Part I

Graph and Network Theory

Chapter 1

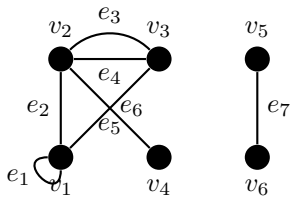
Graphs and Subgraphs

1.1 Graph

Definition 1.1.1 (Graph)**.** A **graph** G consists of a finite set $V(G)$ on vertices, a finite set $E(G)$ on edges and an **incident relation** than associates with any edge $e \in E(G)$ an unordered pair of vertices not necessarily distinct called **ends**.

Example. The following graph can be represented as

$$V = V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6\} \tag{1.1}$$
$$E = E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\} \tag{1.2}$$
$$e_1 = v_1v_2, \quad e_2 = v_2v_4, \quad \dots \tag{1.3}$$



Definition 1.1.2 (Loop, Parallel, Simple Graph)**.** An edge with identical ends is called a **loop**, Two edges having the same ends are said to be **parallel**, A graph without loops or parallel edges is called **simple graph**

Definition 1.1.3 (Adjacent)**.** Two edges of a graph are **adjacent** if they have a common end, two vertices are **adjacent** if they are jointed by an edge.

1.2 The Adjacency and Incidence Matrices

Saving a graph in computer program can be implemented in the following ways:

- Adjacency matrix: $m \times n$ matrix, for $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
- Linked list: For every vertex v, there is a linked list containing all neighbors of v.

Assuming we are dealing with undirected graphs, n is the number of vertices, m is the number of edges, $n - 1 \leq m \leq n(n-1)/2$, d_v is the number of neighbors of v then

	Matrix	Linked lists
memory usage	$O(n^2)$	$O(m)$
time to check $(u, v) \in E$	$O(1)$	$O(d_u)$
time to list all neighbors of v	$O(n)$	$O(d_v)$

1.3 Subgraph

Definition 1.3.1 (Subgraph)**.** Given two graphs G and H, H is a **subgraph** of G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ and an edge has the same ends in H as it does in G. Furthermore, if $E(H) \neq E(G)$ then H is a proper subgraph.

Definition 1.3.2 (Spanning)**.** A subgraph H on G is **spanning** if $V(H) = V(G)$

Definition 1.3.3 (Vertex-induced, Edge-induced)**.** For a subset $V^{'} \subseteq V(G)$ we define an **vertex-induced** subgraph $G[V^{'}]$ to be the subgraph with vertices $V^{'}$ and those edges of G having both ends in $V^{'}$. The **edge-induced** subgraph $G[E^{'}]$ has edges $E^{'}$ and those vertices of G that are ends to edges in $E^{'}$.

Notice: If we combine node-induced or edge-induced subgraphs $G(V^{'})$ and $G(V - V^{'})$, we cannot always get the entire graph.

1.4 Degree

Definition 1.4.1 (Degree)**.** Let $v \in V(G)$, then the **degree** of $v \in V(G)$ denote by $d_G(v)$ is defines to be the number of edges incident of v. Loops counted twice.

Theorem 1.1. *For any graph $G = (V, E)$*

$$\sum_{v \in V} d(v) = 2|E| \tag{1.4}$$

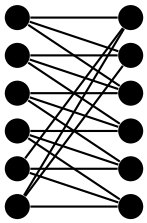
Proof. $\forall$ edge $e = uv$ with $u \neq v$, e is counted once for u and once for v, a total of two altogether. If $e = uu$, a loop, then it is counted twice for u. □

Problem 1.1. Explain clearly, what is the largest possible number of vertices in a graph with 19 edges and all vertices of degree at least 3. Explain why this is the maximum value.

Solution. The maximum number is 12.

Proof. First we prove 12 vertices is possible, then we prove 13 vertices is not possible

- The following graph contains 12 vertices and 18 edges, each vertex has a degree of 3.



- For 13 vertices and each vertex has a degree of at least 3 will require at least

$$2|E| = \sum_{v \in V} d(v) \geq 3 \times |N| = 3 \times 13 \Rightarrow |E| \geq 19.5 > 19 \tag{1.5}$$

  edges, i.e., 13 vertices is not possible.

□

Corollary 1.1.1. *Every graph has an even number of odd degree vertices.*

Proof.

$$V = V_E \cup V_O \Rightarrow \sum_{v \in V} d(v) = \sum_{v \in V_E} d(v) + \sum_{v \in V_O} d(v) = 2|E| \tag{1.6}$$

□

1.5 Special Graphs

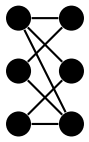
Definition 1.5.1 (Complete Graph)**.** A **complete** graph $K_n (n \geq 1)$ is a simple graph with n vertices and with exactly one edge between each pair of distinct vertices.

Definition 1.5.2 (Cycle)**.** A **cycle** graph $C_n (n \geq 3)$ consists of n vertices $v_1, ...v_n$ and n edges $\{v_1, v_2\}, \{v_2, v_3\}, ...\{v_{n-1}, v_n\}$

Definition 1.5.3 (Wheel)**.** A **wheel** graph $W_n (n \geq 3)$ is a simple graph obtains by adding one vertex to the cycle graph C_n, and connecting this new vertex to all vertices of C_n

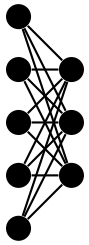
Definition 1.5.4 (Bipartite Graph)**.** A simple graph is said to be **bipartite** if the vertex set can be expressed as the union of two disjoint non-empty subsets V_1 and V_2 such that every edges has one end in V_1 and another end in V_2

Example. Here is an example for bipartite graph



Definition 1.5.5 (Complete Bipartite Graph)**.** The **complete bipartite graph** K_{mn} is the bipartite graph V_1 containing m vertices and V_2 containing n vertices such that each vertiex in V_1 is adjacent to every vertex in V_2

Example. Here is an example for K_{53}

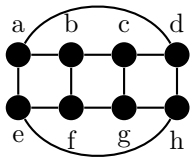


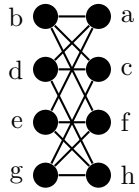
Theorem 1.2. *A graph G is bipartite iff every cycle is even.*

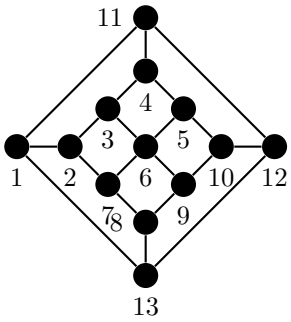
Proof. ($\Rightarrow$) If the graph G is bipartite, by definition, the vertices of graph can be partition into two groups, that within the group there is no connection between vertices. Therefore, for each cycle, the odd index of vertices and even index of vertices has to be choose alternatively from each groups. Therefore the cycle has to be even.
($\Leftarrow$) Prove by contradiction. A graph can be connected or not connected.
If G is connected and has at least two vertices, for an arbitrary vertex $v \in V(G)$, we can calculate the minimum number of edges between the other vertices v' and v (i.e., length, denoted by $l(v', v)$), for all the vertices that has odd length to v, assign them to set V_1, for the rest of vertices (and v), assign to set V_2. Assume that G is not bipartite, which means there are at least one edge between distinct vertices in set V_1 or set V_2, without lost of generality, assume that edge is uw, $u, w \in V_1$. For all vertices in V_1 there is an odd length of path between the vertex and v, therefore, there exists an odd $l(u, v)$, and an odd $l(w - v)$. The length of cycle $l(u, w, v) = 1 + l(u, v) + l(w, v)$, which is an odd number, it contradict with the prerequisite that all cycles are even, which means the assumption that G is not bipartite is incorrect, G should be bipartite.
If G is not connected. Then G can be partition into a set of disjointed subgraphs which are connected with at least two vertices or contains only one vertex. For the component that has more that one vertices, we already proved that it has to be bipartite. For the subgraph $G_i \subset G, i = 1, 2, ..., n$, the vertices can be partition into $V_{i1} \in V(G_i)$ and $V_{i2} \in V(G_i)$, where $V_{i1} \cap V_{i2} = \emptyset$, the union of those subgraphs are bipartite too because $V_1 = \cup_{i=1}^{n} V_{i1} \in V(G)$ and $V_2 = \cup_{i=1}^{n} V_{i2} \in V(G)$ satisfied the condition of bipartite. For the subgraph that has one one vertices, those vertices can be assigned into either V_1 or V_2. □

Example. The following graph is bipartite, it only contains even cycles.
We can rearrange the graph to be more clear as following
The vertices of graph G can be partition into two sets, $\{a, c, f, h\}$ and $\{b, d, e, g\}$

Example. The following graph is not bipartite
The cycle $c = v_1 v_1 1 v_4 v_3 v_2$ have odd number of vertices.







