Advanced Algorithms and Datastructures - Full Notes

André O. Andersen

2021

Max Flow

General Knowledge

In the maximum-flow problem, we wish to compute the greatest rate at which we can ship material from the source to the sink without violating any capacity constraints.

A ***flow network*** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v) \geq 0$. We further require that if E contains an edge (u, v), then here is no edge (v, u) in the reverse direction. If $(u, v) \notin E$, then we define $c(u, v) = 0$, and we disallow self-loops. We distinguish two vertices in a flow network a ***source*** s and a ***sink*** t. For convenience, we assume that each vertex lies on some path from the source to the sink.

A ***flow*** in G is a real-valued function $f : V \times V \rightarrow \mathbb{R}$ taht satisfies the following two properties:

1. **Capacity constraint**: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$
2. For all $u \in V - \{s, t\}$, we require
$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$
When $(u, v) \notin E$ there can be no flow from u to v, hence $f(u, v) = 0$

We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v. The ***value*** $|f|$ of a flow f is defined as
$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s).$$

In the ***maximum-flow problem***, we are given a flow network G with source s and t, and we wish to maximize $|f|$ such the capacity constraint and the flow conservation are not violated.

We call the two edges (v_1, v_2) and (v_2, v_1) ***antiparallel***. If we wish to model a flow problem with antiparallel adges, we must transform the network into an equivalent one containing no antiparallel edges. This is done by choosing one of the two antiparallel edges, in this case (v_1, v_2), and split it by adding a new vertex v' and replacing edge (v_1, v_2) with the pair of edges (v_1, v') and (v', v_2). We also set the capacity of both new edges to the capacity of the original edge. The resulting network satisfies the property that if an edge is in the network, the reverse edge is not.

A maximum-flow problem may have several sources and sinks, rather than just one of each. Fortunately, this problem is no harder than ordinary maximum flow. We can reduce the problem of determining a maximum flow in a network with multiple sources and multiple sinks to an ordinary maximum-flow problem. We add a ***supersource*** s and add a directed edge (s, s_i) with capacity $c(s, s_i) = \infty$ for each $i = 1, 2, ..., m$. We also create a new ***supersink*** t and add a directed edge (t_i, t) with capacity $c(t_i, t) = \infty$ for each $i = 1, 2, ..., n$, where m and n are the amount of sources and sinks in the original graph, respectively.

The Ford-Fulkerson method is a method for solving the maximum-flow problem. It iteratively increases the value of the flow. We start with $f(u, v) = 0$ for all $u, v \in V$, giving an initial flow of value 0. At each iteration, we increase the flow value in g by finding and "augmenting path" (a path from the source to sink in the residual network) in an associated "residual network" G_f. Once we know the edges of an augmenting path in G_f, we can easily identify specific edges in G for which we can change the flow so that we increase the value of the flow. We repeatedly change the flow until the residual network has no more augmenting paths. The max-flow min-cut theorem will show that upon termination, this process yields a maximum flow.

The residual network G_f consists of edges with capacities that represent how we can change the flow on edges of G. An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. If that value if positive, we place that edge into G_f with a "residual capacity" of $c_f(u, v) = c(u, v) - f(u, v)$. The only edge in G that are in G_f are those that can admit more flow.

The residual network G_f may also contain edges that are not in G. In order to represent a possible decrease of a positive flow $f(u,v)$ on an edge in G, we place an edge (v,u) into G_f with residual capacity $c_f(v,u) = f(u,v)$ - that is, an edge that can admit flow in the opposite direction to (u,v), at most conaceling out the flow on (u,v).

The ***residual capaicty*** $c_f(u,v)$ is thus

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

Because of our assumption that $(u,v) \in E$ implies $(v,u) \notin E$, exactly one case in equation (1) applies to each ordered pair of vertices.

Given a flow network $G = (V,E)$ and a flow f, the ***residual network*** of G induced by f is $G_f = (V, E_f)$, where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}.$$

That is, each edge of the residual network, or ***residual edge***, can admit a flow that is greater than 0. The edge in E_f are either edges in E or their reversals, and thus $|E_f| \leq 2|E|$.

A flow in a residual network pro vides a roadmap for adding flow to the original flow network. If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the ***augmentation*** (or "change") of flow f by f', to be a functio nfrom $V \times V$ to $\mathbb{R}$, defined by

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) & \textit{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

An ***augmenting path*** p is a simple path from s to t in the residual network G_f. We call the maximum amount by which we can increase the flow on e ach edge in an augmenting path p the ***residual capacity*** of p, given by

$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{is on } p\}.$$

Lemma 26.2
Let $G = (V,E)$ be a flow network, let f be a flow in G and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

$$f_p(u,v) = \begin{cases} c_f(p) & \text{if } (u,v) \text{ is on } p \\ 0 & \textit{otherwise.} \end{cases} \tag{3}$$

Then, f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$.

A ***cut*** (S,T) of flow network $G = (V,E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. If f is a flow, then the ***net flow*** $f(S,T)$ across the cut (S,T) is defiend to be

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u). \tag{4}$$

The ***capacity*** of the cut (S, T) is

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v).$$

A ***minimum cut*** of a network is a cut whose capacity is minimum over all cuts of the network.

The running time of Ford-Fulkerson depends on how we find the augmenting path p. If we choose it poorly, the algorithm might not even terminate - the ford-fulkerson method might fail to terminate only if edge capacities are irriational numbers. If we assume that the capacities are integers, then Ford-Fulkerson has a running time of $O(E \cdot |f^*|)$, where f^* is a max flow. This is because the capacities are integers, we know Ford-Fulkerson increases the value by atleast 1 at each step, making the runtime be bounded by the max flow ($|f^*|$) and E is used for finding an augmenting path.

We can improve the bound on Ford-Fulkerson by finding the augmenting path p with a breadth-first search. That is, we choose the augmenting path as a *shortest* path from s to t in the residual network, where each edge has unit distance (weight). This is called the ***Edmonds-Karp Algorithm*** and it runs in $O(VE^2)$.

Because we can implement each iteration of Ford-Fulkerson in $O(E)$ time when we find the augmenting path by breadth-first search and the total number of flow augmentations is $O(VE)$, the total running time of the Edmonds-Karp algorithm is $O(VE^2)$.

Examples

Maximum bipartite matching

Given an undirected graph $G = (V, E)$, a ***matching*** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is ***matched*** by the matching M if some edge in M is incident on v; otherwise, v is ***unmatched***. A ***maximum matching*** is amatching of maximum cardinality. In this section, we shall restrict our attention to finding maximum matchings in bipartite graphs: graphs in which the vertex set can be partitioned into $V = L \cup R$, where L and R are disjoint and all edges in E go between L and R. We further assume that every vertex in V has at least one incident edge.

We can use the Ford-Fulkerso nmethod to find a maximum matching in an undirected bipartite graph $G = (V, E)$ in time polynomial in $|V|$ and $|E|$. The trick is to construct a flow network in which flows correspond to matchings. We define the ***corresponding flow network*** $G' = (V', E')$ for the bipartite graph G as follows. We let the source s and sink t be new vertices not in V, $V' = V \cup \{s, t\}$, and

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in R\}.$$

To complete the construction, we assign unit capacity to each edge in E'. Since each vertex in v has at least one incident edge, $|E| \geq |V|/2$. Thus, $|E| \leq |E'| = |E| + |V| \leq 3|E|$, and so $|E'| = \Theta(E)$.

Proofs

Lemma 26.1

Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Proof We first verify that $f \uparrow f'$ obeys the capacity constraint for each edge in E and flow conservation at each vertex in $V - \{s, t\}$.
For the capacity constraint, first observe that if $(u, v) \in E$, then $c_f(v, u) = f(u, v)$. Therfore, we have $f'(v, u) \leq c_f(v, u) = f(u, v)$, and hence

$$\begin{aligned}
(f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) && \text{(by equation (2))} \\
&\geq f(u, v) + f'(u, v) - f(u, v) && \text{(because } f'(v, u) \leq f(u, v)\text{)} \\
&= f'(u, v) \\
&\geq 0.
\end{aligned}$$

In addition,

$$\begin{aligned}
(f \uparrow f')(u, v) &= f(u, v) + f'(u, v) - f'(v, u) && \text{(by equation (2)} \\
&\leq f(u, v) + f'(u, v) && \text{(because flows are nonegative)} \\
&\leq f(u, v) + c_f(u, v) && \text{(capacity constraint)} \\
&= f(u, v) + c(u, v) - f(u, v) && \text{(definition of } c_f\text{)} \\
&= c(u, v).
\end{aligned}$$

To show that flow conservation holds and that $|f \uparrow f'| = |f| + |f'|$, we first prove the claim that for all $u \in v$, we have

$$\sum_{v \in v} (f \uparrow f')(u, v) - \sum_{v \in V} (f \uparrow f')(v, u) = \sum_{v \in V} f(u, v) - \sum_{v \in V} f(v, u) + \sum_{v \in V} f'(u, v) - \sum_{v \in V} f'(v, u). \tag{5}$$

Because we disallow antiparallel edges in G, we know that for each vertex u, there can be an edge (u, v) or (v, u) in g, but never both. For a fixed vertex u, let's define $V_1(u) = \{v : (u, v) \in E\}$ and $V_2(u) = \{v : (v, u) \in E\}$. By the definition of flow augmentation in euqation (2), only vertices in $V_1(u)$ can have positive $(f \uparrow f')(u, v)$, and only vertices in $V_2(u)$ can have positive $(f \uparrow f')(v, u)$. Thus, we have

$$\begin{aligned}
&\sum_{v \in V} (f \uparrow f')(u, v) - \sum_{v \in V} (f \uparrow f')(v, u) \\
&= \sum_{v \in V_1(u)} (f \uparrow f')(u, v) - \sum_{v \in V_2(u)} (f \uparrow f')(v, u) \\
&= \sum_{v \in V_1(u)} (f(u, v) + f'(u, v) - f'(v, u)) - \sum_{v \in V_2(u)} (f(v, u) + f'(v, u) - f'(u, v)) \\
&= \sum_{v \in V_1(u)} f(u, v) + \sum_{v \in V_1(u)} f'(u,) - \sum_{v \in V_1(u)} f'(v, u) - \sum_{v \in V_2(u)} f(v, u) - \sum_{u \in V_2(u)} f'(v, u) + \sum_{v \in V_2(u)} f'(u, v) \\
&= \sum_{v \in V_1(u)} f(u, v) - \sum_{v \in V_2(u)} f(v, u) + \sum_{v \in V_1(u)} f'(u, v) + \sum_{v \in V_2(u)} f'(u, v) - \sum_{v \in V_1(u)} f'(v, u) - \sum_{v \in V_2(u)} f'(v, u)
\end{aligned}$$

In the last equation we can extend all four summations to sum over v, since each additional term has value 0. With all four summations over V, instead of just subsets of V, we get equation (5).
Now we are ready to prove flow conservation for $f \uparrow f'$ and that $|f \uparrow f'| = |f| + |f'|$. For the latter property, let $u = s$ in equation (5). Then, we have

$$\begin{aligned}
|f \uparrow f'| &= \sum_{v \in V} (f \uparrow f')(s, v) - \sum_{v \in V} (f \uparrow f')(v, s) \\
&= \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) + \sum_{v \in V} f'(s, v) - \sum_{v \in V} f'(v, s) \\
&= |f| + |f'|.
\end{aligned}$$

For flow conservation, observe that for any vertex u that is neither s not t, flow conservation for f and f' means that the right-hand side of equation (5) is 0, and thus $\sum_{v \in V} (f \uparrow f')(u, v) = \sum_{v \in V} (f \uparrow f')(v, u)$.

Corollary 26.3

Let $G = (V, E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (3), and suppose that we augment f by f_p. then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.
Proof Immediate from Lemmas 26.1 and 26.2.

Lemma 26.4

Let f be a flow in a flow network G with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$.
Proof
We extend the definition of net flow from cuts to arbitrary pairs of subsets of V as follows. For any $A \subseteq V$ and $B \subseteq V$,

$$f(A, B) = \sum_{u \in A} \sum_{v \in B} f(u, v) - f(v, u).$$

Now we proceed with the proof. The definition of net flow above immediately implies $f(S, S) = 0$, since the we are at some point adding $f(u, v)$, however, at some point u and v switches place, hence why we subtract $f(u, v)$ again. Thus, we have

$$f(S, T) = f(S, S) + f(S, T) = f(S, V) = \sum_{u \in S} \sum_{v \in V} (f(u, v) - f(v, u)).$$

On the right-hand side, every $u \in S \backslash \{s\}$ contributes 0 to the sum due to flow conservation. Thus,

$$f(S, T) = \sum_{u \in \{s\}} \sum_{v \in \{V\}} (f(u, v) - f(v, u)) = \sum_{v \in V} (f(s, v) - f(v, s)) = |f|.$$

Corollary 26.5

The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G. ***Proof*** Let (S, T) be any cut of G and let f be any flow. By Lemma 26.4 and the capacity constraint,

$$\begin{aligned}
|f| &= f(S,T) \\
&= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&\leq \sum_{u \in S} \sum_{v \in T} f(u,v) \\
&\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\
&= c(S,T).
\end{aligned}$$

Theorem 26.6 (Max-flow min-cut theorem)

If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a maximum flow in G
2. The residual network G_f contains no augmenting paths
3. $|f| = c(S,T)$ for some cut (S,T) of G.

Proof
(1) ⇒ (2): Suppose for the sake of contradiction that f is a maximum flw in G but that G_f has an augmenting path p. Then, by Corollary 26.3, the flow found by augmenting f by f_p, where f_p is given by equation (3), is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.
(2) ⇒ (3): Suppose that G_f has no augmenting path, that is, that G_f contains no path from s to t. Define

$$S = \{v \in V : \text{there is a path from } s\text{to } vinG_f\}$$

and $T = V - S$. The parition (S,T) is a cut: we have $s \in S$ trivially and $t \notin S$ because there is no path from s to t in G_f. Now consider a pair of vertices $u \in S$ and $v \in T$. If $(u,v) \in E$, we must have $f(u,v) = c(u,v)$, since otherwise $(u,v) \in E_f$, which would place v in set S. If $(v,u) \in E$, we must have $f(v,u) = 0$, because otherwise $c_f(u,v) = f(v,u)$ would be positive and we we ould have $(u,v) \in E_f$, which would place v in S. Of course, if neither (u,v) nor (v,u) is in E, htne $f(u,v) = f(v,u) = 0$. We thus have

$$\begin{aligned}
f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{v \in T} \sum_{u \in S} f(v,u) \\
&= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{v \in T} \sum_{u \in S} 0 \\
&= c(S,T).
\end{aligned}$$

By Lemma 26.4 therefore $|f| = f(S,T) = c(S,T)$.
(3) ⇒ (1): By Corollary 26.5, $|f| \leq c(S,T)$ for all cuts (S,T). The condition $|f| = c(S,T)$ thus implies that f is a maximum flow.

