SAPIENZA
UNIVERSITÀ DI ROMA

"Sapienza" University of Rome
Faculty of Information Engineering,
Informatics and Statistics
Department of Computer Science

Advanced Algorithms

Lecture notes integrated with the book "Algorithm Design",
J. Kleinberg, É. Tardos

Author
Simone Bianco

March 4, 2025

Contents

Information and Contacts

Personal notes and summaries collected as part of the *Advanced Algorithms* course offered by the degree in Computer Science of the University of Rome "La Sapienza".

Further information and notes can be found at the following link:
https://github.com/Exyss/university-notes. Anyone can feel free to report inaccuracies, improvements or requests through the Issue system provided by GitHub itself or by contacting the author privately:

- Email: **bianco.simone@outlook.it**
- LinkedIn: **Simone Bianco**

The notes are constantly being updated, so please check if the changes have already been made in the most recent version.

Suggested prerequisites:

Sufficient knowledge of computability theory, algorithm complexity, number theory and probability

Licence:

1

Approximation algorithms

1.1 Coping with untractability

In computer science and optimization, approximation algorithms are algorithms designed to find near-optimal solutions to computational problems that are NP-hard, i.e. every problem that is verifiable in polynomial time can be reduced to them. Even though the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ question is still unsolved – which corresponds to determining if every problem efficiently verifiable is also efficiently solvable – lots of theoretical results make us believe that $\mathsf{P} \neq \mathsf{NP}$. Hence, we usually assume that the conjecture has actually been proven as false. This means that every NP-hard problem is **untractable**, meaning that there is no polynomial-time algorithm that can solve them exactly in all cases. Hence, **approximation algorithms** are used to find solutions that are "good enough" or "close enough" to the optimal, with a known error bound, sacrificing exactness for efficiency.

In particular, approximation algorithms are used for **optimization problems**, i.e. every type of problem that asks to find a structure that maximizes or minimizes a property. An approximation algorithm is typically evaluated based on how close its solution is to the optimal solution. The approximation ratio is defined as the worst-case ratio between the cost of the solution produced by the algorithm and the cost of the optimal solution.

For instance, if an algorithm for a minimization problem has an approximation ratio of ρ, then the value ℓ of the approximate solution is guaranteed to be at most ρ times as large as the optimal solution value ℓ^*.

$$\ell \leq \rho\ell^* \implies \rho = \frac{\ell}{\ell^*}$$

For a maximization problem, instead, the solution is guaranteed to be at least ρ times as large as the optimal solution value ℓ^*.

$$\ell \geq \rho\ell^* \implies \rho = \frac{\ell^*}{\ell}$$

Approximation algorithms are used to find near-optimal solutions to NP-hard problems when exact solutions are computationally infeasible. Several techniques are employed to design such algorithms:

- **Greedy Algorithms**: Make locally optimal choices at each step.
- **Dynamic Programming**: Breaks a problem into smaller subproblems.
- **Linear Programming Relaxation**: Converts integer problems to linear ones, solves them, and uses rounding to approximate the solution.
- **Primal-Dual Method**: Simultaneously considers primal and dual solutions to provide approximations.
- **Randomized Algorithms**: Use randomness to find good solutions quickly. *Randomized Rounding* is one such technique, often used with LP relaxation.
- **Local Search**: Iteratively improves a solution by making small changes.
- **Factorization and Decomposition**: Breaks down complex problems into smaller parts.
- **Simulated Annealing**: Probabilistically explores the solution space and escapes local optima.

1.2 Examples of approximating algorithms

1.2.1 The Maximum Cut problem

The **Maximum Cut** problem is a fundamental optimization problem in graph theory and combinatorial optimization. In particular, the problem has numerous practical applications, including in network design, statistical physics (particularly in the study of spin glasses), and in various areas of machine learning, where it is used to model problems such as clustering and data partitioning. Given an undirected graph, the goal of the Max-cut problem is to **partition** the graph's vertices into two disjoint subsets such that the number of edges between the two subsets, i.e. outgoing from one subset to the other, is maximized. This partition is referred to as a **cut**, while the set of edges whose endpoints don't lie in the same subset is called **cut-set**.

