Efficient Algorithms

Summer Semester 2000

Prof. Dr. Sami Khuri[1]

Draft, Version of February 18, 2001

[1]These are the lecture notes of a course taught at Technische Universität München. The notes were typed by Christian Osendorfer and Johannes Altaner.

Contents

1 Maximum Flow **1**
- 1.1 Flow networks . . . 1
  - 1.1.1 Multiple sources and sinks . . . 5
- 1.2 The Ford-Fulkerson method . . . 5
  - 1.2.1 Residual networks . . . 6
- 1.3 Max-Flow Min-Cut Theorem . . . 12
- 1.4 The basic Ford-Fulkerson algorithm . . . 13
- 1.5 Analysis of the Ford-Fulkerson algorithm . . . 16
- 1.6 Variations of the Ford-Fulkerson algorithm . . . 18
  - 1.6.1 Edmonds-Karp algorithm . . . 18
  - 1.6.2 Preflow-push algorithms . . . 18
  - 1.6.3 The preflow-push procedure . . . 20
  - 1.6.4 Correctness of the preflow-push algorithm . . . 24
  - 1.6.5 Complexity of the preflow-push algorithm . . . 25
  - 1.6.6 Specific implementations of the preflow-push algorithm . . . 29
  - 1.6.7 FIFO preflow-push algorithm . . . 30
- 1.7 Applications of the maximum flow problem . . . 33
  - 1.7.1 Scheduling on uniform parallel machines . . . 33
  - 1.7.2 Distributed computing on a two-processor computer . . . 36
  - 1.7.3 The baseball elimination problem . . . 38

2 Graph Matching **46**
- 2.1 Maximum Bipartite Matching . . . 50
- 2.2 Applications . . . 53
  - 2.2.1 Bipartite Personnel Assignment . . . 53
  - 2.2.2 Non bipartite Personnel Assignment . . . 54
- 2.3 Transversals for families of subsets . . . 54
- 2.4 Hall's theorem . . . 55
- 2.5 Maximum Matchings and Minimum Vertex Covers . . . 58
  - 2.5.1 The Bottleneck Problem . . . 61
- 2.6 Weighted matching . . . 62

3 String Matching **64**
- 3.1 Naive String Matching . . . 64
- 3.2 String Matching Algorithms [CLR90] . . . 65
- 3.3 Knuth-Morris-Pratt Algorithm [KMP] . . . 65
  - 3.3.1 Analysis of Prefix-Function . . . 70
  - 3.3.2 Analysis of KMP . . . 70
- 3.4 Rabin-Karp Algorithm . . . 71
  - 3.4.1 Analysis of Rabin-Karp . . . 73
- 3.5 Boyer-Moore . . . 74
  - 3.5.1 The Bad-Character Heuristic . . . 75

3.5.2 The Good-Suffix Heuristic 77
3.5.3 Analysis of Boyer-Moore 79
3.5.4 Longest Common Subsequence 79
3.5.5 More Applications: Computer Viruses 81
3.5.6 The Longest Upsequence [by E.W.Dijkstra] 83

4 Approximation Algorithms **84**
4.1 The vertex cover problem 84
4.2 The Traveling-Salesperson Problem 85
4.3 The Set-Covering Problem 87
4.3.1 Analysis of Greedy-Set-Cover 88
4.4 The Subset-Sum problem 90

Chapter 1

Maximum Flow

Flow in a network can be:

a.) flow of oil or water through a system of pipelines

b.) flow of electricity

c.) flow of phone calls

d.) flow of email messages

e.) flow of commodities being transported across truck routes

The flow of the material at any point in the system is the rate at which the material moves.

Directed edges in flow networks are conduits for the material. Each edge has a capacity which is the maximum rate at which the material can flow through the conduit.

Vertices are conduit junctions. There are two special vertices: the source and the sink.

Material flows through all other vertices without stopping at the vertices.

The study of network flow bridges several diverse and seemingly unrelated areas of combinatorial optimization, such as matchings used to solve scheduling and assignment problems, the edge and vertex connectivity problems, and the minimum cut problem.

1.1 Flow networks

A flow network is a directed graph $G = (V, E)$ with two special vertices: the source $s \in V$ and the sink $t \in V$ [CLR90]. Each edge $(u, v) \in E$ has a nonnegative capacity $c(u, v) \geq 0$. We assume that:

a.) $c(u, v) = 0$ if $(u, v) \notin E$

b.) every vertex lies on some path from s to t (so the graph is connected)

A flow in G is a real-valued function $f : V \times V \to \mathbf{R}$ such that for all $u \in V$, $v \in V$:

a.) $f(u, v) \leq c(u, v)$ – capacity constraint

b.) $f(u, v) = -f(v, u)$ – skew symmetry
Note that this implies that $f(u, u) = 0$, because $f(u, u) = -f(u, u)$

c.) $\sum_{v \in V} f(u, v) = 0$ for all $u \in V \backslash \{s, t\}$ – flow conservation

The quantity $f(u, v)$ is called the net flow from vertex u to vertex v. The value of a flow f is the total net flow out of the source and is denoted by $|f|$. In other words:

$$|f| = \sum_{v \in V} f(s, v)$$

The maximum-flow problem consists in finding a flow of maximum value from s to t for a given flow network G with source s and sink t. The positive net flow entering a vertex $v \in V$ is given by:

$$\sum_{u \in V, f(u,v) > 0} f(u, v)$$

The positive net flow leaving a vertex $v \in V$ is given by:

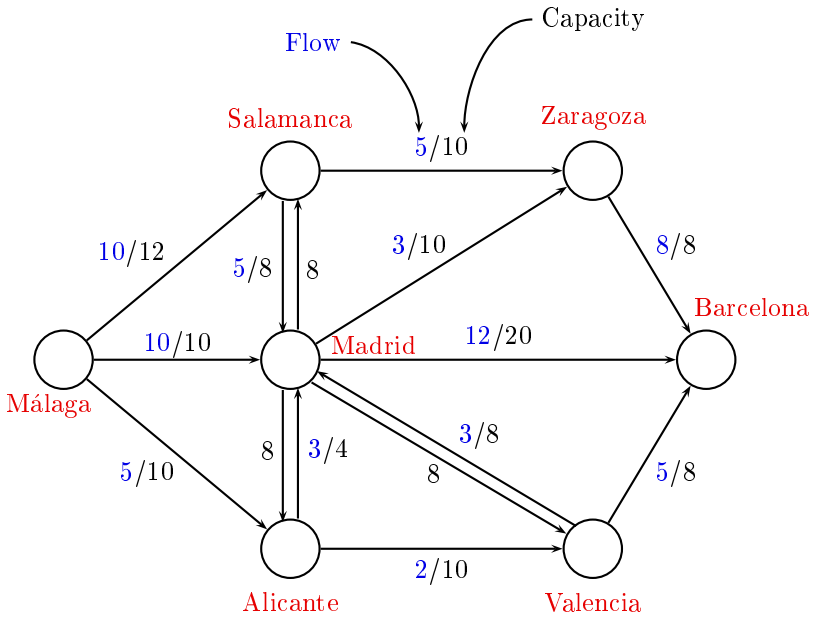
$$\sum_{u \in V, f(v,u) > 0} f(v, u)$$

By the flow conservation property, the positive net flow entering $v \in V$ where $v \neq s$ and $v \neq t$, is equal to the positive net flow leaving vertex $v \in V$.

EXAMPLE: The ANDALUCIA ACEITUNA COMPANY (AAC) buys black and green olives form the farmers in southern Spain and puts them in jars in Málaga. The jars that are not destined for local consumption are placed in one-ton containers, called crates, and are then sent to Barcelona to be shipped to the rest of Europe.

AAC does not own trucks and therefore leases space on trucks from the BARCELONA CAMIÓN COMPANY (BCC) trucking company. The trucks of BCC travel over specified routes between major cities in Spain and have only limited capacity for AAC's crates since BCC also does business with other manufacturers.

So AAC has no control over the routes and capacities of the trucks of BCC. AAC can ship at most $c(u, v)$ crates per day between each pair of cities u and v. The goal of AAC is to ship the largest number p of crates per day that can reach Barcelona (that same day). Recall that olives cannot pile up in intermediate cities. The rate at which olives enter an intermediate city is equal to the rate at which olives leave. In our EXAMPLE we have: $s =$ Málaga and $t =$ Barcelona. The intermediate cities are: Salamanca, Zaragoza, Madrid, Alicante, and Valencia. Only $c(u, v)$ crates per day can go from u to v. For example: AAC can move a maximum number of 8 crates of olives from Madrid to Alicante per day. Only positive netflows are shown in the diagram. If $f(u, v) > 0$, then edge (u, v) is labeled with $f(u, v)/c(u, v)$.



The flow in the above example is:

$$|f| = \sum_{v \in V} f(\text{Málaga}, v) = 25$$

Is this the maximum possible flow ?

For the above figure, please consider: edges with one number only, such as Madrid–Alicante, have zero flow. Also note that in the example:

$$\begin{aligned} f(\text{Salamanca, Madrid}) &= 5 \\ f(\text{Madrid, Salamanca}) &= -5 \quad \text{by skew symmetry} \end{aligned}$$

likewise:

$$\begin{aligned} f(\text{Madrid, Valencia}) &= 3 \\ f(\text{Valencia, Madrid}) &= -3 \end{aligned}$$

and

$$\begin{aligned} f(\text{Madrid, Zaragoza}) &= 3 \\ f(\text{Zaragoza, Madrid}) &= -3 \\ f(\text{Madrid, Barcelona}) &= 12 \\ f(\text{Barcelona, Madrid}) &= -12 \end{aligned}$$

So that

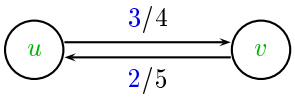
$$\begin{aligned} f(\text{Madrid}, v) &= \sum_{v \in V} f(\text{Madrid}, v) \\ &= f(\text{Madrid, Salamanca}) + f(\text{Madrid, Zaragoza}) \\ &+ f(\text{Madrid, Barcelona}) + f(\text{Madrid, Valencia}) \\ &+ f(\text{Madrid, Alicante}) + f(\text{Madrid, Málaga}) \\ &= (-5) + (+3) + (+12) + (+3) + (-3) + (-10) \\ &= 0 \quad \text{as expected by flow conservation} \end{aligned}$$

□

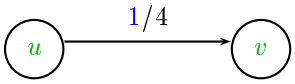
Remark: Without loss of generality, we can say that positive flow goes either from u to v, or from v to u but not both.

If we have positive flow in both directions, we can transform the scenario into an equivalent one which has only positive flow.

EXAMPLE:



We can transform it into the following scenario by "cancelling" 2 units of flow in each direction:



We do not show the edge from v to u since the flow is zero ($f(v,u) = 0$). Note that the flow from u to v in both cases is one unit. Note that the capacity constraint and flow conservation are still satisfied □

Notation: For $X \subseteq V$ and $Y \subseteq V$ we write:

$$f(X,Y) = \sum_{u \in X} \sum_{v \in Y} f(u,v)$$

Remarks: Let $G = (V,E)$ be a flow network and f a flow in G.

1. $f(X,X) = 0$ for all $X \subseteq V$
2. $f(X,Y) = -f(Y,X)$ for all $X \subseteq V$ and $Y \subseteq V$
3. for $X \subseteq V$, $Y \subseteq V$, $Z \subseteq V$ and $X \cap Y = \emptyset$, we have:
   (a) $f(X \cup Y, Z) = f(X,Z) + f(Y,Z)$
   (b) $f(Z, X \cup Y) = f(Z,X) + f(Z,Y)$

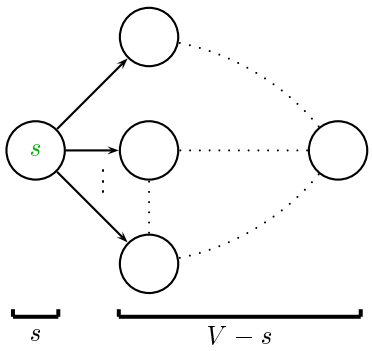
Result :

$$|f| = f(V,t)$$

for any flow f in a flow network

Proof: $|f| = \sum_{v \in V} f(s,v)$ by definition

$$= \sum_{u \in V} \sum_{v \in V} f(u,v) - \sum_{u \in V \setminus s} \sum_{v \in V} f(u,v)$$

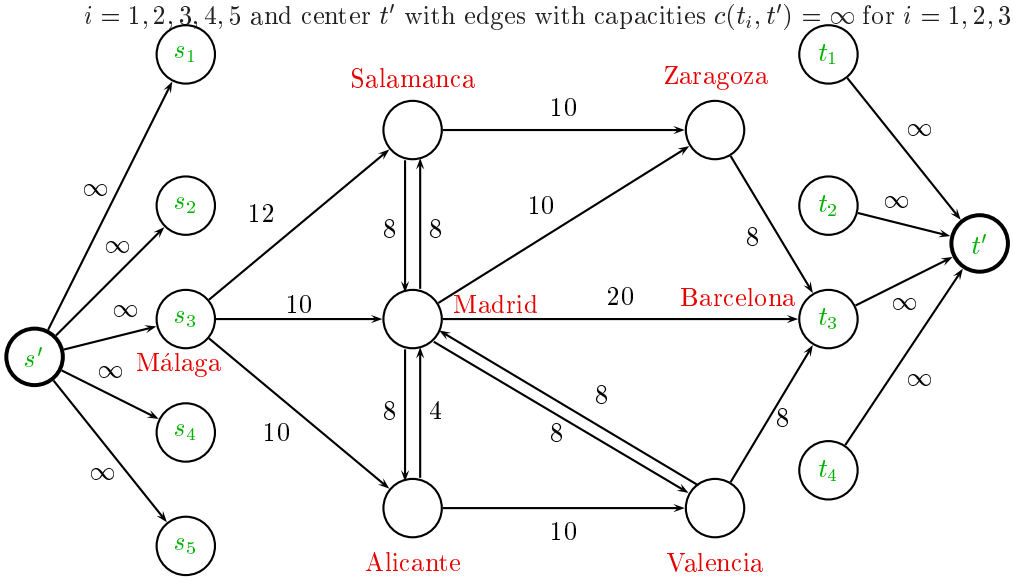


$$
\begin{aligned}
|f| &= \sum_{u\in V}\sum_{v\in V} f(u,v) - \sum_{u\in V\setminus s}\sum_{v\in V} f(u,v) \\
&= f(V,V) - f(V\setminus\{s\},V) \text{ by definition of } f(X,Y) \\
&= -f(V\setminus s,V) \text{ by Remark 1} \\
&= f(V,V\setminus s) \text{ by Remark 2} \\
&= f(V,V-\{s\}+\{t\}\cup\{t\}) \text{ by Remark 3}b) \\
&= f(V,t) \text{ by flow conservation}
\end{aligned}
$$

□

1.1.1 Multiple sources and sinks

A maximum flow problem may have more than one source, s_1, s_2, ..., s_m, and more than one sink, t_1, t_2, ..., t_3. Reduce the problem with multiple sources and sinks to one with only one source s' and only one sink t' by adding edges in the following way: (s', s_i) and costs $c(s', s_i) = \infty$ for $i = 1, 2, \ldots, m$ and (t_i, t') and costs $c(t_i, t') = \infty$ for $i = 1, 2, \ldots, n$.
EXAMPLE: If AAC puts olives in jars in centers s_1, s_2, s_3, s_4, s_5 and ships them to t_1, t_2, t_3, t_4; create a center s' and edges with capacities $c(s', s_i) = \infty$ for $i = 1, 2, 3, 4, 5$ and center t' with edges with capacities $c(t_i, t') = \infty$ for $i = 1, 2, 3, 4$.



□

1.2 The Ford-Fulkerson method

Solving the maximum flow problem consists in finding paths from s to t and pushing the maximum flow over these paths.

The Ford-Fulkerson method is based on three ideas:

- residual network
- augmenting paths
- cuts

It is called a method and not an algorithm because Ford-Fulkerson's method consists of several different algorithms with different running times.

FORD-FULKERSON-METHOD
initialize flow f to 0
while there exists an augmenting path P
do augment flow f along P
return f

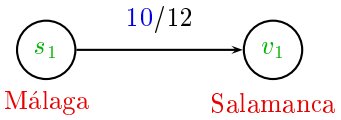
An augmenting path is a path from s to t along which we can push more flow.

1.2.1 Residual networks

Given a flow network $G = (V, E)$ with source s, sink t, and a flow f in G. The residual capacity of edge (u, v) is the net flow we can push from u to v before exceeding the capacity $c(u, v)$:

$$c_f(u, v) = c(u, v) - f(u, v)$$

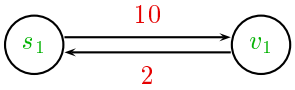
Example: Consider the edge joining Málaga to Salamanca in our previous example:



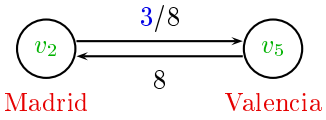
Let s_1 = Málaga, v_1 = Salamanca, $c(s_1, v_1) = 12$, and $f(s_1, v_1) = 10$. So $c_f(s_1, v_1) = 12 - 10 = 2$, and

$$\begin{aligned} c_f(v_1, s_1) &= c_f(v_1, s_1) - f(v_1, s_1) \\ &= 0 - (-10) \text{ since } f(s_1, v_1) = 10 \\ &= 10 \end{aligned}$$

So $c_f(v_1, s_1) = 10$. Let us replace edge (s_1, v_1) by edges (s_1, v_1) and (v_1, s_1) labeled with the residual capacities.



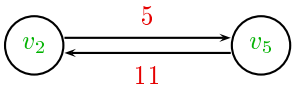
Similarly for edge (Madrid, Valencia).



$$\begin{aligned} c_f(v_2, v_5) &= c(v_2, v_5) - f(v_2, v_5) \\ &= 8 - 3 = 5 \\ c_f(v_5, v_2) &= c(v_5, v_2) - f(v_5, v_2) \\ &= 8 - (-3) = 11 \end{aligned}$$

(Note that $f(v_5, v_2) = 0 + (-f(v_2, v_5)) = -3$.)

By replacing the values between Madrid and Valencia by the residual capacities, we get:



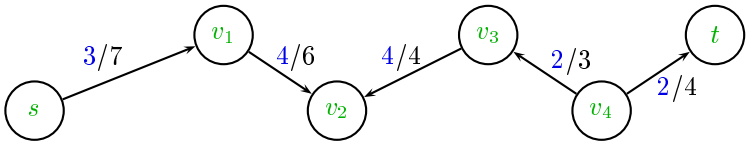
Note that when the net flow is negative, $f(v_5, v_2) = -3$ in the second example, then the residual capacity $c_f(v_5, v_2) = 11$ is greater than the capacity $c(v_5, v_2) = 8$. Are we violating the capacity from v_5 to v_2 ? No, since in reality we have 3 crates that go from v_2 to v_5, $f(v_2, v_5) = 3$, which cancel out 3 of the units that go from v_5 to v_2, and thus we have $11 - 3 = 8$ crates that travel from v_5 to v_2. □

The residual network of G, given a flow network $G = (V, E)$ and a flow f, is the graph $G_f = (V, E_f)$ induced by f, where:

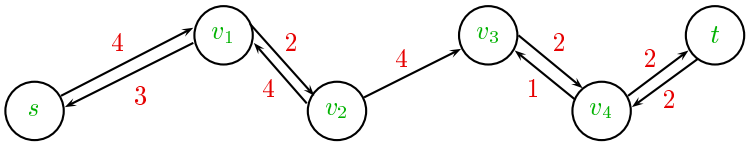
$$E_f = \{(u, v) \in V \times V \text{ such that } c_f(u, v) > 0\}$$

In other words, the edges of G are replaced by edges in G_f labeled with the residual capacities.

EXAMPLE: Consider the following path through G. Note that this is only one path (conservation law does not hold since we don't see the rest of the flow network).



path in G



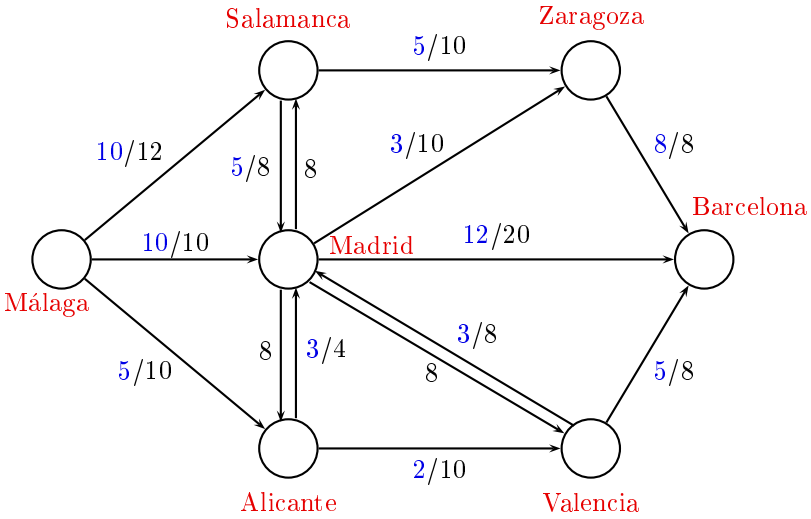
edges in the residual graph G_f

Note that the path in G can be augmented by 2 units. □

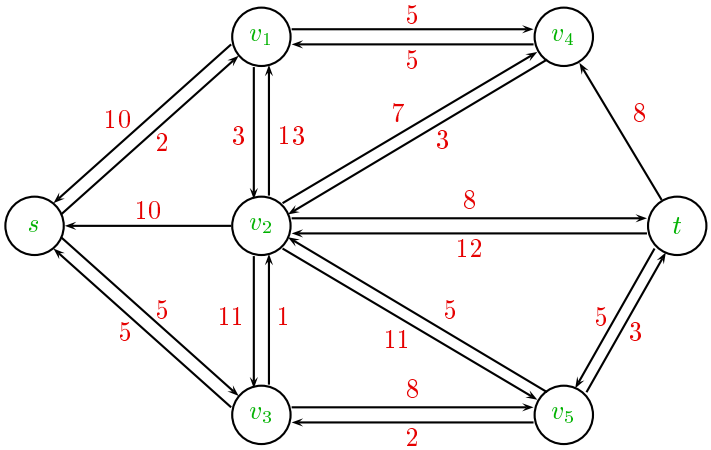
As pointed out in the example, new edges may emerge in G_f and thus E_f is rarely equal to E.

EXAMPLE: Let us consider the flow network of the AAC example with flow function f given by the BCC.

So we construct the residual network G_f induced by f.



Given flow network with flow f



residual network induced by f

Since an edge (u, v) can appear in G_f only if at least one of (u, v) and (v, u) appears in the original network G, we have:

$$|E_f| \leq 2|E|$$

In our example:

$$|E| = 15 \text{ and } |E_f| = 22$$

Also note that G_f is itself a flow network with capacities given by c_f. There exists a relationship between a flow in G and a flow in G_f.

Result: $G = (V, E)$ is a flow network with flow f. $G_f = (V, E_f)$ is the residual network induced by f. Let f' be a flow in G_f. Define the flow sum $f + f'$ to be $f + f' : (V \times V) \to \mathbf{R}$), where

$$(f + f')(u, v) = f(u, v) + f'(u, v).$$

Then:

a.) $f + f'$ is a flow in G

b:) $|f + f'| = |f| + |f'|$

Proof:

a:) We show that $f + f'$ is a flow in G:

i.) skew symmetry:

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &= -f(v, u) - f'(v, u) \\ &= -(f(u, v) + f'(u, v)) \\ &= -(f + f')(v, u)\end{aligned}$$

ii.) capacity constraints:

$$\begin{aligned}(f + f')(u, v) &= f(u, v) + f'(u, v) \\ &\leq f(u, v) + c_f(u, v) \\ \text{but } c_f(u, v) &= c(u, v) - f(u, v) \\ \Rightarrow (f + f')(u, v) &\leq f(u, v) + [c(u, v) - f(u, v)] \\ &= c(u, v) \\ \text{i. e. } (f + f')(u, v) &\leq c(u, v)\end{aligned}$$

iii.) flow conservation: for all $u \in V \backslash \{s, t\}$:

$$\begin{aligned}\sum_{v \in V} (f + f')(u, v) &= \sum_{v \in V} (f(u, v) + f'(u, v)) \\ &= \sum_{v \in V} f(u, v) + \sum_{v \in V} f'(u, v) \\ &= 0 + 0 \\ &= 0\end{aligned}$$

b.) We show that $|f + f'| = |f| + |f'|$

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

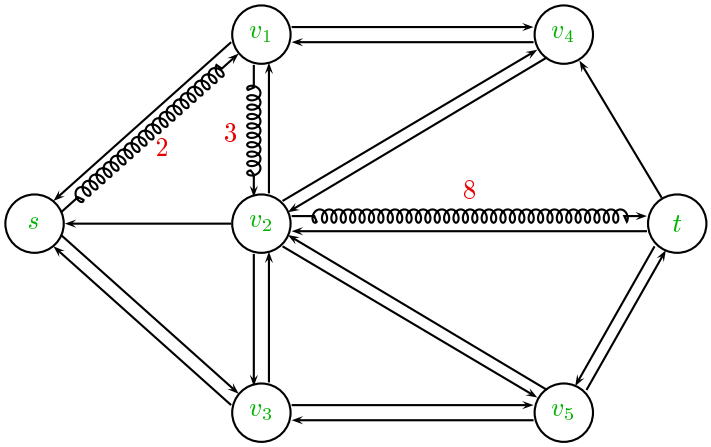
□

Recall that an augmenting path is a path such that we can push more flow through it. Given a flow network $G = (V, E)$ and a flow f, an augmenting path p is a simple path from s to t in the residual network G_f.

By finding a path from s to t in G_f, we know that we can push more flow through that path. We shall formally prove this fact later.

Let us consider our example once more and look for a path in G_f.

EXAMPLE:

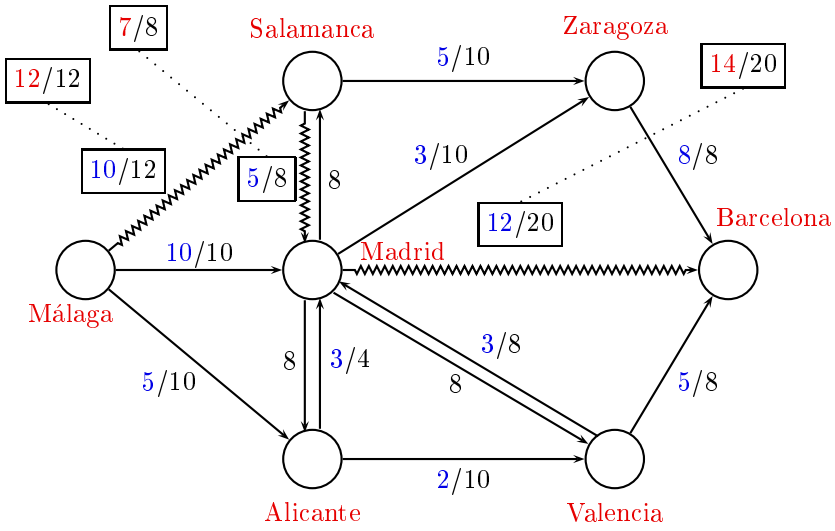


One augmenting path p is the one that goes from s to v_1 to v_2 to t. □

The residual capacity of a path p is:

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

It is the maximum amount of net flow that we can move along the edges of p. In our example, $c_f(p) = \min\{2,3,8\}$. So 2 more crates of olives could be shipped from Málaga to Barcelona via Salamanca then Madrid



The flow in G that results from augmenting along path p by its residual capacity $c_f(p) = 2$.

Now that we have a "new" flow f, we can repeat the process:

first: construct the new residual network

second: find an augmenting path p and compute its residual capacity

third: augment the path by its residual capacity

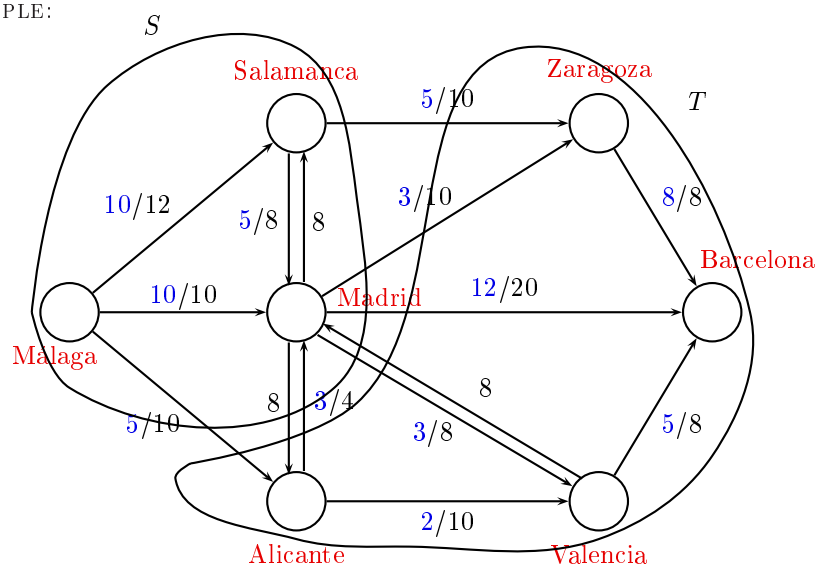
To prove that a flow is maximum if and only if its residual network contains no augmenting path, we need to study cuts first.

A cut (S, T) of a flow network $G = (V, E)$ is a partition of V into two disjoint sets $S \subseteq V$ and $T \subseteq V$ such that $S \cup T = V$, $s \in S$, and $t \in T$. The net flow across the cut (S, T) is $f(S, T)$ and the capacity of the cut (S, T) is $c(S, T)$.

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

Note that the net flow across a cut can have negative net flows between vertices. On the other hand, the capacity of a cut is made up of nonnegative-valued capacities.

EXAMPLE:



Consider the cut (S, T) where

$$S = \{s, v_1, v_2\} \text{ and } T = \{v_3, v_4, v_5, t\}$$

The net flow across the cut is:

$$\begin{aligned} f(S,T) &= f(v_1,v_4) + f(v_2,v_4) + f(v_2,t) + f(v_2,v_5) + f(v_2,v_3) + f(s,v_3) \\ &= 5 + 3 + 12 + 3 + (-3) + 5 = 25 \end{aligned}$$

And its capacity is:

$$\begin{aligned} c(S,T) &= c(v_1,v_4) + c(v_2,v_4) + c(v_2,t) + c(v_2,v_5) + c(v_2,v_3) + c(s,v_3) \\ &= 10 + 10 + 20 + 8 + 8 + 10 \\ &= 66 \end{aligned}$$

Remark: Not that $f(S, T) = 25$. But

$$|f| = \sum_{v \in V} f(s, v) = 10 + 10 + 5 = 25$$

That is no coincidence.

Result:

$$f(S, T) = |f|$$

Proof: Use Remark 1, 2, 3 seen earlier.

$$f(S,T) = f(S,V) - f(S,S)$$

$$
\begin{aligned}
&= f(S,V) \\
&= f(s \cup (S-s), V) \\
&= f(s,V) + f(S-s,V) \\
&= f(s,V) \\
&= |f|
\end{aligned}
$$

□

Remark: Note that because we have

$$|f| = f(S,T)$$

and

$$f(S,T) = \sum_{u \in V} \sum_{v \in T} f(u,v) \leq \sum_{u \in S} \sum_{v \in T} c(u,v)$$

and

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

we conclude that

$$|f| \leq c(S,T).$$

1.3 Max-Flow Min-Cut Theorem

Theorem 1.3.1 *If f is a flow in a flow network $G = (V,E)$ with source s and sink t, then the following statements are equivalent:*

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

Proof:

1 ⇒ 2: Proof by Contradiction: Suppose that f is a maximum flow in G but G_f contains an augmenting path p. The proof is a direct result of the following lemma:

Lemma: *If we define a function $f_p : V \times V \to \mathbf{R}$ such that:*

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

where p is an augmenting path in G_f, then f_p is a flow in G_f and $|f_p| = c_p(p) > 0$

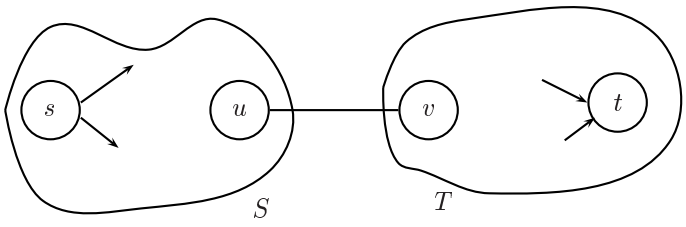
2 ⇒ 3: Suppose that G_f contains no path from s to t. Let

$$S = \{v \in V | \exists \text{ a path from } s \text{ to } v \text{ in } G_f\}$$

and let

$$T = V - S$$

(S,T) is a cut. $s \in S$ and since we have no path from s to t, $t \in T$.



