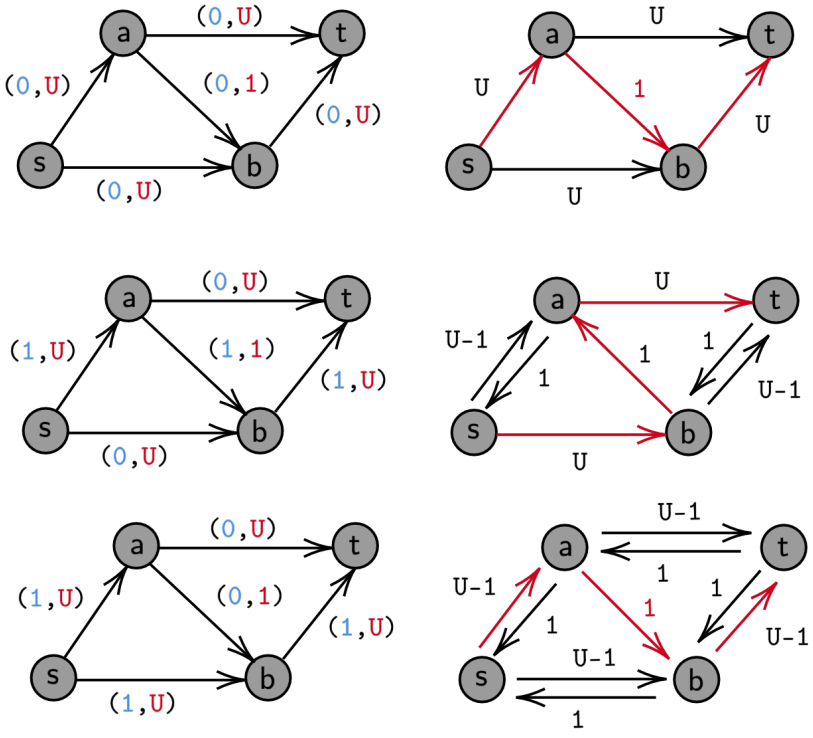
Winter 2022
Instructors: Vetta, Adrian Roshan

Honours Algorithm Design



Contents

Network Flow

Maximum Flow Problem

Definition 1 (Directed Graph). *A **directed graph** is an ordered pair with a set of vertices $V(G)$ and an arc set $A(G) \subseteq \binom{V}{2}$ containing directed edges.*

Let $G = (V, A)$ be a directed graph,

- $|V(G)| = n$
- $|A(G)| = m$

Definition 2 (Directed Path). *A **directed path** in a directed graph is a path in which every edge is traversed from tail to head.*

Definition 3 (Directed Neighborhood). *$\delta^+(X)$ is the set of arcs with a tail in X and a head in $V(G) - X$. In contrast, $\delta^-(X) = \delta^+(V(G) - X)$.*

Definition 4 (Capacity). *Every arc $a = (i, j)$ in $A(G)$ has an integral[1] **capacity** $u_a = u_{ij}$, which is the maximum amount that flows through it.*

[1] $u_a > 0$ for all arcs $a \in A(G)$. If an arc does not exist, we can assume that it has capacity zero.

Definition 5 (Flow). *Let $G = (V, A)$ be a directed graph. A **flow** f from a source s to a sink t is a function $f : A(G) \to R^+$ that satisfies,*

- $0 \leq f_a \leq u_a$ *for all* $a \in A(G)$
- $\sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a$ *for all* $v \in V(G) - \{s, t\}$

The first condition, capacity constraints, states that the flow on an arc $a \in A(G)$ is non-negative and at most its capacity. The second condition, flow conservation, states that the inflow of a vertex $v \notin \{s, t\}$ equals its outflow.

Definition 6 (Flow Value). *Assume that $\delta^+(t) = \delta^-(s) = \emptyset$. Then the **value** of a flow f is the quantity of flow that reaches the sink,*

$$|f| = \sum_{a \in \delta^-(t)} f_a = \sum_{a \in \delta^+(s)} f_a$$

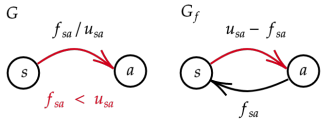
Definition 7 (Maximum Flow). *Let $G = (V, A)$ be a directed graph. The **maximum flow problem** is the problem of finding f^*, where,*

$$f^* = \max \left\{ \sum_{a \in \delta^-(t)} f_a \;\middle|\; \begin{array}{ll} \sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a & \forall v \in V(G) - \{s, t\} \\ 0 \leq f_a \leq u_a & \forall a \in A(G) \end{array} \right\}$$

Definition 8 (Augmenting Path). *An $(s - t)$ path P is **augmenting** if*[2],

[2]

- $f(a) \leq u_a - 1$ *for every arc a used in the forward direction of P*
- $f(a) \geq 1$ *if $a \in A(P)$ is traversed in the backwards direction*



Equivalently, P is a directed path from s to t in the residual network G_f.

Definition 9 (Bottleneck Capacity). *The **bottleneck capacity** of an augmenting path P with respect to a flow f is the maximum amount $b(P, f)$ that we can increase the flow along P by,*

$$b(P, f) = \min \left\{ \min_{\substack{(i,j) \in A \\ i \to j}} u_{ij} - f_{ij}, \min_{\substack{(i,j) \in A \\ i \leftarrow j}} f_{ij} \right\}$$

Definition 10 (Residual Graph). *Let $G = (V, A)$ be a directed graph, and suppose that f is a flow on G. The **residual graph** G_f satisfies,*

- $\exists a_1 \in A(G_f)$ *such that* $u_{a_1} = u_a - f_a$
- $\exists a_2 \in A(G_f)$ *such that* $u_{a_2} = f_a$

for all $a \in A(G)$.

Remark 11. *The bottleneck capacity of a path is the minimum capacity of an arc in the corresponding directed path in the residual graph.*

Definition 12 (Ford-Fulkerson). *Let $G = (V, A)$ be a directed graph. The following algorithm produces a maximum flow f^*,*

Algorithm 1: Ford-Fulkerson

```
   // f is globally given, and initially set to 0.
1  function Augment(P, f)
2      b ← bottleneck capacity b(P, c)
3      foreach a ∈ A(P) do
           // |A(P)| is at most |V(G)| because P is acyclic.
4          if a = (i, j) is a forward arc then
               // Increase f(a) in G by b.
5              f(a) ← f(a) + δ
6          else
               // Decrease f(a') in G by b, where a' = (j, i).
7              f(a^reverse) ← f(a^reverse) + δ
8      return f
9  function Ford-Fulkerson(G)
10     foreach a ∈ A(G) do
11         f(a) ← 0
12         G_f ← residual network of G with respect to f;
13     while ∃(s − t) path P in G_f do
14         f ← Augment(P, f)
15         Update G_f;
16     return f
```

Definition 13 (Cut). *$(S, V(G) - S)$ is an $(s - t)$ **cut** if $s \in S$ and $t \notin S$.*

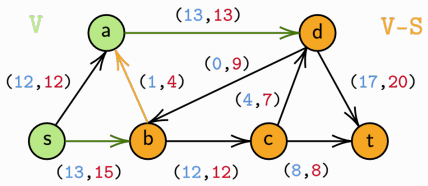
Remark 14. *Let $G = (V, A)$ be a directed graph, and suppose that f^* is the maximum $(s - t)$ flow on G. If S^* is the set of vertices that are reachable from s in the residual graph G_{f^*}, that is,*

$$S^* = \{v \mid \exists \text{ directed } (s - v) \text{ path in } G_{f^*}\}$$

then $(S^*, V(G) - S^*)$ is an $(s - t)$ cut[3] of G and $\delta^+_{G_{f^*}}(S^*) = \emptyset$.

[3] Trivially, $s \in S^*$. By the termination condition in `Ford-Fulkerson`, $t \notin S^*$. If it was, then there would be an $(s - t)$ path in G_{f^*} and the algorithm would continue for another iteration.

Example 1: Example of the Cut Lemma



$$|f| = (13+13) - (1) = 25$$

Lemma 15 (Cut Lemma). *Let $G = (V, A)$ be a directed graph, and suppose that f is an $(s-t)$ flow on G. For any $(s-t)$ cut $(S, V-S)$,*

$$|f| = \sum_{a \in \delta^+(S)} f_a - \sum_{a \in \delta^-(S)} f_a$$

Proof. The value of f is the amount of flow leaving s,

$$|f| = \sum_{a \in \delta^+(s)} f_a$$

Since $\delta^-(\{s\}) = \emptyset$,

$$= \sum_{a \in \delta^+(s)} f_a - \sum_{a \in \delta^-(s)} f_a$$

By Flow Conservation,

$$= \Big(\sum_{a \in \delta^+(s)} f_a - \sum_{a \in \delta^-(s)} f_a \Big) + \sum_{v \in S - \{s\}} \Big(\sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a \Big)$$

Combining the terms to include s,

$$= \sum_{v \in S} \Big(\sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a \Big)$$

Let $a = (i, j) \in A$. There are four cases,

1. $(i, j) \in \delta^+(S) \implies f_a$ appears once with coefficient $+1$.
2. $(i, j) \in \delta^-(S) \implies f_a$ appears once with coefficient -1.
3. $(i, j) \notin S \implies f_a$ does not appear in the sum.
4. $(i, j) \in S \implies f_a$ appears twice with coefficients $+1, -1$.

Consequently,

$$= \sum_{a \in \delta^+(S)} f_a - \sum_{a \in \delta^-(S)} f_a$$

□

Definition 16 (Cut Capacity). ***Capacity*** *of an $(s-t)$ cut $(S, V-S)$ is,*

$$cap(S) = \sum_{a \in \delta^+(S)} u_a$$

Lemma 17. *Let $G = (V, A)$ be a directed graph, and suppose that f is an $(s-t)$ flow on G. $|f| \leq cap(S)$ for any $(s-t)$ cut $(S, V-S)$.*

Proof. By the Cut Lemma,

$$\begin{aligned} |f| &= \sum_{a \in \delta^+(S)} f_a - \sum_{a \in \delta^-(S)} f_a \\ &\leq \sum_{a \in \delta^+(S)} f_a \\ &= cap(S) \end{aligned}$$

□

Theorem 18 (Maxflow-Mincut)**.** *The maximum value of an $(s-t)$ flow is equal to the minimum capacity of an $(s-t)$ cut.*

Proof. Let f^* be the flow output by `Ford-Fulkerson`. Recall that,

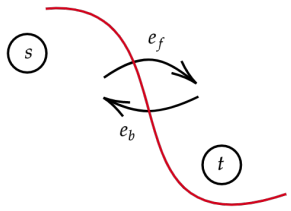
$$S^* = \{v \mid \exists \text{ directed } (s-v) \text{ path in } G_{f^*}\}$$

We showed in Lemma 17 that $|f^*| \leq cap(S^*)$, but we can show that $|f^*| = cap(S^*)$. We saw that $\delta^+_{G_{f^*}}(S^*) = \emptyset$ because we could otherwise grow S^*. Now, any arc $a \in \delta^+(S^*)$ is not in the residual graph because `Ford-Fulkerson` requires it to have reached its capacity: $f^*_a = u_a$. Similarly, the reverse of any arc $a \in \delta^-(S^*)$ is not in the residual graph because `Ford-Fulkerson` requires that $f^*_a = 0$. Now,

$$\begin{aligned} |f^*| &= \sum_{a \in \delta^+(S^*)} f_a - \sum_{a \in \delta^-(S^*)} f_a && \text{by the Cut Lemma} \\ &= \sum_{a \in \delta^+(S^*)} u_a - \sum_{a \in \delta^-(S^*)} f_a && \text{by Observation 1 above} \\ &= \sum_{a \in \delta^+(S^*)} u_a - \sum_{a \in \delta^-(S^*)} 0 && \text{by Observation 2 above} \\ &= cap(S^*) \end{aligned}$$

□

f "saturates" the edge e if $f(e) = c(e)$, i.e., f maximizes the flow on e. Moreover, if e is not saturated in some maximum $(s-t)$ flow, then e does not occur in any min cut.



Remark 19 (Running Time)**.** *`Ford-Fulkerson` (see Algorithm 1) runs in **pseudo-polynomial time**. Its running time is polynomial in the numeric value of the input, but not in the number of bits required to represent it.*

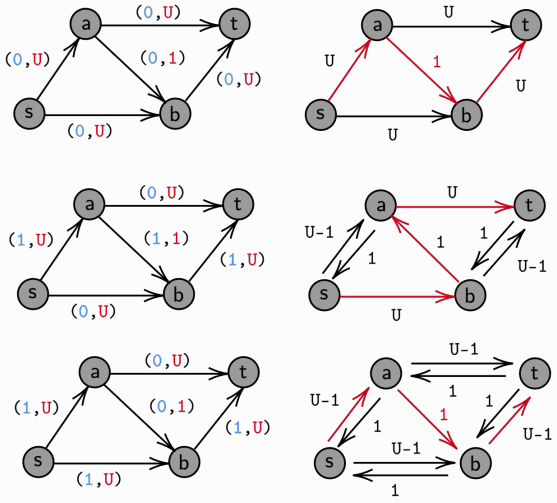
Proof. A single iteration of `Ford-Fulkerson` runs in $O(m)$ because G_f can be found in $O(m)$, an $(s-t)$ path P can be found in $O(m)$ with Breadth-First Search, and `Augment` runs in $O(n)$.

However, the number of iterations is at most $n \cdot U$, where U can be exponential in the input size. The algorithm terminates when

$b(P, f) = 0$ for every $(s - t)$ path P. Since arc capacities are integral, $b(P, f) \geq 1$. This means that in the worst case, flow is increased by 1 at every iteration. Consequently, the minimum value of any $(s - t)$ cut C satisfies $C \leq n \cdot \max_{a \in A(G)} u_a$. The bound is in terms of n, not m, because an $(s - t)$ cut separates $V(G)$. Since the flow is at most $n \cdot U$, the number of iterations is bounded by $n \cdot U$. □

Example 2: Ford-Fulkerson Running Time Analysis

`Ford-Fulkerson` can take $2U$ iterations on this example,



Bipartite Matching Problem

Definition 20 (Independent Set). *An **independent set** is a set of pairwise non-adjacent vertices. The independence number $\alpha(G)$ is the maximal size of an independent set in G.*

Definition 21 (Vertex Cover). *A **vertex cover** $X \subset V(G)$ is a set so that every edge of G has an end in X. The vertex cover number $\tau(G)$ is the minimum size of a vertex cover in G.*

Definition 22 (Matching). *A **matching** $M \subset E(G)$ is a set so that every vertex of G is incident to at most one edge of M. The matching number $\nu(G)$ is the maximum size of a matching in G.*

Definition 23 (Perfect Matching). *A **perfect matching** covers $V(G)$.*

Definition 24 (Bipartition). *A **bipartition** of a graph G is a pair of subsets (A, B) of $V(G)$ so that $A \cap B = \emptyset$, $A \cup B = V(G)$, and every edge of G has one end in A and another in B.*

Definition 25 (Bipartite Matching). *Let $G = (V, E)$ be an undirected bipartite graph. The **bipartite matching problem** is the problem of finding $\nu(G)$, the maximum cardinality matching in G.*

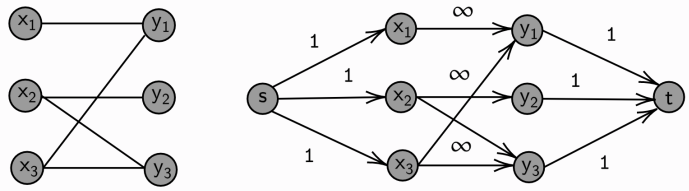
The Ford-Fulkerson Algorithm can be used to solve the bipartite matching problem. To see this, construct an auxiliary network $G = (V, E)$ with bipartition (X, Y) by doing the following,

1. Direct each edge (x_i, y_i) from $x_i \to y_i$
2. Add a source vertex s with an outgoing arc to each vertex in X
3. Add a sink vertex t with an incoming arc from each vertex in Y
4. Give internal and external arcs capacities of ∞ and 1, respectively[4]

[4] This is the same as giving every arc a capacity of 1 because the unique arc into each X-vertex still has capacity 1

Example 3: Constructing the Auxiliary Network

The auxiliary network for the following bipartite graph is,



Theorem 26 (Ford-Fulkerson for Matching). *There is a polynomial time algorithm to find a maximum cardinality matching in a bipartite graph.*

Proof. A maximum flow on the auxiliary network is a maximum matching in the original graph. This is because a flow of value k in the auxiliary network consists of k arc-disjoint[5] paths, each of which gives an edge in a matching in G. Now, since the maximum cardinality of a matching is $\max\{|X|, |Y|\} \leq \frac{n}{2}$, the number of iterations required to run `Ford-Fulkerson` is at most n. □

[5] Each arc has capacity 1.

Lemma 27. *Let S^* be a minimum capacity cut in the auxiliary network of an undirected graph G. Every arc in $\delta^+(S^*)$ is of the form (s, x_i) or (y_j, t).*

Proof. Suppose not. Then $\delta^+(S^*)$ would contain an infinite capacity arc, and so the minimum capacity of a cut in G would be infinite. But the maximum value of a flow on the auxiliary network is n since s has n outgoing arcs, a contradiction. □

Theorem 28 (Bipartite Vertex Cover). *If a vertex cover C and a matching M in a bipartite graph G have the same cardinality, then they are optimal,*

- *C is a minimum vertex cover*

- *M is a maximum matching*

Proof. Let $X^* = X \cap S^*$. Then $N(X^*) \subseteq S^* \cap Y$. Otherwise,

$$cap(S^*) = \sum_{a \in \delta^+(S^*)} u_a = \infty$$

$X^* \cup N(X^*) \cup (X - X^*) \cup (N(X - X^*))$ is a partition of $V(G)$. This means that there are three types of edges in G,

- Edges from X^* to $N(X^*)$
- Edges from $X - X^*$ to $N(X^*)$
- Edges from $X - X^*$ to $Y - N(X^*)$

We saw above that the fourth type of edge, $X - X^*$ to $N(X^*)$ cannot occur. But then $X - X^* \cup N(X^*)$ is a vertex cover and its cardinality is the capacity of the $(s - t)$ cut. □

Corollary 29. *There is a polynomial time algorithm to find a minimum cardinality vertex cover in a bipartite graph.*

Extensions to Maximum Flow

Definition 30 (Circulations with Demands)**.** *Let $G = (V, A)$ be a directed graph. We associate a demand d_v for flow with each node*[6]*,*

- *$S = \{s_1, \ldots, s_k\}$ is the set of sink nodes with positive demand*
- *$T = \{t_1, \ldots, t_l\}$ is the set of source nodes with negative demand*
- *s is a super-source with arcs (s, s_i) and capacities d_{s_i} for all $s_i \in S$*
- *t is a super-sink with arcs (t_j, t) and capacities $-d_{t_j}$ for all $t_j \in T$*

*A **circulation with demands** $f : A(G) \to \{0\} \cup \mathbb{R}^+$ satisfies,*

- *$0 \leq f_a \leq u_a$ for all $a \in A(G)$*
- *$\sum_{a \in \delta^-(v)} f_a - \sum_{a \in \delta^+(v)} f_a = d_v$ for all $v \in V(G) - \{s, t\}$*

There exists a feasible circulation to the multi-source supply and demand problem if and only if there is a flow from s to t of value $\sum_{v:d_v>0} d_v = 0$.

[6] If $d_v < 0$, this indicates that v has supply of $-d_v$. If $d_v = 0$, then the node is neither a source nor a sink.