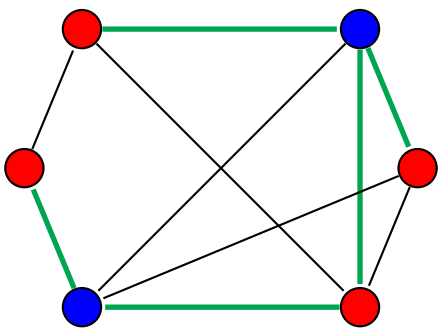

Figure 1.1: The red vertices and the blue vertices form a cut of the graph. The green edges are the edges of the cut-set.

To distinguish between directed and undirected graphs, in the undirected case we'll define the set of edges E of a graph $G = (V, E)$ as $E \subseteq \{\{u, v\} \mid u, v \in V\}$, while in the directed case we have that $E \subseteq \{(u, v) \mid u, v \in V\}$, where (u, v) represents and edge $u \to v$. Instead of an endpoint-based definition, the cut of a graph can also be defined in the following way.

Definition 1: Cut of a graph

Given an undirected graph G, a cut of G is a bipartition (S, T) of G where $T = V - S$. The cut-set of a cut (S, T) is defined as the set $\text{cut}(S, T) = \{e \in E(G) \mid |S \cap e| = 1\}$

The MAXCUT problem is concerned with finding the cut that maximizes the number of edges crossing between the two subsets of the cut (or the total weight of the edges in the cut-set in the weighted case). In particular, we'll focus on the unweighted case of this problem. Unlike its minimization counterpart, i.e. the Min-cut problem, the Max-cut problem is notable for being NP-hard by reduction from the Maximum Independent set problem [GJ90]. The Min-cut problem, instead, is known to lie in P by reduction to the s-t Maximum Cut problem, which is equivalent to the maximum network flow problem.

While finding the optimal solution for the Max-cut problem is computationally intractable for large graphs, significant progress has been made in designing algorithms that can find near-optimal solutions efficiently. One such approach is the famous Goemans-Williamson algorithm, which provides a $(0.878\ldots)$-approximation for MAXCUT, i.e. a solution that has at least a $(0.878\ldots)$-th of the edges of the optimal solution. using semidefinite programming and randomization. We'll see this algorithm in later sections. It is known that there exists a constant $c < 1$ such that there cannot exist any c-approximation algorithm for MAXCUT unless $\mathsf{P} = \mathsf{NP}$ is true [ALM+98], which we assume to be false. For MAXCUT, this constant is known to be as small as $\frac{83}{84} \approx 0.988$.

For now, we'll focus on showing that **randomness** can be used to get a trivial expected $\frac{1}{2}$-approximation of the problem in polynomial time with a sufficiently high probability. This represents the typical case where randomness can be used to get a good enough polynomial time solution with the small trade-off of having a low probability of getting a solution that is below-expectations.

The runtime of this algorithm is clearly $O(n)$ if S is stored using a set data structure. We also notice that the RANDOM-CUT algorithm actually doesn't even care about the graph structure: we're just flipping coins. This idea can be used for many other problems. We now prove that it yields an expected $\frac{1}{2}$-approximation of MAXCUT.

Algorithm 1.1 The random-cut algorithm

Input: an undirected graph G
Output: a cut (S, T) of G

```
1: function RANDOM-CUT(G)
2:     S ← ∅
3:     for v ∈ V(G) do
4:         Flip a fair independent coin and set c_v as the outcome
5:         if c_v = 1 then                          ▷ 1 is heads, 0 is tails
6:             S ← S ∪ {v}
7:         end if
8:     end for
9:     Return (S, V − S)
10: end function
```