For each pair (u, v) in the cut (with $u \in S$ and $v \in T$), we have $(u, v) \notin E_f$ (because we don't have a path from s to v). So for every such pair, we have $f(u, v) = c(u, v)$ (i.e. (u, v) has 0 residual capacity). We know that $f(S, T) = |f|$. But here $|f| = c(S, T)$, so $|f| = c(S, T)$.

$3 \Rightarrow 1$: We know by a previous result that $|f| \leq c(S, T)$, i.e. $c(S, T)$ is an upper bound for $|f|$. Since $|f| = c(S, T)$, we know that f is a maximum flow.

□

We are now ready to present the Ford-Fulkerson algorithm that simply implements Ford-Fulkerson's method.

1.4 The basic Ford-Fulkerson algorithm

The algorithm is labeled "basic" since after its appearance, many variations (and improvements) were designed.

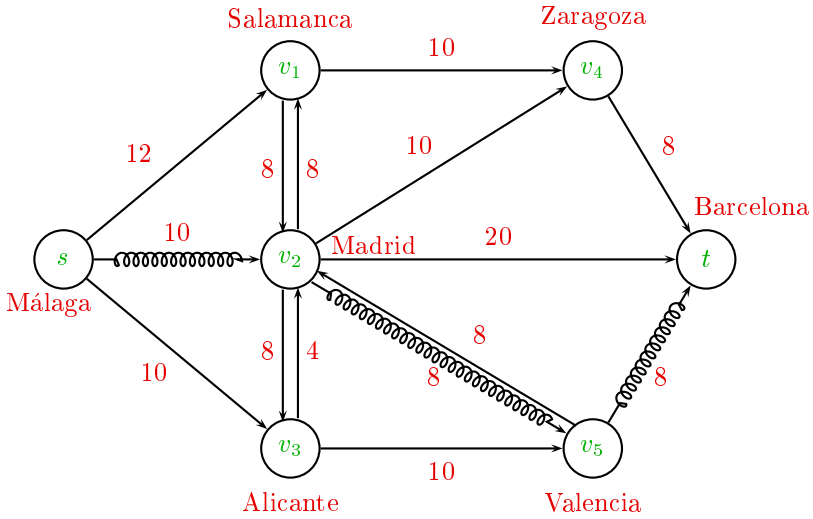
Ford-Fulkerson algorithm starts by initializing the flow to zero. It then constructs the residual graph G_f. The main loop of the algorithm will be executed as long as we can find an augmenting path p in G_f.

Recall that the residual capacity $c_f(u, v)$ is given by

$$c_f(u, v) = c(u, v) - f(u, v)$$

```
FORD-FULKERSON(G)
begin
  for each edge (u, v) ∈ E
    f(u, v) ← 0
    f(v, u) ← 0
  while there exists an augmenting path p in G_f
    c_f(p) ← min{c_f(u, v) | (u, v) is in p}        determine residual capacity
    for each edge (u, v) in p
      f(u, v) ← f(u, v) + c_f(p)
      f(v, u) ← −f(u, v)
    Find the new residual graph after having
    pushed c_f(p) along the path p
    Construct the residual graph G_f
  return the function f
end
```

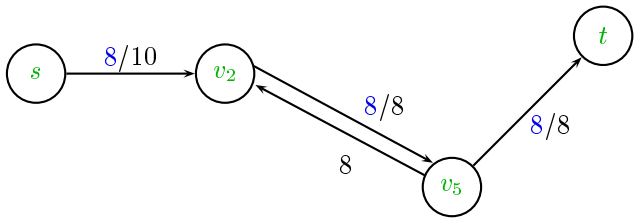
EXAMPLE: Consider our flow network once more with $f(u, v) = 0$ for all edges $(u, v) \in E$.



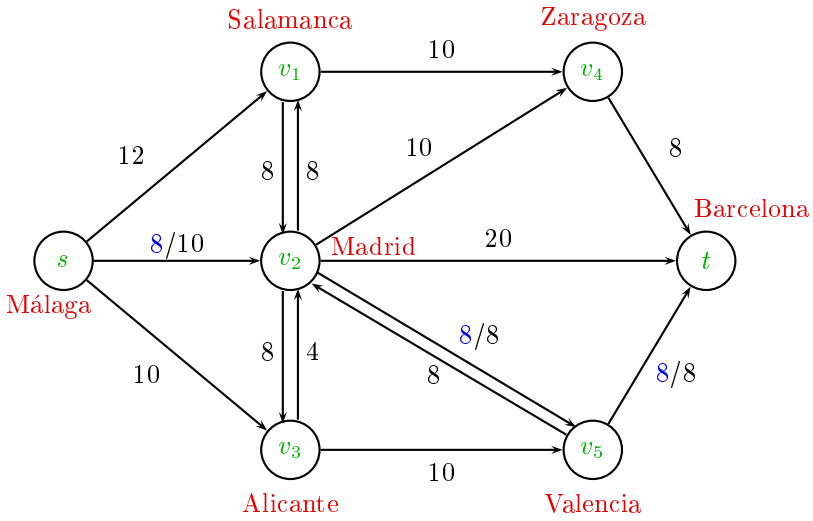
Assume the algorithm picks $s \to v_2 \to v_5 \to t$ to be the augmenting path p.

$$c_f(p) = \min\{10, 8\} \Rightarrow c_f(p) = 8$$

The flow network will be exactly the same as above except for the augmenting path, which will be:



The following is the flow network obtained after adding $c_f(p) = 8$ to the augmenting path p.

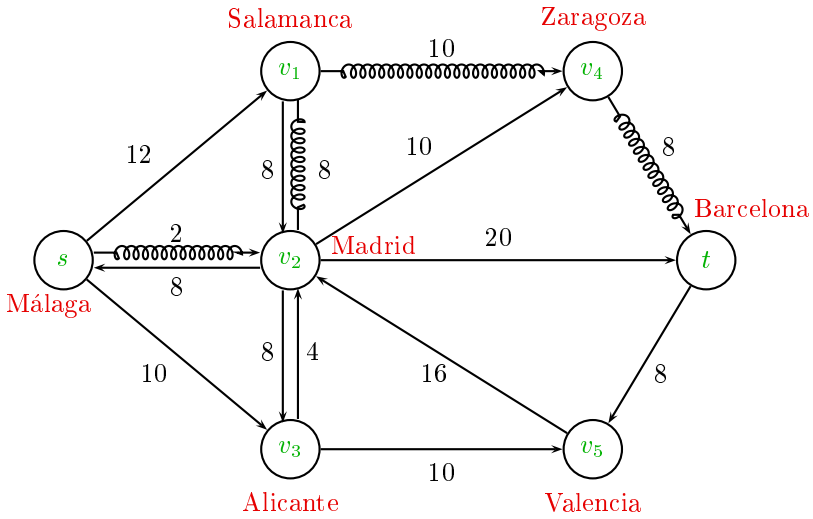


We now construct the residual network G_f of the above flow network

$$\text{edge}(s, v_2) : c_f(s, v_2) \quad = \quad c(s, v_2) - f(s, v_2)$$

$$
\begin{aligned}
&= 10 - 8 = 2 \\
\text{edge}(v_2, s) : c_f(v_2, s) &= c(v_2, s) - f(v_2, s) \\
&= 0 - (-8) = 8 \\
\text{edge}(v_2, v_5) : c_f(v_2, v_5) &= 8 - 8 = 0 \\
\text{edge}(v_5, v_2) : c_f(v_5, v_2) &= 8 - (-8) = 16 \\
\text{edge}(v_5, t) : c_f(s, v_2) &= 8 - 8 = 0 \\
\text{edge}(t, v_5) : c_f(s, v_2) &= 0 - (-8) = 8
\end{aligned}
$$

The new residual network G_f induced by the flow f in the previous figure.



One possible choice for an augmenting path p is:

$$s \to v_2 \to v_1 \to v_4 \to t$$

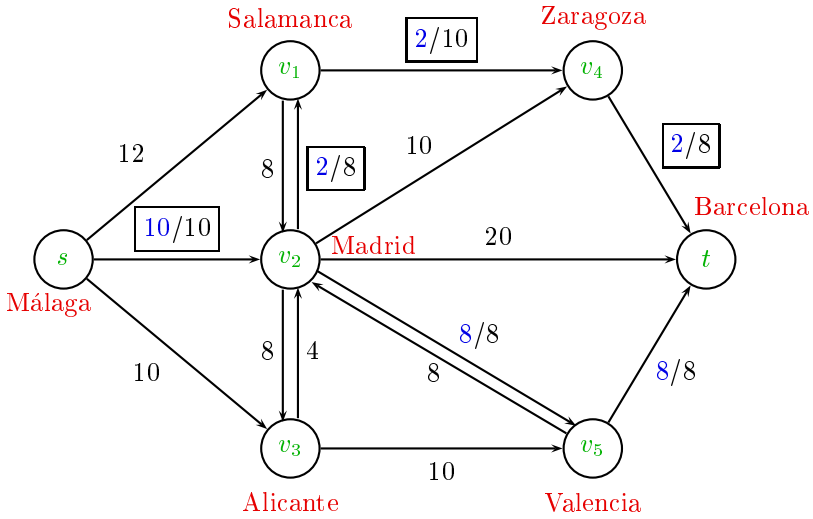
$$c_f(p) = \min\{2, 8, 10\} = 2$$

¿From the last flow network:

$$f(u, v) \leftarrow f(u, v) + c_f(p) \text{ and } f(v, u) \leftarrow -f(u, v)$$

In other words

$$
\begin{aligned}
f(s, v_2) &= 8 + 2 = 10 & f(v_2, s) &= -10 \\
f(v_2, v_1) &= 0 + 2 = 2 & f(v_1, v_2) &= -2 \\
f(v_1, v_4) &= 0 + 2 = 2 & f(v_1, v_4) &= -2 \\
f(v_4, t) &= 0 + 2 = 2 & f(t, v_4) &= -2
\end{aligned}
$$



The flow network after pushing two more crates along path p:

$$s \to v_2 \to v_1 \to v_4 \to t$$

Note that we don't show $f(u, v)$ when its value is nonpositive (as is the convention with flow networks). Nonpositive $c_f(u, v)$ are shown in residual networks only. The value/capacity that have changed, since the last iteration, are in the red rectangles.

The next step involves constructing the residual network G_f, then choosing an augmenting path and so on. The algorithm stops when no such path can be found in the residual network.

1.5 Analysis of the Ford-Fulkerson algorithm

Only one algorithm for the maximum flow-problem appeared before Ford–Fulkerson's algorithm. The 1951 network simplex method of Dantzig for the transportation problem solved the maximum flow problem as a natural special case.

Since the development of the augmenting path method by Ford and Fulkerson, many more efficient algorithms were (and are being) developed by several computer scientists.

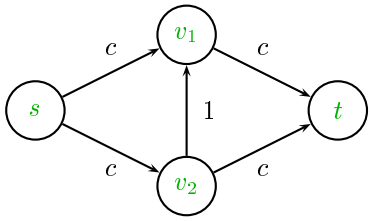
The following table from "Beyond the flow decomposition barrier" by A. Goldberg and Rao gives a history of maximum flow bounds.

year	discoverer(s)	exact bounds
1951	Dantzig	$O(n^2mU)$
1955	Ford & Fulkerson	$O(nmU)$
1970	Dinitz Edmonds & Karp	$O(nm^2)$
1970	Dinitz	$O(n^2m)$
1972	Edmonds & Karp Dinitz	$O(m^2 \log U)$
1973	Dinitz Gabow	$O(nm \log U)$
1974	Karzanov	$O(n^3)$
1977	Cherkassky	$O(n^2\sqrt{m})$
1980	Galil & Naamad	$O(nm \log^2 n)$
1983	Sleator & Tarjan	$O(nm \log n)$
1986	Goldberg & Tarjan	$O(nm \log(n^2/m))$
1987	Ahuja & Orlin	$O(nm + n^2 \log U)$
1987	Ahuja et al.	$O(nm \log(n\sqrt{\log U}/(m+2)))$
1989	Cheriyan & Hagerup	$O(nm + n^2 \log^2 n)$
1990	Cheriyan et al.	$O(n^3/\log n)$
1990	Alon	$O(nm + n^{8/3} \log n)$
1992	King et al.	$O(nm + n^{2+\epsilon})$
1993	Phillips & Westbrook	$O(nm(\log_{m/n} n + \log^{2+\epsilon} n))$
1994	King et al.	$O(nm \log_{m/(n \log n)} n)$
1997	Goldberg & Rao	$O(m^{3/2} \log(n^2/m) \log U)$ $O(n^{2/3}m \log(n^2/m) \log U)$

In the table, n denotes the number of vertices, m the number of edges and the capacities of the edges are integers in the range $[1 \ldots U]$.

Back to Ford-Fulkerson's algorithm. Its running time heavily depends on the way the augmenting path p in the head of the **while**-loop is chosen. The result can be pretty bad with poor choices of the path. Let us assume that all capacities have integral values. Let us also assume that the augmenting path is arbitrarily chosen. Under these assumptions, the Ford-Fulkerson algorithm runs in time $O(|E||f^*|)$, where $|f^*|$ is the maximum flow found by the algorithm. The computation is straightforward: It takes $\Theta(|E|)$ to initialize the edges of G and at most $|f^*|$ iterations of the **while**-loop are performed.

When the capacities are integral and the optimal flow value $|f^*|$ is small, we obtain a good running time. But if we have a network as shown in the figure below, the algorithm will take exponential time (even though the capacities have integer values).



1.6 Variations of the Ford-Fulkerson algorithm

1.6.1 Edmonds-Karp algorithm

A natural variant of the Ford-Fulkerson algorithm is to choose the shortest augmenting path in the residual network G_f. This algorithm can be implemented by using breadth first search and can be shown to run in polynomial time. It was independently developed by Dinitz and by Edmonds and Karp in the early 1970's.

Note that by shortest augmenting path we mean the path from s to t that contains as few edges as possible. As before, Edmonds-Karp algorithm repeats this step until no further paths can be found. A crucial observation is that the length of the shortest path increases with each iteration of the main loop in the algorithm. This fact is not obvious since new edges can be added to the residual graph as we augment a path. Note that the added edges are along the augmenting path and are directed in the opposite direction of the flow. It can be shown that the Edmonds-Karp algorithm runs in $O(|V||E|^2)$ time.

1.6.2 Preflow-push algorithms

The preflow-push algorithms are the fastest maximum-flow algorithms to date. Preflow-push algorithms work in a more localized fashion than the Ford-Fulkerson algorithm. Rather than examining the entire residual network to find an adequate augmenting path, preflow-push algorithms concentrate on one vertex at a time, looking only at the vertex's immediate neighbors in the residual network.

The flow-conservation property is not maintained throughout the execution. Instead, the algorithm maintains a preflow which is a function $f : V \times V \to \mathbf{R}$ that satisfies:

i) skew symmetry

ii) capacity constraint

ii) $f(V, u) \geq 0$ for all $u \in V \backslash \{s\}$

The third property of preflow is a relaxation of flow conservation that postulates that the net flow into each vertex other than the source is nonnegative.

The excess flow into u, given by $e(u) = f(V, u)$ is the net flow into u. A vertex $u \in V \backslash \{s, t\}$ is overflowing if $e(u) > 0$.

The algorithm can be understood, intuitively, by thinking of the flow network as being a system of interconnected pipes of given capacities. The Ford-Fulkerson method iteratively adds more streams of flow until no more can be added. On the other hand, vertices for preflow-push algorithms are pipe junctions with two properties: First, each vertex has an outflow pipe leading to an arbitrarily large reservoir, to accommodate excess flow. Second, each vertex, its reservoir, and all its pipe connections are on a platform whose height increases as the algorithm progresses. Naturally, flow is pushed from a high vertex to a lower vertex.

The height of the source is $|V|$ and of the sink is 0. Other vertex heights start at 0 and increase with time. The increase of height of a vertex is performed through the operation of lifting. The height is increased to one unit more than the height of the lowest of its neighbors to which it has an unsaturated pipe. So the lifting of the vertex will result in the pushing of flow to at least that unsaturated pipe. We now formally present the preflow-push algorithm. As usual, assume that $G = (V, E)$ is a flow network with source s and sink t and f a preflow function in G. A height function $h : V \to \mathbf{N}$ is a function such that:

$$h(s) = |V|,\ h(t) = 0 \text{ and}$$
$$h(u) \leq h(v) + 1 \text{ for every residual edge } (u, v)$$

Note that the definition of h implies that if $h(u) > h(v) + 1$ then (u, v) is not an edge in the residual graph G_f.

Operation Push

PUSH(u, v) is an operation that is activated if u is an overflowing vertex (i.e., $e(u) > 0$) and is such that $c_f(u, v) = 0$ and $h(u) = h(v)+1$. PUSH(u, v) updates the preflow f in a flow network $G = (V, E)$. The amount of flow that can be pushed from u to v is stored in $d_f(u, v)$. The excess flow stored at vertex u is maintained as $e(u)$ and its current height is $h(u)$.

Note that at any given time, we do not want to push a quantity of flow that will cause $e(u)$ to become negative, or that will exceed the capacity of the edge. So, $d_f(u, v) = \min\{e(u), c_f(u, v)\}$ units of flow will be allowed to be pushed.

```
PUSH(u, v)
begin
//applicability: u is overflowing, (i.e. e(u) > 0)
//c_f(u, v) > 0 and h(u) = h(v) + 1
//action: PUSH d_f(u, v) = min{e(u), c_f(u, v)}
//units of flow from u to v
  d_f(u, v) ← min{e(u), c_f(u, v)}              amount to push
//move the flow from u to v
//by updating f
  f(u, v) ← f(u, v) + d_f(u, v)
  f(v, u) ← −f(u, v)
//and by updating e
  e(u) ← e(u) − d_f(u, v)
  e(v) ← e(v) + d_f(u, v)
end //of PUSH(u, v)
```

Note that PUSH(u, v) is activated only when $h(u) = h(v) + 1$. If after a PUSH(u, v), edge (u, v) becomes a saturated edge, i.e. $c_f(u, v) = 0$ after the push, then the push is called a saturating push, otherwise it is called a nonsaturating push. Note that saturated edges do not show up in the residual network.

Operation Lift

We lift an overflowing vertex u for every vertex v for which there is residual capacity from u to v. Flow cannot be pushed from u to v because v is not downhill from u. In other words, LIFT(u) is activated if u is overflowing, and if

$$c_f(u, v) > 0 \Rightarrow h(u) \leq h(v) \text{ for all vertices } v$$

```
LIFT(u)
begin
//applicability: u is overflowing (e(u) > 0)
//and for all v ∈ V, we have
//(u, v) edge in G_f ⇒ h(u) ≤ h(v)
//action: increase the height of u
  h(u) ← 1 + min{h(v)|(u, v) is an edge in G_f}
end //of PUSH(u)
```

After the execution of LIFT(u), we say that u is lifted. Recall that neither s nor t can be lifted. We note that $\{h(v)|(u, v)$ is an edge in $G_f\}$ is not an empty set since u is an overflowing vertex (i.e. $c_f(u, v) > 0$).

1.6.3 The preflow-push procedure

The preflow-push procedure basically consists of two parts: initialization and a push or lift loop.

Initially, each edge leaving s is filled to capacity, while all other edges carry no flow at all. In other words:

$$f(u,v) = \begin{cases} c(u,v) & \text{if } u = s \\ -c(u,v) & \text{if } v = s \\ 0 & \text{otherwise} \end{cases}$$

As for the height function, it initially assigns the following values:

$$h(u) = \begin{cases} |V| & \text{if } u = s \\ 0 & \text{otherwise} \end{cases}$$

And for each vertex v adjacent to s we have $e(v) = c(s,v)$, otherwise $e(u) = 0$ for all other vertices.

Recall the definition of a height function, $h : V \to \mathbf{N}$ such that

$$\begin{aligned} h(s) &= |V| \\ h(t) &= 0 \\ h(u) &\leq h(v) + 1 \text{ for every residual edge } (u,v) \end{aligned}$$

In our initialization, we have $h(s) = |V|$, $h(t) = 0$ and the edges (u,v) that violate $h(u) \leq h(v)+1$ (that is for which $h(u) > h(v)+1$), are the one for which $u = s$ and those are saturated ($f(u,v) = c(u,v)$) and so are not present in the residual graph G_f.

In other words, these edges are not residual edges, and so the initial function is indeed a height function.

After the initialization, the procedure repeatedly performs the PUSH or LIFT procedures seen in the previous sections. Note that as long as an overflowing vertex exists, at least one of the two operations applies. This is true since for any residual edge (u,v), we have $h(u) \leq h(v)+1$ (since h is a height function). Now if PUSH(u,v) is not applicable, then we must have $h(u) < h(v) + 1$ for all residual edges (u,v). In other words, $h(u) \leq h(v)$ and thus a LIFT operation can be applied to u.

```
PREFLOW-PUSH(G)
begin
//Initialization phase
  for each vertex u ∈ V
    h(u) ← 0
    e(u) ← 0
  for each edge (u, v) ∈ E
    f(u, v) ← 0
    f(v, u) ← 0
  h(s) ← |V|                          height of s is the only nonzero
                                      valued height in the beginning
//saturate edges leaving the source
  for each vertex u ∈ Adj[s]
    f(s, u) ← c(s, u)
    f(u, s) ← −c(s, u)
    e(u) ← c(s, u)
//PUSH or LIFT phase
  while there exists an applicable PUSH
        or LIFT operation
    do select an applicable PUSH
        or LIFT operation and perform it
  return f //f is the maximum flow of G
end
```

Remarks:

1. Only when the algorithm terminates does the preflow become a flow, and then it is a maximum flow. For that reason it is labeled preflow. Karzanov was the first to use the idea of preflow.

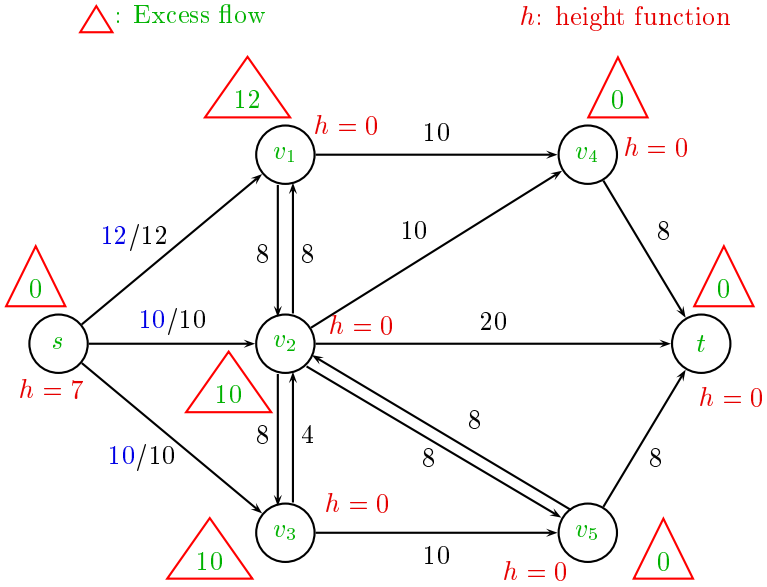
2. Note that the condition of "overflowing" is present for both operations: push and lift. Thus, for the implementation of the preflow-push algorithm, we could maintain a list (or set) of active vertices, where a vertex u is active if it is overflowing (i.e. $e(u) > 0$).

The **while**-loop of the preflow-push repeatedly performs the following steps, in any order, until there are no active vertices:

push: Select any active vertex u. Select any residual edge (u, v) with $h(u) = h(v) + 1$. Send $d_f(u, v) = \min\{e(u), c_f(u, v)\}$ units of flow from u to v. The push has the effect of increasing $f(u, v)$ and $e(v)$ by $d_f(u, v)$ and decreasing $f(v, u)$ and $e(u)$ by $d_f(u, v)$. The push is saturating if $d_f(u, v) = c_f(u, v)$ and nonsaturating otherwise. After performing push, if $e(u)$ is still positive, then of course u remains an active vertex.

lift: Select any active vertex u for which $(u, v) \in E_f \Rightarrow h(u) \leq h(v)$. Replace $h(u)$ by $1 + \min\{h(v)|(u, v) \in E_f\}$.

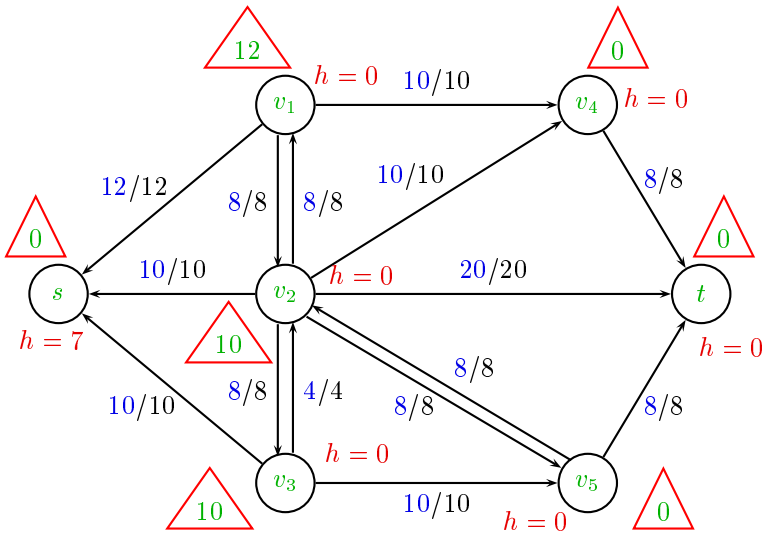
EXAMPLE: Consider the AAC example once more.



Network after the initialization phase

The active nodes are: v_1, v_2, v_3. At this point we have only one possible operation: lift. To be able to push we should have $h(u) = h(v) + 1$, where u is an active vertex, and $(u, v) \in E_f$. But $h = 0\ \forall u \in V \backslash \{s\}$. So no pushes are possible.

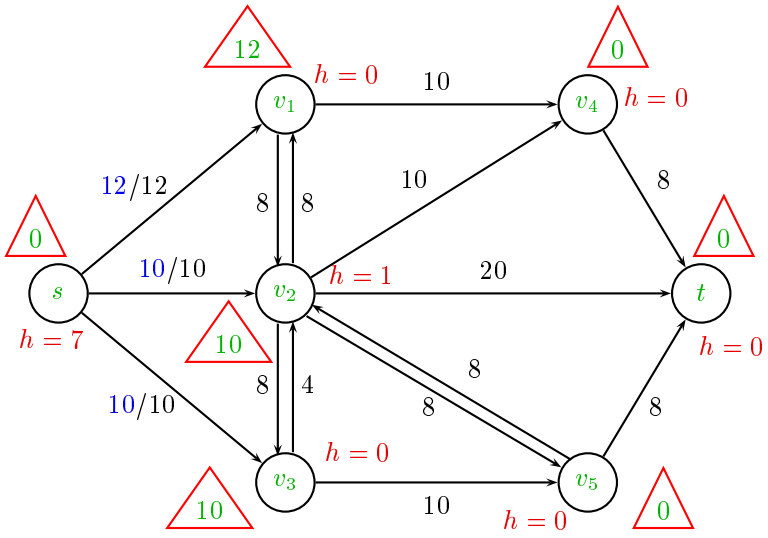
The residual network G_f induced by the initial preflow in G.



Again, notice that the nodes that have an excess flow: v_1, v_2, and v_3 are such that $h(u) = h(v) = 0$ for $u \in \{v_1, v_2, v_3\}$ and $(u, v) \in E_f$ and $v \neq s$.

Suppose that preflow-push chooses v_2 to lift. LIFT(v_2) will increase the height of v_2 by $1 + \min\{h(v) | (v_2, v) \in E_f\} = 1 + 0 = 1$. So the new height of v_2 will be 1.

The network after performing LIFT(v_2) now has $h(v_2) = 1$ (instead of $h(v_2) = 0$).



G after the lifting of v_2

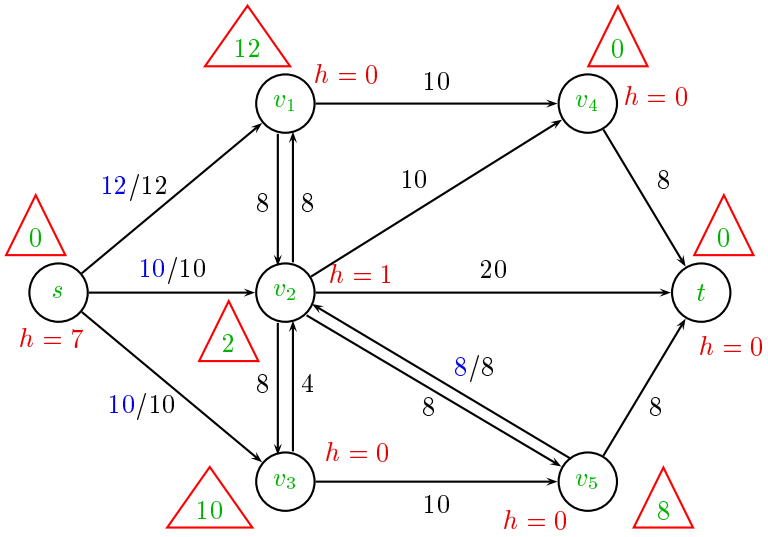
At this point we still have the same three active nodes: v_1, v_2 and v_3 since $e(u) > 0$ for $u \in \{v_1, v_2, v_3\}$. Preflow-push can choose between three options:

a) LIFT(v_1)

b) LIFT(v_3) or

c) push the excess flow found in vertex v_2.

Assume that preflow-push chooses option c), i.e. push the excess flow of v_2.



Network after performing PUSH(v_2, v_5)

Moreover, suppose that the excess flow is pushed to v_5. Note that v_1, v_3, v_4 and t would have been other possible recipients of the excess flow of v_2 (since $h(v_2) = h(v) + 1$ and $c_f(v_2, v) > 0$ for $v \in \{v_1, v_3, v_4, t\}$. PUSH($v_2, v_5$) will send $c_f(v_2, v_5) = \min\{e(v_2), c_f(v_2, v_5)\} = 8$ units of flow to v_5. So $f(v_2, v_5) = 8$, $f(v_5, v_2) = -8$ and $e(v_2) = 10 - 8 = 2$, $e(v_5) = 0 + 8 = 8$.

Note that v_2 remains an active vertex even after PUSH(v_2, v_5) since $e(v_2) = 2$. So now the set of active vertices comprises v_1, v_2, v_3 and v_5.

At this point preflow-push can choose between pushing the excess flow of v_2 or choosing a vertex from $\{v_1, v_3, v_5\}$ to perform a lift operation.

And so preflow-push goes on pushing or lifting until there are no active vertices (i.e., until no vertex is overflowing).

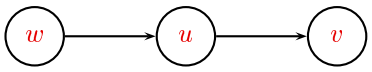
1.6.4 Correctness of the preflow-push algorithm

We want to show that the preflow-push algorithm terminates and that when it does, the flow it produces is the maximum possible flow. We start by showing that vertex heights never decrease. Heights are modified only by the lift operation which "raises" the value of the height of the vertex. For vertex u to be lifted, it must be such that $h(u) \leq h(v)$ for all $(u, v) \in E_f$. In other words $h(u) < 1 + \min\{h(v)|(u, v) \in E_f\}$ before the lift. And after the lift of u we have $h(u) = 1 + \min\{h(v)|(u, v) \in E_f\}$. So vertex heights

1. never decrease
2. increase by at least 1

Another important observation about the height function is to realize that the height function will remain a height function (recall the conditions $h(s) = |V|, h(t) = 0$, and $h(u) \leq h(v) + 1$ for every $(u, v) \in E_f$) after the execution of the preflow-push algorithm. We already saw that the initialization phase does indeed define a height function on the vertices of the network. Let us show that the operations of lift and push do indeed leave h a height function.