Definition 31 (Circulations with Demands and Lower Bounds)**.** *Let $G = (V, A)$ be a directed graph. We associate a lower bound $l_a = l_{ij}$ on each arc $a = (i, j)$, where $0 \leq l_a \leq u_a$. A **circulation with demands and lower bounds** can be reduced to one without lower bounds,*

- *$l_e \leq f_a \leq u_a \iff 0 \leq f_a \leq u_a - l_a$ for all $a \in A(G)$*
- *$d_v^* = d_v + \sum_{(k,v) \in A} l_{kv} - \sum_{(v,k) \in A} l_{vk}$ for all $v \in V(G)$*

Example 4: Airline Scheduling

Origin and Destination	Time
Boston - San Francisco	7am - 9am
Toronto - Boston	7am - 8am

We create a network flow as follows,

- o_i is a vertex representing the origin of flight i
- d_i is a vertex representing the destination of flight i
- s is a source vertex with supply k
- t is a sink vertex with demand k
- (d_i, t) has capacity 1 for every flight i
- (d_i, o_j) has capacity 1 if flight i can be serviced after flight j
- (o_i, d_i) has a lower bound of 1

There is a way to perform all flights using at most k planes $\iff$ There is a feasible circulation in our network.

Example 5: Open-Pit Mining

We are given that,

- V is a set of blocks, each of which generates a profit π_i
- 3 upper neighbors must be removed to dig block i

We create a network flow as follows,

- $v_i \in V$ represents block i in our pit
- s and t are the source and sink vertices
- (s, v_i) has capacity π_i if and only if $\pi_i > 0$
- (v_i, t) has capacity $|\pi_i|$ if and only if $\pi_i < 0$
- Infinite capacity arcs exist between i and its 3 neighbors

We know that $cap(S^*)$ is finite because,

$$cap(\{s\}) = \sum_{i:\pi_i>0} \pi_i$$

is finite by construction. The capacity of the an $(s-t)$ cut S is,

$$\begin{aligned}
cap(S) &= \sum_{a \in \delta^+(S)} u_a \\
&= \sum_{i \notin S : \pi_i > 0} \pi_i + \sum_{i \in S : \pi_i > 0} |\pi_i| \\
&= \left(\sum_{i \in V : \pi_i > 0} \pi_i - \sum_{i \in S : \pi_i > 0} \pi_i \right) + \sum_{i \in S : \pi_i > 0} |\pi_i| \\
&= \left(\sum_{i \in V : \pi_i > 0} \pi_i - \sum_{i \in S : \pi_i > 0} \pi_i \right) - \sum_{i \in S : \pi_i > 0} \pi_i \\
&= \underbrace{\sum_{i \in V : \pi_i > 0} \pi_i}_{constant} - \sum_{i \in S} \pi_i
\end{aligned}$$

Minimizing $cap(S)$ solves the open-pit mining problem, since it requires us to maximize the profit associated with S.

Example 6: Image Segmentation

We create a network flow as follows,

- v_i is a vertex representing the pixel i
- s is a source vertex representing the foreground
- t is a sink vertex representing the background
- f_i is the likelihood that the pixel i is in the foreground
- b_i is the likelihood that the pixel i is in the background
- ρ_{ij} is a penalty for separating adjacent pixels i and j
- (s, v_i) has capacity f_i
- (v_j, t) has capacity b_j
- (v_i, v_j) and (v_j, v_i) have capacity ρ_{ij}

The capacity of the an $(s-t)$ cut S is,

$$\begin{aligned}
cap(S) &= \sum_{a \in \delta^+(S)} u_a \\
&= \sum_{i \notin S} f_i + \sum_{j \in S} b_j + \sum_{(i,j) \in \delta^+(S)} \rho_{ij} \\
&= \left(\sum_{i \in V} f_i - \sum_{i \in S} f_i \right) + \left(\sum_{j \in V} b_j - \sum_{j \notin S} b_j \right) + \sum_{(i,j) \in \delta^+(S)} \rho_{ij} \\
&= \underbrace{\sum_{i \in V} (f_i + b_i)}_{constant} - \sum_{i \in S} f_i - \sum_{j \notin S} b_j + \sum_{(i,j) \in \delta^+(S)} \rho_{ij}
\end{aligned}$$

Minimizing $cap(S)$ is solves the segmentation problem,

$$\max_S \sum_{i \in S} f_i + \sum_{j \notin S} b_j - \sum_{(i,j) \in \delta^+(S)} \rho_{ij}$$

- There is a bonus f_i if pixel i is placed in the foreground
- There is a bonus b_j if pixel j is placed in the background
- There is a penalty ρ_{ij} for separating pixels i and j

Recap on Flow Decomposition: Let $G = (V, A)$ be a directed graph and suppose that f is an $(s-t)$ flow of value k. Then there exist directed paths $P_1, \ldots, P_k$ (possibly repeated) from s to t in G, and every edge belongs to at most $f(e)$ paths.

Flow Decomposition Theorem

Lemma 32. *Let $G = (V, A)$ be a directed graph and suppose that f is an $(s-t)$ flow of value $k > 0$ Then f contains a path or a cycle.*

Proof. If f contains a directed cycle C, then we are done. Assume that this is not the case. Since $k > 0$,

$$\sum_{a \in \delta^+(\{s\})} f_a \geq 1$$

so there is at least one arc $a_1 = (s, v_1)$ with $f_{a_1} \geq 1$. By flow conservation, there exists an arc $a_2 = (v_1, v_2)$ ($v_2 \neq s$ since the flow is acyclic). Repeating inductively, we can construct an $(s-t)$ path with $\min_{a \in P} f_a \geq 1$. □

Theorem 33 (Flow Decomposition Theorem)**.** *Let $G = (V, A)$ be a directed graph and suppose that f is an $(s-t)$ flow of value $k > 0$. Then f can be decomposed into at most m $(s-t)$ paths and directed cycles.*

Proof. We proceed by induction on the number of arcs in the graph.

- (Base Case) If $m = 0$, then f is empty and trivially consists of zero paths and cycles. If $m = 1$, then f is a single arc (s, t) and thus can be decomposed into one (s, t) path.
- (Induction Hypothesis) Assume that any flow with $k < m$ arcs can be decomposed into a collection of at most k cycles and $(s-t)$ paths. Consider a flow f with m arcs. By Lemma 32, f contains a path or a cycle. There are two cases,
  1. If f contains a cycle C, then let f^* be the flow obtained by removing $\min_{a \in C} f_a$ units of flow from each arc in C. Then f^* has at least one fewer arc than f. By the induction hypothesis, f^* decomposes into at most $m - 1$ paths and cycles. Thus, f decomposes into at most m paths and cycles.

2. If f contains a path P, then let f^* be the flow obtained by removing $\min_{a \in C} f_a$ units of flow from each arc in P. Proceed as in Case 1 to decompose f^* into $m-1$ paths and cycles.

□

Corollary 34. *The Ford-Fulkerson algorithm can terminate in m iterations.*

Fast Flow Algorithms

Remark 35. *Let $G = (V, A)$ be a directed graph and suppose that f^* is an maximum $(s-t)$ flow on G. Then there is a path P in G with,*

$$u_P \geq \frac{1}{m}|f^*|$$

Proof. By the Flow Decomposition Theorem, f^* consists of at most m paths. Thus, at least one of these paths carries $\frac{1}{m}|f^*|$ flow. □

Remark 36. *Let $G = (V, A)$ be a directed graph and suppose that f^* is an maximum $(s-t)$ flow on G. Let f be any other $(s-t)$ flow on G. Then,*

$$b(P, f) \geq \frac{|f^*| - |f|}{m}$$

for some path in the residual graph G_f.

Proof. $f^* - f$ satisfies the flow conservation constraints. By the Flow Decomposition Theorem, $f^* - f$ can be decomposed into at most $2m$ paths and cycles. Only one direction of each arc is used, so we can assume that $(f^* - f)$ can be decomposed into at most m paths. Thus, at least one of these paths carries $\frac{|f^*|-|f|}{m}$. □

Definition 37 (Maximum Capacity)**.** *The **maximum capacity augmenting path algorithm** chooses augmenting paths greedily by capacity.*

(Recap) $1 - x < e^{-x}$ for all $x \neq 0$.

Theorem 38. *The maximum capacity augmenting path algorithm terminates in at most $m \cdot (\ln n + \ln U)$ iterations[7], where $U = \max_{a \in A} u_a$.*

[7] Weakly polynomial algorithms depend on the size of the input.

Proof. Suppose that the algorithm finds paths $\{P_1, P_2, \ldots, P_T\}$. We need to show that $T \leq m \cdot (\ln n + \ln U)$. Denote by f_t the flow after t iterations. By Remark 36, the path P_{t+1} satisfies,

$$b(P_{t+1}, f_t) \geq \frac{|f^*| - |f_t|}{m} := \frac{\triangle_{t+1}}{m}$$

$\triangle_{t+1}$ is the flow quantity that remains to be found at step $t+1$. In particular, $\triangle_{t+1} \leq \triangle_t - \frac{1}{m} \cdot \triangle_t$ since we fall by $\frac{1}{m}$ at each iteration. Continuing inductively, we get that $\triangle_{t+1} \leq (1 - \frac{1}{m})^t \cdot \triangle_1 = (1 - \frac{1}{m})^t \cdot |f^*|$. But then, $\triangle_{t+1} \leq e^{-\frac{t}{m}} \cdot |f^*|$. Setting $t = m \cdot ln|f^*|$,

$$\triangle_{t+1} < e^{-\frac{m \cdot ln|f^*|}{m}}|f^*| = 1$$

After $T = m \cdot \ln |f^*|$ steps, the quantity of flow remaining to be found is less than one. Flows are integral, so we have found $|f^*|$. Since the maximum cut in G has value $n \cdot U$, $\ln |f^*| \leq \ln u \cdot U = \ln u + \ln U$. Thus, the number of iterations is at most $m \cdot (\ln u + \ln U)$. □

Algorithm 2: Maximum Capacity Augmenting Paths

```
// f is globally given, and initially set to 0.
// Augment(P, f) is globally given
function Ford-Fulkerson(G)
    foreach a ∈ A(G) do
        f(a) ← 0
        G_f ← residual network of G with respect to f;
    while ∃(s − t) path P in G_f do
        P* ← maximum capacity augmenting path
         f ← Augment(P*, f)
        Update G_f;
    return f
```

Theorem 39. *The maximum capacity augmenting path algorithm takes $O(m^2)$ time per iteration. Binary search reduces it to $O(m \cdot \log m)$.*

Proof. Label the arcs in G_f by $\{1, 2, \ldots, 2m\}$ in decreasing order of residual capacity. We can test if there is an $(s - t)$ path that uses arcs $\{1, \ldots k\}$ in $O(m)$ time by Breadth-First Search. Repeating this process for all k gives a total time in $O(m^2)$. □

Corollary 40. *It total, the algorithm runs in $O(m^3 \cdot (\ln u + \ln U))$ time.*

Definition 41 (Shortest Length)**.** *The **shortest length residual path algorithm** chooses augmenting path greedily by length*[8]*.*

[8] The length of a path is the number of arcs comprising it.

Remark 42. *An iteration of the shortest length augmenting path algorithm is in $O(m)$ if the shortest $(s - t)$ paths are found with Breadth-First Search.*

Theorem 43. *The shortest length augmenting path algorithm terminates in at most $m \cdot n$ iterations. This puts its runtime in $O(m^2 \cdot n)$.*

Proof. Suppose that the algorithm finds paths $\{P_1, P_2, \ldots, P_T\}$. We need to show that $T \leq m \cdot n$. Let f_t be the flow after t iterations, and consider its residual graph G_{f_t}. Define $d_t(v)$ as the distance of a vertex v from the source s in G_{f_t}, and consider $\phi_t = \sum_{v \in V} d_t(v)$. We will use a potential energy function argument to show that,

$$d_t(u) \leq d_{t+1}(u) \quad \text{for all } t$$

The Potential Function Argument:
If ϕ satisfies,

- ϕ decreases at a rate of δ
- ϕ is lower bounded by ℓ

Then ϕ becomes fixed at time T. Given the starting value ϕ_0 and ℓ, we know that ϕ becomes fixed at $-\phi_0 * |\delta| = \phi$.

If $t = 0$, then $\phi_0 = \sum_{v \in V} d_0(v) \geq 0$ since d_0 is a distance. Moreover, $d_t(v) \leq n - 1$ since G has n vertices. This means that,

$$\phi_t = \sum_{v \in V} d_t(v) \leq n^2$$

To apply the argument, we need to prove that ϕ_t is non-decreasing. It suffices to show that $d_t(v)$ is non-decreasing for each vertex v. Suppose not. Then there exists $t \in \mathbb{N}$ and a vertex $v \in V(G)$ such that $d_t(v) > d_{t+1}(v)$. Consider the closest such vertex to s. Let u be the vertex preceding it on the shortest path from s to v in $G_{f_{t+1}}$. Then $d_{t+1}(v) = d_{t+1}(u) + 1$ and $d_t(u) \leq d_{t+1}(u)$.

- If $(u, v) \in G_{f_t}$, then,

$$\begin{aligned} d_t(v) &\leq d_t(u) + 1 \\ &\leq d_{t+1}(u) + 1 \text{ since } d_t(u) \leq d_{t+1}(u) \\ &= d_{t+1}(v) \text{ since } d_{t+1}(v) = d_{t+1}(u) + 1 \\ &< d_{t+1}(v) \text{ since } d_t(v) > d_{t+1}(v) \end{aligned}$$

  a contradiction.

- If $(u, v) \notin G_{f_t}$, $(v, u) \in P_t$ since $(u, v) \in G_{f_{t+1}}$. But,

$$\begin{aligned} d_t(u) &= d_t(v) + 1 \text{ since } P_t \text{ is the shortest path in } G_{f_t} \\ &> d_{t+1}(v) + 1 \\ &= d_{t+1}(u) + 2 \text{ since } d_{t+1}(v) = d_{t+1}(u) + 1 \\ &\geq d_t(u) + 2 \end{aligned}$$

  a contradiction.

P_t is augmented by its bottleneck capacity, and there is at least one arc a_t with that capacity. It suffices to prove that each arc can be the bottleneck arc at most $\frac{n}{2}$ times[9], because this means that the number of iterations is at most $2m \cdot \frac{n}{2} = mn$. □

[9] This argument works because we have proven that ϕ is a bounded function.

Lemma 44. *Each arc can be the bottleneck arc at most $\frac{n}{2}$ times.*

Proof. Let $(u, v) \in A(G_{f_t})$ be the bottleneck arc in the augmenting path P_t. Since P_t is the shortest $(s - t)$ path in G_{f_t}, we have that $d_t(v) = d_t(u) + 1$. Now, $(u, v) \notin G_{f_{t+1}}$ since (u, v) is the bottleneck arc in iteration t. Suppose that (u, v) is a bottleneck in iteration[10] $t + 1 + k > t + 1$ ($k > 0$). Then (u, v) must have been added back into the residual graph $G_{f_{t+1+k}}$. This means that the reverse arc (v, u) must have been used at some time $t + 1 \leq t' < t + 1 + k$. As $P_{t'}$ is the shortest $(s - t)$ path in $G_{f_{t'}}$, we have:

[10] (u, v) cannot be a bottleneck at iteration $t + 1$ because it is not in the residual graph $G_{f_{t+1}}$.

$$\begin{aligned} d_{t'}(u) &= d_{t'}(v) + 1 \\ &\geq d_t(v) + 1 \\ &= d_t(u) + 2 \end{aligned}$$

Since the distance label of u must have increased by at least 2, this can happen at most $\frac{n}{2}$ times. Otherwise, we get that $d_t(u) > n$. □

Algorithm 3: Shortest Length Augmenting Paths

```
// f is globally given, and initially set to 0.
// Augment(P, f) is globally given
function Ford-Fulkerson(G)
    foreach a ∈ A(G) do
        f(a) ← 0
        G_f ← residual network of G with respect to f;
    while ∃(s − t) path P in G_f do
        P* ← shortest length augmenting path
         f ← Augment(P*, f)
        Update G_f;
    return f
```

Linear Programming

Formulating Linear Programming Problems

Example 7: The Diet Problem

A student is subject to the following dietary requirements,

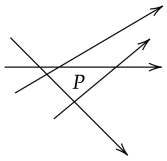
Food	Energy	Protein	Calcium	Price	Servings
Porridge	110	4	2	10	4
Chicken	205	32	12	200	3
Eggs	160	13	54	40	4
Milk	160	8	285	35	8
Apple Pie	420	4	22	190	3
Pork and Beans	260	14	80	90	2
Requirement	2000	55	800	None	None

The goal is to find,

$$\min 10x_1 + 200x_2 + 40x_3 + 35x_4 + 190x_5 + 90x_6$$

subject to the constraints,

$$\begin{aligned} x_i &\geq 0 \text{ for all } i \in [6] \\ x_1 \leq 4, x_2 \leq 3, x_3 \leq 4, x_4 \leq 8, x_5 \leq 3, x_6 &\leq 2 \\ 110x_1 + 205x_2 + 160x_3 + 160x_4 + 420x_5 + 260x_6 &= 2000 \\ 4x_1 + 32x_2 + 13x_3 + 8x_4 + 4x_5 + 14x_6 &= 55 \\ 2x_1 + 12x_2 + 54x_3 + 285x_4 + 22x_5 + 80x_6 &= 800 \end{aligned}$$



Any linear equation in n variables defines a hyperplane in $\mathbb{R}^n$, and the intersection of a finite number of hyperplanes is called a polyhedron. By rotating the space so that the objective function points downward, any linear program is the problem of finding the lowest point in a given polyhedron.

Definition 45 (Linear Program). *A **linear program** minimizes or maximizes a linear objective function subject to a set of linear constraints.*

Definition 46 (Feasibility). *A point $x \in \mathbb{R}^n$ is **feasible** with respect to some linear program if it satisfies all of the linear constraints.*

Definition 47 (Primal Linear Program). *A primal linear program is,*

$$\max \left\{ \sum_{j=1}^{n} c_j \cdot x_j \;\middle|\; \begin{array}{l} \sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_j \text{ for all } i \in [m] \\ x_j \geq 0 \text{ for all } j \in [n] \end{array} \right\}$$

Remark 48. *The matrix encoding of a linear program in standard form is,*

$$\max \left\{ c^T \cdot x \;\middle|\; \begin{array}{l} Ax \leq b \\ x \geq 0 \end{array} \right\}$$

Example 8: Standard Form

Any linear program can be converted into standard form,

$$\begin{array}{ll} \text{maximise} & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \forall i \in [m] \\ & x_j \geq 0 \quad \forall j \in [n] \end{array}$$

- For each variable x_j, add:
  - Two new variables x_i^+, x_i^-
  - Recover x_j by taking, $x_j = x_j^+ - x_j^-$
  - Inequalities, $x_j^+ \geq 0$ and $x_j^- \geq 0$
- Replace any equality constraint $\sum_j a_{ij} x_j = b_i$ with:
  - An inequality constraint, $\sum_j a_{ij} x_j \geq b_i$
  - An inequality constraint $\sum_j a_{ij} x_j \leq b_i$
- Replace any upper bound constraint $\sum_j a_{ij} x_j \geq b_i$ with,
  - The equivalent lower bound, $\sum_j -a_{ij} x_j \leq -b_i$

Definition 49 (Dual Linear Program)**.** *For every primal linear program,*

$$\begin{array}{lll} \text{maximize} & \sum_{j=1}^{n} c_j \cdot x_j & \\ \textit{subject to} & \sum_{j=1}^{n} a_{ij} \cdot x_j & \leq b_i \quad \forall i \in [m] \\ & x_j & \geq 0 \quad \forall j \in [n] \end{array}$$

there is a corresponding ***dual linear program****,*

$$\begin{array}{lll} \text{minimize} & \sum_{i=1}^{m} b_i \cdot y_i & \\ \textit{subject to} & \sum_{i=1}^{m} a_{ij} \cdot y_i & \geq c_j \quad \forall j \in [n] \\ & y_i & \geq 0 \quad \forall i \in [m] \end{array}$$

that can be obtained by swapping the constraints and the variables.

