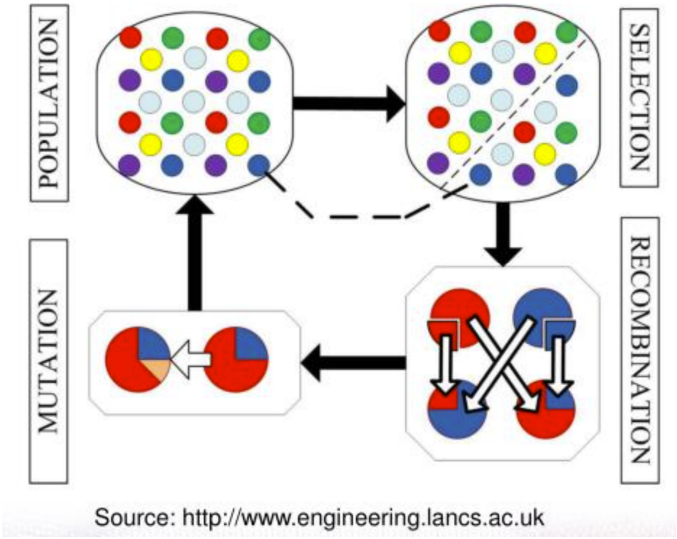
Population The population might be represented by bit strings (0101..1100), real numbers, permutations of elements, lists of rules, program elements or any data structure.

Evaluation The evaluator decodes a chromosome (part of the population) and assigns it a fitness measure: the evaluator is the only link between a classical GA and the problem it is solving.

5.1.1 Genetic algorithms

The idea of genetic algorithms is to start with a **population** of candidate solutions, and then evolve it by applying the so-called **stochastic operators**:

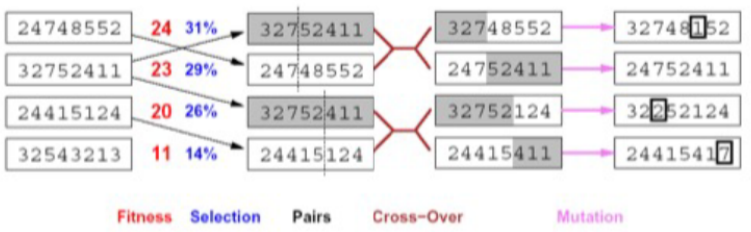
- **Selection**: replicates most successful solutions at a rate proportional to their relative quality (according to the fitness function);
- **Recombination** (cross-over): decomposes two solutions and randomly recompose them to form new solutions;
- **Mutation**: randomly perturbs a candidate solution.



Moreover, we can classify two types of genetic algorithms:

- *Generational GA*, where the entire population is replaced in each iteration;
- *Steady-state GA*, where a few members replaced in each generation.

Example.



There exist a lot of variants of genetic algorithms with different selection, crossover, and mutation rules, and in general GA have a wide application in optimization (e.g., circuit layout and job shop scheduling). Much work remains to be done to formally understand GA's and to identify the conditions under which they perform well.

5.1.2 When to use

- Alternate solutions are too slow or overly complicated;
- Need an exploratory tool to examine new approaches;
- Problem is similar to one that has already been successfully solved by using a GA;
- Want to hybridize with an existing solution.

5.2 The Maxone problem

In this problem we want to maximize the number of ones in a binary string of l (here we assume $l = 10$) bits. In this sense, an individual is encoded as a string of 10 bits, e.g. 0000000001. Clearly, the fitness f of a candidate to the Maxone problem is the number of ones in its genetic code. We start with a population of n random strings (we assume that $l = 10$ and $n = 6$).

5.2.1 Initialization

Suppose we toss a fair coin 60 times, and we get the following initial population:

$$\begin{aligned} s_1 &= 1111010101 \quad f(s_1) = 7 \\ s_2 &= 0111000101 \quad f(s_2) = 5 \\ s_3 &= 1110110101 \quad f(s_3) = 7 \\ s_4 &= 0100010011 \quad f(s_4) = 4 \\ s_5 &= 1110111101 \quad f(s_5) = 8 \\ s_6 &= 0100110000 \quad f(s_6) = 3 \end{aligned}$$

5.2.2 Selection

Next, we apply fitness proportionate selection with the *roulette wheel method*: according to this method individual i will have a probability

$$\frac{f(i)}{\sum_i f(i)}$$

to be chosen. As we can see, this probability is directly proportional to its fitness: the biggest the fitness value, the bigger the probability of an element of the population to be chosen.
We repeat the extraction as many times as the number of individuals we need to have the same parent population size (6 in our case).