Lemma 6.7

If the Edmonds-Karp algorithm is run on a flow network $G = (V,E)$ with source s and sink t, then for all verties $v \in V - \{s,t\}$, the shortest-path distance $\delta_f(s,v)$ in the residual network G_f increases monotonically with each flow augmentation.
Proof Let f be the flow just before the first augmentation that decreases some shortest-path distance, and let f' be the flow just afterward. Let v be the vertex with minimum $\delta_{f'}(s,v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s,v) < \delta_f(s,v)$. Let $p = s \rightsquigarrow u \rightarrow v$ be a shortest path from s to v in $G_{f'}$, so that $(u,v) \in E_{f'}$, and

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1. \tag{6}$$

Because of how we chose v, we know that the distance of vertex u from the source s did not no decrease, i.e.,

$$\delta_{f'}(s,u) \geq \delta_f(s,u). \tag{7}$$

We claim that $(u,v) \notin E$. Why? If we had $(u,v) \in E_f$, then we would also have

$$\begin{aligned}
\delta_f(s,v) &\leq \delta_f(s,u) + 1 && \text{(by lemma 24.10, the triangle inequality)} \\
&\leq \delta_{f'}(s,u) + 1\text{(by inequality (7))} \\
&= \delta_{f'}(s,v)\text{(by equation (6)}
\end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$.
How can we have $(u,v) \notin E_f$ and $(u,v) \in E_f$? The augmentation must ha ve increased the flow from v to u. the Edmonds-Karp algorithm always augments flow along shortest paths, and therefore it augmented along a shortest path from s to u in G_f that has (v,u) as its last edge. Therefore,

$$\begin{aligned}
\delta_f(s,v) &= \delta_f(s,u) - 1 \\
&\leq \delta_{f'}(s,u) - 1 && \text{(by inequality (7))} \\
&= \delta_{f'}(s,v) - 2 && \text{(by equation (6))}
\end{aligned}$$

which contradicts our assumption that $\delta_{f'}(s,v) < \delta_f(s,v)$. We conclude that our assumption that such a vertex v exists is incorrect.

Theorem 26.8

If the Edmonds-Karp algorithm is run on a flow network $G = (V,E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.
Proof We say that an edge (u,v) i na residual network G_f is ***critical***on an augmenting path p if the residual capacity of p is the residual capacity of (u,v). After we have augmented flow along an augmenting path, anycritical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical.
Let u and v be vertices in V that are connected by an edge in E. Since augmenting paths are shortest paths, when $(u, v9$ is critical for the first time, we have

$$\delta_f(s,v) = \delta_f(s,u) + 1.$$

Once the flow is augmented, the edge (u,v) disappears from the residual network. It cannot reappear later on another augmenting path until after the flow from u to v is decreased, which occurs only if (v,u) appear on an augmenting path. If f' is the flow in G when this event occurs, then we have

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1.$$

Since $\delta_f(s,v) \leq \delta_{f'}(s,v)$ by lemma 26.7, we have

$$\begin{aligned}
\delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1 \\
&\geq \delta_f(s,v) + 1 \\
&= \delta_f(s,u) + 2.
\end{aligned}$$

Consequently, from the time (u, v) become critical to the time when it next becomes critical, the distance of u from the source increases by at least 2- The distance of u from the source is initially at least 0. The intermediate vertices on a shortest path from s to u cannot contain s, u or t. Therefore, until u becomes unreachable from the source, if ever, its distance is at most $|V| - 2$. Thus, after the first time that (u, v) becomes critical, it can become critical at most $(|V| - 2)/2 = |V|/2 - 1$ times more, for a total of at most $|V|/2$ times. Since there are $O(E)$ pair of vertices that can have anedge between them in a residual network, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is $O(VE)$. Each augmenting path has at least one critical edge, and hence the theorem follows.

Lemma 26.9

Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let $G' = (V', E')$ be its corresponding flow network. If M is amatching in G, then there is an integer-valued flow f in G' with value $|f| = |M|$. Conversely, if f is an integer-valued flow in G', then there is a matching M in G with cardinality $|M| = |f|$.

Proof We first show that a matching M in G corresponds to an integer-valued flow f in G'. Define f as follows. If $(u, v) \in M$, then $f(s, u) = f(u, v) = f(v, t) = 1$. For all other edges $(u, v) \in E'$, we define $f(u, v) = 0$. It is simple to verify that f satisfies the capacity constraint and flow conservation.

Intuitively, each edge $(u, v) \in M$ corresponds to one uit of flow in G' that traverses the path $s \to u \to v \to t$. Moreover, the paths induced by edges in M are vertex-disjoint, except for s and t. The net flow across cut $(L \cup \{s\}, R \cup \{t\})$ is equal to $|M|$; thus the value of the flow is $|f| = |M|$.

To prove the converse, let f be an integer-valued flow in G', and let

$$M = \{(u, v) : U \in L, v \in R, \text{ and } f(u, v) > 0\}.$$

Each vertex $u \in L$ has only one entering edge and its capacity is 1. Thus, each $u \in L$ has t most one unit of flow entering it, and if one unit of flow does enter, by flow conservation, one unit of flow must leave. Furthermore, since f is integer-valued, for each $u \in L$, the one unit of flow can enter on at most one edge and can leave on at most one edge. Thus, one unit of flow enters u iff there is exactly one vertex $v \in R$ such that $f(u, v) = 1$, and at most one edge leaving each $u \in L$ carries positive flow. A symmetric argument applies to each $v \in R$. The set M is therefore at matching.

To see that $|M| = |f|$, observe that for every matched vertex $u \in L$, we have $f(s, u) = 1$, and for every edge $(u, v) \in E - M$, we have $f(u, v) = 0$. Consequently, $f(L \cup \{s\}, R \cup \{t\})$, the net flow across cut $(L \cup \{s\}, R \cup \{t\})$, is equal to $|M|$. Thus, we have $|f| = f(L \cup \{s\}, R \cup \{t\}) = |M|$.

Linear Programming and Optimization

General Knowledge

Many problems take the form of maximizing or minimizing an objective, given limited resources and competing constriants. If we can specify the objective as linear function of certain variables, and if we can specify the constrants on resources as equalities or inequalities on those variables, then we have a ***linear-programming problem***.

In the general linear-programming problem, we wish to optimize a linear function subject to a set of linear inequalities. Given a set of real numbers $a_1, a_2, ..., a_n$ and a set of variables $x_1, x_2, ..., x_n$, we define a ***linear function*** f on those variables by

$$f(x_1, x_2, ..., x_n) = a_1x_1 + a_2x_2 + ... + a_nx_n = \sum_{j=1}^{n} a_jx_j.$$

If b is a real number and f is a linear function, then the equation

$$f(x_1, x_2, ..., x_n2) = b$$

is a ***linear equality*** and the inequalities

$$f(x_1, x_2, ..., x_n) \leq b$$

and

$$f(x_1, x_2, ..., x_n) \geq b$$

are ***linear inequalities***. We use the general term ***linear constraints*** to denote either linear equalities or linear inequalities. In linear programming , we do not allow strict inequalities. Formally, a ***linear-programming problem*** is the problem either minimizing or maximizing a linear function subject to a finite set of linear constraints.

We call any setting of the variables $x_1, x_2, ..., x_m$ that satisfies all the constraints a ***feasible solution*** to the linear program. If we graph the constraints in the $(x_1, ..., x_m)$-Cartesian coordinate system, we see that each constraint corresponds to ahalf-space in m-dimensional space. The intersection of these half-spaces forms the ***feasible region*** and the function we wish to maximize the ***objective function***. We call the feasible region formed by the intersection of these half-spaces a ***simplex***. We call the value of the objective function at a particular point the ***objective value***. A feasible solution whose objective value is maximum over all feasible solutions is an ***optimal objective value***. If a linear program has no feasible solutions, we say that the linear program is ***infeasible***; otherwise it it ***feasible***. If a linear program has some feasible solutions but does not have a finite optimal objective value, we say that the linear program is ***unbounded***.

The ***simplex algorithm*** takes as input a linear program and returns an optimal solution. It starts at some vertex of the simplex and performs a sequence of iterations. In each iteration, it moves along an edge of the simplex from a current vertex to a neighboring vertex whose objective value is no smaller than that of the current vertex. The simplex algorithm terminates when it reaches a local maximum, which is a vertex from which all neighboring vertices have a smaller objective value. Because the feasible region is convex and the objective function is linear, the local optimum is actually a global optimum.

If we add to a linear pgoram the additional requirement that all variables take on integer values, we have an ***integer linear program***. Finding a feasible solution to this problem is NP-hard. In constrast, we can solve a general linear-programming problem in polynomial time.

If we have a linear program with variables $x = (x_1, x_2, ..., x_n)$ and wish to refer to a particular setting of the variables, we shall use the notation $\bar{x} = (\bar{x}_1, \bar{x}_2, ..., \bar{x}_n)$.

In ***standard form***, we are given n real numbers $c_1, c_2, ..., c_n$; m real numbers $b_1, b_2, ..., b_m$; and mn real numbers a_{ij} for $i = 1, 2, ..., m$ and $j = 1, 2, ..., n$. We wish to find n real numbers $x_1, x_2, ..., x_n$ that maximize

$$\sum_{j=1}^{n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq bi \qquad \text{for } i = 1, 2, ..., m$$

$$x_j \geq 0 \qquad \text{for } j = 1, 2, ..., n.$$

where the last constraint are the ***nonnegativity constraints***, which are required in standard form. Sometimes we find it convenient to express a linear program in a more compact form. If we create an $m \times n$ matrix $A = (a_{ij})$, an m-vector $b = (b_i)$, an n-vector $c = (c_j)$, and an n-vector $x = (x_j)$, then we can rewrite the linear program to

maximize

$$c^T x$$

subject to

$$Ax \leq b$$

$$x \geq 0.$$

We see that we can specify a linear program in standard form by a tuple (A, b, c).

A linear program might not be in standard form of any of four possible reasons

1. The objective function might be a minimization rather than a maximization
2. There might be variables without nonnegativity constraints
3. There might be equality constraints, which have an equal sign rather than a less-than-or-equal-to sign
4. There might be inequality constraints, but instead of having a less-than-or-equal-to sign, they have a greater-than-or-equal-to sign.

We say that two maximization linear programs L and L' are ***equivalent*** if for each feasible solution $\bar{x}$ to L with objective value z, there is a corresponding feasible solution $\bar{x}'$ to L' with objective value z, and for each feasible solution $\bar{x}'$ to L' with objective value z, there is a corresponding feasible solution $\bar{x}$ to L with objective value z. A minimization linear program L and a maximization linear program L' are equivalent if for each feasible soltuion $\bar{x}$ to L with objective value z, there is a corresponding feasible solution $\bar{x}'$ to L' with objective value $-z$, and for each feasible solution $\bar{x}'$ to L' with objective value z, there is a corresponding feasible solution $\bar{x}$ to L with objective value $-z$.

(1). To convert a minimization linear program L into an equivalent maximization linear program L', we simply negate the coefficients in the objective function. Since L and L' have identical sets of feasible solutions and, for any feasible solution, the objective value in L is the negative of the objective value in L', these two linear pr ograms are equivalent

(2). Suppose that some variable x_k does not have a nonnegativity constraint. Then, we replace each occurrence of x_j by $x'_j - x''_j$, and add the nonnegativity constraints $x'_j \geq 0$ and $x''_j \geq 0$. Thus, if the objective function has a term $c_j x_j$, we replace it by $c_j x'_j - c_j x''_j$, and if constraint i has a term $a_{ij} x_j$, we replace it by

$a_{ij}x'_j - a_{ij}x''_j$. Any feasible solution $\hat{x}$ to the new linear program corresponds to a feasible solution $\bar{x}$ to the original linear program with $\hat{x}_j = \hat{x}'_j - \hat{x}''_j$ and with the same objective value. Also, any feasible solution $\hat{x}$ to the original linear program corresponds to a feasible solution $\hat{x}$ to the new linea rprogram with $\hat{x}'_j = \bar{x}_j$ and $\hat{x}''_j = 0$ if $\hat{x}_j \geq 0$, or with $\hat{x}''_j = -\bar{x}_j$ and $\hat{x}'_j = 0$ if $\bar{x}_j < 0$. Thus, the two linear programs are equivalent.

(3). Suppose that a linear program has an equality constraint $f(x_1, x_2, ..., x_n) = b$. Since $x = y$ iff both $x \geq y$ and $x \leq y$, we can replace this equality constraint by the pair of inequality constraints $f(x_1, x_2, ..., x_n) \leq b$ and $f(x_1, x_2, ..., x_n) \geq b$.

(4) We can convert the greater-than-or-equal-to constraints to less-than-or-equal-to constraints by multiplying these constraints through by -1. That is, any inequality of the form

$$\sum_{j=1}^{n} a_{ij}x_j \geq b_i$$

is equivalent to

$$\sum_{j=1}^{n} -a_{ij}x_j \leq -b_i.$$

To solve a linear program with the simplex algorithm, we prefer to express it in a form in which the nonegativity constraints are the only inequality constrants, and the remaining constrants are equalities. Let

$$\sum_{j=1}^{n} a_{ij}x_j \leq b_i$$

be an inequality constraint. We introduce a new variable s and rewrite this inequality as the two constraints

$$s = b_i - \sum_{j=1}^{n} a_{ij}x_j$$

$$s \geq 0$$

where s is a ***slack variable***. When converting from standard to slack form, we shall use x_{n+i} (instead of s) to denote the slack variable associated with the ith inequality. The ith constraint is therefore

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_j,$$

along with the nonnegativity constraint $x_{n+i} \geq 0$. We call the variables on the left-hand side of the equalities ***basic variables*** and those on the right-hand side ***nonbasic variables***.

For linear programs that satisfy these conditions, we shall sometimes omit the words "maximize" and "subject to", as well as the explicit nonegativity constraint. We shall also use the variable z to denote the value of the objective function. We call the resulting format ***slack form***.

In order to use the simplex algorithm, we must convert the linear program into slack form. We say that an equality constraint is ***tight*** for a particular setting of its nonbasic variables if they cause the constraint's basic variable to become 0. Similarly, a setting of the nonbasic variables that would make a basic variable become negative ***violates*** that constraint. Thus, the slack variables explicitly maintain how far each constraint is from being tight, and so they help to determine how much we can increase values of nonbasic variables without violating any constraints.

We focus on the ***basic solution***: set all the nonbasic variables on the right-hand side to 0 and then compute the values of the basic variables on the left-hand side.

Our goal, in each iteration, is to reformulate the linear pr ogram so that the basic solution has a grater objective value. We select a nonbasic variable x_e whose coefficient in the objective function is positive, and we increase the value of x_e as much as possible without violating any of the constraints. The variable x_e becomes basic, and some other variable x_l becomes nonbasic. This last operation is called a ***pivot***. A pivot chooses a nonbasic variable x_e, called the ***entering variable***, and a basic variable x_l, called the ***leaving variable***, and exchanges their roles. We perform two operations in the simplex algorithm: rewrite equations so that variables move between the left-hand side and the right-hand side, and substritute one equation into another. This is done iteratively untill all coefficients in the objective function are negative.