Theorem 1

Given a graph G, let (S^*, T^*) be an optimal solution to MAXCUT(G). Given the output (S, T) of RANDOMCUT(G), it holds that:

$$\mathbb{E}[|\text{cut}(S, T)|] \geq \frac{|\text{cut}(S^*, T^*)|}{2}$$

Proof. For any edge $e \in E(G)$, we know that $e \in \text{cut}(S, T)$ if and only if $|S \cap e| = 1$. If $e = \{u, v\}$, this is also also equivalent to saying that $u \in S, v \notin S$ or $u \notin S, v \in S$. We notice that:

$$\begin{aligned}
\Pr[e \in \text{cut}(S, T)] &= \Pr[(u \in S, v \notin S) \vee (u \notin S, v \in S)] \\
&= \Pr[u \in S, v \notin S] + \Pr[u \notin S, v \in S] - \Pr[u \in S, v \notin S, u \notin S, v \in S] \\
&= \frac{1}{4} + \frac{1}{4} + 0
\end{aligned}$$

thus, we get that:

$$\mathbb{E}[|\text{cut}(S, T)|] = \sum_{e \in E(G)} 1 \cdot \Pr[e \in \text{cut}(S, T)] = \frac{|E(G)|}{2}$$

Finally, since each cut-set is by definition a subset of $E(G)$, we know that $|\text{cut}(S^*, T^*)| \leq E(G)$, concluding that:

$$\mathbb{E}[|\text{cut}(S, T)|] = \frac{|E(G)|}{2} \geq \frac{|\text{cut}(S^*, T^*)|}{2}$$

□

On first impact, this algorithm may seem useless: the solution is only *expected* to be a $\frac{1}{2}$-approximation of the optimal maximum cut of the input graph. In fact, if we are very unlucky, the solution could contain all the edges or the graph or even no edges at all.

However, this algorithm is actually enough. In fact, we can show that, by running this algorithm a sufficient amount of times, the probability of getting a bad solution can be highly reduced.

Algorithm 1.2 The t-times random-cut algorithm

Input: an undirected graph G and a non-negative integer t
Output: a cut (S, T) of G

1: **function** t-RANDOM-CUT(G, t)
2: **for** $i \in [\lceil t \rceil]$ **do**
3: $(S_i, T_i) \leftarrow$ RANDOM-CUT(G)
4: **end for**
5: Return $(S, V - S) \in \arg\max_{i \in [\lceil t \rceil]} |\text{cut}(S_i, T_i)|$
6: **end function**

Theorem 2

Given a graph G and a non-negative integer t, let (S^*, T^*) be an optimal solution to MAXCUT(G). Given the output (S, T) of t-RANDOMCUT(G), it holds that:

$$\Pr\left[|\text{cut}(S,T)| > \frac{(1-\varepsilon)}{2}|\text{cut}(S^*,T^*)|\right] > 1-\delta$$

where $t = \frac{2}{\varepsilon}\ln\frac{1}{\delta}$ and $0 < \varepsilon, \delta < 1$.

Proof. For each $i \in [\lceil t \rceil]$, let $C_i = \text{cut}(S_i, T_i)$, where $(S_1, T_1), \ldots, (S_t, T_t)$ are the cuts yielded by the algorithm, and let $N_i = |E(G)| - C_i$. Since N_i is a non-negative random variable, by Markov's inequality we have that:

$$\Pr[N_i \geq (1+\varepsilon)\,\mathbb{E}[N_i]] \leq \frac{1}{1+\varepsilon} = 1 - \frac{\varepsilon}{1+\varepsilon} \leq 1 - \frac{\varepsilon}{2}$$

Through some algebraic manipulation, and by linearity of the expected value operator, we get that:

$$\begin{aligned} 1 - \frac{\varepsilon}{2} &\geq \Pr[N_i \geq (1+\varepsilon)\,\mathbb{E}[N_i]] \\ &= \Pr[|E(G)| - C_i \geq (1+\varepsilon)(|E(G)| - \mathbb{E}[C_i])] \\ &= \Pr[-\varepsilon\,|E(G)| \geq C_i - (1+\varepsilon)\,\mathbb{E}[C_i]] \end{aligned}$$