Remark 50. *The matrix encoding of the dual linear program is,*

$$\min \underbrace{\left\{ b^T y \;\middle|\; \begin{array}{l} A^T y \geq c \\ y \geq 0 \end{array} \right\}}_{\textit{Dual}} \iff \max \underbrace{\left\{ c^T x \;\middle|\; \begin{array}{l} Ax \leq b \\ x \geq 0 \end{array} \right\}}_{\textit{Primal}}$$

Definition 51 (Dictionaries)**.** *Given a linear program in standard form,*

$$\max \left\{ \sum_{j=1}^{n} c_j \cdot x_j \;\middle|\; \begin{array}{l} \sum_{j=1}^{n} a_{ij} \cdot x_j \leq b_j \textit{ for all } i \in [m] \\ x_j \geq 0 \textit{ for all } j \in [n] \end{array} \right\}$$

we introduce ***slack variables***[11] $x_{n+1}, x_{n+2}, \cdots, x_{n+m}$*,*

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} \cdot x_j \quad \textit{for } i \in [m] \textit{ and } \quad z = \sum_{j=1}^{n} c_j \cdot x_j$$

and denote the objective function by z*.*

[11] Slack variables are variables that has been introduced to turn an inequality constraint into an equality constraint.

Definition 52 (Basic Variable)**.** ***Basic*** *variables are the variables that appear on the left-hand side of a dictionary, and they constitute a basis.*

Definition 53 (Nonbasic Variable)**.** ***Nonbasic*** *variables are the variables that appear on the right-hand side of a dictionary.*

Remark 54. *Dictionaries define basic variables in terms of non-basic ones.*

The Simplex Algorithm: Method

Definition 55 (Simplex Algorithm)**.** *The* ***simplex algorithm*** *works as the Gauss-Jordan elimination method on inequalities and constraints,*

- *Represent the linear program in slack form*
- *Convert one slack form into an equivalent slack form, ensuring that the value of the objective function does not decrease while likely increasing it*
- *Repeat until the optimal solution becomes apparent*

Example 9: Slack Variables

Take the linear program,

$$\begin{array}{ll} \text{maximize} & 5x_1 + 4x_2 + 3x_3 \\ \text{subject to} & 2x_1 + 3x_2 + x_3 \leq 5 \\ & 4x_1 + x_2 + 2x_3 \leq 11 \\ & 3x_1 + 4x_2 + 2x_3 \leq 8 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

Write it in dictionary form,

$$\begin{array}{lcrcrcrcr} z & = & & & 5x_1 & + & 4x_2 & + & 3x_3 \\ \hline x_4 & = & 5 & - & 2x_1 & - & 3x_2 & - & x_3 \\ x_5 & = & 11 & - & 4x_1 & - & x_2 & - & 2x_3 \\ x_6 & = & 8 & - & 3x_1 & - & 4x_2 & - & 2x_3 \end{array}$$

and restate our problem as,

$$\begin{array}{ll} \text{maximize} & z \\ \text{subject to} & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{array}$$

Example 10: The Simplex Algorithm

The **feasible solution** that is implicit in our dictionary is,

$$(x_1, x_2, x_3, x_4, x_5, x_6) = (0, 0, 0, 5, 11, 8)$$

This gives an **objective value** of,

$$z = 5x_1 + 4x_2 + 3x_3 = 0$$

In the first iteration, we attempt to increase the value of z by making one of the right-hand side variables positive.

Definition 56 (Entering Variable)**.** *The **entering variable** at each iteration is the nonbasic variable that enters the basis to increase z*[12]*.*

Definition 57 (Leaving Variable)**.** *The **leaving variable** at each iteration is the variable that is removed from the basis for the entering variable*[13]*.*

Example 11: The Simplex Algorithm

The entering variable in the first iteration is x_1. Since x_1, x_2, x_3 are all positive, we choose the variable with the largest coefficient in order to increase z at the fastest rate.

Linear Programming Algorithms:

- *Simplex Algorithm.* In the feasible region, x moves from vertex to vertex in the direction of c. The algorithm is simple, but runs in exponential time in the worst case.
- *Ellipsoid Algorithm.* The algorithm begins with an ellipsoid that includes the optimal solution, and continues to shrink the ellipsoid until the optimal solution is found.
- *Interior Point Method.* x moves inside the polytope following c.

[12] We may choose any nonbasic variable with a positive coefficient in the top row of the dictionary.

[13] We may choose any basic variable whose non-negativity imposes the most stringent upper bound on the increment of the entering variable.

$$\begin{array}{rcrcrcrcr} z & = & \frac{25}{2} & - & \frac{7}{2}x_2 & + & \frac{1}{2}x_3 & - & \frac{5}{2}x_4 \\ \hline x_1 & = & \frac{5}{2} & - & \frac{3}{2}x_2 & - & \frac{1}{2}x_3 & - & \frac{1}{2}x_4 \\ x_5 & = & 1 & + & 5x_2 & & & + & 2x_4 \\ x_6 & = & \frac{1}{2}x_2 & + & \frac{1}{2}x_3 & - & \frac{1}{2}x_3 & + & \frac{3}{2}x_4 \end{array}$$

This completes the first iteration of the simplex method, and,

$$(x_1, x_2, x_3, x_4, x_5, x_6) = \left(\frac{5}{2}, 0, 0, 0, 1, \frac{1}{2}\right)$$

The Simplex Algorithm: Termination

Definition 58 (Cycling)**.** *The simplex algorithm **cycles** if one dictionary appears in two or more iterations. Cycling prevents termination.*

Definition 59 (Smallest Subscript Rule)**.** *The **smallest-subscript rule** is a rule for breaking ties in the choice of the entering and leaving variables. It always chooses the candidate x_k by the smallest subscript k.*

Theorem 60 (Bland's Rule)**.** *The simplex method terminates as long as the entering and leaving variables are selected by the smallest-subscript rule.*

Proof. Assume not. Then there exists a sequence of degenerate iterations that produces dictinaries $D_1, D_2, \cdots D_k$ such that $D_k = D_0$. A variable is called **fickle** if it is nonbasic in some dictionaries, but basic in others. Let x_t be the fickle variable with the largest subscript. Then there is a dictionary D in the sequence $D_0, \cdots, D_k$ such that x_t leaves and some other fickle variable x_s enters. Further along,

$$\underbrace{D_0}_{=D_k}, D_1, \cdots, D_k, D_1, \cdots, D_k$$

there is another dictionary D^* with x_t entering,

$$x_i = b_i - \sum_{j \notin B} a_{ij} x_j$$
$$z = v + \sum_{j \notin B} c_j x_j.$$

where $i \in B$. Since all iterations from D to D^* are degenerate, the objective function z has the same value v in both dictionaries.

$$z = v + \sum_{j=1}^{n+m} c_j^* x_j \quad \text{is the last row}$$

with $c_j^* = 0$ whenever x_j is basic in D^*. This equation has been obtained from D by algebraic manipulations, so it must satisfy every solution of the linear program[14]. In particular, it must be satisfied by,

[14] Moreover, the simplex algorithm remains on the boundary of the feasible region at every iteration.

$$
\begin{aligned}
x_s &= y \\
x_j &= 0 \quad (j \notin B, j \neq s) \\
x_i &= b_i - a_{is} y \quad (i \in B) \\
z &= v + c_s y \quad \forall y
\end{aligned}
$$

Thus, for every choice of y,

$$v + c_s y = v + c_s^* y + \sum_{i \in B} c_i^* \left(b_i - a_{is} y\right)$$

and then,

$$\underbrace{\left(c_s - c_s^* + \sum_{i \in B} c_i^* a_{is}\right)}_{constant} y = \underbrace{\sum_{i \in B} c_i^* b_i}_{constant}$$

We have an equation $\lambda_1 \cdot y = \lambda_2$ with y variable. Thus,

$$
\begin{aligned}
(\star) \quad c_s - c_s^* + \sum_{i \in B} c_i^* a_{is} &= 0 \\
\sum_{i \in B} c_i^* b_i &= 0
\end{aligned}
$$

Since x_s is entering in D, $c_s > 0$. Since x_s is not entering in D^* and $s < t$ (by assumption), $c^* \leq 0$. If not, then by the Smallest-Subscript Rule x_i would be entering. Hence, by $(\star)$,

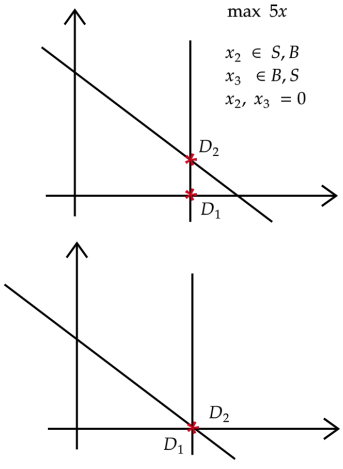
$$c_r^* a_{rs} < 0 \quad \text{for } r \in B$$

Since $r \in B$, x_r is basic in D. Since $c_r^* \neq 0$, x_r is nonbasic in D^*. Hence, x_r is fickle and $r \leq t$. More simply, $r \neq t$ or else $c_t^* a_{ts} > 0$ because $a_{ts} > 0$. Now, $c_r^* > 0$ since x_r is not entering in D^* (x_t is entering). Therefore,

$$a_{rs} > 0$$

to satisfy $c_r^* a_{rs} < 0$. All iterations from D to D^* are degenerate, so both dictionaries satisfy the same solution. In particular x_r is zero since it is non-basic D^*. Therefore, $b_r = 0$ and x_r was a candidate for leaving the basis of D. This is a contradiction because $r < t$. □

Example of Degeneracy:
In the example below, x_2, x_3 are fickle.



The Simplex Algorithm: Initialization

Definition 61 (Auxiliary Problem)**.** *Given a linear program,*

$$
\begin{aligned}
\text{maximize} \quad & \textstyle\sum_{j=1}^{n} c_j x_j \\
\textit{subject to} \quad & \textstyle\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (i \in [m]) \\
& x_j \geq 0 \quad (j \in [n])
\end{aligned}
$$

*the **auxiliary problem** is defined as,*

$$
\begin{aligned}
\text{minimize} \quad & x_0 \\
\textit{subject to} \quad & \textstyle\sum_{j=1}^{n} a_{ij} x_j - x_0 \leq b_i \quad (i \in [m]) \\
& x_j \geq 0 \quad (j \in [n])
\end{aligned}
$$

Example 12: Initialization

The **all-zero** solution is not always feasible,

$$\begin{array}{lrcrcrcr} \text{maximize} & x_1 & - & x_2 & + & x_3 & & \\ \text{subject to} & 2x_1 & - & x_2 & + & 2x_3 & \leq & 4 \\ & 2x_1 & - & 3x_2 & + & x_3 & \leq & -5 \\ & -x_1 & + & x_2 & - & 2x_3 & \leq & -1 \\ & & & x_1, & & x_2, \quad x_3 & \geq & 0 \end{array}$$

Remark 62. *The original problem has a feasible solution if and only if the optimum value of the auxiliary problem is zero*[15].

[15] We solve the auxiliary problem first.

The Simplex Algorithm: Efficiency

Theorem 63. *With Bland's Rule, the simplex algorithm terminates in,*

$$\binom{m+n}{m} >> 2^{m+n} \quad \textit{iterations.}$$

(Conjecture) The simplex algorithm is polynomial time if, given a choice, it chooses the pivot variables at random.

Proof. This follows from the fact that there are $\binom{m+n}{m}$ bases cases, and the dictionary corresponding to each basis can only appear once. □

Remark 64. *The simplex algorithm typically makes $O(m)$ pivots, where m is the number of constraints*[16]*. Each pivot takes $O(mn)$ with dictionaries.*

[16] Since the number of constraints in the dual is the number of variables in the primal, $O(n)$ pivots are typically needed to solve the dual. If $n < m$, taking the dual gives a quicker solution.

Linear Programming Duality

Theorem 65 (Weak Duality Theorem)**.** *For any primal feasible solution x and any dual feasible solution y, we have $c^T x \leq b^T y$.*

Proof. Using the matrix encodings[17] of the linear programs x and y,

$$\begin{aligned} c^T \cdot x &\leq (A^T y)^T \cdot x \text{ since } A^T y \geq c \text{ and taking the transpose} \\ &= (y^T A) \cdot x \\ &= y^T \cdot (Ax) \\ &\leq y^T \cdot b \text{ by the dual feasibility of } x \end{aligned}$$

□

[17] $\geq$ applies to all entries.

Example 13: Duality

Consider the primal,

$$\begin{array}{lrcrcrcrcl} \text{maximise} & 4x_1 & + & x_2 & + & 5x_3 & + & 3x_4 & & \\ \text{subject to} & x_1 & - & x_2 & - & x_3 & + & 3x_4 & \leq & 1 \\ & 5x_1 & + & x_2 & + & 3x_3 & + & 8x_4 & \leq & 55 \\ & -x_1 & + & 2x_2 & + & 3x_3 & - & 5x_4 & \leq & 3 \\ & & & & & x_1, & x_2, & x_3 \ , x_4 & \geq & 0 \end{array}$$

and its dual,

$$\begin{array}{lrcrcrcl} \text{minimize} & y_1 & + & 55y_2 & + & 3y_3 & & \\ \text{subject to} & y_1 & + & 5y_2 & - & y_3 & \geq & 4 \\ & -y_1 & + & y_2 & + & 2y_3 & \geq & 1 \\ & -y_1 & + & 3y_2 & + & 3y_3 & \geq & 5 \\ & 3y_1 & + & 8y_2 & - & 5y_3 & \geq & 3 \\ & & & y_1, & y_2 & y_3 & \geq & 0 \end{array}$$

The primal has optimal solution,

$$(x_1, x_2, x_3, x_4) = (0, 14, 0, 5) = 29$$

The dual has optimal solution,

$$(y_1, y_2, y_3) = (11, 0, 6) = 29$$

Theorem 66 (Strong Duality Theorem)**.** *Let x be an optimal primal solution and y be an optimal dual solution*[18]*. Then $c^T x = b^T y$.*

[18] An optimal solution might not exist:

- The feasible region is empty
  - $x_1 \leq 1$
  - $x_2 \geq 2$
- The optimal value is infinite
  - $\max x_1 + x_2$
  - $x_1, x_2 \geq 0$

Proof. The simplex algorithm finds the optimal primal solution. It has decision variables $(x_1^*, \cdots, x_n^*)$ and slack variables $(x_{n+1}^* \cdots, x_{n+m}^*)$. The top row of the final dictionary is,

$$z = \texttt{OPT} - \sum_{k=1}^{n+m} \gamma_k x_k$$

where $\gamma_k \geq 0$, with equality for basis variables. Define dual variables $y_i^* = \gamma_{n+i}$ for $i \in [m]$. We need to show that,

1. $(y_1^*, \cdots, y_m^*)$ is a feasible solution for the dual,

$$\sum_{i=1}^{m} a_{ij} y_i^* \geq c_j \quad \forall j \in [n]$$

2. The value of $(y_1^*, \cdots, y_m^*)$ is equal to the optimal primal value,

$$\sum_{j=1}^{n} c_j \cdot x_j^* = \sum_{i=1}^{m} b_i \cdot y_i^*$$

Let $z^* = \texttt{OPT(primal)} = \sum_{j=1}^{n} c_j \cdot x_j^*$. The final dictionary states that,

$$
\begin{aligned}
z &= \sum_{j=1}^{n} c_j \cdot x_j^* - \sum_{k=1}^{n+m} \gamma_k \cdot x_k \\
&= z^* - \sum_{k=1}^{n+m} \gamma_k \cdot x_k \\
&= \texttt{OPT(primal)} - \sum_{k=1}^{n+m} \gamma_k \cdot x_k \\
&= \texttt{OPT(primal)} - \sum_{k=1}^{n} \gamma_k \cdot x_k \; - \sum_{k=n+1}^{n+m} \gamma_k \cdot x_k
\end{aligned}
$$

Re-indexing the sum to work with the dual variables,

$$= \texttt{OPT(primal)} - \sum_{j=1}^{n} \gamma_j \cdot x_j \; - \sum_{i=1}^{m} \gamma_{n+i} \cdot x_{n+1}$$

Since $y_i^* = \gamma_{n+i}$ for $i \in [m]$ by definition of the optimal dual solution,

$$= \texttt{OPT(primal)} - \sum_{j=1}^{n} \gamma_j \cdot x_j \; - \sum_{i=1}^{m} y_i^* \cdot x_{n+1}$$

But x_{n+i} are our slack variables,

$$= \texttt{OPT(primal)} - \sum_{j=1}^{n} \gamma_j \cdot x_j \; - \sum_{i=1}^{m} y_i^* \cdot \left(b_i - \sum_{j=1}^{n} a_{ij} \cdot x_j \right)$$

Re-arranging,

$$= \left(\texttt{OPT(primal)} - \sum_{i=1}^{m} b_i \cdot y_i^* \right) + \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i^* - \gamma_j \right) \cdot x_j$$

Since this equality holds for all choices of $\{x_1, \cdots x_n\}$,

$$c_j = \sum_{i=1}^{m} a_{ij} \cdot y_i^* - \gamma_j$$

But $\gamma_j \geq 0$, so we have feasibility,

$$\sum_{i=1}^{m} a_{ij} \cdot y_i^* \geq c_j$$

and it must be that,

$$\texttt{OPT(primal)} - \sum_{i=1}^{m} b_i \cdot y_i^* = 0$$

so we have strong duality,

$$\sum_{i=1}^{m} b_i \cdot y_i^* = 0 = \texttt{OPT(primal)} = \sum_{j=1}^{n} c_j \cdot x_j^*$$

□

Remark 67. *The primal is infeasible if the optimal of the dual is $-\infty$ or ∞.*

Theorem 68 (Completemenary Slackness Theorem)**.** *TFAE,*

- *x and y are optimal solutions*
- *$c \cdot x = y \cdot b$*
- *$c \cdot x = yAx$*
- *$yAx = y \cdot b$*
- *x and y satisfy the complementary slackness conditions*

Proof. This is a result of strong duality. □

Applications of Linear Programming

Example 14: Matching as a Motivating Example

Let $G = (V, E)$ be an undirected graph. The integer formulation of the **matching problem** is,

- An edge e in the graph is picked if $x_e = 1$,

$$x_e \in \{0, 1\} \quad \forall e \in E$$

- We want to maximize the number of edges in the matching,

$$\max \sum_{e \in E} x_e$$

- One edge is incident to each vertex,

$$\sum_{e \in \delta(v)} x_e \leq 1 \quad \forall e \in E$$

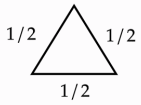
In summary,

$$\begin{array}{ll} \text{maximize} & \sum_{e \in E} x_e \\ \text{subject to} & \sum_{e \in \delta(v)} x_e \leq 1 \qquad \forall v \in V \\ & x_e \in \{0, 1\} \qquad \forall e \in E \end{array}$$

We do not have a polynomial time algorithm for solving integer programs. If we relax the integrality constraints,

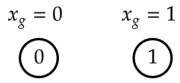
$$\begin{array}{ll} \text{maximize} & \sum_{e \in E} x_e \\ \text{subject to} & \sum_{e \in \delta(v)} x_e \leq 1 \qquad \forall v \in V \\ & x_e \geq 0 \qquad \forall e \in E \end{array}$$

we may obtain a fractional solution,

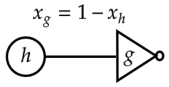


Linear programs can model **Boolean combinatorial circuits**. For each gate g, there is a variable $0 \leq x_g \leq 1$. Then,

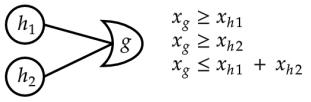
- `INPUT` gates are set to their input



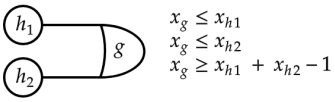
- `NOT` gates are set to the opposite



- `OR` gates are set $\max\{x_{h_1}, x_{h_2}\}$



- `AND` gates are set to $\min\{x_{h_1}, x_{h_2}\}$



Theorem 69 (Bipartite Matching). *Bipartite graphs have an optimal integral solution to the following linear program,*

$$\begin{array}{lll} \text{maximize} & \sum_{e \in E} x_e & \\ \textit{subject to} & \sum_{e \in \delta(v)} x_e \leq 1 & \forall v \in V \\ & x_e \geq 0 & \forall e \in E \end{array}$$

Recall the following,

- The simplex algorithm outputs a basic solution, which is an extreme point of the feasible region.
- The feasible region is the convex hull of the extreme points.
- A basic solution is not a convex combination of two feasible ones.

where a **convex combination** is a linear combination of points where all coefficients are non-negative and sum to 1.

