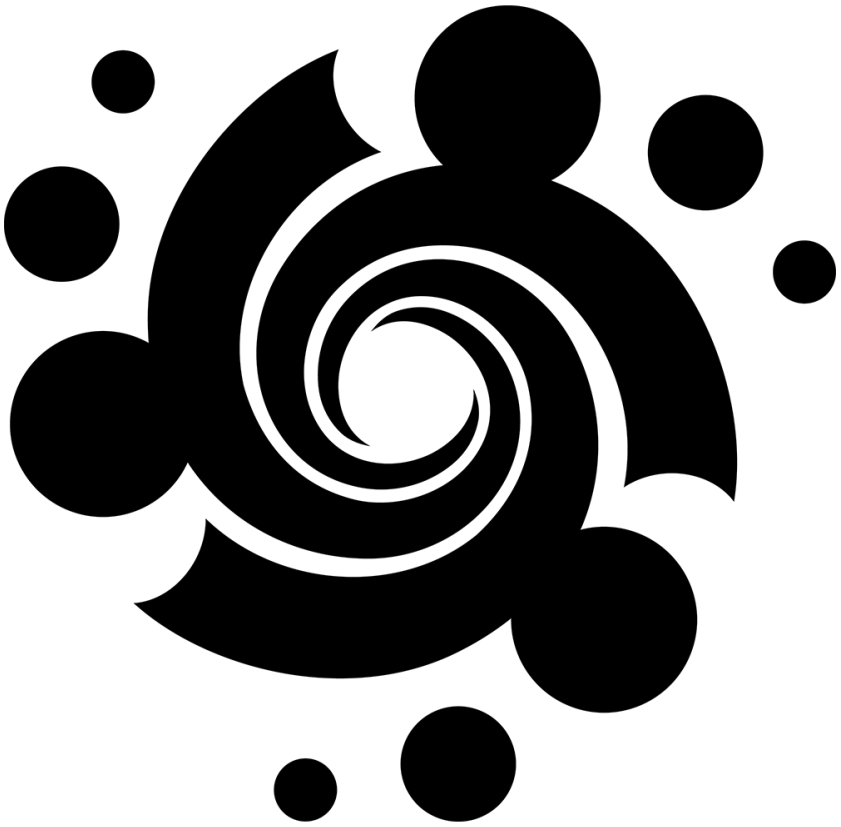
ETH Zurich

Department of Computer Science

Algorithms and Probability PVK Skript

Author:
Soel Micheletti



Preface

This script is a summary of the ETH Course *Algorithmen und Wahrscheinlichkeit*. Some topics are more presented more in depth than others, but I hope you'll get a good overview about all the important concepts taught in the course. If you find any error or if you have any suggestion, don't hesitate to contact me at *msoel@ethz.ch*: I'm happy to hear your feedbacks!

Contents

Graph Theory 1

In the course *Algorithms and Data Structures* last semester, you had a first encounter with graphs: you have seen the definition, some property and algorithms such as BFS, DFS, Topological Sorting, algorithms for shortest paths... In this course we go deeper in the topic and we introduce other exciting concepts. Some of them are exposed in this chapter, others (the ones that exploit randomization) are presented in Chapter 3.

1.1 Recap from Algorithms and Data Structures

Concepts and Notation

Graphs are a powerful mathematical tool that allows us to model many different problems. The main idea is to have a collection of elements (*nodes*), and the *relationship* between them (*edges*). An example could be to model locations as nodes and the streets between them as edges, or persons and the relationship between them. Both nodes and edges can be augmented to contain more information such as color, location, length, capacity, flow and others to describe them.

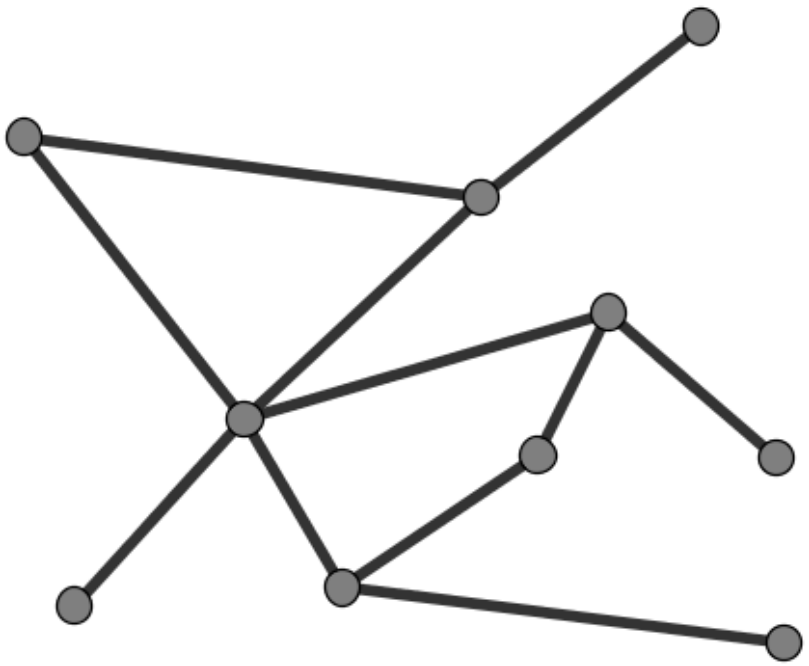


Figure 1.1: Simple undirected graph

Definition 1.1.1 (Graph) *A graph is a tuple* $G = (V, E)$ *with*
$V = \{v_1, ..., v_n\}, |V| = n$ *set of nodes*
$E = \{e_1, ..., e_m\}, |E| = m$ *set of edges*

If the underlying relationship is symmetric, meaning that v is connected to u only if also u is connected to v, then the graph is said to be *undirected*.

$E \subseteq \{\{u, v\} | u, v \in V\}$ undirected edges
$e_k = \{v_i, v_j\}$
v_i, v_j adjacent if $\{v_i, v_j\} \in E$
v_i, e_k incident if $v_i \in e_k$

If however there is the possiblity of having v in relation to u without u being in relation to v (e.g. father-of relationship) then the graph is *directed*. The direction is represented by an arrow.

$E \subseteq V \times V$ directed edges
$e_k = (v_i, v_j)$

Bipartite Graphs

If our graph contains two disjoint sets of nodes who aren't allowed to be in relationship within themselves (e.g. $\nexists e = \{v_1, v_2\} | v_1, v_2 \in U \vee v_1, v_2 \in W$) then the graph is said to be *bipartite*.

$V = U \cup W$ two disjoint sets of nodes ($U \cap W = \varnothing$)
$E \subseteq \{\{u, w\} | u \in U, w \in W\}$ (undirected bipartite graph)
$E \subseteq (U \times W) \cup (W \times U)$ (directed bipartite graph)

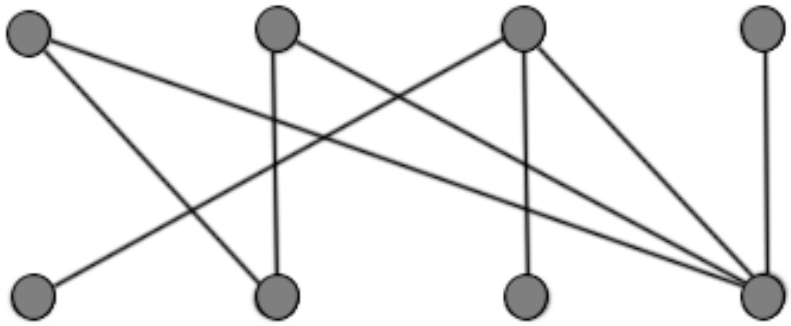


Figure 1.2: Simple bipartite graph

Sequences

We can select an ordered sequence of vertices, and based on edges between them and repetition of edges/vertices in our sequence, we can distinguish them as follows:
let's consider the *sequence* $\langle v_1, v_2, ..., v_k \rangle$

- *Weg* (walk): a sequence with edges between v_i and $v_i + 1$ for all $i \in \{1..k-1\}$. The length is $k - 1$.
- *Pfad* (path): a Weg where all vertices are distinct.
- *Reise* (tour/trail): a Weg where all edges (but not necessarily vertices) are distinct.
- *Zyklus* (cycle): a Weg with $v_1 = v_k$ (starts and ends at the same vertex)
- *Schleife* (loop): a Weg $\langle v_i, v_i \rangle$ with length 1 (special case of Zyklus).
- *Kreis* (Circle): a Zyklus with the property $\forall i, j \in \{1, .., k-1\}, i \neq j \cdot v_i \neq v_j$ (no vertex is visited more than once). Same as a Pfad with same start and end.

- *Rundreise* (circuit): a Reise with $v_1 = v_k$

Here it's a handy table that summarize these concepts:

	all	closed
all	Weg	Zyklus
distinct V	Pfad	Kreis
distinct E	Reise	Rundreise

Degree

The degree (or valence) of a vertex v of a graph is the number of edges incident to the vertex, with loops counted twice. It is denoted $deg(v) = |N_G(v)|$, where $N_G(v)$ is the neighborhood of v.
For directed graph, we have the *in-degree* $\deg_G^-(v) = |N_G^-(v)|$ and the *out-degree* $\deg_G^+(v) = |N_G^+(v)|$.

Theorem 1.1.1

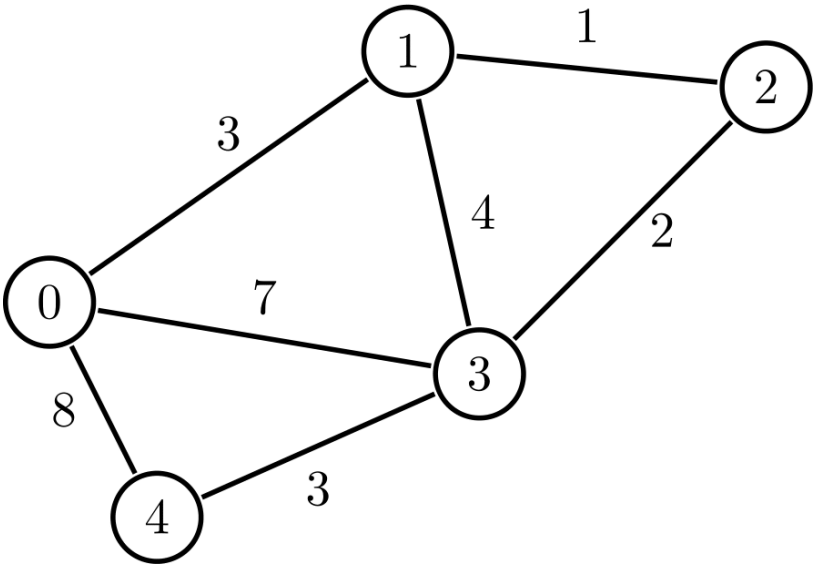
$$\sum_{v \in V} deg(v) = 2|E|$$

This formula implies the following:

- the amount of vertices with odd degree is even
- the average degree in a graph is $\frac{2m}{n}$

Graph Data Structures

Until now a graph $G = (V, E)$ is an abstract data type which can be represented as follows:



But how can we represent a graph in a computer? How can you code algorithms that works with graphs? In order to do this you need a *data structure* which represent the graph. Here there are the three most popular solutions.

Adjacency matrix We simply do a matrix with $|V|$ rows and $|V|$ columns. The entry (i, j) has value 0 if there are no edges between nodes i and j and has value w if there is an edge between nodes i and jJ with weight w (in the case of unweighted graphs all edges have weight 1). In the case of the previous image we would have the following adjacency matrix:

$$\begin{bmatrix} 0 & 3 & 0 & 7 & 8 \\ 3 & 0 & 1 & 4 & 0 \\ 0 & 1 & 0 & 2 & 0 \\ 7 & 4 & 2 & 0 & 3 \\ 8 & 0 & 0 & 3 & 0 \end{bmatrix}$$

Note that if the graph is undirected the matrix is symmetric, i.e. $A^T = A$. Now we take a closer look of the asymptotic efficiency of some common operations on adjacency matrices.

- **Memory space:** $\mathcal{O}(|V|^2)$
- **Add vertex:** $\mathcal{O}(|V|^2)$
- **Remove vertex:** $\mathcal{O}(|V|^2)$
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(1)$
- **Are u and v adjacent?** $\mathcal{O}(1)$
- **Given a node v, find $deg(v)$:** $\mathcal{O}(|V|)$
- **Given a node v, give an arbitrary neighbour of v:** $\mathcal{O}(|V|)$
- **Given a node v, find all incidents edges to v:** $\mathcal{O}(|V|)$

Adjacency list We do a list of lists. The list has a list for every node. The list of node u contains all v and w such that $(u, v, w) \in E$. In the case of the previous image we would have the following adjacency list:

$\{[(0,1,3),(0,3,7),(0,4,8)],[(1,0,3),(1,3,4),(1,2,1)],[(2,1,1),(2,2,3)],$
$[(3,2,2),(3,1,4),(3,0,7),(3,3,4)],[(4,0,8),(4,3,3)]\}$

Let's take a closer look on the asymptotic efficiency of some common operations on adjacency lists.

- **Memory space:** $\mathcal{O}(|V| + |E|)$
- **Add vertex:** $\mathcal{O}(1)$
- **Remove vertex:** $\mathcal{O}(|E|)$. You can do it in $\mathcal{O}(1)$ if you implement it in a clever way, think about it and if you don't find the solution you can ask me ;)
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(|V|)$
- **Are u and v adjacent?** $\mathcal{O}(\min(deg(u), deg(v)))$
- **Given a node v, find $deg(v)$:** $\mathcal{O}(deg(v))$
- **Given a node v, give an arbitrary neighbour of v:** $\mathcal{O}(1)$
- **Given a node v, find all incidents edges to v:** $\mathcal{O}(deg(v))$

Obviously, if you have a complete graph (i.e. every node is connected to all other nodes), the runtimes of many operations becomes the same of

the ones for adjacency matrices. The intuition behind adjacency matrices is that the efficiency of many operations is inversely proportional to the number of edges (which, you should remind, is $\leq |V|^2$ in the graphs you consider in this course): if the graph has zero edges the operations are very efficient, if the graph has many edges the operations have the same efficiency they would have in adjacency matrices. As you'll see during the semester, some algorithms on graphs have different complexity if we use adjacency matrices or adjacency lists (e.g. Dijkstra and Prim's algorithms).

List of edges We simply do a list of $|E|$ tuples (u, v, w), where u is the source node, v the destination node and w the weight of the edge. In the case of the previous image we would have the following list:

$$[(0,1,3),(0,3,7),(0,4,8),(1,0,3),(1,2,1),(1,3,4),(2,3,2),$$
$$(2,1,1),(3,2,2),(3,1,4),(3,0,7),(3,4,3),(4,3,3),(4,0,8)]$$

Note that if the graph is undirected we can either put directed edges in the list (as in the example) or undirected edges (i.e. instead of putting $(0,1,3)$ and $(1,0,3)$ we put only one of the two possibilities with the convention that we consider the graph undirected). Pay attention to the fact that the list does not have to be sorted in any way! Now we take a closer look of the asymptotic efficiency of some common operations on list of edges (in those cases the facts whether we put $|E|$ or $2|E|$ edges in the graph is irrelevant).