1.6 Directed Graph

Definition 1.6.1. A graph $G = (V, E)$ is called directed if for each edge $e \in E$, there is a **head** $h(e) \in V$ and a **tail** $t(e) \in V$ and the edges of e are precisely $h(e)$ and $t(e)$, denoted $e = (t(e), h(e))$

Definition 1.6.2. We call directed graphs **digraphs**, we call edges in a digraph are called **arcs**, and vertices in a digraph **nodes**

Definition 1.6.3. Similar as in the undirected case we have walks, traces, paths and cycles in digraphs.

Definition 1.6.4. A vertex $v \in V$ is **reachable** from a vertex $u \in V$ if there is a (u, v)-dipath. If at the same time u is reachable from v, they are **strongly connected**

Definition 1.6.5. A digraph is strongly connected if every pair of vertices are strongly connected.

Definition 1.6.6. A digraph is **strict** if it has no loops and whenever e and f are parallel, $h(e) = t(f)$

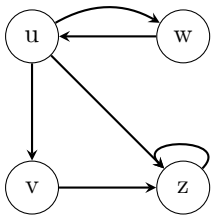
Definition 1.6.7. For a vertex v in a digraph D, the **indegree** of v in D, denoted by $d^+(v)$ is the number of arcs of D having head V. The **outdegree** of v is denoted by $d^-(v)$ is the number of arcs of D having tail v.

Let $D = (V, A)$ be a digraph with no loops a vertex-arc **incident matrix** for D is a $(0, 1, -1)$ matrix N with rows indexed by $V = \{v_1, ..., v_n\}$ and column indexed by $A = \{e_1, ..., e_m\}$ and where entry (i, j) in the matrix n_{ij} is

$$n_{ij} = \begin{cases} 1, & \text{if } v_i = h(e_j) \\ -1, & \text{if } v_i = t(e_j) \\ 0, & \text{otherwise} \end{cases} \tag{1.7}$$

$$\begin{bmatrix} -1 & 0 & -1 & -1 & 1 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{1.8}$$

1.7 Sperner's Lemma


u
w
v
z

Chapter 2

Paths, Trees, and Cycles

2.1 Walk

Definition 2.1.1 (walk)**.** A **walk** in a graph G is a finite sequence $w = v_0e_1v_1e_2...e_kv_k$, where for each $e_i = v_{i-1}v_i$ the edge and its ends exists in G. We say that walk v_0 to v_k on (v_0, v_k)-walk.

Example.

$$w = v_2e_4v_3e_4v_2e_5v_3 \tag{2.1}$$

is a walk, or (v_2, v_3)-walk

Definition 2.1.2 (origin, terminal, internal, length)**.** For (v_0, v_k)-walk, The vertices v_0 and v_k are called the **origin** and the **terminal** of the walk w, $v_1..v_{k-1}$ are called **internal** vertices. The integer k is the **length** of the walk. Length of w equals to the number of edges.

We can create a reverse walk w^{-1} by reversing w.

$$w^{-1} = v_ke_kv_{k-1}e_{k-1}...e_2v_1 \tag{2.2}$$

(The reverse walk is guaranteed to exist because it is an undirected graph)
Given two walks w and w' we can create a third walk denoted by ww' by concating w and w'. The new walk's origin is the same as terminal.

2.2 Path and Cycle

Definition 2.2.1 (trail)**.** A **trail** is a walk with no repeating edges. e.g., $v_3e_4v_2e_5v_3$

Definition 2.2.2 (path)**.** A **path** is a trail with no repeating vertices. e.g., $v_3e_4v_2$

Notice: Paths $\subseteq$ Trails $\subseteq$ Walks

Definition 2.2.3 (closed, cycle)**.** A path is **closed** if it has positive length and its origin and terminal are the same. e.g., $v_1e_2v_2e_4v_3e_3v_1$. A closed trail where origin and internal vertices are distinct is called a **cycle** (The only time a vertex is repeated is the origin and terminal)

Definition 2.2.4 (even/odd cycle)**.** A cycle is **even** if it has a even number of edges otherwise it is **odd**.

Problem 2.1. Prove that if C_1 and C_2 are cycles of a graph, then there exists cycles $K_1, K_2, ..., K_m$ such that $E(C_1)\Delta E(C_2) = E(K_1) \cup E(K_2) \cup ... \cup E(K_m)$ and $E(K_i) \cap E(K_j) = \emptyset, \forall i \neq j$. (For set X and Y, $X\Delta Y = (X - Y) \cup (Y - X)$, and is called the symmetric difference of X and Y)

Proof. Proof by constructing $K_1, K_2, ...K_m$. Denote

$$C_1 = v_{11}e_{11}v_{12}e_{12}v_{13}e_{13}...v_{1n}e_{1n}v_{11} \tag{2.3}$$

$$C_2 = v_{21}e_{21}v_{22}e_{22}v_{23}e_{23}...v_{2k}e_{2k}v_{21} \tag{2.4}$$

Assume both cycle start at the same vertice, $v_{11} = v_{12}$. (If there is no intersected vertex for C_1 and C_2, just simply set $K_1 = C_1$ and $K_2 = C_2$)
The following algorithm can give us all $K_j, j = 1, 2, ..., m$ by constructing $E(C_1)\Delta E(C_2)$. Also, the complexity is $O(mn)$, which makes the proof doable.

Now we prove that $K_i \cap K_j = \emptyset, \forall i \neq j$. For each K_j, it is defined by two (v_o, v_t)-paths in the algorithm. From

Algorithm 1 Find $K_1, K_2, ...K_m$ by constructing $E(C_1)\Delta E(C_2)$

Require: Graph G, cycle C_1 and C_2
Ensure: $K_1, K_2, ...K_m$
1: Initial, $K \leftarrow \emptyset$, $j = 1$
2: Set temporary storage units, $v_o \leftarrow v_{11}$, $v_t \leftarrow \emptyset$
3: **for** $i = 1, 2, ..., n$ **do**
4: **if** $e_{1i} \in C_2$ **then**
5: **if** $v_o \neq v_{1i}$ **then**
6: $v_t \leftarrow v_{1i}$
7: concate (v_o, v_t)-path $\subset C_1$ and (v_o, v_t)-path $\subset C_2$ to create a new K_j
8: Append K with K_j, $K \leftarrow K \cup K_j$
9: Reset temporary storage unit. $v_o \leftarrow v_{1(i+1)}$ (or v_{11} if $i = n$), $v_t \leftarrow \emptyset$
10: **else**
11: $v_o \leftarrow v_{1(i+1)}$ (or v_{11} if $i = n$)

the algorithm we know that all the edges in (v_o, v_t)-path in C_1 are not intersecting with C_2, because if the edge in C_1 is intersected with C_2, either we closed the cycle K_j before the edge, or we updated v_o after the edge (start a new K_j after that edge). By definition of cycle, all the (v_o, v_t)-path that are subset of C_1 are not intersecting with each other, as well as all the (v_o, v_t)-path that are subset of C_2. Therefore, $K_i \cap K_j = \emptyset, \forall i \neq j$. □

Definition 2.2.5 (connected vertices)**.** Two vertices u and v in a graph are said to be **connected** if there is a path between u and v.

Definition 2.2.6 (component)**.** Connectivity between vertices is an equivalence relation on $V(G)$, if $V_1, ...V_k$ are the corresponding equivalent classes then $G[V_1]...G[V_k]$ are **components** of G. If graph has only one component, then we say the graph is connected. A graph is connected iff every pair of vertices in G are connected, i.e., there exists a path between every pair of vertices.

Problem 2.2. If G is a simple graph with at least two vertices, prove that G has two vertices with the same degree.

Proof. A simple graph can only be connected or not connected.

- If G is connected, i.e., for all vertices, the degree is greater than 0. Also the graph is simple, for a graph with $|N|$ vertices, the degree of each vertex is less or equal to $|N| - 1$ (cannot have loop or parallel edge). For $|N|$ vertices, to make sure there is no two vertices that has same degree, it will need $|N|$ options for degrees, however, we only have $|N| - 1$ option. According to pigeon in holes principle, there has to be at least two vertices with the same degree.

- If G is not connected, i.e., the graph has more than one component. One of the following situation will happen:

  - For all components, each component contains only one vertex. Since we have at least two vertices, which means there are at least two component that has only one vertex. For those vertices, at least two vertices has the same degree as 0.

  - At least one component has more than one vertices. In this situation, we can find a component that has more than one vertices as a subgraph G' of the graph G. That G' is a connected simple graph by definition. We have already proved that a connected simple graph has two vertices with the same degree, which means G has two vertices with the same degree.

□

2.3 Tree and forest

Definition 2.3.1 (acyclic graph)**.** A graph is called **acyclic** if it has no cycles

Definition 2.3.2 (forest, tree)**.** A acyclic graph is called a **forest**. A connected forest is called a **tree**.