Duality enables us to prove that a solution is indeed optimal: given a maximization problem, we define a related minimization problem such that the two problems have the same optimal objective values. Given a linear program in which the objective is to maximize, we shall describe how to formulate a ***dual*** linear program in which the objective is to minimize and whose optimal value is identical to that of the original linear program. When refering to dual linear programs, we call the original linear program the ***primal***.

Given a primal linear program in standard form we define the dual linear program as

$$\begin{aligned} &\text{minimize} \\ &\qquad \sum_{i=1}^{m} b_i y_i \\ &\text{subject to} \\ &\qquad \sum_{i=1}^{m} a_{ij} y_i \geq c_j && \text{for } j = 1, 2, ..., n \\ &\qquad y_i \geq 0 && \text{for } i = 1, 2, ..., m. \end{aligned}$$

To form the dual, we change the maximization to a minimization, exchange the roles of coefficients on the right-hand sides and the objective function, and replace ea ch $\leq$ by a $\geq$. Each of the m constraints in the primal has an associated vairable y_i in the dual, and each of the n constraints in the dual has an associated variable x_j in the primal. For example, consider the following linearprogram

$$\begin{aligned} &\text{maximize} \\ &\qquad 3x_1 + x_2 + 2x_3 \\ &\text{subject to} \\ &\qquad x_1 + x_2 + 3x_2 \leq 30 \\ &\qquad 2x_1 + 2x_2 + 5x_3 \leq 24 \\ &\qquad 4x_1 + x_2 + 2x_3 \leq 36 \\ &\qquad x_1, x_2, x_3 \geq 0 \end{aligned}$$

The dual of this linear program is

$$\begin{aligned} &\text{minimize} \\ &\qquad 30y_1 + 24y_2 + 36y_3 \\ &\text{subject to} \\ &\qquad y_1 + 2y_2 + 4y_3 \geq 3 \\ &\qquad y_1 + 2y_2 + y_3 \geq 1 \\ &\qquad 3y_1 + 5y_2 + 2y_3 \geq 2 \\ &\qquad y_1, y_2, y_3 \geq 0 \end{aligned}$$

The simplex algorithm actually implicitly solves both the primal and the dual linear programs simultaneously, thereby providing aproof op optimality.

Suppose that SIMPLEX returns values $\bar{x} = (\bar{x}_1, \bar{x}_2, ..., \bar{x}_n)$ for a linear program. Let N and B denote the nonbasic and basic variablesr for the final slack form, let c' denote the coefficients in the final slack form, and let $\bar{y} = (\bar{y}_1, \bar{y}_2, ..., \bar{y}_m)$ be defined by

$$\bar{y}_i = \begin{cases} -c'_{n+i} & \text{if } (n+i) \in N, \\ 0 & \text{otherwise.} \end{cases}$$

Then $\bar{x}$ is an optimal solution to the primal linear program, $\bar{y}$ is an optimal solution to the dual linear program, and

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i .$$

Proofs

Lemma 29.8 (Weak linear-programming duality)

Let $\bar{x}$ be any feasible solution to a primal linear program in standard form and let $\bar{y}$ be any feasible solution the corresponding dual linear program. Then, we have

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{i=1}^{m} b_i \bar{y}_i .$$

Proof We have

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j &\leq \sum_{j=1}^{m} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j && \left(\text{Using } \sum_{i=1}^{m} a_{ij} y_i \geq c_j\right) \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i \\
&\leq \sum_{i=1}^{m} b_i \bar{y}_i && \left(\text{Using } \sum_{j=1}^{n} a_{ij} x_j \leq b_i\right)
\end{aligned}$$

Corollary 29.9

Let $\bar{x}$be a feasible solution to aprimal linear program, and let $\bar{y}$ be a feasible solution to the corresponding dual linear program. If

$$\sum_{j=1}^{n} c_j \bar{x}_j = \sum_{i=1}^{m} b_i \bar{y}_i ,$$

then $\bar{x}$ and $\bar{y}$ are optimal solutions to the primal and dual linear programs, respectively.
Proof By Lemma 29.8, the objective value of a feasible solution to the primal cannot exceed that of a feasible solution to the dual. The primal linear program is a maximization problem and the dual is a minimization problem. Thus, if feasible solutions $\bar{x}$ and $\bar{y}$ have the same objective value, neither can be improved.

Randomized Algorithms

General Knowledge

There are two principal advantages to randomized algorithms. The first is performance. Second, many randomized algorithms are simpler to describe and implement than deterministic algorithms of comparable performance.

Let Gbe a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimal cardinality. We now study an algorithm for finding a min-cut of a graph.

We repeat the follwing step: pick an edge uniformly at random and merge the two vertices at its end-points. If as a result there are several edges between some pair of vertices, retain them all. Edges between vertices that are merged are removed, so that there are never any self-loops. We refer to this process of merging the two end-points of an edge into a single-vertex as the *contraction* if that edge. An edge contraction does not reduce the min-cut size in G. This is because every cut in the graph at any intermediate stage is a cut in the original graph.

Suppose we were to repeat the min-cut algorithm $n^2/2$ times, making independent random choices each time. The probability that a min-cut is not found in any of the $n^2/2$ attempts is at most

$$\left(1 - \frac{2}{n^2}\right)^{n^2/2} < \frac{1}{e}.$$

A *Las Vegas algorithm* is an algorithm that always gives the correct solution. The only variation from one run to another is its running time. On the other hand a *Monte Carlo algorithm* is an algorithm that may sometimes produce a solution that is incorrect. If the algorithm is run repeatedly with independent random choices each time, the failure probability can be made arbitrarily small, at the expense of running time. A Monte Carlo algoritm is said to have *two-sided error* if there is a non-zero probability that it errs when it outputs either yes or no. It is said to have one-sided error if the probability that it errs is zero for at least one of the possible outputs that it produces.

We say that a Las V egas algorithm is an *effecient Las Vegas* algorithm if on any input its expected running time is bounded by a polynomial function of the input size. Similarly, we say a montecarlo algorithm is an *efficient Monte Carlo* algorithm if on any input its worst-case running time is bounded by a polynomial function of the input size.

Proofs

Expected runtime of randomized quicksort

The expected number of comparisons performed by randomized qucksort is $O(n \log n)$.
Proof For $1 \leq i \leq n$, let $S_{(i)}$ denote the ith smallest element in the sorted set S. Then, define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and S_j are compared in an execution, and the value 0 otherwise. Thus the total number of comparisons is

$$\sum_{i<j} X_{ij} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Then, we are interested in the expected number of comparisons

$$\mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}\left[X_{ij}\right].$$

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared in an execution. Since X_{ij} only assumes the values 0 and 1, we have

$$\mathbb{E}\left[X_{ij}\right] = p_{ij} \cdot 1 + (1 - p_{ij}) \cdot 0 = p_{ij}.$$

Each recursive call returns some sublist $[S_{(a)}, ..., S_{(b)}]$. Let $x = S_{(c)}$ be the pivot. Suppose $a \leq i < j \leq b$. If $c < i$ or $c > j$, then $S_{(i)}$ and $S_{(j)}$ are not compared now, but are in the same subtree and might be compared later. If $i < c < j$, then $S_{(i)}$ and $S_{(j)}$ are never compared. If $c = i$ or $c = j$ then $S_{(i)}$ and $S_{(j)}$ are compared once, right now. Thus, $S_{(i)}$ and $S_{(j)}$ are compared iff $S_{(i)}$ or $S_{(j)}$ is first of $S_{(s)}, ..., S_{(j)}$ to be chosen as pivot. Hence why p_{ij} is the conditional probability of picking $S_{(i)}$ or $S_{(j)}$ given that the pivot is picked uniformly at random in $S_{(i)}, S_{(i+1)}, ..., S_{(j)}$

$$p_{ij} = \mathbb{P}\left[c \in \{i, j\} | c \in \{i, i+1, ..., j\}\right] = \frac{2}{|\{i, i+1, ..., j\}|} = \frac{2}{j - i + 1}.$$

We thus have

$$\begin{aligned}
\mathbb{E}\left[\sum_{i<j} X_{ij}\right] &= \sum_{i<j} \mathbb{E}\left[X_{ij}\right] && \text{(Linearity of expectation)} \\
&= \sum_{i<j} p_{ij} \\
&= \sum_{i<j} \frac{2}{j - i + 1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1} \frac{2}{j - i + 1} && \text{(Using} \sum_{i<j} = \sum_{i=1}^{n-1} \sum_{j=i+1} \text{)} \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&< \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} && \text{(Adding more positive terms increases total value)} \\
&= 2n \sum_{k=2}^{n} \frac{1}{k} \\
&= 2n \left(\left(\sum_{k=1}^{n} \frac{1}{k} \right) - 1 \right) \\
&= 2n(H_n - 1) && \left(\text{Using } H_n = \sum_{k=1}^{n} \frac{1}{k} \right) \\
&\leq 2n \int_1^n \frac{1}{x} dx = 2n \ln n = O(n \log n) && \text{(Using } H_n - 1 \leq \int_1^n \frac{1}{x} dx = \ln n
\end{aligned}$$

Lower-bound of finding min-cut

The min-cut algorithm has a probability of $\frac{2}{n(n-1)}$ of finding a specific min-cut.
For any vertex v in a multigraph G, the *neighborhood* of v, denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v. The *degree* of v, denoted $d(v)$, is the number of edges incident on v. For a set S of vertices of G, the neighborhood of S, denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

Let k be the min-cut size and $G_i = (V_i, E_i)$ be the graph G after i iterations, which has $n_i = n - i$ vertices. We fix our attention on a particular min-cut C with k edges. Then, G_i has at least $n_i|C|/2$ edges because

$$\frac{1}{2}\sum_{v\in V_i} d_i(v) \geq \frac{1}{2}n_i|C| \quad \text{(Using } d_i(v) \geq |C|\text{)}.$$

We will bound from below the probability that no edge of C is ever contracted during an execution of the algorithm.

Let ϵ_i denote the event of not picking an edge of C at the ith step, for $1 \leq i \leq n-2$. Let $p_i = \mathbb{P}\left[\epsilon_i | \cap_{j=i}^{i-1} \epsilon_j\right]$. The probability of picking an edge of C in the ith iteration, given that no edge of C has been picked in a previous iteration, is thus

$$\begin{aligned}
1 - p_i &= \mathbb{P}\left[\epsilon_i | \cap_{j=i}^{i-1} \epsilon_j\right] \\
&= \mathbb{P}\left[e \in E_{i-1} is in C | \cap_{j=1}^{i-1} \epsilon_j\right] \\
&= \frac{|C|}{|E_{i-1}|} \leq \frac{|C|}{\frac{1}{2}n_{i-1}|C|} \qquad\qquad = \frac{2}{n_{i-1}} = \frac{2}{n-(i-1)} p_i \geq 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}. \qquad \Rightarrow
\end{aligned}$$

Thus, the probability of finding C can then by found by

$$\begin{aligned}
\mathbb{P}\left[C \text{is found}\right] &= \prod_{i=1}^{n-2} p_i \\
&\geq \prod_{i=1}^{n-2} \frac{n-1-i}{n+1-i} \\
&= \frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2}\cdots\frac{3}{5}\frac{2}{4}\frac{1}{3} \\
&= \frac{2}{n(n-1)}.
\end{aligned}$$

Hashing

General Knowledge

We have a large universe U of keys, that we wish to map randomly to a range $[m] = \{0, ..., m-1\}$ of hash values. A *truly random hash function* $h : U \to [m]$ assigns an independent uniformly random variable $h(x)$ to each key in x. Unfortunately truly random hash functions are idealized objects that cannot be implemented.

For a hash function, we care about roughly three things:

1. The size of the random seed that is necessary to calculate $h(x)$ given x
2. The time it takes to calculate $h(x)$ given x
3. Properties of the random variable.

In *universal hashing* we wish to generate a random hash function $h : U \to [m]$ from key universe U to a set of hash values $[m] = \{0, ..., m-1\}$. We think of h as a random variable following some distribution over functions $U \to [m]$. We want h to be *universal*, which means that for any given distinct keys $x, y \in U$, when h is picked at random, we have *low collision probability*, defined by

$$\mathbb{P}[h(x) = h(y)] \leq 1/m.$$

For many applications, it suffices if for some $c = O(1)$, we have

$$\mathbb{P}[h(x) = h(y)] \leq c/m.$$

Then h is called *c-approximately universal.*

One classic application of universal hashing is *hash tables with chaining.* We have a set $S \subseteq U$ of keys that we wish to store so that we can find any key from S in expected constant time. Let $n = |S|$ and $m \geq n$. We now pick a universal hash function $h : U \to [m]$, and then create an array L of m lists/chains so that for $i \in [m]$, $L[i]$ is the list of keys that hash to i. Now to find out if a key $x \in U$ is in S, we only have to check if x is in the list $L[h(x)]$. This takes time proportional to $1 + |L[h(x)]|$ (we add 1 because it takes constant time to look up the list even if it turns out to be empty).

Another application is that of assigning an unique *signature* $s(x)$ to each key. Thus we want $s(x) \neq s(y)$ for all distinct keys $x, y \in S$. To get this, we pick an universal hash function $s : U \to [n^3]$. The probability of an error (collision) is less than $1/2n$, where $n = |S|$ and $S \subseteq U$.

One classic universal hash function is *multiply-mod-prime*, which is based on a prime number $p \geq u$, where $U = [u] = \{0, ..., u-1\}$. We pick an uniformly random $a \in [p]_+ = \{1, ..., p-1\}$ and $b \in [p] = \{0, ..., p-1\}$, and define $h_{a,b} : [u] \to [m]$ by

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m.$$

Another hash function is *Multiply-shift*, which ashes from w-bit integers to l-bit integers. We pick a uniformly random odd w-bit integer a, and the nwe computer $hh_a : [2^w] \to [2^l]$ as

$$h_a(x) = \left\lfloor (ax \bmod 2^w)/2^{w-l} \right\rfloor.$$

A random hash function $h : U \to [m]$ is *strongly universal* (aka. *2-independent*) if

1. Each key is hashed uniformly into $[m]$ (that is, $\forall x \in U, q \in [m] : \mathbb{P}[h(x) = q] = \frac{1}{m}$), and
2. Any two distinct keys hash independently.

Or, equivalently, for all $x \neq y \in U$, and $q, r \in [m]$ we have

$$\mathbb{P}[h(x) = q \wedge h(y) = r] = \frac{1}{m^2}.$$

We say a random hash function $h : U \to [m]$ is *c-approximately strongly universal* if

1. h is c-approximately uniform, and
2. Every pair of distinct keys hash independently.

One application of strongly universal hashing is *coordinated sampling.* The basic idea is that we based on small samples can reason about the similarity of huge sets.

First we consider sampling from a single set $A \subseteq U$ using a strongly universal hash function $h : U \to [m]$ and a threshold $t \in \{0, ..., m\}$. We now sample x if $h(x) < t$, which happens with probability t/m for any x. Let $S(A) = \{x \in A | h(x) < t\}$ denote the resulting sample from A. Then, we have $\mathbb{E}[|S(A)|] = |A| \cdot t/m$. Conversely, this means that if we have $S(A)$, then we can estimate $|A|$ as $|S(A)| \cdot m/t$.