- **Memory space:** $\mathcal{O}(|E|)$
- **Add vertex:** $\mathcal{O}(1)$
- **Remove vertex:** $\mathcal{O}(|E|)$.
- **Add edge:** $\mathcal{O}(1)$
- **Remove edge:** $\mathcal{O}(|E|)$
- **Are u and v adjacent?** $\mathcal{O}(|E|)$
- **Given a node v, find $deg(v)$:** $\mathcal{O}(|E|)$
- **Given a node v, give an arbitrary neighbour of v:** $\mathcal{O}(|E|)$
- **Given a node v, find all incidents edges to v:** $\mathcal{O}(|E|)$

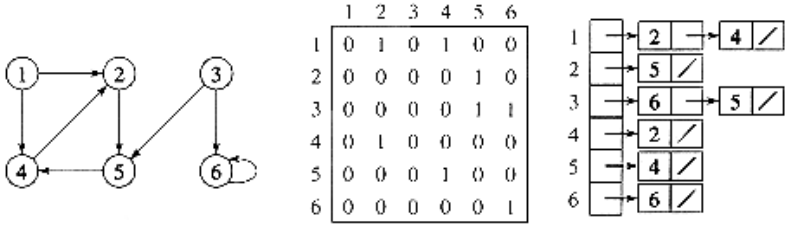


Figure 1.3: Different ways of storing a graph. Adj. Matrix and Adj. List

Trees

A tree T is a graph on n vertices that satisfies the following properties:

- it is connected
- it has no cycles
- it has $n - 1$ edges

Note that if T satisfies two of these properties, it automatically also satisfies the third.

Trees also have a few simple properties. We consider the general case of a tree T with $n \geq 2$.

- T contains at least two leaves.
- by removing a leaf from T, we obtain a graph G' which is also a tree.
- between any two nodes x, y there is exactly one path in T.

1.2 Minimum Spanning Tree

Given a graph $G = (V, E)$ and a cost function $c : E \to \mathbb{R}$ that assigns a cost $c(e)$ to each edge in G, a minimum spanning tree (MST) T over G is a tree with $|V|$ nodes, such that the sum

$$\sum_{e \in E(T)} c(e)$$

is minimized. This means that among all possible complete trees over G, T minimizes the total cost. The MST has many applications.

Cuts

A cut consists of a disjoint partition of the vertices V of G into $S \subset V$ with $S \neq \varnothing$ and $\overline{S} = V \backslash S$. The edges along this cut are all the edges $e = \{u, v\}$ such that $u \in S, v \in V \backslash S$.

Blue and Red rule

All algorithms that compute the MST of a graph either use the blue rule or the red rule, or a combination of the two. A brief overview of the rules is given.

Blue rule Given a cut $S, V \backslash S$ with no edges painted blue on this cut, paint blue the edge e with minimum cost $c(e)$.
Idea: the tree must be connected, therefore pick the cheapest edge that connects these two partitions.

Red Rule Given a cycle C with no edges painted red, paint red the edge e with maximum cost $c(e)$.
Idea: the tree contains no cycles, therefore we can safely discard the edge that costs the most.

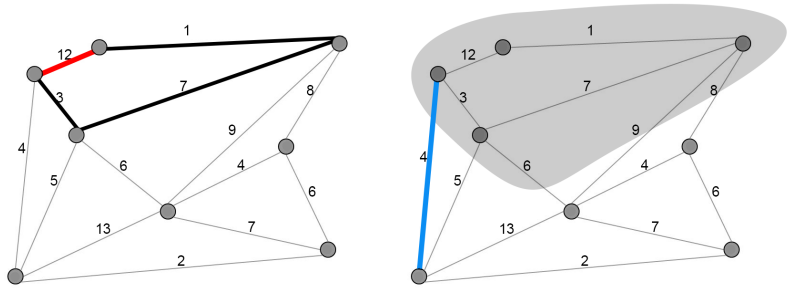


Figure 1.4: Red and Blue rules visualized

Prim's Algorithm

This is an algorithm that given as input a graph G and a starting node a returns the MST in time $O(|E|\log|V|)$ or even the better time $O(|E|+|V|\log|V|)$ if using advanced data structures such as Fibonacci heaps. In general the runtime is

$$\mathcal{O}(|V|(C(\textsc{Insert})+C(\textsc{Extract-Min}))+|E|C(\textsc{Decrease-Key}))$$

The idea is to store a set S of vertices already in the solution, and always add the cheapest edge along the $(S, V\backslash S)$ cut to extend the built part. Note: this is simply equal to applying the blue rule multiple times, always picking the cheapest edge. Although the most efficient way to implement the algorithm is with Fibonacci Heaps, Priority Queues are a good practical solution.

```
1 T ← ∅
2 S ← {a}
3 while S ≠ V
4     find e = (u, v) such that u ∈ S, v ∈ V\S and c(e) minimized along
       cut
5     T ← T ∪ {e}
6     S ← S ∪ {v}
```

Algorithm 1.1: Prim's Algorithm

Kruskal's Algorithm

This is also an algorithm that given a graph G computes the MST in time $O(|E|\log|V|)$. It works in a greedy fashion by considering edges in increasing order of weights, each time checking if the insertion in the tree would create a cycle. If it's not the case the edge is added. It uses a Union-Find DS to check this condition. The runtime is given by

$$\mathcal{O}(|V|(C(\textsc{Insert})+C(\textsc{Union}))+|E|C(\textsc{Find}))$$

```
1 T ← ∅
2 for all v ∈ V
3     makeset(v)
4 sort E by increasing cost c(e)
```

Algorithm 1.2: Kruskal's Algorithm

```
5 for e = (u, v) ∈ E in order
6     if findset(u) ≠ findset(v)
7     T ← T ∪ e
8     union(u, v)
9 return T
```

1.3 Advanced Graph Concepts

Definition 1.3.1 *A graph is* k-connected *iff*

- $|V| \geq k+1$
- $\forall X \subseteq V, |X| < k \cdot G[V \setminus X]$ *is connected*

Definition 1.3.2 *A graph* k-edges-connected *iff*

- $|E| \geq k+1$
- $\forall X \subseteq E, |X| < k \cdot G[E \setminus X]$ *is connected*

Informally this means that we need to remove at least k vertices/edges to disconnect the graph.

Theorem 1.3.1 *Menger stated that a graph is* k-connected $\iff$

$$\forall u, v \in V, u \neq v \cdot \exists k \text{ vertex-}\textit{disjoint } u - v \textit{ paths}$$

This means that since there are k paths that do not share vertices, we cannot disconnect the graph by removing any $k-1$ vertices, which is the definition of k-connectedness.
Similarly, for a k-edges-connected *the same holds but with k* edge-*disjoint $u - v$ paths.*

Single nodes and edges that disconnect the graph (therefore with $k = 1$) are called *Articulation points* and *Bridges* respectively. Another name for articulation points is *cut vertex*, and for bridges is *cut edge*.

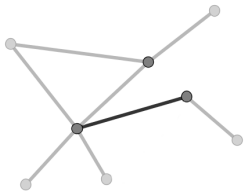


Figure 1.5: Bridge (1) and articulations (3) marked darker in the graph

How can we wind articulation points and bridges efficiently in a graph $G = (V, E)$?

- perform *DFS* on G, assigning a number $dfs[v]$ to each vertex (the visit order)
- compute $low[v]$ = min dfs number reachable from v via *forward*-edges (any number) and at most one *backward*-edge
- v is an articulation point $\iff$
  - $v = s$ and s has degree ≥ 2 in T, or
  - $v \neq s$ and $\exists w \in V$ with $\{v, w\} \in E(T)$ and $low[w] \geq dfs[v]$.

- $\{u, v\}$ is a bridge if either u or v have degree one or are articulation points.

T is the tree built by running the *DFS* algorithm, and s is the node from which the *DFS* started. Forward-edges are edges actively traversed by *DFS*, oriented in that direction (that make up T), backward-edges are the others.

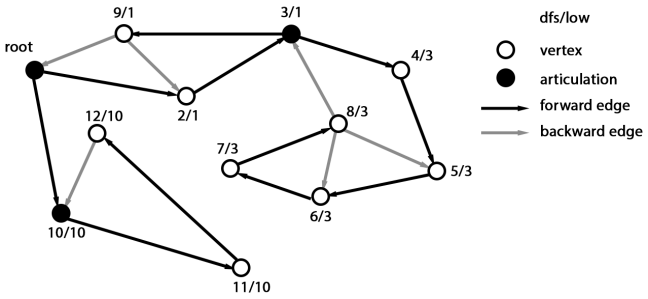


Figure 1.6: A graph showing how to find articulations. Dark edges are forward-edges, light edges are backward-edges, and dark vertices are articulations. The notation x/y indicates the *dfs* and *low* numbers.

Both articulation points and bridges can be found in $\mathcal{O}(|E|)$ with the following algorithm.

Algorithm 1.3: DFS-Visit(G,v)

```
1  num← num+1
2  dfs[v] ←num
3  low[v] ← num
4  isArtPoint[v] ← FALSE
5  for all {v, w} ∈ E do
6      if dfs[w]=0 then
7          T ← T + {v, w}
8          val←DFS−Visit(G,w)
9          if val≥dfs[v] then
10             isArtPoint[v]=TRUE
11         low[v] ← min{low[v], val}
12         else dfs[w] ≠ 0 and {v, w} ∉ T
13             low[v] = min{low[v], dfs[w]}
14 return low[v]
```

This code does not check the condition for the root of the DFS-tree. This can be easily modified at the end: if the root has degree at least two in the DFS tree, then it is an articulation point.

1.4 Matchings

A set of edges $M \subseteq E$ is called *matching* in a graph $G = (V, E)$ if no vertex in the graph is incident to more than an edge in M. Formally

$$e \cap f = \varnothing \qquad \text{for all } e, f \in M \text{ with } e \neq f$$

In this section we are interested in computing a matching M of maximum size. We introduce two important concepts:

- A matching M is maximal if any edge in E not already in M added to M causes M not to be a matching anymore. It's easy to find, *e.g.* with *greedy* strategy by iterating over all the edges (and picking

only those who have both vertices still free) in time $O(m)$. The greedy strategy finds in $O(|E|)$ a maximal matching with size

$$|M_{greedy}| \geq \frac{1}{2}|M^*|$$

- A matching M^* that contains the maximum number of edges. This number $|M^*| = k$ is called matching number. Informally, this is the best matching. It holds that

$$|M| \leq |M^*| \wedge |M^*| \leq 2|M| \implies |M| \geq \frac{k}{2} = \frac{|M^*|}{2}$$

An algorithm exists that computes it in $O(m\sqrt{n})$.

Perfect Matching

A matching M_p with $|M_p| = \frac{|V|}{2}$, meaning that every vertex of the graph is incident to exactly one edge of the matching. Not every graph possesses it.

If $G = (V, E)$ is a 2^k-regular bipartite graph, then we can find a perfect matching in $O(|E|)$.

If $G = (A \uplus B, E)$ is a k-regular bipartite graph, then there exist $M_1, \ldots, M_k$ such that $E = M_1 \uplus \ldots \uplus M_k$ and all M_i with $1 \leq i \leq k$ are perfect matchings.

Augmenting Path

A path in $G = (V, E)$ is an *augmenting* path iff its edges are alternatively not in M and M, starting and ending not in M. This means that an augmenting path has odd length. The idea of some matching algorithms is to find such an augmenting path and then remove all the edges in the augmenting path from M and add all those not currently in M, effectively swapping the contained edges and increasing the matching size by 1. An augmenting path can be found in $O(n + m)$ in a *bipartite* graph. The idea is the following: we orient edges not in the matching from A to B, and edges in the matching from B to A. Moreover, we add two nodes s and t: s has edges to all nodes in A that are not touched by any edge in the matching; t has incoming edges from all nodes in B that are not touched by any edge in the matching. Every path from s to t represents an augmenting path.

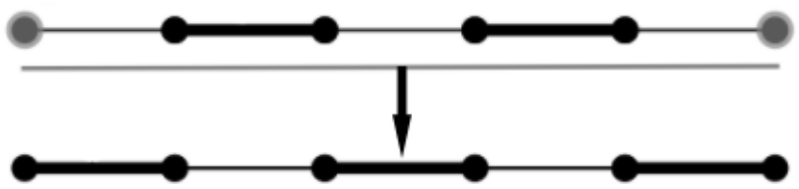


Figure 1.7: Augmenting path once the edges are swapped

Blossom's Algorithm

An algorithm that finds a maximum matching in a given graph. It does so by iteratively improving the solution using augmenting path, until no more augmenting path exists (meaning the matching is maximum).

Hall's Marriage Theorem

Necessary and sufficient condition to have a matching that covers at least one side of the bipartite graph. Given a bipartite graph $G = (X \cup Y, E)$. For a set $W \subseteq X$, let $N_G(W)$ be the neighbourhood of W in G (all vertices in Y adjacent to W.

$$\exists \text{ matching that entirely covers X} \iff \forall\, W \subseteq X \cdot \quad |W| \leq |N_G(W)|$$

Informally Every finite subset W has enough adjacent vertices in Y. This makes intuitive sense as if there are not enough possible vertices to be matched, then it's simply not possible.

Vertex Cover

Given a graph, the problem of finding a vertex cover consists of finding a subset $C \subseteq V$ such that every edge is incident to at least one vertex in C. This informally could be understood as the problem of placing cameras in vertices such that no street (edge) is left unwatched. We usually are interested in minimizing the size of C (to save some cash on expensive cameras), therefore we look for a minimum vertex cover (minVC). The problem is NP-complete and therefore hard to solve in general.

Definition 1.4.1 *Find $C \subseteq V$ with minimal $|C|$ such that*

$$\forall e = \{u, v\} \in E \cdot \quad u \in C \vee v \in C$$

König's Theorem This theorem states the equivalence between the minimum vertex cover and the matching problem in bipartite graphs. Therefore the size of a maximum matching (number of edges) corresponds to the size of a minimum vertex cover (number of vertices). This allows us to compute the size of the minVC using more efficient matching algorithms.

$$|M^*| = |\text{minVC}|$$

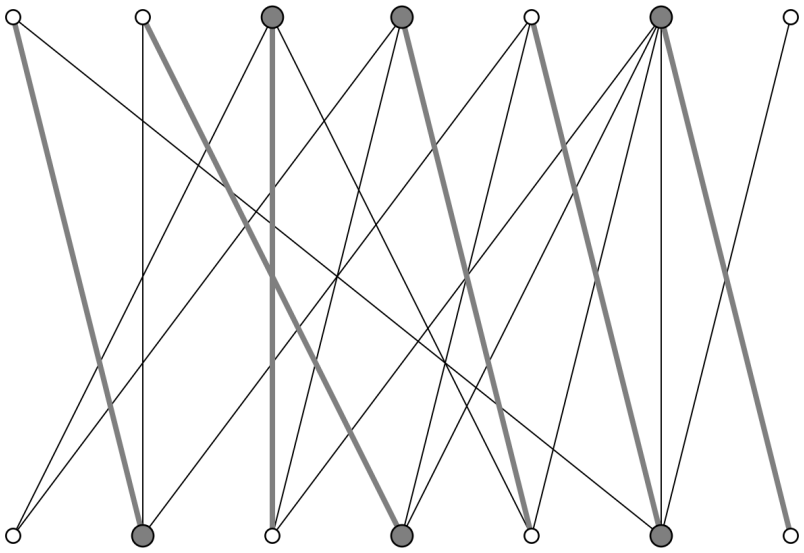


Figure 1.8: Duality between matching (edges) and vertex cover (vertices)

1.5 Eulerian Circuits

Könisberg was a town in Prussia, divided in four land regions by the river Pregel. The regions were connected with seven bridges. Leonhard Euler gave a formal solution to the problem of finding a tour through the town that crosses each bridge exactly once and so established the graph theory field in mathematics. An Eulerian circuit in a graph $G = (V, E)$ is a closed path (cycle) that contains each edge of G exactly once. One can show the following theorem, which is the crucial observation for our algorithm.

Theorem 1.5.1 *A connected graph $G = (V, E)$ contains an Eulerian circuit $\iff$ all nodes in G have even degree.*

Hence, in order to check whether a graph contains an Eulerian circuit, we iterate through each vertex and we check the parity of the degree of each node. Another similar concept is the one of Eulerian path: instead of starting from v and coming back to v, we have a path that touches all edges. A graph has an Eulerian path if and only if there are at most two edges with odd degree. Intermezzo closed, now we want to design an algorithm to find an Eulerian circuit in a connected graph where all nodes have even degree.