5.2.3 Crossover

Next, we mate strings for crossover: for each couple we decide according to crossover probability (for instance 0.6) whether to actually perform crossover or not. Suppose that we decide to actually perform crossover only for couples (s'_1, s'_2) and (s'_5, s'_6): for each couple, we randomly extract a crossover point, for instance 2 for the first and 5 for the second.
Before crossover:

$$\begin{aligned} s'_1 &= 11111010101 \quad s'_5 = 0100010011 \\ s'_2 &= 1110110101 \quad s'_6 = 1110111101 \end{aligned}$$

After crossover:

$$s'_1 = 1110110101 \quad s'_5 = 0100011101$$
$$s'_2 = 1111010101 \quad s'_6 = 1110110011$$

5.2.4 Mutation

The final step is to apply random mutation: for each bit that we are to copy to the new population we allow a small probability of error (for instance 0.1).
Before applying mutation:

$$s''_1 = 1110110101 \quad s''_4 = 0100010011$$
$$s''_2 = 1111010101 \quad s''_5 = 0100011101$$
$$s''_3 = 1110111111 \quad s''_6 = 1110110011$$

After applying mutation:

$$s'''_1 = 1110100101 \quad f(s'''_1) = 6$$
$$s'''_2 = 1111110100 \quad f(s'''_2) = 7$$
$$s'''_3 = 1110111111 \quad f(s'''_3) = 9$$
$$s'''_4 = 0100010011 \quad f(s'''_4) = 4$$
$$s'''_5 = 0100011101 \quad f(s'''_5) = 5$$
$$s'''_6 = 1110110001 \quad f(s'''_6) = 6$$

In one generation, the total population fitness changed from 34 to 37, thus improved by $\sim 9\%$. At this point, we go through the same process all over again, until a stopping criterion is met.

5.3 Traveling Salesman Problem

In this case, given a complete graph, the goal is to find a Hamiltonian tour of a given set of cities so that each city is visited only once and the total distance traveled is minimized. In this case the encoding is the following:

- Label the cities $1, 2, .., n$;
- One complete tour is one permutation (e.g., for $n = 4$, $[1, 2, 3, 4], [3, 4, 2, 1]$ are OK)

As we can see the search space is huge: How many solutions do you have for 30 cities? 30!, so in general, for n cities we have $n!$ solutions.

5.3.1 Selection

In this case the fitness f of a solution is the inverse cost of the corresponding tour (recall: $\min \text{cost} = \max \frac{1}{\text{cost}}$.)

5.3.2 Crossover

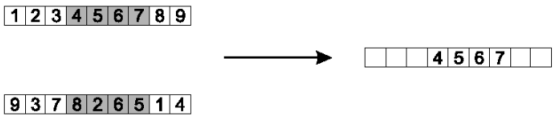
As for the mutation, the normal crossover operators will often lead to inadmissible solutions, so we need a different rule. The idea is to preserve the relative order in which the elements occur:

1. Choose an arbitrary part from the first parent;

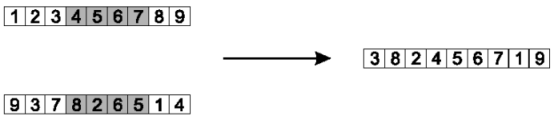
2. Copy this part to the first child;
3. Copy the numbers that are not in the first part, to the first child: starting right from cut point of the copied part, using the order of the second parent and wrapping around at the end;
4. Analogous for the second child, with parent roles reversed.

Example.