Proofs

Expected number of elements in each list in hashing with chaining

For $x \notin S$ and a universal hash function h, the expected number of elements in $L[h(x)]$ is less than or equal to 1.
Proof We have

$$\begin{aligned}
\mathbb{E}\left[|L[h(x)]|\right] &= \mathbb{E}\left[|\{y \in S | h(y) = h(x)\}|\right] \\
&= \mathbb{E}\left[\sum_{y \in S} [h(y) = h(x)]\right] \\
&= \left[\mathbb{E} \sum_{y \in S} [h(y) = h(x)]\right] \\
&= \left[\mathbb{P} \sum_{y \in S} [h(y) = h(x)]\right] \\
&\leq |S| \frac{1}{m} = \frac{n}{m} \leq 1.
\end{aligned}$$

Upperbound of error of unique signatures

Let $x \in S \subseteq U$, $|S| = n$ and $s : U \to [n^3]$ be a universal hash function. The probability of an error (collision) is less than $1/2n$, where $n = |S|$ and $S \subseteq U$.
Proof We have

$$\begin{aligned}
\mathbb{P}\left[\exists \{x, y\} \subseteq S | s(x) = s(y)\right] &\leq \sum_{\{x,y\} \subseteq S} \mathbb{P}[s(x) = s(y)] \\
&\leq \frac{\binom{n}{2}}{n^3} \\
&< \frac{1}{2n}.
\end{aligned}$$

Connection between strongly universal and universal

Consider distinct keys $x, y \in [u]$, possibly non-distinct hash values $q, r \in [m]$, $h : [u] \to [m]$, and $h(x) = q$ and $h(y) = r$. Then, if h is strongly universal, it is also universal.
Proof

$$\mathbb{P}[h(x) = h(y)] = \sum_{q \in [m]} \mathbb{P}[h(x) = q \wedge h(y) = q] = m/m^2 = 1/m.$$

Equivalent definitions of strong universality

The two equivalent definitions

1. A random hash function $h : U \to [m]$ is *strongly universal* (aka. 2-*independent*) if
   (a) Each key is hashed uniformly into $[m]$ (that is, $\forall x \in U, q \in [m] : \mathbb{P}[h(x) = q] = \frac{1}{m}$), and
   (b) Any two distinct keys hash independently.

2. for all $x \neq y \in U$, and $q, r \in [m]$ we have

$$\mathbb{P}[h(x) = q \wedge h(y) = r] = \frac{1}{m^2}.$$

are equivalent.

Proof First assume strong universality and consider distinct keys $x, y \in U$. For any hash value $q \in [m]$, we have

$$\mathbb{P}[h(x) = q] = \sum_{r \in [m]} \mathbb{P}[h(x) = q \wedge h(y) = r] = \frac{m}{m^2} = \frac{1}{m},$$

so $h(x)$ is uniform in $[m]$, and the same holds for $h(y)$. Moreover, for any hash value $r \in [m]$,

$$\begin{aligned}\mathbb{P}[h(x) = q | h(y) = r] &= \frac{P[h(x) = q \wedge h(y) = r]}{\mathbb{P}[h(y) = r]} \\ &= \frac{1/m^2}{1/m} = \frac{1}{m} = \mathbb{P}[h(x) = q],\end{aligned}$$

so $h(x)$ is independent of $h(y)$. For the converse direction, when $h(x)$ and $h(y)$ are independent we have

$$\mathbb{P}[h(x) = q \wedge h(y) = r] = \mathbb{P}[h(x) = q] \cdot \mathbb{P}[h(y) = r],$$

and when $h(x)$ and $h(y)$ are uniform we have

$$\mathbb{P}[h(x) = q] = \mathbb{P}[h(y) = r] = \frac{1}{m}$$

so $\mathbb{P}[h(x) = q] \cdot \mathbb{P}[h(y) = r] = \frac{1}{m^2}$.

Van Emde Boas Trees

General Knowledge

Van Emde Boas trees support `MEMBER`, `INSERT`, `DELETE`, `MINIMUM`, `MAXIMUM`, `SUCCESSOR`, and `PREDECESSOR` in $O(\lg\lg u)$ worst-case time. The hitch is that the keys must be integers in the range 0 to $u-1$, with no duplicates allowed. We call the set $\{0, 1, ..., u-1\}$ the ***universe*** of values that can be stored and u ***universe size***. We assume that u is an exact power of 2.

To allow u to be able to be of the form 2^{2k+1}, we let

$$\sqrt[\uparrow]{u} = 2^{\lceil (\lg u)/2 \rceil}$$

and

$$\sqrt[\downarrow]{u} = 2^{\lfloor (\lg n)/2 \rfloor}.$$

Then we define the following functions

1. $high(x) = \left\lfloor \frac{x}{\sqrt[\downarrow]{u}} \right\rfloor$: The high half of x's bits
2. $low(x) = x \bmod \sqrt[\downarrow]{u}$: The low half of x's bits
3. $index(x, y) = x\sqrt[\downarrow]{u} + y$: The value of element at cluster x index y.

NOTE: we expect x to have $\sqrt{u}$ bits. e.g. for $u = 16$ we have $2 = 0b0010$, even though the first two bits does not add anything they are still important for computing *high* and *low*.

Each node in the van Emde Boas tree has the following structure:

- A field $vEB(u)$ denoting the universe size u
- A field *min* that stores the minimum element in the vEB tree
- A field *Max* that stores the maximum element in the vEB tree
- A field *Summary* that points to a summary node, that shows shows whether other clusters contains any elements.
- An array *cluster* of pointers that points to sub-vEB trees, each of size $\sqrt{u}$.

However, if $u = 2$, then it only has $vEB(u)$, *min* and *max*.

The element stored in *min* is not stored in any of the clusters. Unless the vEB tree contains just one element the element stored in max is stored in the clusters.

We can tell whether a vEB tree has 0 elements, 1 element or more elements in constant time from its *min* and *max* values. If $min = max = NIL$, then it has no elements. If $min = max \neq NIL$, then it has 1 element. If $NIL \neq min \neq max$, then it has two or more elements. Before using a van Emde Boas tree, we must know the universe size u, so that we can create a van Emde Boas tree of the appropriate size that initially represents an empty set. The total space requirement of a van Emde Boas tree is $O(u)$, and we can create an empty tree in $\Theta(u)$.

The recursive procedures that implement the vEB-tree operations will all have running times characterized by the recurrence

$$T(u) \leq T(\sqrt[\uparrow]{u}) + O(1).$$

Letting $m = \lg u$, we rewrite it as

$$T(2^m) \leq T(2^{\lfloor m/2 \rfloor}) + O(1).$$

Noting that $\lfloor m/2 \rfloor \leq 2m/3$ for all $m \geq 2$, we have

$$T(2^m) \leq T(2^{2m/3}) + O(1).$$

Letting $S(m) = T(2^m)$, we rewrite this last recurrence as

$$S(m) \leq S(2m/3) + O(1),$$

which, by case 2 of the master method, has the solution $S(m) = O(\lg m) = O(\lg \lg u)$.

Algorithm 1 vEB-TREE-MEMBER

```
Require: vEB tree $V$
Require: Key to search for $x$
if $x == V.min$ or $x == V.max$ then
    return True
else if $V.u == 2$ then
    return False
else
    return vEB-TREE-MEMBER($V.cluster[high(x)], low(x)$)
```

Algorithm 2 vEB-TREE-SUCCESSOR

```
Require: vEB tree $V$
Require: Key to find successor of $x$
if $V.u == 2$ then
    if $x == 0$ and $V.max == 1$ then
        return 1
    else
        return $NIL$
else if $V.min \neq NIL$ and $x < V.min$ then
    return $V.min$
else
    $max - low = V.cluster[high(x)].max$
    if $max - low \neq NIL$ and $low(x) < max - low$ then
        $offset$ =vEB-TREE-SUCCESSOR($V.cluster[high(x)], low(x)$)
        return $index(high(x), offset)$
    else
        $succ - cluster$ =vEB-TREE-SUCCESSOR($V.summary, high(x)$)
        if $succ - cluster == NIL$ then
            return $NIL$
        else
            $offset = V.cluster[succ - cluster].min$
            return $index(succ - cluster, offset)$
```

Algorithm 3 vEB-TREE-PREDECESSOR

```
Require: vEB tree $V$
Require: Key to find predecessor of $x$
if $V.u == 2$ then
    if $x == 0$ and $V.max == 1$ then
        return 1
    else
        return $NIL$
else if $V.min \neq NIL$ and $x < V.min$ then
    return $V.min$
else
    $max - low = V.cluster[high(x)].max$
    if $max - low \neq NIL$ and $low(x) < max - low$ then
        $offset =$vEB-TREE-PREDECESSOR($V.cluster[high(x)], low(x)$)
        return $index(high(x), offset)$
    else
        $pred - cluster =$vEB-TREE-PREDECESSOR($V.summary, high(x)$)
        if $pred - cluster == NIL$ then
            if $V.min \neq NIL$ and $x > V.min$ then
                return $V.min$
            else
                return $NIL$
        else
            $offset = V.cluster[pred - cluster].min$
            return $index(pred - cluster, offset)$
```

Algorithm 4 vEB-TREE-INSERT

```
Require: vEB tree $V$
Require: Key to insert $x$
if $V.min == NIL$ then
    $V.min = V.max = x$
else
    if $x < V.min$ then
        Swap $x$ and $V.min$
    if $V.u > 2$ then
        if $V.cluster[high(x)].min == NIL$ then
            vEB-TREE-INSERT($V.summary, high(x)$)
            vEB-EMPTY-TREE-INSERT($V.cluster[high(x)], low(x)$)
        else
            $V.cluster[high(x)].min = V.cluster[high(x)].max = low(x)$
    if $x > V.max$ then $V.max = x$
```

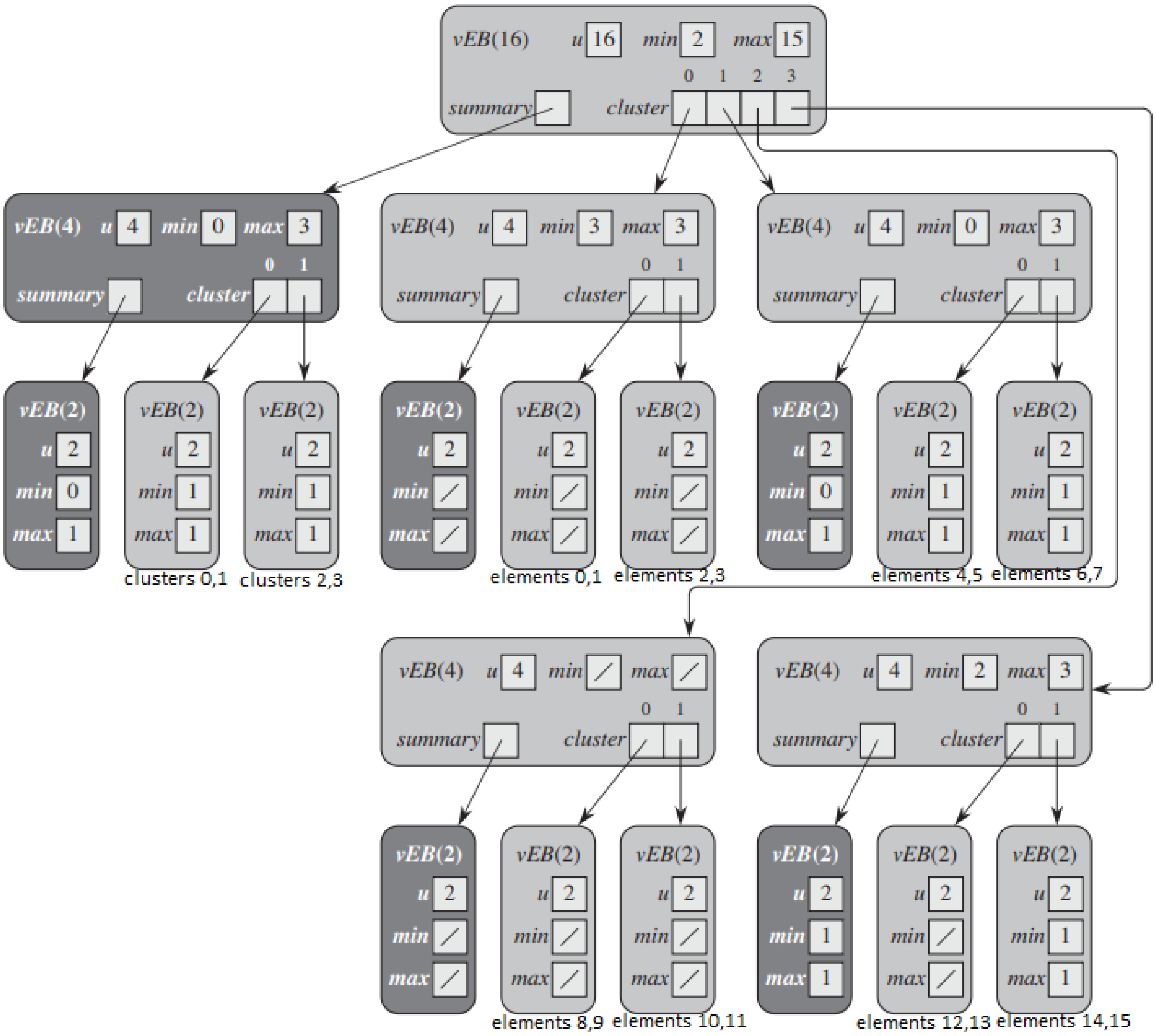


Figure 20.6 A *vEB*(16) tree corresponding to the proto-vEB tree in Figure 20.4. It stores the set {2, 3, 4, 5, 7, 14, 15}. Slashes indicate NIL values. The value stored in the *min* attribute of a vEB tree does not appear in any of its clusters. Heavy shading serves the same purpose here as in Figure 20.4.

Proofs

Examples

NP-Completeness

General Knowledge

We shall refer to three classes of problems:

- P: Problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem
- NP: problems that are "verifiable" in polynomial time. Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time.
- NPC/Np-Complete: Problems that are in NP and are as "hard" as any problem in NP.