We start in an arbitrary vertex $v \in V$ and we imagine a runner that starts in v. The runner runs across an arbitrary walk in the graph and thereby it deletes all edges that he used, such that those edges will not be used again. The runner will be over in v (in facts, it is impossible that it gets stuck in another vertex, otherwise we would have a contradiction to the fact that all vertices have even degree). We call W the circuit that we obtained. W could either be an Eulerian circuit if it contains all edges of the graph, or a part of it. In order to check this we use another runner that runs on W: if it gets back to v without finding any untouched edge we are over, otherwise this runner finds another circuit W' in the graph (again by deleting used edges) and merges it with W. The second runner then goes on in W merged with W' and, whenever he find unused edges, he finds a new circuit to merge with the solution. When the second

runner gets back to v we are over. More formally we have the following algorithm

```
1 W ←RandomTour(G, v)
2 secondRunner← v
3 while secondRunner is not equal to v do
4     u ← successor of secondRunner in W
5     if u has some neighbours then
6         W′ ←RandomTour(G, u)
7         Merge W and W′
8         secondRunner← successor of secondRunner in W
```

Algorithm 1.4: EulerTour(G,v)

```
1 W ←< v >
2 while v has some neighbours do
3     Choose an arbitrary neighbour u of v
4     Add u to W
5     Delete {v, u} from G
6     v ← u
7 return W
```

To implement the algorithm we need to find $|E|$ times an incident edge to a vertex and delete it from the graph. Both operations can be implemented in $\mathcal{O}(1)$ with double linked adjacency list. By looking at the algorithm above, we see also that we can merge two cycles in constant time. For this reasons, the complexity of the algorithm is $\mathcal{O}(|E|)$.

1.6 Hamiltonian Cycles

In the previous chapter we studied Eulerian circuits and we have seen an $\mathcal{O}(|E|)$ algorithm to find a circuit which uses each edge exactly once. In this chapter we study an apparently similar problem: finding a circuit in a graph that uses each vertex exactly once. Although this might seem just a variant of the previous problem, this is very different and (presumably) more difficult. In facts, determining whether a graph contains an Hamiltonian cycle is an NP-complete problem. This means that, given a possible solution, we can check in polynomial time whether it is correct or no. However, it is not known whether a polynomial time algorithm for such problem exists or not. If one would find a polynomial time algorithm for this problem, he would show that P=NP and thereby he would solve the most famous open problem in Computer Science. We don't go in further detail about the P vs NP problem, but keep in mind that for the general problem of deciding whether a graph contains an Hamiltonian cycle no polynomial time algorithm is know. In the script there is a dynamic programming approach that solves the problem in exponential time, but we don't present it here. However we state some important results about special cases of this problem:

- A complete grid with m rows and n columns contains an Hamiltonian cycle if and only if $m \cdot n$ is even.
- A bipartite graph with partitions A and B can not contain an Hamiltonian cycle if $|A| \neq |B|$.

- Hypercubes of dimension d contain an Hamiltonian cycle.
- Dirac's theorem: every graph $G = (V, E)$ with $|V| \geq 3$ and minimal degree $\geq |V|/2$ contains an Hamiltonian cycle.

1.7 Travelling Salesman Problem

The travelling salesman problem (TSP) is a famous generalization of the Hamiltonian cycle problem. A businessman wants to visit n cities exactly once by starting from its own city and coming back. He knows the time to travel between each pair of cities and he wants to pick the shortest circuit. Formally: we are given a complete graph K_n and a function $l : \binom{[n]}{2} \to \mathbb{N}_0$ that assign a weight to each edge. We want to find an Hamiltonian cycle C in K_n with $sum_{e \in C} l(e) = \min\{\sum_{e \in C'} |C' \text{ is an Hamiltonial cycle in } K_n\}$.

The problem is not easier as determining whether a graph contains an Hamiltonian cycle. In facts, if we had an algorithm to find the optimal tour, we could decide whether a graph G contains an Hamiltonian cycle in the following way: we build a complete graph with the same number of vertices as G and we give weight zero to the edges of G and one to the others. If the optimal tour has weight zero, then G contains an Hamiltonian cycle. In general, in the context of optimisation problems such as TSP, one could be happy of finding a solution that is not worse as α times the value of the optimal solution. In this case this is not possible: in facts, if we had such an algorithm, we could determine whether a graph contains an Hamiltonian cycle as before (because α time zero, i.e. the value of the optimal solution if the graph contains an Hamiltonian cycle, is still zero). Hence, having an α approximation algorithm for TSP would be equal to solving the NP-complete problem of the Hamiltonial cycle.

For this reason, we introduce a relaxed version of the TSP that makes sense in several real world scenarios: the metric TSP. Given a complete graph K_n and a function $l : \binom{[n]}{2} \to \mathbb{N}_0$ such that $l(\{x, z\}) \leq l(\{x, y\}) + l(\{y, z\})$ for all $x, y, z \in [n]$. Similarly as before we want to find an Hamiltonian cycle C in K_n with $sum_{e \in C} l(e) = \min\{\sum_{e \in C'} |C' \text{ is an Hamiltonial cycle in } K_n\}$. The condition $l(\{x, z\}) \leq l(\{x, y\}) + l(\{y, z\})$ is called triangle inequality and means: going from x to z directly is shorter or equal than doing a stop in y.

2-Approximation Algorithm for the Metric TSP

In this section we explain an algorithm that finds a solution to the metric TSP that is at most twice as costly as the optimal solution. The runtime of the algorithm is $\mathcal{O}(n^2)$. The algorithms works as follows:

1. We compute an MST in the graph in $\mathcal{O}(n^2)$.
2. We follow the tree. When we have to reuse the same edge to visit another one, we go directly to the target vertex. Since the weights satisfy the triangle inequality, this will not make the run longer as twice the cost of the MST. Since leaving away an edge from an Hamiltonian cycle returns a tree, we have that the cost of the MST

is at most as large as the cost of the optimal tour. Hence we have an algorithm that returns a circuit with cost less equal than the double of the optimal one.

1.5-Approximation Algorithm for the Metric TSP

The runtime of this approximation is $\mathcal{O}(n^3)$. We assume, as a black-box subroutine, that it is possible to find a perfect matching of minimal cost in a complete graph with n nodes and n even. The idea is similar to the one of the previous subsection: we compute a MST, we modify the graph in order to get an Euler tour and then we do the shortcuts. The only difference is than, instead of doubling each edge of the MST, we do something more clever: after we computed the MST, we denote with S the set of nodes that have odd degree in the MST. According to the sum of degree formula, the cardinality of S is even. We consider the complete subgraph with nodes from S. Since $|S|$ is even, the complete subgraph contains perfect matchings. We use our black-box algorithm to comptue the perfect matching with minimum cost in the complete subgraph with nodes from S in $\mathcal{O}(n^3)$. Now we consider the union of the MST T and the perfect matching with minimal cost M: this graph contains an Euler tour because now each node has even degree. Similar as before, we go on this Euler tour and we do the shortcuts. We have to show that this is a 1.5 approximation. Using the same notation as in the script and the triangle inequality, we get:

$$\ell(C) \leq \ell(T) + \ell(M)$$

and in order to prove the statement we have to show that $\ell(M) \leq \frac{1}{2}OPT$. In order to do that consider the optimal tour on S: this is has at most the same length of the optimal tour in G. On this tour there are two perfect matchinngs and at least one of them costs at most $\frac{1}{2}OPT$. The perfect matching with minimal cost in the complete subgraph of S has at most this cost, and this proves that this is a 1.5 approximation.

1.8 Graph colouring

One can solve many problems by finding in an appropriate graph a partition of the vertices, such that edges connect only vertices in different partitions. As an example consider exam scheduling. We consider a graph $G = (V, E)$ where V is the set of exams and we have an edge $\{u, v\}$ if and only if exam u and exam v have a student in common. Hence we can say that edges represent *conflicts*. The goal is to find the minimal number of partitions of such graph such that there are no edges within the same partition (an edge within the same partition would mean that two exams with at least a student in common would take place at the same time). Such a value represents the minimum number of time slots needed for such an exam session.

In general we define a **(vertex) colouring** of a graph $G = (V, E)$ with k colors as a mapping $c : V \to [k]$ such that $c(u) \neq c(v)$ for all edges

$\{u, v\} \in E$. Moreover we define the **chromatic number** $\chi(G)$ as the minimum number of colors needed to color G.

An important example is the case where $\chi(G) = 2$. Such graphs are called **bipartite**. In general we have:

Theorem 1.8.1 *A graph $G = (V, E)$ is bipartite $\Leftrightarrow$ it does not have any cycle of odd length as subgraph.*

Proof. $\Rightarrow$ Consider a simple triangle as an example for an indirect proof.

$\Leftarrow$ We start a BFS from an arbitrary vertex s and we color a vertex with color 1 (2) if and only if its distance from s is even (odd). Since there is no cycle of odd length there could be no edge such that its vertices have the same color. □

A classical graph colouring problem is the colouring of maps, where neighbouring lands should be assigned to different colors. This problem comes with the assumption that the territory of each land is connected and that lands that touch in a single point can be coloured with the same color. An important theoretical result in graph colouring is that every such map can be coloured with (at least) four colors.

How can we determine the colouring of a graph with the least number of colors? In order to decide, whether a graph is bipartite or not, a BFS is sufficient, but how can we extend to a larger chromatic number? In general, graph colouring is a difficult problem. Already the question *does it holds for a given graph $G = (V, E)$, that $\chi(G) \leq 3$?* is NP complete. This means that (with the assumption P≠ NP) there is no polynomial algorithm that computes the chromatic number of a graph. In practice this means that we have to find approximations of the optimal solution.

The following algorithm computes a colouring of the vertices of the graph by visiting the vertices in an arbitrary order $v_1, \ldots, v_n$ and assigning to each vertex the lowest id of a colour, which is not used among its neighbours.

Algorithm 1.6: Greedy-Colouring(G)

```
1 Choose an arbitrary order of the nodes v_1, ..., v_n
2 c(v_1) ← 1
3 for i = 2 to n do
4     c(v_i) ← min{k ∈ ℕ | k ≠ c(u) for all u ∈ N(v_i) ∩ {v_1, ..., v_{i-1}}}
```

We begin with the observation that we can implement the algorithm by initialising at each step an array of length $deg(v_i) + 1$ and iterating on each neighbour of v_i. Whenever we find a neighbour which is already coloured we put the relative entry in the array to true. At the end we iterate in such an array and we assign to v_i the color corresponding to the lowest false entry in the array. This means that the algorithm has complexity $\mathcal{O}(\sum_{v \in V} deg(V)) = \mathcal{O}(|E|)$.

It is clear that the algorithm returns a correct colouring, because given this construction the colour of a vertex is always different from the one of its neighbours. Now the question is, how many colors does the algorithm uses in the worst case? Since the algorithm chooses the smallest color that is not yet used in one of the neighbours, we have the worst case scenario

when the neighbours of a vertex v_i are already coloured with colors $1, \ldots, deg(v_1)$. In this case v_i gets the color $deg(v_i) + 1$. This means that the algorithm uses at most $\Delta(G)+1$ colors, where $\Delta(G) := \max_{v \in V} deg(V)$ is the maximal degree of a node in G. It also holds that the algorithm returns a colouring with at least $\chi(G)$ colours (this follows from the fact that the colouring we return is correct). The number of colors used by the algorithm depends on which sequence of vertices we consider. It always exists a sequence that yields to the correct solution but, since we don't know this sequence, we can get a worse result. In general, choosing the correct sequence is not an easy task and there are various heuristics. In the remainder of this section we summarize some results.

Theorem 1.8.2 (Brook's Theorem) *There is an algorithm with complexity $\mathcal{O}(|E|)$ that computes a sequence for which we will use at most $\Delta(G)$ colours, if such a colouring exists.*

Theorem 1.8.3 *Let $G = (V, E)$ be a graph and k a natural number such that the induced sub-graph of G contains a node with degree at most k. It holds $\chi(G) \leq k + 1$ and there is an algorithm that computes a coloring with $k + 1$ colors in $\mathcal{O}(|E|)$*

Theorem 1.8.4 (Mycielski Construction) *For all $k \geq 2$ there is a triangle-free graph G_k with $\chi(G_k) \geq k$.*

Theorem 1.8.5 *A graph $G = (V, E)$ with $\chi(G) = 3$ can be colored in $\mathcal{O}(|E|)$ with $\mathcal{O}(\sqrt{|V|})$ colors.*

1.9 Network Flow

Network Definition A *network* is a tuple (V, A, c, s, t) with

- $G = (V, A)$ a directed graph
- $c : A \to \mathbb{R}_0^+$ the capacity function
- $s \neq t \in V$ the source and target nodes

This is simply a directed graph with two nodes marked as source and target, and a non-negative value assigned to each edge stating the capacity for transport of that edge.

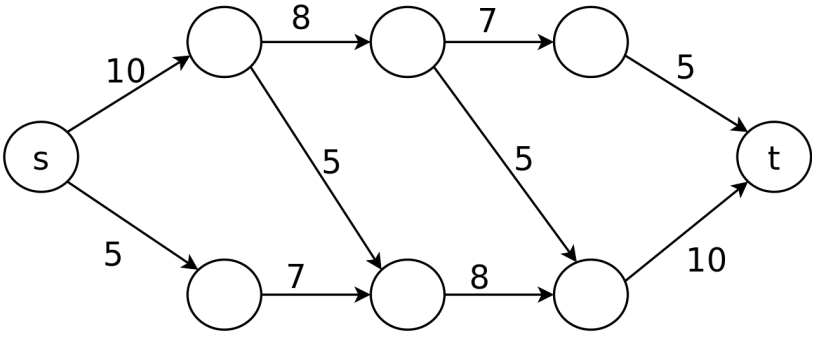


Figure 1.9: Simple network graph with capacities marked

Flow Definition A *flow* is a function $f : A \to \mathbb{R}_0^+$ with the following conditions:

- $0 \leq f(e) \leq c(e) \quad \forall e \in A$
- $\sum_{u \in V:(u,v)\in A} f(u,v) = \sum_{u \in V:(v,u)\in A} f(v,u) \quad \forall v \in V \backslash \{s,t\}$

The first condition means that on every edge a non-negative amount of flow not exceeding the capacity might pass. The second condition implies that no flow disappears or appears at any given node (except source and target), i.e. it is conserved in the network.

Inflow and Outflow We can define the following quantities:

- $\text{inflow}(v) := \sum_{u \in V:(u,v)\in A} f(u,v)$
- $\text{outflow}(v) := \sum_{u \in V:(v,u)\in A} f(v,u)$

With these quantities we can say

$$\forall v \in V \backslash \{s,t\} \cdot \text{inflow}(v) = \text{outflow}(v)$$

which is the same as the previous flow conservation condition.

Flow Value The value of a flow is given by

$val(f) :=$ netoutflow$(s) :=$ outflow$(s)-$inflow(s) = inflow$(t)-$outflow(t) =: netinflow(t)

This shows that due to the conservation of flow property, the flow generated by the source is the same as the flow reaching the target.

Duality with the Min Cut Problem The algorithmic problem we study in this section, is to efficiently compute a maximum flow in the graph, *i.e.* a flow with maximum value. We observe that the existence of a maximum flow is not obvious: there could be a situation similar to finding a maximum in the interval $(0,1)$, where a maximum does not exist. In the general case, a network has infinitely many flows and, although if we knew that there exist a maximum flow, it would not be easy to show that this is indeed the maximum flow in the graph. In order to get insights about this issues we introduce a dual problem.