1. *Copy randomly selected set from the first parent;*



2. *Copy the rest* $(1, 2, 3, 8, 9)$ *using the order of the second parent* $(1, 9, 3, 8, 2)$ *starting from the cut point. As we can see, after the 7 we write 1, since after the cut point, in the second*



   parent we have a 1. Then, we write 9 since it is the second element we encounter when visiting the second parent from the beginning, and so on..
3. *Repeat for the other child.*

5.3.3 Mutation

In general, the mutation operation is quite tricky, since when mutating the population we must take into consideration the restrictions of the problem, i.e. of having an Hamiltonian tour that visits the cities only once. Thus, we have multiple choices.

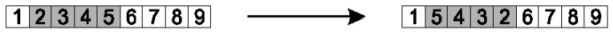
Insert mutation for permutations

1. Pick two allele values at random;
2. Move the second to follow the first, shifting the rest along to accommodate.

Note that this preserves most of the order and the adjacency information.

Swap mutation for permutations Pick two alleles at random and swap their positions.

Inversion mutation for permutations Pick two alleles at random and then invert the substring between them. This preserves most adjacency information.



Scramble mutation for permutations Pick a subset of genes at random and randomly rearrange the alleles in those positions.

The GA stops when the system has converged or a certain number of iterations have been performed.

6 Randomized algorithms

6.1 Introduction and motivations

Randomization and probabilistic analysis are themes that cut across many areas of computer science, including algorithm design, and when one thinks about random processes in the context of computation, it is usually in one of two distinct ways. One view is to consider the world as behaving randomly: One can consider traditional algorithms that confront randomly generated input. This approach is often termed average-case analysis, since we are studying the behavior of an algorithm on an "average" input (subject to some underlying random process), rather than a worst-case input.
A second view is to consider algorithms that behave randomly: The world provides the same worst-case input as always, but we allow our algorithm to make random decisions as it processes the input. Thus the role of randomization in this approach is purely internal to the algorithm and does not require new assumptions about the nature of the input. It is this notion of a randomized algorithm that we will be considering in this chapter.
Why might it be useful to design an algorithm that is allowed to make random decisions? A first answer would be to observe that by allowing randomization, we've made our underlying model more powerful. Efficient deterministic algorithms that always yield the correct answer are a special case of efficient randomized algorithms that only need to yield the correct answer with high probability; they are also a special case of randomized algorithms that are always correct, and run efficiently in expectation. Even in a worst case world, an algorithm that does its own "internal" randomization may be able to offset certain worst-case phenomena. So problems that may not have been solvable by efficient deterministic algorithms may still be amenable to randomized algorithms.
But this is not the whole story, and in fact we'll be looking at randomized algorithms for a number of problems where there exist comparably efficient deterministic algorithms. Even in such situations, a randomized approach often exhibits considerable power for further reasons: It may be conceptually much simpler; or it may allow the algorithm to function while maintaining very little internal state or memory of the past. The advantages of randomization seem to increase further as one considers larger computer systems and networks, with many loosely interacting processes—in other words, a distributed system. Here random behavior on the part of individual processes can reduce the amount of explicit communication or synchronization that is required; it is often valuable as a tool for symmetry-breaking among processes, reducing the danger of contention and "hot spots." A number of our examples will come from settings like this: regulating access to a shared resource, balancing load on multiple processors, or routing packets through a network. Even a small level of comfort with randomized heuristics can give one considerable leverage in thinking about large systems.

6.2 General features

The general features of randomized algorithms can be resumed as follows:

1. The same randomized algorithm may provide **different solutions** on the **same input**;
2. A randomized algorithm may provide a **wrong result**, but this should happen with a **small probability** ($<< 1$) for every instance of the problem;
3. By **increasing** the **number of times** in which we run the randomized algorithm, we **increase** the **confidence** of the result;
4. Usually, a randomized algorithm has a **better average case complexity** compared to a deterministic algorithm.