Proof. Let x be a basic solution to our linear program. We need to show that $x_e \in \{0,1\}$ for all $e \in E$. Suppose not. Then, $x_e \in (0,1)$ for all $e \in E$. If $x_e = 0$, then we can recurse on $G - \{e\}$ to produce a strictly fractional collection of x_e values. Similarly, if $x_e = 1$, then we can recurse on $G - e$ since no edge adjacent to e will produce a matching on this subgraph.

- **Case 1.** G contains a cycle C.
  - C contains an even number of edges, and therefore we can partition C into two equal sized matchings M_1, M_2
  - Define two linear programs y, z as follows,

$$y_e = \begin{cases} x_e + \epsilon & \text{if } e \in M_1 \\ x_e - \epsilon & \text{if } e \in M_2 \\ x_e & \text{if } e \notin C \end{cases} \qquad z_e = \begin{cases} x_e - \epsilon & \text{if } e \in M_1 \\ x_e + \epsilon & \text{if } e \in M_2 \\ x_e & \text{if } e \notin C \end{cases}$$

  - Observe that x is a convex combination of y and z,

$$x = \frac{1}{2} \cdot (y + z)$$

  - Furthermore, x is fractional,

$$0 < x_e < 1 \quad \forall e \in E$$

  - Thus, we can pick ϵ such that both y and z are fractional,

$$0 < y_e < 1 \quad \forall e \in E \qquad 0 < z_e < 1 \quad \forall e \in E$$

  - Adding and subtracting ϵ along the red, we see that,

$$\forall v \in V \quad \sum_{e \in \delta(v)} x_e = \sum_{e \in \delta(v)} y_e = \sum_{e \in \delta(v)} z_e \leq 1$$

  - This implies that x is a convex combination of two feasible solutions. Thus, x is not a basic solution.
- **Case 2.** G is acyclic, i.e., G is a forest.

– Let P be a maximal path in G. We can partition P into two matchings P_1, P_2 and define two linear programs y, z,

$$y_e = \begin{cases} x_e + \epsilon & \text{if } e \in P_1 \\ x_e - \epsilon & \text{if } e \in P_2 \\ x_e & \text{if } e \notin P \end{cases} \qquad z_e = \begin{cases} x_e - \epsilon & \text{if } e \in P_1 \\ x_e + \epsilon & \text{if } e \in P_2 \\ x_e & \text{if } e \notin P \end{cases}$$

– Repeating the same argument from Case 1 with $x = \frac{1}{2} \cdot (y + z)$,

$$0 < y_e < 1 \quad \forall e \in E \qquad 0 < z_e < 1 \quad \forall e \in E$$

– For any vertex $v \in V\{v_1, v_2\}$,

$$\forall v \in V \setminus \{v_1, v_2\} \quad \sum_{e \in \delta(v)} x_e = \sum_{e \in \delta(v)} y_e = \sum_{e \in \delta(v)} z_e \leq 1$$

– v_1 is necessarily a leaf, meaning that it is incident to one edge,

$$\sum_{e \in \delta(v_1)} y_e < 1 \quad \sum_{e \in \delta(v_1)} z_e < 1$$

– Since the same holds for v_2, x is a convex combination of two feasible solutions. Thus, x is not a basic solution.

Hence, the optimal solution to the program is integral. □

Corollary 70. *A similar argument shows that we can solve the* ***maximum weight matching problem*** *for bipartite graphs in polynomial time,*

$$\begin{array}{lll} \text{maximize} & \sum_{e \in E} w_e \cdot x_e & \\ \textit{subject to} & \sum_{e \in \delta(v)} x_e \leq 1 & \forall v \in V \\ & x_e \geq 0 & \forall e \in E \end{array}$$

A is the incidence matrix,

$$\begin{array}{c} \\ v_1 \\ \dots \\ \dots \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & \dots & e & \dots \end{array} \\ \begin{pmatrix} & & 1 & \\ 1 & \dots & 0 & 0 \\ & & \dots & \\ & & 0 & \end{pmatrix} \end{array}$$

Remark 71. *The matrix encoding of the dual linear program is,*

$$\min \underbrace{\left\{ 1^T y \;\middle|\; \begin{array}{l} A^T y \geq c \\ y \geq 0 \end{array} \right\}}_{\textit{Dual}} \iff \max \underbrace{\left\{ 1^T x \;\middle|\; \begin{array}{l} Ax \leq 1 \\ x \geq 0 \end{array} \right\}}_{\textit{Primal}}$$

allowing us to re-interpret the dual as the ***vertex cover problem****,*

$$\begin{array}{lll} \text{minimize} & \sum_{v \in V} y_v & \\ \textit{subject to} & y_u + y_v \geq 1 & \forall (u, v) \in E \\ & y_v \geq 0 & \forall v \in V \end{array}$$

Example 15: The Shortest Path Problem

Given a directed graph $G = (V, A)$ with a source vertex s and a sink vertex t, we can set up the shortest path problem,

- Let each arc $a \in A$ have non-negative length l_a
- The length of a path P is,

$$l(P) = \sum_{a \in P} l_a$$

We can formulate this as an **integer program**,

$$\begin{array}{ll} \text{minimize} & \sum_{a \in A} \ell_a \cdot x_a \\ \text{subject to} & x_a \in \{0,1\} \quad \forall a \in A \\ & \sum_{a \in \delta^-(v)} x_a - \sum_{a \in \delta+(v)} x_a = \begin{cases} -1 & v = s \\ 0 & v \neq \{s,t\} \\ 1 & v = t \end{cases} \end{array}$$

and do a constraint relaxation.

Theorem 72 (Shortest Path)**.** *The shortest path problem has an optimal integral solution to the following linear program,*

$$\begin{array}{ll} \text{minimize} & \sum_{a \in A} \ell_a \cdot x_a \\ \textit{subject to} & x_a \geq 0 \quad \forall a \in A \\ & \sum_{a \in \delta^-(v)} x_a - \sum_{a \in \delta+(v)} x_a = \begin{cases} -1 & v = s \\ 0 & v \neq \{s,t\} \\ 1 & v = t \end{cases} \end{array}$$

Proof. Any basic solution x represents an $(s-t)$ flow of value 1. By the Flow Decomposition Theorem, x decomposes into (s,t) paths,

$$\mathbf{x} = \alpha_1 \cdot P_1 + \alpha_2 \cdot P_2 + \cdots + \alpha_k \cdot P_k$$
$$\text{where } \sum_{i=1}^{k} \alpha_i = 1 \text{ and } \alpha_i \geq 0 \quad \forall i$$

□

but each P_i is a feasible solution to the linear program. Either we have a contradiction, or $k = 1$. If $k = 1$, then x is an $(s-t)$ path itself and is an integral solution.

Remark 73. *The matrix encoding of the dual linear program, even though the primal is not in standard form, is,*

$$\max \left\{ y_t - y_s \;\middle|\; \begin{array}{ll} y_v \textit{ unrestricted} & \forall v \in V \\ A^T y \leq l & \end{array} \right\}$$

A_{ij} is negative if v_i is the tail of a_j,

$$\begin{array}{c} \\ v_1 \\ \dots \\ \dots \end{array} \begin{array}{c} \begin{array}{cccccc} a_1 & \dots & a & \dots & & \end{array} \\ \begin{pmatrix} & & -1 & & \\ 1 & \dots & 0 & -1 & \\ & & \dots & & \\ & & 0 & & \end{pmatrix} \end{array}$$

Changing labels of y to d to represent distances,

$$\max \left\{ d_t - d_s \;\middle|\; \begin{array}{ll} d_v \text{ unrestricted} & \forall v \in V \\ d_v - d_u \leq l_{uv} & \forall a = (u,v) \in A \end{array} \right\}$$

setting $d_s = 0$, we see that d_v is the distance from v to s.

Example 16: Maximum Flow Problem

Recalling our formulation of maximum flow,

$$\begin{array}{rl} \text{maximize} & \sum_{a \in \delta^-(t)} f_a \\ \text{subject to} & 0 \leq f_a \leq u_a \quad \forall a \in A \\ & \sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a \quad \forall v \in V - \{s,t\} \end{array}$$

Example 17: Zero-Sum Games

The **Minimax Theorem** is a result of the linear programming duality. It states that for any matrix A,

$$\max_{\mathbf{x}} \left[\min_{\mathbf{y}} \mathbf{x}^T A \mathbf{y} \right] = \min_{\mathbf{y}} \left[\max_{\mathbf{x}} \mathbf{x}^T A \mathbf{y} \right]$$

Computational Complexity

Polynomial-Time Reductions

A reduction $Q \rightsquigarrow R$ from a problem Q to another problem R represents Q as a case of R, which we already know how to solve. Examples of reductions that we have seen are,

Bipartite Matching $\rightsquigarrow$ Maximum Flow

Bipartite Vertex Cover $\rightsquigarrow$ Minimum Cut

Maximum Flow $\rightsquigarrow$ Linear Programming

Definition 74 (Karp Reductions)**.** *A **Karp reduction** $Q \rightsquigarrow R$ from Q to R is an algorithm that produces an instance `I'` of R given an instance `I` of R. The algorithm runs in polynomial time in the size of `I` and `I'` is a `YES` instance of R if and only if `I` is a `YES` instance of Q*[19]*.*

Remark 75. *Let $Q \rightsquigarrow R$.*

- *If R is easy to solve, then Q is easy to solve*

$$R_{\mathsf{easy}} \implies Q_{\mathsf{easy}}$$

- *If Q is hard to solve, then R is hard to solve*

$$Q_{\mathsf{hard}} \implies R_{\mathsf{hard}}$$

[19] We write $Q \leq_p R$, which can be read as "Q is polynomial-time reducible to R" or "R is at least as hard as Q."

- *If Q is easy to solve, then this tells us nothing about R*

$$Q_{\texttt{hard}} \implies \textit{Nothing}$$

Definition 76 (Vertex Cover Problem)**.** *We are given a graph $G = (V, E)$ and an integer c. We want to know if G contains a set of at most c vertices that are incident to each edge.*

Definition 77 (Independent Set Problem)**.** *We are given a graph $G = (V, E)$ and an integer k. We want to know if G contains a set of at least k vertices that are mutually non-adjacent.*

Example 18: Independent Set $\rightsquigarrow$ Vertex Cover

If $S \subseteq V$ is an independent set, then $V - S$ is a vertex cover. A polynomial time algorithm for **Vertex Cover** can be used to give a polynomial time algorithm for **Independent Set**.

There is an independent set of size at least k

$$\iff$$

There is a vertex cover of size at most $n - k$

Similarly, Vertex Cover $\rightsquigarrow$ Independent Set.

Definition 78 (Set Cover Problem)**.** *We are given sets $S_1, S_2, \cdots, S_n \subseteq W$ and an integer k. We want to know if there is a collection of at most k sets that cover every element of W.*

Example 19: Vertex Cover $\rightsquigarrow$ Set Cover

Suppose that we are given a graph $G = (V, E)$. Define,

$$S_v := \{e \mid v \text{ is an endpoint of } e\}$$

for each $v \in V$. Set $W = \{e \mid e \in E\}$. Then a set cover of size k corresponds to a vertex cover of size k, and a polynomial time algorithm for **Set Cover** can be used to give a polynomial time algorithm for **Vertex Cover**.

Definition 79 (Satisfiability Problem)**.** *Suppose that we have,*

- *Boolean variables $x_1, \cdots, x_n$*
- *Clauses $C_1, \cdots, C_m$ which are disjunctions of literals,*

$$C_j = x_2 \vee \bar{x}_5 \vee \bar{x}_6 \vee x_8 \vee \bar{x}_9$$

- *Literals x_i and $\bar{x}_i \in \{\texttt{True}, \texttt{False}\}$ for each variable x_i*

We want to know if there is an assignment that satisfies every clause.

Definition 80 (3-Satisfiability)**.** ***3-satisfiability*** *is a special case of the satisfiability problem, where every clause has exactly three literals.*

e.g., $(x_1 \vee \bar{x}_2 \vee x_5) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_3 \vee x_4) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee \bar{x}_5)$

Example 20: SAT $\rightsquigarrow$ 3-SAT

Clearly SAT $\rightsquigarrow$ 3-SAT. We can also show that SAT $\rightsquigarrow$ 3-SAT.

Remark 81. *A variable assignment satisfies $\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k$ if and only if the same variable assignment satisfies,*

$$(\ell_1 \vee \cdots \vee \ell_p \vee y) \wedge (\bar{y} \vee \ell_{p+1} \vee \cdots \vee \ell_k)$$

for some `True` / `False` *assignment of y.*

- We need to convert an instance `I` of SAT into an instance `I'` of 3-SAT. To do this, take each clause C in `I` and mimic it via a set of clauses of size 3 with additional variables
- Take a clause $C = \ell_1 \vee \ell_2 \vee \cdots \ell_k$. There are three cases,
  - If $k = 2$, we can pick y and $\bar{y}$ as follows:
    $\ell_1 \vee \ell_2 \implies (\ell_1 \vee \ell_2 \vee y) \wedge (\ell_1 \vee \ell_2 \vee \bar{y})$
  - If $k = 1$, we can pick y_1, y_2 and $\bar{y_1}, \bar{y_2}$ as follows:
    $\ell_1 \implies (\ell_1 \vee y_1 \vee y_2) \wedge (\ell_1 \vee y_1 \vee \bar{y}_2) \wedge (\ell_1 \vee \bar{y}_1 \vee y_2) \wedge (\ell_1 \vee \bar{y}_1 \vee \bar{y}_2)$
  - If $k \geq 3$, we apply Remark 81 recursively.

. Thus, for $C = \ell_1 \vee \ell_2 \vee \cdots \ell_k$, we have:

- $k - 2$ clauses in `I'`
- $k - 3$ new variables in `I'`

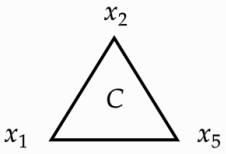
so a polynomial time algorithm for **3-SAT** can be used to give a polynomial time algorithm for **SAT**.

Example 21: 3-SAT $\rightsquigarrow$ Independent Set

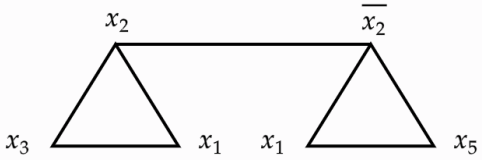
- We need to convert an instance `I` of 3-SAT,

  $$C = x_1 \vee \bar{x}_2 \vee x_5$$

  into an instance `G` of Independent Set. For each clause C in `I`, we have a triangle in `G`,



- We add an edge between each copy of x_i and $\bar{x}_i$,



There is a satisfying assignment for `I` if and only if there is an independent set in G whose size is equal to the number of clauses. To see this, note that,

- There are m triangles in G and an independent set S uses at most one vertex per triangle. This means that,

$$|S| \leq m$$

- If S selects x_i, then x_i is set to `True`. We cannot also select a vertex for x_i, so S induces a valid assignment
- If S selects $x_i \in C_j$, then C_j is satisfied
- Thus, an independent set of size m in G gives a satisfying assignment for `I` and any satisfying assignment for `I` induces an independent set of size m in G

NP Completeness

Definition 82 (P)**.** *The class **P** is the set of decision problems which can be solved in polynomial time by a deterministic computer.*

Definition 83 (NP)**.** *The class **NP** is the set of decision problems which can always be solved in polynomial time by a non-deterministic computer.*

Remark 84 ($P \neq NP$)**.** *The **conjecture** that $P \neq NP$ states that computation is harder than verification. We know that $P \subseteq NP$ since we can compute and verify a solution to a problem $R \in P$ in polynomial time.*

Definition 85 (NP Complete)**.** *A problem R is **NP Complete** if for every $Q \in NP$, there is a polynomial time reduction[20] from Q to R.*

Equivalently, NP is,

- The set of decision problems whose `YES` instances can be verified in polynomial time
- The set of decision problems that can be solved in exponential time by Brute-Force Search

These definitions are equivalent because any `YES` instance has a certificate of size $poly(n)$, and there are an exponential, i.e., $2^{poly(n)}$ number of such certificates.

[20] The reduction maps `YES` instances of Q into `YES` instances of R, and it maps `NO` instances of Q into `NO` instances of R

Definition 86 (coNP)**.** *The class **coNP** is the set of decision problems whose* `NO` *instances can be verified in polynomial time.*

Definition 87 (coNP Complete)**.** *A problem R is **coNP Complete** if for every $Q \in coNP$, there is a polynomial time reduction from Q to R.*

Definition 88 (Good Characterization)**.** *A problem $R \in NP \cap coNP$ has a **good characterization** if has a polynomial* `YES` *and* `NO` *certificate.*

Example 22: Good Characterizations

The following problems have a good characterization,

Problem	`YES` Instance	`NO` Instance
Maximum Flow	Flow	Minimum Cut
Bipartite Matching	Matching	Vertex Cover
Linear Programming	Feasible Primal	Feasible Dual

Remark 89. *The complement of an NP Complete problem is in coNP.*

Remark 90. $P = NP \implies NP = coNP$

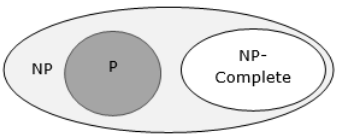
Proof. Problems in P have a `YES` and `NO` certificate. If $P = NP$ then every problem in NP has a short `YES` and `NO`. This implies,

$$P \subseteq coNP$$

Any problem $R \in coNP$ has $R^c \in NP = P \implies R \in P = NP$. □

Example 23: Cliques

The problem of determining if a set S of vertices is a clique is in NP. We can check if the vertices are pairwise adjacent, which is polynomial in the size of S.



3 − *SAT*, Independent Set, Vertex Cover, and Set Cover are NP Complete.