Theorem 2.1. *Prove that T is a tree, if T has exactly one more vertex than it has edges.*

Proof. 1. First we prove for any tree T that has at least two vertices, there has to be at least one leaf, i.e., now we prove that we can find u with degree of 1. Proof by constructing algorithm. (In fact we can prove that there are at least two leaves.)
The above algorithm is guaranteed to have an end because a tree is acyclic by definition

Algorithm 2 Find one leaf in a tree

Require: $d(u) = 1$
Ensure: A tree T has at least one vertex
1: Let u and v be any distinct vertex in a tree T
2: Let p be the path between u and v
3: **while** $d(u) \neq 1$ **do**
4: **if** $d(u) > 1$ **then**
5: Let $n(u)$ be the set of neighboring vertices of u
6: In $n(u)$, find a u' that the edge between u and u', denoted by e, $e \notin p$
7: $u \leftarrow u'$
8: $p \leftarrow p \cup e$

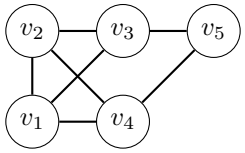
2. Then, if we remove one leaf in the tree, i.e., we remove an edge and a vertex, where that vertex only connects to the edge we removed. One of the following situations will happen:
   (a) Situation 1: The remaining of T is one vertex. In this case, T has two vertices an one edge. (Exactly one more vertex than it has edges)
   (b) Situation 2: The remaining of T is another tree $T^{'}$ (removal of edges will not change acyclic and connectivity), where $|V(T)| = |V(T^{'})| + 1$ and $|E(T)| = |E(V^{'}| + 1$. (one edge and one vertex has been removed)
3. Do the leaf removal process recursively to $T^{'}$ if Situation 2 happens until Situation 1 happens.

□

2.4 Spanning tree

Definition 2.4.1 (spanning tree)**.** A subgraph T of G is a **spanning tree** if it is spanning ($V(T) = V(G)$) and it is a tree.

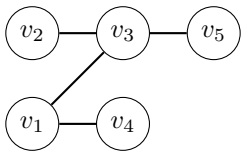
Example. In the following graph
This is a spanning tree



Problem 2.3. Prove that if T_1 and T_2 are spanning trees of G and $e \in E(T_1)$, then there exists a $f \in E(T_2)$, such that $T_1 - e + f$ and $T_2 + e - f$ are both spanning trees of G.

Proof. One of the following situation has to happen:

1. If for given $e \in E(T_1)$, $\exists f = e \in E(T_2)$, then $T_1 - e + f = T_1$, $T_2 + e - f = T_2$ are both spanning trees of G



2. If for given $e \in E(T_1)$, $e \notin E(T_2)$, the following will find an edge f that $T_1 - e + f$ and $T_2 + e - f$ are both spanning trees of G.
   (a) T_1 is a spanning tree, removal of $e \in E(T_1)$ will disconnect the spanning tree into two components (by definition of spanning tree), denoted by $G_1 \subset G$ and $G_2 \subset G$, by definition, $V(G_1)$ and $V(G_2)$ is a partition of $V(G)$.
   (b) Add e into T_2. We can proof that by adding an edge into a tree will create exactly one cycle, denoted by C, $e \in E(C)$.
   (c) For C, since it is a cycle and one end of e is in $V(G_1)$, the other end of e is in $V(G_2)$, there has to be at least two edges (can be more) that has one end in $V(G_1)$ and the other end in $V(G_2)$, denote the set of those edges as $E \subset E(C)$, one of those edges is $e \in E$
   (d) Choose any $f \in E$ and $f \neq e$, for that f, $T_1 - e + f$ and $T_2 + e - f$ are both spanning trees of G.
   (e) Prove that $T_1 - e + f$ is a spanning tree
      i. $T_1 - e + f$ have the same set of vertices as T_1, therefore it is spanning.
      ii. It is connected both within G_1 and G_2, for f, one end is in $V(G_1)$, the other end is in $V(G_2)$ therefore $T_1 - e + f$ is connected.
      iii. $T_1 - e + f$ have the same number of edges as T_1, which is $|T_1| - 1$, therefore $T_1 - e + f$ is a tree. (We have proven the connectivity in the previous step.)
      iv. $T_1 - e + f$ is spanning, connected, a tree, therefore it is a spanning tree.
   (f) Prove that $T_2 + e - f$ is a spanning tree
      i. $T_2 + e - f$ have the same set of vertices as T_2, therefore it is spanning.
      ii. T_2 is connected, adding an edge will not break connectivity, therefore $T_2 + e$ is connected, removing an edge in a cycle will not break connectivity, therefore $T_2 + e - f$ is connected.
      iii. $T_2 - e + f$ have the same number of edges as T_2, which is $|T_2| - 1$, therefore $T_2 + e - f$ is a tree. (We have proven the connectivity in the previous step.)
      iv. $T_2 - e + f$ is spanning, connected, a tree, therefore it is a spanning tree.

□

Theorem 2.2. *Every connected graph has a spanning tree.*

Proof. Prove by constructing algorithm: □

Algorithm 3 Find a spanning tree for connected graph (Prim's Algorithm in unweighted graph)

Require: a connected graph G and an enumeration $e_1, ...e_m$ of the edges of G
Ensure: a spanning tree T of G
1: Let T be the spanning subgraph of G with $V(T) = V(G)$ and $E(T) = \emptyset$
2: $i \leftarrow 1$
3: **while** $i \leq |E|$ **do**
4: **if** $T + e_i$ is acyclic **then**
5: $T \leftarrow T + e_i$
6: $i \leftarrow i + 1$

Notice: This algorithm can be improved, one idea is to make summation of edges in spanning subgraph less or equation to $|V| - 1$

For the complexity of spanning tree algorithm:

1. Space complexity, $2|E|$, which is $O(|E|)$
2. Time complexity
   (a) How to check for acyclic?
      i. At every stage T has certain components $V_1, ...V_t$, (every time we add an edge, the number of components minus 1)
      ii. So at the beginning $t = |V|$ with $|V_i| = 1 \forall i$ and at the end, $t = 1$.
   (b) Count the amount of work for the algorithm.
      i. Need to check for acyclic for each edge, which costs $O(|E|)$
      ii. Need to flip the pointer for each vertex, for each vertex, at most will be flipped $\log |V|$ times, altogether $|V| \log |V|$ times.
      iii. The time complexity is $O(|E| + |V| \log |V|)$
3. First we need to input the data, create an array such that the first and the second entries are the ends of e_1, third and fourth are the ends of e_2, and so on.
4. The amount of storage needs in $2|E|$, which is $O(|E|)$
5. The main work involved in the algorithm is for each edges e_i and the current T, to determine if $T + e_i$ creates a cycle.
6. suppose we keep each component V_i by keeping for each vertex a pointer from the vertex to the name of the component containing it. Thus if $\mu \in V_3$, there will be a pointer from μ to integer 3.
7. Then when edge $e_i = \mu v$ is encountered in Step 2, we see that $T + e_i$ contains a cycle if and only if μ and v point to same integer which means they are in the same component
8. If they are not in the same component, we want to add the edge which means then I have to update the pointers.

To prove algorithm we need to show the output is a spanning tree, which means three properties must hold:

- spanning (Step I)
- acyclic (We never add an edge that create a cycle)
- connected (Proof by contradiction)

So it is sufficient to show that the output will be connected.

Proof. (Proof by Contradiction) Suppose the output graph T of the algorithm is NOT connected. Let T_1 be a component of T, let $x \in T_1$ and $y \notin T_1$. But G is a connected graph (given from the beginning), so there must be a path in G that connects x and y. Let such a path in G be $p = xe_1v_1e_2, ..v_{k-1}e_ky$. Clearly, $p \notin T_1$. So there must be a first vertex in P that not in T_1. So $e_i \notin E(T)$, the only way this can happen when applying the algorithm is if $T + e_i$ creates a cycle C, i.e., $e_i \in C$, so $C - e_i$ is a path connecting v_{i-1} and v_i. So $c - e_i \in T$, so v_{i-1} is connected to $v_i \in T$. Contradiction. □

2.5 Cayley's Formula

2.6 Connectivity, DFS, BFS

2.6.1 Connectivity Problem

For connectivity problem, the input is a graph $G = (V, E)$, with linked list representation, and two vertices $s, t \in V$. The problem is to find whether there is a path connecting s to t in G
There are two commonly use methods to solve connectivity problem: depth-first search and breath-first search.

2.6.2 Depth-First Search (DFS)

The idea of DFS is enumerating children before siblings when search in the graph/tree. It is a recursive algorithm. First, start with s, then travel through the fist edge leading out of the current vertex, when reached a "visited" vertex, go back and travel through next edge. If tried all edges leading out of the current vertex, go back.
The algorithm is as following

Algorithm 4 Depth-First Search

```
1: Initialize, make all vertices as "unvisited"
2: RecursiveDFS(s)
```

Algorithm 5 RecursiveDFS(v)

```
1: Mark v as "visited"
2: for ∀u ∈ d_s do:
3:     if u is "unvisited" then
4:         RecursiveDFS(u)
```

The running time of DFS is $O(n + m)$.
If we only want to know if s and t are connected, the algorithm can be terminated if $u = t$.