Operation lift: For a residual edge (u, v), we shall have $h(u) \leq h(v) + 1$ after the execution of LIFT(u).



As for residual edge (w, u), we have $h(w) \leq h(u)+1$ before and $h(w) < h(u)+1$ after the execution of LIFT(u). In other words, after the execution of LIFT(u), we'll have $h(u) \leq h(v) + 1$ for every $(u, v) \in E_f$ and so h is still a height function.

Operation push: Consider PUSH(u, v). We have 2 possibilities:

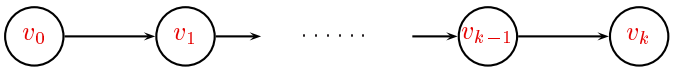
a) It may add (v, u) to E_f (if in our example, we perform a PUSH(v_2, t) instead of PUSH(v_2, v_5), then we add (v_2, t) to E_f with $f(v_2, t) = 10$). In this case we have $h(v) = h(u) - 1$ and so h remains a height function (in our example, we have: $h(v_2) = 1$ and $h(t) = 0$, i.e. $h(t) = h(v_2) - 1$).

b) It may remove (u, v) from the residual network. (That is indeed the case after the execution of PUSH(v_2, v_5) in our example. Now (v_2, v_5) is no longer in the residual graph.) But the removal of (u, v) from G_f also removes the corresponding constraint, and so h will still be a height function after the execution of PUSH(u, v). □

We prove one last property before showing the correctness of the preflow-push algorithm.

Theorem 1.6.1 *$G = (V, E)$ is a flow network with source s and sink t, f is a preflow in G and h is a height function on V. There is no path from the source s to the sink t in the residual network G_f.*

Proof: The proof is by contradiction.



Suppose there is a path $p =< v_0, v_1, \ldots, v_k >$ from $s = v_0$ to $t = v_k$ in the residual graph G_f. W.l.o.g. (without loss of generality), p is a simple path (does not include cycles) so that $k < |V|$. So for $i = 0, 1, \ldots, k-1$, we have edge (v_i, v_{i+1}) in the residual graph G_f. h is a height function, so we must have $h(v_i) \leq h(v_{i+1}) + 1$ for $i = 0, 1, 2, \ldots, k-1$. Thus we have:

$$\begin{aligned} h(s) &\leq h(v_1) + 1 \\ &\leq (h(v_2) + 1) + 1 = h(v_2) + 2 \\ &\leq h(v_3) + 1 + 2 \\ &\vdots \\ &\leq h(t) + k \end{aligned}$$

But $h(t) = 0$ since h is a height function. So, $h(s) \leq k$ and $k < |V|$ i.e. $h(s) < |V|$ which contradicts the fact that $h(s) = |V|$ (since h is a height function). So no such path exists from s to t in G_f. □

Theorem 1.6.2 *When the preflow-push algorithm terminates when run on a flow network $G = (V, E)$ with source s and sink t, then the preflow f it computes is indeed a maximum flow for G.*

Proof: When the preflow-push algorithm terminates, then the condition in the **while**-loop is not satisfied and thus there are no active vertices. In other words, no vertex in $V \backslash \{s, t\}$ has an excess flow (i.e., there are no overflowing vertices). Therefore, f is a flow for the flow network G. It is a maximum flow because we know that f is a maximum flow if and only if the residual network G_f contains no path from s to t (max-flow min-cut theorem) and from the previous theorem we know that since h is a height function, there are no paths from s to t in the residual graph.

So when the preflow-push algorithm terminates, it delivers a maximum flow from s to t. □

1.6.5 Complexity of the preflow-push algorithm

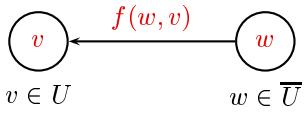
We shall need the following theorem to find the running time function of the preflow-push algorithm.

Theorem 1.6.3 *$G = (V, E)$ is a flow network with source s and sink t, and f is a preflow in G. For any overflowing vertex u in G, there exists a simple path from u to s in the residual network G_f.*

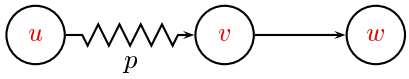
Proof: Suppose that $u \in V$ is overflowing (i.e. $e(u) > 0$). Let

$$U = \{v | \exists \text{ a simple path from } u \text{ to } v \text{ in } G_f\} .$$

If we can show that $s \in U$, then we are done. The proof is by contradiction. Suppose that $s \notin U$. Let $\overline{U} = V - U$.
Claim: $f(w, v) \leq 0$ for any $w \in \overline{U}$ and any $v \in U$.



Proof of Claim: *(by contradiction)* If $f(w,v) > 0$, then $f(v,w) < 0$ but $c_f(v,w) = c(v,w) - f(v,w)$ (by definition), and so if $f(v,w) < 0$, we will have $c_f(v,w) > 0$, which means that there exists an edge (v,w) in the residual graph G_f.



And so in the residual graph G_f we have:

- a path p from u to v (since $v \in \overline{U}$)
- an edge from v to w

And therefore we have a path from u to w which contradicts the fact that $w \in \overline{U}$. So $f(w,v) \leq 0$.

Back to the main theorem. Since $f(w,v) \leq 0$ for every $w \in \overline{U}$ and every $v \in U$, we have $f(\overline{U}, U) \leq 0$.

$$\begin{aligned} e(U) &= f(V,U) \quad \text{by definition of excess flow} \\ &= f(\overline{U} \cup U, U) \\ &= f(\overline{U},U) + f(U,U) \quad \text{by a property already studied} \end{aligned}$$

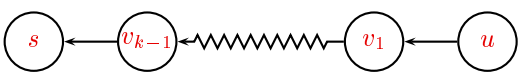
But $f(U,U) = 0$ and $f(\overline{U},U) \leq 0$, so $e(U) \leq 0$.

By definition, an excess flow for any vertex in $V \backslash \{s\}$ is nonnegative, $U \subseteq V \backslash \{s\}$ and therefore we have $e(v) = 0$ for all vertices $v \in U$ (since $e(U) \leq 0$). In particular, $e(U) = 0$ which contradicts the fact that u is an overflowing vertex. So for any overflowing u in G, there exists a simple path from u to s in the residual network G_f. □

We next find an upper bound on the heights of vertices.

Theorem 1.6.4 *At any time during the execution of the preflow-push algorithm, we have* $h(u) \leq 2|V| - 1$ *for all* $u \in V$.

Proof: Vertices s and t are never overflowing (by definition) and so the values of their heights never change: $h(s) = |V|$ and $h(t) = 0$. Vertices have their heights altered (increased) only when they are overflowing. For any overflowing vertex $u \in V \backslash \{s,t\}$, we have a simple path $p =< v_0, v_1, \ldots, v_k >$ with $u = v_0$, $s = v_k$ in the residual network G_f (by the previous theorem).



Note that $k \leq |V| - 1$ since p is simple. Moreover,

$$(v_i, v_{i+1}) \in E_f \Rightarrow h(v_i) \leq h(v_{i+1}) + 1$$

So

$$\begin{aligned} h(u) &\leq h(v_1) + 1 \\ &\leq (h(v_2) + 1) + 1 = h(v_2) + 2 \\ &\vdots \\ &\leq h(s) + k \end{aligned}$$

$h(u) \leq h(s) + k$, $h(s) = |V|$ and $k \leq |V| - 1$, so

$$h(u) \leq |V| + (|V| - 1) = 2|V| - 1$$

□

As an immediate result of the previous theorem, we can show that the number of lift operations is at most $2|V|-1$ per vertex and at most $(2|V|-1)(|V|-2) < 2|V|^2$ overall. Note that any vertex in $V \backslash \{s, t\}$ can be lifted, and when it is, its value grows from 0 (initially $h(u) = 0$) to at most $2|V| - 1$ (by the previous theorem). So each vertex is lifted at most $2|V| - 1$ times, and the total number of lift operations is at most $(2|V| - 1)(|V| - 2) < 2|V|^2$.

That was the bound on lift operations. To bound the push operations, we consider the saturating pushes and the nonsaturating pushes separately.

Theorem 1.6.5 *During the execution of the preflow-push algorithm, the number of saturating pushes is at most* $2|V||E|$.

Proof: Recall that PUSH(u, v) is a saturating push if edge (u, v) becomes saturated ($c_f(u, v) = 0$) after the push, and that saturated edges do not appear in the residual network. When (u, v) is saturated, $h(u) = h(v) + 1$, and $(u, v) \notin E_f$.

For (u, v) to reappear in the residual network, we must send flow in (v, u) and thus have $h(v)$ increase by at least 2. Likewise, $h(u)$ must increase by at least 2 between saturating pushes from v to u.

Consider the sequence A of integers given by $h(u) + h(v)$ for each saturating push that occurs between vertices u and v.

We want to upper bound the length of sequence A. Note that after the first push (i.e. PUSH(u, v) or PUSH(v, u)), we will have $h(u) + h(v) \geq 1$. In other words, the first integer in sequence A is at least 1. When the last push occurs, we will have:

$$\begin{aligned} h(u) + h(v) &\leq (2|V| - 1) + (2|V| - 2) \text{ recall } h(u) \leq 2|V| - 1 \\ &= 4|V| - 3 \end{aligned}$$

So the last integer in sequence A is at most $4|V| - 3$. Since heights must increase by at least 2 between saturating pushes, the number of integers in sequence A is at most

$$\frac{(4|V| - 3) - 1}{2} + 1 = 2|V| - 1.$$

Note that 1 is added on the left hand side of the equality to make sure that the first and last numbers of sequence A are taken into consideration. The total number of saturating pushes between vertices u and v is at most $2|V| - 1$.

So the total number of saturating pushes is at most $(2|V| - 1)|E| < 2|V||E|$. □

We now would like to bound the nonsaturating pushes. We use a potential function to achieve the goal.

Theorem 1.6.6 *During the execution of the preflow-push algorithm, the number of nonsaturating pushes is at most* $4|V|^2(|V| + |E|)$.

Proof: Define a potential function $\Phi = \sum_{v \in X} h(v)$ where $X \subseteq V$ is the set of overflowing vertices. Initially, $\Phi = 0$, since we have no overflowing vertices. Recall that at each iteration of the **while**-loop, we have to perform one of the following operations:

a) a lift

b) a saturating push

c) a nonsaturating push

Let $\Phi(k)$ denote the potential value at step k of the algorithm. If at the k^{th} step, the algorithm picks $u \in X$ and performs a:

a) lift, then $\Phi(k) - \Phi(k-1) \leq 2|V|$, since only $u \in X$ is lifted (heights of other elements in X do not change) and u cannot be lifted by more than its maximum possible height.

b) saturating push, PUSH(u, v), then $\Phi(k) - \Phi(k-1) \leq 2|V|$, since again no heights change and only vertex v, whose height is at most $2|V|$, can possibly have a positive excess flow.

c) nonsaturating push, PUSH(u, v), then $\Phi(k) - \Phi(k-1) \leq -1$, since u is not an overflowing vertex after the push, v now joins X (even it wasn't an overflowing vertex at step $k-1$), and $h(v) - h(u) = -1$

If K denotes the last step of the algorithm, then

$$\begin{aligned}
\Phi(K) - \Phi(0) &= \sum_{k=1}^{K} \Phi(k) - \Phi(k-1) \\
&\leq (\text{max number of lifts})(2|V|) \\
&\quad +(\text{max number of saturating pushes})(2|V|) \\
&= (2|V|^2)(2|V|) + (2|V||E|)(2|V|) \text{ (from previous theorems)} \\
&= 4|V|^2(|V| + |E|)
\end{aligned}$$

So $\Phi(K) - \Phi(0) \leq 4|V|^2(|V| + |E|)$. Since $\Phi > 0$, the total decrease (due to $\Phi(k) - \Phi(k-1) \leq -1$ when a nonsaturating push is performed), and therefore the total number of nonsaturating pushes, is at most $4|V|^2(|V| + |E|)$. □

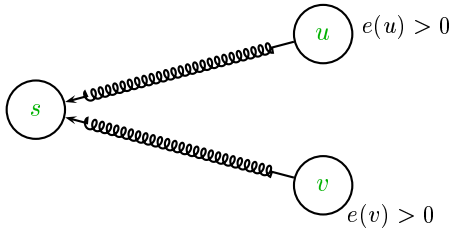
Since $|E| > |V|$, we have really shown the following theorem:

Theorem 1.6.7 *During the execution of the preflow-push algorithm on any flow network $G = (V, E)$, the number of basic operations is $O(|V|^2|E|)$.* □

Complexity of preflow-push algorithm – Summary

1. Existence of path from overflowing node to s in G_s.

   If u is overflowing $\Rightarrow$ $\exists$ a simple path from u to s in G_f.



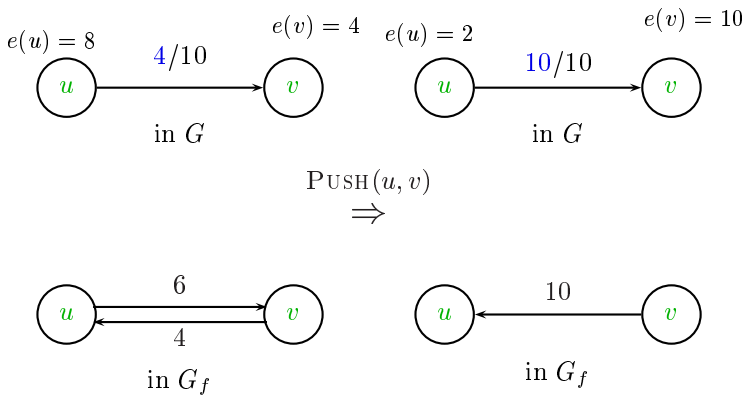
2. Upper bound on height of an vertex: $h(u) \leq 2|V| - 1$ for all $u \in V$.

3. Number of lift operations:

$$< 2|V| - 1 \text{ per vertex}$$
$$< 2|V|^2 \text{ overall}$$

4. Number of saturating pushes $< 2|V||E|$

   EXAMPLE:



$$A = \{a_1, a_2, \ldots, a_q\} \text{ find a bound on } q$$

   a_i is the i^{th} saturated push. Because after the 1^{st} push $h(u) + h(v) \geq 1$, we know that $a_1 \geq 1$. We also know that $a_q \leq 4|V| - 3$. Furthermore, $a_{i+1} - a_i \geq 2$ holds.

   EXAMPLE: Consider the sequence

$$1, 3, 5, 7, 9, 11, 13, 15, 17, 19$$

   Count the number of integers in this sequence:

$$\frac{\text{last} \ - \ \text{first}}{2} + 1 \ \Rightarrow \ \frac{19 - 1}{2} + 1 = 10$$

5. Number of nonsaturating pushes $< 4|V|^2(|V| + |E|)$. Use $\Phi = \sum_{v \in X} h(v)$, where X is the set of overflowing vertices. We denote by $\Phi(k)$ the potential value of the algorithm at step k.

$$\Phi(k) - \Phi(k-1) \leq 2|V|$$

   and thus $\Phi(K) - \Phi(0) \leq 4|V|^2(|V| + |E|)$ where K denotes the last step of the algorithm.

1.6.6 Specific implementations of the preflow-push algorithm

The preflow-push algorithm is flexible. By specifying different rules for selecting active nodes for the push and lift operations one can derive many different algorithms, each with different worst-case complexity than the preflow-push algorithm we have just studied.

The bottleneck operation in the preflow-push algorithm is the number of nonsaturating pushes. Different rules for examining active nodes can produce meaningful reductions in the number of nonsaturating pushes.

1.6.7 FiFo preflow-push algorithm

As the title suggests, the algorithm examines the active nodes in the first-in, first-out (FIFO) order [AMB93]. It can be shown, that this algorithm runs in $O(|V|^3)$ time. At any given iteration, we saw that the preflow-push algorithm selects an active node u and performs

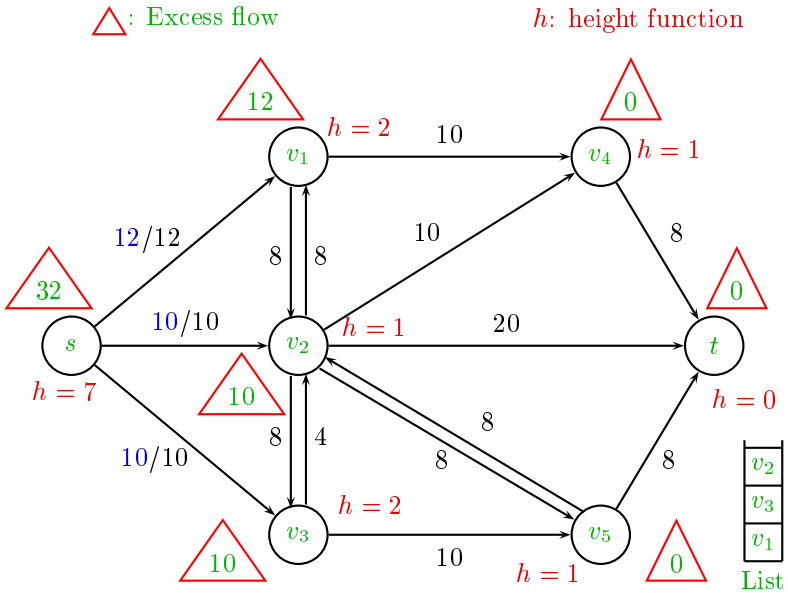
- a saturating push, PUSH(u, v), or
- a nonsaturating push, PUSH(u, w), or
- a lift operation, LIFT(u).

If the algorithm performs a saturating push, it is possible that u still remains active. The algorithm might select u again or might select another active node instead. In our FIFO preflow-push algorithm we want to force the selection of u again and keep pushing flow from u util either $e(u) = 0$ or the algorithm performs a LIFT(u) operation. Consequently, the algorithm might perform several saturating pushes followed either by a nonsaturating push or a lift operation. This sequence of operations is known as a node examination or we say that an overflowing vertex u is discharged by pushing all of its excess flow through admissible edges (u, v) (an edge (u, v) where $c_f(u, v) > 0$ and $h(u) = h(v) + 1$) to neighboring vertices, lifting u as necessary to cause edges leaving u to belong admissible.

The FIFO preflow-push algorithm examines active nodes in the FIFO order, by maintaining a set list as a queue. It selects a node u from the front of the list, performs pushes from this node, and adds newly active nodes to the rear of the list. The algorithm repeatedly considers u (i.e. examines u) until either it becomes inactive or it is lifted (in which case it is added to the end of the queue). As before, the algorithm terminates when the queue of active nodes is empty.

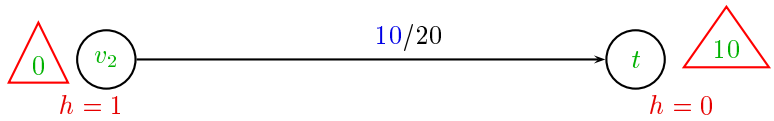
To cut down on the number of lifts that the algorithm has to perform in the early stages of the run, we do not initialize the height values to zero. Instead, we perform a backward breadth-first search starting at the sink t. So the sink will have a height of 0, vertices that share an edge with t will be of height one, vertices two hops from t have 2 as initial height, etc. As for the source, we keep the initial height value to be $|V|$. But the source's excess flow is initialized to $e(s) = \sum_{v \in V} c(s, v)$. We illustrate the FIFO preflow-push algorithm using the AAC example.
EXAMPLE: The initialization phase creates a network with height $h(t) = 0$, $h(v_2) = h(v_4) = h(v_5) = 1$, $h(v_1) = h(v_3) = 2$ and $h(s) = |V| = 7$.

As is the case with the preflow-push algorithm of the previous section, the algorithm then saturates edges leaving the source. Thus we'll have an excess flow of 12 crates at v_1 and an excess of 10 crates of olives at each of the vertices v_2 and v_3. Suppose that the queue of active nodes at this stage is LIST $= \{v_2, v_3, v_1\}$. The algorithm next removes vertex v_2 from the queue and "examines it.

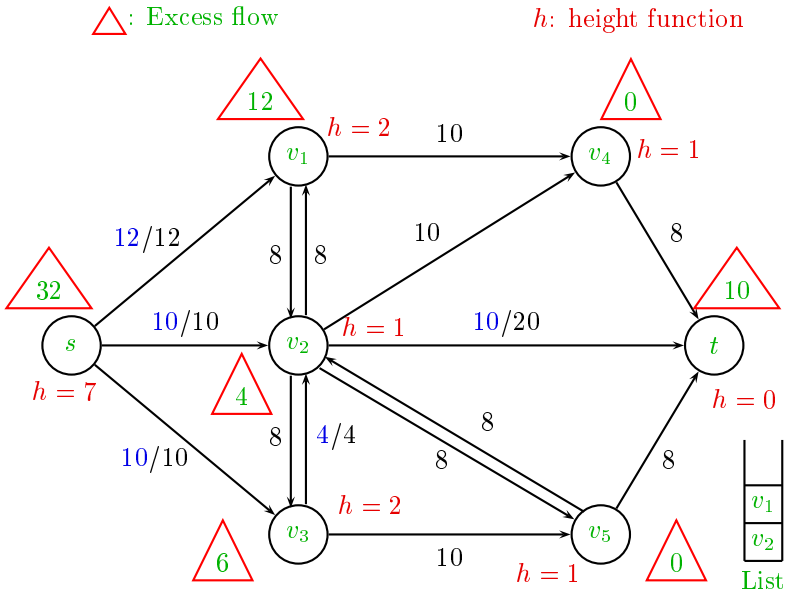


Snapshot of network after saturating the edges leaving s

At this stage, the algorithm performs a nonsaturating push, PUSH(v_2, t), and v_2 gets rid of its excess overflow.



Note that v_2 is not an active node after performing PUSH(v_2, t). So the algorithm next examines v_3 (the current node in LIST). At this stage, the algorithm could perform a saturating push, PUSH(v_3, v_2) of 4 units or a saturating push, PUSH(v_3, v_5) of 10 units. Suppose it does PUSH(v_3, v_2).



So now v_2 has an excess flow of 4 units and is therefore put in the queue. As mentioned earlier, the algorithm stays with v_3 since $e(v_3) = 6 > 0$, and performs PUSH(v_3, v_5) [a nonsaturating push of 6 units]. So now v_5 is active and enters the queue. v_3 is not active anymore and the algorithm then considers vertex v_1. And so on.

Theorem 1.6.8 *The* FIFO *preflow-push algorithm runs in* $O(|V|^3)$ *time.*

Proof: To analyze the worst-case complexity of the FIFO preflow-push algorithm, we partition the total number of node examinations into several phases.

Phase one: Node examination of the nodes that become active because of the saturating of the edges that leave s (i.e. the last step in the algorithm before the **while**-loop). In our example, phase one consists in examining nodes v_1, v_2, v_3.

Phase two: Node examination of all the nodes that are in the queue after the algorithm has examined the nodes of phase one. In our example, phase two consists in examining nodes v_2, v_5, v_4.

Similarly, *phase three* consists of the node examination of all the nodes that are in the queue after the algorithm has examined the nodes in the second phase. And so on.

Notice that the algorithm examined a node at most once during a phase.

We shall show that the FIFO preflow-push algorithm performs at most $2|V|^2+|V|$ phases. Each phase examines any node at most once and each node examination performs at most **one** nonsaturating push. So if we can proof that we have at most $2|V|^2 + |V|$ phases, then we'll have at most $|V|(2|V|^2 + |V|)$ nonsaturating pushes. But recall that the number of nonsaturating pushes is the bottleneck of the preflow-push algorithm (and is also the bottleneck of the FIFO preflow-push algorithm). So this would imply that the FIFO preflow-push algorithm runs in $O(|V|^3)$.

Claim: FIFO preflow-push performs at most $2|V|^2 + |V|$ phases
Proof of Claim: We consider the total change in the potential function $\Phi = \max\{h(u)|u \text{ is active }\}$ over an entire phase.

Once again, we would like to compute the difference between the initial and final values of the potential function during a phase. Note that the potential function here is slightly different than the one we use in the previous section. Back to the proof of the claim. We consider two cases:

Case one: FIFO preflow-push performs at least one lift operation during a phase. We already saw in the previous section that $h(u) \leq 2|V| - 1$ for any $v \in V$, and that the total number of lift operations is at most $2|V|^2$. Since Φ cannot increase more than the maximum increase produced by a lift operation, the total increase in Φ over all the phases is at most $2|V|^2$.

Case two: FIFO preflow-push performs no lift operation during a phase. As was seen in the previous section, Φ decreases by at least 1 unit, since the excess of every node that was active at the beginning of phase moves to nodes with smaller weights.

So by combining cases one and two, and by realizing that the initial value of Φ could be at most $|V|$, we have that the total number of phases is at most $2|V|^2 + |V|$.

So this proves the claim that FIFO preflow-push performs at most $2|V|^2 + |V|$ phases. By the remarks preceding the claim, we have proven that the FIFO preflow-push algorithm runs in $O(|V|^3)$ time. □

Other possible implementations of the preflow-push algorithm include the highest-label preflow algorithm that always pushes flow from an active node with the highest

distance value, and the excess scaling algorithm. The highest-label preflow push algorithm runs in $O(|V|^2\sqrt{|E|})$ and is considered to be the most efficient maximum flow algorithm in practice.

1.7 Applications of the maximum flow problem

1.7.1 Scheduling on uniform parallel machines

We are interested in scheduling a set of J jobs on M uniform parallel machines [AMB93]. Each job $j \in J$ has:

p_j: number of machine days required to complete the job

r_j: release date, i.e. the beginning of the day when job j becomes available for processing

d_j: due date, that is the beginning of the day by which the job must be completed

Note that $r_j + pj \leq d_j$.

We make the following assumptions:

1. A machine can work on only one job at a time
2. Each job can be processed by at most one machine at a time
3. Preemption is allowed. In other words, we can interrupt a job and process it on different machines on different days.

The scheduling problem consists in determining a feasible schedule that computes all jobs before their due dates or to show that no such schedule exists.

Note that scheduling problems of this kind arise in batch processing systems involving batches with a large number of units.

The feasible scheduling problem as defined here can be used by more general scheduling problems, such as

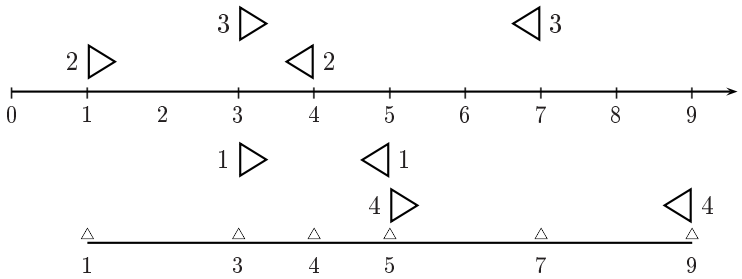
- the maximum lateness (tardy) problem
- the (weighted) minimum completion time problem
- the (weighted) maximum utilization problem

The feasible scheduling problem could be formulated as a maximum flow problem. We show the transformation of this problem into a maximum flow problem by considering the following example.

EXAMPLE: Suppose we have $M = 3$ machines and 4 Jobs as given in the table:

Job	j	1	2	3	4
Processing time	p_j	1.5	1.25	2.1	3.6
Release time	r_j	3	1	3	5
due time	d_j	5	4	7	9

Let us rank all the release and due date, r_j and d_j for all j in ascending order and determine mutually disjoint intervals of dates between consecutive "milestones".



By ordering r_j and d_j in ascending order, we get:

$$1, 3, 4, 5, 7, 9$$

and so we get 5 mutually disjoint intervals of dates (or we can allow the intervals to share end points):

$$[1, 3], [3, 4], [4, 5], [5, 7], [7, 9]$$

Note that within each interval, the set of jobs that are available (i.e., jobs that have been released but not due yet) does not change.

Another way of seeing the 5 intervals is by considering the graph above, where j denotes the release time of j and j denotes the due date of job j.

Note that the maximum number of disjoint intervals we can have is $2|J| - 1$ (we have 2 entities for each job: due and release dates) where $|J|$ is the number of jobs.

Let $T_{k,l}$ denote the interval that starts at the beginning of day k and ends at the beginning of day $l + 1$. In our example, the five intervals are:

$$T_{1,2} \quad T_{3,3} \quad T_{4,4} \quad T_{5,6} \quad \text{and} \quad T_{7,8}$$

Notice that within each interval $T_{k,l}$ we can process all jobs with

a) $r_j \leq k$ and

b) $d_j \geq l + 1$

in the interval. For instance, in the interval 3 to 4, i.e. $T_{3,3}$ we can process the following jobs:

- 2 since $r_2 \leq 3$ and $d_2 = 4 \geq 3 + 1$
- 1 since $r_1 \leq 3$ and $d_1 = 5 \geq 3 + 1$ and
- 3 since $r_3 \leq 3$ and $d_3 = 7 \geq 3 + 1$

To cast this problem as a maximum flow problem, we create 4 levels of vertices:

Level one: The source node s.

Level two: Nodes corresponding to the $|J|$ jobs (one vertex per job).

Level three: Interval vertices; one vertex per interval.

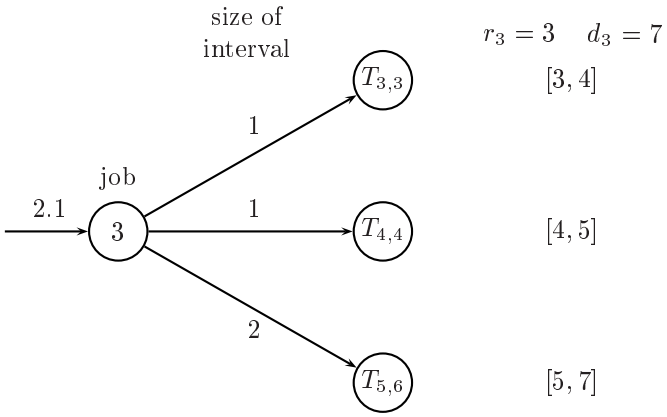
Level four: The sink node t.

As for the edges and capacities:

- Connect the source node to every job vertex in level two with an edge with capacity p_j. This means that we need to assign p_j days of machine time to job j.
- Connect each job vertex j to every interval vertex $T_{k,l}$ if

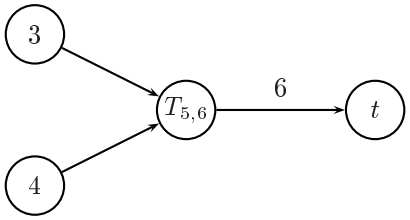
$$r_j \leq k \text{ and } d_j \geq l+1$$

by an edge with capacity $l-k+1$ which represents the maximum number of machine days we can allot to job j on the days from k to l. For instance:



Job 3 is ready for processing at $r_3 = 3$ and has a due date of $d_3 = 7$. Its $p_3 = 2.1$ of processing time could be scheduled during intervals $T_{3,3}$, $T_{4,4}$ and/or $T_{5,6}$.