Definition 91 (Cook's Theorem)**.** *Satisfiability is NP Complete.*

Remark 92 (Proving NP Completeness)**.** *Do the following,*

1. *For your new problem R, show that $R \in NP$*
2. *Find an NP-complete problem Q such that $Q \rightsquigarrow R$*

Definition 93 (Graph-Coloring Problem)**.** *A **vertex coloring** of a graph $G = (V, E)$ is an assignment of colors to vertices such that adjacent vertices recieve different colors. The chromatic number $\chi(G)$ is the minimum number of colors required for a valid coloring to exist. We want to know if there exists a 3-coloring of G.*

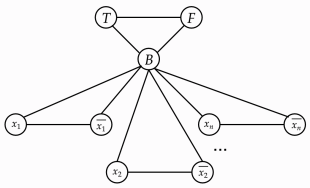
Example 24: 3-SAT $\rightsquigarrow$ 3-Coloring

First, we can show that 3-coloring is in NP by checking,

- ≤ 3 colors are used
- Every vertex recieves a color c
- $c(u) \neq c(v)$ for all $(u, v) \in E$

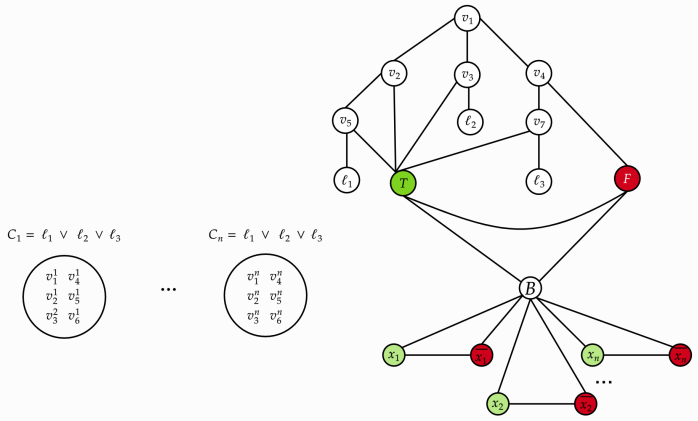
Second, we reduce from $3 - SAT$ to 3-Coloring:

- We need to convert an instance `I` of 3-SAT into an instance `I'` of 3-Coloring. To do this, we build a graph G as follows,
  - There is a vertex for each literal $x_1, \bar{x_1}, x_2, \cdots, x_n, \bar{x_n}$
  - There are three vertices, T, F, B
  - The vertices are connected as follows,



- Define the following,
  - `Green` = `True`
  - `Red` = `False`
  - `White` = `BLANK`

  so that the colors of $x_1, \bar{x_1}, x_2, \cdots, x_n, \bar{x_n}$ give a valid `True` and `False` assignment of variables.
- Design a gadget for each clause which can be 3-colored if and only if the clause is satisfied. Each gadget clause contains 6 new vertices which we connect to the assignment



- The clause for C is $\ell_1 \vee \ell_2 \vee \ell_3$. We can check coloring for,

$$\text{Case 1. } (\ell_1 = F, \ell_2 = F, \ell_3 = T)$$
$$\text{Case 2. } (\ell_1 = F, \ell_2 = T, \ell_3 = F)$$
$$\dots$$
$$\text{Case 8. } (\ell_1 = F, \ell_2 = F, \ell_3 = F)$$

- We discover that Case 8 cannot be colored validly, so G can be 3-coloured if and only if `I` is satisfiable. This completes the polynomial reduction 3-SAT $\rightsquigarrow$ 3-Colouring

Definition 94 (Prime Factorization Problem)**.** *Given integers N and k, we want to determine if N has a prime factor $\leq k$.*

The YES instance for prime factorization works because even if p is not prime, it has some prime factorization.

Example 25: Prime Factorization

We believe that prime factorization is in $NP \cap coNP - P$.

- Prime factorization is in NP
  - (YES Instance) Give an integer $p \leq k$ such that $p|N$.
- Prime factorization is in coNP
  - (NO Instance) Given the unique prime factorization $N = p_1 p_2 \cdots p_n$, we verify that each $p_i > k$ and confirm that p_i is prime in polynomial time

Well-Known NP Complete Problems:

- (Hamiltonial Cycle Problem) Given $G = (V, E)$, we want to determine if G contains a cycle that uses every vertex exactly once
- (Hamiltonian Path Problem) Given $G = (V, E)$, we want to determine if G contains a path that uses every vertex exactly once
- (Partition Problem) Given integers $x_1, \cdots, x_n \geq 0$, we want to determine if $\exists$ a subset $S \subseteq [n]$ s.t.,
$$\sum_{i \in S} x_i = \sum_{i \notin S} x_i$$
- (Maximum Cut Problem) Given $G = (V, E)$ and an integer k, we want to determine if G contains a cut $\delta(S)$ containing at least k edges

Example 26: Taxonomy of Easy and Hard Problems

Easy Problems	Hard Problems
Minimum Cut	Maximum Cut
Euler Circuit	Hamiltonian Cycle
2-SAT	3-SAT
2-Coloring	3-Coloring
Shortest Path	Longest Path
Shortest Even $(s-t)$ Path	Shortest Even $(s-t)$ Dipath
Linear Programming	Integer Programming

Example 27: Taxonomy of Hard Problems

Group	Example
Packing Problems	• Independent Set • Clique • Set Packing
Covering Problems	• Vertex Cover • Set Cover
Partitioning Problems	• 3-Coloring • 3D-Matching • Maximum Cut
Sequencing Problems	• Hamiltonian Path • Travelling Salesman
Numerical Problems	• Partition • Knapsack

Example 28: Integer Programming

We are given an integer problem and an integer k. We want to determine if there is a feasible solution with value at least k. The minimization case is analogous.

PSpace and Complexity Classes

Definition 95 (Exp)**.** *The class **Exp** is the set of decision problems that can be solved in exponential time by a deterministic computer.*

Definition 96 (NExp)**.** *The class **NExp** is the set of decision problems that can be solved in exponential time by a non-deterministic computer*[21]*.*

Definition 97 (PSpace)**.** *The class **PSpace** is the set of decision problems that can be solved by an algorithm using polynomial space*[22]*.*

[21] Equivalently, this is the set of decision problems whose YES instances can be verified in exponential time.

[22] $P \subseteq PSpace$ since a problem that requires polynomial time uses a polynomial amount of space.

Definition 98 (PSpace Complete)**.** *A problem R is **PSpace Complete** if for every $Q \in PSpace$, there is a polynomial space reduction from Q to R.*

Definition 99 (ExpSpace)**.** *The class **ExpSpace** is the set of decision problems that can be solved by an algorithm using exponential space.*

Theorem 100. *$NP \subseteq PSpace$.*

Proof. It suffices to show that 3-SAT $\in$ PSpace since there exists a polynomial reduction for any problem $Q \in NP$ such that $Q \rightsquigarrow$ 3-SAT. This reduction necessarily uses a polynomial amount of space. Take an instance `I` of 3-SAT. Let each binary string $b_1 b_2 \cdots b_n$ of length n encode a `True/False` assignment of the variables.

$$b_i = \begin{cases} 1 & x_i = \texttt{True} \\ 0 & \bar{x_i} = \texttt{True} \end{cases}$$

Algorithm 4: 3SAT $\in$ PSpace

```
// Initialize b to be the 0 vector.
function 3SAT(b, I)
    foreach b ∈ C(I) do
        if b does not satisfy I then
            // Increase b_n by 1.
            b ← b + 1
        else
            return b
```

Algorithm 6 either finds a satisfying assignment, or it confirms that `I` is not satisfiable. It runs in exponential time by counting to 2^n, but it only uses a polynomial amount of space. □

Remark 101. *If $Q \in PSpace$, then $Q^c \in PSpace$. Thus, $coNP \subseteq PSpace$.*

Remark 102. *It has been conjectured that,*

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP} \subseteq \mathbf{NEXP} \subseteq \mathbf{EXPSPACE}$$

Definition 103 (Q-Satisfiability)**.** ***Q-satisfiability** is a special case of the satisfiability problem, where every clause alternates between $\forall$ and $\exists$.*

$$\textit{e.g.,}\ \exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \quad C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

The mix of universal and existential quantifiers arises commonly in games,

$\exists$ a move such that,

$\forall$ moves by the opponent,

etc.

Example 29: QSat $\in$ PSpace

Let $\phi(x_1, x_2, \cdots, x_n) = C_1 \wedge C_2 \wedge \cdots C_m$ and consider,

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \cdots \Phi(x_1, x_2, \ldots, x_n)$$

We want to solve the assignment recursively. At step i,

$$x_1 = \ell_1, \quad x_2 = \ell_2 \quad \cdots \quad x_{i-1} = \ell_{i-1}$$

Case 1. i is odd. x_i is associated with $\exists$. We want,

$$\Phi(\ell_1, \ldots, \ell_{i-1}, x_i, x_{i+1}, \ldots, x_n) = 1$$

for $\ell_1, \ldots, \ell_{i-1}$ fixed. This is true if and only if,

$$\Phi(\ell_1, \ldots, \ell_{i-1}, 0, x_{i+1}, \ldots, x_n) = 1$$

or

$$\Phi(\ell_1, \ldots, \ell_{i-1}, 1, x_{i+1}, \ldots, x_n) = 1$$

Case 2. i is even. x_i is associated with $\forall$. We want,

$$\Phi(\ell_1, \ldots, \ell_{i-1}, x_i, x_{i+1}, \ldots, x_n) = 1$$

for $\ell_1, \ldots, \ell_{i-1}$ fixed. This is true if and only if,

$$\Phi(\ell_1, \ldots, \ell_{i-1}, 0, x_{i+1}, \ldots, x_n) = 1$$

and

$$\Phi(\ell_1, \ldots, \ell_{i-1}, 1, x_{i+1}, \ldots, x_n) = 1$$

Time Complexity:
To solve $\phi(x_1, x_2, \cdots, x_n)$, we solve 2 subproblems,

$$T(n) \leq 2 \cdot \underbrace{T(n-1)}_{x_n \text{ is fixed}} + \text{poly}(n, m)$$

for n variables and m clauses.

Space Complexity:
We can re-use space for each subproblem.

- Solve the case $x_1 = 0$
- Save the solution $\phi(0, x_2, \cdots, x_n) = 0$ or $\phi(0, x_2, \cdots, x_n) = 1$
- Delete all other memory and reuse the space to solve $x_1 = 1$

This gives the space complexity,

$$S(n) \leq S(n-1) + \text{poly}(n, m)$$

Definition 104 (2Exp)**.** *The class **2Exp** is the set of decision problems that can be solved in double exponential time, $2^{2^{poly(n)}}$ by a deterministic computer. Similar definitions exist for **2NExp** and **2ExpSpace**.*

Theorem 105 (Time and Space Hierarchy)**.** *We know that,*

- *Time Hierarchy Theorem I*

$$\mathbf{P} \subset \mathbf{EXP} \subset \mathbf{2EXP} \subset \mathbf{3EXP} \subset \cdots$$

- *Time Hierarchy Theorem II*

$$\mathbf{NP} \subset \mathbf{NEXP} \subset \mathbf{2NEXP} \subset \mathbf{3NEXP} \subset \cdots$$

- *Space Hierarchy Theorem*

$$\mathbf{PSPACE} \subset \mathbf{EXPSPACE} \subset \mathbf{2EXPSPACE} \subset \cdots$$

Search and Decision Problems

Definition 106 (NP Hard)**.** *A problem R is **NP Hard**[23] if for every $Q \in NP$, there is a polynomial time reduction from Q to R.*

[23] An NP Hard problem may not be in NP. For example *QSAT* is NP Hard. In fact, it may not even be a decision problem, e.g., "Satisfy as many clauses as possible" is an optimization problem.

The optimization version of an NP Complete problem is NP Hard.

Definition 107 (Max-SAT)**.** *Given a set of clauses,*

$$C_1, \cdots, C_m$$

we need to find a `True/False` *assignment of the variables $x_1, \cdots, x_n$ that maximizes the number of satisfied clauses.*

Definition 108 (FNP)**.** *Unlike a decision problem, a **search problem** requires a solution. The search analogue of NP is **FNP** (Functional Nondeterministic Polynomial Time). Given x and a polynomial predicate $f(x, y)$, output y such that $f(x, y)$ is* `True` *if y exists.*

Definition 109 (TFNP)**.** *The class **TFNP** is the subset of problems (called "total") in FNP for which a solution is known to exist.*

Definition 110 (PLS)**.** *The class **PLS** is the set of total search problems that can be solved in exponential time by best-response dynamics.*

Definition 111 (PPAD)**.** *The class **PPAD** is the set of total search problems that can be solved in exponential time by path traversal.*

Note: No PPAD problem is FNP Complete unless NP = coNP.

Definition 112 (GD)**.** *The class **GD** is the set of total search problems that can be solved approximately in exponential time by gradient descent.*

Note: $GD = PLS \cap PPAD$.

Heuristic Algorithms

Backtracking and Branch-and-Bound

Backtracking algorithms search the exponential state space of solutions using a depth-first search tree. Within this tree, interior nodes correspond to partial solutions, and leaf nodes correspond to complete solutions. Every node of the tree is labelled,

1. Success, if the partial solution can be extended to a YES solution
2. Failure, if the partial solution cannot be extended to a YES solution
3. Active, if the partial solution is indeterminate

In the case of a success, the algorithm outputs the solution. In the case of a failure, the algorithm backtracks. In the case of an active label, the algorithm continues searching and pruning the tree.

2-SAT is solvable in polynomial time by the backtracking heuristic, but backtracking can take exponential time to solve an instance of SAT.

Definition 113 (Backtracking). *The **backtracking** procedure is,*

Algorithm 5: Backtracking Procedure

```
  // Start with some problem P_0
1 function Backtrack(P_0)
    // Initialize the set of active problems
2   S ← {P_0}
3   while S ≠ ∅ do
      // Choose a subproblem P ∈ S, expand it into
         smaller subproblems, and remove P from S
4     P ← P = {P_1, P_2, ··· P_k} ∈ S
5     foreach P_i ∈ P do
6       if (P_i) = Success then
7         return P_i
8       if (P_i) = Failure then
          // Discard P_i
9         P ← P − P_i
10      else
          // P_i is indeterminate
11        S ← S + P_i
```

The basic approach of backtracking can be extended to optimization problems via the branch-and-bound method. A node is either,

1. Infeasible, if there are no feasible completions of a partial solution

2. Sub-Optimal, if there are feasible completions of the partial solution but they are worse than the optimal solution[24]
3. Feasible, if there are feasible completions to the partial solution and we know the value of the best completion
4. Active, if the partial solution is indeterminate[25]

Definition 114 (Branch-and-Bound). *Suppose that we have a minimization problem. Each subproblem will be eliminated if the lower bound on its cost exceeds that of some other solution that we have already encountered. The **branch-and-bound** procedure is,*

Algorithm 6: Branch-and-Bound Procedure

```
// Start with some problem P_0
function BranchBound(P_0)
    // Initialize the set of active problems
    S ← {P_0}
    bestSoFar ← ∞
    while S ≠ ∅ do
        // Choose a subproblem P ∈ S, expand it into
           smaller subproblems, and remove P from S
        P ← P = {P_1, P_2, ··· P_k} ∈ S
        foreach P_i ∈ P do
            if (P_i) = Success then
                // Update best_so_far
                bestSoFar ← C(P_i)
            if L(P_i) < bestSoFar then
                // Lower bound of P_i is less than bestSoFar
                S ← S + P_i
```

We will see an example of the Branch-and-Bound Procedure applied to the Knapsack Problem. First, we will use three facts that were proven in the context of linear programming. These are,

Remark 115. *The inear programming relaxation of an integer programming problem removes the integrality constraint of each variable,*

$$\begin{aligned} &\max \textstyle\sum_{i=1}^{n} v_i \cdot x_i \\ &s.t.\ \textstyle\sum_{i=1}^{n} w_i \cdot x_i \leq W \\ &\underbrace{x_i \in \{0,1\}}_{x_i \in \mathbb{Z}^+} \quad \forall i \in [n] \end{aligned} \qquad \begin{aligned} &\max \textstyle\sum_{i=1}^{n} v_i \cdot x_i \\ &s.t.\ \textstyle\sum_{i=1}^{n} w_i \cdot x_i \leq W \\ &\underbrace{0 \leq x_i \leq 1}_{x_i \in \mathbb{R}^+} \quad \forall i \in [n] \end{aligned}$$

Corollary 116. *The feasible region of the relaxed linear program is larger than the feasible region of the original integer linear program[26].*

[24] To show that a subtree is sup-optimal, we need a feasible solution to compare it against. Typically, Branch-and-Bound will use Depth-First Search to find an initial feasible solution.

[25] **Example:** Typically, we branch on the variable that is the most fractional in the linear programming relaxation.

[26] The feasible region of the linear program allows for fractional solutions.

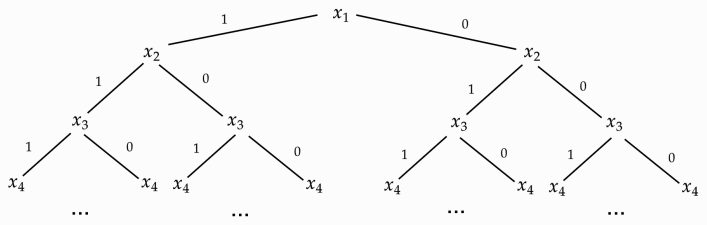
Corollary 117. *If the linear program is infeasible, then the integer program is infeasible. Conversely, if the linear program is feasible, then its value is at least as large as the value of the integer program by the previous corollary.*

Example 30: Branch-and-Bound with Knapsack

Suppose that a thief has a bag with capacity W. There are n items to steal, and each item has an associated value v_i and weight w_i. The thief wants to steal the subset of items of **maximum total value** that fit into the knapsack,

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{n} v_i \cdot x_i \\ \text{subject to} & \sum_{i=1}^{n} w_i \cdot x_i \leq W \\ & x_i \in \{0,1\} \quad \forall i \in [n] \end{array}$$

We can solve this via exhaustive search using a search tree T,



but each subtree may be of exponential size to solve. Thus, we require a method for determining if a subtree is either infeasible or worse than the optimal solution. Consider both possibilities for an assignment of x_1. Each decision node in the binary search tree T corresponds to a partial solution,

$$\{x_1, x_2, x_3, x_4\} = \{0, *, *, *\} \quad \{x_1, x_2, x_3, x_4\} = \{1, *, *, *\}$$

In particular, at the root of the search tree no variables have been assigned so the corresponding partial solution is empty,

$$\{x_1, x_2, x_3, x_4\} = \{*, *, *, *\}$$

This is useful for two reasons,

1. If the partial solution is infeasible, then every completion of the partial solution to a full solution will be infeasible.
2. If the partial solution leads to low quality solutions, then every completion of the partial solution to a full solution will have a sub-optimal value.

We can use **integer program relaxation** to,

1. Test if every completion of the partial solution is infeasible
2. Obtain an upper bound on the value of any completion of the partial solution, which bounds the complete solution

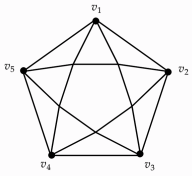
at the root of every subtree in T.

Solving the Knapsack Problem:
The Knapsack Problem can be solved quickly using a greedy algorithm:

1. Compute the value per weight $V_i := v_i / w_i$ for each item
2. Determine the object with the maximum ratio, $x^* = \arg\max_i V_i$
3. Assign the knapsack as much of the item x^* as the weight W allows
4. If the knapsack is not full, recurse on the remaining objects

Example 31: Bad Estimators with Maximum Independent Set

Linear programs may lead to bad estimators for integer solutions. Consider the problem of finding the maximum size of an independent set on the graph K_5,



$$\max \sum x_i \qquad \max \sum x_i$$
$$x_i + x_j \leq 1 \quad (i,j) \in E(K_5) \qquad x_i + x_j \leq 1 \quad (i,j) \in E(K_5)$$
$$x_i \in \{0,1\} \quad v_i \in V(K_5) \qquad 0 \leq x_i \leq 1 \quad v_i \in V(K_5)$$

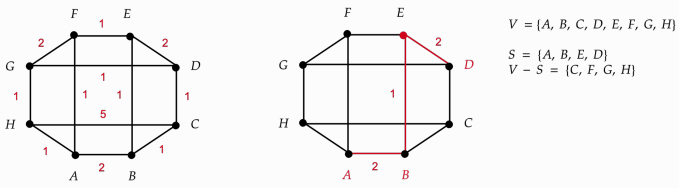
The optimal value of the linear program is $\frac{|V(G)|}{2}$, while the optimal value of the integer program is 1. This becomes increasingly inaccurate as $|V(G)|$ grows.

One possible solution is to use combinatorial estimators[27].