6.3 Examples

- **Numerical algorithms**: provide an approximate result with a certain confidence. By repeating the algorithms we increase the precision. E.g.: *With probability* 90% *the answer is* 20 ± 1;
- **Monte Carlo algorithms**: they provide a correct answer with very high probability. In some cases the answer is wrong. By repeating the algorithms we increase the probability of getting a correct answer. The execution time is deterministic. E.g.: *With probability* 99%, *the answer is 20*;
- **Las Vegas algorithms**: it always provides a correct answer or may not return a result. The execution time may vary from one run to another. E.g.: *The answer is 20.*

Example. *Consider the problem of finding an 'a' in an array of n elements. In this case the input is an array of* $n \geq 2$ *elements, in which half are 'a's and the other half are 'b's. Notice that the complexity of a standard iterative algorithm is* $O(\frac{n}{2})$.
Let's consider a Monte Carlo algorithm for this problem. As we can see, if an 'a' is found, the

```
findingA_MC(array A, n, k)
begin
    i=0
    repeat
        Randomly select one element out of n elements.
        i = i + 1
    until i=k or 'a' is found
end
```

algorithm succeeds, else the algorithm fails. After k iterations, the probability of finding an 'a' is

$$Pr(\textit{find an 'a'}) = 1 - \left(\frac{1}{2}\right)^k$$

The algorithm does not guarantee success, but the running time is fixed. It is executed an ***expected*** *number of* $1 \leq k < 2$ *times, therefore the running time is* $O(1)$.
We now consider a Las Vegas algorithm for the same problem. In this case, the algorithm suc-

```
findingA_LV(array A, n)
begin
    repeat
        Randomly select one element out of n elements.
    until 'a' is found
end
```

ceeds with probability 1 (I stop when I find an 'a'). The expected running time over many calls is $O(1)$. *Notice that the Las Vegas version is exactly the same of the Monte Carlo without the termination condition on the variable i.*

As a general rule:

- A Monte Carlo algorithm is guaranteed to run in poly-time, likely to find correct answer. Ex: Contraction algorithm for global min cut;
- A Las Vegas algorithm is guaranteed to find correct answer, likely to run in poly-time. Ex: Randomized quicksort, Johnson's MAX-3SAT algorithm.

Remark: we can always convert a Las Vegas algorithm into Monte Carlo, but no known method to convert the other way.

6.4 Content resolution problem

We begin with a first application of randomized algorithms: contention resolution in a distributed system. In particular, it is a chance to work through some basic manipulations involving events and their probabilities, analyzing intersections of events using independence as well as unions of events using a simple union bound.

6.4.1 The problem

Suppose we have n processes $P_1, P_2, .., P_n$, each competing for access to a single shared database. We imagine time as being divided into discrete rounds. The database has the property that it can be accessed by at most one process in a single round; if two or more processes attempt to access it simultaneously, then all processes are "locked out" for the duration of that round. So, while each process wants to access the database as often as possible, it's pointless for all of them to try accessing it in every round; then everyone will be perpetually locked out. What's needed is a way to divide up the rounds among the processes in an equitable fashion, so that all processes get through to the database on a regular basis.
If it is easy for the processes to communicate with one another, then one can imagine all sorts of direct means for resolving the contention. But suppose that the processes can't communicate with one another at all; how then can they work out a protocol under which they manage to "take turns" in accessing the database?

6.4.2 Randomized algorithm

Randomization provides a natural protocol for this problem, which we can specify simply as follows. For some number $p > 0$ that we'll determine shortly, each process will attempt to access the database in each round with probability p, independently of the decisions of the other processes. So, if exactly one process decides to make the attempt in a given round, it will succeed; if two or more try, then they will all be locked out; and if none try, then the round is in a sense "wasted".
This type of strategy, in which each of a set of identical processes randomizes its behavior, is the core of the symmetry-breaking paradigm that we mentioned initially: If all the processes operated in lockstep, repeatedly trying to access the database at the same time, there'd be no progress; but by randomizing, they "smooth out" the contention.

6.4.3 Analyzing the algorithm

Claim. *Let $S(i,t)$ denote the event that process P_i succeeds in accessing the database at a time t. Then*

$$\frac{1}{en} \leq Pr(S(i,t)) \leq \frac{1}{2n}$$

Proof. By independence, we have that

$$Pr(S(i,t)) = p(1-p)^{n-1}$$

where:

- p represents the probability that process P_i requests the access;
- $(1-p)^{n-1}$ represents the probability that the other $n-1$ processes do not request access.