When we demonstrate that a problem is NP.complete, we are making a statement about how hard it is (or at least how hard we think it is), rather than about how easy it is. We are not trying to prove the existence of an efficient algorithm, but instead that no efficient algorithm is likely to exist. We rely on three key concepts in showing a problem to be NP-complete

1. *Decision problems vs. opitmization problems*: Many problems of interest are ***optimization problems***, in which each feasible solution has an associated value, and we wish to find a feasible solution with the best value. NP-completeness applies directly not to opimization problems, however, but to ***decision problems***, in which the answer is simply "yes" or "no". If we can provide evidence that a decision problem is hard,we also provide evidence that its related optimization problem is hard.
2. *Reductions*: Let us consider a decision problem A, which we would like to solve in polynomial time. We call the input to a particular problem an ***instance*** of that problem. Now suppose that we already know how to solve a different decision problem B in polynomial time. Finally, suppose that we have a procedure that transforms any instance α of A into some instance β B with the following characteristics:
   - The transformation takes polynomial time
   - The answers are the same

   We call such procedure a polynomial-time ***reduction algorithm*** and it provides us a way to solve problem A in polynomial time. As long as this takes polynomial time, we have a way to decide α in polynomial time. We use polynomial-time reductions in the opposite way to show that a problem is NP-complete. For NP-completeness, we cannot assume that there is no polynomial-time algorithm for problem A. However, we can prove that problem B is NP-complete on the assumption that problem A is also NP-complete.
3. *A first NP-complete problem*: Because the technique of reduction relies on having a problem already known to be NP.complete in order to prove adifferent problem NP-complete, we need a "first" NP-complete problem. The problem we shall use is the *circuit-satisfiabilitty problem*

We define an ***abstract problem*** Q to be a binary relation on a set I of problem ***instances*** and a set S of problem ***solutions***. As we saw above, the theory of NP-completenss restricts attention to ***decision problems***: those having a yes/no solution. In this case, we can view an abstract decision problem as a function that maps the instance set I to the solution set $\{0, 1\}$. Many abstract problems are not decision problems, but rather ***optimization problems***. As we saw above, however, we can usually recast an optimization problem as a decision problem that is no harder.

An ***encoding*** of a set S of abstract objects is a mapping e from S to the set of binary strings. Polygons, graphs, functions, ordered pairs, programs - all can be encoded as binary strings. Thus, a computer algorithm that "solves" some abstract decision problem actually takes an encoding of a problem instance as input. We call a problem whose instance set is the set of binary strings a ***concrete problem***. We say that an algorithm ***solves*** a concrete problem in time $O(T(n))$ if, when it is provided a problem instance i of length $n = |i|$, the algorithm can produce the solution in $O(T(n))$ time. A concrete problem is ***polynomial-time solvable***, therefore, if there exists an algorithm to solve it in time $O(n^k)$ for some constant k.

We can now formally define the ***complexity class P*** as the set of concrete decision problems that are polynomial-time solvable.

We can use encodings to map abstract problems to concrete problems. Given an abstract decision problem Q mapping an instance set I to $\{0,1\}$, an encoding $E : I \rightarrow \{0,1\}^*$ can induce a related concrete decision problem, which we denote by $e(Q)$. If the solution to an abstract-problem instance $i \in I$ is $Q(i) \in \{0,1\}$, then the solution to the concrete-problem instance $e(i) \in \{0,1\}^*$ is also $Q(i)$.

Suppose that an integer k is to be provided as the sole input to an algorithm, and suppose that the running time of the algorithm is $\Theta(k)$. If the integer k is provided in unary, then the running time of the algorithm is $O(n)$ on length-n inputs, which is polynomial time. If we use binary representation of the integer k, however, then the input length is $n = \lfloor \lg k \rfloor + 1$. In this case, the running time of the algorithm is $\Theta(k) = \Theta(2^n)$. Thus, depending on the encoding, the algorithm runs in either polynomial or superpolynomial time.

We say that a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ is ***polynomial-time computable*** if there exists a polynomial-time algorithm A that, given any input $x \in \{0,1\}^*$, produces as output $f(x)$. For some set I of problem instances, we say that two encodings e_1 and e_2 are ***polynomially related*** if there exist two polynomial-time computable functions f_{12} and f_{21} such that for any $i \in I$, we have $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_(i)$. If two encodings e_1 and e_2 of an abstract problem are polynomially related, whether the problem is polynomial-time solvable or not is independent of which encoding we use. Thus, whether an abstract problem has its instances encoded in binary or base 3 does not affect whether it is polynomial-time solvable or not, but if instances are encoded in unary, its complexity may change. To denote the standard encoding of an object, we sehall enclose the object in angle braces. Thus, $\langle G \rangle$ denotes the standard encoding of a graph G.

An ***alphabet*** Σ is a finite set of symbols. A ***language*** L over Σ is any set of strings made up of symbols from Σ. We denote the ***empty string*** by ε, the ***empty language*** by $\emptyset$, and the language of all strings over Σ by Σ^*.

From the point of view of language theory, the set of instances for any decision problem Q is simply the set Σ^*, where $\Sigma = \{0,1\}$. Since Q is entirely characterized by those problem instances that produce a 1 answer, we can view Q as a language L over $\Sigma = \{0,1\}$, where

$$L = \{x \in \Sigma^* : Q(x) = 1\}.$$

For example, the decision problem PATH has the corresponding language

$$\begin{aligned} PATH = \{\langle G, u, v, k \rangle : \ & G = (V,E) \text{ is an undirected graph,} \\ & u, v \in V \\ & k \geq 0 \text{ is an integer, and} \\ & \text{there exists a path from } u \text{ to } v \text{ in } G \text{ consisting of at most } k \text{ edges.}\}. \end{aligned}$$

We say that an algorithm A ***accepts*** a string $x \in \{0,1\}^*$ if, given input x, the algorithm's output $A(x)$ is 1. The language ***accepted*** by an algorithm A is the set of strings $L = \{x \in \{0,1\}^* : A(x) = 1\}$.

A language L is ***decided*** by an algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A. A language L is ***accepted in polynomial time*** by an algorithm A if it is accepted by A and if in addition there eixsts a constant k such taht for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$. A language L is ***decided in polynomial time*** by an algorithm A if there eixsts a constant k such that for any length-n string $x \in \{0,1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$.

Using this language-theoretic framework, we can provide an alternative definition of the complexity class P:

$$P = \{L \subseteq \{0,1\}^* : \text{ there exists and algorithm } A \text{ that decides } L \text{ in polynomial time}\}.$$

In fact, P is also the class of languages that can be accepted in polynomial time.

We define a ***verfirication algorithm*** as being a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a ***certificate***. A two-argument algorithm A ***verifies*** an input string x if there exists a certificate y such that $A(x,y) = 1$. The ***language verified*** by a verification algorithm A is

$$L = \{x \in \{0,1\}^* : \text{ there exists } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\}.$$

Intuitively, an algorithm A verifies a language L if for any string $x \in L$, there exists a certificate y that A can use to prove that $x \in L$. Moreoever, for any string $x \notin L$, there must be no certificate proving that $x \in L$.

The ***complexity class NP*** is the class of languages that can be verified by a polynomiial-time algorithm. More precisely, a language L belongs to NP iff thereexist a two-input polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0,1\}^* : \text{ there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x,y) = 1\}.$$

We say that algorithm A ***verifies*** language L in ***polynomial time***.

Intuitively, a problem Q can be reduced to another problem Q' if any instance of Q can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q. Returning to our formal-language framework for decision problems, we say that a language L_1 is ***polynomial-time reducible*** to a language L_2, written $L_1 \leq_p L_2$, if there exists a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$$x \in L_1 \text{ iff } f(x) \in L_2.$$

We call the function f the ***reduction function***, and a polynomial-time algorithm F that computes f is a ***reduction algorithm***.

We can now define the set of NP-complete languages, which are the hardest problems in NP. A language $L \subseteq \{0,1\}^*$ is ***NP-complete*** if

1. $L \in NP$, and
2. $L' \leq_p L$ for every $L' \in NP$

If a language L satisfies property 2, but not necessarily property 1, we say that L is ***NP-hard***. We also define NPC to be the class of NP-complete language.

The ***circuit-satisfiability problem*** is "Given a boolean combinational circuit composed of AND, OR and NOT gates, is it satisfiable - that is, does it have a truth assignment that causes the output of the circuit to be 1".

In order to pose this question formally, however, we must agree on a standard encoding for circuits. The ***size*** of a boolean combinational circuit it the number of booleam combinational elements plus the number of wires in the circuit. We could devise a graphlike encoding that maps any given circuit C into a binary string $\langle C \rangle$ whose length is polynomial in the size of the circuit itself. As a formal language, we can therefore define

$$CIRCUIT - SAT = \{\langle C \rangle : C \text{ is a satisfiable boolean combinational circuit}\}.$$

There is strong evidence that no polynomial-time algorithm exists that solves the circuitsatisfiability problem.

We can prove that a language L is NP-complete by

1. Prove $L \in NP$
2. Select a known NP-complete language L'
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L
4. Prove that the function f satisfies $x \in L'$ iff $f(x) \in L$ for all $x \in \{0,1\}^*$
5. Prove that the algorithm computing f runs in poylnomial time.

We formulate the ***(formula) satisfiability*** problem in terms of the language SAT as follows. An instance of SAT is a boolean formula ϕ composed of

1. n boolean variables: $x_1, x_2, ..., x_n$
2. m boolean connectivies: any boolean function with one or two inputs and one output
3. parentheses.

We can easily encode a boolea formula ϕ in a length that is polynomial in $n+m$. As in boolea combinational circuits, a ***truth assignment*** for a boolean formula ϕ is a set of values for the variables of ϕ, and a ***satisfying assignment*** is a truth assignment that causes it to evaluate to 1. A formula with a satisfying assignment is a ***satisfiable*** formula. The satisfiability problem asks whether a given boolean formula is satisfiable; in formal-language terms

$$SAT = \{\langle\phi\rangle : \phi \text{ is a satisfiable boolean formula}\}.$$

The naive algorithm to determine whether an arbitrary boolean formula is satisfiable does not run in polynomial time. A formula with n variables has 2^n possible assignments. If the length of $\langle\phi\rangle$ is polynomial in n, then checking every assignment requires $\Omega(2^n)$ time, which is superpolynomial in the length of $\langle\phi\rangle$.

We define $3-CNF$ satisfiability using the following terms. A ***literal*** in a boolean formula is an occurence of a variable or its negation. A boolean formula is in ***conjunctive normal form***, or ***CNF***, if it is expressed as an AND of ***clauses***, each of which is the OR of one or more literals. A boolean formula is in ***3-conjunctive normal form***, or ***3-CNF***, if each clause has exactly three distinct literals. For example, the boolean formula

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

is in $3-CNF$. In $3-CNF-SAT$, we are asked whether a given boolean formula ϕ in $3-CNF$ is satisfiable. A polynomial-time algorithm that can determine the satisfiability of boolean formulas is unlikely to exist.

A ***clique*** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, such that all vertices are adjacent to each other. The ***size*** of a clique is the number of vertices it contains. The ***clique problem*** is the optimization problem of finding a clique of maximum size in a graph. As a decision problem, we ask whether a clique of a given size k exists in the graph. The formal definition is

$$CLIQUE = \{\langle G, k\rangle : G \text{ is a graph containing a clique of size } k\}.$$

A naive algorithm for determining whether a graph with $|V|$ vertices has a clique of size k is to list all k-subsets of V, and check each one to see whether it forms a clique. The running time of this algorithm is $\Omega(k^2\binom{|V|}{k})$, which is polynomial if k is a constant. In general, however, k could be near $\frac{|V|}{2}$, in which case the algorithm runs in superpolynomial time. Indeed, an efficient algorithm for the clique problem is unlikely to exist.

A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such taht if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). That is, each vertex "covers" it incident edges, and a vertex cpver for G is a set of vertices that covers all the edge in E. The ***size*** of a vertex cover is the number of certices in it. The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given graph. Restating this optimization problem as a decision problem, we wish to determine whether a graph has a vertex cover of a given size k. As a language we define

$$VERTEX-COVER = \{\langle G, k\rangle : G \text{ is a graph containing a vertex cover of size } k\}.$$

A ***hamiltionian cycle*** of an undirected graph $G = (V, E)$is a simple cycle that contains each vertex in V. We can define the ***hamiltonian-cycle problem***, "Does a graph G have a hamiltionian cycle?", as a formal language

$$HAM-CYCLE = \{\langle G\rangle : G \text{ is a hamiltonian graph}\}.$$

In the ***traveling-salesman problem*** a salesman must visit n cities. Modeling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a ***tour*** visiting each city exactly once and finishing at the city he starts from. The salesman incurs a nonnegative integer cost $c(i, j)$ to travel from city i to city j, and the salesman wishes to make the tour whose total cost i minimum, where the total cost is the sum of the individial costs along the edges of the tour. The formal language for the corresponding decision problem is

$$\begin{aligned} TSP = \{\langle G, c, k\rangle : & G = (V, E) \text{ is a complete graph} \\ & c \text{ is a function from } V \times V \to \mathbb{N} \\ k \in \mathbb{N} & \\ & G \text{ has a traveling-salesman tour with cost at most } k.\} \end{aligned}$$

A fast algorithm for the traveling salesman problem is unlikely to exist.

In the ***subset-sum problem***, we are given a finite set S of positive integers and an integer ***target*** $t > 0$. We ask wehter there exists a subset $S' \subseteq S$ whose elements sum to t. As usual we define the problem as a language

$$SUBSET-SUM = \{\langle S, t\rangle : \text{ there exists a subset } S' \subseteq S \text{ such that } t = \sum_{s \in S'} s\}.$$

It is important to recall that our standard encoding assumes that the input integers are coded in binary. With this assumption in mind, we can show that the subset-sum problem is unlikely to have a fast algorithm.

Proofs

Lemma 34.1

Let Q be an abstract decision proble on an instance set I, and let e_1 and e_2 be polynomially related encodings on I. Then, $e_1(Q) \in P$ iff $e_2(Q) \in P$.
Proof Not done

Theorem 34.2

$$P = \{L : L \text{ is accepted by a polynomial-time algorithm}\}$$

Proof Not done.

Lemma 34.3

If L_1, $L_2 \subseteq \{0,1\}^*$ are languages such that $L_1 \leq_p L_2$, then $L_2 \in P$ implies $L_1 \in P$.
Proof Not done

Theorem 34.4

If any NP-complete problem is polynomial-time solvable, then $P = NP$. Equivalently, if any problem in NP is not polynomial-time solvable, thene no NP-complete problem is polynomial-time solvable.
Proof Not done

Lemma 34.5

The circuit-satisfiability problem belongs to the class NP.
Proof Not done

Lemma 34.6

The circuit-satisfiability problem is NP-hard.
Proof Not done

Lemma 34.8

If L is a language such that $L' \leq_p L$ for some $L' \in NPC$, then L is NP-hard. If, in addition, $L \in NPC$, then $L \in NPC$.
Proof Not done

Theorem 34.9

Satisfiability of boolean formulas is NP-complete.
Proof Not done

Theorem 34.10

Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.
Proof Not done

Theorem 34.11

The clique problem is NP-complete.
Proof Not done

Theorem 34.12

The vertex-cover problem is NP-complete.
Proof Not done

Theorem 34.13

The hamiltonian cycle problem is NP-complete.
Proof Not done

Theorem 34.14

The traveling salesman problem is NP-complete.
Proof Not done

Theorem 34.15

The subset-sum problem is NP-complete.
Proof Not done

Exact Exponential Algorithms and Parameterized Complexity

General Knowledge

For functions f and g we write $f(n) = O^*(g(n))$ if $f(n) = O(g(n)poly(n))$, where $poly(n)$ is a polynomial.

In *Ecat exponential algorithm* we find an exact solution for all instances in superpolynomial time. In *paramerized algorithms* we find an exact solution in polynomial time for only some instances.