- Connect each interval node $T_{k,l}$ to t by an edge with capacity $(l-k+1)M$, representing the total number of machine days available on the days from k to l. For instance:

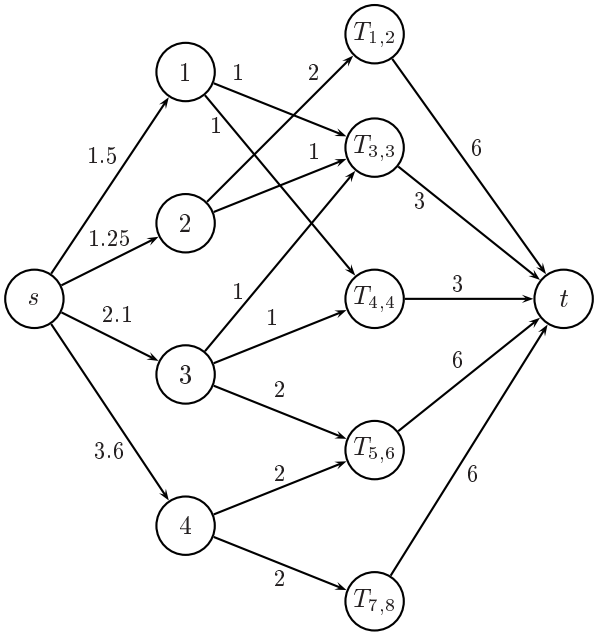


The size of the interval is $6-5+1=2$. But this represents the number of days per machine. So we have 6 machine days (3×2) to service the processing demands of jobs 3 and 4 that arrive at node $T_{5,6}$. □

So now we solve the problem and find a maximum flow.

$$\left.\begin{array}{c}\text{The scheduling problem} \\ \text{has a feasible solution}\end{array}\right\} \Leftrightarrow \begin{array}{c}\text{The maximum flow} \\ \text{is equal to } \sum_{j \in J} p_j\end{array}$$

□



□

1.7.2 Distributed computing on a two-processor computer

A computer system has two (not necessarily identical) processors. We wish to execute a very large program that consists of several modules (subroutines) that will need to interact with each other during the execution of the program [AMB93]. The cost of executing a module on a processor is known in advance. Let

α_i: cost of computation of module i on processor 1

β_i: cost of computation of module i on processor 2

In general $\alpha_i \neq \beta_i$ because the memory, control, speed and arithmetic capabilities of the two processors are not the same. Let

c_{ij}: interprocess communication cost when modules i and j are assigned to different processors

Note that $c : ij$ is in general quite high and that if instead, modules i and j are assigned to the same processor, we do not have any cost.

We wish to allocate modules of the program on the two processors so that we minimize the total cost of processing and interprocess communication. This problem can be formulated as a minimum cut problem on undirected network. We show the formulation by considering an example.

EXAMPLE: Suppose that a program can be broken into 4 modules and that the costs of computation of the different modules on processors one and two are as given as in table 1.1.

Moreover, the interprocess communication costs, c_{ij}, of assigning module i and j on different processors are given in the symmetric table 1.2 below. So for instance,

$$c_{3,2} = 6$$

To cast this problem as a minimum cut problem on an undirected network, we create three levels of vertices:

i	1	2	3	4	Modules
α_i	6	5	10	4	Processor one
β_i	4	10	3	8	Processor two

Table 1.1: "In-house" costs

	1	2	3	4
1	0	5	0	0
2	5	0	6	2
3	0	6	0	1
4	0	2	1	0

Table 1.2: Interprocess costs

Level one: source vertex s which represents processor one

Level two: vertices corresponding to the program modules: one vertex per module

Level three: sink vertex t which represents processor two

As for the edges and capacities:

a) connect the source node to every module vertex i in level two with an edge with capacity β_i

b) connect two module vertices i and j with an edge (i, j) of capacity c_{ij} if modules i and j interact during the execution of the program

c) connect every module vertex i with t to form arc and give a capacity value of α_i

Notice that there is a one-to-one correspondence between s–t cuts in the network and assignments of modules to the two processors.

Consider the extreme case of scheduling all four modules on processor two. Here, there is no interprocess communication happening and the cost is just

$$\sum_{i=1}^{4} \beta_i = 4 + 10 + 3 + 8 = 25$$

Now consider the cut $c(\{s\}, V - \{s\})$ of the network. Note that its cost is $4 + 10 + 3 + 8 = 25$.

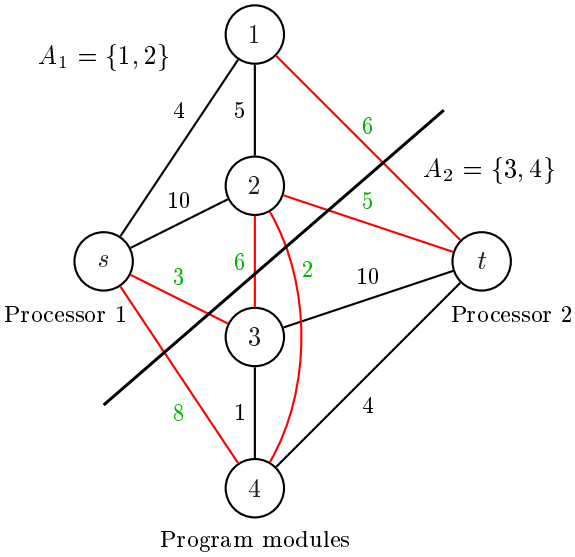
Indeed we shall always have the capacity of a cut equal to the cost of the corresponding assignment. In other words, if A_1 is the set of modules assigned to processor one and A_2 is the set of modules assigned to processor two, then the cost of this assignment is

$$\sum_{i \in A_1} \alpha_i + \sum_{i \in A_2} \beta_i + \sum_{(i,j) \in A_1 \times A_2} c_{ij}$$

The s–t cut correspondence to this assignment is

$$(\{s\} \cup A_1, \{t\} \cup A_2)$$

EXAMPLE: With $A_1 = \{1, 2\}$ and $A_2 = \{3, 4\}$ we get the following flow network:



As shown in above figure,

$$\begin{aligned} c(\{s\} \cup A_1, \{t\} \cup A_2) &= 8 + 3 + 6 + 2 + 5 + 6 \\ &= 30 \end{aligned}$$

In our example of $A_1 = \{1, 2\}$ and $A_2 = \{3, 4\}$ we saw that

$$c(\{s\} \cup A_1, \{t\} \cup A_2) = 30 \ .$$

Note that this cut contains:

1. edge $(1, t)$ of capacity α_1 and edge $(2, t)$ of capacity α_2, which implies that modules 1 and 2 are assigned to process 1

2. edge $(s, 3)$ of capacity β_3 and edge $(s, 4)$ of capacity β_4 which represents the fact that modules 3 and 4 are assigned to process 2

3. edge $(2, 3)$ of capacity c_{23} and edge (", 4) of capacity c_{24} which represents the interprocess cost for the assignment of modules 1 and 2 to processor one and modules 3 and 4 to processor two.

   We showed that the cost of the assignment A_1 and A_2 = capacity of the cut $(\{s\} \cup A_1, \{t\} \cup A_2)$. This is true for any assignment A_1, A_2. Consequently, the minimum s–t cut in the network gives the minimum cost assignment of the modules to the two processors. □

1.7.3 The baseball elimination problem

This application has nothing to do with baseball (especially the way it is played now).

There are n teams: team 1, team 2, ..., team n. Team n is our favorite team. We are in mid-season and we are interested in knowing if team n still has a chance of ending the season in first place [AMB93] [Wes96]. We assume that no ties are possible.

Suppose that team i has won $w(i)$ of the games it has already played, and that $g(i, j)$ is the number of games that team i and team j have yet to play with each other.

We say that team t is eliminated if it has no chance of finishing first; in other words, if for every possible outcome of the unplayed games, at least one team will have more wins than team t.

Team n currently has $w(n)$ wins, and there are

$$\sum_{j=1}^{n-1} g(j,n)$$

games remaining on its schedule. The best scenario for team n is if it wins all of its remaining games. Let

$$W = w(n) + \sum_{j=1}^{n-1} g(j,n)$$

Team n cannot be eliminated if the rest of the games between the other teams is such that each other team has at most W wins at the end of the season. Recall there are $g(i,j)$ games remaining between team i and team j.

Let $x(i,j)$ denote the number of games that team i wins over j (from the $g(i,j)$ games they still have to play). Note that

$$\begin{aligned} x(i,j) + x(j,i) &= g(i,j) \\ &= g(j,i) \\ \text{for each } j &= 1,2,\ldots,n \\ \text{and } i &= 1,2,\ldots,n \end{aligned}$$

Let $W(i) = w(i) + \sum_{j=1}^{n} x(i,j)$ denote the number of wins that team i has at the end of the season, for $i = 1,2,\ldots,n-1$. Team n finishes the season in first place if there is a way of completing the season such that

$$W \geq W(i) \text{ for } i = 1,2,\ldots,n-1$$

In other words, team n is not eliminated if there is some assignment of integer values $x(i,j) \geq 0$, $i = 1,2,\ldots,n$ such that

$$\begin{aligned} x(i,j) + x(j,i) &= g(i,j) \\ &= g(j,i) \\ \text{for } i &= 1,2,\ldots,n-1 \\ \text{and } j &= 1,2,\ldots,n-1 \end{aligned}$$

and

$$W = w(n) + \sum_{j=1}^{n-1} g(j,n) \geq w(i) + \sum_{j=1}^{n-1} x(i,j) \text{ for each } i = 1,2,\ldots,n-1$$

Our problem is to decide whether team n is eliminated or not.

While in some problem instances it is easy to answer this question, in others it is a challenging question.

EXAMPLE: There are 4 teams, and in the entire season, each team plays each of the other 3 teams for 3 games each. In other words, each team plays a 9–game season.

Suppose that team 1 has already played and beaten team 3 twice and team 4 twice. In other words

$$g(1,3) = 3 - 2$$

and

$$g(1,4) = 1$$

and $w(1) = 4$. Similarly, suppose that

$$g(2,3) = 1 g(2,4) = 1 \text{ and } w(2) = 4$$

and these are the only games played so far. So $w(3) = 0$ and $w(4) = 0$ (see table below).

Team i	wins $w(i)$
1	4
2	4
3	0
4	0

Table 1.3: Wins so far in the season

And there are quite a few remaining games, as the following table shows.

$i \backslash j$	1	2	3	4
1	—	3	1	1
2	3	—	1	1
3	1	1	—	3
4	1	1	3	—

Table 1.4: The remaining games

Is team 4 eliminated ?

$$\begin{aligned} w &= w(4) + \sum_{j=1}^{3} g(j,4) \\ &= 0 + 5 \\ &= 5 \end{aligned}$$

None of the teams has more than 4 wins so far in the season. But team 1 and team 2 still have to play 3 games. At the end of these games, either

$$x(1,2) \geq 2$$

(i.e. team 1 wins at least two of the 3 games) or

$$x(2,1) \geq 2$$

(that is, team 2 wins at least two of the 3 games). Thus either

$$W(1) = w(1) + \sum_{j=2}^{4} x(1,j) \geq 4 + 2 = 6$$

or

$$W(2) = w(2) + x(2,1) + x(2,3) + x(2,4) \geq 4 + 2 = 6$$

So either $W(1) > W$ or $W(2) > W$ and therefore our favorite team (team 4) is eliminated. □

The following are 2 equivalent ways of thinking of the problem: Can the games between all of the teams $1, 2, \ldots, n-1$ be played such that:

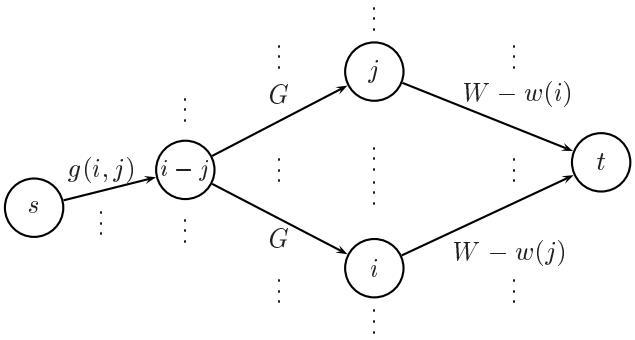
1. ate the end of the season, each team's total does not exceed W ?

2. by the end of the season, team i wins at most $W - w(i)$ of its remaining games, for $i = 1, 2, \ldots, n-1$?

The formulation of this problem as a maximum flow problem is based on the second interpretation where the commodity that is being shipped from the source s to the sink t is "remaining games between team i and team j for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$. We transform the baseball elimination problem into a maximum flow problem.

Create a source s, pair nodes (a.k.a. game nodes), one for each pair of teams, team nodes, one for each team $i = 1, 2, \ldots, n-1$ and a sink t. Note that there are $\frac{(n-1)(n-2)}{2}$ pair nodes.

We create an edge leaving s to each $< i - j >$ game node and give it capacity $g(i, j)$.

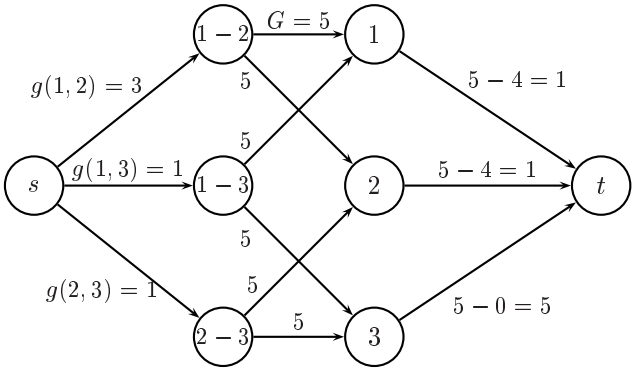


Create an edge leaving game node $< i - j >$ to team node i and another to team node j. The capacity of each of these edges is G, where G is the total number of games remaining among teams $\{1, 2, \ldots, n-1\}$. In other words

$$\begin{aligned} G &= \sum_{i=1}^{n-1} \sum_{j=1}^{i-1} g(i, j) \\ &= \sum_{< i - j > \text{ is a pair node}} g(i, j) \end{aligned}$$

Create an edge from each team node team i to t with capacity $W - w(i)$. Note that we are assuming that $W \geq w(i)$. If that is not the case, then the current number of wins of team i $w(i)$ is greater than W and so team n is eliminated (and no need to construct the network).

EXAMPLE: Let us go back to our example.



Recall that the flow here is "remaining game". A unit of flow from s to game node $< i - j >$ represents a game between team i and team j. The unit of flow then either goes to team i or team j, depending on whether team i or team j won. The unit of flow will go from the winning node, team i lets say to sink t.

The capacity of the edge from the source to game node $< i-j >$ is $g(i,j)$ and so will ensure that we cannot have more than $g(i,j)$ games between team i and team j.

Similarly, the capacity of $W - w(i)$ on the edge between team i and t will ensure that at most $W - w(i)$ games are won by team i (among its remaining games). Obviously, if that is not the case, then team n is indeed eliminated.

Also note that the cut $(\{s\}, V - \{s\})$, where V is the set of all vertices, is such that for its capacity holds

$$\begin{aligned} c(\{s\}, V - \{s\}) &= \sum_{< i - j > \text{ is a pair node}} g(i,j) \\ &= G \end{aligned}$$

Next we show that:

$$\text{Team } n \text{ is noteliminated} \quad \Leftrightarrow \quad \text{Maximum flow } = G$$

Proof:

($\Rightarrow$): We want to show that:

$$\text{Team } n \text{ is not eliminated} \quad \Rightarrow \quad \text{Maximum flow } = G$$

Team n is not eliminated, hence there is some assignment of integer values $x(i,j) \geq 0$, $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$ such that

a) $x(i,j) + x(j,i) = g(i,j) = g(j,i)$ for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$

and

b) $W = w(n) + \sum_{j=1}^{n-1} g(j,n) \geq w(i) + \sum_{j=1}^{n-1} x(i,j)$ for each $i = 1, 2, \ldots, n-1$.

We define a flow function f in the following way: For each edge leaving s, we have

$$f(s, < i - j >) = g(i,j)$$

This assignment satisfies the capacity constraint of the edges $(s, < i - j >)$. For each edge $(< i - j >, i)$, let

$$f(< i - j >, i) = x(i, j)$$

and

$$f(< i - j >, j) = x(j, i)$$

for edge $(< i - j >, j)$.

Since $g(i, j) = x(i, j) + x(j, i)$ (by a)) the node flow conservation constraint at each game vertex $< i - j >$ is satisfied. In other words:

$$\begin{array}{c}\text{flow in vertex} \\ < i - j >: \quad g(i,j)\end{array} \quad = \quad \begin{array}{c}\text{flow out of} \\ \text{vertex } < i - j >: \\ f(< i - j >, i) + f(< i - j >, j)\end{array}$$

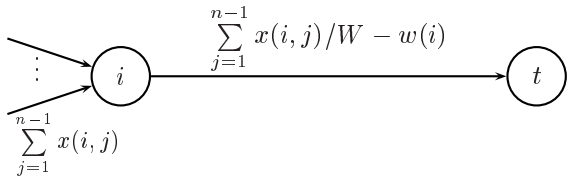
The total flow assigned to enter team i is

$$\sum_{j=1}^{n-1} x(i, j)$$

So we assign the following flow to each edge (i, t):

$$f(i, t) = \sum_{j=1}^{n-1} x(i, j)$$

Here too, the node flow conservation constraint is satisfied at each team i node.



Note that capacity constraint is also satisfied: Capacity of edge $(i, t) = W - w(i)$. Assigned flow is: $f(i, t) = \sum_{j=1}^{n-1} x(i, j)$. But by b) we know that

$$W \geq w(i) + \sum_{j=1}^{n-1} x(i, j) \text{ for each } i = 1, 2, \ldots, n - 1$$

So $W - w(i) \geq \sum_{j=1}^{n-1} x(i, j)$. In other words, $f(i, t) \leq W - w(i)$. Let us compute the value of f:

$$\begin{aligned} |f| &= \sum_{< i - j > \text{ pair node}} f(s, < i - j >) \text{ (by definition of flow value)} \\ &= \sum_{< i - j > \text{ pair node}} g(i, j) \text{ (by our flow assignment)} \\ &= G \end{aligned}$$

But there exists a cut of capacity G. Recall that cut $(\{s\}, V - \{s\})$ has capacity

$$c(\{s\}, V - \{s\}) = G$$

Therefore f as defined here is a maximum flow and $|f| = G$ (by the Max Flow-Min Cut theorem).

($\Leftarrow$): We want to show:

$$\text{Maximum flow } = G \quad \Rightarrow \quad \begin{array}{c}\text{Team } n \text{ is not} \\ \text{eliminated}\end{array}$$

Suppose that we have a maximum flow function f and $|f| = G$. Assume that f assigns integer values to each edge. It is a reasonable assumption since all capacities are integer values.

To show that team n is not eliminated, we have to show that there is an assignment of integer values $x(i,j) \geq 0$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$ such that:

a) $x(i,j)+x(j,i) = g(i,j) = g(j,i)$ for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$

and

b) $W = w(n) + \sum_{j=1}^{n-1} g(j,n) \geq w(i) + \sum_{j=1}^{n-1} x(i,j)$ for each $i = 1, 2, \ldots, n-1$.

As seen previously, $c(\{s\}, V - \{s\}) = G$ (and $|f| = G$), so:

$$f(s, < i - j >) = g(i,j) \text{ for each game node } < i - j >$$

Set $\overline{x}(i,j) = f(< i - j >, i)$ and $\overline{x}(j,i) = f(< i - j >, j)$. Since $g(i,j)$ units of flow f go into game node $< i - j >$ and there are only two edges leaving that node, we know by the conservation constraint that

$$\begin{aligned} g(i,j) &= f(s, < i - j >) \text{ in flow} \\ &= f(< i - j >, i) + f(< i - j >, j) \text{ out flow} \\ &= \overline{x}(i,j) + \overline{x}(j,i) \end{aligned}$$

So property a) is satisfied by our assignment of $\overline{x}$. Let us show that property b) is also met. Note that $\sum_{j=1}^{n-1} \overline{x}(i,j)$ units of flow f enter team i vertex.

Since there is only one edge leaving team i (going to the sink), we have

$$f(i,t) = \sum_{j=1}^{n-1} \overline{x}(i,j) \text{ for each } i = 1, 2, \ldots, n-1$$

Since f is a flow, it satisfies the capacity constraints. So we have

$$f(i,t) \leq W - w(i) \text{ for } i = 1, 2, \ldots, n-1 \ .$$

But since $f(i,t) = \sum_{j=1}^{n-1} \overline{x}(i,j)$ for $i = 1, 2, \ldots, n-1$, we have

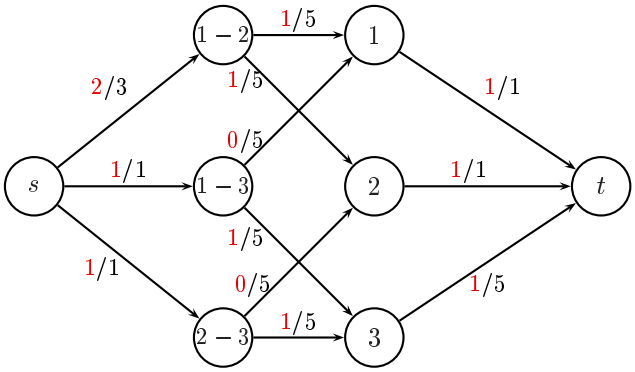
$$\begin{aligned} \sum_{j=1}^{n-1} \overline{x}(i,j) &\leq W - w(i) \text{ for } i = 1, 2, \ldots, n-1 \\ \Rightarrow W &\geq w(i) + \sum_{j=1}^{n-1} \overline{x}(i,j) \ . \end{aligned}$$

Therefore $\overline{x}$ satisfies property b). Also $\overline{x}(i,j)$ are all non-negative integer values (since we assumed that flow f takes only integer values - apply the integrality property). So team n is not eliminated and so we have shown that

$$\begin{array}{c}\text{Team } n\text{is not} \\ \text{eliminated}\end{array} \quad \Rightarrow \quad \text{Maximum flow} = G$$

□

Example:



By computing a maximum flow for our problem, we get the flow in the above figure with values in red. So

$$|f| = 4$$

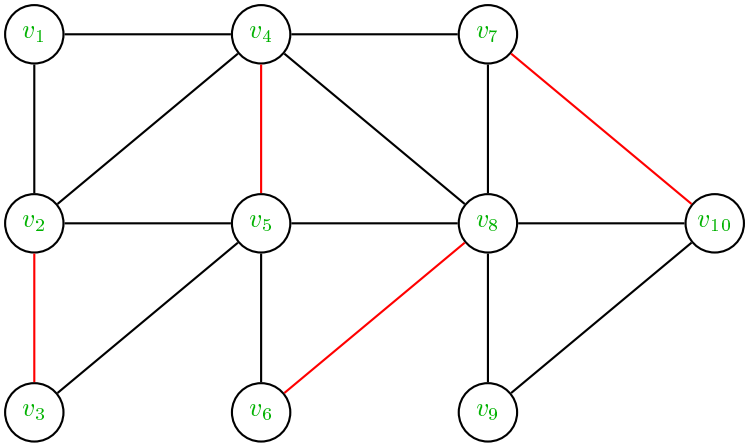
According to the theorem, team 4 is eliminated, since the total number G of games remaining among teams $1, 2, 3$ is $5 \geq 4$. □

Chapter 2

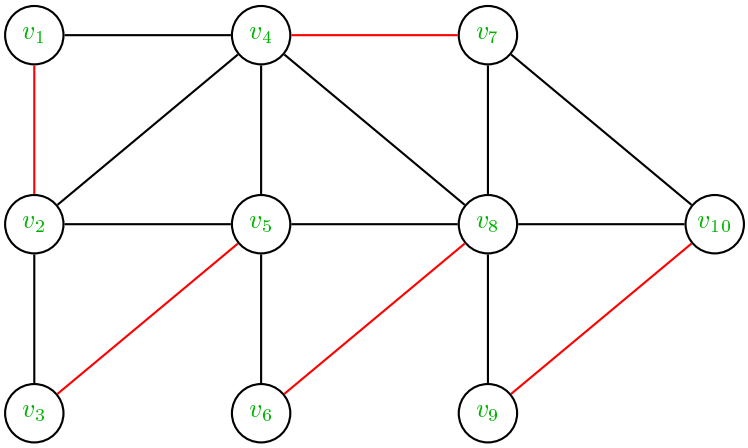
Graph Matching

A matching in a graph $G = (V, E)$ is a set M of edges, $M \subseteq E$, such that no two of the edges of M share a vertex. A vertex $v \in V$ is matched by matching M if some edge in M is incident on v; otherwise, v is unmatched. We are interested in finding a maximum matching M, in other words, a matching of maximum cardinality ($|M| \geq |M'|$ for any other matching M').

Example:



In the above graph, $M_1 = \{(v_2.v_3), (v_4, v_5), (v_6, v_8), (v_7, v_{10})\}$ is a matching. Is M_1 a maximum matching?



No. $M_2 = \{(v_1, v_2), (v_3, v_5), (v_4, v_7), (v_6, v_8), (v_9, v_{10})\}$ has more edges.

As a matter of fact, M_2 is a maximum matching since any matching of G can never have more than $\frac{|V|}{2} = 5$ edges. □

When the cardinality of a matching M is such that $|M| = \lfloor \frac{|V|}{2} \rfloor$, the largest possible matching in a graph with $|V|$ nodes then matching M is a perfect matching.

Note that in a perfect matching, every vertex $v \in G$ (except maybe one) is matched by M.

The edges of matching M are called matched edges; other edges are free.

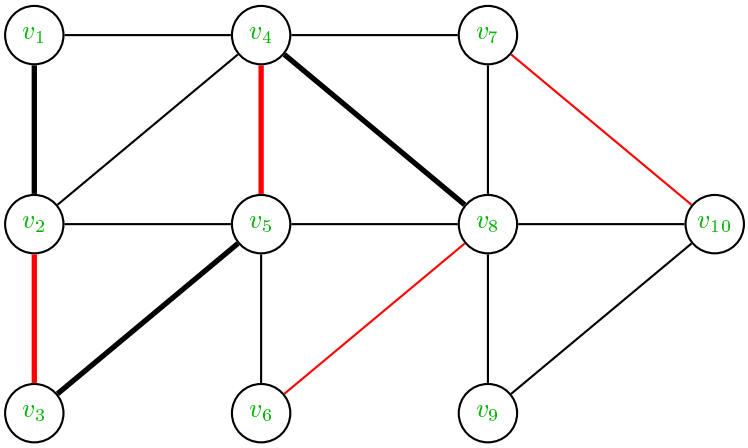
If (u, v) is a matched edge, then u is the mate of v.

Vertices that are not incident upon any matched edge are called exposed; the remaining vertices are matched.

A path $p = \langle u_1, u_2, ..., u_k \rangle$ is called an alternating path if edges (u_1, u_2), (u_3, u_4), $..., (u_{2j-1}, u_{2j}), ...$ are free, whereas $(u_2, u_3), (u_4, u_5), ..., (u_{2j}, u_{2j+1}), ...$ are matched.

Vertices that lie on an alternating path starting with an exposed vertex and have odd rank on this path are called outer. Those with even rank are called inner.

EXAMPLE: Consider the previous graph with matching:
$M_1 = \{(v_2, v_3), (v_4, v_5), (v_6, v_8), (v_4, v_{10})\}$
$P_1 = \langle v_1, v_2, v_3, v_5, v_4, v_8 \}$ is an alternating path.



The edges (v_1, v_2), (v_3, v_5) and (v_4, v_8) are free edges, while (v_2, v_3) and (v_4, v_5) are matched edges. P_1 starts with an exposed vertex: v_1.

So v_1, v_3 and v_4 (vertices with odd rank) are called outer, and v_2, v_5, v_8 occupy even positions in P_1 and are therefore inner vertices.

Similarly for $P_2 = \langle v_1, v_4, v_5, v_6, v_8, v_7, v_{10}, v_9 \rangle$; it is an alternating path with:
v_1 exposed, and outer vertices: v_1, v_5, v_8, v_{10}, and inner vertices: v_4, v_6, v_7, v_9.

$P_3 = \langle v_9, v_8, v_9 \rangle$ is also an alternating path; v_9 is exposed, v_9 and v_6 are outer, v_8 is inner. □

An alternating path $P = \langle u_1, u_2, ..., u_k \rangle$ is called augmenting if both vertices u_1 and u_k are exposed.

In our previous example: $P_2 = \langle v_1, v_4, v_5, v_6, v_8, v_7, v_{10}, v_9 \rangle$ is an augmenting path.

Recall that if A and B are sets, then $A \triangle B = (A - B) \cup (B - A)$ is the symmetric difference of A and B.

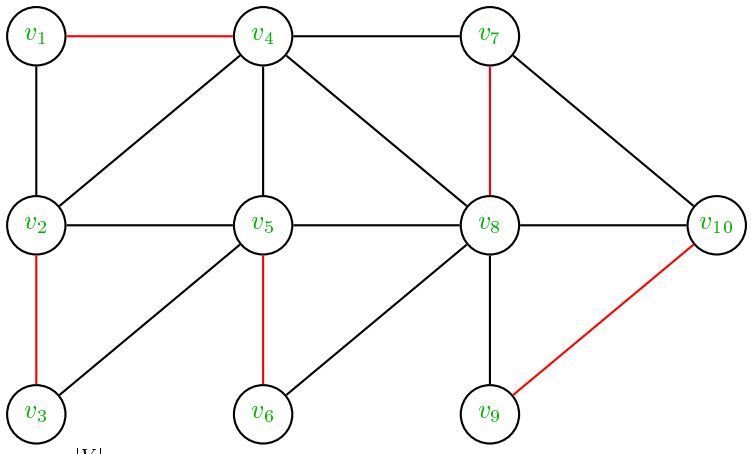
Theorem 2.0.1 *Let P be the set of edges on an augmenting path $p = \langle u_1, u_2, ..., u_{2k} \rangle$ in a graph G w.r.t. the matching M.*
Then $M' = M \triangle P$ is a matching of cardinality $|M| + 1$.

Note that in our example: $P_2 = \langle v_2, v_4, v_5, v_6, v_7, v_{10}, v_9 \rangle$ is an augmenting path w.r.t. M_1.

$P = \{(v_1, v_4), (v_4, v_5), (v_5, v_6), (v_6, v_8), (v_8, v_7), (v_7, v_{10}), (v_{10}, v_9)\}$.
$M_1 = \{(v_2, v_3), (v_4, v_5), (v_6, v_8), (v_7, v_{10}\}$.
$M_1 \triangle P = (M_1 - P) \cup (P - M_1) = \{(v_2, v_3)\} \cup \{(v_1, v_4), (v_5, v_6), (v_8, v_7), (v_{10}, v_{19}\}$.
$\Rightarrow M' = \{(v_2, v_3), (v_1, v_4), (v_5, v_6), (v_8, v_7), (v_{10}, v_{19}\}$.
Note that

1. M' is a matching
2. $|M'| = |M_1| + 1 = 4 + 1 = 5$.

EXAMPLE:



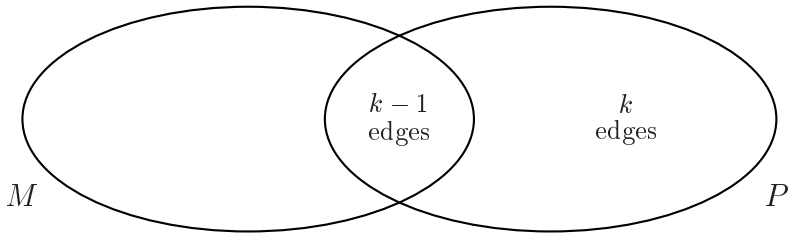
Note that $|M'| = \frac{|V|}{2} = 5$, it does not make any sense to try and look for an augmenting path w.r.t. M'. M' is an maximum matching.
Now let us go back to proving the theorem.

Proof:

1. Show that $M' = M \triangle P$ is a maximum matching.
   We want to show that no two edges in $M \triangle P$ share the same vertex in G.
   The proof is by contradiction.
   Suppose that $e \in M \triangle P$ and $e' \in M \triangle P$ are incident upon the the same vertex.
   Since $M \triangle P = (M - P) \cup (P - M)$, we have 3 cases.
   Case 1: $e \in M - P$ and $e' \in M - P$,
   Case 2: $e \in P - M$ and $e' \in P - M$,
   Case 3: $e \in M - P$ and $e' \in P - M$.
   In case 1, we have two edges, e and e', sharing the same vertex in M. But it is impossible since M is a matching.
   In case 2, since both edges are not in M, they are free edges and both are of the form (u_{2j-1}, u_{2j}) (for different values of j), and therefore cannot be incident to the same vertex.
   In case 3, suppose that $e' \in P - M, e' = (u_{2j-1}, u_{2j})$ is such that u_{2j} is such that u_{2j} is also an endpoint of some edge $e \in M - P$. But u_{2j} is a vertex of the edge $e'' = (u_{2j}, u_{2j+1}) \in M$ (a matching edge).
   So now we have two edges, e and e', in a matching M that share a common vertex. It is impossible.
   So we have just shown that M' is a matching.