[27] See Dasgupta, Chapter 9 (p. 273)

Example 32: Combinatorial Estimators with Salesman

Suppose that we are given a graph $G = (V, E)$ with non-negative edge costs c_e. The goal is to design a tour T that starts and ends at A, includes all other vertices exactly once, and has minimum total cost $c(T) = \sum_e c_e$. A partial solution Q is a simple path P with endpoints a and b passing through $S \subseteq V \cup \{a, b\}$. The corresponding subproblem is to find the best completion of the tour, that is, the cheapest complementary path from b to a with intermediate nodes $V - S$.



The cost of a completion is at least the sum of,

1. The lightest edge from a to $V - S$
2. The lightest edge from b to $V - S$
3. The minimum spanning tree of $V - S$

Local Search

Definition 118 (Local Search)**.** ***Local search*** *attempts to find a solution S^* to a problem by making local improvements to the current solution S.*

Algorithm 7: Local Search (Minimization)

```
// Start with some initial solution s
function LocalSearch(s)
    // Search the neighborhood Γ of s for an alternate
       solution s′ with a lower cost
    while ∃s′ ∈ Γ(s) do
        if cost(s′) < cost(s) then
            s ← s′
```

Theorem 119. *Local search produces a locally optimal solution.*

Proof. We cannot return to a solution that was found previously. □

The neighborhood structure is imposed upon the problem, and it is a central design decision in local search. For example, an algorithm based on Local Search will run quickly with a small neighborhood. Conversely, bigger neighborhoods enable us to find a better solution.

Example 33: Local Search with Maximum Cut

In the **Maximum Cut Problem**, we are given an undirected graph $G = (V, E)$ and a weight $w(e)$ on each edge. We want to separate $V(G)$ into two sets S and $V - S$ so that the total weight of the edges between the two sets is as large as possible. A local improvement is a move of one vertex that

produces an increase in the weight of the cut.

Let $\mathcal{L} = \varnothing$ and $\mathcal{R} = \mathrm{V}$

Repeat:

If $\exists v \in V$ such that $\mathrm{cap}(\mathcal{L} \cup v) > \mathrm{cap}(\mathcal{L})$:

Set $\mathcal{L} \longleftarrow \mathcal{L} \cup v$

Else, If $\exists v \in V$ such that $\mathrm{cap}(\mathcal{L} \backslash v) > \mathrm{cap}(\mathcal{L})$:

Set $\mathcal{L} \leftarrow \mathcal{L} \backslash v$

Theorem 120. *Any local maximum cut $\delta(\mathcal{L})$ has capacity at least half the capacity of the global maximum cut $\delta(\mathcal{L}^*)$.*

Proof. Let $\delta(\mathcal{L})$ be a local maximum cut, and let $\delta(\mathcal{L}^*)$ be the global maximum cut. For any vertex v, let E_v be the set of vertices incident to v. There are no local improvements for $\mathcal{L}$,

$$\sum_{e \in E_v \cap \delta(\mathcal{L})} u_e \geq \sum_{e \in E_v \cap \delta(\mathcal{L})^c} u_e$$

In particular, what this means is,

$$\sum_{e \in E_v \cap \delta(\mathcal{L})} u_e \geq \frac{1}{2} \cdot \sum_{e \in E_v} u_e$$

Therefore,

$$\begin{aligned} \mathrm{cap}(\mathcal{L}) &= \frac{1}{2} \cdot \sum_{v \in V} \sum_{e \in E_v \cap \delta(\mathcal{L})} u_e \\ &\geq \frac{1}{4} \cdot \sum_{v \in V} \sum_{e \in E_v} u_e \text{ plugging in the inequality} \\ &\geq \frac{1}{2} \cdot \mathrm{cap}(\mathcal{L}) \end{aligned}$$

□

Example 34: Minimum Spanning Tree Problem

The Local Search Algorithm can be used to construct an optimal solution to the **Minimum Spanning Tree Problem**. We say that a spanning tree $\hat{T}$ is in the neighborhood of a spanning tree T if they differ in exactly one edge,

$$\Gamma(\mathcal{T}) = \{\hat{\mathcal{T}} : \mathcal{T} = (\hat{\mathcal{T}} \cup e) - \hat{e} \text{ where } e \in \mathcal{T} \text{ and } \hat{e} \in \hat{\mathcal{T}}\}$$

We will search for an improving swap that reduces the total

Cut Property of Minimum Spanning Trees: Assume that edge costs are distinct. If e is the cheapest edge in some cut $\delta(S)$, then e must be in the minimum spanning tree.

Proof. Let $\mathcal{T}^*$ be a minimum spanning tree. Assume that there exists a cut $\delta(S)$ whose cheapest edge $e = (u, v)$ is not in $\mathcal{T}^*$. Since $\mathcal{T}^*$ is a spanning tree, there exists a unique path $P \subseteq \mathcal{T}^*$ from u to v. If $\hat{e} \in P$ then $(\mathcal{T}^* \backslash \hat{e}) \cup e$ is a spanning tree. Moreover, $\hat{e} \in P \cap \delta(S)$ since $|P \cap \delta(S)|$ is odd and consequently ≥ 1. Swapping e for $\hat{e}$ results in a cheapter spanning tree than $\mathcal{T}^*$. □

cost of the spanning tree.

Let $\mathcal{T}$ be a spanning tree.
While $\exists \hat{\mathcal{T}} \in \Gamma(\mathcal{T})$ s.t. $c(\hat{\mathcal{T}}) < c(\mathcal{T})$:
  Set $\hat{\mathcal{T}} \leftarrow \mathcal{T}$

This outputs a locally minimum spanning tree $\mathcal{T}$, but we can show that it is in fact the globally minimum spanning tree $\mathcal{T}^*$. Assume not. If $\mathcal{T} \neq \mathcal{T}^*$, then $\exists e \in \mathcal{T}^* - \mathcal{T}$. By the Cut Property, e is the cheapest edge in some cut $\delta(S)$. Since $\mathcal{T}$ is a spanning tree, there is a unique path P in $\mathcal{T}$ from u to v. Thus, there is an edge $\hat{e} \in P \cap \delta(S)$ with $\hat{e} \neq e$. But then $\hat{\mathcal{T}} = (\mathcal{T} \cup e) - \hat{e}$ is a cheaper spanning tree than $\mathcal{T}$.

Example 35: Local Search with Salesman

A tour $\hat{\mathcal{T}}$ is said to be in the neighborhood of $\mathcal{T}$ if they differ in exactly two edges. The Local Search Algorithm looks for an improving 2-swap that reduces the cost of the tour. However, there might be an exponential number of iterations needed to solve the Travelling Salesman Problem in this way. Moreover, the final tour is only guaranteed to be locally optimal.

Approximation Algorithms

Bounding the Optimum

Definition 121 (Approximation Algorithm)**.** *An algorithm* `A` *is an* α***-approximation*** *(*$\alpha \geq 1$*) for a problem* Q *if for every instance* I*,*

1. `A` *outputs a feasible solution* `S`
2. `A` *runs in polynomial time*
3. *If* Q *is a minimization problem, then* $cost(\texttt{S}) \leq \alpha \cdot cost(\texttt{OPT})$
4. *If* Q *is a maximization problem, then* $val(\texttt{S}) \geq 1/\alpha \cdot val(\texttt{OPT})$

Application 1: Travelling Salesman Problem

Suppose that we are given a complete, undirected graph K_n with non-negative integer costs c for each edge. The goal is to find the cheapest Hamiltonian cycle of G.

Definition 122 (Triangle Inequality). *The edge-cost function* $c : V(K_n) \times V(K_n) \to \mathbb{R}^+$ *satisfies the* ***triangle inequality*** *if the following holds,*

$$c\underbrace{(v_1, v_2)}_{e_1} \leq c\underbrace{(v_1, v_3)}_{e_2} - c\underbrace{(v_3, v_2)}_{e_3} \quad \forall e_1, e_2, e_3 \in E(K_n)$$

Definition 123 (Walk). *A* ***walk*** *is a sequence of vertices,*

$$v_1, \cdots, v_n \text{ such that } (v_i, v_j) \in E(G) \text{ for all } i, j \in [n]$$

Definition 124 (Eulerian Graph). *Let* $G = (V, E)$ *be a multigraph.* G *is* ***Eulerian*** *if it has a closed walk that uses every edge exactly once.*

Theorem 125 (2-Approximation TSP). *The* `TreeDoubling` *algorithm is a* ***2-approximation algorithm*** *for the Metric TSP.*

Algorithm 8: 2-Approximation Travelling Salesman

```
// Prim(G,c) uses Prim's Algorithm to find a minimum
   spanning tree in G, given the weight function c
1 function TreeDoubling(K_n, c)
    // Find a minimum spanning tree T of K_n
2   T ← Prim(K_n, c)
    // Duplicate each edge in T to obtain a Eulerian
       multigraph T' (all vertex degrees in T' are even)
3   T' ← (V(K_n), 2 · E(T))
    // Compute a Eulerian tour H of T'.  Whenever a
       vertex v is visited in H that was already
       visited, skip v and proceed with the next
       unvisited node along the cycle
4   H ← PreOrder(T*)
    // Return resulting Hamiltonian tour H
5   return H
```

Proof. `TreeDoubling` is a 2-approximation algorithm if,

1. It outputs a feasible tour H
2. It runs in polynomial time
3. Its cost is at most $2 \cdot c(\texttt{OPT})$

Polynomial running time is guaranteed since `Prim` runs in $O(|V(K_n)|^2)$ and `PreOrder` is a form of Depth First Search, which is in $O(|V(K_n)| + |E(K_n)|)$. Moreover, the `TreeDoubling` algorithm clearly outputs a feasible solution since H is a Eulerian tour by construction.

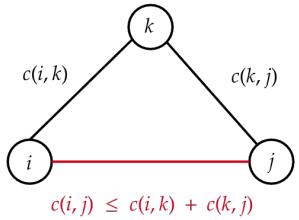


Figure 1: Illustration of metric costs.

1. A **path** is a walk with no vertex repeated, while a **trail** is a walk with no edge repeated.
2. An **Euler trail** is a trail that uses every edge, and an **Euler tour** is a closed Euler trail.

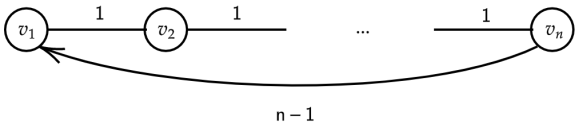
It remains to prove that $c(H) \leq 2 \cdot \texttt{OPT}$, where OPT is the optimal tour in K_n. Let T be a minimum spanning tree of K_n with respect to c. We know that $c(T) \leq c(\texttt{OPT})$ since deleting any edge of a Hamiltonian tour gives a spanning tree. Therefore,

$$\begin{aligned} c(H) &\leq c(2 \cdot T) \text{ by short-cutting} \\ &= 2 \cdot c(T) \\ &\leq 2 \cdot (\texttt{OPT}) \end{aligned}$$

Thus, we have an approximation algorithm with $\alpha = 2$. □

The natural question that arises is whether an approximation guarantee of $\alpha = 2$ is sufficient. There are four questions to ask,

1. Is our analysis tight with respect to the lower bound T? Yes.

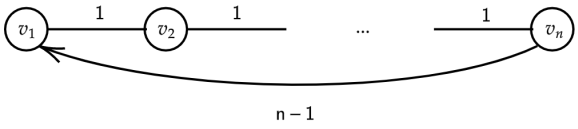


$c(T) = n-1$

$c(H) = 2(n-1)$

$c(\texttt{OPT}) = 2(n-1)$

2. Is the lower bound closer to OPT than a factor of 2? No.

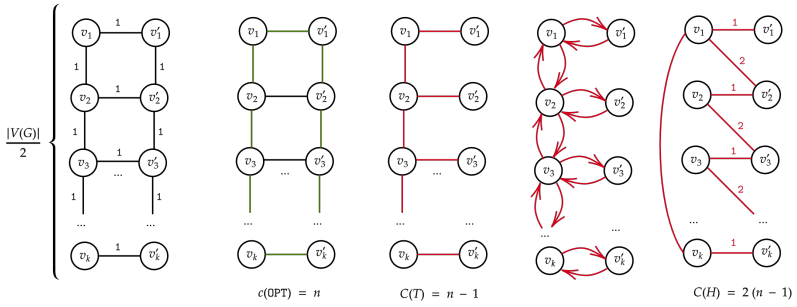


$c(T) = n-1$

$c(H) = 2(n-1)$

$c(\texttt{OPT}) = 2(n-1)$

3. Is the analysis tight with respect to OPT? Yes.



4. Is there a better approximation algorithm? Yes. there is a $\frac{3}{2}$-approximation algorithm for the metric TSP.

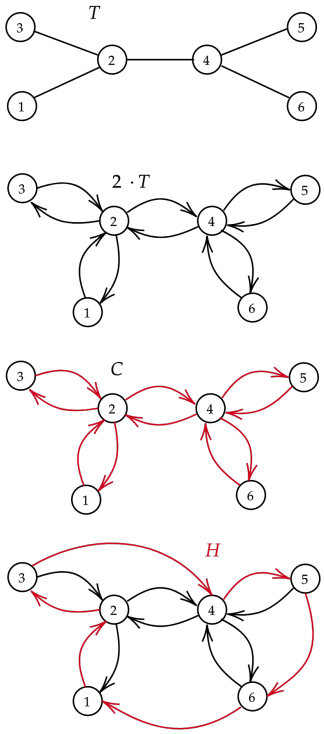


Figure 2: Illustration of the TreeDoubling algorithm.

Suppose that H' is a lower bound for OPT. We can ask **four questions**,

1. Is the analysis tight with respect to the lower bound for OPT? Find an example where $c(H) = \alpha \cdot H'$
2. Is the lower bound for OPT closer to OPT than a factor of 2? Find an example where $\texttt{OPT} = \alpha \cdot H'$
3. Is the analysis tight with respect to OPT? Find an example where $c(H) = \alpha \cdot \texttt{OPT}$
4. Is there a better approximation algorithm for our problem?

See the following paper for more information on the Metric TSP.

Definition 126 (Matching on a Set). *A matching M of G is called a **matching on** $U \subseteq V(G)$ if all edges of M consist of two vertices from U*[28].

Theorem 127 (3/2-Approximation TSP). *The `Christofides` algorithm is a **3/2-approximation algorithm** for the Metric TSP*[29].

Algorithm 9: 3/2-Approximation Travelling Salesman

```
// Prim(G,c) uses Prim's Algorithm to find a minimum
   spanning tree in G, given the weight function c
function Christofides(K_n, c)
    // Find a minimum spanning tree T of K_n
    T ← Prim(K_n, c)
    // Let U ⊆ V(T) be the odd degree vertices in T
    U ← {v | v ∈ V(T) and deg(v) = 2k+1}
    // Compute a minimum weight perfect matching M on
       the subgraph induced by U
    M ← Ford-Fulkerson(K_n, U, c)
    // Compute a Eulerian tour H of T ∪ M, taking
       shortcuts to obtain a Hamiltonian tour
    H ← PreOrder(T*)
    return H
```

Proof. `Christofides` is a 3/2-approximation algorithm if,

1. It outputs a feasible tour H
2. It runs in polynomial time
3. Its cost is at most $3/2 \cdot c(\texttt{OPT})$

First observe that the number of odd degree vertices of the spanning tree T is even, since the sum of the degrees of all vertices is $2(n-1)$ by the Handshaking Lemma. Thus, a perfect matching on U exists. Moreover, it can be found using maximum flows in $O(n^3)$. Hence, the algorithm is polynomial. Moreover, the `Christofides` algorithm outputs a feasible solution since H is a Eulerian tour by construction.

The weight of the Eulerian tour H is at most $c(T) + c(M)$, and it was proven earlier that $c(T) \leq \texttt{OPT}$[30]. It suffices to show that $c(M) \leq \frac{1}{2} \cdot \texttt{OPT}$, where `OPT` is the optimal tour in K_n.

Since `OPT` is a Hamiltonian Cycle, we can shortcut to obtain a cycle $\hat{C}$ on the set of odd degree vertices. Clearly $|\hat{C}| = |U|$, so $\hat{C}$ can be partitioned into two matchings M_1, M_2. In particular,

$$\begin{aligned} \frac{1}{2}(\texttt{OPT}) &= \frac{1}{2}(c(M_1) + c(M_2)) \\ &\geq c(M) \end{aligned}$$

[28] Recall that the matching is called "perfect" if every vertex of U is incident with an edge of M.

[29] Instead of doubling the number of edges, we can add edges between odd degree vertices. The graph will still have even degree vertices and therefore an Euler tour.

[30] $c(H) < c(T) + c(M)$ by shortcutting and applying the triangle inequality.

since M was computed to be the minimum weight perfect matching. □

Theorem 128. *There is no polynomial-time α-approximation algorithm for the traveling salesman problem on general weighted graphs, unless $P = NP$.*

Proof. Construct a cost function for K_n as follows,

$$c(e) := \begin{cases} 1 & e \in E(G) \\ \alpha \cdot n & \text{otherwise} \end{cases}$$

Suppose not. If G has a Hamiltonian cycle, then its cost in K_n is n. Any other Hamiltonian cycle has cost $\geq \alpha \cdot n + (n-1) > \alpha n$. Thus, if G has a Hamiltonian cycle, then the α-approximation algorithm for the traveling salesman problem must find a tour in K_n of cost $\leq \alpha \cdot n$. The only such tours have cost n, and they are Hamiltonian cycles in G. This means that the α-approximation algorithm can distinguish between `YES` and `NO` instances of the Hamiltonian Cycle Problem, which is known to be an NP-complete decision problem. □

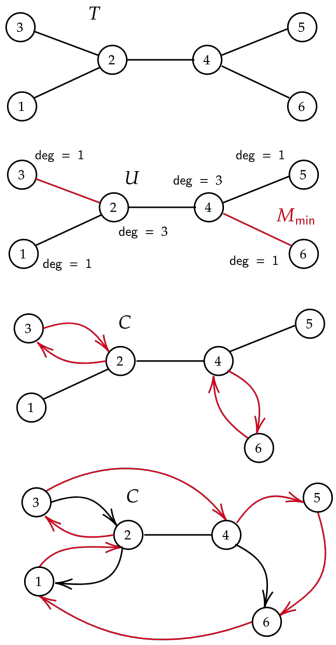


Figure 3: Illustration of the `Christofides` algorithm.

Application 2: Multiway Cut

Suppose that we are given an undirected graph $G = (V, E, c)$ with non-negative edge weights c, and a set of terminals,

$$X = \{x_1, \cdots, x_k\} \subseteq V(G)$$

Definition 129 (Multiway Cut). *A **multiway cut** is a set of edges that leaves each of the terminals in a separate component*[31].

Definition 130 (Multiway Cut Problem). *The **Multiway Cut Problem** is the problem of finding a minimum weight set of edges $F \subseteq E(G)$ such that removing F from G separates all terminals*[32].

Theorem 131 (2-Approximation Multiway Cut). *The `MultiApprox` algorithm is a **2-approximation algorithm** for Multiway Cut.*

Proof. `MultiApprox` is a 2-approximation algorithm if,

1. It outputs a feasible cut C
2. It runs in polynomial time
3. Its cost is at most $2 \cdot$ `OPT`

The cut C_i can be computed efficiently at each iteration by running the `Ford-Fulkerson` algorithm to find maximum flow. We call a polynomial algorithm a constant number of times, so `MultiApprox` is polynomial. Moreover, $C = \bigcup_{i=1}^{k} \delta(C_i)$ is a feasible multiway cut[33].

[31] The Multiway Cut Problem is NP-complete for $k \geq 3$. When $k = 2$, this is precisely finding the minimum $(s-t)$ cut, which can be computed efficiently using `Ford-Fulkerson`.