2.6.3 Breadth-First Search (BFS)

The idea of BFS is enumerating siblings before children when search in the graph/tree. The general steps of BFS is as following: First, build layers $L_0, L_1, \cdots$; Next, set $L_0 = \{s\}$, where s is the starting vertex; Then, L_{j+1} contains all nodes that are not in $\cup_{i=0}^{j} L_i$ and have an edge to a vertex in L_j
The algorithm is as following

Algorithm 6 Breadth-First Search

```
1: Initialize, head ← 1, tail ← 1, queue[1] ← s, mark all vertices as "unvisited"
2: Mark s as "visited"
3: while head ≥ tail do
4:     v ← queue[tail], tail ← tail + 1
5:     for ∀u ∈ d_v do
6:         if u is "unvisited" then
7:             head ← head + 1, queue[head] = u
8:             Mark u as "visited"
```

The running time of BFS is $O(n + m)$
If we only want to know if s and t are connected, the algorithm can be terminated if $u = t$.

2.6.4 Cycle detection

The following algorithm is for connected graph, if the graph is not connected, run the algorithm for each component until cycle is detected or all the components have been calculated. Since the complexity for running in connected graph is $O(n + m)$, n as the number of vertices/nodes, and m as the number of edges/arcs, the running time of disconnected graph is the **summation** of running time in each component, where each component is connected. Therefore the complexity is the same in disconnected graph as in connected graph.
The main idea is starting with arbitrary vertex/node, using DFS or BFS to search on the graph try to revisit the vertex/node we start with. If succeed, a cycle is detected, otherwise if all the vertices/nodes has been visited, then no cycle exists. And in linked-list representation, the complexity is $O(|V| + |E|)$, i.e. $O(n + m)$. However, there is slightly different in undirected graph and directed graph, for undirected graph needs at least three vertices to form a cycle while directed graph needs at least two.
Here is the detail algorithm (DFS) for undirected graph:

Algorithm 7 Main algorithm

For all nodes, labeled as "unvisited"
Arbitrary choose a vertex v, add a dummy vertex w, add a dummy edge (w, v), label w as "visited"
run $DFS(w, v)$
Remove dummy vertex w and dummy edge (w, v)
if $DFS(w, v)$ returns "Cycle is found" **then**
    return "Cycle is found"
else
    return "No cycle detected"

Algorithm 8 DFS(w, v)

Label v as "visited"
if number of v 's neighbor is 1 **then**
    return null
else
    for all neighbor u in linked-list of v excepts w **do**
        if u is labeled as "visited" **then**
            return "Cycle is found"
        else
            run $DFS(v, u)$

Now check the complexity. Denote $v \in N$ as a node in graph G, total number of nodes denoted by n, denote d_v as number of neighbors of node v. The complexity of $DFS(w, v)$ is $O(d_v)$ for each node v it visited (it should be $O(v)$ because we need $O(1)$ to check if a node is w), each node can only be visited, by "visited" it means $DFS(w, v)$ is executed, at most once, which is controlled by the "visited" label. The total complexity is $O(n + \sum_{v \in N} d_v) = O(n + m)$
For directed graph, we assume in the linked-list of each node, 1) it only contains the vertices where the node leading out to, or 2) we can distinguish the vertices it leading out to in $O(1)$. Here is the detail algorithm (DFS) for directed graph:

Algorithm 9 Main algorithm

For all nodes, labeled as "unvisited"
Arbitrary choose a node v, run $DFS(v)$
if $DFS(v)$ returns "Cycle is found" **then**
    return "Cycle is found"
else
    return "No cycle detected"

Algorithm 10 DFS(v)

Label v as "visited"
if number of vertices v leading out to is 0 **then**
    return null
else
    for all leading out vertices u in linked-list of v **do**
        if u is labeled as "visited" **then**
            return "Cycle is found"
        else
            run $DFS(u)$

Now check the complexity. Similar to undirected case, denote $v \in N$ as a node in graph G, total number of nodes denoted by n, denote d_v as number of edges leading out from node v. The complexity of $DFS(v)$ is $O(d_v)$ for each

node v it visited, each node can only be visited, by "visited" it means $DFS(v)$ is executed, at most once, which is controlled by the "visited" label. The total complexity is $O(n + \sum_{v \in N} d_v) = O(n + m)$
Also, for undirected (connected) graph, if we don't need to find the cycle and only need to decide if there is a cycle or not, we can just check the number of vertices, n, and number of edges, m. If $n \leq m$ then G contains a cycle, otherwise no cycle, the complexity is $O(1)$.

2.6.5 Test Bipartiteness

We have proved that a bipartite graph only has even cycles, and the graph with only even cycles are bipartite graph, however, that is not very continent to test if a graph is bipartite because it needs to enumerate all cycles.
The other idea to test bipartiteness is try to color the vertices of the graph, if it can be 2-colored, then the graph is bipartite, otherwise it is not.
The following is the algorithm (using BFS)

Algorithm 11 Test Bipartiteness

```
1: Initialize, head ← 1, tail ← 1, queue[1] ← s, mark all vertices as "unvisited"
2: Mark s as "visited"
3: color[s] ← 0
4: while head ≥ tail do
5:     v ← queue[tail], tail ← tail + 1
6:     for ∀u ∈ d_v do
7:         if u is "unvisited" then
8:             head ← head + 1, queue[head] = u
9:             Mark u as "visited"
10:            color[u] ← 1 − color[v]
11:        else
12:            if color[u] == color[v] then return False
   return True
```

2.6.6 Topological Ordering

The topological ordering problem is given a directed acyclic graph $G = (V, E)$, output a 1-to-1 function $\pi : V \to \{1, 2, \cdots, n\}$ so that if $(u, v) \in E$, $\pi(u) < \pi(v)$
The idea is each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges. To make the algorithm efficient, we can 1) use linked lists of outgoing edges; 2) maintain the in degree d_v of vertices; 3) maintain a queue (or stack) of vertices v with $d_v = 0$
The following is the algorithm

Algorithm 12 topological-sort(G)

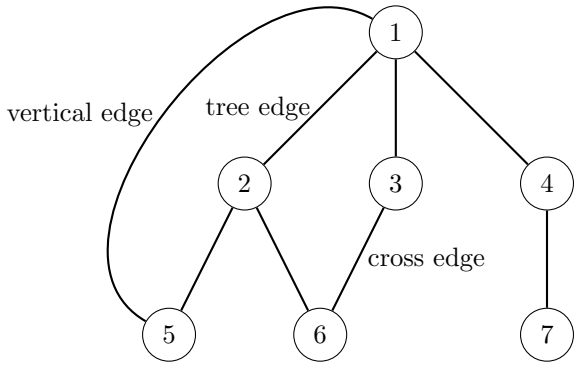
```
1: Initialize, let ∀v ∈ V, d_v = 0
2: for v ∈ V do
3:     for u, (u, v) ∈ E do
4:         d_u ← d_u + 1
5: S ← {v : d_v = 0}, i ← 0
6: while S ≠ ∅ do
7:     v ← v ∈ S, S ← S \ {v}
8:     i ← i + 1, π(v) ← i
9:     for u, (u, v) ∈ E do
10:        d_u ← d_u − 1
11:        if d_u = 0 then
12:            S ← S ∪ {u}
13: if i < n then return Not directed acyclic graph
```

Running time $O(n + m)$

2.6.7 Bridge

Definition 2.6.1 (Tree Edge, Cross Edge, Vertical Edge)**.** Given a graph $G = (V, E)$ and a rooted tree T in G, edge in G can be classified by three types:

- Tree edges: edge in T
- Cross edges: (u, v), u and v do not have an ancestor-descendant relation
- Vertical edges: (u, v), u is an ancestor of v, or v is an ancestor of u



In a BFS tree T of a graph G, there can not be vertical edges, there cannot be cross edges (u, v) with u and v 2 levels apart. (Cross edge at most 1 level apart)
In a DFS tree T of a graph G, there can not be cross edges, there can only be tree edges and vertical edges.

Definition 2.6.2 (bridge)**.** Given a connected graph $G = (V, E)$, an edge $e \in E$ is called a **bridge** if the graph $G = (V, E \setminus \{e\})$ is disconnected.