Using the same argument of the previous theorem, we know that $\mathbb{E}[C_i] = \frac{|E|}{2}$. Hence, we get that:

$$\begin{aligned} 1 - \frac{\varepsilon}{2} &\geq \Pr[-\varepsilon\,|E(G)| \geq C_i - (1+\varepsilon)\,\mathbb{E}[C_i]] \\ &= \Pr[C_i \leq \frac{1-\varepsilon}{2}\,|E|] \\ &= \Pr[C_i \leq (1-\varepsilon)\,\mathbb{E}[C_i]] \end{aligned}$$

We notice that the event of the last probability corresponds to a "bad solution", i.e. one whose value is at most $(1-\varepsilon)$-th of the expected value. Since each run of RANDOM-CUT is independent from the others, the probability of all the solutions being bad is bounded by:

$$\Pr[\forall i \in [t]\ C_i \leq (1-\varepsilon)\,\mathbb{E}[C_i]] = \prod_{i=1}^{\lceil t \rceil} \Pr[C_i \leq (1-\varepsilon)\,\mathbb{E}[C_i]] \leq \left(1-\frac{\varepsilon}{2}\right)^{\lceil t \rceil}$$

Since $0 < 1 - \frac{\varepsilon}{2} < 2$ and $1 - \frac{\varepsilon}{2} \leq e^{-\frac{\varepsilon}{2}}$ (this last fact comes from the definition of e itself), we get that:

$$\Pr[\forall i \in [t]\ C_i \leq (1-\varepsilon)\,\mathbb{E}[C_i]] \leq \left(1-\frac{\varepsilon}{2}\right)^{\lceil t \rceil} \leq \left(1-\frac{\varepsilon}{2}\right)^{t} \leq e^{\frac{\varepsilon}{2}\left(\frac{2}{\varepsilon}\ln\frac{1}{\delta}\right)} = \delta$$

Hence, the probability of at least one solution being good is bounded by:

$$\Pr[\exists i \in [t]\ C_i > (1-\varepsilon)\,\mathbb{E}[C_i]] = 1 - \Pr[\forall i \in [t]\ C_i \leq (1-\varepsilon)\,\mathbb{E}[C_i]] \geq 1-\delta$$

Finally, since the argmax operation inside the t-RANDOM-CUT algorithm will select (in the worst case) such good solution, we conclude that:

$$\begin{aligned}
\Pr\left[|\mathrm{cut}(S,T)| > \frac{1-\varepsilon}{2}\,|\mathrm{cut}(S^*,T^*)|\right] &\geq \Pr[\exists i \in [t]\ C_i > \frac{1-\varepsilon}{2}\,|\mathrm{cut}(S^*,T^*)|] \\
&\geq \Pr[\exists i \in [t]\ C_i > (1-\varepsilon)\,\mathbb{E}[C_i]] \\
&\geq 1-\delta
\end{aligned}$$

□

We observe that the result that we have just proved is very powerful. For instance, by choosing $\varepsilon, \delta = 0.1$, we get that:

$$\Pr\left[|\mathrm{cut}(S,T)| > (0.45)\,|\mathrm{cut}(S^*,T^*)|\right] \geq 0.9$$

and $t \approx 46$, meaning that we have to run RANDOM-CUT approximately 46 times in order to almost certainly get a solution that is better than a (0.45)-approximation. We also notice that notice that $0 < \frac{1-\varepsilon}{2} < 0.5$ since $0 < \varepsilon < 1$, meaning that we will always sacrifice some optimality to boost our probability. This trade-off idea between optimality and probability by running multiple times the same algorithm can be applied to many other problems.

1.2.2 The Minimum Vertex Cover problem

The **Minimum Vertex Cover** problem is a well-known optimization problem in graph theory and combinatorial optimization. It involves finding the smallest subset of vertices in a graph such that every edge is incident to at least one vertex in the set. Like the Max-cut problem, the Minimum Vertex Cover problem is also NP-hard by reduction from the Maximum Clique problem.