An $S-T$ cut in a network is a partition of V into $S, T \subset V$ such that $S \cap T = \varnothing, S \cup T = V$ with $s \in S, t \in T$. The capacity of this cut is how much flow can pass maximally from S to T and is defined as

$$cap(S,T) = \sum_{(u,v)\in(S\times T)\cap A} c(u,v)$$

Since an $S-T$ cut defines how much capacity can go maximally through the network at some point, we have

$$val(f) \leq cap(S,T) \quad \forall f, S, T$$

Furthermore, the Maxflow-Mincut Theorem states that there exists a flow f_{max} that makes the above inequality hold exactly, i.e.

$$f_{max} = \max_{f} val(f) = \min_{(S,T)\text{-cut}} cap(S,T)$$

This gives an answer to our questions: yes, there is a maximum flow in the graph and its value is equal to the minimum $S-T$ cut in the graph. We will look to efficient algorithms to compute min cuts in Chapter 3, but since cuts in a graph are finite, we have an algorithm to determine it (*i.e.* enumerating them and keeping the minimum). Summarising: if we find for a flow f and $S-T$ cut with $cap(S,T) = val(f)$, we have a certificate that f is the maximum flow. Moreover, the existence of a minimum $S-T$ cut implies the existence of a maximum flow.

Augmenting Paths The idea of the algorithm that we present here is to improve a given flow. In order to do that we search an *augmenting* path in the graph, *i.e.* a path from s to t in the graph, such that the flow on each edge of the path is strictly less than the capacity of the edge. Then, we can increase the flow on each edge on the path with the quantity $\delta := \min_{e \in \text{ path}} c(e) - f(e)$ without breaking any flow property. Unfortunately, there are sub-optimal flows that can not be improved in this way. A key observation here is that the increase of the flow on an edge can not only be compensated with an increase of the flow of an outgoing edge, but also with the decrease of the flow of another incoming edge.

Residual Network Given a network $G = (V, A)$ with a flow, we define the residual network as follows: for each edge $e \in A$ with $f(e) < c(e)$ in G, we have an edge with weight $c(e) - f(e)$ in the residual network. Moreover, for each edge $e \in A$ with $f(e) > 0$ in G, we have an edge with opposite direction with weight $f(e)$ in the residual graph. Shortly: for each directed edge in G we have two edges in the residual graph. One in the same direction as the original edge that represents the margin with respect to the flow considering the capacity of the edge and another one in the opposite direction with the same value of the flow on that edge. Residual networks are useful because we can show that if there is no path from s to t in them, then we have a maximum flow.

Ford-Fulkerson Algorithm This algorithm finds a maximal flow in a graph by iteratively finding an augmenting flow in the residual network. The algorithm is guaranteed to terminate if the capacities are integers (but not if they are rational). It works as follows:

```
1 f ← 0
2 while ∃ s−t−Path P in the residual network
3     increase flow f along P
4 return f
```

Algorithm 1.7: Ford-Fulkerson

Since in every iteration the flow is augmented by at least 1 unit, and the algorithm stops when we have a max flow with value f (as then no augmenting path exists), we know the algorithm will run a maximum of

f times. Each time we must construct the residual network, and this takes $O(|E|)$. Therefore the algorithm takes in total $O(f|E|)$. We can estimate $f \leq nU$, where $U = \max_{e \in E} cap(e)$, meaning that no flow can be higher than the maximum capacity times the amount of vertices (due to any cut). This however is a very imprecise estimate in general, but gives an upper-bound.

Useful Tricks Flow problems have many different application in the real world, most of which have to do with scheduling the use of a finite amount of resources or find some optimal utilization and transport of quantity in a network. The following tricks allow to adapt problems that appear to not fit into this model:

- Multiple sources/targets: add a master source/target and connect it to each source/target with ∞ capacity
- Undirected graphs: duplicate the edges, make them directed with the same original capacity
- Vertex capacities: split the vertex v into two vertices v_{in} and v_{out} with only incoming/outgoing edges, and limit the capacity with an additional edge between them.

The following figure should help remember these tricks.

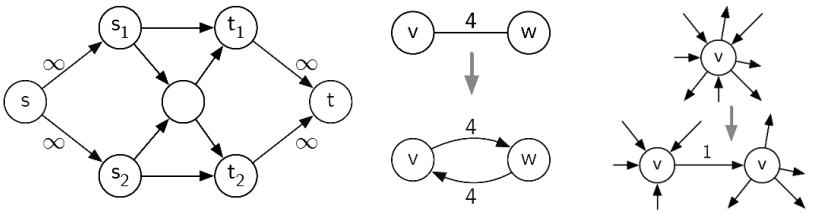


Figure 1.10: The tricks shown graphically

Another interesting observation is that we can compute maximum matchings in bipartite graphs with a network flow approach: we add a source s with edges of infinite capacity to each node of the first partition and from each node of the other partition we add edges with infinite capacity to the target. Edges of the bipartite graph have capacity one. By following the maximum flow in this graph we find a perfect matching in the underlying bipartite graph.

Probability Theory 2

In this course you had your first contact with probability theory, at least with respect to your ETH studies in Computer Science. Probability theory is a wonderful theory that has both mathematical foundations (which you will study in more detail in the course *Wahrscheinlichkeit and Statistik* in the fourth semester) and practical applications (which you have seen in this course and you will see in other courses at CADMO or in the area of Machine Learning). In the context of this course, the chapter about probability theory can be divided into two fundamental parts: a mathematical approach to the essential concepts of (discrete) probability and an algorithmic approach to some problems that can be solved by exploiting the idea of randomization.

In this chapter we first introduce notions of probability theory which will be useful in the next chapter, where we will exploit them in the context of randomized algorithms.

2.1 Basic Concepts of Discrete Probability Theory

Definition 2.1.1 *When doing a random experiment, there is a set of possible outcomes. These outcomes form the* sample space. *A discrete sample space* $\Omega = \{w_1, \ldots, w_n\}$ *is composed of* elementary events $w_i \in \Omega$, *all of which have a probability* $Pr[w_i]$. *The probability is a function that measures how "likely" is an event and has two fundamental properties:*

- $0 \leq Pr[w_i] \leq 1$
- $\sum_{i=1}^{n} Pr[w_i] = 1$

A set $E \subseteq \Omega$ *is called* event. *The probability* $Pr[E]$ *is defined as the sum of the elementary events included in* E. *We also define the* complementary event $\bar{E} := \Omega \backslash E$.

In general, the probability is a function from the set of of events (often denoted 2^{Ω}) to the interval $[0, 1]$. The probability function satisfies the following properties (some of those are axioms, other can be easily checked via Venn's diagram representation, we don't want to be too formal here):

- $Pr[\Omega] = 1$
- $Pr[\varnothing] = 0$
- $Pr[\cup_i E_i] = \sum_i Pr[A_i] \iff A_i \cap A_j = \varnothing \forall i \neq j$
- $Pr\left[\bar{A}\right] = 1 - Pr[A]$
- $A \subseteq B \Rightarrow Pr[A] \leq Pr[B]$
- $Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B]$

Inclusion/ Exclusion Principle

In the previous section, we stated different properties of the probability function. Particularly, we stated

$$Pr\left[\cup_i E_i\right] = \sum_i Pr\left[A_i\right] \Longleftrightarrow A_i \cap A_j = \varnothing \forall i \neq j$$

Theorem 2.1.1 generalizes this result.

Theorem 2.1.1 *If the events $A_1, \ldots, A_n$ are pairwise disjunct (i.e. if for all pairs $i \neq j$ it holds $A_i \cap A_j = \varnothing$), then*

$$Pr\left[\cup_{i=1}^n A_i\right] = \sum_{i=1}^n Pr\left[A_i\right]$$

But what happens if the events are not disjunct? The general case is covered by the *inclusion/exclusion principle*, stated in the Theorem 2.1.2.

Theorem 2.1.2 (Inclusion/ exclusion principle) *For events $A_1, \ldots, A_n$ ($n \geq 2$), we have*

$$\begin{aligned} Pr\left[\cup_{i=1}^n A_i\right] &= \sum_{l=1}^n (-1)^{l-1} \sum_{1 \leq i_1 < \cdots < i_l \leq n} Pr\left[A_{i1} \cap \cdots \cap A_{il}\right] \\ &= \sum_{i=1}^n Pr\left[A_i\right] - \sum_{1 \leq i_1 < i_2 \leq n} Pr\left[A_{i1} \cap A_{i2}\right] \\ &+ \sum_{1 \leq i_1 < i_2 < i_3 \leq n} Pr\left[A_{i1} \cap A_{i2} \cap A_{13}\right] - \cdots + (-1)^{n+1} \cdot Pr\left[A_1 \cap \cdots \cap A_n\right] \end{aligned}$$

This result is very important and gives the exact result to the problem. However, for a large value of n, it gets tedious to compute. The *Union Bound* comes to rescue by giving an approximation to the value of the same probability.

Theorem 2.1.3 *For events $A_1, \ldots, A_n$ it holds*

$$Pr\left[\cup_{i=1}^n A_i\right] \leq \sum_{i=1}^n Pr\left[A_i\right]$$

For the sake of simplicity, we avoid giving a formal proof of Theorems 2.1.2 and 2.1.3. The intuition for Theorem 2.1.2 can be given with Venn's diagram, for Theorem 2.1.3 just consider that we don't subtract any overlap.

Principle of Laplace

In general, the probability of an event is the sum of the individual probabilities of the elementary events that compose it.

$$Pr[A] = \sum_{w \in A} Pr[w]$$

A fundamental question is how to determine the probabilities of elementary events. Laplace proposed to assume that all elementary events have the same probability. This choice maximizes the entropy (in facts, if we give to an elementary event a larger probability we assume prior knowledge). Under *principle of Laplace* we understand that if we are considering a discrete sample space Ω of cardinality n, then all elementary events have probability $\frac{1}{n}$. Under this assumption we get

$$Pr[A] = \frac{|A|}{n}$$

In other words, we describe the probability of an event as the number of "favorable" events divided by the total number of "possible" events.

Conditional Probability

Until know we considered the case where we don't have any prior information regarding the probability space. But what happens if we already have some information about the outcome of an experiment?

Example 2.1.1 Throw a fair dice. You know that the result is an odd number. What is the probability, that you got a prime number?

The prior knowledge that the result is odd reduces the probability space from $\{1, 2, \ldots, 6\}$ to $\{1, 3, 5\}$. Since both 3 and 5 are prime, we get a probability of $\frac{2}{3}$.

We can formalise the argument in the following way. Let $A := \{1, 3, 5\}$ be the event that the result is odd and $B := \{2, 3, 5\}$ the event that the result is prime. We can read the probability of observing a result from B when we already know that we observed a result from A as follows: we know that the "good events" are the one contained in the set $A \cap B$ and the "possible events" are the one contained in A. Hence we get

$$Pr\left[B \text{ given } A\right] = \frac{Pr[A \cap B]}{Pr[A]}$$

In general, given two events A and B in a sample space Ω, we want to determine the probability of event B knowing already that A happened. We write it $Pr[B|A]$ and we say *probability of B given A*. Similarly as above we argue that the fact that A happened induces a new sample space. This explains the following theorem.

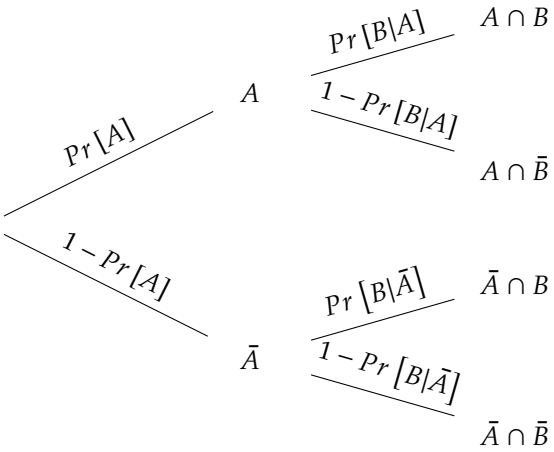
Theorem 2.1.4 (Conditional Probability) *Let Ω be a sample space and A, B two events in Ω. We get*

$$Pr[B|A] = \frac{Pr[A \cap B]}{Pr[A]}$$

We can rearrange the formula for condition probability in the following way

$$Pr[A \cap B] = Pr[B|A] \cdot Pr[A]$$

This argument can be easily visualized in a probability tree.



The above tree can be useful to compute other probabilities: $Pr[B]$ and $Pr[A|B]$. We first compute them in the example above and then we present two theorems than generalize the concept.

- $Pr[B] = Pr[A] \cdot Pr[B|A] + Pr\left[\bar{A}\right] \cdot Pr\left[B|\bar{A}\right]$

Theorem 2.1.5 (Total Probability) *If we divide Ω into a disjunct partition of A_i for $i \in [m]$, we have*

$$Pr[B] = \sum_{i=1}^{m} Pr\left[A_j\right] \cdot Pr\left[B|A_j\right]$$

- $Pr[A|B] = \frac{Pr[B|A] \cdot Pr[A]}{Pr[B]}$

Theorem 2.1.6 (Bayes'Theorem) *If we divide Ω into a disjunct partition of A_i for $i \in [m]]$ and we consider an event B with $Pr[B] > 0$, we get*

$$Pr[A_i|B] = \frac{Pr[A \cap B]}{Pr[B]} = \frac{Pr[B|A_i] \cdot Pr[A_i]}{\sum_{j=1}^{m} Pr\left[B|A_j\right] \cdot Pr\left[A_j\right]}$$

Exercise 1. You are on a TV show and the questions are extracted u.a.r from a pool of questions. With probability p you know the answer and you answer correctly. Otherwise you guess and you give the right answer with probability $1/4$. What is the probability that you answer correctly to a u.a.r extracted question?

Exercise 2. You are on another TV show and you have to choose one

out of three closed doors. Behind one door there is a wonderful race car, behind the other two doors some friendly goats. You choose a door. The TV host opens a door that you have not chosen and behind there are goats. Then he makes you this offer: you can change your choice. Do you have a probabilistic advantage in accepting the offer?

Independence of Events

In the previous subsection we studied the formula for conditional probability. When we consider $Pr[B|A]$, the prior knowledge that B happened can increase, decrease or not influence the probability of A. If the even A does not influence event B, we have $Pr[B|A] = Pr[B]$. This fact also implies that B does not influence A

$$\begin{aligned} Pr[B] = Pr[B|A] &= \frac{Pr[A \cap B]}{Pr[A]} \\ \iff Pr[A] \cdot Pr[B] &= Pr[A \cap B] \\ \iff Pr[A] = \frac{Pr[A \cap B]}{Pr[B]} &= Pr[A|B] \end{aligned}$$

The intuitive concept of independence is the following: two events are independent if they don't influence each other. In other words, knowing that an event happened does not give us any additional information about the probability of another event. This last observation motivates the following definition.

Definition 2.1.2 (Independence of two Events) *Two events A and B are (stochastic) independent if one of the following three (equivalent) proposition hold.*

- $Pr[B|A] = Pr[B]$
- $Pr[A|B] = Pr[A]$
- $Pr[A \cap B] = Pr[A] \cdot Pr[B]$

How can we generalise the concept of independence of more than two events?

Definition 2.1.3 (Independence of n Events) *The events $A_1, \dots, A_n$ are (stochastic) independent if for all subsets $I \subseteq \{1, \dots, n\}$ with $I = \{i_1, \dots, i_k\}$ we have*

$$Pr\left[A_{i_1} \cap \cdots \cap A_{i_k}\right] = \prod_{j=1}^{k} Pr\left[A_{i_j}\right]$$

We point out that checking independence for all pair of events does not work, as shown in the following example.

Example 2.1.2 Consider the Laplace space $\Omega = \{1, 2, 3, 4\}$ and the following events:

- $A := \{1, 2\}$

- $B := \{1, 3\}$
- $C := \{1, 4\}$

We have

$$Pr[A] = Pr[B] = Pr[C] = \frac{1}{2}$$

We also have that

$$Pr[A \cap B] = Pr[A \cap C] = Pr[B \cap C] = Pr[\{1\}] = \frac{1}{4}$$

Hence

- $Pr[A \cap B] = Pr[A] \cdot Pr[B]$
- $Pr[A \cap C] = Pr[A] \cdot Pr[C]$
- $Pr[B \cap C] = Pr[B] \cdot Pr[C]$

Thus, the events A, B, C are *pairwise* independent. Now consider

$$Pr[A \cap B \cap C] = Pr[\{1\}] = \frac{1}{4}$$

But $Pr[A] \cdot Pr[B] \cdot Pr[C] = \frac{1}{8} \neq \frac{1}{4} = Pr[A \cap B \cap C]$

Summarising: independence implies pairwise independence, but the opposite direction does not hold.