If we set $p = \frac{1}{n}$, we have that $P(S(i,t)) = \frac{1}{n}\left(1-\frac{1}{n}\right)^{n-1}$. It's worth getting a sense for the asymptotic value of this expression, with the help of the following extremely useful fact from basic calculus: as n increases from 2,

- The function $\left(1-\frac{1}{n}\right)^n$ converges monotonically from $\frac{1}{4}$ up to $\frac{1}{e}$;
- The function $\left(1-\frac{1}{n}\right)^{n-1}$ converges monotonically from $\frac{1}{2}$ down to $\frac{1}{e}$;

Using these relations, we can see that

$$\frac{1}{en} \leq Pr(S(i,t)) \leq \frac{1}{2n} \qquad \blacksquare$$

Waiting for a particular process to succeed

Let's consider this protocol with the optimal value $p = 1/n$ for the access probability. Suppose we are interested in how long it will take process P_i to succeed in accessing the database at least once. We see from the earlier calculation that the probability of its succeeding in any one round is not very good, if n is reasonably large. How about if we consider multiple rounds?

Claim. *The probability that process P_i fails to access the database in en rounds is at most $\frac{1}{e}$. After $en(c\ln n)$ rounds, the probability is at most n^{-c}, i.e. the event happens with small probability.*

Proof. Let $F(i,t)$ denote the "failure event" that process P_i does not succeed in any of the rounds 1 through t. This is clearly just the intersection of the complementary events $S(i,r)$ for $r = 1, 2, .., t$. Moreover, since each of these events is independent, we can compute the probability of $F(i,t)$ by multiplication:

$$Pr(F(i,t)) \leq \left(1-\frac{1}{en}\right)^t$$

If we set $t = \lceil en \rceil$, we get:

$$Pr(F(i,t)) \leq \left(1-\frac{1}{en}\right)^{\lceil en \rceil} \leq \left(1-\frac{1}{en}\right)^{en} \leq \frac{1}{e}$$

This is a very compact and useful asymptotic statement: The probability that process P_i does not succeed in any of rounds 1 through $\lceil en \rceil$ is upper-bounded by the constant e^{-1}, independent of n.

Now, if we increase t by some fairly small factors, the probability that P_i does not succeed in any of rounds 1 through t drops precipitously: If we set $t = \lceil en \rceil (c \ln n)$, then we have:

$$Pr(F(i,t)) \leq \left(1-\frac{1}{en}\right)^{c\ln n} = n^{-c}$$

Waiting for all processes to get through

Finally, we're in a position to ask the question that was implicit in the overall setup: How many rounds must elapse before there's a high probability that all processes will have succeeded in accessing the database at least once?

To address this, we say that the protocol fails after t rounds if some process has not yet succeeded in accessing the database. Let $F(t)$ denote the event that the protocol fails after t rounds; the goal is to find a reasonably small value of t for which $Pr(F(t))$ is small.

Claim. *The probability that all the processes succeed within $2en\ln n$ rounds is at least $1-\frac{1}{n}$.*

The event $F(t)$ occurs if and only if one of the events $F(i,t)$ occurs; so we can write

$$Pr(F(t)) = Pr\left(\vee_{i=1}^{n} F(i,t)\right) \leq \sum_{i=1}^{n} Pr(F(i,t)) \leq n\left(1 - \frac{1}{en}\right)^t$$

Notice that:

- The first inequality comes from the union bound, which states that $Pr\left(\vee_{i=1}^{n} E_i\right) \leq \sum_{i=1}^{n} Pr(E_i)$;
- The second inequality comes from the fact that $Pr(F(i,t)) \leq (1 - \frac{1}{en})^t$.

6.5 Randomized quicksort

Divide and conquer often works well in conjunction with randomization, and we illustrate this by giving divide-and-conquer algorithms for two fundamental problems: computing the median of n numbers, and sorting. In each case, the "divide" step is performed using randomization; consequently, we will use expectations of random variables to analyze the time spent on recursive calls.