The idea to find bridge is through a DFS tree. Notice that there are only tree edges and vertical edges in DFS tree. Vertical edges are not bridges, a tree edge (u, v) is not a bridge if some vertical edge jumping from below u to above v. Other tree edges are bridges.
Define $level(v)$ as the level of vertex v in DFS tree. T_v as the sub tree rooted at v, $h(v)$ as the smallest level that can be reached using a vertical edge from vertices in T_v. $(parent(u), u)$ is a bridge if $h(u) \geq level(u)$. The algorithm is as following:

Algorithm 13 FindBridge(G)

```
1: Mark all vertices as "unvisited"
2: for v ∈ V do
3:     if v is "unvisited" then
4:         level(v) ← 0
5:         RecursiveDFS(v)
```

2.7 Blocks

Algorithm 14 RecursiveDFS(v)

```
1: mark v as "visited"
2: h(v) ← ∞
3: for u ∈ d_v do
4:     if u is "unvisited" then
5:         level(u) ← level(v) + 1
6:         RecursiveDFS(u)
7:         if h(u) ≥ level(u) then
8:             (u, v) is a bridge
9:         if h(u) < h(v) then
10:            h(v) ← h(u)
11:    else
12:        if level(u) < level(v) − 1 and level(u) < h(v) then
13:            h(v) ← level(u)
```

Chapter 3

Euler Tours and Hamilton Cycles

Chapter 4

Matriod, Planarity

4.1 Plane and Planar Graphs

4.2 Dual Graphs

4.3 Matroids

Definition 4.3.1 (Matroids)**.** Let S be a finite set of **elements** and let d be a collection of subsets of S satisfying the property

$$\text{If } x \leq y, y \in d, \Rightarrow x \in d \tag{4.1}$$

The pair (S, d) is called an **independent system** and the members of d are called **independent sets**.

Example. Let G be a graph and let $S \in E(G)$ define $M \subseteq S$ to be independent if M is a matching

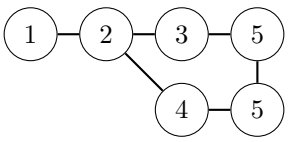
$$S = \{(1,2),(2,3),(2,4),(3,5),(4,6),(5,6)\} \tag{4.2}$$
$$d = \{\emptyset, \{(1,2)\}, \{(2,3)\}, ..., \{\text{other matching...}\}\} \tag{4.3}$$

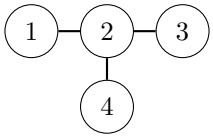
Example. Let G be a graph and let $S = V(G)$ define $X \subseteq S$ to be independent if no two member of x are adjacent.

$$S = \{1,2,3,4\} \tag{4.4}$$
$$d = \{\emptyset, 1, 2, 3, 4, (1,3), (1,4), (3,4), (1,3,4)\} \tag{4.5}$$

Example. Let G be a connected graph and let $S = E(G)$, define $X \subseteq S$ to be independent if $G[X]$ contains cycles.

Given any independent system, there is a natural combinatorial optimization problem of finding the maximum cardinality independent set.
Let's try the following: **Greedy algorithm**
Step 1: Set $I = \emptyset$
Step 2: If there exists $e \in S \setminus I$ such that $I + e$ is independent, set $I \leftarrow I + e$ and go to Step 1, otherwise stop.
Those Independent systems for which the greedy algorithm guarantee to find a maximum cardinality independent set are very special called **matroids**





4.4 Independent Sets

4.5 Ramsey's Theorem

4.6 Turán's Theorem

4.7 Schur's Theorem

4.8 Euler's Formula

4.9 Bridges

4.10 Kuratowski's Theorem

4.11 Four-Color Theorem

4.12 Graphs on other surfaces

Chapter 5

Minimum Spanning Tree Problem

5.1 Basic Concepts

Example. A company wants to build a communication network for their offices. For a link between office v and office w, there is a cost c_{vw}. If an office is connected to another office, then they are connected to with all its neighbors. Company wants to minimize the cost of communication networks.

Definition 5.1.1 (Cut vertex)**.** A vertex v of a connected graph G is a **cut vertex** if $G \setminus v$ is not connected.

Definition 5.1.2 (Connection problem)**.** Given a connected graph G and a positive cost C_e for each $e \in E$, find a minimum-cost spanning connnected subgraph of G. (Cycles all allowed)

Lemma 5.1. *An edge $e = uv \in G$ is an edge of a cycle of G iff there is a path $G \setminus e$ from u to v.*

Definition 5.1.3 (Minumum spanning tree problem)**.** Given a connected graph graph G, and a cost $C_e, \forall e \in E$, find a minimum cost spanning tree of G

The only way a connection problem will be different than MSP is if we relax the restriction on $C_e > 0$ in the connection problem.

5.2 Kroskal's Algorithm

Algorithm 15 Kroskal's Algorithm, $O(m \log m)$

Require: A connected graph
Ensure: A MST
  Keep a spanning forest $H = (V, F)$ of G, with $F = \emptyset$
  while $|F| < |V| - 1$ **do**
    add to F a least-cost edge $e \notin F$ such that H remains a forest.

5.3 Prim's Algorithm

Algorithm 16 Prim's Algorithm, $O(nm)$

Require: A connected graph
Ensure: A MST
  Keep $H = (V(H), T)$ with $V(H) = \{v\}$, where $r \in V(G)$ and $T = \emptyset$
  while $|V(T)| < |V|$ **do**
    Add to T a least-cost edge $e \notin T$ such that H remains a tree.

5.4 Comparison between Kroskal's and Prim's Algorithm

- Kroskal start with a forest that contains all vertices, Prim start with a tree that only contain one vertex.
- Kroskal cannot gurantee every step it is a tree but can gurantee it is spanning, Prim can gurantee every step it is a tree but cannot gurantee spanning.

5.5 Extensible MST

Definition 5.5.1 (cut)**.** For a graph $G = (V, E)$ and $A \subseteq V$ we denote $\delta(A) = \{e \in E : e \text{ has an end in } A \text{ and an end in } V \setminus A\}$. A set of the form $\delta(A)$ for some A is called a **cut** of G.

Definition 5.5.2. We also define $\gamma(A) = \{e \in E : \text{both ends of } e \text{ are in } A\}$

Theorem 5.2. *A graph $G = (V, E)$ is connected iff there is no $A \subseteq V$ such that $\emptyset \neq A \neq V$ with $\delta(A) = \emptyset$*

Definition 5.5.3. Let us call a subset $A \in E$ **extensible** to a minimum spanning tree problem if A is contained in the edge set of some MST of G

Theorem 5.3. *Suppose $B \subseteq E$, that B is extensible to an MST and that e is a minimum cost edge of some cut D satisfying $D \cap B = \emptyset$, then $B \cup \{e\}$ is extensible to an MST.*

5.6 Solve MST in LP

Given a connected graph $G = (V, E)$ and a cost on the edges C_e for all $e \in E$, Then we can formulate the following LP

$$X_e = \begin{cases} 1, \text{if edge } e \text{ is in the optimal solution} \\ 0, \text{otherwise} \end{cases} \tag{5.1}$$

The formulation is as following

$$\min \quad \sum_{e \in E} c_e x_e \tag{5.2}$$

$$\text{s.t.} \quad \sum_{e \in E} x_e = |V| - 1 \tag{5.3}$$

$$x_e \geq 0 \tag{5.4}$$

$$e \in E \tag{5.5}$$

$$\sum_{e \in E(S)} x_e = |S| - 1, \forall S \subseteq V, S \neq \emptyset \tag{5.6}$$

$$\tag{5.7}$$

Chapter 6

Shortest-Path Problem

6.1 Basic Concepts

All Shortest-Path methods are based on the same concept, suppose we know there exists a dipath from r to v of a cost y_v. For each vertex $v \in V$ and we find an arc $(v, w) \in E$ satisfying $y_v + v_{vw} < y_w$. Since appending (v, w) to the dipath to v takes a cheaper dipath to w then we can update y_w to a lower cost dipath.

Definition 6.1.1 (feasible potential)**.** We call $y = (y_v : v \in V)$ a **feasible potential** if it satisfies

$$y_v + c_{vw} \geq y_w \quad \forall (v, w) \in E \tag{6.1}$$

and $y_r = 0$

Proposition 1. *Feasible potential provides lower bound for the shortest path cost.*

Proof. Suppose that you have a dipath $P = v_0 e_1 v_1, ..., e_k v_k$ where $v_0 = r$ and $v_k = v$, then

$$C(P) = \sum_{i=1}^{k} C_{e_i} \geq \sum_{i=1}^{k} (y_{v_i} - y_{v_{i-1}}) = y_{v_k} - y_{v_0} \tag{6.2}$$

□

6.2 Breadth-First Search Algorithm

6.3 Ford's Method

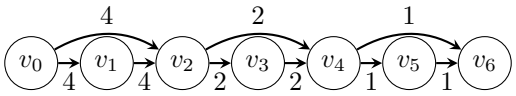
Define a predecessor function $P(w), \forall w \in V$ and set $P(w)$ to v whenever y_w is set to $y_v + c_{vw}$

Algorithm 17 Ford's Method

Ensure: Shortest Paths from r to all other nodes in V
Require: A digraph with arc costs,starting node r
  Initialize, $y_r = 0$ and $y_v = \infty, v \in V \setminus r$
  Initialize, $P(r) = 0, P(v) = -1, \forall v \in V \setminus r$
  while y is not a feasible potential **do**
    Let $e = (v, w) \in E$ (this could be problematic)
    if $y_v + c_{vw} < y_w$ (incorrect) **then**
      $y_w \leftarrow y_v + c_{vw}$ (correct it)
      $P(w) = v$ (set v as predecessor)

Notice: Technically speaking, this is not an algorithm, for the following reasons: 1) We did not specify how to pick e, 2) This procedure might not stop given some situations, e.g., if there is a cycle with minus total weight

Notice: This method can be really bad. Here is another example that could take $O(2^n)$ to solve.