Independence is useful because it allows to compute the probability of the intersection of events simply with multiplications. This simplicity is often exploited in Machine Larning models that you will learn further in your studies. But what can we do to compute the intersection of non independent events? The next theorem helps us solving the problem in general.

Theorem 2.1.7 (Full Multiplication Rule) *For arbitrary random variables $A_1, \ldots, A_n$ with $Pr[A_1 \cap \cdots \cap A_n] > 0$ we have*

$$Pr[A_1 \cap \cdots \cap A_n] = Pr[A_1] \cdot Pr[A_2|A_1] \cdot Pr[A_3|A_1 \cap A_2] \ldots Pr[A_n|A_1 \cap \cdots \cap A_{n-1}]$$

Exercise 3. Show that if $Pr[A|B] = Pr\left[A|\bar{B}\right]$ and $\bar{B} = \Omega \backslash B$, then A and B are independent.

2.2 Discrete Random Variables

Often we are not really interested in the result of a random experiment. What we are *really* interested in are the consequences of such a result. For example when we play the roulette, we are not really interested in the number that comes out, but in the possible win or loss.

Example 2.2.1 Consider an unfair coin with probability of head (H) of 0.4 and a probability of tail (T) of 0.6. We do the following bet: if the coin shows head we lose 100$, otherwise we win 80$. How large is the win? It is useful to consider a function X that maps every possible

result of the random experiment to the corresponding win (or loss). We have

$$X(H) = -100 \text{ with probability } 0.4$$
$$X(T) = 80 \text{ with probability } 0.6$$

or, in a more compact form

$$X \sim \begin{cases} 0.4 & 0.6 \\ -100 & 80 \end{cases}$$

This example motivates the following definition.

Definition 2.2.1 (Random Variable) *A function*

$$X : \Omega \to \mathbb{R}$$
$$\omega \to X(\omega)$$

is called random variable. The function

$$f : X(\Omega) \to [0,1]$$
$$x \to f(x) := Pr[X = x]$$

is called probability distribution of the random variable X. If X takes a countable number of values, we call it discrete random variable.

It is useful to represent a discrete random variable X that takes values $\{x_1, x_2, \dots\}$ and with $p_k := f(x_k) = Pr[X = x_k]$ for $k = 1, 2, \dots$ in the following form

$$X \sim \begin{cases} x_1 & x_2 & \dots & x_k & \dots \\ p_1 & p_2 & \dots & p_k & \dots \end{cases}$$

Example 2.2.2 (Indicator Random Variable) A very important example of random variable is the indicator random variable. For an event $A \subseteq \Omega$, we define the corresponding indicator random variable as

$$X(\omega) = \begin{cases} 1 & \text{if } \omega \in A \\ 0 & \text{otherwise} \end{cases}$$

Definition 2.2.2 (Cumulative Distribution) *Let X be a random variable. We define the cumulative distribution function $F(X)$ as*

$$F(x_i) := Pr[X \leq x_i] = \sum_{k=1}^{i} p_k$$

$F(X)$ returns the probability, that X takes a value less equal than a given x.

Note that we can use $F(x)$ also to express $Pr[X > x]$

$$Pr[X > x] = 1 - Pr[X \leq x] = 1 - F(x)$$

Definition 2.2.3 (Operations on Random Variables) *Let X be a discrete random variable and $g : \mathbb{R} \to \mathbb{R}$ be a real function. We describe the distribution of $g(X)$ as*

$$X \sim \begin{cases} g(x_1) & g(x_2) & \dots & g(x_k) & \dots \\ p_1 & p_2 & \dots & p_k & \dots \end{cases}$$

Independence of Random Variables

Definition 2.2.4 *The random variables $X_1, \dots, X_n$ are independent if and only if for all $\{x_1, \dots, x_n\}$ in the codomain of $X_1, \dots, X_n$ it holds*

$$Pr[X_1 = x_1, \dots, X_n = x_n] = \prod_{i=1}^{n} Pr[X_i = x_i]$$

An important theorem about independent random variables is the following.

Theorem 2.2.1 *Let $f_1, \dots, f_n$ be real functions. If the random variables $X_1, \dots, x_n$ are independent, then also $f_1(X_1), \dots, f_n(X_n)$ are independent.*

Expected Value

The expected value is useful to determine the average outcome of a random experiment. We define it as follows.

Definition 2.2.5 (Expected Value) *We define the expected value of a random variable X as*

$$\mathbb{E}[X] := \sum_{\omega \in \Omega} X(\omega) \cdot Pr[\omega]$$

if this sum is absolute convergent. Otherwise, we say that the expected value is not defined.

One can show that, if the random variable maps results of a random experiment to natural numbers. an equivalent definition of expected value is given by

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} Pr[X \geq i]$$

A crucial property of the expected value is *linearity*. This property is very useful to solve a lot of exercises in this course. Formally, we have

Theorem 2.2.2 *For random variables $X_1, \dots, X_n$ and $a_1, \dots, a_n, b \in \mathbb{R}$, we have*

$$\mathbb{E}\left[b + \sum_{i=1}^{n} a_i X_i\right] = b + \sum_{i=1}^{n} a_i \mathbb{E}[X_i]$$

This property is particularly useful if a random variable can be expressed as a sum of simpler random variables with known expected value.

Example 2.2.3 We toss a fair coin 100 times. How many heads do we expect? We denote with X the number of heads and for $i = 1, \dots, 100$ we define X_i as the indicator random variable for head. We get

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{100} X_i\right] = \sum_{i=1} 100\mathbb{E}[X_i] = \sum_{i=1}^{100} \frac{1}{2} = 50$$

Theorem 2.2.3 *For independent random variables $X_1, \dots, X_n$, we have*

$$\mathbb{E}\left[\prod_{i=1}^{n} X_i\right] = \prod_{i=1}^{n} \mathbb{E}[X_i]$$

Exercise 4. Define a random variable X for which $\mathbb{E}[X]$ is undefined.

Exercise 5. Given a graph G with $2n$ vertices, with n vertices blue and n vertices red. The probability that there is an edge between two edges is $\frac{1}{2}$ for all pair of nodes. What is the expected number of edges between vertices of the same color?

Exercise 6. Consider a sequence of natural numbers in the interval $[0, 9]$. What is the expected length of the sequence until we get 0, 1, ..., 9 (not consecutively, but in this order)?

Variance

The expected value $\mathbb{E}[X[$ of a random variable X gives some useful information about it, but it does not say how X is spread. We would "like" to have a random variable such that $Pr\left[|X - \mathbb{E}[X]| \text{ large }\right]$ is small. This definitely does not always hold, consider for example a random variable such that $Pr[X = 0] = 0.5 = Pr\left[X = 10^{10}\right]$. Therefore we introduce another quantity to measure the spread of the distribution around its mean.

Definition 2.2.6 (Variance) *For a random variable X with $\mu = \mathbb{E}[X]$, we define its variance as*

$$Var[X] := \mathbb{E}\left[(X - \mu)^2\right] = \sum_{\omega \in \Omega} (X(\omega) - \mu)^2 Pr[\omega]$$

We also define the standard deviation $\sigma(x) := \sqrt{Var[X]}$

Computing the variance with the definition is quite tedious. One can show that the definition is equivalent to the following formula

$$Var[X] = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2$$

Two other important results about variance are given in the following theorems.

Theorem 2.2.4 *For an arbitrary random variable* X *and* $a, b \in \mathbb{R}$ *we have*

$$Var[a \cdot X + b] = a^2 \cdot Var[X]$$

Theorem 2.2.5 *For independent random variables* $X_1, \ldots, X_n$*, we have*

$$Var\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} Var[X_i]$$

Exercise 7. Given is the following distribution of a random variable X

$$X \sim \begin{cases} -4 & -2 & 5 & 8 & 10 \\ 0.25 & 0.10 & 0.20 & 0.15 & 0.30 \end{cases}$$

Compute $\mathbb{E}[2X + 8]$ and $Var[2X + 8]$.

Multiple Random Variables

We might find experiments in which we always observe multiple random variables at the same time, such as throwing a dice with both number and color on each face. We then obtain a joint probability

$$f_{X,Y}(x, y) = P[X = x, Y = y]$$

which expresses the probability of observing values x and y for the two random variables X and Y

It is possible to extract the individual probability distributions by *marginalization*

$$P[X = x] = f_X(x) = \sum_{y \in \mathcal{W}_Y} f_{X,Y}(x, y)$$

And similarly for $P[Y = y]$.

Theorem 2.2.6 *Let* X *and* Y *be two independent RV, and* $Z = X + Y$*. We then have*

$$f_Z(z) = \sum_{x \in \mathcal{W}_X} f_X(x) \cdot f_Y(z - x)$$

This allows us to compute the density function of a random variable using those of others, which might be helpful if we don't have the analytic function for f_Z.

Theorem 2.2.7 *Wals's equation states the following: let X and N be two independent random variables, with $\mathcal{W}_N \subseteq \mathbb{N}$. We further have $Z = \sum_{i=1}^{N} X_i$ where all X_i are independently distributed according to X. It then holds*

$$E[Z] = E[N] \cdot E[X]$$

It's easy to intuitively see that this holds since we expect to have $\mathbb{E}[N]$ copies of the same random variable, with the same expected value.

2.3 Important Discrete Distributions

Following, some of the most common and useful distributions are given, with density, expected value and variance.

Bernoulli Distribution

A Bernoulli random variable can take two values, 1 and 0 with probability p and $1-p$ respectively. This is useful to describe events that either happen or don't.

$$X \sim Ber(p)$$

$$f_X(x) = \begin{cases} p & \text{for } x = 1 \\ 1-p & \text{for } x = 0 \\ 0 & \text{else} \end{cases}$$

$$\mathbb{E}[X] = p \qquad Var[X] = p(1-p)$$

Exercise 8. Compute the variance of $X \sim Ber(p)$.

Binomial Distribution

A Binomial random variable is useful in the case of repeated Bernoulli events. For example, we can describe the probability of scoring a certain number of points in a game. We have n repeated Bernoulli events, each distributed with probability p. The assumption is that the events are i.i.d., i.e. *independently* and *uniformly distributed*.

$$X \sim Bin(n,p) = \sum_{i=1}^{n} Ber(p)$$

$$f_X(x) = \begin{cases} \binom{n}{x} p^x (1-p)^{n-x} & \text{for } x \in \{0,1,..n\} \\ 0 & \text{else} \end{cases}$$

$$\mathbb{E}[X] = np \qquad Var[X] = np(1-p)$$

Exercise 9. Consider a football league. A victory gives 3 points and a loss 1 points. There is no draw possible. Each team plays 15 games. Consider a team that wins each game independently with probability $p = 0.6$. Let X be the random variable for the number of points obtained by the team. Compute $\mathbb{E}[X]$ and $Var[X]$.

Geometric Distribution

A geometric random variable is used to describe the amount of time or events we have to wait for a Bernoulli event to happen. For example, it can be used to estimate the MTBF (mean time between failures), or the expected life of a hard disk before it dies.

$$X \sim Geo(p)$$

$$f_X(x) = \begin{cases} p(1-p)^{x-1} & \text{for } x \in \mathbb{N} \\ 0 & \text{else} \end{cases}$$

$$\mathbb{E}[X] = \frac{1}{p} \qquad Var[X] = \frac{1-p}{p^2}$$

An important property of the geometric distribution is the fact that it is *memoryless*. The fact that an event happened in the past doesn't change the probability of it happening in the future (for independent variables).

$$P[X \geq s + t | X > s] = P[X \geq t]$$

Exercise 10. Compute the expected value of $X \sim Geo(p)$

Negative Binomial Distribution

The negative binomial distribution is a generalization of the geometric distribution: instead of waiting for the first success, we repeat the experiment until we got n successes. Of course, if $n = 1$, we are back to the geometric distribution. But what if n is larger? The random variable X describes the number of repetitions until we see n successes of an event with probability p. Since the last repetition must be a success, we can obtain the distribution of X by distributing the other repetitions. We get

$$Pr[X = z] = \binom{n-1}{z-1} \cdot p^n (1-p)^{z-n}$$

Exercise 11. Compute $\mathbb{E}[X]$

Poisson Distribution

A Poisson random variable with parameter λ models the probability that a particular number of events will happen in a given time frame. The parameter means that in the given time frame, on average, λ such events happen. One example could be the number of births in Switzerland per week.

$$X \sim Po(\lambda)$$

$$f_X(x) = \begin{cases} \frac{e^{-\lambda}\lambda^x}{x!} & \text{for } x \in \mathbb{N}_0 \\ 0 & \text{else} \end{cases}$$

$$\mathbb{E}[X] = \lambda \quad Var[X] = \lambda$$

When we consider the limit $\lim_{n\to\infty} Bin(n, \frac{\lambda}{n})$ we obtain that it equals $Po(\lambda)$.

Exercise 12. Compute the expected value of $X \sim Po(\lambda)$.

2.4 Coupon Collector Problem

In this script this problem is presented in a paragraph after the presentation of the geometric distribution. However, this problem is so important that I think it's worth being subject of an entire section. Consider that you have to complete a collection of n items $a_0, \ldots, a_{n-1}$. At each iteration, you get an object u.a.r from the collection of objects. The Coupon Collector Problem studies the random variable X that describes the number of iterations until the collection is completed. The key idea to approach the situation is to break it down into phases: phase i describes the random variable to collect $(i-1)$ items. Let X_i be the number of iterations in phase i. We have $X = \sum_{i=1}^{n} X_i$. We observe that phase i ends, when we get one of the $n - i + 1$ items that we don't have yet. Hence X_i is a geometric random variable with parameter $\frac{n-i+1}{n}$ and $\mathbb{E}[X_i] = \frac{n}{n-i+1}$. We can now compute the expected value of the Coupon Collector Problem.

$$\mathbb{E}[X] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{n}{n-i+1} = n \sum_{i=1}^{n} \frac{1}{i} = n \cdot H_n = n \ln n + \mathcal{O}(n)$$

Exercise 13. Consider the coupon collector of n elements starting with already $n/2$ elements. Compute the expected number of rounds until completion.

2.5 Important Inequalities

In the script you have seen an important example that shows that the expected value describes the mean of the results of several repetitions of a random experiment. However, if we consider a single realization of the random experiment, this value could be quite far from the expected

value in some (but not all) cases. This fact motivates us to introduce three useful inequalities.

Theorem 2.5.1 (Markov's Inequality) *Let X be a random variable that takes only non-negative values. Then, for all $t \in \mathbb{R}_+$, we have*

$$Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$$

Proof. We have

$$\begin{aligned}\mathbb{E}[X] &= \sum_{\omega \in \Omega} X(\omega) Pr[\omega] \\ &\geq \sum_{\omega \in \Omega, X(\omega) \geq t} X(\omega) Pr[\omega] \\ &\geq t \sum_{\omega \in \Omega, X(\omega) \geq t} Pr[\omega] = t \cdot Pr[X \geq t]\end{aligned}$$

□

Theorem 2.5.2 (Chebyshev's Inequality) *Let X be a random variable and $t \in \mathbb{R}_+$. We have*

$$Pr[|X - \mathbb{E}[X]| \geq t] \leq \frac{Var[X]}{t^2}$$

Proof. We have $Pr[|X - \mathbb{E}[X]| \geq t] = Pr\left[(X - \mathbb{E}[X])^2 \geq t^2\right]$. Since the latter is a non-negative random variable we can apply Markov's inequality and get

$$Pr[|X - \mathbb{E}[X]| \geq t] = Pr\left[(X - \mathbb{E}[X])^2 \geq t^2\right] \leq \frac{\mathbb{E}\left[X - \mathbb{E}[X])^2\right]}{t^2} = \frac{Var[X]}{t^2}$$

□

Informally, this estimates the probability that the realization of a random variable will differ too much from the expected value. This is the probability that $X \in [\mathbb{E}[X] - t, \mathbb{E}[X] + t]$. An upper bound is given, that depends on the variance (the larger the variance, the more likely that X will lie at the extremes) and t (the bigger, the less likely it is to vary by that amount).