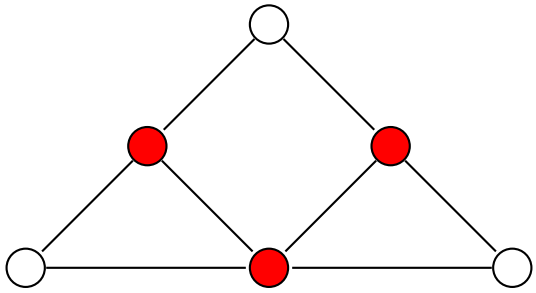

Figure 1.2: The red nodes are the smallest possible vertex cover of the graph.

Definition 2: Vertex Cover

Given an undirected graph G, a vertex cover over G is a subset $C \subseteq V(G)$ such that $\forall e \in E(G)$ there is a vertex $v \in C$ such that $v \in e$.

Before proceeding, it's important to distinguish between the concepts of *minimal* and *minimum*. In general, given a property P, a sub-structure X of a structure S is said to be minimal for P over S if $P(X)$ is true and there is no other sub-structure X' of S such that $P(X')$ is true and X is contained inside X'. Instead, X is said to be the minimum for P over S if $P(X)$ is true and there is no other sub-structure X' of S with a lower value for the property $P(X)$. For instance, a minimal vertex cover is a vertex cover that doesn't contain another vertex cover inside it – meaning that we cannot remove vertices and keep the property true – while a minimum vertex cover is a vertex cover with the lowest possible cardinality. Clearly, a minimum vertex cover is always a minimal one, but the reverse is not always true.

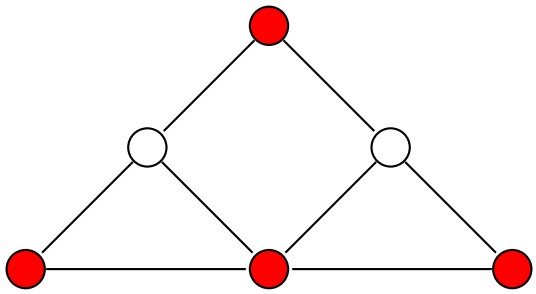

Figure 1.3: The red nodes form a minimal vertex cover of the graph since removing any of them wouldn't preserve the cover property. This vertex cover is not a minimum one.

Vertex covers are highly related to the concept of **matching**. In fact, an approximation

for the Minimum Vertex Cover problem can be achieved through the Maximal Matching problem. A matching over a graph is a subset of edges that share no common endpoint. The difference between maximality and maximum is the same as the one between minimality and minimum.

Definition 3: Matching

Given an undirected graph G, a matching over G is a subset $M \subseteq E(G)$ such that $\forall e, e' \in M$ it holds that

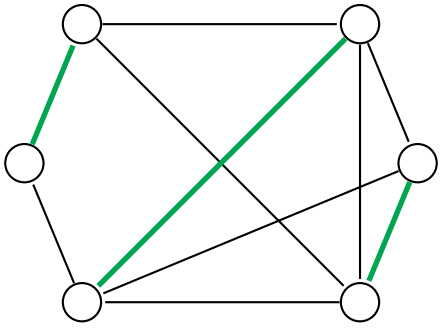

Figure 1.4: The green edges form a maximal matching of the graph.

Clearly, a maximal matching can be constructed in polynomial time through by simply adding edges until the property is preserved. In an even more efficient way, it can be computed by the following algorithm.

Algorithm 1.3 The maximal matching algorithm

Input: an undirected graph G
Output: a maximal matching M of G

```
1:  function MAXIMAL-MATCHING(G)
2:      M ← ∅
3:      E' ← E(G)
4:      while E' ≠ ∅ do
5:          Choose e ∈ E'
6:          S ← S ∪ {e}
7:          E' ← E' − {f ∈ E' | e ∩ f ≠ ∅}
8:      end while
9:      Return M
10: end function
```

We observe that the edges $e_1, \ldots, e_t$ picked by the algorithm are always a maximal matching: if $M = \{e_1, \ldots, e_t\}$ is not maximal then at least one edge could still get picked by the algorithm, meaning that it shouldn't have stopped.

Lemma 1

Let G be an undirected graph. For any matching M of G and any vertex cover C of G it holds that $|S| \geq |M|$.