2. Show that $|M'| = |M| + 1$ where $M' = M \triangle P$.
   $P = \langle u_1, u_2, ..., u_{2k} \rangle$ is such that u_1 and u_{2k} are exposed;
   $(u_1, u_2), (u_3, u_4), ..., (u_{2j-1}, u_{2j})$ are free (not in M)
   and $(u_2, u_3), (u_4, u_5), ..., (u_{2j}, u_{2j+1})$ are matched (are in M).

Note that there are $2k-1$ edges in P:
$(u_1, u_2), (u_3, u_4), ..., (u_{2k-1}, u_{2k}), ...$ are free, i.e. k of the edges of P are free, and $(u_2, u_3), (u_4, u_5), ..., (u_{2k}, u_{2k+1}), ...$ are in M, i.e. $k-1$ of the edges of P are matched.



$M' = (M - P) \cup (P - M)$	$\lvert M' \rvert = \lvert M - P \rvert + \lvert P - M \rvert$ (since $(M - P) \cap (P - M) = \emptyset$)
$\Rightarrow \lvert M' \rvert = \lvert M \rvert + 1$	[$M \cap P$ has one vertex less than $P - M$] $\square$

Note that in the previous theorem we increased the size of matching M by one, to obtain matching M', by inverting the edges of the augmenting path P: The matched edges of M are free edges w.r.t. M', and the free edges of M are now matched edges in M'. (This is true for edges that are on the augmenting path). The edges that are in M and not part of P, will of course be in M'.

Theorem 2.0.2 *A matching M in a graph G is maximum* $\Leftrightarrow$ *there is no augmenting path in G w.r.t. M*

Proof:
$\boxed{\Rightarrow}$ Direct result from the previous theorem. If there is such an augmenting path w.r.t. M then we have $M' = M \triangle P$ a new matching with $\lvert M' \rvert = \lvert M \rvert + 1$ which contradicts the fact that M is a maxmimum matching.
$\boxed{\Leftarrow}$ We have to show:
If there is no augmenting path in G w.r.t. M, then M is a maximum matching. The proof is by contradiction.
Suppose there is no augmenting path in G w.r.t. M and that M is not a maximum matching.
So there exists some matching M' of G such that $\lvert M' \rvert > \lvert M \rvert$.
Consider $M \triangle M' = (M - M') \cup (M' - M)$
The edges in $M \triangle M'$ form a subgraph (that might be disconnected) of G. Since two edges of a matching cannot be incident upon the same vertex, the subgraph $G' = (V, M \triangle M')$ will have the following properties:

1. All vertices will have degree 2 or less. If the degree of a vertex is 2, one of the edges is in M and the other in M'.
2. G' is a collection of cycles and paths.
3. The cycles are of even length and we have the same number of edges in M as in M'.

But since $\lvert M' \rvert > \lvert M \rvert$, it must be the case that in one of the paths, P, we have more edges from M' than from M.
P begins and ends with edges in M', in other words, the first and last vertices of

P are free vertices (w.r.t. M), and so P is an augmenting path in G w.r.t. M.
But this is a contradiction since we assumed that G does not have an augmenting path.
Therefore M is a maximum matching. □

Note the big similarity between this theorem and what we have seen with the construction of maximum flows in flow networks.
Edmonds was the first one to show how to find an augmenting path w.r.t. a matching M, in any given graph, in polynomial time.
Edmonds algorithm starts with an empty matching set M, searching for an augmenting path P, increases M by 1 and repeats the process until no augmenting path ca be found.
Next, we restrict our attention to finding maximim matchings in bipartite graphs.

2.1 Maximum Bipartite Matching

A bipartite graph is an undirected graph $G = (V, E)$ in which V is partitioned into two sets $L \subseteq V$ and $R \subset V$ such that

$$(u, v) \in E \Rightarrow \begin{cases} u \in L \text{ and } v \in R \\ \text{or} \\ u \in R \text{ and } v \in L \end{cases}$$

In other words, all the edges of E go between L and R.
The problem of finding a maximum matching in a bipartite graph has many applications.
For instance L: set of matchings,
R: set of tasks to be performed simultaneously.
Edge (u, v) in E: a particular machine $u \in L$ is capable of performing a particular task $v \in R$.
A maximum matching provides work for as many machines as possible.
One way of solving the matching problem for bipartite graphs is to relate it (or reduce it) to the maximum flow problem for flow networks. Thus, any problem that solves the maximum flow problem could now be used to solve the bipartite matching problem.
The flow in the network problem will correspond to matchings in the maximum bipartite matching problem.
Given a bipartite graph $G = (V, E)$ with vertex partition:
$V = L \cup R$, we define the corresponding flow network $G' = (V', E')$ as follows:

a) Add two new vertices, $s \in V'$ and $t \in V'$ and let them be the source and the sink of G'. So $V' = V \cup \{s, t\}$.

b) Add new directed edges from s to the vertices of L, and add new directed edges from the vertices of R to t.

c) Give each edge in the original graph G a direction so that all edges go from L to R.
So $E' = \{(s, u) | u \in L\} \cup \{(u, v) | u \in L, v \in R \text{ and } (u, v) \in E\} \cup \{(v, t) | v \in R\}$.

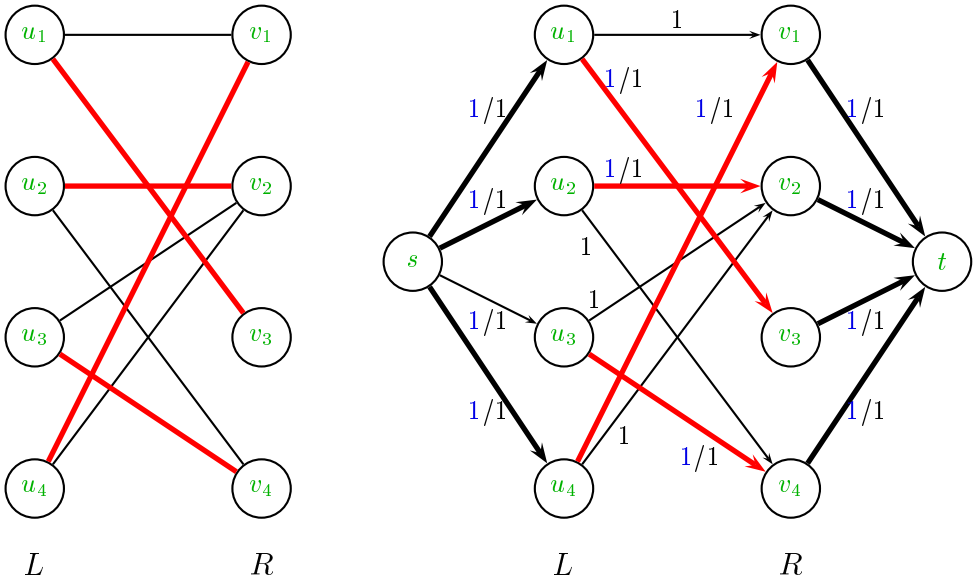
d) Assign unit capacity to each edge in E'.

Example: Given the bipartite graph $G = (V, E)$,
$L = \{u_1, u_2, u_3, u_4\}, R = \{v_1, v_2, v_3, v_4\}$ with edges shown in the figure.

The figure also shows a maximum matching, the corresponding flow network G'.

$M = \{(u_1, v_3), (u_2, v_2), (u_3, v_4), (u_4, v_1)\}$ $\qquad$ $|f| = \sum_{i=1}^{4} f(s, u_i) = 4.$

$|M| = 4$



If $f(u, v)$ is an integer for all $(u, v) \in V \times V$ then we say that the flow f is integer-valued.

Theorem 2.1.1 *$G = (V, E)$ is a bipartite graph with vertex partition $V = L \cup R$. $G' = (V', E')$ is the corresponding flow network.*

a) M is a matching in $G \Rightarrow$ there is an integer-valued flow f in G' with $|f| = |M|$,

b) f is an integer-valued flow in $G' \Rightarrow$ there is a matching M in G with $|M| = |f|$.

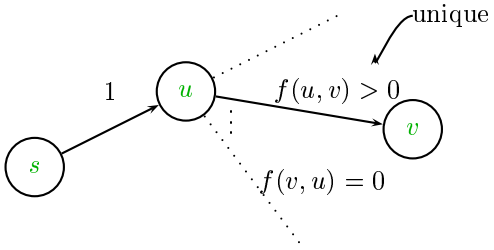
Proof:

a) Suppose that M is a matching in G.
Define the following flow function f in the corresponding flow network G':

$$(u, v) \in M \quad \Rightarrow \quad \begin{array}{l} f(s, u) = f(u, v) = f(v, t) = 1 \\ f(u, s) = f(v, u) = f(t, v) = -1 \end{array}$$

In other words, each edge in M, $(u, v) \in M$, corresponds to 1 unit of flow in G' that goes along the path $s \to u \to v \to t$.
For all other edges $(u, v) \in E'$, we define $f(u, v) = 0$.
It is easy to show that f thus defined satisfies the three properties of:
skew symmetry, capacity constraint and flow conservation.
Note that we have a cut $(L \cup \{s\}, R \cup \{t\})$, in the flow network G' and that the capacity of the cut is such that:
$C(L \cup \{s\}, R \cup \{t\}) = |M|$.
By a previous theorem we know that $c(S, T) = |f|$ for any cut (S, T).
So we have that the value of the flow we defined: $|f| = |M|$.

b) Let f be an integer-valued flow in G'. We need to define a matching M such that: $|M| = |f|$.
Define $M = \{(u,v) | u \in L, v \in R \text{ and } f(u,v) > 0)\}$.
First, we show that M is a maximum matching in G.



From the construction of G' we have that: Each $u \in L$ has one entering edge (s,u) and its capacity is 1.
Vertex u has at most one unit of positive net flow entering it.
Flow function f is integer-valued $\Rightarrow$ for each $u \in L$:

one unit of positive net flow enters u	$\Leftrightarrow$	there exists one $v \in R$ such that $f(u,v) = 1$

Thus, at most one edge leaving each $u \in L$ carries positive net flow.
Similarly for each vertex v in R; there is at most one edge entering each vertex $v \in R$ with positive net flow.
So $M = \{(u,v) | u \in L, v \in R \text{ and } f(u,v) > 0\}$ is indeed a matching in G.
Second, we show that $|M| = |f|$.
But $R = V' - (\{s,t\} \cup L)$, and by using properties studied in the section on maximum flow:
$f(L,R) = f(L,V') - f(L,L) - f(L,s) - f(L,t)$ and
$f(L,V') = 0, f(L,L) = 0, f(L,t) = 0$ (no edges between L and t)
$\Rightarrow |M| = -f(L,s) = f(s,L) = f(s,V') = |f|$ □

So to compute a maximum matching in G, we run the Ford-Fulkerson algorithm in G'. The integrality theorem:

> *If all capacities are integers, then the maximum flow f produced by the Ford-Fulkerson algorithm is such that $|f|$ is integer-valued and all $f(u,v)$ have integer values;*

assures the success of finding a matching in G.
Moreover, it is not difficult to show that the cardinality of a maximum matching in a bipartite graph G is the value of a maximum flow in its corresponding flow network G'.
As for the running time of finding a maximum matching in a bipartite graph, the complexity is $O(|V||E|)$.
Any matching in a bipartite graph has cardinality at most $min(L,R) = O(|V|)$, and since $|f| = |M|$, the value of the maximum flow in G' is $O(|V|)$.

Bipartite problems are easy to solve since they can be modeled as network flow problems and thus can be solved by using any of the many algorithms we have studied.
Note that the resulting network flow problem has a special structure, and thus one can refine algorithms with better worst-case complexity for bipartite matching problems than normal network problems.
Nonbipartite matching problems are more difficult to solve because they cannot be transformed to standard network flow problems. Nonbipartite matching problems require specialized combinatorial algorithms.

In the next section we cover some applications that can be found in [AMB93] [Wes96] and [CLR90].

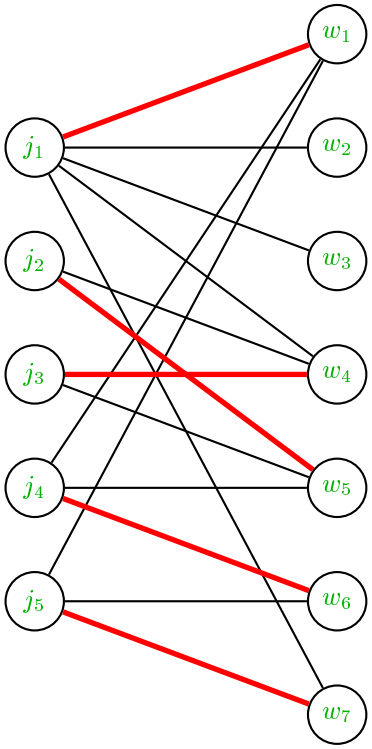
2.2 Applications

2.2.1 Bipartite Personnel Assignment

In many problems we wish to assign people to objects: e,g, jobs, machines, rooms or even each other.

a) The Personnel-assignment problem:
A company requires a number of different jobs, and each worker is suited for some of these jobs, but not others.
Assuming that each person can perform at most one job at a time, how should the jobs be assigned so that the maximum number of jobs can be performed simultaneously?

EXAMPLE:



The maximum matching whose edges appear in red shows that all five jobs can be assigned to workers.
There are 5 jobs and 7 workers. Each edge represents a suitable job assignment. How can we assign the jobs so that the maximum number of jobs can be performed simultaneously?

b) A swimming coach must select from her best 8 swimmers a medley relay team of four, each of whom will then swim one of the 4 stroaks: back, breast, butterfly and free-style. The coach knows the time of each swimmer in each stroke. The problem is to identify the team of the 4 best swimmers out of the eight that are available. The sum of times obtained by optimally matching four strokes gives the minimum feasible relay time and the corresponding team is the best team.

c) Assigning medical school graduates to hospitals
Thousands of doctors graduate from medical school in the USA. They are then eligible for residences at the different hospitals around the country. The American medical association (AMA) performs a matching process in which the graduates rank the hospitals according to their preferences and the hospitals rank the graduates according to their preferences. The AMA assigns the graduates to hospitals so that the matching is "stable".
An assignment is unstable if some graduate i is nor assigned a hospital j. But the graduate prefers hospital j over his/her current assignment, and at the same time, hospital j prefers graduate i over one of the graduates assigned to it. This alignment is unstable because both the graduate i and the hospital j have an incentive to change their current assignments. An assignment that is not unstable is said to be stable.
The objective of the AMA is to identify a stable assignment.

2.2.2 Non bipartite Personnel Assignment

Pairing volunteers for a rescue mission
Several first-aid workers from around the world have volunteered for a rescue mission in some third-world country that has been struck by a disaster. The volunteers are to be divided into two-person teams, but the members of each team must speak the same language. What is the maximum number of pairs that can be sent out on the rescue mission?
Use a graph where each vertex represents a volunteer, and an edge between volunteers i and j means that both speak the same language. A maximum matching in the graph corresponds to a maximum set of two-person teams.

□

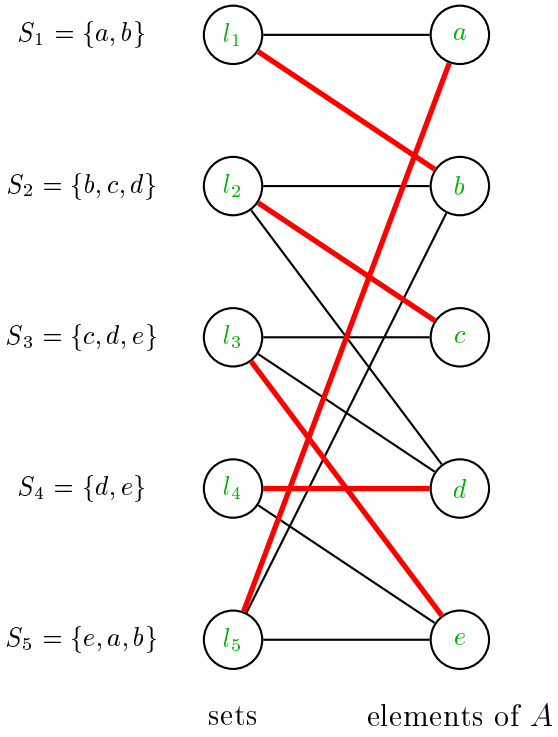
2.3 Transversals for families of subsets

Suppose that A is a nonempty finite set, and $F = \{S_1, S_2, ..., S_r\}$ is a family of (not necessary distinct) nonempty subsets of A.
A transversal (or system of distinct representatives) of family F is a sequence $T => a_1, a_2, ..., a_r >$ of r distinct elements of A, such that:
$a_i \in S_i$ for each $i = 1, 2, ..., r$.

EXAMPLE: $A = \{a, b, c, d, e\}$
$F = \{S_1, S_2, S_3, S_4, S_5\}$ where:
$S_1 = \{a, b\}, S_2 = \{b, c, d\}, S_3 = \{c, d, e\}, S_4 = \{d, e\}$ and $S_5 = \{e, a, b\}$. Then $T =< b, c, d, e, d, a >$ is a transversal, or a system of distinct representatives for family F. □
Note that to solve the "assigning medical school graduates to hospitals" application is in essence reduced to finding a transversal for the family $F = \{S_1, S_2, ..., S_r\}$ where:
S_i: is the set of all hospitals that find applicant i acceptable, and we have r graduates who applied for residences.
In this application, we assume that we do not have any ranking of hospitals by the graduates or any ranking of graduates by hospitals. If we do take the ranking into consideration, then one has to check the transversal to see if the assignment is stable.
Suppose that $G = (V, E)$ is a bipartite graph with vertex partition:
$V = L \cup R$, and that M is a matching in G.
Matching M is said to be L-saturating if each vertex in L is an endpoint of an edge

in M.
Similarly, M is R-saturating if each vertex in R is an endpoint in M.
Note that every L-saturating matching and every R-saturating matching must be a maximum matching, since in either case, either all vertices of L are matched (L-saturating) or all vertices of R are matched (R-saturating).
An L-saturating matching is also known as a complete matching from L into R.
Given a set $A = \{a_1, a_2, ..., a_r\}$ and a family $F = \{S_1, S_2, ..., S_r\}$ of (not necessarily distinct) subsets of A, we construct a bipartite $G = (V, E)$ with vertex partition $V = L \cup R$ in the following fashion:
$L = \{l_1, l_2, ..., l_r\}$ where each l_i corresponds to subset S_i in F, and $R = \{a_1, a_2, ..., a_n\}$ (i.e. $R = A$), and an edge exists between l_i and a_j if $a_j \in S_i$.
Then a transversal for family F corresponds to an X-saturating matching of graph G.

EXAMPLE: Consider our previous example where $A = \{a, b, c, d, e\}, F = \{S_1, S_2, S_3, S_4, S_5\}$ and $S_1 = \{a, b\}, S_2 = \{b, c, d\}, S_3 = \{c, d, e\}, S_4 = \{d, e\}$ and $S_5 = \{e, a, b\}$.



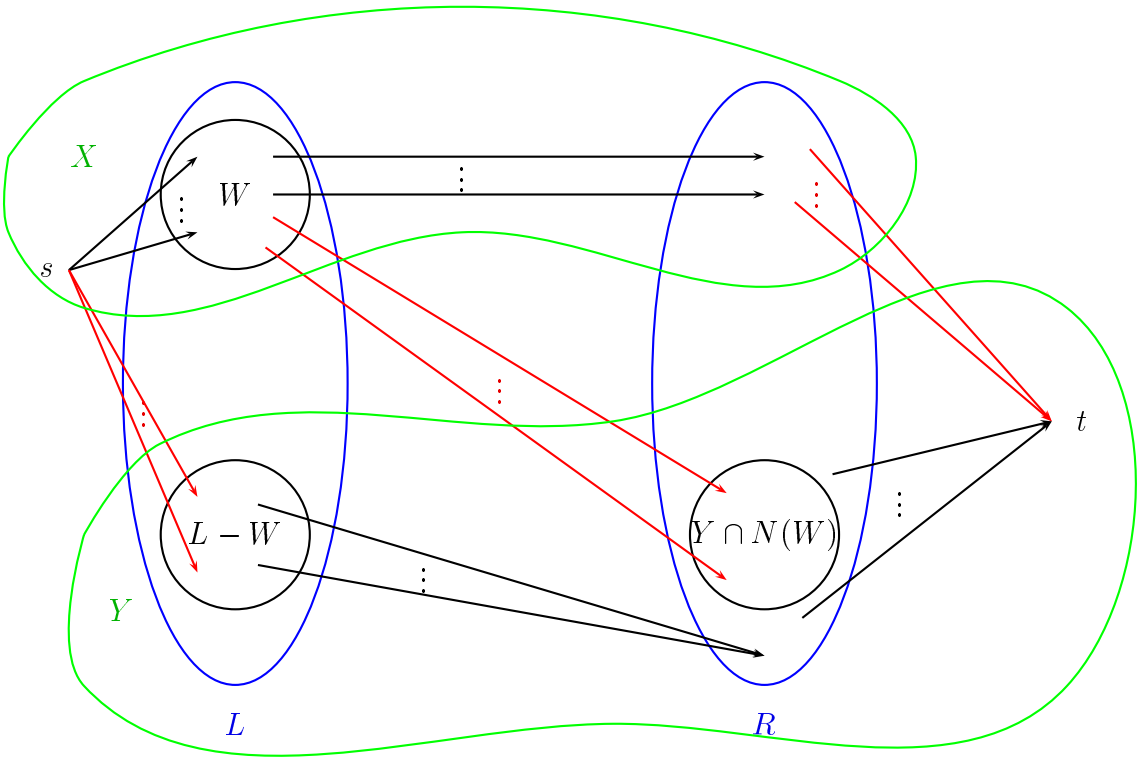
L-saturating matching : $\{(l_1, b), (l_2, c), (l_3, e), (l_4, d), (l_5, a)\}$ that corresponds to the transversal $T =< b, c, e, d, a >$ □

2.4 Hall's theorem

Hall's theorem gives a necessary and sufficient condition for a bipartite graph to have an L-saturating matching.
The theorem makes use of the set of neighbors of a subset of vertices.
Suppose that $G = (V, E)$ is a bipartite graph with vertex partition $V = L \cup R$.
Let $W \in L$. Then $N(W)$ denotes the subset of R consisting of all vertices in R that are adjacent to at least one vertex in W.
$N(W)$ is the set of neighbors of W.
The following is the 1935 theorem for bipartite graphs by Hall.

Theorem 2.4.1 *Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$.*
G has an L-saturating matching $\Leftrightarrow$ for each subset $W of L$, $|W| \leq |N(W)|$.

Proof:
$\boxed{\Rightarrow}$ Suppose that we have an L-saturating matching M in G.
Let W be any subset of L. Since M is L-saturating, every vertex $w \in W$ is matched to a vertex $y \in R$ by an edge in M.
Each such y is in $N(w)$.
So $|W| \leq |N(W)|$.
$\boxed{\Leftarrow}$ Suppose that Hall's condition holds; i.e. $|W| \leq |N(W)|$ for every $W \subseteq L$.
Construct the corresponding flow network: $G' = (V', E')$.
Note that an L-saturating matching in G would be a maximum matching in G and would correspond to a maximum flow in G' with value $|L|$.
By the max-flow min-cut theorem, it suffices to show that a minimum cut in G' has capacity $|L|$.
We know, by the construction of G' itself, that:
$cut(\{s\}, V' - \{s\} - \{t\})$ has capacity $|L|$, since all edges leaving s to L have capacity one.
So the only thing we have to prove is that any other cut of G', (X, Y), will be such that:
capacity $(X, Y) \geq |L|$.
Let (X, Y) be any cut in G', and let $W = X \cap L$.
Then $cut(X, Y)$ can be expressed as the disjoint union of three arc sets.



Note that all the edges of $cut(X, Y)$ are in 3 sets:
C_1: the set of edges from s to $L - W$.
C_2: the set of edges from W to $Y \cup N(w)$.

C_3: the set of edges from $X \cap R$ to t.
Note that these edges are red in the figure.
So $cut(X, Y) = C_1 \cup C_2 \cup C_3$.
$\Rightarrow$ capacity of $cut(X, Y) = |C_1 + |C_2| + |C_3|$ (since the sets are pairwise disjoint)
$= |L - W| + |C_2| * |X \cap R|$ (by construction of $G'-$ all capacities $= 1$).
$|C_2| \geq |Y \cap N(w)|$ (by definition of neighbors)
$= |N(w)| - |X \cap N(w)|$
Since $N(w) = [N(W)| - |X \cap N(w)] \cup [Y \cap N(w)]$.
So $|C_2| \geq |N(w)| - |X \cap N(w)|$ and:
capacity of $cut(X, Y)$
$\geq |L - W| + |N(W)| - |X \cap N(w)|X \cap R|$
Since $N(w) \subseteq R$, we have: $|X \cap N(w)| \leq |X \cap R|$.
Therefore: capacity of $cut(X, Y)$
$\geq |L - W| + |N(W)|X \cap R| + |X \cap R|$
$= |L - W| + |N(W)|$
But we assume that Hall's condition holds.
So $|N(W)| \geq |W|$.
Capacity of $cut(X, Y) \geq |L - W| + |N(W)| \geq |L - W| + |W| = |L|$
$\Rightarrow$ Minimum cut in G' has capacity $= |L|$ □

Hall's theorem can be restated in terms of a family of subsets, giving the necessary and sufficient condition for the family to have a transversal.
Obviously, the proof is exactly the same as the one we just saw.
Hall's theorem for transversals:
Let A be a nonempty finite set, and $F = \{S_1, S_2, ..., S_r$ be a family of nonempty subsets of A.

family F has a transversal	$\Leftrightarrow$	the union of any k of the subsets S_i contains at least k elements of A, for $1 \leq k \leq r$.

One of the first applications of Hall's theorem is the marriage problem:
Given a set of women, each of whom knows a subset of men, under what conditions can each of the women marry a man whom she knows?
Note that this problem has many variations.
For example, if $|L| = |R|$, Hall's theorem is known as the "marriage theorem".
The name arises from the scenario of a symmetric compatibility relation between a set of n men and a set of n women. If every man is compatible with k women and every women is compatible with k men, then there must be a complete matching using compatible pairs.

We know that a complete matching is a maximum matching.
But what if G does not have a complete matching? How can we show that a certain matching is maximum? We know that:

A matching M in a graph G is a maximum matching in G	$\Leftrightarrow$	G has no augmenting path w.r.t. M

But this is not very practical: exploring all alternating paths w.r.t. M would take a very long time.
We wold rather find an explicit structure in G that forbids a matching larger than M.

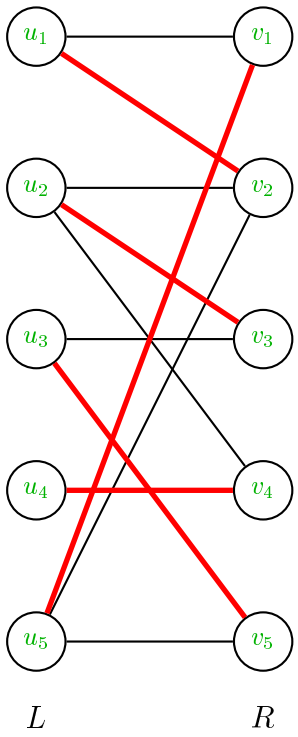
2.5 Maximum Matchings and Minimum Vertex Covers

A vertex cover of a graph $F = (V, E)$ is a set S of vertices, $S \subseteq V$, such that S contains at least one endpoint of every edge of G. We say that the vertices in S cover the edges of G.
A minimum vertex cover is a vertex cover with the least number of vertices.

EXAMPLES:

1. If the graph G represents a road network in some village that contains no traffic lights, and such that the roads are straight (and contains no isolated vertices), then the problem of assigning the minimum number of police person in order to observe and direct traffic in the entire road network is in essence solvable by finding the minimum vertex cover of the network.

2. Consider the following bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$.

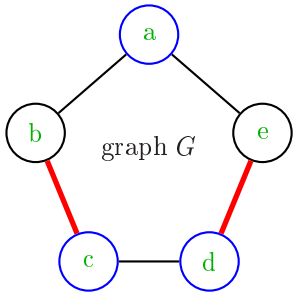


   Matching $M = \{(u_1, v_2), (u_2, v_3), (u_3, v_5), (u_4, v_4, (u_5, v_1)\}$ is a maximum matching. $|M| = 5$.
   $S = \{v_1, v_2, v_3, v_4, v_5\}$ is a minimum vertex cover of G. $|S| = 5$.

IF M is a matching in a graph G and S is a vertex cover of G, then $|M| \leq |S|$.
This is due to the fact that no two edges e and e' of M can be covered by a single vertex v, and so the size of S is at least the size of M.
Consequently, if one can find a matching M in a graph G and a vertex cover S such that $|M| = |S|$, then we know that M is a maximum matching and that S is a minimum vertex cover.
We can always find a maximum matching M and a minimum vertex cover S (with

$|M| = |S|$) for bipartite graphs.
But that does not hold for nonbipartite graphs.

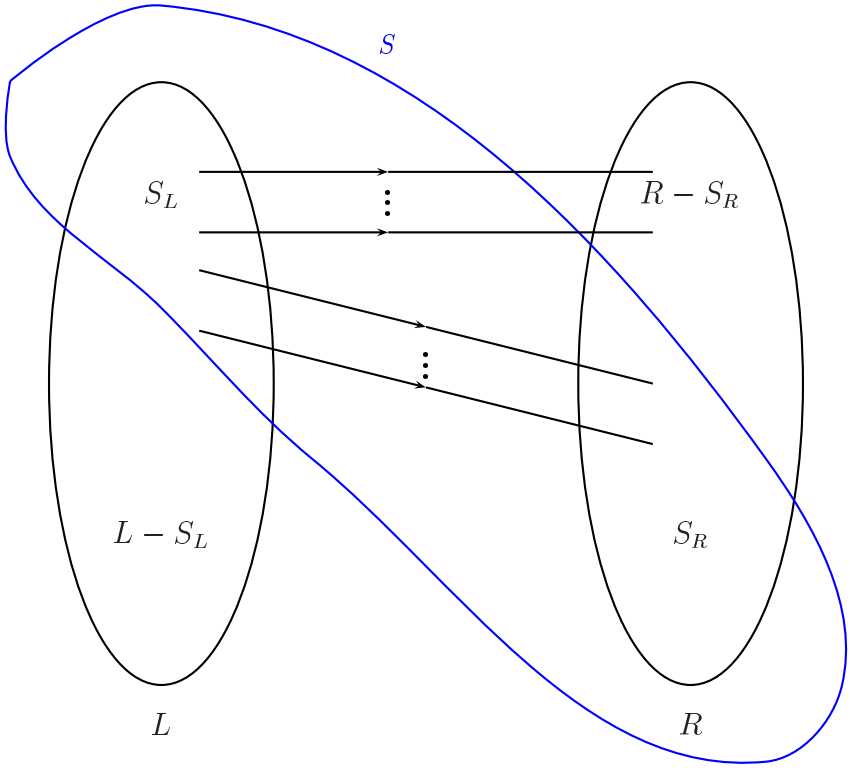
EXAMPLE: In graph G: $M = \{(b, c), (d, e)\}$ is a maximum matching.
$S = \{a, c, d\}$ is a minimum vertex cover and $|M| < |S|$.



Next we formally prove that for bipartite graphs, the size of maximum matchings and minimum vertex covers are identical.
The theorem is due to König.
König's theorem is a min-max theorem since it states equality between answers to a minimization problem and a maximization problem.

Theorem 2.5.1 *Theorem (König)*
Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$. The number of edges in a maximum matching in G is equal to the number of vertices in minimum vertex cover of G.

Proof:
Let S be a minimum vertex cover of G.
$S = (S \cap L) \cup (S \cap R)$
Let $S_L S \cap L$ and $S_R = \cap R$.
Consider the bipartite subgraph $G_1 = (V_1, E_1)$ of G induced on the vertex bipartition (i.e. with vertex partition):
$V_1 = S_L \cup (R - S_R)$.