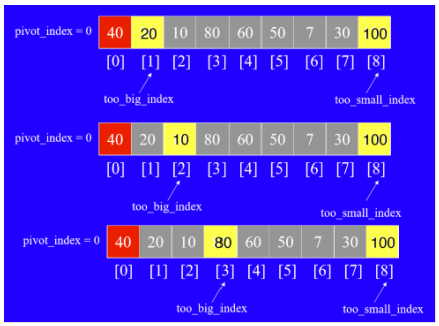
6.5.1 Standard quicksort

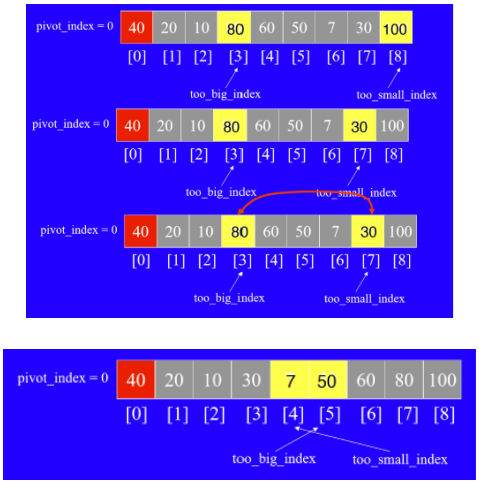
Before analyzing the randomized version, we first take into consideration the standard version of this sorting algorithm.
Given an array of n elements, the algorithm works as follows:

1. If the array only contains one element, return;
2. Otherwise:
   (a) Pick one element and use it as a *pivot*;
   (b) Partition the elements into two sub-arrays: elements smaller or equal to the pivot, elements bigger that the pivot;
   (c) Quicksort the two sub-arrays;
   (d) Return results.

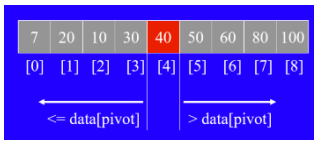
How do we partition the arrays at step (b)? The idea is to keep two pointers:

1. The first one points to the element right after the pivot;
2. The second one points to the last element of the array;
3. We start my moving forward the first pointer: when it finds a number which is $\geq$ pivot, then it stops;
4. Then, we move the second pointer backward: when it finds a number which is $<$ pivot, it stops;





5. We swap the two elements and we continue to move the pointers;
6. When the two pointers cross, we stop;



7. Finally, we swap the pivot with the element pointed by the second pointer: now, the pivot is the only element in the array that is in the right position. Then, we recursively apply the same algorithm to the two sub-arrays.

The worst-case complexity of the standard algorithm is $O(n^2)$, since if the array is in decreasing order, at each step we should compare all the elements.

6.5.2 Randomized quicksort

```
RandomizedQuicksort(S) {
   if |S| = 0 return

   choose a splitter a_i ∈ S uniformly at random
   foreach (a ∈ S) {
      if      (a < a_i) put a in S⁻
      else if (a > a_i) put a in S⁺
   }
   RandomizedQuicksort(S⁻)
   output a_i
   RandomizedQuicksort(S⁺)
}
```

Figure 10: *Randomized quicksort.*

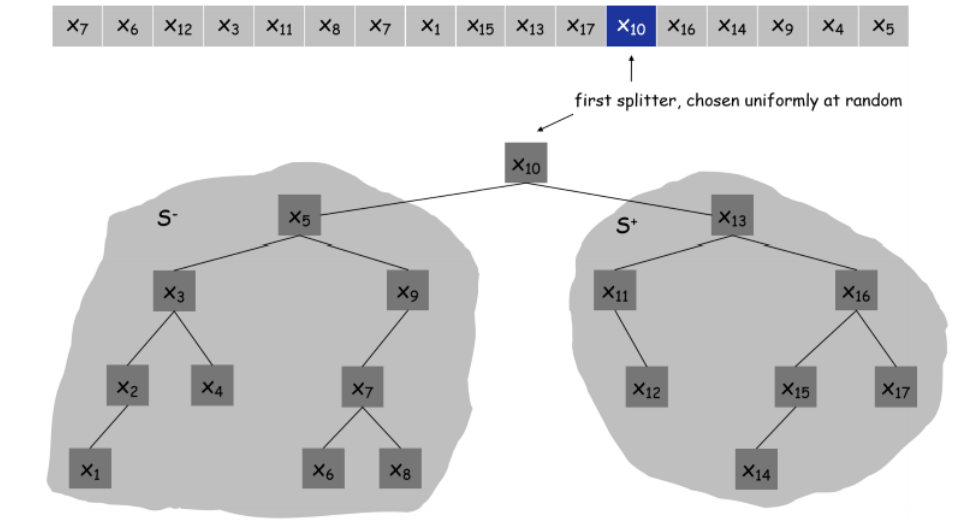
This algorithm is a Las Vegas algorithm.