6.4 Ford-Bellman Algorithm

Algorithm 18 Ford-Bellman Algorithm

Ensure: Shortest Paths from r to all other nodes in V
Require: A digraph with arc costs,starting node r
  Initialize y and p
  for $i = 0; i < N; i++$ **do**
    for $\forall e = (v, w) \in E$ **do**
      if $y_v + c_{vw} < y_w$ (incorrect) **then**
        $y_w \leftarrow y_v + c_{vw}$ (correct it)
        $P(w) = v$ (set v as predecessor)
  for $\forall e = (v, w) \in E$ **do**
    if $y_v + c_{vw} < y_w$ (incorrect) **then**
      Return error, negative cycle

Notice: Only correct the node that comes from a node that has been corrected.

A usual representation of a digraph is to store all the arcs having tail v in a list L_v to **scan** v means the following:

- For $(v, w) \in L_v$, if (v, w) is incorrect, then correct (v, w)

For Bellman, will either terminate with shortest path from r to all $v \in V \setminus r$ or it will terminate stating that there is a negative cycle. In $O(mn)$
In the algorithm if $i = n$ and there exists a feasible potential, the problem has a negative cycle.
Suppose that the nodes of G can be ordered from left to right so that all arcs go from left to right. That is suppose there is an ordering $v_1, v_2, ..., v_n \in V$ so that $(v_i, v_j) \in V$ implies $i < j$. We call such an ordering **topological** sort. If we order E in the sequence that $v_i v_j$ precedes $v_k v_i$ if $i < k$ based on topological order then ford algorithm will terminate in one pass.

6.5 SPFA Algorithm

6.6 Dijkstra Algorithm

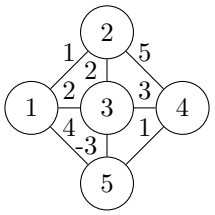
6.7 A* Algorithm

6.8 Floyd-Warshall Algorithm

If all weights/distances in the graph are nonnegative then we could use Dijkstra within starting nodes being any one of the vertices of the graph. This method will take $O(n^3)$
If weight/distances are arbitrary and we would like to find shortest path between all pairs of vertices or detect a negative cycle we could use Bellman-Ford Algorithm with $O(n^4)$
We would like an algorithm to find shortest path between any two pairs in a graph for arbitrary weights (determined, negative, cycles) in $O(n^3)$

Algorithm 19 Dijkstra Algorithm

Ensure: Shortest Paths from r to all other nodes in V
Require: A digraph with arc costs,starting node r
  Initialize y and p
  $S \leftarrow V$
  while $S \neq \emptyset$ **do**
    Choose $v \in S$ with minimum y_v
    $S \leftarrow S \setminus v$
    for $\forall w, (v, w) \in E$ **do**
      if $y_v + c_{vw} < y_w$ (incorrect) **then**
        $y_w \leftarrow y_v + c_{vw}$ (correct it)
        $P(w) = v$ (set v as predecessor)

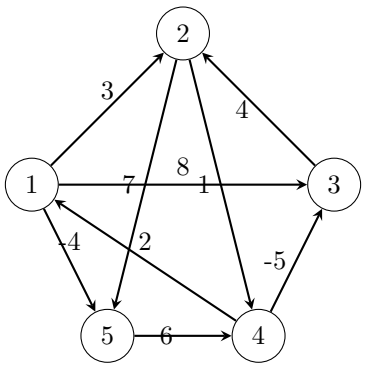


Let d_{ij}^k denote the length of the shortest path from i to j such that all intermediate vertices are contained in the set $\{1, ..., k\}$
In this case $d_{14}^5 = 5$
If the vertex k is not an intermediate vertex on p, then $d_{ij}^k = d_{ij}^{k-1}$, notice that $d_{15}^4 = -1$, node 4 is not intermediate, so $d_{15}^3 = -1$
If the vertex k is an intermediate on p, then $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$, $d_{14}^5 = 0$ ($p = 1 \to 3 \to 5 \to 4$), i.e., $d_{14}^5 = d_{15}^4 + d_{54}^4 = 0$
Therefore $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$
Input: graph $G = (V, E)$ with weight on edges Output: Shortest path between all pairs of vertices on existence of a negative cycle Step 1: Initialize

$$d_{ij}^0 = \begin{cases} c_{ij} & \text{distance from } i \text{ to } j \text{ if } (i,j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \notin E \end{cases} \tag{6.3}$$

Step: For k = 1 to n For i = 1 to n For j = 1 to n $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$ Next j Next i Next k Between optimal matrix D^n

$$D^0 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{6.4}$$



$$\Pi^0 = \begin{bmatrix} & 1 & 1 & & 1 \\ & & & 2 & 2 \\ & 3 & & & \\ 4 & & 4 & & \\ & & & 5 & \end{bmatrix} \tag{6.5}$$

$$D^1 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \mathbf{5} & -5 & 0 & \mathbf{-2} \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{6.6}$$

$$\Pi^1 = \begin{bmatrix} & 1 & 1 & & 1 \\ & & & 2 & 2 \\ & 3 & & & \\ 4 & \mathbf{1} & 4 & & \mathbf{1} \\ & & & 5 & \end{bmatrix} \tag{6.7}$$

$$D^2 = \begin{bmatrix} 0 & 3 & 8 & \mathbf{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \mathbf{5} & \mathbf{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{6.8}$$

$$\Pi^2 = \begin{bmatrix} & 1 & 1 & \mathbf{2} & 1 \\ & & & 2 & 2 \\ & 3 & & \mathbf{2} & \mathbf{2} \\ 4 & 1 & 4 & & 1 \\ & & & 5 & \end{bmatrix} \tag{6.9}$$

$$D^3 = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & \mathbf{-1} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{6.10}$$

$$\Pi^3 = \begin{bmatrix} & 1 & 1 & 2 & 1 \\ & & & 2 & 2 \\ & 3 & & 2 & 2 \\ 4 & \mathbf{3} & 4 & & 1 \\ & & & 5 & \end{bmatrix} \tag{6.11}$$

$$D^4 = \begin{bmatrix} 0 & 3 & \mathbf{-1} & 4 & -4 \\ \mathbf{3} & 0 & \mathbf{-4} & 1 & \mathbf{-1} \\ \mathbf{7} & 4 & 0 & 5 & \mathbf{3} \\ 2 & -1 & -5 & 0 & -2 \\ \mathbf{8} & \mathbf{5} & \mathbf{1} & 6 & 0 \end{bmatrix} \tag{6.12}$$

$$\Pi^4 = \begin{bmatrix} & 1 & \mathbf{4} & 2 & 1 \\ \mathbf{4} & & \mathbf{4} & 2 & \mathbf{1} \\ \mathbf{4} & 3 & & 2 & \mathbf{1} \\ 4 & 3 & 4 & & 1 \\ \mathbf{4} & \mathbf{3} & \mathbf{4} & 5 & \end{bmatrix} \tag{6.13}$$

$$D^5 = \begin{bmatrix} 0 & \mathbf{1} & \mathbf{-3} & \mathbf{2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \tag{6.14}$$

$$\Pi^5 = \begin{bmatrix} & \mathbf{3} & 4 & \mathbf{5} & 1 \\ 4 & & 4 & 2 & 1 \\ 4 & 3 & & 2 & 1 \\ 4 & 3 & 4 & & 1 \\ 4 & 3 & 4 & 5 & \end{bmatrix} \tag{6.15}$$

Time complexity $O(n^3)$
If during the previous processes, there exist an element of negative value in the diagonal, it means there exists negative cycle.

6.9 Johnson's Algorithm

Chapter 7

Maximum Flow Problem

7.1 Basic Concept

Let $D = (V, A)$ be a strict diagraph with distinguished vertices s and t. We call s the source and t the sink, let $u = \{u_e : e \in A\}$ be a nonnegative integer-valued capacity function defined on the arcs of D. The maximum flow problem on (D, s, t, u) is the following Linear program.

$$\max \quad v \tag{7.1}$$

$$\text{s.t.} \quad \sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = \begin{cases} -v, & \text{if } i = s \\ v, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \tag{7.2}$$

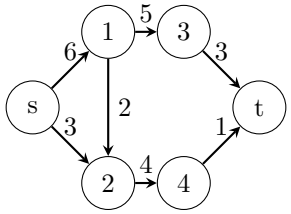
$$0 \le x_e \le u_e, \quad \forall e \in A \tag{7.3}$$

We think of x_e as being the flow on arc e. Constraint says that for $i \neq s, t$ the flow into a vertex has to be equal to the flow out of vertex. That is, flow is conceded at vertex i for $i = s$ and for $i = t$ the net flow in the entire digraph must be equal to v. A $\mathbf{x_e}$ that satisfied the above constraints is an (s, t)-flow of value v. If in addition it satisfies the bounding constraints, then it is a feasible (s, t)-flow. A feasible (s, t)-flow that has maximum v is optimal on maximum.