$G_1 = (V_1, E_1)$ $\quad V_1 = S_L \cup (R - S_R)$
Suppose that W is any subset of S_L.
Let $N_{G_1}(W)$ denote the neighbor set of W of vertices in G_1.
In other words, $N_{G_1}(W) \subseteq R - S_R$.
We are going to show that $|W| \leq |N_{G1}(W)|$.
Suppose that $|W| > |N_{G1}(W)|$.
$\Rightarrow \exists w \in W$ such that $N_{G_1}(W - \{w\}) = N_{G_1}(W)$
$\Rightarrow (S_L - \{w\} \cup S_R = S$ is a vertex cover of G, contradicting the fact that S is a minimum vertex cover.
So $|W| \leq |N_{G_1}(W)|$.
By Hall's theorem, since G_1 satisfies Hall's condition, G_1 has an S_L-saturating matching M_1, with $|M_1| = S_L|$.
Now, let $G_2 = (V_2, E_2)$ be a bipartite subgraph with vertex partition:
$V_2 = (L - S_L) \cup S_R$.
By an argument similar to the above, we have that G_2 has an S_R-saturating matching M_2, with $|M_2| = |S_R|$.
Note that $|M = M_1| \cup M_2$ is a matching in G and
$|M| = |M_1| + |M_2| = |S_L| + |S_R| = |S|$.
Since in general $|M| \leq |S|$ and here we exhibited a matching and a vertex cover of equal size, we have that M is a maximum matching and S is a minimum vertex cover. □

The König-Egerváry theorem is just a different interpretation of König's theorem which involves 0/1 matrices.

Theorem 2.5.2 *The König-Egerváry theorem:*
Let A be a 0/1 matrix.
Then the maximum number of ones in A, no two of which lie in the same row or column, is equal to the minimum number of rows and columns that together contain all the ones in A.

EXAMPLE:

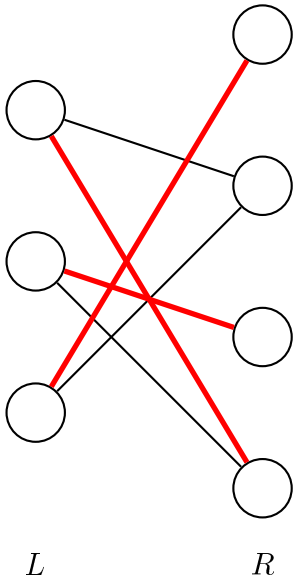
$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \boxed{1} \\ 0 & 0 & \boxed{1} & 1 \\ \boxed{1} & 1 & 0 & 0 \end{bmatrix}$$

The ones in red are such that no two are in the same row or column.
There are three of them.
There are 3 rows and 4 columns of ones.
$min\{3, 4\} = 3$ (which is the same as the number of ones marked in red) □

As to the **proof** of the theorem, it consists of two steps.
Step one:
Given A, construct a bipartite graph $G = (V, E)$ with vertex partition $V = L \cup R$ and use A as an adjacency matrix of G, where L is the set of vertices corresponding to the rows of matrix A, and R is the vertex set corresponding to the columns.

EXAMPLE:

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \boxed{1} \\ 0 & 0 & \boxed{1} & 1 \\ \boxed{1} & 1 & 0 & 0 \end{bmatrix}$$



Step two:
Apply König's theorem.

EXAMPLE:
In the above example, note that the matching marked in red in G corresponds to the three ones marked in red in A. And the three vertices of L is a minimum vertex cover. □

The König-Egerváry theorem has a few applications, such as in scheduling.

2.5.1 The Bottleneck Problem

Suppose that a manufacturing process consists of five operations that are to be performed simultaneously on five machines.

machine / operations	M1	M2	M3	M4	M5
op1	4	5	3	6	4
op2	5	6	2	3	5
op3	3	4	5	2	4
op4	4	8	3	2	7
op5	2	6	6	4	5

The time in minutes it takes each machine to perform each of the five operations is given in the table.
We would like to determine if it is possible to schedule all five operations so that the whole process is completed within 4 minutes.
Construct a matrix A whose a_{ij} entry is such that:

$$a_{ij} = \begin{cases} 1 & \text{if op}i \text{ takes at most 4 minutes on machine M}j \\ 0 & \text{otherwise} \end{cases}$$

A is a 5×5 matrix.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 1 & 0 & \boxed{1} \\ 0 & 0 & \boxed{1} & 1 & 0 \\ 1 & \boxed{1} & 0 & 1 & 1 \\ \boxed{1} & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & \boxed{1} & 0 \end{bmatrix}$$

It is possible to find 5 ones no two of which are in the same row or column.
They are marked in red.
Therefore, it is possible to complete the whole process within 4 minutes.
As a matter of fact, the five ones give the assignments.

	duration
op1 is assigned to $M5$:	4 minutes
op2 is assigned to $M3$:	2 minutes
op3 is assigned to $M2$:	4 minutes
op4 is assigned to $M1$:	4 minutes
op5 is assigned to $M4$:	4 minutes

Since the operations are performed in parallel, all 5 will finish within 4 minutes. □

2.6 Weighted matching

We are given a graph $G = (V, E)$ in which each edge $e \in E$ has a weight $w(e) = w_{ij}$ where $e = (v_i, v_j)$ and $w_{ij} \geq 0$.
The problem consists in finding a matching of G with the largest possible sum of weights.
Note that the previous (unweighted) graph matching problem is a special case of the weighted matching problem; just let $w_{ij} = 1$ for all $(v_i, v_j) \in E$.
One can also assume that instead of G we have a complete graph letting the weights of the edges that are missing in G be equal to zero.
Similarly, we can assume that we have an even number of nodes. If it is not the case, just add a node with edges of weight zero incident to it.
So, a bipartite weighted graph can be assumed to be a complete bipartite graph with vertex partition $V = L \cup R$ where $|L| = |R|$.
Note that the solution with largest sum of weights will always be a complete matching.

It is sometimes more convenient to think of this problem as a minimization problem by considering costs c_{ij} where:
$c_{ij} = W - w_{ij}$ with $W = max\{w_{ij}\} + 1$.
Once again, tackling the weighted bipartite problem is easier than dealing with general weighted graphs.
The bipartite weighted matching problem is also known as the assignment problem, since it solves the best assignment of tasks to workers problem where the value of w_{ij} represents the "production" of the i^{th} worker when given task j to perform. The assignment problem can be described by a linear program. The Hungarian method solves the weighted matching problem for a complete bipartite graph with $2|V|$ nodes in $O(|V|^3)$ arithmetic operations.

Chapter 3

String Matching

Given a string P_m called a pattern and a longer string T_n called the text, the (exact) string matching problem is to find all occurrences, if any, of pattern P_m in text T_n.

Example: $P_m = aba, T_n = bbabaxababay$
P_m occurs in T_n starting at locations: $3, 7, 9$. Note that two occurrences of P_m may overlap. □

Some off the more common applications are in:

a) word processors

b) utilities such as grep on UNIX

c) textual information retrieval programs such as MEDLINE, LEXIS or NEXIS

d) library catalog searching programs that have replaced physical card catalogs in most large libraries

e) Internet browsers and crawlers that sift through massive amounts of text available on the Internet for material containing specific keywords

f) Internet news readers that can search for the articles for topics of interest

g) telephone directory assistance

h) on-line dictionaries and thesauri

i) numerous specialized databases, such as molecular biology databases holding raw DNA, RNA and amino acid strings, or processed patterns called motifs derived from the raw string data. Genbank is the major U.S. DNA database for example.

3.1 Naive String Matching

The naive method aligns the left end of P_m with the left end of T_n and then compares the characters of P_m and T_n left to right until either two unequal characters are found or until P_m is exhausted, in which case an occurrence of P_m is reported. In either case, P_m is then shipped one place to the right, and the comparisons are restarted from the left end of P_m. The process is repeated until the right end of P_m shifts past the right end of T.

NAIVE-STRING
for $(i = 0; T[i] \neq' \backslash 0'; i++)$
for $(j = 0; (T[i+j] \neq' \backslash 0') \wedge (P[j] \neq' \backslash 0') \wedge (T[i+j] == P[j]); j++);$
if $(P[j] ==' \backslash 0')$ **then** output "found a match"

Naive-String has two nested loops: The outer loop runs in $O(n)$ time, and the inner loop runs in $O(m)$ time. So the running time of Naive- String is $O(mn)$. The naive string matching algorithm can be visualized as a sliding template containing the pattern P_m over the text T_n, and marking down the shifts for which all of this characters on the template equal the corresponding characters in the text T_n.

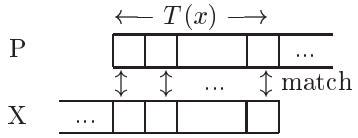
T_n AGRICULTURE...
P_m SHIFT → INA

Here, the template (pattern $P_m = INA$ slides next to $T_n = AGRICULTURE$, from left to right. The pattern in the figure has been shifted 3 ($s = 3$) units, and we still have no match. Naive-String is not optimal for the string matching problem. In the next section, we introduce notions and terminology that we use to investigate string matching algorithm that are more efficient than the naive algorithm.

3.2 String Matching Algorithms [CLR90]

Notation:

$P[1..m]$	array of text of length m (also denoted by P_m)
$\omega \sqsubset x$	ω is a prefix of x, i.e. $x = \omega y$ for some string $y \in \sum^*$
$\omega \sqsupset x$	ω is a suffix of x, i.e. $x = y\omega$ for some string $y \in \sum^*$
P_k	k-character prefix $P[1..k]$ of $P1..m]$
	Note: $P_0 = \epsilon \qquad P_m = P = P[1..m]$
τ	auxiliary function (aka suffix function corresponding to P)
	$\tau : \sum^* \rightarrow \{0, 1, ..., m\}$
	$x \rightarrow \tau(x) =$ length of longest prefix of P that is a suffix of x
	i.e. $\tau(x) = max\{k \mid P_k \sqsubset x\}$

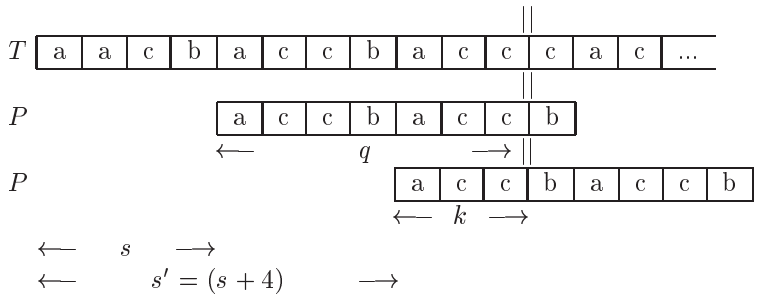


Example: $\sum = \{a, b, c\}, P = abbac$
Then $\tau(\epsilon) = 0, \tau(caba) = 1, \tau(bbcababb) = 3$

3.3 Knuth-Morris-Pratt Algorithm [KMP]

KMP is a linear-time string-matching algorithm. It uses an auxiliary function $\pi[1..m]$ which is precomputed from the pattern.

The speed up in KMP is based on the following observation:



The first 7 characters of P match the characters of T after a shift s of length 4.

From the 7 (value of q) characters we know that we can get

no match by performing a $s+1$ shift,
no match by performing a $s+2$ shift,
no match by performing a $s+3$ shift,
but a potential match at a $s+4$ shift since $\boxed{a}$ of P is aligned with $\boxed{a}$ of T.

KMP will not perform the $s+1, s+2, S+3$ shifts. It will proceed directly to $s+4$.

How?

The information it needs is contained in π that it computes from P before even starting the matching process.

Back to the strings T and P:

Given that pattern characters $P[1..9]$ match text characters $T[s+1..s+9]$, what is the least shift $s' > s$ such that:

$P[1..k] = T[s'+1..s'+k]$ where $s'+k = s+9$? Such a shift $s'(s' = s+4$ in our example)is the first shift greater than s that would yield a potential match. Note that *acc* (i.e. P_k) is a suffix of P_q. So an equivalent question to the one above is: What is the longest value of $k, k < 9$ such that $P_k \sqsubset P_q$? Once we compute k, we know that the next potential match might occur at a shift $s' = s + (9-k)$. The precomputed information that KMP uses is the prefix function for the pattern P:

$\pi : \{1, 2, ..., m\} \rightarrow \{0, 1, ..., m-1\}$
$9 \rightarrow \pi[9]$

where $\pi[9] = max\{k | (k < 9) \wedge (P_k \sqsubset P_q)\}$ i.e. $\pi[9]$ is the length of the longest prefix of P that is a **proper** suffix of P_q.

EXAMPLES:

1. Let $P = accbaccb$

P_q		P_k	
P_1	a		
		a	$\pi[1] = 0$
P_2	ac		
		ac	$\pi[2] = 0$
P_3	acc		
		acc	$\pi[3] = 0$
P_4	accb		
		accb	$\pi[4] = 0$
P_5	accba		
	a	ccba	$\pi[5] = 1$
P_6	accbac		
	ac	cbac	$\pi[6] = 2$
P_7	accbacc		
	acc	bacc	$\pi[7] = 3$
P_8	accbaccb		
	accb	accb	$\pi[8] = 4$

2. If $P = abababacca$ then $\pi[1] = \pi[2] = \pi[9] = 0, \pi[3] = \pi[10] = 1, \pi[4] = 2, \pi[5] = 3, \pi[6] = 4, \pi[7] = 5$ and $\pi[8] = 6$.

q	1	2	3	4	5	6	7	8	9	10
$P[q]$	a	b	a	b	a	b	a	b	c	a
$\pi[q]$	0	0	1	2	3	4	5	6	0	1

P_q	P_k	
P_1 a		
	a	$\pi[1] = 0$
P_2 ab		
	ab	$\pi[2] = 0$
P_3 aba		
a	ba	$\pi[3] = 1$
P_4 abab		
ab	ab	$\pi[4] = 2$
P_5 ababa		
aba	ba	$\pi[5] = 3$
P_6 ababab		
abab	ab	$\pi[6] = 4$
P_7 abababa		
ababa	ba	$\pi[7] = 5$
P_8 abababab		
ababab	ab	$\pi[8] = 6$
P_9 abababac		
	abababab	$\pi[9] = 0$
P_{10} abababababca		
a	babababca	$\pi[10] = 1$

$P = abababacca$
For example for $q = 9$ we have:

P_7 = a b a b a b a

a b a b a b a b a $[k = 1]$
a b a b a b a $[k = 3]$
a b a b a b a $[k = 5]$

$\pi[7] = max\{k|(k < 7) \wedge (P_k \sqsubset P_7\} = max\{1, 3, 5\} = 5$

Similarly, for $q = 8$ well have:
$\pi[8] = 6$
$P_8 = abababab$
But note that for $q = 9$ we have:
$P_q = abababac$ and so $\pi[9] = 0$
and finally $P_{10} = abababacca$ and $\pi[10] = 1$ □

Remark:
$\pi : \{1, 2, ..., m\} \to \{0, 1, 2, ..., m - 1\}$
$9 \to \pi[q]$
where $\pi[9] = max\{k|(k < q) \wedge /P_k \sqsubset P_q)\}$
i.e., $\pi[9]$ is the length of the longest prefix of P that is a proper suffix of P_q

In EXAMPLE 2:

q	1	2	3	4	5	6	7	8	9	10
$P[q]$	a	b	a	b	a	b	a	b	c	a
$\pi[q]$	0	0	1	2	3	4	5	6	0	1

The auxiliary function π indicates how much of the last comparison can be reused if it fails. It shows how much of the beginning of the string matches up to

the portion immediately preceding a failed comparison. If the comparison fails at $q = 9$ for instance, then we know that a and b in positions 3 and 4 are identical to a and b in positions $\pi[3] = 1$ and $\pi[4] = 2$. The comparison fails at $q = 9$. The portion immediately preceding $q = 9$ is: $\pi[8] = 6$, so *ababab* in positions 3 to 8 match *ababab* in positions 1 to 6. For t hat reason, it is sometimes called the failure function.

```
PREFIX-FUNCTION(G)
m ←length of pattern P
π[1] = 0                                   //this is a special case
k ← 0
for q ← 2 to m
2   while 1 (k>0) ∧P[k + 1] ≠ P[q])        //is this the end of
      k ← π[k]
                                           //the match ?
  if P[k + 1] = P[q]                       //we have a match so
                                           //increment k
    then k ← k + 1
  π[q] ← k
return π
```

1 Note that the whole loop is activated only when $k > 0$
(and not the very first time, when $q = 2$).
First we must have a match $P[k + 1] = P[q]$ to increment k (to $k + 1$).

2 $P[q] \neq P[k + 1]$ and so now we go back in P to find the
1 character that matches $P[q]$.
Note that <u>all</u> positions $q - 1, q - 2, q - 3, q - 4, ...$ are checked.

```
KMP(T, P)
n ←length of T
m ←length of pattern P
π[1] ← PREFIX-FUNCTION(P)
q ← 0
for i ← 1 to n
  while (q > 0) ∧ P[q + 1] ≠ T[i])     //pattern match has failed
    q ← π[q]                           //move pattern back
  if P[q + 1] = T[i]                   //potential pattern match
    then q ← q + 1
  if q = m                             //success of pattern match
    then print "Pattern occurs
                with shift i − m"
      q ← π[q]
end
```

EXAMPLE:
$T = BANANORANANANO$
$P = NANA$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
B	A	N	A	N	O	R	A	N	A	N	A	N	A	N	O
		N	A	N	A										
				N	A										

no comparison is performed here.
$T[i] \neq P[q + 1] : T[6] \neq P[3 + 1]$
$\Rightarrow q \leftarrow \pi[3]$ [i.e.$q \leftarrow 1$].

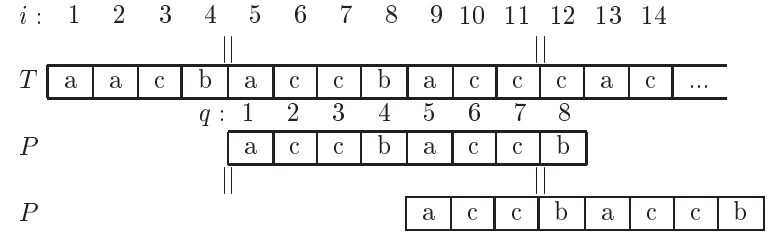
after $T[6] \neq P[3+1]$ we compare $T[6]$ and $P[1+1]$
because we know that $T[5] = P[1]$.

The procedure outputs:
Pattern occurs with shift 8
Pattern occurs with shift 10

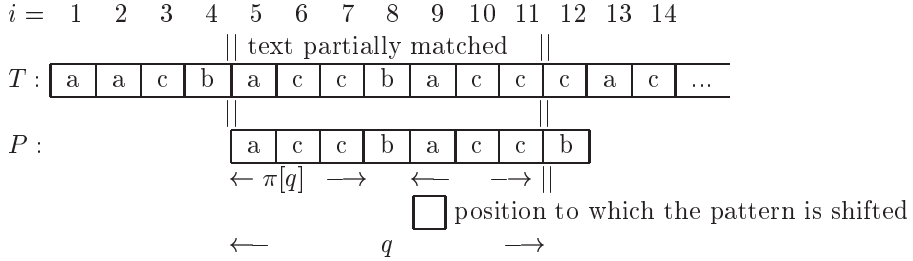
KMP(T, P) assumes that we have computed the auxiliary function.
PREFIX-FUNCTION(P) gives:

P	N	A	N	A
π	0	0	1	2

□



In the **while** loop:
$P[7+1] \neq T[12]$
$q \leftarrow \pi[7]$ (i.e. $q \leftarrow 3$)
Compare $P[\pi[q]+1]$ to $T[12]$
which has the effect of having shifted $\pi[9]+1$ characters
(where q is the last position in P where we have a character match).



Suppose $T[i-q, ..., i-1]$ matches $P[1, ..., q]$ we have a mismatch at i:
$T[i] \neq P[q+i]$.
At this point we know that:
$P[1..\pi[q]]$ is the longest proper prefix of $P[1..q]$ that is also a suffix of $P[1..q]$.
So P is shifted $\pi[q]+1$ characters to the right so that we now have:
$T[i-\pi[q]..i-1]$ matching $P[1..\pi[q]]$ and we continue. KMP now compares $T[i]$ to $P[\pi[q]+1]$ □

Procedure KMP(T, P) compares $P[q+1]$ and $T[i]$. In every iteration through the for loop, one of three things happens:

1. $q > 0$ and $P[q+1] \neq T[i]$
   which signals the end of a character match and therefore the current potential pattern match has failed. So now the question is how far back do we move the pattern to continue with the search (and to look for another potential pattern match).

2. $P[q+1] = T[i]$
   which means that we have a character match and the current potential pattern match is still possible.
   So now let us compare:
   $P[q+2]$ and $T[i+1]$ (at the next iteration of the for loop).

3. $q = m$
   signals the success of a pattern match (all m characters of the pattern were matched). So the procedure announced it, and as was the case with 1., we now move back the pattern to resume the search for the next possible pattern match.

3.3.1 Analysis of Prefix-Function

```
     PREFIX-FUNCTION(G)
     m ←length of pattern P
     π[1] = 0                                  //this is a special case
(1)  k ← 0
     for q ← 2 to m
       while (k>0) ∧P[k + 1] ≠ P[q])           //is this the end of the match ?
(2)      k ← π[k]
       if P[k + 1] = P[q]                      //we have a match so increment k
(3)      then k ← k + 1
       π[q] ← k
     return π
```

Claim: The running time of PREFIX-FUNCTION is $O(m)$.
Perform an amortized analysis where the potential is the variable k of the algorithm. The potential has 0 as initial value (see (1)). Since $\pi[k] < k, k$ is decreased (see (2)) but never becomes negative since $\pi[k] > 0$ for all k. The other place where k can change value is (3), where it can be incremented by one for each iteration of the for loop. So we have $k < 9$ upon entering the for loop, and since q is incremented by one at each iteration while k is either:

a) incremented by one (for some iterations (see 3)) or

b decremented (see 2)

Well still have $k < q$ (at all times). So the amortized cost of the for loop body is $O(1)$. The final potential function is at least as great as the initial potential function, and the for loop iterates $m - 1$ times. So the total actual worst-case running time of PREFIX-FUNCTION is $O(m)$.

3.3.2 Analysis of KMP

```
     KMP(T, P)
     n ←length of T
     m ←length of pattern P
(5)  π[1] ← PREFIX-FUNCTION(P)
(1)  q ← 0
     for i ← 1 to n
       while (q > 0) ∧ P[q + 1] ≠ T[i])        //pattern match has failed
(2)      q ← π[q]                              //move pattern back
       if P[q + 1] = T[i]                      //potential pattern match
(4)      then q ← q + 1
       if q = m                                //success of pattern match
         then print "Pattern occurs
                    with shift i − m"
(3)        q ← π[q]
     end
```

Claim: The running tome of KMP is $O(n)$.

Perform an amortized analysis there the potential is the value of q. The potential has an initial value of 0 (see (1)). Since $\pi[q] < q$, q is decreased (see (2 and 3)) but never becomes negative since $\pi[q] \geq 0$. The potential function is increased by at most one (see (4)), so the amortized cost of the for loop body is $O(1)$. The final potential function is at least as great as the initial potential function, and the for loop iterates n times. So the total actual worst-cast running time of the for loop of KMP is $O(n)$.

But KMP makes use of PREFIX-FUNCTION (at (5)) which is $O(m)$. So the time complexity of KMP is $O(m+n)$. □

KMP solves the pattern matching problem in $O(m+n)$ instead of $O(mn)$ of the naive algorithm. The whole idea is based on shifting the pattern as far to the right as possible without bypassing potential matches, whenever a complete match occurs or a character mismatch occurs.

3.4 Rabin-Karp Algorithm

The Rabin-Karp algorithm interprets the symbols of the alphabet, Σ, as numbers and considers the strings, T and P, as numbers to the base d, where $d = |\sum|$. So for instance the string 31415 corresponds to the decimal number $31,415$ when we let $\Sigma = \{0, 1, ..., 9\}$.

Let p be the number corresponding to $P[1..m]$, and t_s be the number corresponding to $T[s+1, ..s+m]$.

Note: $t_s = p \Leftrightarrow T[s+1..s+m]$

In other words, s is a valid shift if and only if $t_s = p$. Comparison of a pattern with part of the text involves comparing 2 numbers, which takes time $O(1)$ (once the numbers are computed). Use Horner's rule to compute p and t_s. The number associated with pattern p, $p = P[m] + d(P[m-1] + d(P[m-2] + ... + d(P[1])...)$ where $P[1]$ is the highest order digit, and $P[m]$ is the lowest order digit.

The time to compute p is $O(m)$ assuming each arithmetic operation takes $O(1)$ time. The value t_0 can be similarly computed from $T[1..m]$ in time $O(m)$.

$t_{s+1} = d(t_s - d^{(m-1)}T[s+1]) + T[s+m+1]$, where

d	:	shifts left
$t_s - d^{(m-1)}T[s+1]$	:	removes the high order digit
$T[s+m+1]$	:	adds the low order digit.

EXAMPLE:

Suppose $d = 10, m = 5$ and $t_s = 31415$. Also suppose that $T[s+5+1] = 2$; in other words, the text has '2' right after 5 of 31415.

$t_{s+1} = d(t_s - d^{(m-1)}T[s+1]) + T[s+m+1]$

$t_{s+1} = 10(31415 - 10^3(3)) + 2$

since $t_s = 31415 = T[s+1..s+m]$

i.e. $T[s+1] = 3$ So $t_{s+1} = 10(31415 - 30000) + 2$

$= 10(1415) + 2$

$= 14152$, which is what we have.

$$\overbrace{3\ 1\ 4\ 1\ 5}^{t_s}\ 2$$

$$3\ \underbrace{1\ 4\ 1\ 5\ 2}_{t_{s+1}}$$

□

Since t_{s+1} can be computed from t_1 in $O(1)$ time, p and $t_0, t_1, ..., t_{n-m}$ can all be computed in $O(n+m)$ and we can thus find all occurrences of the pattern

$P = P[1..m]$ in the text $T = T[1..n]$ in time $O(n+m)$. However, if m and d are large, this running time is unrealistic. Perform all the arithmetic $mod\ q$, for some prime number q of base d, such that dq fits within one computer word. This will allow all of the necessary computations to be performed with single-precision arithmetic.

So the recurrence equation with $mod\ q$ is:

$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1])mod\ q$

where $h \equiv d^{(m-1)}(mod\ q)$.

EXAMPLE: $T = 5\quad 5\quad 3\quad 1\quad 2\quad 2\quad 7\quad 3\quad 1$

$\Sigma = \{0, 1, ..., 9\}$ $P = 12$ and $q = 3$.

Note $P\ mod\ q = 0$.

$$\underbrace{5\ \ 5}_{55\ mod\ 3=1}\ \ 3\ \ 1\ \ 2\ \ 2\ \ 7\ \ 3\ \ 1$$

$$5\ \ \underbrace{5\ \ 3}_{2}\ \ 1\ \ 2\ \ 2\ \ 7\ \ 3\ \ 1$$

$$5\ \ 5\ \ \underbrace{3\ \ 1}_{1}\ \ 2\ \ 2\ \ 7\ \ 3\ \ 1$$

$$5\ \ 5\ \ 3\ \ 1\ \ \underbrace{2\ \ 2}_{0}\ \ 7\ \ 3\ \ 1$$

match at position 4

$$5\ \ 5\ \ 3\ \ 1\ \ 2\ \ \underbrace{2\ \ 7}_{0}\ \ 3\ \ 1$$

spurious hit at position 6

$$5\ \ 5\ \ 3\ \ 1\ \ 2\ \ 2\ \ \underbrace{7\ \ 3}_{1}\ \ 1$$

$$5\ \ 5\ \ 3\ \ 1\ \ 2\ \ \underbrace{2\ \ 7}_{1}\ \ 3\ \ 1$$

$$5\ \ 5\ \ 3\ \ 1\ \ 2\ \ 2\ \ 7\ \ \underbrace{3\ \ 1}_{1}$$

Note that

$t_6 = T[6,7] = 27$ and $27 \equiv p(mod\ 3)$

But $27 \neq p$

So we might have $t_s = p(mod\ q)$ but $T[s+1..s+m] \neq P[1..m]$ This is called a spurious hit. In this case, we need to explicitly compare $P[1..m]$ and $T[s+1..s+m]$ which takes $O(m)$ time. Thus, whenever we have $t_s \equiv p(mod\ q)$ we should compare t_s and p to determine if we have a valid pattern match or just a spurious hit. □

```
RABIN-KARP(T, P, d, q)
n ←length of T
//p mod q and t_s mod q
//are the fingerprints of P and t_s
m ←length of pattern P
h ← d^{m-1} mod q
p ← 0
t_0 ← 0
for i ← 1 to m                          //loop computes:
    p ← (dp + P[i]) mod q               //p ← P[1..m] mod q
    t_0 ← (dt_0 + T[i]) mod q           //t_0 ← T[1..m] mod q
for s ← 0 to n − m                      //iterate through all pos. shifts
(*)if p = t_s                           //is p = t_s where
//if p = t_s, and P[1..m] ≠ T[s + 1..s + m]
//then we have a spurious hit.          //t_s ≡ T[s + 1..s + m] mod q
    then if P[1..m] = T[s + 1..s + m]
            then print "Pattern occurs
            with shift s"               //we have a pattern match
            if s < n − m then           //success of pattern match
                t_{s+1} ← (d(t_s − T[s + 1]h)
                        +T[s + m − 1]) mod q
end
```

To make sure that $t_s = T[s+1..s+m] mod\ q$ whenever we check $p = t_s$ at (*), we need the last **if** statement in the **for** loop (**if** $s < n-m$) and obviously t_{s+1} should not be updated if we reach the end of T.

EXAMPLE: $P = 31415$

```
RABIN-KARP(T, P, d, q)
n ←length of T                          T = 235902314152...
m ←length of pattern P                  P = 31415, q = 13
h ← d^{m-1} mod q                       h = 10^4 mod 13
p ← 0
t_0 ← 0
for i ← 1 to m                          31415 ≡ 7 mod 13
    p ← (dp + P[i]) mod q               p ≡ 7 mod 13
    t_0 ← (dt_0 + T[i]) mod q           23590 = t_0 ≡ 8 mod 13
for s ← 0 to n − m                      t_0 = T[1..m] = 23590
  if p = t_s                            p = 7 and t_0 = 8
    then if P[1..m] = T[s + 1..s + m]
            then print "Pattern occurs
            with shift s"
            if s < n − m
              then t_{s+1} ← (d(t_s − T[s + 1]h)
                      +T[s + m − 1]) mod q
end
```

$t_1 = 10[23590 - 2(1000)] + T[6] = 10(3590) + 2 = 35902$

So next time through the loop check $p = 7$ and $t_1 = 9$ (since $35902 \equiv 9\ mod\ 13$).

3.4.1 Analysis of Rabin-Karp

A) Worst-case analysis:

```
RABIN-KARP(T, P, d, q)
n ←length of T                                  |
m ←length of pattern P                          |
h ← d^{m-1} mod q                               | Example (of worst case)
p ← 0                                           | P = a^m and T = a^n
t_0 ← 0                                         |
for i ← 1 to m                                  | Since each one of the
    p ← (dp + P[i]) mod q                       | n − m + 1 shifts is valid,
    t_0 ← (dt_0 + T[i]) mod q                   | the verification will take
                                                | Θ((n − m + 1)m) time
                                                | (m characters per
                                                | iteration)
for s ← 0 to n − m
  if p = t_s
    then if P[1..m] = T[s + 1..s + m]
           then print "Pattern occurs
           with shift s"
           if s < n − m
             then t_{s+1} ← (d(t_s − T[s + 1]h)
                    +T[s + m − 1]) mod q
end
```

The running time of RABIN-KARP is $\Theta((n-m+1)m)$ in the worst case since we verify **every** valid shift. Here $p = t_s$ is always true.