Proof. By definition, we observe that if C is a vertex cover for G then it is also a vertex cover for $G' = (V, E')$, where $E' \subseteq E(G)$. Hence, C is also a vertex cover for any $G_M = (V, M)$, where M is a matching of G. By definition of matching, in G_M any vertex has either degree 0 or 1. Thus, each vertex of S can cover at most one edge of M, meaning that S has to have at least $|M|$ vertices to cover all the edges of M. □

We observe that the lemma above is valid for any matching and any vertex cover, not only maximal and minimum ones, making it less specific for our situation. Nonetheless, we can use it to show that the following algorithm is actually a 2-approximation of MINVERTCOV.

Algorithm 1.4 2-approximation of MINVERTCOV

Input: an undirected graph G

Output: a vertex cover for G with at most double the minimum vertices.

1: **function** 2-APPROX-VC(G)
2: $M \leftarrow$ MAXIMAL-MATCHING(G)
3: Return $C = \bigcup_{e \in M} e$
4: **end function**

Theorem 3

Given a graph G, let C^* be an optimal solution to MINVERTCOV(G). Given the output C of 2-APPROX-VC(G), it holds that $|C| \leq 2\,|C^*|$.

Proof. Let $M = \{e_1, \ldots, e_t\}$ be the maximal matching returned by MAXIMAL-MATCHING(G). Since $\forall e, e' \in M$ it holds that $e \cap e' = \varnothing$ by definition of maximal matching, it holds that $|C| = 2\,|M|$. Hence, since C^* is a vertex cover, by the previous lemma we get that $|C| = 2\,|M| \leq 2\,|C^*|$. □

This result look quite easy, making us believe that this bound can be highly improved. However, it is conjectured that MINVERTCOV may be NP-hard to approximate to any ratio $2 - \varepsilon$ for any constant $\varepsilon > 0$ – the Unique Games Conjecture implies this result, which is conjectured to be true. Hence, this simple approximation algorithm may actually be the best we can achieve.

The vertex cover is also known to lie in the class of **Fixed-parameter Tractable (FPT)** problems, i.e the set of problems that can be solved in time $f(k) \cdot n^{O(1)}$, where f is a computable function and k is a fixed input parameter. We observe that, since k is fixed, the value $f(k)$ becomes a "constant", making the running time polynomial with respect to the size of the input. The crucial part of the definition is to exclude functions of the form $f(k, n)$, such as k^n.

In the particular case of the MINVERTCOV problem, the fixed parameter k corresponds to the size of the vertex cover to be found. If k is fixed, a simple back-tracking algorithm allows us to solve the decision version of the Minimum Vertex Cover problem, i.e. the set $VC = \{\langle G, k\rangle \mid \exists C \subseteq V(G) \text{ s.t. } C \text{ is a V.C. with } |C| \leq k\}$, in time $O(2^k n)$.

Algorithm 1.5 Membership in the set VC

Input: an undirected graph G an a non-negative integer k
Output: True if $\langle G, k\rangle \in VC$, false otherwise

```
1:  function VC-BACKTRACKING(G, k)
2:      if E(G) ≠ ∅ then
3:          Return True
4:      else if k = 0 then
5:          Return False
6:      else
7:          Choose {u, v} ∈ E(G)
8:          if VC(G[V − {u}]), k − 1 then
9:              Return True
10:         end if
11:         if VC(G[V − {v}]), k − 1 then
12:             Return True
13:         end if
14:         Return False
15:     end if
16: end function
```

Here, the notation $G[V - \{u\}]$ (and $G[V - \{v\}]$) corresponds to the **induced subgraph** by $V - \{u\}$ on G, i.e. the graph obtained by removing the vertices in $V - \{u\}$ from G and all the edges that had one of such vertices as endpoints.

Definition 4: Induced subgraph

Given a graph G and a subset $S \subseteq V(G)$, the subgraph induced by S on G is the graph $G[S] = (S, E')$ such that $E' = \{e \in E(G) \mid S \cap e \neq \varnothing\}$.