[32] No connected component of $G(V, E - F)$ contains two terminals from S.

[33] To see this, note that for any pair x_i, x_j, $\delta(S_i)$ and $\delta(S_j)$ separate x_i and x_j.

Algorithm 10: 2-Approximation Multiway Cut

```
1 function MultiApprox(G, c)
2     foreach i ∈ [k] do
          // Find a minimum weight cut δ(S_i) separating x_i
             from a super-vertex X_i := X − {x_i}
3         X_i ← X − {x_i}
4         C_i ← Ford-Fulkerson(G, c, x_i, X_i)
      // Return the union of each cut C_i
5     C ← ⋃_{i=1}^{k} δ(C_i)
6     return C
```

Let OPT denote the optimal multiway cut in G. Then $G - \texttt{OPT}$ has components $T_1, \cdots, T_k$, where $x_i \in T_i$. Thus, $\texttt{OPT} = \bigcup_{i=1}^{k} \delta(T_i)$ and $c(\texttt{OPT}) = \frac{1}{2} \cdot \sum_i c(\delta(T_i))$ since $e \in \texttt{OPT}$ appears in two of the $\delta(T_i)$,

$$\begin{aligned} c(\texttt{OPT}) &= \frac{1}{2} \cdot \sum_i c(\delta(T_i)) \\ &\geq \frac{1}{2} \cdot \sum_i c(\delta(C_i)) \\ &= \frac{1}{2} \cdot c(C) \end{aligned}$$

where the second last line follows by the minimality of the cut returned by `Ford-Fulkerson`, and the last line follows by the union bound. That is, an edge $e \in C$ can appear in only one $\delta(C_i)$ since $C_1 \cup \cdots \cup C_k$ need not equal $V(G)$. □

Application 3: Weighted Vertex Cover

Given a graph $G = (V, E, c)$ and non-negative costs c, the goal is to find a minimum cost vertex cover $S \subseteq V(G)$.

Theorem 132. *The `GMatching` algorithm is a **2-approximation algorithm** for the Unweighted Vertex Cover Problem*[34].

[34] The following bound can be seen to be tight by looking at the unweighted vertex cover for a star.

Proof. `GMatching` is a 2-approximation algorithm if,

1. It outputs a feasible vertex cover C
2. It runs in polynomial time
3. Its cost is at most $2 \cdot \texttt{OPT}$

`GMatching` runs in polynomial time because we can find a maximal matching and its endpoints in polynomial time. Moreover, `GMatching`

Algorithm 11: 2-Approximation Vertex Cover

```
1 function GMatching(G, c)
      // Find a maximal cardinality matching M in G
2     M ← MaxMatching(G)
      // Output C, the end vertices of edges in M
3     C ← V(M)
4     return C
```

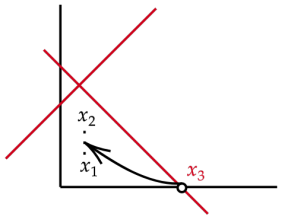
outputs a feasible solution because the maximality of M guarantees that C is a vertex cover of G. Since $C^c := V - C$ is an independent set, adding any edge $(i,j) \in E(G - C)$ to M creates a bigger matching.

We need to show that $|C| \leq 2 \cdot \texttt{OPT}$. Let C^* be the minimum vertex cover in G, so that $|C^*| = \texttt{OPT}$. Then $\texttt{OPT} \geq |M|$, where M is the maximal matching in G. This is because C^* must contain at least one endpoint of each edge in M. But, $|C| = 2 \cdot |M|$, so,

$$|C| = 2 \cdot |M| \leq 2|C^*| = 2 \cdot \texttt{OPT}$$

□

Recall: A basic solution to a linear program is a feasible solution that is not a convex combination of two other feasible solutions.



Recall: A convex combination is a linear combination of points where all coefficients are non-negative and sum to 1. Moreover, there is always an optimal solution that is basic.

Theorem 133. *The* `Rounding` *algorithm is a* ***2-approximation algorithm*** *for the weighted Vertex Cover Problem. The integer program is,*

$$\begin{array}{lll} \text{minimize} & \sum_{i \in V(G)} c_i \cdot x_i & \\ \textit{subject to} & x_i + x_j \geq 1 & \forall (i,j) \in E(G) \\ & x_i \in \{0,1\} & \forall i \in V(G) \end{array}$$

but it requires exponential time to solve. Relaxing to a linear program,

$$\begin{array}{lll} \text{minimize} & \sum_{i \in V(G)} c_i \cdot x_i & \\ \textit{subject to} & x_i + x_j \geq 1 & \forall (i,j) \in E(G) \\ & x_i \in [0,1] & \forall i \in V(G) \end{array}$$

we output $C := \{i \in V \mid x_i \geq \frac{1}{2}\}$.

Proof. `Rounding` is a 2-approximation algorithm if,

1. It outputs a feasible vertex cover C
2. It runs in polynomial time
3. Its cost is at most $2 \cdot \texttt{OPT}$

`Rounding` clearly runs in polynomial time since we have applied relaxation. We need to show that the algorithm outputs a feasible

vertex cover. Consider any edge $(i,j) \in E(G)$. By the linear programming constraints, $x_i + x_j \geq 1$, so,

$$\max\{x_i, x_j\} \geq \frac{1}{2}(x_i + x_j) = \mu$$

and at least one endpoint of (i,j) is in C.

To find the approximation guarantee, let C^* be the minimum cost vertex cover in G. $\hat{x}$ is a feasible linear programming solution if[35],

[35] The optimal solution is feasible.

$$\hat{x}_i = \begin{cases} 1 & i \in C^* \\ 0 & i \notin C^* \end{cases}$$

The optimal relaxed linear programming solution x satisfies,

$$c(x) = \sum_{i \in V} c_i x_i \leq \sum_{i \in V} c_i x_i^* = \texttt{OPT}$$

This gives that,

$$\begin{aligned} c(C) &= \sum_{i : x_i \geq 1/2} c_i \\ &\leq \sum_{i : x_i \geq 1/2} c_i(2x_i) \text{ since } 2x_i \geq 1 \\ &\leq 2 \cdot \sum_{i \in V} c_i x_i \\ &= 2 \cdot \texttt{OPT} \end{aligned}$$

□

Theorem 134. *The linear program for Vertex Cover is* $1/2$*-integral. That is, any basic solution* $\vec{x}$ *satisfies that* $x_i \in \{0, 1/2, 1\}$.

Proof. Let $\vec{x}$ be a basic solution to our linear program. We can assume that $\vec{x}$ is fractional, i.e., $0 < x_i < 1$. Suppose not. Then,

1. $x_i = 1$ for some $i \in V$. Select $i \in C$ and consider $G - i$.
2. $x_i = 0$ for some $i \in V$. Discard $i \in C$ and consider $G - i$[36]

[36] Any edge with x_i satisfies that its other endpoint is assigned a value of 1. This follows from the constraints of our integer program, i.e., $x_i + x_j \geq 1$.

We need to prove that any x_i with a fractional value is $1/2$.

$$\begin{aligned} V^+ &:= \{i \mid x_i > 1/2\} \\ V^- &:= \{i \mid x_i < 1/2\} \\ V^{1/2} &:= \{i \mid x_i = 1/2\} \end{aligned}$$

Observe that $\vec{x} = \frac{1}{2}(\vec{y} + \vec{z})$, where y_i and z_i are defined as follows,

$$y_i = \begin{cases} x_i & i \in V^{1/2} \\ x_i + \delta & i \in V^+ \\ x_i - \delta & i \in V^- \end{cases} \qquad z_i = \begin{cases} x_i & i \in V^{1/2} \\ x_i - \delta & i \in V^+ \\ x_i + \delta & i \in V^- \end{cases}$$

so $\vec{x}$ is a convex combination of $\vec{y}$ and $\vec{z}$. We will show that $\vec{y}$ and $\vec{z}$ are feasible solutions to obtain a contradiction. There can be no edges within V^- or between V^- and $V^{1/2}$ since the constraint $x_1 + x_j \geq 1$ would not be satisfied. The only edges that we can have are,

1. E_1, between V^- and V^+
2. E_2, between V^+ and $V^{1/2}$
3. E_3, within V^+
4. E_4, within $V^{1/2}$

This implies that $\vec{y}$ is feasible for the linear program,

1. $(i,j) \in E_1$, then $y_i + y_j = (x_i + \delta) + (x_i - \delta) = x_i + x_j \geq 1$
2. $(i,j) \in E_4$, then $y_i + y_j = x_i + x_j \geq 1$
3. $(i,j) \in E_2$, then $y_i + y_j = x_i + (x_j - \delta) = x_i + x_j - \delta \geq 1$
4. $(i,j) \in E_3$, then $y_i + y_j = (x_i - \delta) + (x_j - \delta) = x_i + x_j - 2\delta \geq 1$

since edges in E_2 and E_3 are such that $x_i + x_j > 1$. Hence, we can choose δ so that $y_i + y_j \geq 1$. The case for the feasibility of $\vec{z}$ is analogous. Thus, $\vec{x} = \frac{1}{2}(\vec{y} + \vec{z})$, where $\vec{y}$ and $\vec{z}$ are feasible. This is a contradiction unless $\vec{y} = \vec{z}$, but then $V^- = V^+ = \emptyset$ so $x_i \in \{0, 1/2, 1\}$. □

Corollary 135. *Every basic solution is integral for a bipartite graph.*

Proof. Take a basic solution $\vec{x}$. We proved that $x_i \in \{0, 1/2, 1\}$, and we can reduce to the case where $\vec{x}$ is all-fractional. Defining $\vec{x} = \frac{1}{2}(\vec{y} + \vec{z})$, where y_i and z_i and L, R are the bipartitions of G,

$$y_i = \begin{cases} x_i + \delta & i \in L \\ x_i - \delta & i \in R \end{cases} \qquad z_i = \begin{cases} x_i - \delta & i \in R \\ x_i + \delta & i \in L \end{cases}$$

Every edge has one endpoint in L and the other in R,

1. $y_i + y_j = (x_i + \delta) + (x_j - \delta) = x_i + x_j \geq 1$
2. $z_i + z_j = (x_i - \delta) + (x_j + \delta) = x_i + x_j \geq 1$

implying that $L = R = \emptyset$. Hence, there are no vertices with $x_i = 1/2$. This implies that $\vec{x}$ is integral, that is, $x_i \in \{0, 1\}$. □

Corollary 136. *Vertex cover is polynomial for a bipartite graph*[37].

[37] We have a 1-approximation algorithm for bipartite graphs.

Theorem 137. *There is a* 3/2***-approximation algorithm*** *for the Vertex Cover Problem in a planar graph.*

Proof. Solve the linear program relaxation to obtain a solution $\vec{x}$ that is 1/2-integral. Let $V^1 := \{i \mid x_i = 1\}$ and $V^{1/2} := \{i \mid x_i = 1/2\}$. The subgraph of G with vertex set $V^{1/2}$ is planar. By the Four Color Theorem, it can be partitioned into four stable sets, Q_1, Q_2, Q_3, Q_4.

$$V^1 \cup Q_1 \cup Q_2 \cup Q_3$$

is a vertex cover[38]. The linear programming solution is $\sum_{i \in V} c_i \cdot x_i$. Without loss of generality, assume that,

[38] This effectively takes everything except V^0 and Q_4, but no edges bridge them by the linear program constraints.

$$\sum_{i \in Q_4} c_i x_i \geq \sum_{i \in Q_l} c_i x_i \qquad \forall l = \{1,2,3\}$$

But $x_i = 1/2$ for all $i \in \cup_i Q_i$. Hence,

$$\sum_{i \in Q_4} c_i \geq \sum_{i \in Q_l} c_i \qquad \forall l = \{1,2,3\}$$

This makes the cost of the algorithm,

$$\begin{aligned} c(A) &= \sum_{i \in V^1} c_i + \sum_{i \in Q_1 \cup Q_2 \cup Q_3} c_i \\ &\leq \sum_{i \in V^1} c_i \cdot x_i + \frac{3}{4} \cdot \sum_{i \in Q_1 \cup Q_2 \cup Q_3 \cup Q_4} c_i \\ &\leq \frac{3}{2} \cdot \sum_{i=V} c_i x_i \\ &\leq \frac{3}{2} \cdot \texttt{OPT} \end{aligned}$$

Thus, we have a 3/2-approximation algorithm for planar graphs[39]. □

[39] The second last inequality follows because the cost of the relaxed program is less than the cost of integer one.

Application 4: Set Cover

We are given n elements $V = \{v_1, \cdots, v_n\}$ and m sets $S_1, \cdots, S_m \subseteq V$ with costs $c_1, \cdots, c_m$. The goal is to find a minimum cost collection of sets that cover every element in V. This problem is NP-Complete.

Theorem 138 (Greedy Set Cover)**.** `GreedySet` *is a* $O(\log n)$*-**approximation algorithm** for the Set Cover Problem. The algorithm is greedy.*

Algorithm 12: Greedy Algorithm for Set Cover

```
1 function GreedySet(X, S)
    // U stores the uncovered elements
2   U ← X
    // C stores the sets of the cover
3   C ← ∅
4   while U ≠ ∅ do
      // Select the set S_j^* which covers the remaining
         uncovered elements at a minimum average cost
5     S_j ← S_j^*
6     C ← C ∪ S_j
7     U ← U − S_j^*
8   return C
```

Proof. The algorithm clearly runs in polynomial time and outputs a feasible set cover. We need to prove that $\alpha = O(\log n)$. Let,

$$\texttt{OPT} := \{S_1^*, \cdots, S_k^*\} \qquad \underbrace{C := \{S_1, \cdots, S_r\}}_{r \text{ possibly } \neq k}$$

and C as the output of `GreedySet`. We want to prove that,

$$\sum_{j=1}^{r} c(S_j) \leq \log n \cdot \sum_{j=1}^{k} c(S_j^*)$$

Label the elements of V by $\{1, 2, \cdots, n\}$, based on the order that they are covered by `GreedySet`. Let U_t be the set of uncovered elements at the start of step t[40]. If S_t is the set selected by `GreedySet` at t, then $n_t = |S_t \cap U_t|$ is the number of new elements covered at t. If $i \in S_t \cap U_t$, then i was covered at step t. Then,

$$\alpha_i = \frac{c(S_t)}{n_t}$$

is the cost of covering i. Hence, $\sum_{t=1}^{k} c(S_t) = \sum_{i=1}^{n} \alpha_i$. To see this,

$$\begin{aligned} \sum_{t=1}^{r} c(S_t) &= \sum_{t=1}^{r} \alpha_i \cdot n_t \\ &= \sum_{i=1}^{n} \alpha_i \quad (\text{where } |V| = n) \end{aligned}$$

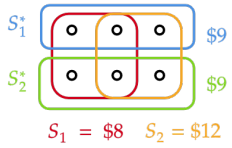


Figure 4: Illustration of `GreedySet`. Sets are sorted by average cost, where the average is taken over cardinality.

[40] $U_1 = V$ and $U_t \subset U_{t-1}$.

To analyze the cost of `GreedySet`, it suffices to bound α_i. When i was covered, there were at least $n-i+1$ uncovered elements. `OPT` can cover all of these elements for an average cost of,

$$\frac{1}{n-i+1} \cdot \left(\sum_{i=1}^{k} c(S_i^*) \right) = \frac{1}{n-i+1} \cdot \texttt{OPT}$$

Since S_t must do at least as well as this, $\alpha_i \leq \frac{1}{n-i+1} \cdot \texttt{OPT}$,

$$\begin{aligned}
\sum_{i=1}^{n} \alpha_i &\leq \sum_{i=1}^{n} \frac{1}{n-i+1} \cdot \texttt{OPT} \\
&= \texttt{OPT} \cdot \sum_{i=1}^{n} \frac{1}{n-i+1} \\
&= \texttt{OPT} \cdot \sum_{l=1}^{n} \frac{1}{l} \\
&= \texttt{OPT} \cdot H_n
\end{aligned}$$

where $H_n \approx \log n$ is the nth partial sum of a Harmonic series. Thus,

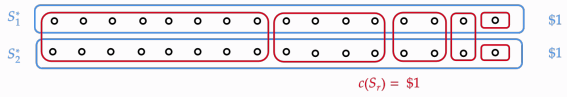
$$c(\{S_1, \cdots, S_r\}) \leq O(\log n) \cdot \texttt{OPT}$$

□

Example 36: Proof of Tighteness

Suppose that all sets have cost \$1. Then,

$$\begin{aligned}
c(\texttt{OPT}) &= c(\{S_1^*, S_2^*\}) = 2 \\
c(C) &= c(\{S_1, \cdots, S_r\}) = r
\end{aligned}$$



But, $r = \Omega(\log n)$ so $n = 2 + \sum_{j=1}^{r-1} 2^j$.

Remark 139. *There is no α-approximation for the Set Cover Problem with $\alpha = c \cdot \log n$ and $c < 1$ unless $P = NP$. We will not see the proof of this.*

Application 5: Hitting Set

We are given m elements $E = \{1, 2, \cdots, m\}$ with respective costs $c_1, c_2, \cdots, c_m$ and sets $T_1, T_2, \cdots T_n \subseteq E$. The goal is to find a minimum cost set of elements that "hit" every set $T_1, \cdots, T_n$.

Remark 140. *Since the Hitting Set Problem is equivalent to the Set Cover Problem, we can use `GreedySet` to solve it.*

Definition 141 (Randomized Rounding). *Let T_j be the indices of the sets S_j that cover element i. The integer problem to solve Hitting Set uses a randomized rounding algorithm,*

$$\begin{array}{lll} \text{minimize} & \sum_{j=1}^{m} c_i \cdot x_i & \\ \textit{subject to} & \sum_{j \in T_i} x_j \geq 1 & \forall T_i \\ & x_i \in \{0,1\} & \forall i \in V(G) \end{array}$$

and it can be relaxed as follows,

$$\begin{array}{lll} \text{minimize} & \sum_{j=1}^{m} c_i \cdot x_i & \\ \textit{subject to} & \sum_{j \in T_i} x_j \geq 1 & \forall T_i \\ & x_i \in [0,1] & \forall i \in V(G) \end{array}$$

where element j is selected with probability x_j.

Remark 142. *The expected cost of the randomized algorithm is,*

$$\sum_{j=1}^{m} c_j x_j = \texttt{LP} \leq \texttt{OPT}$$

where `LP` *is the value of our linear program. While the algorithm is polynomial, it may not output a feasible solution.*

Arithmetic-Mean Inequality: For any $\alpha_1 \cdots \alpha_k \geq 0$,

$$\frac{\sum_i \alpha_i}{k} \geq \sqrt[k]{\pi \alpha_i}$$

Remark 143. *If X_i be the event that T_i is hit, then $P(X_i) \geq 1 - \frac{1}{e}$.*

Proof. Put $T_i := \{1, 2, \cdots, k\}$. By the linear programming constraints, $\sum_{j=1}^{k} x_j \geq 1$. Hence, the probability that T_i is missed is,

$$\begin{aligned} P(X_i^c) &= \prod_{j=1}^{k} (1 - x_j) \\ &= \prod_{j=1}^{k} \alpha_j \\ &\leq \left(\frac{1}{k} \cdot (\alpha_1 + \cdots + \alpha_k \right)^k \\ &= (1 - \frac{1}{k} \cdot \sum x_j)^k \\ &\leq (1 - \frac{1}{k})^k \text{ since } \sum x_j \geq 1 \end{aligned}$$

But, $(1 - \frac{1}{k})^k \leq \frac{1}{e} = \lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n$. Thus, the probability that T_i is hit is strictly bigger than $\frac{1}{2}$. □

Corollary 144. *Our solution may miss half the sets. If the algorithm is run $\ell = 2 \log n$ times, then the probability of missing a set is reduced to $\frac{1}{n}$*[41].