Running time In the **best case** the quicksort selects the median element as the splitter, and it makes $\Theta(n \log n)$ comparisons.

In the **worst case** the quicksort selects the smallest or biggest element as the pivot, making $\Theta(n^2)$ comparisons.

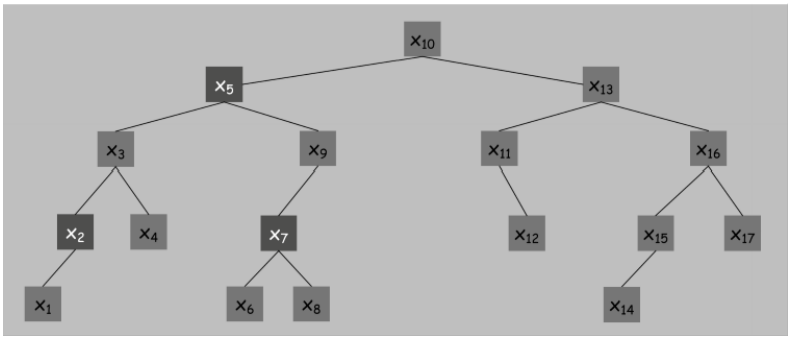
Randomization In this case randomization protects against the worst-case scenario, by choosing a split at random. Intuitively, if we always select an element that is bigger than 25% of the elements and smaller than 25% of the elements, then quicksort makes $\Theta(n \log n)$ comparisons.

BST representation We draw a recursive BST of the splitters.



The elements are only compared with its ancestor and descendants, and we call it a portion, defined as Z_{ij}. We assume $i = 2$, $j = 7$:

- x_2 and x_7 are compared if their $lca = x_2$ or x_7, i.e., one of them is a pivot;
- x_2 and x_7 are not compared if their $lca = x_3$ or x_4 or x_5 or x_6;
- Portion Z_{ij} contains $j - i + 1$ elements.



Claim. *$Pr(x_i$ and x_j are compared$) = \frac{2}{|j-i+1|}$*

This comes from the fact that the probability that i or j is the pivot is $\frac{1}{|j-i+1|}$, i.e. that x_i is in the path between x_j and the root and vice-versa (definiton of ancestor).

Theorem. *The expected number of comparisons is $O(n \log n)$.*

Proof.

$$\sum_{1\le i<j\le n} \frac{2}{j-i+1} = 2\sum_{i=1}^{n}\sum_{j=2}^{i}\frac{1}{j} \le 2n\sum_{j=1}^{n}\frac{1}{j} \approx 2n\int_{x=1}^{n}\frac{1}{x}dx = 2n\ln n$$

Example. *If $n = 1$ million, then the probability that randomized quicksort takes less than $4n \ln n$ comparisons is at least 99.94%.*

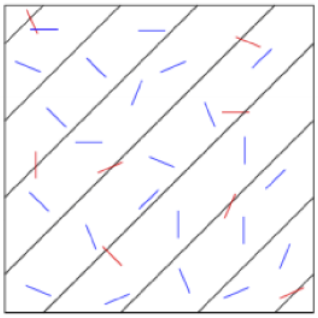
6.6 Numerical algorithms

As we said earlier in this chapter, numerical algorithms provide an approximate result with a certain confidence: with probability p the correct solution is $y \pm \epsilon$.