We conclude this section by stating without proof, three other important inequalities known as Chernoff's bounds. The inequality of Chernoff gives a usually much more precise result than Markov and Chebyshev. This is because the previous two inequalities work with any non-negative random variable, whereas Chernoff works only for the sum of independent Bernoulli variables with the same parameter p.

Theorem 2.5.3 *If we have $X_1, \ldots, X_n$ iid $\sim Ber(p)$ then it holds for $X =$*

$\sum_{i=1}^{n} X_i$ *and every* $\delta \in]0,1]$

$$P[X \geq (1+\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{3}\delta^2\mathbb{E}[X]}$$
$$P[X \leq (1-\delta)\mathbb{E}[X]] \leq e^{-\frac{1}{2}\delta^2\mathbb{E}[X]}$$
$$P[X \geq t] \leq 2^{-t} \text{ for } t \geq 2e\mathbb{E}[X]$$

Exercise 14. Show by an example that non-negativeness of the random variable is necessary to use the Markov's inequality.

Exercise 15. Consider a test with 18 questions and two possible answers to each question. If one guesses each answer, what is the probability of answering correctly to between 5 and 13 answers? Use Chebyshev's inequality.

2.6 Solutions

Solution 1. By the law of total probability the answer is

$$P + (1-p) \cdot 0.25$$

Solution 2. Yes, it is better to accept the offer. Let A be the event "you chose the door with the car" and let B the event "by changing door you win the car". We have

$$Pr[B] = Pr[B|A] \cdot Pr[A] + Pr\left[B|\bar{A}\right] \cdot Pr\left[\bar{A}\right] = 0 + 1 \cdot \frac{2}{3} = \frac{2}{3}$$

You can see this result by enumerating the sample space and looking at what happens.

Solution 3. By the law of total probability we have

$$\begin{aligned} Pr[A] &= Pr[A|B] \cdot Pr[B] + Pr\left[A|\bar{B}\right] \cdot Pr\left[\bar{B}\right] \\ &= Pr[A|B] \cdot (Pr[B] + Pr\left[\bar{B}\right]) \\ &= Pr[A|B] \end{aligned}$$

which is a sufficient condition for independence.

Solution 4. We have the following bet: we thrown fair coin until we get an head. If the needed number of tosses is odd, we win 2^k, otherwise we lose 2^k. Let X be the random variable for the expected win. If we plug the data in the formula for the expected value we get

$$\sum_{k=1}^{\infty}(-1)^{k-1} \cdot 2^k \cdot \left(\frac{1}{2}\right)^k = 1 - 1 + 1 - 1 + \ldots$$

which does not converge.

Solution 5. We have $2 \cdot \binom{n}{2}$ tuples of nodes with the same color. We define a random variable X_i for each of this tuples. The expected number of edges between nodes of the same color is

$$\mathbb{E}\left[\sum_{i=1}^{2\binom{n}{2}} X_i\right] = \sum_{i=1}^{2\cdot\binom{n}{2}} \mathbb{E}[X_i] = 2 \cdot \binom{n}{2} \cdot \frac{1}{2} = \binom{n}{2}$$

Solution 6. We define Y as the random variable for the requested quantity. We then define Y_k as the length to get the k-th number after we already got a sequence with the $k-1$ numbers. Y_k is geometric distributed with parameter 0.1. Since $Y = \sum_{i=0}^{9} Y_k$, we get

$$\mathbb{E}\left[\sum_{i=0}^{9} Y_k\right] = \sum_{i=0}^{9} \mathbb{E}[Y_k] = 100$$

Solution 7. We define the indicator random variable S for the number of victories in 15 matches. S is binomial distributed with parameters 15 and 0.6. We have $X = 3S + (15 - S) = 2S + 15$. By using the properties of expected value and variance we get

$$\begin{aligned} \mathbb{E}[2S + 15] &= 2\mathbb{E}[S] + 15 = 2 \cdot 15 \cdot 0.6 + 15 = 33 \\ Var[2S + 15] &= 4Var[S] = 14.4 \end{aligned}$$

Solution 8. We have

$$\begin{aligned} \mathbb{E}[X] &= 4 \\ Var[X] &= 33 \\ \mathbb{E}[2X + 8] &= 16 \\ Var[2X + 8] &= 132 \end{aligned}$$

Solution 9. We compute

$$\mathbb{E}\left[X^2\right] = \mathbb{E}[X] = p$$

Hence we have

$$Var[X] = \mathbb{E}\left[X^2\right] - \mathbb{E}[X]^2 = p - p^2 = p(1 - p)$$

Solution 10. We get

$$\begin{aligned}\mathbb{E}[X] &= \sum_{i=0}^{\infty} i \cdot p(1-p)^i = p(1 + 2(1-p) + 3(1-p)^2 + \cdots + k(1-p)^{k-1}) \\ &= p((1 + (1-p) + (1-p)^2 + \dots) + ((1-p) + (1-p)^2 + \dots) + ((1-p)^2 + (1-p)^3 + \dots) + \dots) \\ &= p(\frac{1}{p} + \frac{(1-p)}{p} + \frac{(1-p)^2}{p} + \dots) \\ &= (1 + (1-p) + (1-p)^2 + \dots) \\ &= \frac{1}{p}\end{aligned}$$

Solution 11. We have

$$\begin{aligned}\mathbb{E}[X] &= \sum_{k=0}^{\infty} k\frac{\lambda^k}{k!}e^{-\lambda} \\ &= \sum_{k=1}^{\infty} k\frac{\lambda^k}{k!}e^{-\lambda} \\ &= \sum_{k=1}^{\infty} \frac{\lambda^k}{(k-1)!}e^{-\lambda} \\ &= \sum_{k=1}^{\infty} \lambda\frac{\lambda^{k-1}}{(k-1)!}e^{-\lambda} \\ &= \lambda\sum_{k=0}^{\infty} \frac{\lambda^k}{k!}e^{-\lambda} = \lambda\end{aligned}$$

Solution 12. We introduce a random variable X_i that describes the number of repetitions for the i-th success. X_i is a geometric distributed with parameter p and $X = \sum_{i=1}^{n} X_i$. We have

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_{i=1}^{n} \frac{1}{p} = \frac{n}{p}$$

Solution 13. We have

$$\sum_{i=n/2}^{n} \frac{n}{n-(i-1)} = n\sum_{i=1}^{n/2+1} \frac{1}{i} = n(H_{n/2+1} + \mathcal{O}(1))$$

Solution 14. Consider the random variable X that has $Pr[X = -2] = 0.5 = Pr[X = 2]$. If we would apply the Markov's inequality we would get for example

$$Pr[X \geq 1] \leq \mathbb{E}[X] = 0$$

But this of course does not hold since X takes value 2 with probability 0.5.

Solution 15. We have a binomial random variable X with parameters 18

and 0.5. We have $\mathbb{E}[X] = 9$ and $Var[X] = 4.5$. We can apply Chebyshev's inequality and we get

$$Pr[5 \leq X \leq 13] = 1 - Pr[|X - \mathbb{E}[X]| \geq 5] \geq 1 - \frac{4.5}{25} = \frac{41}{50}$$

Randomized Algorithms 3

In the course *Algorithms and Data Structures* and in the first chapter of this course you have encountered many algorithms: Karatsuba's algorithm, dynamic programming algorithms, merge sort, DFS, Kruskal's algorithm... All this algorithms are *deterministic*, *i.e.* given an input they always return the same output. Here we study *randomized* algorithms, *i.e.* algorithms that may return different outputs in different executions. A more formal approach to define randomized algorithms (which is not absolutely essential for the purpose of this course) could be: randomized algorithm are deterministic algorithms if we consider that their input consists not only of the data relative to the problem, but also of an (infinite long) sequence of random bits. A more intuitive definition could be that randomized algorithms have access to the (pseudo) random number generator of Java, and hence you can generate samples from a distribution of your choice. Randomized algorithms are beautiful. First, they are often very elegant and their analysis is often very clean. Second, they are powerful: some problems that are very difficult to solve, such as the NP-complete longest path problem, can be efficiently approximated via randomized algorithms (at least for short longest paths).

We distinguish two classes of randomized algorithms: Monte Carlo algorithms and Las Vegas algorithms. Monte Carlo algorithms have a deterministic complexity, but they can return wrong answers. Las Vegas algorithms can either return "'???" or the correct answer, but their complexity is randomized. Of course, we consider useful only Monte Carlo algorithms that return the right answer with high probability (*e.g.* just guessing the result would be a very fast Monte Carlo algorithm, but this would be completely useless) and Las Vegas algorithms that, in expectation, don't take too much time to return the correct answer. We point out that, in their most wild form, Las Vegas algorithms don't stop running until they have an answer. Since we can not run a program forever, we use stopping criteria, *e.g.* if the algorithm does not have an answer in an hour we return "???". So, when a Las Vegas algorithm returns "???" it either means that there is no solution (and hence the algorithm without stopping criterion would have never stopped) or that there is a solution but the algorithm has not found it yet.

Exercise 16. Which of the following Monte Carlo algorithms are useful for a problem that requires a binary answer?

- An algorithm that returns the correct answer with probability 0.6
- An algorithm that returns the correct answer with probability 0.5
- An algorithm that returns the correct answer with probability 0.4

Exercise 17. Does the answer to the previous exercise change if we consider a maximization problem?

Exercise 18. How can we transform a Monte Carlo algorithm to a Las Vegas algorithm? What about the opposite direction?

3.1 Success Probability Amplification

In this section we explore some techniques useful to design Las Vegas and Monte Carlo algorithms with certain properties. .

Las Vegas Algorithms

We know that a Las Vegas algorithm either returns the correct solution or "???". We assume that the probability that the Las Vegas algorithm returns the correct solution is at least ε. How many times should we repeat the algorithm in order to get the correct answer with probability at least $1 - \delta$? If we repeat the algorithm N times, the probability to always get "???"is at most

$$(1 - \varepsilon)^N \leq e^{-\varepsilon \cdot N}$$

So we just fix our goal $e^{-\varepsilon \cdot N} \leq \delta$ and we obtain $N \geq \varepsilon^{-1} \ln(\delta^{-1})$.

Monte Carlo Algorithms: One Sided Errors

Here we consider Monte Carlo algorithms for decisional problems. Some algorithms return always YES, if the answer is YES and either YES or NO when the answer is NO. In this scenario, if we get a NO we are sure that the answer is correct; if we get a YES we remain with the doubt. We assume that the probability of saying NO when the answer is NO is at least ε. In order to get a good error probability, we do the following: we repeat the algorithm N times and as soon we get an answer NO we return NO. If we get N times YES we return YES. How big should N be in order to have an error probability of at most δ? Our algorithm is wrong if we have a NO instance and the algorithm returns N YES in a row. Such probability is bounded by

$$(1 - \varepsilon)^N \leq e^{-\varepsilon \cdot N}$$

Similarly as before, we set our goal $e^{-\varepsilon \cdot N} \leq \delta$ and we get $N \geq \varepsilon^{-1} \ln(\delta^{-1})$.

Monte Carlo Algorithms: Two Sided Errors

Some Monte Carlo algorithms for decisional problems does not have the property above. They just return the correct answer with probability $\frac{1}{2} \pm \varepsilon$ for $\varepsilon > 0$ (if the algorithm is more likely to fail, we just have to flip its answer). An algorithm to amplify the success probability is the following: we run the algorithm N times and we return the majority of the answers. As before, we have to choose N large enough in order to have confidence of our answer. The calculations in this scenario are a little bit more complicated than before (they can be derived with Chernoff Bound as you have seen in class). Here we just state that in order to have an error probability of at most δ, we have to choose $N \geq 4\varepsilon^{-2} \ln(\delta^{-1})$.

Monte Carlo Algorithms: Optimisation Problems

Monte Carlo algorithms can be useful also in the context of maximization (or minimization) problems. Assume that we have an algorithm that returns the optimal solution with probability at least ε. The strategy is to repeat the algorithm N times and keep the best solution. We want to choose N such that the probability to return a sub-optimal solution is at most δ. We return a sub-optimal solution if all N repetitions of the algorithm fail. This probability is at most

$$(1 - \varepsilon)^N \leq e^{-\varepsilon \cdot N}$$

Hence, in order to have a success probability of at least $1 - \delta$, we choose $N \geq \varepsilon^{-1} \ln(\delta^{-1})$.

3.2 Target Shooting

The goal of this section is to analyse an algorithm to compute $\frac{|S|}{|U|}$, where S and U are finite sets with $S \subseteq U$. Here we assume that we can generate elements $u \in U$ u.a.r and that we can efficiently compute an indicator function $I_S(u)$ which, given an element $u \in U$, tells us whether u is in S or not. Popular examples of this algorithms include computing the area of an object in a geographic map. The algorithm that we use is simple: we sample N elements from U and we use the samples with the test function to estimate the desired ration. Formally

```
1 Choose u_1, ..., u_N u.a.r fro U
2 return (1/N) Σ_{i=1}^{N} I_S(u_i)
```

Algorithm 3.1: Target-Shooting

It is intuitively clear that this value approximates $\frac{|S|}{|U|}$ and that a larger value N leads to a better result. We want to give quantitative arguments to this idea and give criteria for the choice of the parameter N.

Theorem 3.2.1 *The expected value of the returned result is equal to the true ratio* $\frac{|S|}{|U|}$.

Proof. We define a variable Y_i for each sample u_i. Y_i is bernoulli distributed with parameter $\frac{|S|}{|U|}$. We return $\frac{1}{N}\sum_{i=1}^{N} Y_i$, which has expected value $\frac{|S|}{|U|}$. □

The variance of the same random variable is $\frac{1}{N}(\frac{|S|}{|U|} - (\frac{|S|}{|U|})^2)$: this tells us that larger value of N lead to an approximation closer to the expected value, and hence closer to the true result. We want to find a value of N such that

$$Pr\left[|Y - \frac{|S|}{|U|}| \leq \varepsilon \frac{|S|}{|U|}\right] \geq 1 - \delta$$

for arbitrary $\varepsilon, \delta > 0$. The answer is given by the following theorem.

Theorem 3.2.2 *Let $\delta, \varepsilon > 0$. If $N \geq 3\frac{|U|}{|S|} \cdot \varepsilon^{-2} \cdot \log(\frac{2}{\delta})$, then the output of the Target-Shooting algorithm is with probability $1 - \delta$ in the interval $\left[(1-\varepsilon)\frac{|S|}{|U|}, (1+\varepsilon)\frac{|S|}{|U|}\right]$*

Proof. We define a variable $Z = \sum_{i=1}^{n} Y_i$ with Y_i defined as above. We have $\mathbb{E}[Z] = N\frac{|S|}{|U|}$. We need to find an N such that

$$Pr\left[|Z - \mathbb{E}[Z]| \geq \varepsilon \cdot \mathbb{E}[Z]\right] \leq \delta$$

Since Z is the sum of independent Bernoulli variables we can use Chernoff Bound and get

$$Pr\left[|Z - \mathbb{E}[Z]| \geq \varepsilon \cdot \mathbb{E}[Z]\right] \leq 2e^{-\varepsilon^2 \mathbb{E}[Z]/3} = 2e^{-\varepsilon^2 N \frac{|S|}{3|U|}}$$

With a choice $N = 3\frac{|U|}{|S|} \cdot \varepsilon^{-2} \cdot \log(2/\delta)$ the probability is at most δ. □

Exercise 19. Design a Target-Shooting algorithm to approximate π with the help of a random number generator that generates only numbers in the interval $[0, 1]$.