[41] Consequently, with high probability, randomized rounding gives a hitting set with cost $\leq 2 \log n \cdot \texttt{OPT}$.

Application 6: Maximum Satisfiability

Let $x_1, x_2, \cdots, x_n$ be boolean variables. Recall that,

1. A positive literal is a variable x_i
2. A negative literal is a negation $\bar{x_i}$.
3. A clause is a disjunction of literals

The goal is to assign the variables `True` and `False` while satisfying as many as clauses possible.

Theorem 145. *Independently assigning each variable x_i to be `True` with probability $1/2$ is a $1/2$-approximation algorithm.*

Proof. Take a clause C_j with k literals. C_j is not satisfied with probability $\frac{1}{2^k}$. Therefore, C_j is satisfied with probability $1 - \frac{1}{2^k}$. The worst case occurs when $k = 1$, but $1 - \frac{1}{2^k} \geq \frac{1}{2}$. This means that each clause is satisfied with a probability of at least $\frac{1}{2}$. Let m be the total number of clauses. By linearity of expectation, the expected number of satisfied clauses is at least $\frac{1}{2} \cdot m \geq \frac{1}{2} \cdot \texttt{OPT}$. □

Remark 146. *This is a $7/8$-approximation algorithm for Maximum 3-SAT because each clause is satisfied with probability $1 - \frac{1}{2^3} = \frac{7}{8}$*[42].

[42] In fact, unless P = NP, there is no better approximation algorithm.

Theorem 147 (Randomized Satisfiability)**.** *The following integer program solves the Maximum Satisfiability Problem,*

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^{m} z_i \\ \textit{subject to} & \sum_{x_i \in C_j} y_j + \sum_{x_i \in C_j} (1 - y_i) \geq z_j \qquad \forall j \\ & y_i, z_j \in \{0, 1\} \qquad \forall i, j \end{array}$$

where clauses are indexed by j, variables are indexed by i, and each clause C_j corresponds to z_j. To see that this works[43]*, let,*

[43] The clause constraint is satisfied if and only if at least one of the literals are assigned correctly.

$$y_i = \begin{cases} 1 & \text{if } x_i = \texttt{True} \\ 0 & \text{if } x_i = \texttt{False} \end{cases} \qquad z_j = \begin{cases} 1 & \text{if } C_j \text{ satisfied} \\ 0 & \text{if } C_j \text{ not satisfied} \end{cases}$$

We solve the linear program relaxation in polynomial time with $0 \leq y_i \leq 1$ and $0 \leq z_i \leq 1$ for all i, j. To do this, set $x_i =$ `True` with probability y_i.

Recall: If $f(x)$ is concave on $[0, 1]$ with $f(0) = t$ and $f(1) = a + b$, then,

$$f(x) \geq ax + b$$

Proof. Since C_j has k literals, it is satisfied with minimum probability,

$$\left(1 + \left(1 + \frac{1}{k}\right)^k\right) \cdot z_j$$

We will prove this using the Arithmetic Mean Inequality. Without loss of generality, let C_j $x_1 \vee x_2 \vee \cdots \vee x_k$. Then,

$$\begin{aligned}\prod_{j=1}^{k}(1-y_i) &\leq \left(\frac{\sum(1-y_i)}{k}\right)^k \text{ by the Arithmetic Mean Inequality} \\ &= \left(1-\frac{\sum y_i}{k}\right)^k \\ &\leq \left(1-\frac{z_j}{k}\right)^k \text{ by the constraint } \sum_{x_i \in C_j} y_i \geq z_j\end{aligned}$$

is the probability that C_j is not satisfied. Note that $g(x) = \left(1-\frac{x}{k}\right)^k$ is convex, and consequently $f(x) = 1 - g(x)$ is concave. Thus,

1. $f(0) = (1-0)^k = 0 = b$
2. $f(1) = (1-\frac{1}{k})^k = a + b = a$ since $b = 0$

Hence, the probability that C_j is satisfied is greater than,

$$a \cdot z_j = \left(1-\frac{1}{k}\right)^k \cdot z_j$$

but we have seen that this gives a guarantee of $\frac{1}{e}$. Thus,

$$\geq \underbrace{\left(1-\frac{1}{e}\right)}_{\approx 0.632} \cdot z_k$$

The expected number of clauses satisfied is greater than or equal to,

$$\sum_j 0.632 \cdot z_j \geq 0.632 \cdot \sum_j z_j \geq 0.632 \cdot \sum_j \texttt{OPT}$$

since $\sum_j z_j$ is our objective function. □

Theorem 148. *Taking the best out of the previous two approximation algorithms, $\mathcal{R}_1$ and $\mathcal{R}_2$, gives a $3/4$-approximation algorithm.*

Proof. Let N_1 and N_2 be the number of clauses satisfied by $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. Simplifying and applying induction with,

$$\begin{aligned}\mathbb{E}[\max\{N_1, N_2\}] &\geq \mathbb{E}\left[\frac{1}{2} \cdot (N_1 + N_2)\right] \text{ since max} \geq \mu \\ &= \frac{1}{2} \cdot \mathbb{E}[N_1] + \mathbb{E}[N_2] \text{ by linearity of expectation}\end{aligned}$$

gives the result. The complete proof is not shown. □

Remark 149. *In fact, we can get a $3/4$ guarantee using non-linear randomized rounding. We solve the linear program to find y_i and z_i, and set $x_i =$ `True` with probability $f(y_i)$. This works if the function f satisfies,*

$$1 - \frac{1}{4^y} \leq f(y) \leq f^{y-1}$$

Proof. The probability that C_j is satisfied is,

$$\begin{aligned}
1 - \prod_{x_i \in C_j} (1 - f(y_i)) \cdot \prod_{\bar{x}_i \in C_j} f(y_i) &\geq 1 - \prod_{x_i \in C_j} \left(1 - \left(1 - \frac{1}{4y_i}\right)\right) \cdot \prod_{\bar{x}_i \in C_j} 4^{y_i - 1} \\
&= 1 - \prod_{x_i \in C_j} \frac{1}{4y_i} \cdot \prod_{\bar{x}_i \in C_j} 4^{y_i - 1} \\
&= 1 - 4^{-\sum_{x_i \in C_j} y_i + \sum_{\bar{x}_i \in C_j} (y_i - 1)} \\
&= 1 - 4^{-\left(\sum_{x_i \in C_j} y_i + \sum_{\bar{x}_i \in C_j} (1 - y_i)\right)} \\
&\geq 1 - 4^{2j} \text{ by our constraints}
\end{aligned}$$

Using concavity, $f(z_j) \geq \frac{3}{4} \cdot z_j$. □

Example 37: Proof of Tightness

This analysis is tight with respect to the upper bound,

1. $C_1 = x_1 \vee x_2$
2. $C_2 = x_1 \vee \bar{x}_2$
3. $C_3 = \bar{x_1} \vee x_2$
4. $C_4 = \bar{x_1} \vee \bar{x_2}$

Moreover, it is tight with respect to `OPT`,

1. $C_1 = x_1 \vee x_2$
2. $C_2 = x_1 \vee \bar{x_2}$
3. $C_3 = \bar{x_1} \vee x_2$
4. $C_4 = \bar{x_1} \vee \bar{x_3}$

Application 7: Steiner-Tree Problem

Definition 150 (Steiner-Tree Problem)**.** *Suppose that we are given a graph $G = (V, E)$ with edge costs $c_e \geq 0$ and a set $R \subseteq V$ of terminals. The goal is to find a minimum cost subgraph T that connects all the terminals. This subgraph is called a **Steiner tree**. The non-terminals are called **Steiner nodes**, and they are only used if they reduce the cost of the tree*[44].

[44] Remark that the leaves of a Steiner tree are necessarily terminals.

Corollary 151. *If $R = V(G)$, then this is the minimum spanning tree problem. If $R \subset V(G)$, then the problem is NP-Complete.*

Theorem 152. `SteinerApprox` *is a 2-approximation algorithm.*

Proof. `SteinerApprox` is a 2-approximation algorithm if,

Algorithm 13: 2-Approximation Algorithm for Steiner Trees

```
1 function SteinerApprox(X, S)
    // For terminals r_1, r_2, let P_ij be the shortest path
       between them.  Denote its length by ℓ_ij
2   P_ij ← δ(i, j)
    // Construct an auxiliary graph H that is a
       complete graph on the set of terminals.  Let
       (i, j) ∈ H have cost ℓ_ij
3   T ← Prim(H, ℓ)
4   return ⋃_{(i,j)∈T} P_ij
```

1. It outputs a feasible vertex cover C
2. It runs in polynomial time
3. Its cost is at most $2 \cdot \texttt{OPT}$

`SteinerApprox` is polynomial, and it outputs a feasible solution. Let T^* be the optimal Steiner tree. We can walk around the outside of T^* to create a circuit C. This circuit can be divided into paths,

$$C = Q_1 \cup Q_2 \cup \cdots \cup Q_k$$

where Q_i is the path from r_i to r_{i+1}. Thus,

$$c(C) = 2 \cdot c(T^*) = \sum_{i=1}^{k} c(Q_i)$$

but the path $P = \{r_1, \cdots, r_k\}$ is a spanning tree in H. Thus,

$$\begin{aligned} c(F) &\leq c(P) \\ &= \sum_{i=1}^{k-1} \ell_{i,i+1} \\ &\leq \sum_{i=1}^{k-1} Q_i \\ &\leq \sum_{i=1}^{k} c(Q_i) \\ &= 2 \cdot \texttt{OPT} \end{aligned}$$

Since we return $T' = \bigcup_{(i,j)\in T} P_{ij}$, we have,

$$C(T') \leq c(F) \leq 2 \cdot \texttt{OPT}$$

□

Application 8: Knapsack Problem

Suppose that we are given a bag with capacities w and n objects, where each object i has weight w_i and value v_i. The goal is to find the subset of items of maximum value that fit in the Knapsack. We saw that this can be formulated as an integer program,

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{n} v_i \cdot x_i \\ \text{subject to} & \sum_{i=1}^{n} w_i \cdot x_i \leq W \\ & x_i \in \{0,1\} \quad \forall i \in [n] \end{array}$$

Lemma 153. *For a basic solution $\vec{x}$, there is at most one item with,*

$$0 < x_i < 1$$

Proof. Recall the Greedy Algorithm for the Knapsack Problem,

1. Compute the value per weight $V_i := v_i/w_i$ for each item

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$$

2. Iterating through the sorted list, put,

$$x_i^G = \begin{cases} 1 & \text{if } i \text{ fits completely} \\ 0 & \text{if the knapsack is full} \\ \frac{W - \sum_{l=1}^{i-1} w_l}{w_i} & \text{otherwise} \end{cases}$$

After the first fractional item $0 < x_k^G < 1$, the bag is full. Thus,

$$\begin{aligned} \vec{x}^G &= \left(x_1^G, \cdots, x_{k-1}^G, x_k^G, x_{k+1}^G, \cdots x_n^G\right) \\ &= \left(1, \cdots, 1, 1, x_k^G, \cdots 0, \cdots, 0\right) \end{aligned}$$

An exchange argument shows that $\vec{x}^G$ is the optimal solution. Assume for a contradiction that it is not. Then some items before k are used below 1. Re-assign weight from x_j to x_i, where $i < k \leq j$ by setting,

$$\begin{aligned} x_j &\leftarrow x_j - \delta \\ x_i &\leftarrow x_i + \frac{\delta \cdot w_j}{w_i} \end{aligned}$$

to save $\delta \cdot w_j$ in weight. The change in value is,

$$-\delta \cdot v_j + \delta \cdot \frac{w_j}{w_j} \cdot v_i = \delta \cdot w_j \cdot \left(\frac{v_i}{w_i} - \frac{v_i}{w_j}\right) \geq 0$$

as i has a better "bang-for-buck" than j. We can repeat this until we obtain $\vec{x}^G$, but this means that $\vec{x}^G$ is an optimal basic solution[45]. □

[45] The solution is basic because it only has one fractional value.

Remark 154. *This gives the following polynomial* `BestOfTwo` *algorithm,*

1. *Solve the linear programming relaxation*
2. *Output the maximum value subset between,*

$$I_1 := \{i \mid x_i = 1\} \quad I_2 := \{i \mid 0 < x_i < 1\}$$

where I_1 and I_2 are both feasible solutions,

1. *I_1 is feasible since it has weight $\leq W$*
2. *I_2 is feasible since each weight is less than the size of the bag*[46]

[46] If not, then we can remove this item and recurse to obtain a solution with the same properties.

Proof. `BestOfTwo` is a 2-approximation algorithm because,

$$\begin{aligned}\max\{v(I_1), v(I_2)\} &\geq \frac{1}{2} \cdot (v(I_1) + v(I_2)) \\ &= \frac{1}{2} \sum_{i=1}^{n} v_i \\ &\geq \frac{1}{2} \sum_{i=1}^{n} v_i \cdot x_i \\ &\geq \frac{1}{2} \cdot \texttt{OPT}\end{aligned}$$

□

Definition 155 (Approximation Scheme)**.** *An algorithm $\mathcal{A}$ is a* ***fully polynomial time approximation scheme*** *for a maximization problem if,*

1. *$\mathcal{A}$ outputs a solution of value $\geq (1 - \epsilon) \cdot$ `OPT`*
2. *$\mathcal{A}$ runs in time polynomial in $|\texttt{I}|$ and $\frac{1}{\epsilon}$*

for any instance `I` *and $\epsilon > 0$.*

Theorem 156. *There is a fully polynomial time approximation scheme for the Knapsack Problem. It is based on dynamic programming.*

Proof. Let $w(i, V)$ be the minimum weight of a subset of the items $[i]$ of value $\geq V$. This is by convention infinite if the value of $[i]$ is less than V. The dynamic program can be solved recursively as $w(i, V) = \min\{w(i-1, V), w_i + w(i-1, V - v_i)\}$ with base cases $w(i, V) = 0$ for all $V \leq 0$. Let $V_{\max} := \max_i v_i$. Then there are n choices for i and at most $n \cdot V_{\max}$ choices for V. This means that there are $O(n^2 \cdot V_{\max})$ subproblems, solvable in $O(2)$.

Since $V_{\max}$ can be exponential in the number of bits, this is a pseudo-polynomial time algorithm. We can resolve this by scaling down each value without substantially losing accuracy. □

Parameterized Complexity

Definition 157 (Fixed-Parameter Tractable)**.** *A problem is **fixed parameter tractable** if it has an algorithm to solve it that runs in time $f(k) \cdot poly(n)$, where n is the problem input size, k is the size of the optimal solution, and f need not be a polynomial function.*

Example 38: Vertex Cover

If we are given that the optimal solution C^* to the Vertex Cover Problem has k vertices, then we can check in $O\left(m \cdot \binom{n}{k}\right) = O(m \cdot n^k)$ if a subset C is a vertex cover.
This running time is exponential in the size of the optimal solution, not in the input size. It serves as a motivating example for the question: *Can we separate the time dependency on n and k completely?* Specifically, we want an algorithm in,

$$O(\text{poly}(n) \cdot f(k))$$

where $f(k)$ is exponential, or worse, in k. If we can do this, then the problem is called **fixed parameter tractable**.

Lemma 158. *Let M^* be a maximum matching and C^* be a minimum vertex cover in a non-bipartite graph. Then,*

$$|M^*| \leq |C^*| \leq 2 \cdot |M^*|$$

Proof. Let $M^* := \{e_1, e_2, \cdots, e_l\}$, where $e_i = (u_i, v_i)$. Then $V - V(M^*)$ is an independent set in G[47]. Moreover, $V(M^*)$ is a vertex cover. This means that the minimum vertex cover C^* is at most the size of C,

$$|C^*| \leq |C| = 2 \cdot l = 2 \cdot |M^*|$$

□

[47] If not, then we could construct a larger matching than M^*.

Theorem 159. *Let G be a non-bipartite graph with minimum vertex cover C^* of size k. Then, C^* can be found in time $3^k \cdot poly(n)$.*

Proof. Find a maximum matching $M^* = \{e_1, \cdots, e_l\}$ in polynomial time. Observe that $l \leq k$, or else $|C^*| \geq k$. At least one endpoint $e_i = (u_i, v_i)$ is in C^*, so there are three possibilities for e_i,

$$u_i \in C^* \wedge v_i \notin C^*$$
$$u_i \notin C^* \wedge v_i \in C^*$$
$$u_i \in C^* \wedge v_i \in C^*$$

This gives 3^l possibilities, producing subsets $C_1, \cdots, C_{3^l}$. Moreover, we know that there exists j such that $C_j = C^* \cap V(M^*)$. To find the correct index j, we can try every possibility. There are two cases,

1. There is an edge e whose endpoints are both in $V(M^*) - C_j$. Then it cannot be covered by adding vertices of $V - M^*$ to C_j. Thus, this is not the correct choice and we can reject it.

2. There are no edges whose endpoints are both in $V(M^*) - C_j$. Any edge e incident to a vertex in C_j is already covered. Since $V - M^*$ is an independent set, any other edge f has one endpoint in $V(M^*) - C_j$ and the other in $V - V(M^*)$. To cover these edges, we select a vertex in $V - V(M^*)$ as C_j are the only vertices in $V(M^*)$ that touch at least one edge uncovered by C_j. Let W_j be the set of vertices in $V - V(M^*)$ that touch at least one edge uncovered by C_j. Thus, $\hat{C}_j = C_j \cup W_j$ is the smallest vertex cover C such that,

$$\hat{C}_j = C_j \cup W_j$$

   We output the smallest $\hat{C}_j$, which is the minimum vertex cover.

We conclude that the total running time is at most,

$$3^l \cdot \text{poly}(n) \leq 3^k \cdot \text{poly}(n)$$

□

We can repeat the same procedure for the Longest Path Problem.

Theorem 160. *Suppose that the longest path P^* contains exactly k vertices,*

$$P^* = \{v_1, v_2, \cdots, v_k\}$$

We can find P^* exhaustively in $O(k \cdot n^k)$ by,

1. *Taking every possible sequence P of k vertices*

2. *Testing if P is a path*

In fact, we can do better. A useful technique in the design of fixed parameter tractable algorithms is the color coding method.

Definition 161 (Color Coding)**.** *The **color coding method** colors each object in the search space, so that the algorithm can refine its search for monochromatic or panchromatic solutions.*

Remark 162. *The color coding method applies to the Longest Path Problem.*

Proof. Assume the longest path is $P^* = \{v_1, v_2, \cdots, v_k\}$. Randomly color the vertices of G with k colors. If $\text{col}(v_i) = i$ for all $1 \leq i \leq k$, then we can find P^* in linear time by Breadth-First Search:

1. Begin with vertices of color 1, and search for neighbors in color 2

2. From these neighbors, search for neighbors of neighbors of color 3

3. Repeat this procedure until color k

The probability that v_i is given color i is $\frac{1}{k}$, so the probability that every vertex in P^* is given the correct color is,

$$\left(\frac{1}{k}\right)^k$$

Suppose that the color coding algorithm is run t times. Each run is independent, so the probability of failure every time is at most,

$$\left(1 - \left(\frac{1}{k}\right)^k\right)^t = \left(1 - \frac{1}{k^k}\right)^t$$

Using the fact that $1 - x < e^{-x} \quad \forall x \neq 0$,

$$\left(1 - \frac{1}{k^k}\right)^t \leq \left(e^{-\frac{1}{k^k}}\right)^t = e^{-\frac{t}{k^k}}$$

If we try this $t = k^k \cdot \log n$ times, then the probability is at most,

$$e^{-\frac{t}{k^k}} = e^{-\log n} = \frac{1}{n}$$

So at least one of the trials succeeds with probability at least $1 - \frac{1}{n}$. Moreover, the algorithm runs in time $k^k \cdot \text{poly}(n)$, making the problem fixed parameter tractable. □