In the *Travelling salesman problem* (TSP) we are given as et of distinct cities $\{c_1, c_2, ..., c_n\}$ and for each pair $c_i \neq c_j$ the distance between c_i and c_j, denoted by $d(c_i, c_j)$. The task is to construct a tour of the travelling salesman of minimum total length which visits all the cities and returnsr to the starting point.

How to find a tour of minimum length? The easy way is to generate all possible solutions. Thus a naive approach here requires at least $n!$ steps. Using dynamic programming one obtains a much faster algorithm.

In the dynamic programming algorithm for TSP define

$$OPT[S, c_i] = \text{"path of minimum length that starts in } c_1\text{, visits all of } S \text{ once, and end in } c_i$$

for all $S \subseteq \{c_2, ..., c_n\}$ and $c_i \in S$. Then

$$\min\{\text{OPT}[\{c_2, ..., c_n\}, c_i] + d(c_i, c_1) | c_i \in \{c_2, ..., c_n\}\}$$

is the length of the minimal tour. Thus, we have

$$OPT[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } s = \{c_i\} \\ \min\{OPT[S \backslash \{c_i\}, c_k] + d(c_k, c_i) | c_k \in S \backslash \{c_i\}\} & \text{if } \{c_i\} \subset S. \end{cases}$$

The corresponding algorithm using dynamic programming for this computes $O(n^2 \cdot 2^n)$ shortest paths.

In the *Maximum Independent Set Problem* (MIS), we are given an undirected graph $G = (V, E)$. The task is to find an independent set $I \subseteq V$, i.e. any pair of vertices of I is non-adjacent, of maximum cardinality. The naive approach is to just try all 2^n subsets. This takes $O^*(2^n)$ time.

Here we present a simple branching algorithm for MIS. If a vertex v is in an independent set I, then none of its neighbors can be in I. On the other hand, if i is a maximum indepndent set, and thus if v is not in I then at least one of its neighbors is in I. This is because otherwise $I \cup \{v\}$ would be an independent set, which contradicts the maximality of I. Thus for every vertex v and every maximal independent set I, there is a vertex y from the closed neighborhood $N[v] = \{v\} \cup \{w \in V | (v, w) \in E\}$ of v, which is the set consisting of v and vertices adjacent to v, such that y is in I, and no other vertex from $N[y]$ is in I. Therefore to solve the problem on G, we solve problems on different reduced instances, and then pick up the best of the obtained solutions.

The algorithm exploits this idea. We pick a vertex of minimum degree and for each vertex from its closed neighborhood we consider a subproblem, where we assume that this vertex belongs to a maximum independent set.

Let $T(n)$ be the maximum number of subproblems considered by the branching algorithm on a graph with n vertices, then $T(n) = O(3^{n/3})$.

A *vertex cover* C of a graph G is a set of vertices that includes at least one endpoint of every edge of the graph. The *k-vertex cover problem* asks whether it is possible to have $|C| \leq k$. C's complement $V \backslash C$ is an independent set of size $n - k$. This can also be seen as the job of a bouncer who wants to block at most k of the n poeple at the door to prevent fights. If that enough to prevent fights, and if so, who should be blocked?

The k-vertex cover problem is NP-complete. One could just brute-force it by trying all 2^n subsets of vertices, which runs in $O(2^k)$, which can be optimized by considering every $v \in V$ and do the following

- If $d(v) = 0$ then we let v in and drop v from G, because v does not have any conflicts
- If $d(v) \geq k+1$, then we decrease k by 1, remove v from G and reject v, since if we were not to do this, then we would have to reject all of v's $> k$ neighbors instead, which would contradict our goal of having rejecting most k people.

Once done we know, that all vertices who has not yet been rejected or accepted will have at most k neighbors and at least 1 neighbor. If $|E| > k^2$, then there is no solution, since each rejection resolves at most k conflicts. This procedure reduces the graph to having $|V| \leq 2k^2$, because

$$|V| = \sum_{v \in V} 1 \leq \sum_{v \in V} d(v) = 2|E| \leq 2k^2.$$

We can then just brute-force the remaining graph by trying all $\binom{2k^2}{k}$ subsets of k people.

We can now introduce a new idea: if a vertice $v \in V$ only has one neighbor $w \in V \backslash v$, then we remove v, w from H, decrease k by 1 and add w to C. This is because we know, that w cannot have less neighbors than v and we cannot use both v and w. This results in a graph with $|V| \leq k^2$ vertices, because

$$|V| = \sum_{v \in V} 1 = \frac{1}{2} \sum_{v \in V} 2 \leq \frac{1}{2} \sum_{v \in V} d(v) = |E| \leq k^2$$

We can then brute-force the remaining graph by trying all $\binom{k^2}{k}$ subsets of k people.

Therefore, instead of using brute force to go through an enormous search space, we use simple observations to reduce the search space to a manageable size. This algorithmic technique, using reduction rules to decrease the size of the instance, is called *kernelization.* The general idea is to use the parameter k to quickly reduce to a smaller subproblem of the same type, whose size ideally depends only on k and not on n.

Thus, we have shown taht if there is a solution for a given k and a given graph G with n vertices and m edges, then we can find a kernel H with at most k^2 vertices. Furthermore, such a kernel can be found in $O(m+n)$ time, and checking if a given subset of size at most k is a solution can be done in $O(k^2)$ time. Thus, for any fixed k, the total running time of this algorithm is

$$O(m + n + \binom{k^2}{k} k^2).$$

This can be improved further by using a Bounded Search Tree. The important observation here is, that for each edge $(u, v) \in E$, at least one of u, v must be rejected. The idea is then to pick an arbitrary edge (u, v), reject u and run the algorithm recursively to check whether the remaining conflicts can be resolved by rejecting at most $k-1$ guests. If this succeeds you already have a solution. If it fails, then move u back into the undecided list, move v to the reject list and run the algorithm recursively to check whether the remaining conflicts can be resolved by tejecting at most $k-$ additional guests. If this recursive call also fails to find a solution, then you can be sure that there is no way to avoid a fight by rejecting at most k guests.

This procedure results in a recursion tree of depth k, thus there is a total of 2^k recursive calls. If we start by rejecting all vertices of degree $d(v) \geq k+1$, the resulting graph has at most $|E| = \frac{1}{2}\sum_{v \in V} d(v) \leq \frac{1}{2} nk$ edges, so constructing each recursive call can be done in $O(nk)$ time. The total running time is then $O(m + nk \cdot 2^k)$.

An important feature of the Bar Fight Prevention problem is the existence of the parameter k. The problem of finding the minimum k that works is NP-complete, but for any fixed constant k we have just seen two linear-time algorithms. We say the problem is *parameterized by the parameter k.*

A parameterized problem is *Fixed Parameter Tractable (FPT)* if it has an algorithm with running time $f(k) \cdot n^c$ for some function f and some constant $c \in \mathbb{R}$

A parameterized problem is *Slice-wise Polynomial (XP)* if it has an algorithm with running time $f(k) \cdot n^{g(k)}$ for some function f, g. We have $FPT \subset XP$, because we can just let $g(k) = c$.

Proofs

Lemma of OPT

We have

$$OPT[S, c_i] = \begin{cases} d(c_1, c_i) & \text{if } s = \{c_i\} \\ \min\{OPT[S\backslash\{c_i\}, c_k] + d(c_k, c_i) | c_k \in S\backslash\{c_i\}\} & \text{if } \{c_i\} \subset S. \end{cases}$$

Proof Let $e = (c_k, c_i)$ be the last edge on one of the paths. If $k = 1$ we are done. If $k \neq 1$ the shortest length through e must be $OPT[S\backslash\{c_i\}, c_k] + d(c_k, c_i)$. The shortest such path must use the minimum over all $c_k \in S\backslash\{c_i\}$.

Lemma of runtime of TSP with dynamic programming

The algorithm runs in $O^*(2^n)$.
Proof
We have

$$\sum_{j=2}^{n-1} \binom{n-1}{j} \sum_{i=1}^{j} (j-1) \leq n^2 \sum_{j=1}^{n} \binom{n}{j} = n^2 \cdot 2^n = O^*(2^n)..$$

Approximation Algorithms

General Knowledge

We have at least three waays to get around NP-completeness. First, if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, we may be able to isolate important special cases that we can solve in polynomial time. Third, we might come up with approaches to find *near-optimal* solutions in polynomial time. We call an algorithm that returns near-optimal solutions an ***approximation algorithm***

We say that an algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n). \tag{8}$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$***-approximation algorithm***. The approximation ratio of an approximation algorithm is never less than 1, since $\frac{C}{C^*} \leq 1$ implies $\frac{C^*}{C} \geq 1$. Therefore, a 1-approximation algorithm produces an optimal solution.

An ***approximation scheme*** for an optimization problem is an approximation algorithm that takes as input not opnly an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1+\epsilon)$-approximation algorithm. We say that an approximation scheme is a ***polynomial-time approximation scheme*** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

We say that an approximation scheme is a ***fully polynomial-time approximation scheme*** if it is an approximation scheme and its running time is polynomial in both $\frac{1}{\epsilon}$ and the size n of the input instance. For example, the scheme might have a running time of $O\left(\left(\frac{1}{\epsilon}\right)^2 n^3\right)$. With such a scheme, any decrease in ϵ comes with an instace in the running time.

We say that a randomized algorithm for a problem has an ***approximation ratio*** of $\rho(n)$ if, for any input of size n, the expected cost C of the solution procuded by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

We call a randomized algorithm that achieves an approximation ratio of $\rho(n)$ a ***randomized*** $\rho(n)$***-approximation algorithm***

Examples

The Vertex-cover Problem

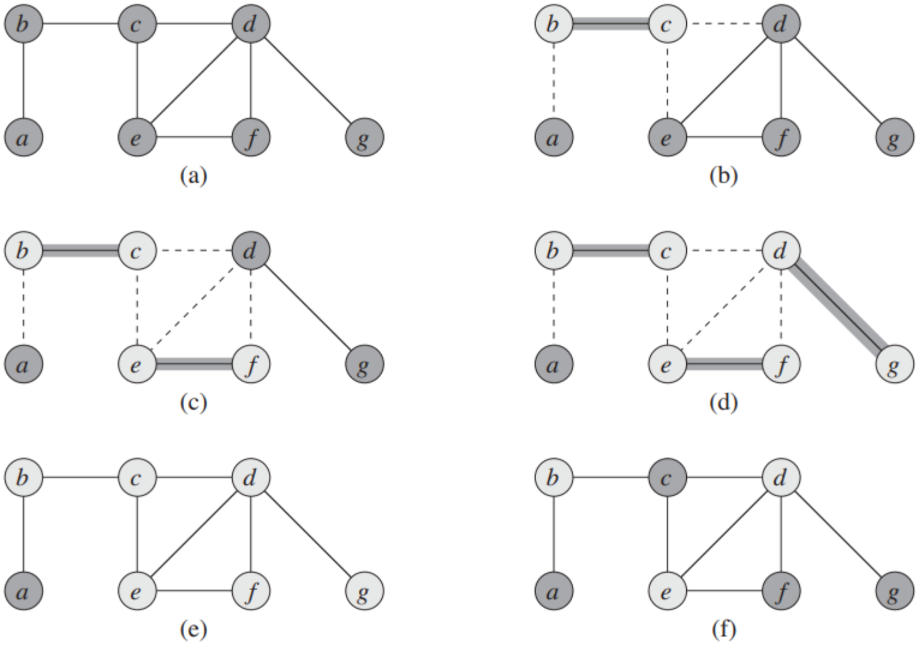


Figure 35.1 The operation of APPROX-VERTEX-COVER. **(a)** The input graph G, which has 7 vertices and 8 edges. **(b)** The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (c, e), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. **(c)** Edge (e, f) is chosen; vertices e and f are added to C. **(d)** Edge (d, g) is chosen; vertices d and g are added to C. **(e)** The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. **(f)** The optimal vertex cover for this problem contains only three vertices: b, d, and e.

A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is a nedge of G, then either $u \in V'$ or $v \in V'$ (or both). The size of a vertex cover is the number of vertices in it. The ***vertex-cover problem*** is to find a vertex cover of minimum size in a given undirected graph. We call such a vertex cover an ***optimal vertex cover***.

The approximation algorithm `APPROX-VERTEX-COVER`(G) returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover. The running time of this algorithm is $O(V + E)$, using adjacency lists to represent E'

The traveling-salesman problem

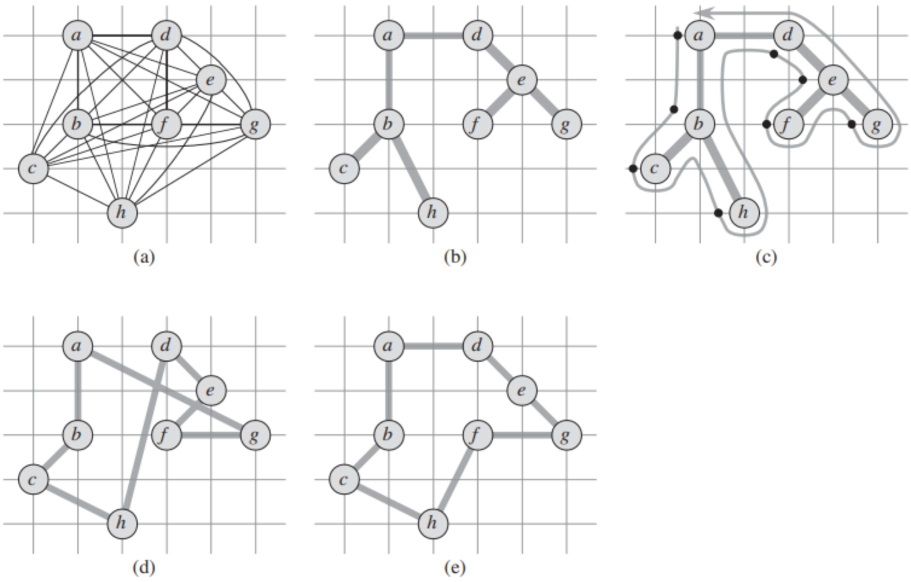
In the traveling-salesman problem, we are given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated wit heach edge $(u, v) \in E$, and we must find a hamiltonian cycle of G with minimum cost. As an extension of our notation, let $c(A)$ denote the total cost of the edges in the subset $A \subseteq E$.

We say that the cost function c satisfies the ***triangle inequality*** if, for all vertices $u, v, w \in V$

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

Algorithm 5 APPROX-VERTEX-COVER

```
Require: Undirected graph $G$
$C = \emptyset$
$E' = G.E$
while $E' \neq \emptyset$ do
    let $(u, v)$ be an arbitrary edge of $E'$
    $C = C \cup \{u, v\}$
    remove from $E'$ edge $(u, v)$ and every edge incident on either $u$ or $v$
return $C$
```



(a) (b) (c)

(d) (e)

Figure 35.2 The operation of APPROX-TSP-TOUR. **(a)** A complete undirected graph. Vertices lie on intersections of integer grid lines. For example, f is one unit to the right and two units up from h. The cost function between two points is the ordinary euclidean distance. **(b)** A minimum spanning tree T of the complete graph, as computed by MST-PRIM. Vertex a is the root vertex. Only edges in the minimum spanning tree are shown. The vertices happen to be labeled in such a way that they are added to the main tree by MST-PRIM in alphabetical order. **(c)** A walk of T, starting at a. A full walk of the tree visits the vertices in the order $a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$. A preorder walk of T lists a vertex just when it is first encountered, as indicated by the dot next to each vertex, yielding the ordering a, b, c, h, d, e, f, g. **(d)** A tour obtained by visiting the vertices in the order given by the preorder walk, which is the tour H returned by APPROX-TSP-TOUR. Its total cost is approximately 19.074. **(e)** An optimal tour H^* for the original complete graph. Its total cost is approximately 14.715.