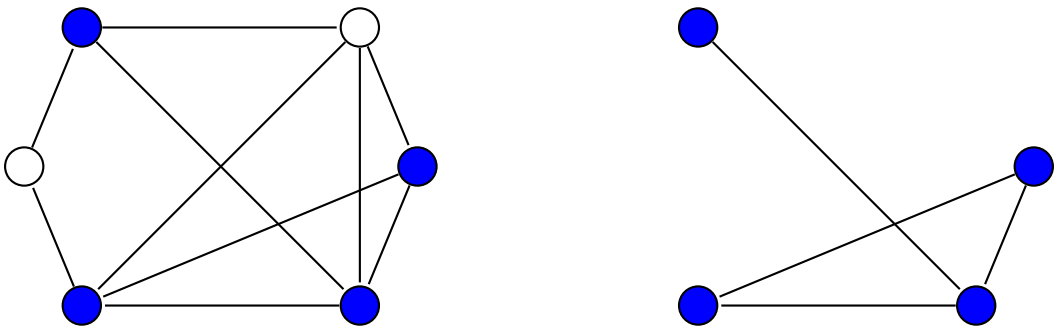

Figure 1.5: The original graph (left) and the subgraph induced by the blue vertices (right).

To analyze the cost of the previous algorithm, we observe that the computational cost $T(n, m, k)$, where n is the number of nodes and m is the number of edges, is upper bounded by

$$T(n, m, k) \leq 2T(n, m, k-1) + O(n+m)$$

where $T(n, m, 0) = T(n, 0, k) = \Theta(1)$. Hence, we get that $T(n, m, k) = O(2^k(n+m))$. To get a more precise computational cost, we observe that if a graph G has a vertex cover C with k vertices then $|E(G)| \leq (n-1)k$ since in the worst case each of the k vertices has degree $n-1$. Since $O(nk) = O(n)$ when k is fixed, we get that $T(n, m, k) = O(2^k n)$.

1.3 Mathematical programming

Mathematical Programming involves using mathematical models and optimization techniques to solve problems that require finding the best solution from a set of possible choices, subject to constraints. It plays a key role in areas like operations research, artificial intelligence, machine learning, and systems design. In mathematical programming, an optimization problem is typically expressed as:

$$\text{Minimize (or Maximize)} \quad f(x)$$

subject to:

$$g_i(x) \leq b_i \quad \forall i \in [m]$$

$$x \ldots 0$$

Here, x is a vector of decision variables inside the vector space $\mathbb{Q}^n$ (or $\mathbb{R}^n$), $f(x)$ is an objective function and $g_i(x) \leq b_i$ represent inequality constraints. More specifically, we'll focus on **Linear Programming (LP)** and **Semi-Definite Programming (SDP)**. In the former, both the objective function and constraints are linear with respect to $\mathbb{Q}^n$ (or $\mathbb{R}^n$).

$$\begin{aligned} \max\, & x_1 + x_2 \\ & x_1 + 6x_2 \leq 15 \\ & 4x_1 - x_2 \leq 10 \\ & x_2 - x_1 \leq 1 \\ & x \geq 0 \\ & x \in \mathbb{R}^n \end{aligned}$$

Figure 1.6: Example of a linear program.

Linear programs can be solved in polynomial time. In particular, if a linear program has n variables, m constraints and each coefficient is representable as the ratio of two t-bits integers then the LP can be solved through the *Ellipsoid method* in time $O((nmt)^c)$ for some $c > 0$. Even though it is theoretically guaranteed to have polynomial time, the Ellipsoid method becomes useful only for very large inputs. For more practical cases, the

Simplex method is used, which is based on *pivot rules*. All of the pivot rules known for the Simplex method have a theoretical exponential lower bound through some particular programs that "fool" the rule, but they have an average complexity that is way better than the Ellipsoid method.

Bibliography

[ALM+98] Sanjeev Arora, Carsten Lund, Rajeev Motwani, et al. "Proof verification and the hardness of approximation problems". In: *J. ACM* (1998). DOI: 10.1145/278298.278306.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990. ISBN: 0716710455.