Example. *With probability 90% the answer is 20 ± 1. By repeating the algorithms we increase the precision.*

6.6.1 Buffon's needle

This example represents an 18th century approach to approximating π. The idea is to drop N needles of unit length (1) randomly (with uniform distribution) over a floor (width of floor boards is 2 units): some will fall across the boards (the red ones), some will not (the blue ones).



Theorem (Leclerc). *If we drop 1 needle of length l on the floor and the boards are at distance d, the probability that it will cross a board is $\frac{2l}{\pi d}$.*

Note that when $l = 1$ and the boards are at distance $d = 2$, we get $\frac{2l}{\pi d} = \frac{2}{2\pi} = \frac{1}{\pi}$. If we throw n needles, the number of the ones that cross the boards is $k = \frac{n}{\pi}$, i.e., the mean of a binomial process w.p. 1.
The algorithm designed by Buffon exploits the following: with a high number n of dropping of needles we can estimate π as the number of needles that cross the board $\pi \approx \frac{n}{k}$ (which is derived from the previous relationship).

Bernullian distribution This distribution best describes all situations where a "trial" is made resulting in either "success" or "failure," such as when tossing a coin. The Bernoulli distribution is a discrete distribution having two possible outcomes labelled by $n = 0$ and $n = 1$, and in which $n = 1$ ("success") occurs with probability p and $n = 0$ ("failure") occurs with probability $(1 - p)$, where $0 < p < 1$. It therefore has the following probability density function:

$$P(n) = \begin{cases} 1-p & \text{for } n = 0 \\ p & \text{for } n = 1 \end{cases}$$

The expected value of a Bernoulli random variable is p, while the variance is $p(1-p)$.

Binomial distribution The Bernoulli distribution is a special case of the binomial distribution with $n = 1$ trial. If X is a binomially distributed random variable, n being the total number of experiments and p the probability of each experiment yielding a successful result, then the expected value of X is np, while the variance is $np(1-p)$.

The algorithm Let X_i be the bernoullian variable that describes the i-th needle: it takes value 1 if the needle crosses the board, 0 otherwise.
For Leclerc's algorithm with $l = 1$ and $d = 2$ we get, $Pr(X_i = 1) = \frac{1}{\pi}$ for each i. According to the bernoullian distribution, $p = \frac{1}{\pi}$, and the variance $p(1-p) = (\frac{1}{\pi})(1-\frac{1}{\pi})$.
Let X be the binomial variable that describes the mean (not the sum) of n drops (k successes in n trials), then:

$$X = \frac{\sum_{i=1}^{n} X_i}{n}$$

so we have that:

$$Pr\left(X = \frac{k}{n}\right) = \binom{n}{k}\left(\frac{1}{\pi}\right)^k\left(1-\frac{1}{\pi}\right)^{n-k}$$

By doing some computations on the variable X, we get:

$$Pr\left(\left|X - \frac{1}{\pi}\right| < \epsilon\right) \geq 99\% \qquad \text{for } n > \frac{1,440}{\epsilon^2}$$

By doing different computations we get an estimate of π with an absolute error $\epsilon < 0,0415$ with 99% of confidence, and we need to repeat the needle drops for $n \geq 1440/\epsilon^2$ times.

6.7 Exercises

1. Given as input an array of n numbers and an integer k between 1 and n, the goal is to provide as output the k-th smallest number in the array:
   (a) Devise a simple algorithm with complexity $O(n \log n)$;
   (b) Devise a randomized algorithm with expected running time $O(n)$, then compute the complexity in the best and in the worst-case.

2. What is the difference between a Las Vegas and a Monte Carlo algorithm? In which context are they used?

   You are given an array A of length n (where n is a multiple of 3), where the elements are 1/3 number 1, 1/3 number 2 and 1/3 number 3; e.g. $A = [1, 2, 1, 1, 3, 2, 3, 3, 2]$ and $n = 9$.
   (a) Define a randomized Monte Carlo algorithm MC_1 that finds a number 2 in the array;
   (b) What is the probability of finding a 2 after k iterations of MC_1?
   (c) Solve the same problem using a Las Vegas algorithm LV_2;
   (d) What is the probability that LV_2 succeeds?