3.3 Long Paths

In the course *Algorithms and Data Structures* you have seen multiple algorithms to compute shortest paths. At this point a natural question is: can we modify those algorithms in order to compute *longest paths*? We start by looking at a special case, directed acyclic graphs. Computing longest paths in this situation is easy: we just multiply all the weights by minus one and we use the tools from the previous semester. For most graphs, this transformation is not useful because it creates cycles of negative length in the graph. But in the case of directed acyclic graphs, then no negative cycles can be created.

In the general case, however, this problem is NP complete. We want to prove this by showing a reduction to the Hamiltonian cycle problem. Before we do that, we introduce the decisional variant of the problem: given a graph G and a natural number B, decide whether there is a path of length B in G. We observe that if we can solve the decisional variant in polynomial time, we can also solve the longest path problem with a "binary search" approach. The plan to show that the long path problem is NP complete is the following: we create a graph G' such that G has an Hamiltonian cycle if and only if G' contains a path of length n (where n is the number of vertices in G). We construct G' as follows. First, we choose an arbitrary vertex v in G and we remove it from the graph. Instead of removing also the incident edges to v, we substitute the edges $\{v, w_1\}, \ldots, \{v, w_{deg(c)}\}$ with edges $\{\bar{w}_1, w_1\}, \ldots, \{\bar{w}_{deg(v)}, w_{deg(v)}\}$, where $\bar{w}_1, \ldots, \bar{w}_{deg(v)}$ are new nodes. We want to show that G has an Hamiltonian cycle if and only if G' has a path of length n. For the first direction we let $< v_1, \ldots, v_n >$ be an Hamiltonian cycle in G. Without loss of generality, we let $v = v_1$ be the node that we removed in the construction of G'. Then $< \bar{v}_2, v_2, \ldots, v_n, \bar{v}_n >$ is a path of length n in G'. For the other direction we let $< u_0, \ldots, u_n >$ be a path of length n in G'. We observe, that the nodes $u_1, \ldots, u_{n-1}$ must have degree at least two, and hence they must be the nodes of G that we didn't remove. It naturally follows that u_0 and u_n are two of the new nodes that we introduced in the construction of G'. Hence $< v, u_1, \ldots, u_{n-1}, v >$ is an Hamiltonian cycle in G. It is easy to see that creating G' from G can be done in polynomial time, hence we have shown a polynomial reduction from the long path problem to the (presumably) hard Hamiltonian cycle problem.

We have shown that the long path problem is NP complete and hence it is at least plausible to think that no polynomial algorithm exists for this task. But this does not mean that we have to give up completely. For example, in biology, longest paths in graphs are often relatively short. Hence we want to solve the decisional problem for small B, concretely for $B \in \mathcal{O}(\log n)$. In order to solve this task we first introduce another problem, the colorful-path problem, and then we show how an algorithm for it can be used as subroutine for a randomized algorithm for the long path problem with small B.

Colorful-Path Problem

Let $k \in \mathbb{N}$. We color a graph $G = (V, E)$ with the function $\gamma : V \to [k]$ (where γ describes an arbitrary coloring, *e.g.* neighbours can get the same color). We say that a path is called *colored* if all its vertices have different color. We now define the colourful-path problem: given a graph G and a color function γ, decide whether there is a colored path of length $k-1$ in G colored with γ. In order to determine, whether the colored graph contains a colored path of length $k-1$, we define the following quantity

$$P_i(v) := \{S \in \binom{[k]}{i+1} | \exists \text{ a colored path that ends in } v \text{ with the colors in } S\}$$

Of course, it hold that

- $\forall S \in P_i(v) : \gamma(v) \in S$
- $P_0(v) = \{\{\gamma(v)\}\}$
- $P_1(v) = \{\{\gamma(x), \gamma(v)\} | x \in N(v), \gamma(x) \neq \gamma(v)\}$

We observe that there is a colored path of length $k-1$ in G if and only if $\cup_{v \in V} P_{k-1}(v) \neq \varnothing$. Hence, to solve the colorful-path problem, we compute $P_{k-1}(v)$ for all vertices v and we check, whether there is one of such paths or not. The next question is how to compute $P_{k-1}(v)$ for a given vertex. Since we have that

$$P_i(v) = \cup_{x \in N(v)} \{R \cup \{\gamma(v)\} | R \in P_{i-1}(x) \text{ and } \gamma(v) \notin R\}$$

and we have a trivial base case for $i = 0$, we can implement use a dynamic programming approach with increasing value of i.

Algorithm 3.2: ColoredPath(G, i)

```
1 for all v ∈ V
2     for all x ∈ N(v)
3         for all R ∈ P_{i-1}(x) with γ(v) ∉ R
4             P_i(v) ← P_i(v) ∪ {R ∪ {γ(v)}}
```

Algorithm 3.3: ShortLongPath(G)

```
1 for all v ∈ V
2     P_0(v) ← {{γ(v)}}
3 for i = 1, ..., k − 1
4     P_0(v) ← {{γ(v)}}
5 return ∪_{v∈V} P_{k-1}(v) ≠ ∅
```

Since ColoredPath(G,i) has complexity

$$\mathcal{O}\left(\sum_{v \in V} deg(v) \cdot \binom{k}{i} \cdot i\right) \in \mathcal{O}\left(\binom{k}{i} \cdot i \cdot m\right)$$

our final algorithm has complexity

$$\mathcal{O}\left(|V| + \sum_{i=1}^{k-1} \left(\binom{k}{i} \cdot i \cdot m\right) + |V|\right) = \mathcal{O}(2^k km)$$

which is polynomial if $k \in \mathcal{O}(\log n)$.

Short Long Path

Now that we have a deterministic algorithm for the colorful-path problem (which is polynomial for $k = \mathcal{O}(\log n)$), we go back to our original problem of determining whether G contains a path of length B. We use the following Monte Carlo algorithm:

1. Set $k = B + 1$ and color G randomly with k colors.
2. Use the algorithm of the previous section to find a colored path with k vertices. Repeat this operation $\lambda \cdot e^k$ times.
3. If at least one of the repetition of the previous step found a colored path with k vertices, we have a long path of such length. Otherwise we return that no such path exists.

The runtime of the Monte Carlo algorithm is simply given by multiplying the complexity of the algorithm of the previous section with the number of repetitions. We get

$$\mathcal{O}\left(\lambda \cdot e^k \cdot 2^k k m\right)$$

which is polynomial if $k = \mathcal{O}(\log n)$. For the success probability we observe the following:

- If there is no path with k vertices, the success probability is one.
- If there is a path with k vertices, we fail only if in every iteration our random coloring is unlucky. The probability of coloring the path of length k with k different colors is $\frac{k!}{k^k} \geq e^{-k}$.

With this argument, it is easy to see that the failure probability is at most $(1 - e^{-k})^{\lambda \cdot e^k} \leq (e^{e^{-k}})^{\lambda \cdot e^k} \leq e^{-\lambda}$.

3.4 Min Cut

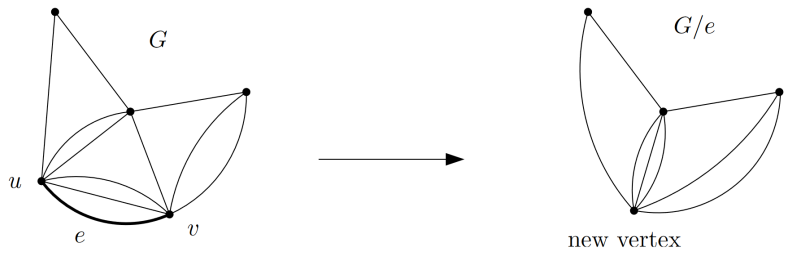
In this section we consider undirected graphs $G = (V, E)$ without loops, but we allow multiple edges of weight one between the same pair of nodes. Such graphs are often known as *multigraphs*. It would be possible, instead of considering multiple edges with weight one, to allow simple edges of positive weight, but with multigraphs everything is much more neat. We say that a set $C \subseteq E$ is a cut in G if $(V, E \backslash C)$ is not connected. We are interested in finding a cut in G of minimum cardinality, a so called *min cut*. With $\mu(G)$ we denote the size of a minimum cut in G (note that the minimum cut does not have to be unique).

Some important facts

Throughout this section we repeatedly use an operation called *edge contraction*. Let G be a multigraph and let $e = \{u, v\}$ be an edge of G. The contraction of e means that we *glue* u and v together into a single new vertex and we remove loops that may have arisen in this way (multiple edges are retained). The resulting graph is denoted by $G \backslash e$. This is illustrated in the following figure:

Every contraction reduces the number of nodes by exactly one and the number of edges by at least one.

Lemma 3.4.1 *Let G be a multigraph and e an edge of G. Then $\mu(G \backslash e) \geq \mu(G)$. Moreover, if there exists a minimum cut C in G such that $e \notin C$, then*



$\mu(G\backslash e) = \mu(G)$.

Lemma 3.4.2 *Let $G = (V, E)$ be a multigraph with n vertices. Then the probability of $\mu(G) = \mu(G\backslash e)$ for a randomly chosen edge $e \in E$ is at least $1 - \frac{2}{n}$.*

Proof. Let C be a min cut in G of size k (*i.e.* we have $k = |C| = \mu(G)$). We observe that the degree of an arbitrary node in G is at least k (otherwise the min cut would have a cardinality less than k). Hence we have: $2|E| = \sum_{v \in V} deg(v) \geq kn$. By applying this formula and Lemma 3.4.1 we get:

$$Pr\left[\mu(G\backslash e) = \mu(G)\right] \geq Pr\left[e \notin C\right] = 1 - \frac{|C|}{|E|} \geq 1 - \frac{k}{k\frac{n}{2}} = 1 - \frac{2}{n}$$

□

Basic Version

In this section we assume that the input graph is connected. We assume that the graph is represented in a way such that we can:

- perform an edge contraction in $\mathcal{O}(n)$
- choose an edge uniformly at random among all edges of the current multigraph in $\mathcal{O}(n)$
- find the number of edges connecting two given vertices in $\mathcal{O}(1)$

The idea is very simple: we consider a graph with two vertices as base case (*i.e.* in this case we count the number of edges between the two vertices and we return the size of the min cut). If we have more than two vertices we repeatedly choose a random edge of the current graph and we contract it, until only two vertices are left.

Algorithm 3.4: BasicMinCut(G)

```
1 while G has more than two vertices do
2     e ← u.a.r. edge in G
3     G ← G\e
4 return size of cut in G
```

The runtime of this algorithm is $\mathcal{O}(n^2)$. This is based on the following observations:

- We execute the body of the while loop $\mathcal{O}(n)$ times.

- Both operations of the while loop take $\mathcal{O}(n)$.

By using the observations of Lemma 3.4.1, we see that this algorithm always returns a number at least as large as $\mu(G)$. If C is a minimum cut in the input graph G, and if we never contract an edge of C during the whole algorithm, then the returned number is exactly $\mu(G)$. At first sight it looks foolish to hope that no edge of C is ever contracted. After all, to this end, the random choice would have to come out *right* (avoid C) in each of the $n-2$ steps. Common sense suggest that making $n-2$ successful random choices in a row is extremely unlikely. The beautiful insight is that in the considered case, a sequence of such *right* choices, while somehow unlikely, is not *extremely* unlikely.

In general we note that the algorithm is correct if:

- $\mu(G) = \mu(G \backslash e)$ for the first contracted edge e
- BasicMinCut succeeds for $G \backslash e$

By applying Lemma 3.4.2 we get the following recurrence for the success probability $p(n)$:

$$p(n) \geq \left(1 - \frac{2}{n}\right) \cdot p(n-1)$$

from which we get:

$$p(n) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{2}{4} \cdot \frac{1}{3} \cdot \underbrace{p(2)}_{=1} = \frac{2}{n(n-1)}$$

In order to get an algorithm with an arbitrary good success probability we use a Monte Carlo approach by running BasicMinCut N times and return the smallest cut size found in all these runs. If the returned size is not correct, it means that the algorithm failed N times in a row. The failures in different runs are independent, and hence the probability of N failures in a row is bounded by:

$$\left(1 - \frac{2}{n(n-1)}\right)^N \leq e^{-\frac{2N}{n(n-1)}}$$

where we used the inequality $1 + x \leq e^x$. If we set, for example, $N = 10n(n-1)$, then the failure probability is bounded above by e^{-20} (which is very small). By increasing the number of repetitions N, the failure probability can be further decreased. Altogether, by setting $N = c \cdot n^2$we have an algorithm with runtime $\mathcal{O}(n^4)$ (because we have $\mathcal{O}(n^2)$ repetitions of the $\mathcal{O}(n^2)$ algorithm) and arbitrary small constant probability of failure.

Bootstrapping

The probability of error in BasicMinCut increases with the number of contractions (*i.e.* with the first contraction we have a probability error of $\frac{2}{n}$ which is quite small, but for example the last iteration has an error probability of $\frac{2}{3}$ which is very large). Why should we take the risk of contracting edges until we have only two vertices? It is better to do less contractions (let's say, until the graph has t nodes, where t has to be chosen carefully) and then going on in another way which is perhaps slower but has a better success probability. What algorithms can we use to calculate the min cut when we have reached the threshold t? The following are two possibilities:

- The deterministic network max flow approach with runtime $\mathcal{O}(n^4 \log(n))$
- The Monte Carlo algorithm we have developed in the previous section with runtime $\mathcal{O}(n^4)$ and success probability $1 - e^{-20}$

The plan is the following: first we explicitly show how a single *layer* of bootstrapping can help to reduce the complexity of the algorithm, then we show a more abstract argument that shows that we can get a limit algorithm with complexity arbitrarily closed to $\mathcal{O}(n^2)$.

First we do a single *layer* of bootstrapping, *i.e.* we contract the edges until a threshold t and then we use the Monte Carlo approach of the previous section to get the answer (*i.e.* we don't take the risk of just contracting edges until we have two vertices but we use something more involved). We get the following algorithm:

Algorithm 3.5: BootstrappingMinCut(G)

```
1 while G has more than t vertices do
2     e ← u.a.r. edge in G
3     G ← G\e
4 return size of cut in G in O(t^4) and success probability 1 − e^{−20}
```

The runtime of a single repetition of the algorithm is $\mathcal{O}(n(n-t)+t^4)$. Now we investigate the success probability (again, the algorithm is successful if we never contract a *wrong* edge and if the algorithm we use when the threshold t is reached returns the correct answer). We get:

$$p(n) \geq \frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdot \frac{n-4}{n-2} \cdots \frac{t+1}{t+3} \cdot \frac{t}{t+2} \cdot \frac{t-1}{t+1} \cdot (1-e^{-20}) = \frac{t(t-1)}{n(n-1)}(1-e^{-20})$$

By using the same Monte Carlo approach we have seen for BasicMinCut, we can do N iterations of BootstrappingMinCut in order to get a good success probability. The algorithm fails if all N repetitions fail. This is bounded by:

$$\left(1 - \frac{t(t-1)}{n(n-1)}(1-e^{-20})\right)^N \leq e^{-N \cdot \frac{t(t-1)}{n(n-1)}(1-e^{-20})}$$

By choosing $N = \frac{20n(n-1)}{t(t-1)\cdot(1-e^{-20})}$ we get a success probability of $1 - e^{-20}$ (which is very high) and a runtime of $\mathcal{O}(\frac{n^2}{t^2}(n^2 + t^4))$. The success probability is independent on the choice of t, hence we must choose the t which minimizes the runtime of our algorithm, *i.e.* we have to choose the t which minimizes the expression:

$$\frac{n^4}{t^2} + n^2 t^2$$

Here one could take the derivative w.r.t. t and then put it equal to zero in order to find the minimum, but a very useful trick is the observation that we obtain the minimum also if we balance the factors, *i.e.* if we find the t such that $\frac{n^4}{t^2} = n^2 t^2$. By solving the equation we get that $t = n^{\frac{1}{2}}$ is optimal and yields an algorithm with runtime $\mathcal{O}(n^3)$ and success probability $1 - e^{-20}$ (or with an arbitrary higher constant probability with the same asymptotic runtime).