We shall first compute a minimum spanning tree, whose weight gives a lower bound on the le ngth of an optimal traveling-salesman tour. We shall then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality. `APPROX-TSP-TOUR`(G, c) implements this approach with a running time of $\Theta(V^2)$ depending on how a simple implementation of how to compute the minimum spanning tree

Algorithm 6 APPROX-TSP-TOUR

```
Require: Complete undirected graph
Require: Cost function $c$ satisfying the triangle inequality
select a vertex $r \in G.V$ to be a "root" vertex
compute a minimum spanning tree $T$ for $G$ from root $r$
let $H$ be a list of vertices, ordering according to when they are first visited in a preorder tree walk of $T$
return the hamiltionian cycle $H$
```

The set-covering problem

An instance (X, F) of the ***set-covering problem*** consists of a finite set X and a family F of subsets of X, such that every element of X belongs to at least one subset in F. We say that a subet $S \in F$ ***covers*** its elements. The problem is to find a minimum-size subset $C \subseteq F$ whose members cover all of X (both F and C are thus sets of multiple sets). We say, that any C that cover all of X, ***covers*** X.

The greedy method works by picking, at each stage, the set S that covers the greatest number of remaining elements that are uncovered We can easily implement the algorithm to run in time polynomial in $|X|$ and $|F|$.

Algorithm 7 GREEDY-SET-COVER

```
Require: A finite set $X$
Require: A family $F$ of subsets of $X$
$U = X$
$C = \emptyset$
while $U \neq \emptyset$ do
    Select an $S \in F$ that maximizes $|S \cap U|$
    $U = U - S$
    $C = C \cup \{S\}$
return $C$
```

Since the number of iterations of the loop is bounded from above by $\min(|X|, |F|)$, and we can implement the loop body to run in time $O(|X||F|)$, a simple implementation runs in time $O(|X||F| \min(|X|, |F|))$.

MAX-3-CNF satisfiability

A particular instance of 3-CNF satisfiability may or may not be satisfiable. In order to be satisfiable, there must exist an assignment of the variables so that every clause evaluates to 1. If an instance is not satisfiable, we may want to compute how "close" to satisfiable it is, that is, we may wish to find an assignment of the variables that satisfies as many clauses as possible. We call the resulting maximization problem ***MAX-3-CNF satisfiability***. The input to MAX-3-CNF satisfiability is the same as for 3-CNF satisfiability, and the goal is to return an assignment of the variables that maximizes the number of clauses evaluating to 1. We require each clause to consist of exactly three distinct literals. We further assume that no clause contains both a variable and its negation.

$$\begin{array}{lrcll}
\text{minimize} & \multicolumn{4}{l}{\displaystyle\sum_{v \in V} w(v)\, x(v)} \\
\text{subject to} & & & & \\
 & x(u) + x(v) & \geq & 1 & \text{for each } (u, v) \in E \\
 & x(v) & \leq & 1 & \text{for each } v \in V \\
 & x(v) & \geq & 0 & \text{for each } v \in V\ .
\end{array}$$

Vertex cover using linear programming

In the ***minimum-weight vertex-cover problem***, we are given an undirected graph $G = (V, E)$ in which each vertex $v \in V$ has an associated positive weight $w(v)$. for any vertex cover $V' \subseteq V$, we define the weight of the vertex cover $w(V') = \sum_{v \in V'} w(v)$. The goal is to find a vertex cover of minimum weight. We shall compute a lower bound on the weight of the minimum-weight vertex cover, by using a linear program. We shall then "round" this solution and use it to obtain a vertex cover.

Suppose, that we associate a variable $x(v)$ with each vertex $v \in V$, and let us require that $0 \leq x(v) \leq 1$ for each $v \in V$. We put v into the vertex cover iff $x(v) \geq 1/2$. Then, we can write the constraint that for any edge (u, v), at least one of u and v must be in the vertex cover as $x(u) + x(v) \geq 1$. This view gives rise to the ***linear-programming relaxation*** for finding a minimum-weight vertex cover The procedure `APPROX-MIN-WEIGHT-VC`$(G,\ w)$ uses the solution to the linear-programming relaxation to construct an approximate solution to the minimum-weight vertex-cover problem

Algorithm 8 APPROX-MIN-WEIGHT-VC

```
Require: Undirected graph $G = (V, E)$
Require: Positive weight function $w(v)$ for each $v \in V$
$C = \emptyset$
Compute $\bar{x}$, an optimal solution to the lienar program
for each $v \in V$ do
    if $\bar{x}(v) \geq 1/2$ then $C = C \cup \{v\}$
return C
```

Proofs

Theorem 35.1

APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm

Proof It has been shown in a previous chapter, that it runs in polynomial time.
The set C of vertices that is returned by the algorithm is a vertex cover, since the laogrithm loops until every edge in $G.E$ has been covered by some vertex in C.
Let A denote the set of edges that line 4 picked. Not two edges in A share an endpoint. Thus no two edges in A are covered by the same vertex from an optimal cover C^*, and we have the lower bound

$$|C^*| \geq |A| \tag{9}$$

on the size of an optimal vertex cover. Since A consists of the edges between two vertices in C (and since all of the elements in C are unique), we have the (exact) upper bound on the size of the vertex cover returned

$$|C| = 2|A| \tag{10}$$

Combining equation (9) and (10), we obtain

$$|C| = 2|A| \leq 2|C^*|$$

Theorem 35.2

APPROX-TSP-TOUR is a polynomial-time 2-approximation algorithm for the traveling-salesman problem with the triangle inequality

Proof It has been shown in a previous chapter, that it runs in polynomial time.
Let H^* denote an optimal tour for the given set of vertices. We obtain a spanning tree by deleting any edge fro ma tour, and each edge cost is nonnegative. Therefore, the weight of the minimum spanning tree T provides a lower bound on the cost of an optimal tour

$$c(T) \leq c(H^*). \tag{11}$$

A ***full walk*** of T lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let us call this full walk W. Since the full walk traverses every edge of T exactly twice, we have

$$c(W) = 2c(T) \tag{12}$$

Inequality (11) and equation (12) imply that

$$c(W) \leq 2c(H^*) \tag{13}$$

and so the cost of W is within a factor of 2 of the cost of an optimal tour.
Unfortunately, the full walk W is generally not a tour, since it visits some vertices more than once. By the triangle inequality, however, we can delete a visit to any vertex from W and the cost does not increase (If we delete a vertex v from W between visits to u and w, the resulting ordering specifies going directly from u to w). By repeatedly applying this operation, we can remove from W all but the first visit to each vertex. This ordering is the same as that obtained by a preorder walk of the tree T. Let H be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since every vertex is visited exacly once, and in fact it is the cycle computed by the algorithm. Since H is obtained by deleting vertices from the full walk W, we have

$$c(H) \leq c(W). \tag{14}$$

Combining inequalities (13) and (14) gives $c(H) \leq 2c(H^*)$, which completes the proof.

Theorem 35.4

`GREEDY-SET-COVER` is a polynomial-time $\rho(n)$-approximation algorithm, where

$$\rho(n) = H(\max\{|S| : S \in F\})$$

and $H(d) = \sum_{i=1}^{d} 1/i$ is the dth harmonic number.

Proof It has been shown in a previous chapter, that it runs in polynomial time.
To show that the algorithm is a $\rho(n)$-approximation algorithm, we assign a cost of 1 to each set selected by the algorithm, distribute this cost over the elements covered for the first time, and then use these costs to derive the desired relationship between the size of an optimal set cover C^* and the size of the set cover C returned by the algorithm. Let S_i denote the ith subset selected by the algorithm. The algorithm incurs a cost of 1 when it adds S_i to C. We spread this cost of selection S_i evenly among the elements covered for the first time by S_i. Let c_x denote the cost allocated to element x, for each $x \in X$. Each element is assigned a cost only once, when it is covered for the first time. If x is covered for the first time by S_i, then

$$c_x = \frac{1}{|S_i - (S_i \cup S_2 \cup ... \cup S_{i-1}|}.$$

Each step of the algorithm assigns 1 unit of cost, and so

$$|C| = \sum_{x \in X} c_x. \tag{15}$$

Each element $x \in X$ is in at least one set in the optimal cover C^*, and so we have $\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x$. Combining equation (**??**) and inequalitiy (**??**), we have that

$$|C| \leq \sum_{s \in C^*} \sum_{x \in S} c_x. \tag{16}$$

The remainder of the proof rests on the following key inequality, which we shall prove shortly. For any set S belonging to the family F

$$\sum_{x \in S} c_x \leq H(|S|). \tag{17}$$

From inequalities (**??**) and (**??**) it follows that

$$|C| \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \cdot H(\max\{|S| : S \in F\}),$$

thus proving the theorem.
All that remains is to pro ve inequality (17). Consider any set $S \in F$ and any $i = 1, 2, ..., |C|$, and let $u_i = |S - (S_1 \cup S_2 \cup ... \cup S_i)$ be the number of elements in S that remain uncovered after the algorithm has selected sets $S_1, S_2, ..., S_i$. We define $u_0 = |S|$ to be the number of elements of S, which are all initially uncovered. Let K be the least index such that $u_k = 0$, so that every element in S is uncovered by at least one of the sets $S_1, S_2, ..., S_k$ and some element in S is uncovered by $S_1 \cup S_2 \cup ... \cup S_{k-1}$. Then, $u_{i-1} \geq u_i$, and $u_{i-1} - u_i$ elements of S are covered for the first time by S_i, for $i = 1, 2, ..., k$. Thus,

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{|S_i - (S_i \cup S_2 \cup ... \cup S_{i-1})|}.$$

Observe that

$$|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1}) \geq |S - (S_1 \cup S_2 \cup ... \cup S_{i-1}| = u_{i-1},$$

because the greedy choice of S_i guarantees that S cannot cover more new elements than S_i does. Consequently, we obtain

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

We now bound this quantitiy as follows

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \frac{1}{u_{i-1}}$$

$$= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_i-1} \frac{1}{u_{u_i-1}}$$

$$\leq \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_i-1} \frac{1}{j} \quad \text{(because } j \leq u_{u_i-1} \text{)}$$

$$= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i))$$

$$= H(u_0) - H(u_k) \quad \text{(because the sum telescopes)}$$

$$= H(u_0) - H(0)$$

$$= H(u_0) \quad \text{(because } H(0) = 0\text{)}$$

$$= H(|S|)$$

which completes the proof of inequalities (17).

Corollary 35.5

GREEDY-SET-COVER is a polynomial-time $(\ln |X| + 1)$-approximation algorithm.

Proof Use inequality (A.14) and Theorem Theorem 35.4

Theorem 35.6

Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, ..., x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability 1/2 and to 0 with probability 1/2 is a randomized 8/7-approximation algorithm.

proof Suppose that we have independently set each variable to 1 with probability 1/2 and to 0 with probability 1/2. For $i = 1, 2, ..., m$, we define the indicator random variable

$$Y_i = I\{\text{clause } i \text{ is satisfied}\}$$

so that $Y_i = 1$ as long as we have set at least one of the literals in the ith clause to 1. Since no literal appears more than once in the same clause, and since we have assumed that no variable and its negation appear in the smae clause, the settings of the three literals in each clause are independent. A clause is not satisfied only if all three of its literals are set to 0, and so $Pr\{\text{clause } i \text{ is not satisfied}\} = (1/2)^3 = 1/8$. Thus, we have $Pr\{\text{clause } i \text{ is satisfied}\} = 1 - 1/8 = 7/8$, and $E[Y_i] = 7/8$. Let Y be the number of satisfied clauses overall, so that $Y = Y_1 + Y_2 + ... + Y_m$. Then, we have

$$E[Y] = E\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} E[Y_i] = \sum_{i=1}^{m} 7/8 = 7m/8.$$

Clearly, m is an upper bound on the number of satisfied clauses, and thence the approximation ratio is at most $m/(7m/8) = 8/7$.

Theorem 35.7

Algorithm `APPROX-MIN-WEIGHT-VC` is a polynomial-time 2-approximation algorithm for the minimum-weight vertex-cover problem.

Proof Because there is a polynomial-time algorithm to solve the lienar program in line 2, and because the for-loop runs in polynomial time, the algorithm is a polynomial-time algorithm.
Nowe we show that the algorithm is a 2-approximation algorithm. Let C^* be an optimal solution to the minimum-weight vertex-cover problem, and let z^* be the value of an optimal solution to the linear program. Since an optimal vertex cover is a feasible solution to thel inear program, z^* must be a lower bound on $w(C^*)$, that is,

$$z^* \leq w(C^*). \tag{18}$$

Next, we claim that by rounding the fractional values of the variables $\bar{x}(v)$, we produce a set C that is a vertex cover and satisfies $w(C) \leq 2z^*$. To see that C is a vertex cover, consider any edge $(, v) \in E$. By the first constraint, we know that $x(u) + x(v) \geq 1$, which implies that at least one of $\bar{x}(u)$ and $\bar{x}(v)$ is at least $1/2$. Therefore, at least one of u and v is included in the vertex cover, and so every edgge is covered.
Now, we consider the weight of the cover. We have

$$\begin{aligned} z^* = \sum_{v \in V} w(v)\bar{x}(v) \geq \sum_{v \in V : \bar{x} \geq 1/2} w(v)\bar{x}(v) \geq \sum_{v \in V : \bar{x}(v) \geq 1/2} w(v) \cdot \frac{1}{2} = \sum_{v \in C} w(v) \cdot \frac{1}{2} = \frac{1}{2} \sum_{v \in C} w(v) = \frac{1}{2} w(C) \\ = \frac{1}{2} w(C). \qquad (19) \end{aligned}$$

Combining inequalities (18) and (19) gives

$$w(C) \leq 2z^* \leq 2w(C^*)$$

and hence `APPROX-MIN-WEIGHT-VC` is a 2-approximation algorithm.

Polygon Triangulation

General Knowledge

This topic is about the *Art Gallary problem*; how many caomeras do we need to guard a given gallary and how do we decide to place them? We model a gallary as a polygonal region in the plane. We further restrict ourselves to geions that are *simple polygons* (that is, a polygon of a single chain that does not intersect itself).

Let $\mathcal{P}$ be a simple polygon with n vertices. Because $\mathcal{P}$ may be a complicated shape, it seems difficult to say anything about the number of cameras we need to guard $\mathcal{P}$. Hence, we first decompose $\mathcal{P}$ into pieces that are easy to guard, namely triangles. We do this by drawing diagonals between pair of vertices.