7.2 Solving Maximum Flow Problem in LP

Theorem 7.1. *For $S \subseteq V$ we define $(S, \bar{S})$ to be a (s, t)-cut if $s \in S$ and $t \in \bar{S} = V - S$, the capacity of the cut, denoted $u(S, \bar{S})$ as $\sum\{u_e : e \in \delta^-(S)\}$ where $\delta^-(S) = \{e \in A : t(e) \in S$ and $h(e) \in \bar{S}\}$*

Example. For the following graph:
Let $S = \{1, 2, 3, s\}$, $\bar{S} = \{4, t\}$

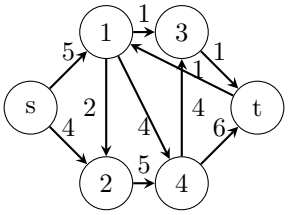


then $\delta^-(S) = \{(2, 4), (3, t)\} \Rightarrow u(S, \bar{S}) = 7$

Definition 7.2.1. If $(S, \bar{S})$ has minimum capacity of all (s, t)-cuts, then it is called **minimum cut**.

Definition 7.2.2. Let $\delta^+(S) = \delta^-(V - S)$

Example. Let $S = \{s, 1, 2, 3\}$, $\bar{S} = \{4, t\}$, $u(S, \bar{S}) = u_{14} + u_{24} + u_{3t} = 10$, $\delta^-(S) = \{(1, 4), (2, 4), (3, t)\}$, $\delta^+ S = \{(4, 3), (t, 1)\}$



Lemma 7.2. *If x is a (s,t) flow of value v and $(S,\bar{S})$ is a (s,t)-cut, then*

$$v = \sum_{e\in\delta^-(S)} x_e - \sum_{e\in\delta^+(S)} x_e \tag{7.4}$$

Proof. Summing the first set of constraints over the vertices of S,

$$\sum_{i\in S}\Big(\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e\Big) = -v \tag{7.5}$$

Now for an arc e with both ends in S, x_e will occur twice once with a positive and once with negative so they cancel and the above sum is reduced to

$$\sum_{e\in\delta^+(S)} x_e - \sum_{e\in\delta^-(S)} x_e = -v \tag{7.6}$$

□

Notice: Flow is the prime variable, capacity is the dual variable.

Corollary 7.2.1. *If x is a feasible flow of value v, and $(S,\bar{S})$ is an (s,t)-cut, then*

$$v \le u(S,\bar{S}) \quad \textit{(Weak duality)} \tag{7.7}$$

Definition 7.2.3. Define an arc e to be **saturated** if $x_e = u_e$, and to be **flowless** if $x_e = 0$

Corollary 7.2.2. *Let x be a feasible flow and $(S,\bar{S})$ be a (s,t)-cut, if $\forall e \in \delta^-(S)$ is saturated, and $\forall e \in \delta^+(S)$ is flowless, then x is a maximum flow and $(S,\bar{S})$ is a minimum cut. (Strong duality)*

Proof. If every arc of $\delta^-(S)$ is saturated then

$$\sum_{e\in\delta^-(S)} x_e = \sum_{e\in\delta^-(S)} u_e \tag{7.8}$$

If every arc of $\delta^+(S)$ is flowless then

$$\sum_{e\in\delta^+(S)} x_e = 0 \tag{7.9}$$

$\Rightarrow x$ is as large as it can get when as $u(S,\bar{S})$ is as small as it can get. □

7.3 Prime and Dual of Maximum Network Flow Problem

The LP of maximum flow can be modeled as following, WLOG, we let $s = v_1 \in V, t = v_{|V|} \in V$.

$$\max \quad f = \begin{bmatrix}0 & 0 & \cdots & 0 & 1\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ f\end{bmatrix} \tag{7.10}$$

$$\text{s.t.} \quad \begin{bmatrix}\mathbf{A} & \mathbf{F}\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ f\end{bmatrix} = \mathbf{0} \tag{7.11}$$

$$\mathbf{I}\mathbf{x} \le \mathbf{u} \tag{7.12}$$

$$\begin{bmatrix}\mathbf{x}\\ f\end{bmatrix} \ge 0 \tag{7.13}$$

In which $\mathbf{A}$ is the vertex-arc incident matrix and $\mathbf{F}$ is a column vector where the first row is -1, last row is 1 and all other rows are 0s, which is because we denote the first vertex as source s and the last vertex as the sink t. $\mathbf{u}$ is the column vector of upper bound of each arcs.

$$\mathbf{A} = \mathbf{A}_{|E|\times|V|} = [a_{ij}], \text{ where } a_{ij} = \begin{cases} 1, & \text{if } v_i = h(e_j) \\ -1, & \text{if } v_i = t(e_j) \\ 0, & \text{otherwise} \end{cases} \tag{7.14}$$

$$\mathbf{F} = \begin{bmatrix} -1 & \cdots & 0 & \cdots & 1 \end{bmatrix}^\top \tag{7.15}$$

$$\mathbf{u} = \begin{bmatrix} u_1 & u_2 & \cdots & u_{|E|} \end{bmatrix}^\top \tag{7.16}$$

Then, we take the dual of LP

$$\min \quad \mathbf{u}\mathbf{w_E} \tag{7.17}$$

$$\text{s.t.} \quad \begin{bmatrix} \mathbf{w_V} & \mathbf{w_E} \end{bmatrix} \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \end{bmatrix} \geq 0 \tag{7.18}$$

$$\begin{bmatrix} \mathbf{w_V} & \mathbf{w_E} \end{bmatrix} \begin{bmatrix} \mathbf{F} \\ \mathbf{0} \end{bmatrix} = 1 \tag{7.19}$$

$$\mathbf{w_V} \quad \text{unrestricted} \tag{7.20}$$

$$\mathbf{w_E} \geq \mathbf{0} \tag{7.21}$$

In which $\mathbf{w_V}$ is "whether or not" vertex v is in S where $(S, \bar{S})$ represents a cut, $\mathbf{w_E}$ is "whether or not" an arc in in $\delta^+(S)$. $\mathbf{u}, \mathbf{E}, \mathbf{F}$ have the same meaning as in prime.

$$\mathbf{w_V} = \begin{bmatrix} w_1 & w_2 & \cdots & w_{|V|} \end{bmatrix}^\top \tag{7.22}$$

$$\mathbf{w_E} = \begin{bmatrix} w_{|V|+1} & w_{|V|+2} & \cdots & w_{|V|+|E|} \end{bmatrix}^\top \tag{7.23}$$

To make it more clear, it can be rewritten as following

$$\min \quad \sum_{e\in E} u_e w_e \tag{7.24}$$

$$\text{s.t.} \quad w_i - w_j + w_{|V|+e} \geq 0, \forall e = (i,j) \in E \tag{7.25}$$

$$-w_1 + w_{|V|} = 1 \tag{7.26}$$

$$\mathbf{w_V} \quad \text{unrestricted} \tag{7.27}$$

$$\mathbf{w_E} \geq \mathbf{0} \tag{7.28}$$

The meaning for the first set of constraint is to decide whether or not an arc is in $\delta^+(S)$ of a $(S, \bar{S})$, which is decided by w_V. The $w_1 - w_{|V|} = 1$, which is the second set of constraint means the source $s = v_1$ and the sink $t = v_{|V|}$ has to be in S and $\bar{S}$ respectively.

7.4 Maximum Flow Minimum Cut Theorem

Definition 7.4.1. Let P be a path, (not necessarily a dipath), P is called **unsaturated** if every **forward** arc is unsaturated ($x_e < u_e$) and ever **reverse** arc has positive flow ($x_e > 0$). If in addition P is an (s,t)-path, then P is called an **x-augmenting path**

Theorem 7.3. *A feasible flow x in a digraph D is maximum iff D has no augmenting paths.*

Proof. (Prove by contradiction)
($\Rightarrow$) Let x be a maximum flow of value v and suppose D has an augmenting path. Define in P (augmenting path):

$$D_1 = \min\{u_e - x_e : e \text{ forward in } P\} \tag{7.29}$$

$$D_2 = \min\{x_e : e \text{ backward in } P\} \tag{7.30}$$

$$D = \min\{D_1, D_2\} \tag{7.31}$$

Since P is augmenting, then $D > 0$, let

$$\hat{x_e} = \begin{cases} x_e + D & \text{If } e \text{ is forward in } P \\ x_e - D & \text{If } e \text{ is backward in } P \\ x_e & otherwise \end{cases} \tag{7.32}$$