So far so good: we have seen that we can improve the runtime of the Monte Carlo algorithm based on BasicMinCut by contracting the graph to a good threshold and then using the same algorithm for the smaller remaining graph (the idea is that by having a smaller error probability from the threshold t we will have to do less repetitions in order to get a high success probability and hence a better runtime). But what if, instead of using the algorithm with runtime $\mathcal{O}(n^4)$ we now use our new algorithm with the same (very high) success probability but improved runtime $\mathcal{O}(n^3)$? By exploiting this idea we can get an additional *layer* of bootstrapping. Now the question is: by using infinite *layers* of bootstrapping, what runtime can we achieve? We give an answer to this question with the following inductive argument, which suggests that there exist a *limit* algorithm that solves the minimum cut problem.

We claim that there exists a sequence of algorithms $\mathcal{A}_i$ (with $i \geq 0$), such that. for all $i \geq 0$, $\mathcal{A}_i$ finds a minimum cut in time $\mathcal{O}(n^{f(i)})$ with probability at least $\frac{1}{2}$ (which is arbitrary, by probability amplification we can increase it to any constant less than one without increasing the asymptotic runtime of the algorithm), where:

$$f(i) := \begin{cases} 4 & \text{for } i = 0 \\ 4 - \frac{4}{f(i-1)} & \text{otherwise} \end{cases}$$

With our first Monte Carlo approach we have proven the cases for $i = 0$ and $i = 1$. Now we show how to construct A_{i+1} in general and we prove that satisfy the desired properties.

Algorithm 3.6: $\mathcal{A}_{i+1}(G)$

```
1 Set parameters N and t suitably
2 Repeat N times
3     H ← RandomContract(G,t)
4     call A_i(H)
5 return smallest value
```

where:

```
1 while G has more than t vertices do
2     e ← u.a.r. edge in G
3     G ← G\e
4 return G
```

Algorithm 3.7: RandomContract(G,t)

The analysis is very similar to the first *layer* of bootstrapping (in facts, this was a special case). Hence we have (by applying the induction hypothesis that $\mathcal{A}_i$ gives the correct answer with probability at least $\frac{1}{2}$, that $\mathcal{A}_{i+1}$ gives the correct answer with probability at least $\frac{t(t-1)}{2n(n-1)}$. Hence the error probability with N repetitions is bounded by:

$$\left(1 - \frac{t(t-1)}{2n(n-1)}\right)^N \leq e^{-N \cdot \frac{t(t-1)}{2n(n-1)}}$$

by choosing $N = \frac{2n(n-1)}{t(t-1)}$ we get a failure probability of at most $e^{-1} \leq \frac{1}{2}$ as desired and runtime $\mathcal{O}(\frac{n^2}{t^2}(n^2 + t^{f(i)}))$. By balancing both terms we get that with $t = n^{\frac{2}{f(i)}}$ we have the desired runtime of $\mathcal{O}(n^{4-4\frac{4}{f(i)}})$, as claimed. Since $f(i)$ approaches 2 with i approaching ∞ we have that the *limit* algorithm with an infinite amount of bootstrapping layers yields an algorithm of complexity $\mathcal{O}(n^2)$.

3.5 Hashing

A *hash table* is a data structure used to implement an *associative array*, a structure that can map keys to values. A *hash function* $h(k)$ is used to map an arbitrary type of key (string, number, ...) to an index of the array, also called bucket or slot.

Ideally, the hash function assigns to each key an unique value, but often this is not possible: this causes collisions, when two or more keys map to the same index. The better the hash function, the lower the collisions.

The function is used to distribute the entries (key, value) across the array. To compute an index, usually two steps are performed, to ensure the index is within bounds:

$$hash = h(key)$$

$$index = hash \quad \% \quad array\_size$$

Crucial to the performance of a hash function is the load factor, defined as

$$L = \frac{n}{k}$$

where n is the number of entries, k is the table size. The closer to 1 the load factor is, the harder is for the hash function to map to an empty bucket.

Closed Hashing

Just one entry per bucket, address might vary.
Some confusion might arise from these terms: the way I like to think about is is that if the hashing is closed, only one (key, value) pair can be stored per bucket (it's closed, no extra space available). This means that collisions must be resolved in some way, where the address might change (open addressing).

Open Hashing

More entries per hash, address doesn't change
On the other hand, on open hashing more entries can fit in the same bucket, which means they will have the same address (closed addressing). This requires the use of an auxiliary data structure (linked lists, trees, ...). The following drawing might help to visualize these concepts:

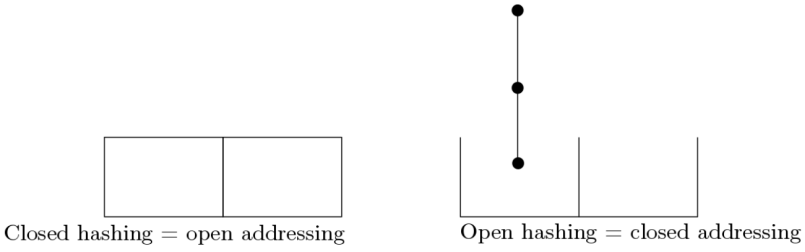


Figure 3.1: Open vs Closed hashing

Collision Resolution

If the hasing is closed, then when a collision occurs we have to find another free bucket where to put our entry. Usually, we try with different offset. If our original hash value is $h(k)$, and our offsets are $o_1, o_2, o_3, \ldots$, then we try the sequence

$$h(k), h(k) - o_1, h(k) + o_1, h(k) - o_2, h(k) + o_2, h(k) - o_3, \ldots$$

A few techniques are available to compute the offsets:

Linear Probing

The idea is to try all adjacent cells in linear order, until we find a free one. The offsets are then simply $1, 2, 3, \ldots$. For example, for a key k, we first try $h(k)$. If it isn't free, then try the others in order. All together this means we try

$$h(k), h(k) - 1, h(k) + 1, h(k) - 2, h(k) + 2, h(k) - 3, \ldots$$

The exact order is implementation dependant, but the main idea is the same.

Quadratic Probing

The idea is somewhat similar to linear probing, but to reduce cluster (many entries close to each other, they all take a lot of time to find a free slot) then each time a collision happens, the probing distance is i^2 instead of i. This means that the offsets are $1^2 = 1, 2^2 = 4, 3^2 = 9, \ldots$. For example, we would try the sequence

$$h(k), h(k) - 1, h(k) + 1, h(k) - 4, h(k) + 4, h(k) - 9, \ldots$$

until either a free entry is found or we run out of possibilities.

Double Hashing

In double hashing, we have a second hash function $h'(k)$ which is used to compute the offset to use for a key. The advantage is that if two keys hash to the same bucket, then a different probing sequence will be used for each of them, thus reducing the number of collisions. Assuming $h'(k) = j$, then we would try the sequence

$$h(k), h(k) - j, h(k) + j, h(k) - 2j, h(k) + 2j, h(k) - 3j, \ldots$$

Cuckoo Hashing

The origin of the name is peculiar: it comes from some species of cuckoo bird, which pushes out the eggs of other males of the nest. In the same way, inserting a new key into a cuckoo hashing table might push a previously inserted key into another position. It works as follows: two hash functions h and h' are used. When a collision occurs, then the previous entry is pushed away and re-hashed with the function h', whereas the newly inserted key takes its place. If another collision happens with the older key, the process is repeated. This might lead to a non terminating amount of swaps, if the initial configuration is reached. To ensure termination, the size of the table might be dynamically resized (e.g. doubling it when a threshold load factor is reached). Another variant is to use two smaller tables, with each hash function mapping to one of them. In the worst case, for both variants each (key, value) pair can be found in only two different location, which bounds the worst case time. This is better than many other variants.

3.6 Smallest Enclosing Circle

Definition 3.6.1 (Smallest Enclosing Circle) *Given a set of points $\mathcal{P}$ in the plane with $|\mathcal{P}| = n$ (i.e. $\mathcal{P} = \{p_1, \ldots, p_n\}, p_i = (x_i, y_i) \in \mathbb{R}^2$) the problem of finding the smallest enclosing circle asks us to find a circle $C(\mathcal{P})$with center (c_x, c_y) and minimum possible radius r such that all points are contained in this circle.*

The following holds:

- points are allowed to be on the boundary

- there exists a subset $Q \subseteq \mathcal{P}$ with $|Q| = 3$ such that $C(\mathcal{P}) = C(Q)$. The points in Q determine C uniquely.

Naive Algorithm

A trivial (and inefficient algorithm) is to simply go over all possible Q, compute the enclosing circle, check if it contains all points, and if so return it. We are guaranteed in this case of having found a minimum circle. However the algorithm takes $O(n^4)$.

```
1 while true
2     iterate over Q ⊆ P, |Q| = 3
3     compute C(Q)
4     if P ⊆ C(Q)
5         return C(Q)
```

Algorithm 3.8: Inefficient smallest enclosing circle

Randomized Algorithm

A better algorithm is to pick points randomly. Every time a point is found to be outside the circle, we know that with higher probability it will be on the border instead of those that are contained. Therefore we increase the probability of picking it in the future by duplicating it.

```
1 while true
2     pick Q ⊆ P, |Q| = 12 uniform at random
3     compute C(Q)
4     if P ⊆ C(Q)
5         return C(Q)
6
7     duplicate all points outside C(Q)
```

Algorithm 3.9: Randomized smallest enclosing circle

The algorithm computes the smallest enclosing circle in expected time $O(n \log n)$.

Note that the number 12 is selected such that in the proof the number of iteration converges.

3.7 Convex Hull

Definition 3.7.1 *In euclidean space, a set S is defined convex if, for every two points p_1, p_2 contained in the set S, all the points on the straight line connecting the two points are also included in the set S. Formally*

$$S \text{ convex} \iff \forall p_1, p_2 \in S, t \in [0,1] : p_1 \cdot t + p_2 \cdot (1-t) \in S$$

Given a set of points P in a d-dimensional space $\mathbb{R}^d$, $|P| = n$, the convex hull H of P is the smallest convex set that contains P.

$$P = \{x_1, \ldots, x_n\}, x_i \in \mathbb{R}^d$$

For the planar case in $2D$ we have $d = 2$, and $P = \{(x_i, y_i)_{i=1..n}\}$.
The convex hull H can also be defined as all the points in space that are a linear combination of the points in the set P, with the condition that the coefficients α assigned to every point must be all positive and sum up to 1. This is the weighted average of positive and normalized weights α_i.

$$H(P) = \left\{ \sum_{i=1}^{n} \alpha_i x_i \mid (\forall i \ \alpha_i \geq 0) \wedge \sum_{i=1}^{n} \alpha_i = 1 \right\}$$

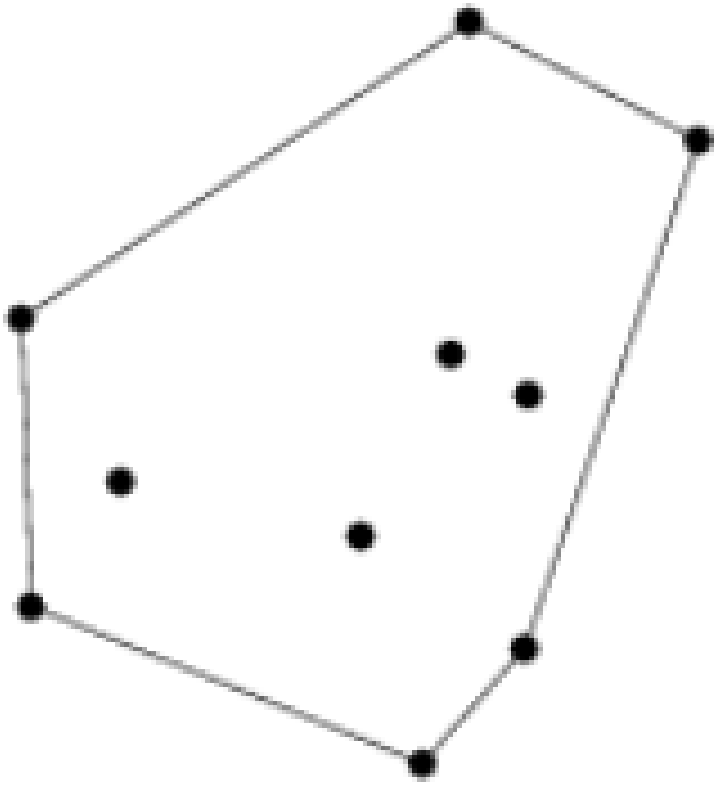


Figure 3.2: Convex hull of a set of points

For simplicity, we here assume that all points are in *general position*, meaning that no three points on the same line and no two points have the same x-coordinate.

Jarvis March

The idea of the algorithm is to start with a point p_0 guaranteed to be on H (e.g. the left-most point) and iteratively looking at all points to find the one that has all other points to its left, which guarantees that it will also be on H. This point can be found in $O(n)$ by iterating through the other points and keeping the one which is the rightmost.

Informally, the algorithm proceeds as follows:

1. Select point p_0 that lies on convex hull (e.g. leftmost)

2. Find next point p_{i+1}, such that all other points lie left of the line going through p_i, p_{i+1}. To do this in $O(n)$, we start with a candidate point and we iterate through all other points. Whenever we find a point on the right of the segment between p_i and the candidate points (this can be computed in constant time with a linear algebra argument), we update the candidate.
3. Repeat until you reach p_0

```
1 c ← leftmost(P)
2 i ← 0
3 repeat
4     H[i++] ← c
5     for j = 1 to n
6     if x_j right of line
7         c ← x_j
8 until c = H[0]
```

Algorithm 3.10: Jarvis March

This algorithm computes the convex hull in $O(n \cdot h)$, where h is the number of points that lie on the convex hull (border). This because the algorithm takes $O(n)$ for each iteration since the angle can be computed in $O(1)$. It requires h loop iterations where $h = |H|$ is the amount of points on the convex hull. This time isn't optimal and there are algorithms that perform much better. Moreover, some modifications are required to address our general position assumption, but the runtime does not change.

3.8 Solutions

Solution 16. An algorithm that returns the correct answer with probability 0.6 is useful (the success probability can be amplificated to an arbitrary threshold by calling the algorithm enough times). The same idea holds for an algorithm that returns the correct answer with probability 0.4: we flip the answer and then we amplificate the probability. The only useless Monte Carlo algorithm is the one that returns the correct answer with probability 0.5: this would be a random guessing and this probability can not be amplificated.

Solution 17. Yes, in fact if we return the correct answer to a maximization problem with probability $p > 0$, we can amplificate it by calling the algorithm k times and returning the maximum. The success probabilty becomes

$$(1-p)^k \leq e^{-pk}$$

which can be arbitrarily small with a proper choice of k. Note that, if k has to be chosen so large that the runtime of k repetitions of the basic algorithm becomes exponential, then our algorithm is very inefficient and usually there is an easy deterministic variant.

Solution 18. Given a Monte Carlo algorithm (fixed runtime, maybe wrong answer) we can do the following to get a Las Vegas algorithm (unfixed runtime, correct answer): we run the Monte Carlo algorithm and we test its solution. If the solution returned by the Monte Carlo algorithm is correct we return the answer and we are over. Otherwise we repeat the process until we have the correct solution.

Given a Las Vegas algorithm we can construct a Monte Carlo algorithm in the following way: we run the Las Vegas algorithm for a fixed amount of time. If the Las Vegas has found a solution we return it (this is the correct solution), otherwise we return a random guess.

Solution 19. We generate N couples of numbers (x_i, y_i) in $[0, 1]$. For every couple we have an hit if $x_i^2 + y_i^2 \leq 1$. We choose N according to the theorem and the obtained result $\frac{|S|}{|U|}$ will approximate $\frac{\pi}{4}$.