B) Average case analysis:
RABIN-KARP$T, P, d, q)$ has an expected time of $O(n+m)$ plus the time required to process spurious hits (when we expect few valid shifts - perhaps as low as $O(1)$. Let us assume that reducing values modulo q is like a random mapping from Σ^* to Z_q $\quad (Z_q = \{0, ..., q-1\})$. Then the expected number of spurious hits is $O(\frac{n}{q})$ since the probability that for any given t_s we have: $t_s = p \ mod \ q$ is $\frac{1}{q}$. So the expected running time is:
$\underbrace{O(n)}_{without\ spurious\ hits}$ $+O(m(v + \frac{n}{q}))$, where s is the number of pattern matches.
Note that if $q \geq$ (q is larger than P) and the expected number of the valid shifts is small ($O(1)$ let's say), then RABIN-KARP runs in $O(n+m)$ time. □

The prime number q should be as large as possible, consistent with $(|\sum| + 1)q$ not causing arithmetic overflow. Sedgewick uses a value of: $q = 33554393$ for $|\Sigma| = 32$ [see Algorithms, Addison-Wesley, 1983]. Pirklbauer uses a value of: $q = 8355967$ for $|\Sigma| = 256$ [see "A study of pattern-matching algorithms" in Structured programming, Volume 13, 1992]. Gonnet and Baeza-Yates present an implementation of the algorithm that goes away with the mod operations by virtue of the implicit modular arithmetic of the target hardware. Spurious hits are also known as signature collisions. A more recent implementation of the algorithm reduces the probability of a large number of collisions by randomly reselecting the prime q (after a collision), reinitializing, and then continuing with the search.

3.5 Boyer-Moore

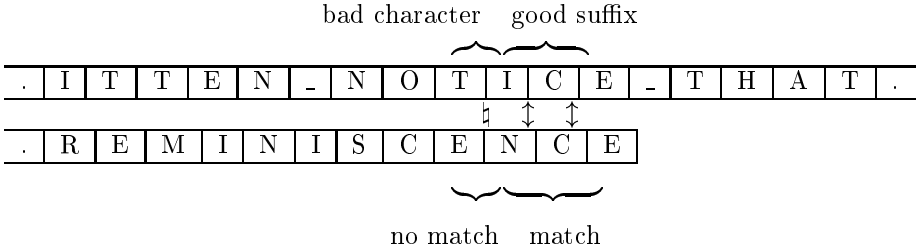
As is the case with KMP, the efficiency of BOYER-MOORE depends on determing the location of the next matching attempt, given that the previous attempt was unsuccessful. Once again, following a mismatch at the $q+1$ position of the pattern,

the preceding q symbols of the pattern and their structure give insight as to where the next matching attempt should begin.For the KMP and for the BOYER-MOORE algorithm, knowing how far to shift the pattern is the most important factor. In both algorithms, the pattern is preprocessed to determine the shift for a failure at the q^{th} symbol of P for each $q = 1, 2, ..., m$. BOYER-MOORE is more efficient than KNUTHMORRISPRATT algorithm, on the average, especially when the pattern P is long and the alphabet $\sum$ is large.

As with KMP, we slide P along T from left to right, checking corresponding characters. But with the BOYER-MOORE algorithm (BM), the characters of P are checked right to left after each movement of the pattern. The information gained at the end of the pattern (i.e., on the right) often allows the algorithm to proceed in large jumps through the text being searched. Thus BM has the unusual property that, in most cases, not all of the first q characters of T are inspected. The number of characters actually inspected (on the average) decreases as a function of the length of the pattern P.

BM uses two heuristics that allow it to avoid much of the work that KMP performs. The heuristics are: the bad-character heuristic and the good suffix heuristic. The heuristics allow BM to skip the examination of many text characters. When a mismatch occurs, each heuristic proposes an amount by which the pattern can be safely shifted without missing a match. BM chooses the larger amount of the two and shifts the pattern by that amount. The two heuristics can be viewed as operating independly in parallel.

EXAMPLE:



Shift s is invalid; although

i) a "good suffix" $\subset E$ of P matched in T, we have:

ii) a "bad character" in T, which did not match pattern character n □

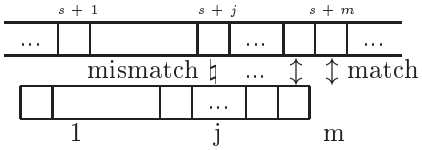
3.5.1 The Bad-Character Heuristic

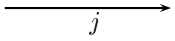
Suppose we have just found a mismatch: $P[j] \neq T[s + j]$ for some $j, 1 \leq j \leq m$. Let k be the largest index, $1 \leq k \leq m$, such that $T[s + j] = P[k]$, if any such k exists. If such k does not exist (if the pattern does not contain a character such that $T[s + j] = P[k], 1 \leq k \leq m$) then let $k = 0$.

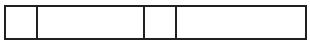
Claim: Increase s by $j - k$. We have three cases to consider:

- Case one: $k = 0$
- Case two: $k < j$
- Case three: $k > j$

Case one: $k = 0$

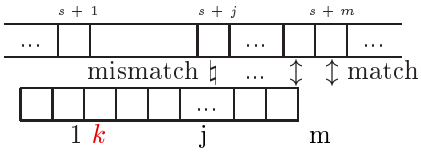


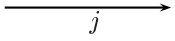




Here, the bad character, $T[s+j]$ is not found in p. $k = 0$. So we shift p until it has passed over the bad character) (i.e., just to the right of the bad character). We increase s by $j - k = j$.

Case two : $k < j$



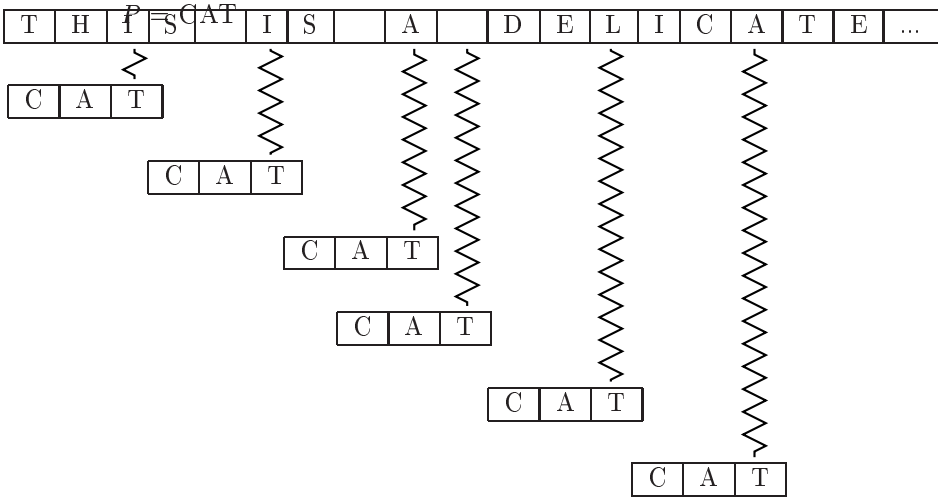


The rightmost occurrence of the bad character is in the pattern to the left of position j.
$T[s+j] = P[k]$
We slide p to the right by $j - k$ characters to align $T[s+j]$ with $P[k]$. Since k is the largest index, $1 \le k \le m$, we can safely increase s by $j - k$ without missing any valid shifts.

Case three: $k > j$

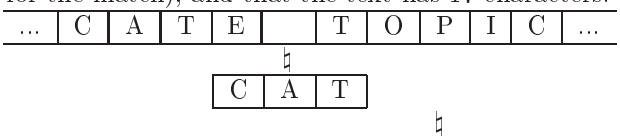
Here $j - k < 0$, and so the bad-character heuristic is proposing to decrease s. BM will ignore this recommendation and will choose the recommendation suggested by the good-suffix heuristic.

EXAMPLE: T = THIS IS A DELICATE TOPIC



and here we find a match: C A T

Note that we have made 9 comparisons up to this point (the 6 mismatches shown +3 for the match), and that the text has 17 characters.



C	A	T

□

COMPUTE-LAST-OCCURRENCE$(P, m, \sum)$ computes the rightmost position in the pattern P (of length m) at which each character of P occurs. The results are saved in λ. In other words, $\lambda[a]$ is the index of the rightmost position at which a occurs for each $a \in \sum$.

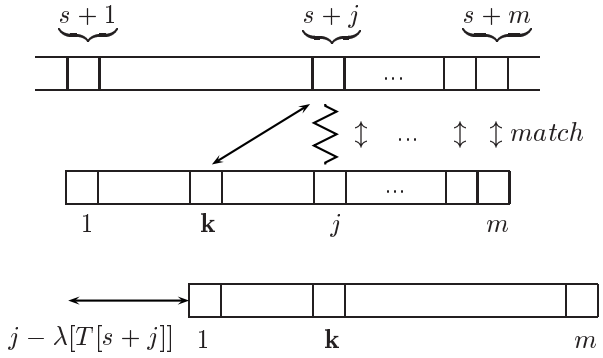
```
COMPUTE-LAST-OCCURRENCE(P, m, Σ)
for each character a ∈ Σ
    λ[a] ← 0
for j ← 1 to m
    λ[P[j]] ← j
return λ
```

The running time of COMPUTE-LAST-OCCURRENCE is: $O(|\sum| + m)$.

EXAMPLE: $P = reminiscence$

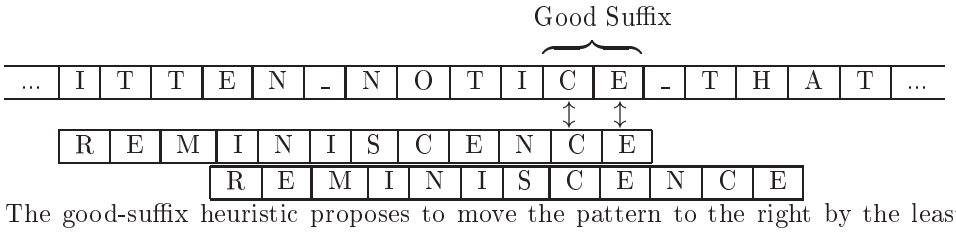
a	$\lambda[a]$
r	1
e	2 9 12
m	3
i	4 6
n	5 10
s	7
c	8 11

Recall that BM is interested in λ in case two and case three. In case two, we increase s by $j - k$ where k is the largest index, $1 \leq k \leq m$, where $T[s + j] = P[k]$. In other words, $k = \lambda[T[s + j]]$. So the bad-character heuristic will recommend to increase s by $j - \lambda[T[s + j]]$.



As for case three, $j - k < 0$, i.e. $j - \lambda[T[s + j]]$ since the bad character $T[s + j]$ occurs to the right of position j. In this case, BM relies on the recommendation given by the good-suffix heuristic (which is always positive).

3.5.2 The Good-Suffix Heuristic



The good-suffix heuristic proposes to move the pattern to the right by the least amount that guarantees that any pattern characters that align with the good suffix

previously found in the text will match those suffix characters. So in out example, we move p 3 characters to the right to align CE of T with the new CE of P. The shifts are computed by procedure COMPUTE-GOOD-SUFFIX and are saved in γ. If $P[j] \neq T[s+j]$, with $j < m$, then the good-suffix heuristic assures that we can safely advance s by $\gamma[j]$ where: $\gamma[j] = m - max\{k | 0 \leq k < m$ and $P_k \sqsubset P[j+1..m]$ or $P[j+1..m] \sqsubset P_k$. In other words, $\gamma[j]$ is the least amount we can advance s and not cause any characters in the good-suffix: $T[s+j+1..s+m]$ to be mismatched against the new alignment of the pattern.

EXAMPLE:

In the previous example: $T[s+j] = \text{I} \neq \text{N} = P[j]$

The good suffix is: $T[s+j+1..m] = \text{CE}$

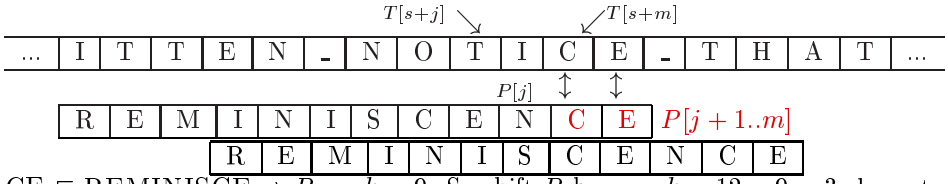
$P = \text{REMINISCENCE}$

CE is a suffix of REMINISCE

So $P_q = \text{REMINISCE}$ and $k = 9$

So shift by $m - k = 12 - 9 = 3$ characters.

EXAMPLE:



CE $\sqsubset$ REMINISCE $\Rightarrow P_q$ so $k = 9$. So shift P by $m - k = 12 - 9 = 3$ characters

1	2	3	4	5	6	7	8	9	10	11	12
R	E	M	I	N	I	S	C	E	N	C	E

We just showed that $\gamma[10] = 3$ since $\gamma[10] = 12 - max\{9\}$

Note that:

$\gamma[11] = 12 - max\{k | E \sqsubset P_k$ or $P_k \sqsubset E\}$

$E \sqsubset RE$ and $E \sqsubset REMINISCE$

$\Rightarrow E \sqsubset P_2$ and $E \sqsubset P_q$

So $\gamma[11] = 12 - max\{2, 9\} \Rightarrow \gamma[11] = 3$

□

It can be shown that:

$\gamma[j] = m - max\{k | 0 \leq k < m$ and $P_k \sqsubset P[j+1..m]$ or $P[j+1..m] \sqsubset P_k$

is equivalent to:

$\gamma[j] = min(\{m - \pi[m]\} \cup \{l - \pi'[l] | 1 \leq l \leq m$ and $j = m - \pi'[l]\})$;

where π' is the prefix function of P', the reverse of P.

Procedure COMPUTE-GOOD-SUFFIX is a straightforward implementation of the last equation.

```
COMPUTE-GOOD-SUFFIX(P, m)
π ← PREFIX-FUNCTION(P)
P' ← REVERSE(P)
π' ← PREFIX-FUNCTION(P')
for j ← 0 to m
   γ[j] ← m − π[m]
for l ← 1 to m
   j ← m − π'[l]
      if γ[j] > l − π'[l]
         then γ[j] ← l − π'[l]
return γ
```

EXAMPLE:

P	R	E	M	I	N	I	S	C	E	N	C	E
π	0	0	0	0	0	0	0	0	0	0	0	0
P'	E	C	N	E	C	S	I	N	I	M	E	R
π'	0	0	0	1	2	0	0	0	0	0	1	0

j	R	E	M	I	N	I	S	C	E	N	C	E
$\gamma[j]$	12	12	12	12	12	12	12	12	12	3	3	1

The running time of COMPUTE-GOOD-SUFFIX is $O(m)$. □

Having covered the bad character procedure and the good suffix procedure, we are now ready to tackle the BOYER-MOORE algorithm.

```
BOYER-MOORE(T, P, ∑)
n ← length of T
m ← length of P
λ ← COMPUTE-LAST-OCCURENCE(P, m, ∑)
γ ← COMPUTE-GOOD-SUFFIX(P, m)
s ← 0
while s ≤ n − m
  j ← m
    while j > 0 and P[j] = T[s + j]
      j ← j − 1
    if j = 0
      then output "Pattern occurs at shift" s
        s ← s = γ[0]
      else s ← s + max(γ[j], j − λ[T[s + j]])
end
```

3.5.3 Analysis of Boyer-Moore

```
BOYER-MOORE(T, P, ∑)
n ← length of T
m ← length of P
λ ← COMPUTE-LAST-OCCURENCE(P, m, ∑)        O(|e| + m)
γ ← COMPUTE-GOOD-SUFFIX(P, m)              O(m)
s ← 0
while s ≤ n − m                            O(n − m + 1)
  j ← m
    while j > 0 and P[j] = T[s + j]        O(m)
      j ← j − 1
    if j = 0
      then output "Pattern occurs at shift" s
        s ← s = γ[0]
      else s ← s + max(γ[j], j − λ[T[s + j]])
end
```

The worst-case running time of the BOYER-MOORE algorithm is $O((n-m+1)m + |\sum|)$. In practice, BM performs better than other string matching algorithms.

3.5.4 Longest Common Subsequence

Suppose we have the same problem as before, a text T of length n and a pattern P of length m. We would like to know if the letters of P appear in order (but possibly separated) in T. If they do, we say that P is a subsequence of Y.
EXAMPLE:

- T = POLYTECHNIC INSTITUTES

- P = TENNIS

Is P a subsequence of T? Yes.

POLYTECHNIC INSTITUTES □

What if the pattern does not occur in T? It makes sense to ask for the longest subsequence that occurs both in the pattern as well as in the text. This is the common subsequence problem. Note that here the text and the pattern have similar roles (or symmetric roles), so that we don't need to call them "text" and "pattern" anymore. Instead, call them $X =< x_1, x_2, ...x_m >$ and $Y =< y_1, y_2, ..., y_n >$.

There are several areas where we would want to solve the longest common subsequence problem:

1) molecular biology: DNA sequences (genes) can be represented as sequences of 4 letters: A, C, G, T, corresponding to the 4 submolecules forming the DNA. When biologists find a new sequence, they typically want to know what other sequences it is most similar to. One way of computing how similar two sequences are is to find the length of their longest common subsequence.

2) file comparison: The UNIX program "diff" compares 2 different versions of the same file, to determine what changes have been made to the file. It works by finding a longest common subsequence of the lines of the 2 files. Note: Any line in the subsequence has not been changed. In other words, what is displayed is indeed the remaining set of lines (that have changed). Here it is convenient to think of each line of a file as being a single character in a string.

3) screen redisplay:
Many text editors, like "Emacs", display part of a file on the screen, updating the screen image as the file is changed. For slow–in terminals, these programs want to send the terminal as few characters as possible to cause it to update its display correctly. Here, the common subsequence is the part of the display that has been corrected and does not not need to be changed. □

To solve the longest common subsequence, we use dynamic programming (recall the string alignment problem and the edit problem). Procedure LCS(X, Y) takes 2 sequences $X =< x_1, x_2, ..., x_m >$ and $Y =< y_1, y_2, ..., y_n >$ as input. It saves the $c[i, j]$ values in a table $c[0..m, 0..n]$ whose entries are computed in a row-major order, where $c[i, j]$ is the length of an LCS of the sequences X_i and Y_j. If either $u = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. In general, we have:

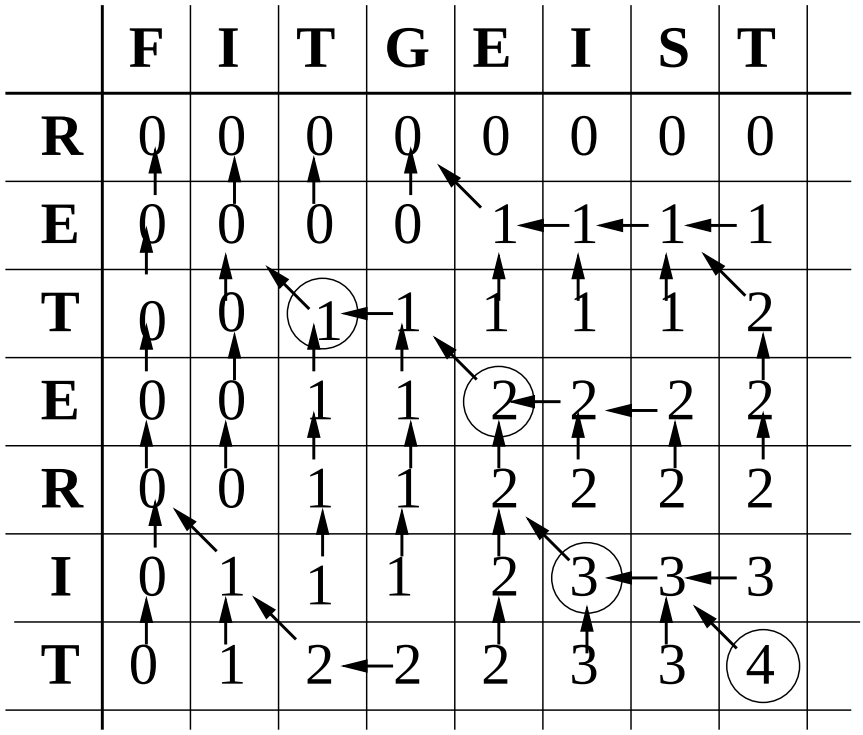
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i > 0, j > 0 \text{ and } x_i = y_j \\ max(c[i, j-1].c[i-1, j]) & \text{if } i > 0, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Procedure LCS(X, Y) also maintains a table $b[1..m, 1..n]$ to simplify the construction of an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$. Procedure LCSX, Y returns the b and c tables, and $c[m, n]$ contains the length of an LCS of X and Y. Note that the elements for which we have " $\nwarrow$ " are the characters that belong to an LCS. Also note that LCS(X, Y) could have been designed without the explicit use of table b.

```
LCS(X, Y)
m ← length of X
n ← length of Y
for  i ← 0 to m
  c[i, 0] ← 0
for  j ← 0 to n
  c[0, j] ← 0
for  i ← 1 to m
  for  j ← 1 to n
    if  x_i = y_j
      then  c[i, j] ← c[i − 1, j − 1] + 1
        b[i, j] ← " ↖"
        else if c[i, j] ≥ c[i − 1, j]
          then  c[i, j] ← c[i − 1, j]
            b[i, j] ← " ↑"
          else  c[i, j] ← c[i, j − 1]
            b[i, j] ← " ←"
return  b and c
```

The running time of LCS(X, Y) is $O(mn)$,
since we are assuming that each table entry takes $O(1)$ time to compute.

EXAMPLE:



3.5.5 More Applications: Computer Viruses

1) Computer Viruses

- From: http://www.mactech.com/articles/mactech/
  Vol.08/08.02/VirusDetection/

- Recently, there has been an explosion in the development rate of computer virus code.
- Computer viruses have attracted the attention of the public, the media, and the scientific community.
- Detection of the virus is simply considered a pattern string to be searched for in a larger text (a possibly infected program).

There are 3 categories of anti-viral products

- Infection Prevention Products: halt the virus replication and prevent the initial infection from occurring. Example: cdev (Control Panel Device) Vaccine on Macintosh.
- Infection Detection Products: detect infection right after it has occurred and mark specific components of system segments that have become infected. It is done by periodically inspecting executable files, and may or may not attempt repair of an infected file. Examples: INIT package GateKeeper and GateKeeper Aid on Macintosh.
- Infection Identification Products: identify specific viral strains on systems that are already infected and may remove the virus, returning the system to its state prior to infection. This antivirus is the most common and its effectiveness dependents on the frequency of the user invocation. Examples: McAfee's MS-DOS program SCAN and the Macintosh Disinfectant series.

Remarks

- Their anti-virus package uses a modification of the Boyer-Moore algorithm. More specifically, they use a randomized system of string selection.
- Work in string matching, like work in virus detection, is by no means complete.
- Modern algorithms make use of parallel hardware and improved data structures, such as suffix trees (which may be respectively applied to randomized matching and dynamic programming).

2) Detecting Software Plagiarism Systems

- A) From: http://wwwipd.ira.uka.de/jplag/
- JPlan can detect software plagiarism.
- JPlag is a system that finds similarities among multiple sets of source code files.
- JPlag currently supports Java, C, C++, and Scheme.
- B) Moss (**M**easure **O**f **S**oftware **S**imilarity) is an automatic system for determining the similarity of C, C++, Java, Pascal, Ada, ML, Lisp, or Scheme programs.

1) Common Subsequences

- From: "The Complexity of Some Problems on Subsequences and Supersequences", by D. Maier.
- $S = s_1 s_2 ... s_m$ is a sequence with m elements; $|S| = m$.
  S' is a subsequence of S, $S' < S$, if it is a sequence which consists of S with a number of terms between 0 & m deleted.

- $R = \{S_1, ..., S_p\}$ is a set of sequences with alphabet A, i.e., the set of values the different $s_i's$ can take.
- LCS(R), the longest common subsequence of R, is the longest sequence S, such that $S < S_i$ for i=1,2,...,p.
- Similarly, SCS(R), the shortest common sequence of R, is the shortest sequence S', such that $S_i < S'$ for i=1,2,...,p.
- Example If R={ababe, cabe, abdec} then
  (a) A = {a,b,c,d,e}, and $|A| = 5$,
  (b) LCS(R) = abe, and
  (c) SCS(R) = cabeabdec.

3.5.6 The Longest Upsequence [by E.W.Dijkstra]

Define an upsequence of length k in an array $L = L_1 L_2 \cdots L_n$ of numbers to be a sequence $i_1, \ldots, i_k$ with $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, for which $L_{i_1} \leq L_{i_2} \leq \cdots \leq L_{i_k}$, i.e., to be a not necessarily contiguous subsequence of L which is non-decreasing from left to right.
The problem, given an array L, is to find the greatest length of any upsequence which occurs in L. For example, if L is sorted in decreasing order, the answer is 1. Note that for any prefix $B = L_1 \cdots L_m$ of L $(0 \leq m \leq n)$, and for any length k of upsequence which occurs at all in B, define the record economical top element (RETE) with respect to k and B to be the smallest number which is the last (i.e., largest) member of an upsequence of length k in B. Proceed from left to right, letting B grow from the empty sequence to all of L, and at all times keep track of all the RETEs which exist. At the end, the biggest k for which a RETE exists will be the answer.
As we consider each of the numbers $L_1, \ldots, L_n$ in turn, we will find that it is a new RETE in exactly one way. For example, suppose that $L_1 = 3$, $L_2 = 5$, $L_3 = 2$, and $L_4 = 7$. RETE w.r.t B and k:

$B = 3$		RETE	UPSEQUENCES
	1-upsequence	3	(3)
$B = 35$			
	1-upsequence	3 (3)	
	2-upsequence	5	(3,5)
$B = 352$			
	1-upsequence	2	(2) [since 2 is smaller than 3]
	2-upsequence	5	(3,5)
$B = 3527$	1-upsequence	2 (2)	
	2-upsequence	5	(3,5)
	3-upsequence	7	(3,5,7)

Note that 2 and 5 and 7 are RETEs for one subsequence (each). $L[5]$ will also be a new RETE in exactly one way.

Chapter 4

Approximation Algorithms

An algorithm that returns near-optimal solutions is called an approximation algorithm.

If C^* denotes the optimal solution of an optimization problem, and C the solution obtained by an approximation algorithm for the same problem, then

if $max\{\frac{C}{C^*}, \frac{C^*}{C}\} \leq \rho(n) \quad \forall n$ instances
then $\rho(n)$ is a ratio bound.

We are assuming that all $C's$ are positive.
For maximization problems : $0 < C \leq C^*$,
for minimization problems : $0 < C^* \leq C$.

Note:

1. i) $\rho(n) \geq 1$ since $max\{\frac{C}{C^*}, \frac{C^*}{C}\} \geq 1$
2. ii) $\rho(n) = 1 \Leftrightarrow C = C^*$.

The relative error of the approximation algorithm is $\frac{|C-C^*|}{C^*}$.
An approximation algorithm has a relative error bound of $\epsilon(n)$ if: $\frac{|C-C^*|}{C^*} \leq \epsilon(n)$.

Fact: $\epsilon(n) \leq \rho(n) - 1$

Our goal is to find approximation algorithms that yield very small ration bounds. For some problems, approximation algorithms have been developed that have fixed ratio bounds, independent of n.

4.1 The vertex cover problem

Given a graph $G = (V, E)$ find a subset $V' \subseteq V$ such that if $(u, v) \in E$, then either $u \in V'$ or $v \in V'$, of smallest size.

	APPROX-VERTEX-COVER(G)
1	$C \leftarrow 0$ {C contains vertex cover being constructed}
2	$E' \leftarrow E[G]$
3	**while** $E' \neq 0$
4	**do** let (u, v) be an arbitrary edge of E'
5	$C \leftarrow C \cup \{u, v\}$
6	remove from E' every edge incident on either u or v
7	**return** C

The algorithm has running time $O(E)$.

Theorem 4.1.1 *Approx-Vertex-Cover has a ratio bound of 2.*

Proof: We show that $\frac{|C|}{|C^*|} \leq 2$.

First note that the algorithm returns a vertex cover C since it loops till every edge in $E[G]$ has been covered by some vertex in C.

Let A be the set of edges that are chosen in step 4 above. No two edges in A share an endpoint, since right after picking an edge, we remove from E' all other edges that are incident on its endpoints (see step 6).

So each execution of step 5 adds two new vertices to C, and $|C| = 2|A|$. No edges of A are adjacent. In other words, any cover contains one vertex for each edge in A.

So $|C^*| \geq |A|$ [since C^* is a cover]
Hence, $(|C| = 2|A|) \wedge (|C^*| \geq |A|) \Rightarrow |C^*| \geq \frac{|C|}{2}$
i.e. $\frac{|C|}{|C^*|} \leq 2$ □

4.2 The Traveling-Salesperson Problem

A complete graph $G = (V, E)$ is given that has nonnegative integer costs $C(u, v)$ for the edges $(u, v) \in E$. We want to find a Hamiltonian cycle (or tour) of G with minimum cost. Let $c(A) = \sum_{(u,v)\in A} c(u, v)$ where $A \subseteq E$. Cost c satisfies the triangular inequality if $c(u, w) \leq c(u, v) + c(v, w)$ for all vertices $u, v, w \in V$.

A) The **TSP** with triangular inequality

	APPROX-TSP-TOUR(G, c)
1	Select a vertex $r \in V[G]$ to be the root
2	Use Prim's algorithm to construct a $MSTT$ using r as root
3	Traverse the tree in preorder to form the list of visited vertices L
4	Return the Hamiltonian cycle H that visits the vertices as specified in L.

□The running time of APPROX-TSP-TOUR is $\Theta(E) = \Theta(V^2)$ since the input is a complete graph.

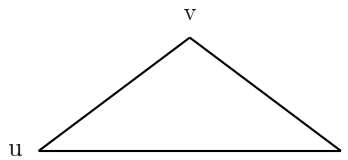
The algorithm returns a tour whose cost is not more than twice the cost of an optimal tour.

Theorem 4.2.1 APPROX-TSP-TOUR *is an approximation algorithm with a ratio bound of 2 for the* ***TSP*** *with triangular inequality.*

Proof: H^* denotes an optimal tour. We want to show that $\frac{c(H)}{c(H^*)} \leq 2$, where H is obtained by the approximation algorithm.
Note: $c(T) \leq c(H^*)$ since **MST** T is obtained by deleting an edge from a tour.

A full walk of T, denoted by W, lists the vertices of T when they are first visited and then when they are returned to after visiting the subtree. Thus W traverses every edge of T exactly twice.

So $c(W) = 2c(T)$ therefore $c(W) \leq 2c(H^*)$. So the cost of W is within a factor of 2 of the cost of an optimal tour. W is generally not a tour. By the triangular inequality, vertices can be deleted from W without having the cost increased.



If v is deleted from W between visits to u and w, the result is having to go directly from u to w.

We can thus systematically remove every vertex that has already been visited in W. We end up with a preorder walk of T. Let H be the cycle associated with that walk. It is a Hamiltonian cycle (every vertex is visited once), and it is the cycle obtained by the approximation algorithm.

So $c(H) \leq c(W)$ $\quad$ [H has less vertices than W]

Recall: $c(W) \leq 2c(H^*)$ therefore $\frac{c(H)}{c(H^*)} \leq 2$. □

B) The general **TSP**

Theorem 4.2.2 *If $P \neq NP$ and the ratio bound $\rho \geq 1$, then there exists no polynomial time approximation algorithm with ration bound for the general **TSP**.*

Proof: Proof by contradiction.

Suppose that for some $\rho \geq 1, \exists$ a polynomial time approximation algorithm A with ratio bound ρ. Without loss of generality assume ρ is an integer. We show, that we can use A to solve **Ham-Cycle** in polynomial time.