A decomposition of a polygon into triangles by a maximal set of non-intersecting diagonals is called a *triangulation* of the polygon. We require that the set of non-intersecting diagonals to be maximal to ensure that no triangle has a polygon vertex in the interior of one of its edges. We can thus guard $\mathcal{P}$ by placing a camera in every triangle of a triangulation $T_{\mathcal{P}}$ of $\mathcal{P}$.

Placing cameras at vertices seems better, because a vertex can be incident to many triangles, and a camera at that vertex guards all of them. This suggest the following approach: let $T_{\mathcal{P}}$ be a triangulation of $\mathcal{P}$. Select a subset of the vertices of $\mathcal{P}$, such that any triangle in $T_{\mathcal{P}}$ has at least one selected vertex, and place the cameras at the selected vertices.

To find such a subset we assign each vertex of $\mathcal{P}$ a color: white, grey or black. The coloring will be such that any vertices connected by an edge or a diagonal have different colors. this is called a *3-coloring* of a triangulated polygon. In a 3-coloring of a triangulated polygon, every triangle has a white, black and gray vertex. Hence, if we place cameras at all gray vertics, say, we have guarded the whole polygon. By choosing the smallest color class to place the cameras, we guard $\mathcal{P}$ using at most $\lfloor \frac{n}{3} \rfloor$ cameras.

Let P be a simple polygon with n vertices. A set of $\lfloor \frac{n}{3} \rfloor$ camera position in such that any point inside is visible from at least one of the cameras can be computed in $O(n \log n)$ time.

The previous triangulation algorithm will take quadratic time in worst case. This can be improved for some classes of polygons. For instance, convext polygons: Pick one vertex and draw diagonals from every other vertex, that is nto a neighbor, to the vertex. This runs in linear time. Thus, a possible solution would be to decompose into convext pieces and then triangulate the pieces. However, it is difficult to partition a polygon into convex pieces. Therefore, we shall decompose into *monotone pieces*, which is a lot easier.

A simple polygon is called *monotone with respect to a line L* if for any line L' perpendicular to L, the intersection of the polygon with L' is connected. A polygon that is monotone with respect to the y-axis is called *y-monotone*. The following property is characteristic for y-monotone polygons: if we walk from a topmost to a bottommost vertex along the left (or the right) boundary chain, then we always move downwards or horizontally, never upwards.

Our strategy to triangulate the polygon $\mathcal{P}$ is to first partition $\mathcal{P}$ into y-monotone pieces, and then triangulate the pieces. We can partition a polygon into monotone pieces as follows. imagine walking from the topmost vertex of $\mathcal{P}$ to the bottommost vertex on either boundary chain. A vertex where the direction in which we walk switches from downward to upward or from upward to downward is called a *turn vertex*. To partition $\mathcal{P}$ into y-monotone pieces, we need to remove the turn vertices, which is done by adding diagonals. If at a turn vertex v both incident edges go down and the interior of the polygon lies above v, then we must choose a diagonal that goes up from v. v cannot be a turn vertex in either of the two resulting subpolygons. If both incident edges of a turn vertex go up and the interior lies bellow it, we have to choose a diagonal that goes down.

We distinguish five types of vertex in P. Four of these types are turn vertices: *Start vertices*, *Split ver-*

tices, *End vertices*, and *merge vertices*. They are defined as follows. A vertex is a start vertex if its two neighbors lie below it and the interior angle at v is less than π. If the interior angle is greater than π then v is a split vertex. A vertex is an end vertex if its two neighbors lie above it and the interior angle at v is less than π. If the interior angle is greater than π then v is a merge vertex. The vertices that are not turn vertices are *regular vertices*. Thus a regular vertex has one of its neighbors above it and the other neighbor below it.

Lemma 3.4 implies, that $\mathcal{P}$ has been partitioned into y-monotone pieces once we get rid of its split and merge vertices. We do this by adding a diagonal going upward from each split vertex and downward from each merge vertex. Once done, $\mathcal{P}$ has been split into y-monotone pieces.

Let's see how we can add the diagonals for the split vertices. Let $v_1, v_2, ..., v_n$ be a counterclock enumeration of the vertces of $\mathcal{P}$. Let $e_1, ..., e_n$ be the set of edges of $\mathcal{P}$, where $e_i = \overline{v_i v_{i+1}}$ for $1 \leq i < n$ and $e_n = \overline{v_n v_1}$.

The algorithm moves a line L downward over the plane. The line halts at certain event points; in our case the vertices of $\mathcal{P}$. The event points are stored in an event queue Q. The event queue is a priority queue, where the priority of a vertex is its y-coordinate. This way the next event to be handled can be found in $O(\log n)$ time. The goal of the sweep is to add diagonals from each split vertex to a vertex lying above it. Let e_j be the dge immediately to the left of v_i on the sweep line, and let e_k be the edge immediately to the right of v_i on the sweep line. Then we can always conenct v_i to the lowest vertex in between e_j and e_k, and above v_i. If there is no such vertex then we can connect v_i to the upper endpoint of e_j or to the upper endpoint of $_k$. This vertex is called the *helper* of e_j and is denoted by *helper*(e_j). Formally, *helper*(ej) is defined as the lowest vertex above the sweepline such that the horizontal segment connecting the vertex to e_j lies inside $\mathcal{P}$

Nowe we know how to get rid of split vertices. What about merge vertices. Suppose the sweep line reaches a merge vertex v_i. let e_j and e_k be the edge simmediately to the right and left of v_i on the sweepline. Observe that v_i becomes the new helper of e_j when we reach it. We would like to connect v_i to the highest vertex below the sweep line in between e_j and e_k. We do not know the highest vertex below the sweep line when we reach v_i. But it is easy to find later on: when we reacj a vertex v_m, that replaces v_i as the hlper of e_j, then this is the vertex we are looking for. So whenever we replace the hlper of some edge, we check whether the old helper is amerge vertex and, if so, we add the diagonal between the old helper and the new one. This diagonal is always added when the new helper is a split vertex, to get rid of the split vertex. It can also happen that the hlper of e_j is not replaced anymore below v_i. In this case we can connect v_i to the lower endpoint of e_j.

In the above approach, we need to ifnd the dge to the left of each vertex. Therefore we store the edges of $\mathcal{P}$ intersecting the sweep line in the leaves of a dynamic binary search tree T. The left-to-right order of the leaves of T corresponds to the left-to-right order of the edges. Because we are only interested in edges to the left of split and merge vertices we only need to store edges in T that have the interior of P to the right. With each edge in T store its helper. The tree T and the helpers stored with the edges form the status of the sweep line algorithms.

The algorithm partitions $\mathcal{P}$ into subpolygons that have to be processed in a later stage. To have easy access to these subpolygons we shall store the subdivision induced by $\mathcal{P}$ and the added diagonals in a doubly-connected edge-list D. We assume P is initially specified as a doubly-connected edge-list. The diagonals outputed for the split and merge vertices are added to the doubly-connected edge-list. To access the doubly-connected edge-list we use cross-pointer between the edges in the status structure and the corresponding edges in the doubly-connected edge-list. Adding a diagonal can then be done in constant time.

Running time: Constructing Q takes linear time and initializing T takes constant time. To handle an event during the sweep we perform one operation on Q, at most one query, one insertion, and one deletion on T, and we insert at most two diagonals into D. Priority queues and balanced search trees allow for queries and updates in $O(\log n)$ time, and an insert into D takes $O(1)$ time. Hence, handling and event takes

$O(\log n)$ time, and the algorithm runs in $O(n \log n)$ time.

In the following we show, that monotone polygons can be triangulated in linear time. Together with the partition into monotone pieces in $O(n \log n)$ time, this implies that any simple polygon can be triangulated in $O(n \log n)$ time.

Let $\mathcal{P}$ be an y-monotone polygon with n vertices. We assume that $\mathcal{P}$ is *strictly y-monotone*, that is, it does not contain horizontal edges.

This property is what makes triangulation a monotone polygon easy: we can work our way through $\mathcal{P}$ from top to the bottom on both chains, adding diagonals whenever this is possible.

The triangulation algorithm handles the vertices in order of decreasing y-coordinate. The algorithm requires a stack S as auxilary data structure. Initially the stack is empty; later it contains the vertices of P that have been encountered but may still need more diagonals. When we handle a vertex we add as many diagonals from this vertex to vertices on the stack as possible. These diagonals split off triangles from $\mathcal{P}$. The vertices that have been handled but not split off - the vertices on the stack - are on the boundary of the part of $\mathcal{P}$ that still needs to be triangulated. The lower the vertex (that is, the more recently it has been encountered), the higher on the stack it is. The part of $\mathcal{P}$ that still needs to be triangulated has a particular shape: it looks like an upside down funnel. One boundary of the funnel consists of a part of a single edge of $\mathcal{P}$, and the other boundary consists of reflex vertices (interior angle is atleast 180^o). Only the highest vertex (which is at the bottom of the stack) is convex. This property remains true after, we have handled the next vertex, hence, it is an invariant of the algorithm.

Ket's see which diagonals we can add when we handle the next vertex. We distinguish two cases: v_j, the next vertex to handle, lies on the same chain as the reflex vertices on the stack, or it lies on the opposite chain

1. If v_j lies on the opposite chain, it must be the funnel. We can add diagonals from v_j to all vertices correctly on the stack, except for the last one; this is the upper vertex, so it is already connected to v_j. All the vertices are popped from the stack v_j and the vertex previously on top of the stack are pushed onto the stack

2. The other case is when v_j is on the same chain as the reflex vertices on the stack. First, pop one vertex from the stack; this vertex is already connected to v_j by an edge of $\mathcal{P}$. Next, pop vertex from the stack and connect them to v_j untill we encounter one where this is not possible. When we find a vertex where we cannot connect v_j, we push the last vertex that has been popped back onto the stack. After this has been done we push v_j onto the stack

What is the running time of this algorithm? Linear - see pseudocode.

Proofs

Theorem 3.1

Every simple polygon admits a triangulation and any triangulation of a simple polygon with n vertices consists of exactly $n-2$ triangles.

Proof
We prove this theorem by inducting on n. When $n=3$ the polygon itself is a triangle and the theorem is trivially true. Let $n>3$ and assume that the theorem is true for all $m<n$. Let $\mathcal{P}$ be a polygon with n vertices

We first prove the existence of a diagonal in $\mathcal{P}$. Let v be the leftmost vertex of $\mathcal{P}$. Let u and w be the two neighboring vertices of v on the boundary of $\mathcal{P}$. If the open segment $\overline{uw}$ lies in the interior of $\mathcal{P}$, we have found a diagonal. Otherwise, there are one or more vertices inside the triangle defined by u, v and w, or on the diagonal $\overline{uw}$. Of those vertices, let v' be the one furthest from the line through u and w. The segment connecting v' and v cannot intersect an edge of $\mathcal{P}$, because such an edge would have an endpoint inside the triangle that is farther from the line through u and w, contradicting the definition of v'. Hence, $\overline{vv'}$ is a diagonal.

Any diagonal cuts $\mathcal{P}$ into two simple subpolygons $\mathcal{P}_1$ and $\mathcal{P}_2$. Let m_1 be the number of vertices in $\mathcal{P}_1$ and m_2 be the number of vertices in $\mathcal{P}_2$. both m_1 and m_2 must be smaller than n, so by induction $\mathcal{P}_1$ and $\mathcal{P}_2$ can be triangulated. Hence, $\mathcal{P}$ can be triangulated as well.

It remains to prove that any triangulation of $\mathcal{P}$ consists of $n-2$ triangles. Consider an arbitrary diagonal in some triangulation $T_{\mathcal{P}}$. This diagonal cuts $\mathcal{P}$ into two subpolygons with m_1 and m_2 vertices, respectively. Every vertex of $\mathcal{P}$ occurs in exactly one of the two subpolygons, except the vertices defining the diagonal, which occurs in both. Hence, $m_1+m_2=n+2$. By induction, any triangulation of $\mathcal{P}_i$ consists of m_i-2 triangles, which implie sthat $\mathcal{P}$ consists of $(m_1-2)+(m_2-2)=n-2$ triangles.

Theorem 3.2

The 3-coloring approach is optimal in worst case.

Proof A 3-coloring always exists. To see this, we look at the *dual graph* of $T_{\mathcal{P}}$. This graph $g(T_{\mathcal{P}})$ has a node for every triangle $T_{\mathcal{P}}$. We denote the triangle corresponding to a node v by $t(v)$. There is an arc between two nodes v and μ if $t(v)$ and $t(\mu)$ share a diagonal. the arcs in $g(T_{\mathcal{P}})$ correspond to diagonals in $T_{\mathcal{P}}$. Because any diagonal cuts $\mathcal{P}$ into two, the removal of an edge from $g(T_{\mathcal{P}})$ splits the graph into two. Hence, $g(T_{\mathcal{P}})$ is a tree. This means that we can find a 3-coloring using a simple graph traversal, such as depth first search.

While we do the depth first search, we maintain the following invariant

1. All vertices of the already encountered trinagles have been colored white, black or gray
2. No two connected vertices have received the same color.

This implies that we have computed a valid 3-coloring when all triangles have been encountered.

The depth first search can be started from any node of $g(T_{\mathcal{P}})$; the three vertices of the corresponding triangle are colored white, gray and black. Now suppose, we reach a node v in g, coming from a node μ. Hence $t(v)$ and $t(\mu)$ share a diagonal. Since the vertices of $t(\mu)$ have already been colored, only one vertex remains to be colored. Because $g(T_{\mathcal{P}})$ is a tree, the other nodes adjacent to v have not been visited yet, and we still have the freedom to give the vertex the remaining color.

We conclude that the 3-coloring approach is optimal in worst case.

Lemma 3.4

A polygon is y-monotone if its has no split vertices or merge vertices.

Proof Suppose $\mathcal{P}$ is not y-monotone. We have to prove that $\mathcal{P}$ contains a split or a merge vertex. Since $\mathcal{P}$ is not monotone, there is a horizontal line $\mathcal{P}$ that intersects $\mathcal{P}$ in more than one connected component. We can choose L such that the leftmost component is a segment and not a single point. Let p be the left endpoint of this segment, and let q be the right endpoint.

Starting at q, we follow the boundary of $\mathcal{P}$ such that p lies to the left of the boundary (this means that we go up from q). At some point r, we will go up again. If $r \neq p$, then the highest vertex we encountered while going from q to r must be a split vertex. If $r = p$, we again follow the boundary of $\mathcal{P}$ starting at q, but in the other direction.

Let r' be the point where the boundary intersects L. We cannot have $r' = p$, because that would mean that the boundary of $\mathcal{P}$ intersects L only twice, contradicting that L intersects $\mathcal{P}$ in more than one component. So we have $r' \neq p$, implying that the lowest vertex we encountered going from q to r' must be a merge vertex.

Lemma 3.5

The algorithm for transforming a polygon monotone adds a set of non-intersecting diagonals that partitions $\mathcal{P}$ into monotone subpolygons.

Proof NOT DONE.