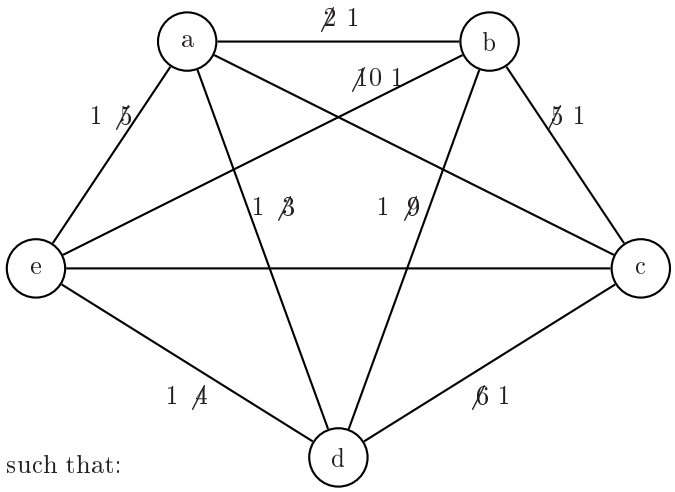
Let $G = (V, E)$ be an instance of **Ham-Cycle**. Convert G into an instance of the **TSP**. Let $G' = (V, E)$ be the complete graph on V: i.e. $E' = \{(u, v) | u, e \in V \text{ and } u \neq v\}$. Assign integer costs to edges of E':

$$c(u, v) = \begin{cases} 1 & \text{if} (u, v) \in E \\ \rho|V| + 1 & \text{otherwise} \end{cases}$$

The conversion from G to G' is in polynomial time in $|V|$ and $|E|$. Consider the **TSP**(G', c). If G has a Hamiltonian circuit H, then (G', c) contains a tour of cost $|V|$ consisting of the edges from H. Recall $c(u, v) = 1$ if $(u, v) \in E$. If G does not contain a Hamiltonian circuit then any tour H' of G' must use some edge not in E. So $c(H') \geq (\rho|V| + 1) + (|V| - 1)$ i.e. $c(H') > \rho|V|$ So a tour on G that is a Hamiltonian circuit costs $|V|$, any other tour costs $> \rho|V|$. Apply A to **TSP** given by (G', c). A returns a tour of cost no more than ρ times the cost of an optimal tour. So if G contains a Hamiltonian circuit, A returns it. If G has no Hamiltonian circuit, then A returns a tour of cost more than $\rho|V|$. So A can be used to solve the HAM-CYCLE in polynomial time. □

So our assumption is incorrect and such an algorithm A cannot exist.

EXAMPLE:



Edges in green are such that:

$c(a,c) = c(e,c) = \rho|V| + 1 = \rho(5) + 1 = 5\rho + 1$ A Hamiltonian cycle of an undirected graph $G = (V, E)$ is a simple cycle that contains each vertex in V. A graph that contains a Hamiltonian cycle is a Hamiltonian graph.

4.3 The Set-Covering Problem

An instance (X, F) of the set-covering problem consists of a finite set X and a family F of subsets of X, such that

$$X = \bigcup_{S \in F} S$$

i.e. every element of X belongs to at least one subset in F. $S \in F$ covers its elements. We are interested in finding a minimum-size subset $C \subseteq F$ whose members cover all of X:

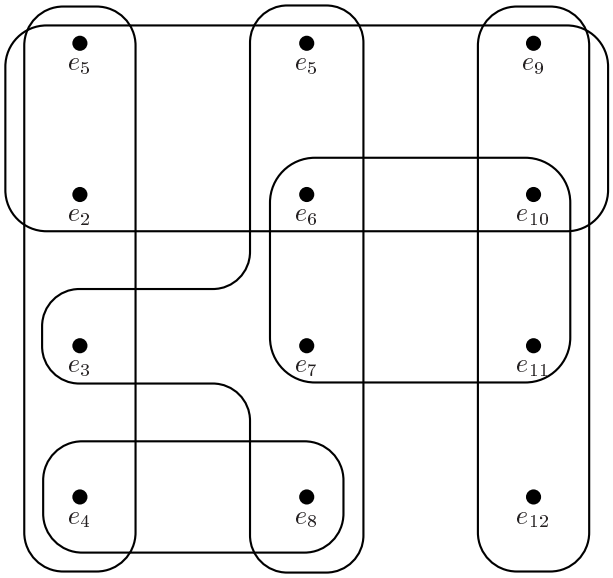
$$X = \bigcup_{S \in C} S$$

C then covers it elements.
Greedy-set-cover picks, at each iteration, the subset that covers as many uncovered elements as possible.

The algorithm can be implemented to run in polynomial time in $|X|$ and $|F|$. The number of iterations of the loop [line 3-6] is at most $min(|X|, |F|)$. The loop body can be implemented to run in $O(|X||F|)$ time. So the algorithm can run in $O(|X||F|min(|X|, |F|))$.

Greedy-Set-Cover(X, F)	
$U \leftarrow X$	*U: set of uncovered elements*
$C \leftarrow \emptyset$	*C: contains cover being constructed*
while $U \neq \emptyset$	
do select $S \in F$ that maximizes $\|S \cap U\|$	
$U \leftarrow U - S$	*choose subset that contains the maximum number of uncovered elements.*
$C \leftarrow C \cup \{S\}$	
return C	

Example:



An instance (X, F) of the set-covering problem, where X consists of 12 black points and $F = \{S_1, S_2, S_3, S_4, S_5, S_6\}$. A minimum-size set cover is $C = \{S_3, S_4, S_5\}$. The greedy algorithm however produces a cover of size 4 by selecting the sets S_1, S_4, S_5 and S_3 in that order.

4.3.1 Analysis of Greedy-Set-Cover

Theorem 4.3.1 GREEDY-SET-COVER *has a ratio bound* $H(max\{|S| : S \in F\})$ *where*

$$H(d) = \sum_{i=1}^{d} \frac{1}{i}$$

is the d^{th} *harmonic number.*

Proof: Let C^* denote an optimal set cover and S_i the i^{th} subset picked by the algorithm and C the set cover constructed by the algorithm.

Spread the cost of selecting S_i evenly among the elements covered for the first time by S_i.

Each element $x \in X$ is assigned a cost C_x when it is covered for the first time:

$$C_x = \frac{1}{|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1})} \; [x \text{ is covered by } S_i]$$

In the EXAMPLE:
$c_{e_1} = c_{e_2} = c_{e_5} = c_{e_6} = c_{e_9} = c_{e_10} = \frac{1}{|S_1|} = \frac{1}{6}$
$c_{e_3} = c_{e_7} = c_{e_8} = \frac{1}{|S_4 - S_1|} = \frac{1}{3}$
$c_{e_4} = c_{e_{12}} = \frac{1}{|S_5 - (S_1 \cup S_4)|} = \frac{1}{2}$
$c_{e_4} = \frac{1}{|S_3 - (|S_1 \cup S_4 \cup S_5)|} = 1$ □

If C is returned by the algorithm, then

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

is the number of sets in C

$$\Rightarrow |C| \leq \sum_{S \in C^*} \sum_{x \in S} c_x$$

The proof is based on:

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

which we prove later.

So

$$\begin{aligned} |C| &\leq \sum_{S \in C^*} \sum_{x \in S} c_x \\ &\leq \sum_{S \in C^*} H(|S|) \\ &\leq |C^*| H(max\{|S| : S \in F\}) \\ \Rightarrow \frac{|C|}{|C^*|} &\leq H(max\{|S| : S \in F\}) \end{aligned}$$

□

We still have to show that $\sum c_x \leq H(|T|) \quad \forall T \in F$. Let $u_i = |T - (T_1 \cup T_2 ... \cup T_i)|$ for any $T \in F$ and $i = 1, 2, ..., |C|$. u_i is the number of elements in T remaining uncovered after $T_1 T_2 ... T_i$ have been selected by the algorithm. Let $u_0 = |T|$ and k e the least index such that $u_k = 0$, so that every element in T is covered by at least one of the sets $T_1 T_2 ... T_k$. In the EXAMPLE: $T_1 = S_1, T_2 = S_4, T_3 = S_5$ and $T_4 = S_3$ [order in which sets are chosen]. If $T = S_6$ then $u_0 = |S_6| = 2$.
$u_1 = |T - T_1| = |S_6 - S_1 = |\{e_4, e_\}| = 2$
$u_2 = |T - (T_1 \cup T_2)|| = |S_6 - (S_1 \cup S_4)| = |\{e_4\}| = 1$
$u_3 = |T - (T_1 \cup T_2 \cup T_3)| =_6 -(S_1 \cup S_4 \cup S_5)| = |\{e_4\}| = 1$
$u_4 = |T - (T_1 \cup T_2 \cup T_3 \cup T_4)| = |S_6 - (S_1 \cup S_4 \cup S_5 \cup S_3)| = 0$
So $k = 4$.
Note that:
$u_0 - u_1 = 0$ elements of S_6 are covered for the first time by S_1
$u_1 - u_2 = 1$ element of S_6 is covered for the first time by S_4
$u_2 - u_3 = 0$ elements of S_6 are covered for the first time by S_5
$u_3 - u_4 = 1$ element of S_6 is covered for the first time by S_3
So, $u_{i-1} \geq u_i$ and $u_{i-1} - u_i$ elements of T are covered for the first time by T_i, for $i = 1, 2, ..., k$. So

$$\sum_{x \in T} c_x = \sum_{i=1} k(u_{i-1} - u_i) \frac{1}{|T_i - (T_1 \cup T_2 \cup ... \cup T_{i-1})|}$$

Note that: $|T_i - (T_1 \cup T_2 \cup ... \cup T_{i-1}) \geq |T - (T_1 \cup T_2 \cup ... \cup T_{i-1})|$ because the algorithm is greedy and chooses T_i over T. But $u_{i-1} = T - (T_1 \cup T_2 \cup ... \cup T_{i1})|$. So

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

Note: For integers a, b with $a < b$:

$$\begin{aligned} H(b) - H(a) &= \frac{1}{a+1} + \frac{1}{a+2} + ... + \frac{1}{b} \\ H(b) - H(a) &= \sum_{i=a+1}^{b} \frac{1}{i} \\ &\geq \underbrace{\frac{1}{b} + ... + \frac{1}{b}}_{b-a \text{terms}} \end{aligned}$$

$$\Rightarrow H(b) - H(a) \geq (b - a) \frac{1}{b}$$

Back to

$$\begin{aligned}\sum_{x\in T} c_x &\leq \sum_{i=1}^{k}(H(u_{i-1}) - H(u_i)) \\ &= H(u_o) - H(u_1) + H(u_1) - H(u_2) - ... + H(u_{k-1} - H(u_k) \\ &= H(u_0) - H(u_k)\end{aligned}$$

But $u_0 = |T|$ and $u_k = 0$

$$\Rightarrow \sum_{x\in T} c_x \leq H(|T|)$$

□

Fact:

$$\text{Since } \sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 \text{ we have } \frac{|C|}{|C^*|} \leq \ln |x| + 1$$

□

4.4 The Subset-Sum problem

Given(S, t) where $S = \{x_1, ..., x_n\}, x_i \in Z^+$ and $t \in Z^+$, can we find a subset S' of S, whose elements add up to the target value t?

In the optimization problem, we want to find a subset S' whose sum is as large as possible but not larger than t.

An exponential time algorithm

If $L =< l_1, l_2, ..., l_m >$ is a list of numbers, $L + x =< l_1 + x, l_2 + x, ..., l_m + x >$.
Similarly, if $S = \{x_i | x_i \in Z^+, 1 \leq i \leq n\}$
then $S + x = \{x_i + x | x_i \in S \wedge x \in Z^+\}$.

Recall that given 2 sorted lists L and L', we can merge them (via MERGE-LISTS(L, L') for example) in $O(|L| + |L'|)$ time.

Algorithm EXACT-SUBSET-SUM(S, t) returns a subset S' of S whose sum is as large as possible but not larger than t.

EXACT-SUBSET-SUM(S, t)
$n \leftarrow \|S\|$
$L_0 \leftarrow < 0 >$
for $i \leftarrow 1$ **to** n
do $L_i \leftarrow$ MERGE-LISTS$(L_{i-1}, L_{i-1} + x_i)$
remove from L_i every element $> t$
return largest element in L_n

Let P_i be the set of all values that can be obtained by choosing a subset of $\{x_1, x_2, ..., x_i\}$ and adding its members. The subset can be Φ.

Example:

$$S = \{2, 5, 9\}$$
$$P_1 = \{0, 2\}$$
$$P_2 = \{0, 2, 5, 7\}$$
$$P_3 = \{0, 2, 5, 7, 9, 14, 16\}$$

It can be shown that:

1. $P_i = P_{i-1} \cup (P_{i-1} + x_i)$, and

2. L_i in EXACT-SUBSET-SUMS, t) is a sorted list containing every element of P_i whose sum is $\leq t$, and

3. EXACT-SUBSET-SUM(S, t) is an exponential time algorithm in general.

1. An approximation scheme for an optimization problem is an algorithm A which takes as input both:

   - the problem instance I, and
   - an error bound ϵ

   and has the performance guarantee:

$$\frac{A(I, \epsilon)}{\text{opt}(I)} \leq 1 + \epsilon$$

   Note: This is a very desirable result.

2. A polynomial approximation scheme is an approximation scheme $\{A_\epsilon\}$ where each algorithm $A_\epsilon$runs in time polynomial in the length of the input instance I.

3. A polynomial approximation scheme is an approximation scheme $\{A_\epsilon\}$ where each algorithm $A_\epsilon$runs in time polynomial in the length of the input instance I and $\frac{1}{\epsilon}$.

Remark: Having established that a certain problem is $\mathcal{NP}$–COMPLETE, the both possible algorithms for that problem would be the existence of a fully polynomial approximation scheme. Very few $\mathcal{NP}$–COMPLETE problems have fully polynomial approximation schemes.

We achieve a fully polynomial time approximation scheme for the subset-sum problem by trimming (removing elements) each list L_i after it is created.

To trim a list L by $\delta, 0 < \delta \leq 1$, means to remove as many elements from L as possible such that L', the resulting list, consists of elements z that r̈epresent ẗhe removed elements $y \in L$. More precisely every $y \in L$ that is removed, is represented by a $z \in L'$ such that $\frac{y-z}{y} > \delta$ (or equivalently: $y - z \leq \delta y$ and therefore: $(1 - \delta)y \leq z \leq y$)

Each removed $y \in L$ is represented by a $z \in L'$ such that the relative error of z, w.r.t. y, is at most δ.

For EXAMPLE: Suppose $L =<10, 11, 12, 15, 20, 21, 22, 23, 24, 29>$ and $\delta = 0.1$.

$10 \in L'$	(smallest element always in L')
$\frac{11-10}{11} = \frac{1}{11} \leq 0.1$	so drop 11 (it is represented by 10)
$\frac{12-10}{12} = \frac{2}{12} > 0.1$	include 12 in L' (cannot be represented by 10)
$\frac{15-12}{15} = \frac{3}{15} > 0.1$	so $15 \in L'$
$\frac{20-15}{20} > 0.1$	so $20 \in L'$
$\frac{22-20}{22} \leq 0.1$	so $22 \notin L'$
$\frac{23-20}{23} > 0.1$	so $23 \in L'$
$\frac{24-23}{24} \leq 0.1$	so $24 \notin L'$
and finally	
$\frac{29-23}{29} > 0.1$	so $29 \in L'$
	$\Rightarrow L' =< 10, 12, 15, 20, 23, 29 >$

Procedure TRIM(L, δ) trims $L =< y_1, \ldots <_m >$ in time $\Theta(m)$. It assumes that L is sorted in nondecreasing order.

It returns L', a trimmed and sorted list.

```
TRIM(L, δ)
m ← |L|
L' ←< y_1 >                          { most recent element to join L' }
for i ← 2 to m
  do if  last < (1 − δ)y_i           { last cannot represent y_i }
    then append y_i onto L'
         last ← y_i
return L'
```

We make use of TRIM (L, δ) to construct an approximation scheme.

```
APPROX-SUBSET-SUM(S, t, ε)
n ← |S|
L_0 ←< 0 >
for i ← 1 to n
  do L_i ← MERGE-LISTS(L_{i−1}, L_{i−1} + x_i)
     L_i ← TRIM(L_i, ε/n)
     remove from L_i elements > t
let z be the longest element of L_n
return z
```

EXAMPLE: Let $L =< 104, 102, 201, 101 >$, $t = 308$ and $\epsilon = 0.20$. Note that the optimal answer is $\{104, 102, 101\}$ and

$$\begin{array}{r} 104 \\ 102 \\ 101 \\ \hline 307 \end{array}$$

APPROX-SUBSET-SUM (S, t, ϵ) yields 302 which is $\frac{307-302}{307} = 0.016\%$ from the optimum.

EXAMPLE: $t = 308$, $\epsilon = 0.2$, $L =<104, 102, 201, 101>$. Note $L_0 \leq 0$. Trimming parameter: $\delta = \frac{\epsilon}{4} = 0.05$

i	$L_i \leftarrow$ MERGE-LISTS	$L_i \leftarrow$ TRIM	after removal
1	$L_1 =<0, 104>$	$L_1 \leftarrow<0, 104>$	$L_1 \leftarrow<0, 104>$
2	$<0, 102, 104, 206>$	$<0, 102, 206>$, because $102 \not\leq (1 - 0.05) \cdot 104$	$<0, 102, 206>$
3	merge $<0, 102, 206>$ and $<201, 303, 407> \Rightarrow$ $<0, 102, 201, 206, 303, 407>$	206 is repr. by 201, so $<0, 102, 201, 303, 407>$	drop 407 since $407 > 308$, so $<0, 102, 201, 303>$
4	merge $<0, 102, 201, 303>$ and $<101, 203, 302, 404>$ $\Rightarrow <0, 101, 102, 201, 203, 302, 303, 404>$	102 is repr. by 101 203 is repr. by 302 303 is repr. by 302 so $<0, 101, 201, 302, 404>$	remove 404 since $404 > 308$, so $<0, 101, 201, 302>$

Hence $z = 302$, while the optimal solution is, as already shown above, equal to $101 + 102 + 104 = 307$.

Theorem 4.4.1 APPROX-SUBSET-SUM *is a fully polynomial time approximation scheme for the subset-sum problem.*

Proof: We have to show that:

1. Approx-Subset-Sum is an approximation scheme
2. Approx-Subset-Sum is fully polynomial time

1. We have to show that the scheme is an an approximation algorithm with relative error bound ϵ. In other words, we have to show that $\frac{C^*-C}{C^*} \leq \epsilon$ or $C^*(1-\epsilon) \leq C$.

   Trimming L_i and removing from it ecery element greater than t is done in such a way that each element of L_i is the sum of elements of some subset of S. In particular, z, returned at the end of Approx-Subset-Sum is the sum of some subset of S. We still have to show that $z = C$ is such that $C^*(1-\epsilon) \leq z$.

   Note that if y represents y', then $\frac{y'-y}{y'} \leq \delta = \frac{\epsilon}{n}$.

   In other words, the relative error between y and y' is at most $\frac{\epsilon}{n}$.

   It can be shown, by induction on i, that for every element $y \in P_i$ where $y \leq t$, there exists a $z \in L_i$ such that:

   $$(1-\frac{\epsilon}{n})^i y \leq z \leq y$$

   In particular, if $y^* \in P_n$ denotes an optimal solution to the subset-sum problem, then there exists a $z \in L_n$ such that:

   $$(1-\frac{\epsilon}{n})^n y^* \leq z \leq y^*$$

   and the largest possible z is returned by Approx-Subset-Sum.

   Consider $\frac{d}{d_n}(1-\frac{\epsilon}{n})^n$.

   $$\begin{aligned}
   \frac{d}{d_n}(1-\frac{\epsilon}{n})^n &= n(1-\frac{\epsilon}{n})^{n-1}\frac{d}{d_n}(1-\frac{\epsilon}{n}) \\
   &= n(1-\frac{\epsilon}{n})^{n-1}(-1)(-\frac{\epsilon}{n^2}) \\
   &= n(1-\frac{\epsilon}{n})^{n-1}(\frac{\epsilon}{n^2}) \\
   &= \frac{\epsilon}{n}(1-\frac{\epsilon}{n})^{n-1}
   \end{aligned}$$

   $\frac{d}{d_n}(1-\frac{\epsilon}{n})^n > 0$ so the function

   $$q(n) = (1-\frac{\epsilon}{n})^n$$

   increases with n.

   So

   $$\begin{aligned}
   n > 1 &\Rightarrow q(n) > q(1) \\
   &\Rightarrow (1-\frac{\epsilon}{n})^n > 1-\epsilon
   \end{aligned}$$

   So $(1-\epsilon)y^* < (1-\frac{\epsilon}{n})^n y^* \leq z$ But recall: $(1-\frac{\epsilon}{n})^n y^* \leq z$
   therefore $(1-\frac{\epsilon}{n})y^* \leq z$

   In other words $\frac{y^*-z}{y^*} \leq \epsilon$
   therefore Approx-Subset-Sum is an approximation scheme.

2. It can be shown the procedure is polynomial in the length of the $L_i's$.

$$\text{HITTINGSTRING} = \{<S> \mid S = \{s_1, s_2, s_3, \ldots, s_k\} \text{ of equallength strings over } \Sigma = \{0, 1, *\} \text{ where } |s_1| = |s_2| = |s_3| = \ldots = |s_k| = n$$

$<S>$ is in HITTINGSTRING if there exists a string $y \in \{0,1\}^n$ such that for each $s_t \in S$ there is some i, $1 \leq i \leq n$, for which the i^{th} character of s_t and the i^{th} character of y are identical.
EXAMPLE:

- $S = \{0*1, 100, 110, **0\}$
  $y = 010$ is a hitting string for S. Therefore $<S> \in$ HITTINGSTRING.

s_t	$\boxed{0}*1$	$1\ 0\boxed{0}$	$1\boxed{1}0$	$**\boxed{0}$
y	$\boxed{0}1\ 0$	$0\ 1\boxed{0}$	$0\boxed{1}0$	$0\ 1\boxed{0}$

- $S = \{*01, 111, **0, 011\}$ does not allow any hitting string.
  So $<S> \notin$ HITTINGSTRING.

Claim: HITTINGSTRING is $\mathcal{NP}$–COMPLETE.
Proof:

- HITTINGSTRING $\in \mathcal{NP}$.
  y is a certificate and it takes polynomial time to check **and** verify.
- HITTINGSTRING is $\mathcal{NP}$–HARD.

$$\text{SAT} \leq_p \text{HITTINGSTRING}$$

Given $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \ldots \wedge \Phi_k$ with n different variables, we want to construct a set S_1 of size k of n–long strings over $\Sigma = \{0, 1, *\}$. To each clause Φ_p we associate an n–long string s_p according to the following rule:

x_i	$\overline{x_i}$	i^{th} symbol of s_p
$\in \Phi_p$	$\in \Phi p$	$*$
$\in \Phi_p$	$\notin \Phi p$	1
$\notin \Phi_p$	$\in \Phi p$	0
$\notin \Phi_p$	$\notin \Phi p$	$*$

EXAMPLE: $\Phi = (x_1 \vee x_2 \vee \overline{x_4}) \wedge \overline{x_3} \wedge (x_1 \vee \overline{x_2} \vee x_3 \vee x_2)$
$n = 4$, $k = 3$

$$\begin{array}{cc} \Phi_1 = x_1 x_2 \overline{x_4} & s_1 = 11*0 \\ \Phi_2 = \overline{x_3} & s_2 = *0** \\ \Phi_3 = (x_1 \overline{x_2} x_3 x_2) & s_3 = 1*1* \end{array}$$

So $S = \{11*0, *0**, 1*1*\}$. Note that Φ is satisfiable; a satisfiable assignment would be: $x_1 = x_3 = 1$ and $x_2 = x_4 = 0$. Note $y = 1010$ is a hitting string for S. The conversion technique (basically a table lookup) is done in polynomial-time (construction of all s_i's takes $O(kn)$ time).

$$\Phi \in \text{SAT} \Leftrightarrow <S> \in \text{HITTINGSTRING}$$

($\Rightarrow$) : Given $\Phi = \Phi_1 \wedge \Phi_2 \wedge \Phi_3 \wedge \ldots \wedge \Phi_k$, construct $<S>$ by the outlined method above.

$$\begin{array}{rcl} \Phi \in \text{ SAT} & \Rightarrow & \exists \text{ a satisfying assignment } A = \{0,1\}^* \\ & \Rightarrow & \text{each } \Phi_p \text{ has at least one literal, which is true} \\ & \Rightarrow & A \text{ is a hitting string for } < S > . \end{array}$$

Or more formally, construct a hitting string y according to the following rule:

* put 1 in i^{th} position of y if x_i is true in A.
* put 0 in i^{th} position of y if x_i is false in A.

Then y is a hitting string for S.

($\Leftarrow$) : Given some set $<S>$ with $<S> \in$ HITTINGSTRING, the following holds:

$$\begin{array}{rcl} < S > \in \text{ HITTINGSTRING} & \Rightarrow & \exists\ y \in \ \{0,1\}^{ast} \text{ such that} \\ & & y \text{ is a hitting string of } <S> \\ & & \text{and } |y| = n \end{array}$$

Since each string in $<S>$ corresponds to a clause Φ_p in Φ, then at least one literal is the one that corresponds to a "hit" in $<S>$.

$$\begin{array}{c} \Rightarrow \Phi \text{ is satisfiable} \\ \Rightarrow \Phi \in \text{SAT} \end{array}$$

HALF–CLIQUE–Problem

$$\text{HALF–CLIQUE} = \{<G> \mid G \text{ is a graph with a clique of size } \left\lceil \frac{|V|}{2} \right\rceil \}$$

Claim: HALF–CLIQUE is $\mathcal{NP}$–COMPLETE
Proof:

- HALF–CLIQUE $\in \mathcal{NP}$.
  Use the set of $\lceil |V|/2 \rceil$ vertices in the clique as certificate of G. Checking whether it is a clique is accomplished in polynomial time by checking whether, for every pair u, v in the set, the edge $(u, v) \in E$.

- HALF–CLIQUE $\in \mathcal{NP}$–HARD.
  Show that

  $$\text{CLIQUE} \leq_p \text{HALF–CLIQUE}$$

  The reduction takes a graph $G(V, E)$ and

  $$<G, k> \in \text{CLIQUE} \Leftrightarrow <G'> \in \text{HALF–CLIQUE}$$

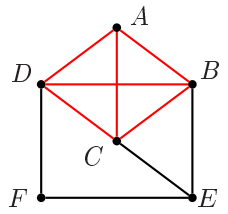
  The reduction depends on k:

  - if $k = \left\lceil \frac{|V|}{2} \right\rceil$, then let $G' = G$
  - if $k > \left\lceil \frac{|V|}{2} \right\rceil$, add m vertices such that $k = \left\lceil \frac{|V|}{2} \right\rceil$, then let $G' = G \cup \{q_1, q_2, \ldots, q_m\}$ where the newly added nodes $q_1, q_2, \ldots, q_m$ are of degree 0 (they are not adjacent to any nodes). Now $k = \left\lceil \frac{|V|}{2} \right\rceil$ and

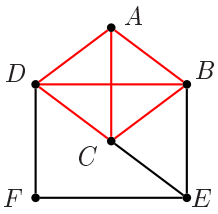
    $$G \text{ has a clique of size } k \Leftrightarrow G' \text{ has a clique of size } \left\lceil \frac{|V|}{2} \right\rceil$$

EXAMPLE:

$G = (V, E)$
$k = 4$
$\left\lceil \frac{|V|}{2} \right\rceil = 3$



$\Rightarrow$



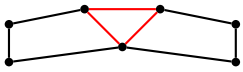
$\bullet\ q_1$

$\{A, B, C, D\}$ is a clique of size 4

$G' = (V', E)$, $\left\lceil \frac{|V|}{2} \right\rceil = 4$
$V' = V \cup \{q_1\}$

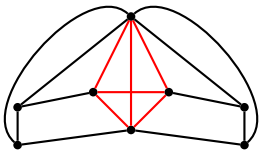
- if $k < \left\lceil \frac{|V|}{2} \right\rceil$, add m vertices such that $k + m = \left\lceil \frac{|V|}{2} \right\rceil$ then let $G' = G \cup \{q_1, q_2, \ldots, q_m\}$ where the newly added vertices $q_1, q_2, \ldots, q_m$ are adjacent to all nodes of G and every q_1 is adjacent to q_j, $i \neq j$.

  EXAMPLE:



Clique of size 3

$\Rightarrow$



Clique of size 4

So now $|V'| = |V| + m$ and

$$G \text{ has a clique of size } k \Leftrightarrow G' \text{ has a clique of size } k + m$$

Note that $k + m = \lceil |V'|/2 \rceil$ and $|V'| = |V| + m$, hence $2k + 2m = |V'|$ and so $2k + 2m = |V| + m$. Therefore $|V| = 2k + m$. In all cases, the "$\Leftrightarrow$" is satisfied. The reduction is in polyonmial time. So

$$\text{CLIQUE} \leq_p \text{HALF–CLIQUE}$$
$$\Rightarrow \text{HALF–CLIQUE} \in \mathcal{NP}\text{–COMPLETE}$$

A hamiltonian path in a graph $G = (V, E)$ is a path that traverses each vertex exactly once. More precisely:

HAM-PATH = $\{<G, u, v> \mid \exists$ a hamiltonian path from u to v in graph $G\}$

Claim: HAM–PATH $\in \mathcal{NP}$–COMPLETE

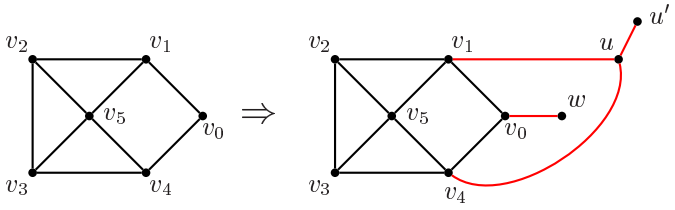
Proof:

- HAMP–PATH $\in \mathcal{NP}$ since a path can serve as a certificate and be checked in polynomial time to be hamiltonian.
- HAM–PATH $\in \mathcal{NP}$–HARD.
  Claim: HAM–CYCLE $\leq_p$ HAM–PATH
  Transform any graph $G = (V, E)$ into $G' = (V', E)$ where

  a) $V' = V \cup \{u, u', w\}$

  b) $E' = E \cup \{(u', u), (w, v_0)\} \cup \{(u, v) | (v_0, v) \in E\}$ for some $v_0 \in E$.

  Adding nodes and check is achieved in polynomial time.



G has a hamiltoninan cycle $\Leftrightarrow$ G' has a hamiltonian path

($\Leftarrow$) Suppose G' has a hamiltonian path P. The extreme edges of P have to be (u', u) and (v_0, w). Suppose $(u, v) \in P$. The rest of P is a path traversing each node in $V - \{v_0, v\}$ exactly once. Call it C.

$$(u, v) \in E' \;\Rightarrow\; (v, v_0) \in E$$

So C with (v_0, v) is a hamiltonian cycle in G.

($\Rightarrow$) If G has a hamiltonian cycle $C = \{v_0, \ldots, v, v_0\}$ then G' will have the hamiltonian path $P = [w, v_0, \ldots, v, u, u']$.

SPATH $= \{<G, a, b, k> \mid G$ is a graph containing
a simpel path of length at most k from a to b $\}$.

Claim: SPATH $\in \mathcal{P}$
Proof: The following is a breadth–first search algorithm which runs in polynomial time.

SPATH*(G, a, b, k)*
begin
  mark a
  repeat k times or **until** b is marked
  (whichever takes fewer iterations)
    for each marked node n not yet
    expanded, expand it
  accept if b is marked and reject otherwise
end

To expand a node means to mark all nodes ajdacent to it. Also recall: u and v are adjacent nodes if $(u, v) \in E$.

LPATH $= \{<G, a, b, k> \mid G$ is a graph containing
a simpel path of length at least k from a to b $\}$.

Claim: LPATH $\in \mathcal{NP}$–COMPLETE.
Proof:

- Given a set of vertices, we can check in polynomial time whether it is a simple path of length at least k.
- To show that LPATH $\in \mathcal{NP}$–HARD, we reduce HAM–PATH to it in polynomial time. That is, we show

  $$\text{HAM–PATH} \leq_p \text{LPATH}$$

  Let $(G, a, b) \in$ HAM–PATH, where $G = (V, E)$. (G, a, b) can be polynomial time reduced to $(G, a, b, n-1)$, where $n = |V|$. Since there are only n vertices in G, a simple path (which cannot repeat vertices) of length exactly $n - 1$ is a hamiltonian path and vice versa.

Bibliography

[AMB93] Ravindra K. Ahuja, Thomas L. Magnanti, and Orlin James B. *Network Flows.* Prentice-Hall, Inc., 1993.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, 1990.

[Wes96] Douglas B. West. *Introducation to Graph Theory.* Prentice Hall, Inc., 1996.