It is easy to see that $\hat{x}$ is feasible flow and that the value is $V + D$, a contradiction.
($\Leftarrow$) Suppose D admits no x-augmenting path, Let S be the set of vertices reachable from s by x-unsaturated path clearly $s \in S$ and $t \notin S$ (because otherwise there would be an augmenting path). Thus, $(S, \bar{S})$ is a (s,t)-cut.
Let $e \in \delta^-(S)$ then e must be saturated. For otherwise we could add the $h(e)$ to S
Let $e \in \delta^+(S)$ then e must be flow less. For otherwise we could add the $t(e)$ to S.
According to previous corollary, that x is maximum. □

Theorem 7.4. *(Max-flow = Minimum-cut) For any digraph, the value of a maximum (s,t)-flow is equal to the capacity of a minimum (s,t)-cut*

7.5 Ford-Fulkerson Method

Finding augmenting paths is the key of max-flow algorithm, we need to describe two functions, labeling and scanning a vertex.
A vertex is first labeled if we can find x-unsaturated path from s, i.e., (s,v)-unsaturated path.
The vertex v is scanned after we attempted to extend the x-unsaturated path.
This algorithm is incomplete/incorrect, needs to be fixed

Algorithm 20 Labeling algorithm

Ensure: Max-flow x with value v
Require: Digraph with source s and sink t, a capacity function u and a feasible flow (could be $x_e = 0$)
  Initialize, $v \leftarrow x$
  Designate all vertices as unlabeled and unscanned
  Label s
  while There exists vertex unlabeled or unscanned **do**
    Let i be such a vertex, for each arc e with $t(e) = i, x_e < u_e$ and $h(e)$ unlabeled, label $h(e)$
    For each arc e with $h(e) = i, x_e > 0$ and $t(e)$ unlabeled, label $t(e)$, designate i as scanned.
    If t is not label
  x is the maximum.

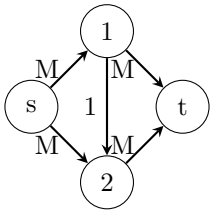
Algorithm 21 Ford-Fulkerson algorithm

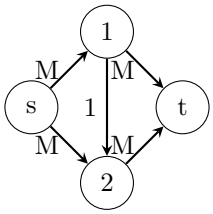
Ensure: Max-flow x with value v
Require: Digraph with source s and sink t, a capacity function u and a feasible flow (could be $x_e = 0$)
  Initialize, $v \leftarrow x$
  Designate all vertices as unlabeled and unscanned
  Label s
  while There exists vertex unlabeled or unscanned **do**
    Let i be such a vertex, for each arc e with $t(e) = i, x_e < u_e$ and $h(e)$ unlabeled, label $h(e)$
    For each arc e with $h(e) = i, x_e > 0$ and $t(e)$ unlabeled, label $t(e)$, designate i as scanned.
    If t is not label
  x is the maximum.

Labeling algorithm can be exponential, the following is an example

7.6 Polynomial Algorithm for max flow

Let (D, s, t, u) be a max flow problem and let x be a feasible flow for D, the **x-layers** of D are define be the following algorithm





Layer algorithm (Dinic 1977) Input: A network (D, s, t, u) and a feasible flow x Output: The **x-layers** $V_0, V_1, ..., V_l$ where $V_i \cap V_j = \emptyset \forall i \neq j$
Step 1: Set $V_0 = \{s\}, i \leftarrow 0 and l(x) = 0$ Step 2: Let R be the set of vertices w such that there is an arc e with either:

- $t(e) \in V_i, h(e) = w, x_e < u_e$ or
- $h(e) \in V_j, t(e) = w, x_e > 0$

Step 3: If $t \in R$, set $V_{i+1} = \{t\}$, $l(t) = i + 1$ and stop. Set $V_{i+1} \leftarrow R \setminus \cup_{0 \leq j \leq i} V_j$, $l \leftarrow i + 1, l(x) = i$, goto Step 2. If $V_{i+1} = \emptyset$, set $l(x) = i$ and Stop.

Example. For the following graph

$$\text{Second iteration} \tag{7.33}$$
$$V_0 = \{s\}, i = 0, l(x) = 0 \tag{7.34}$$
$$R = \{1, 2\} \tag{7.35}$$
$$V_1 \leftarrow \{1, 2\}, i = 1, l(x) = 1 \tag{7.36}$$
$$R = \{3, 4, 5\} \tag{7.37}$$
$$V_2 \leftarrow \{3, 4\}, i = 2, l(x) = 2 \tag{7.38}$$
$$R = \{1, 5, 6, 3\} \tag{7.39}$$
$$V_3 \leftarrow \{5, 6\}, i = 3, l(x) = 3 \tag{7.40}$$
$$R = \{4, t\} \tag{7.41}$$
$$V_4 = \{t\} \tag{7.42}$$
$$A_1 = \{(s, 1), (s, 2)\} \tag{7.43}$$
$$A_2 = \{(1, 3), (2, 4)\} \tag{7.44}$$
$$A_3 = \{(3, 5), (4, 6)\} \tag{7.45}$$
$$A_4 = \{(5, t), (6, t)\} \tag{7.46}$$

The layer network D_x is defined by $V(D_x) = V_0 \cup V_1 \cup V_2 \cdots \cup V_{l(x)}$
Suppose we have computed the layers of D and $t \in V_l(x)$, the last layer (last layer I am goin to V_e)
For each $i, 1 \leq i \leq l$, define a set of arcs A_i and a function $\hat{u}$ on A_i as following. For each $e \in A(D)$

- If $t(e) \in V_{i-1}, h(e) \in V_i$ and $x_e < u_e$ then add arc e to A_i and define $\hat{u}_e = u_e - x_e$
- If $h(e) \leftarrow V_{i-1}, t(e) \in V_i$ and $x_e > 0$ then add arc $e' = (h(e), t(e))$ to A_i with $\hat{u}_e - x_e$

Let $\hat{u}$ be the capacity function on D_x and let the source and sink of D_x be s and t
We can think of D_x as being make of arc shortest (in terms of numbers of arcs) x-augmenting paths.
A feasible flow in a network is said to be maximal (does not means maximum) if every (s, t)-directed path contains at least one saturated arc.
For layered algorithm $V_0, V_1, ..., V_L$
Arcs:

- If $t(e) \in V_{i-1}$, $h(e) \in V_i$ and $x_e < u_e$, then add e to A_i with $\hat{u_e} = u_e - x_e$
- If $h(e) \in V_{i-1}$, $t(e) \in V_i$ and $x_e > 0$, then add arc $e' = (h(e), t(e))$ to A_i and define $\hat{u_e} = x_e$

Maximal Flow: If every directed (s,t)-path has at least one saturated arc.
Computing maximal flow is easier than computing maximum flow, since we never need to consider canceling flows on reverse arcs,
Let $\hat{x}$ be a maximal flow on the layered network D_x, we can define new flows in $D(x')$ by

$$x'_e = x_e + \hat{x_e}, \quad \text{If } t(e) \in V_{i-1}, h(e) \in V_i \tag{7.47}$$
$$x'_e = x_e - \hat{x_e}, \quad \text{If } h(e) \in V_{i-1}, t(e) \in V_i \tag{7.48}$$

7.7 Dinic Algorithm

Input: A layered network $(D_x, s, t, \hat{u})$ and a feasible flow x Output: A maximal flow $\hat{x}$ from D_x
Step 1: Set $H \leftarrow D_x$ and $i \leftarrow S$ Step 2: If there is no arc e with $t(e) = i$, goto Step 4, otherwise let e be such an arc Step 3: Set $T(h(e)) \leftarrow i$ and $i \leftarrow h(e)$, if $i = t$ goto Step 5, otherwise goto Step 2. Step 4: If $i = s$, Stop, Otherwise delete i and all incident arcs with H, set $i \leftarrow T(i)$ and goto Step 2 Step 5: Construct the directed path, $s = i_0 e_1 i_1 e_2 ... e_k i_k = t$ where $i_{j-1} = T(i_j), 1 \leq j \leq k$. Set $D = \min\{\hat{u_{e_j}} - \hat{x_{e_j}} : i \leq j \leq k\}$, set $\hat{x_{e_j}} \leftarrow \hat{x_{e_j}} + D, i \leq j \leq k$. Delete from H all saturated arcs on this path, set $i \leftarrow 1$ and goto Step 2.

Algorithm 22 Dinic Algorithm

Ensure: A maximal flow $\hat{x}$ from D_x
Require: A layered network $(D_x, s, t, \hat{u})$ and a feasible flow x
Initialize $H \leftarrow D_x$ and $i \leftarrow S$

Theorem 7.5. *Dinic algorithm runs in $O(|E||V|^2)$*

Proof. Step 1 is $O(|E||V|)$ Step 2 runs Step 1 for $O(|V|)$ times □

Chapter 8

Minimum Weight Flow Problem

8.1 Transshipment Problem

Transshipment Problem (D, b, w) is a linear program of the form

$$\begin{aligned} \min \quad & wx && (8.1)\\ \text{s.t.} \quad & Nx = b && (8.2)\\ & x \geq 0 && (8.3) \end{aligned}$$

Where N is a vertex-arc incident matrix. For a feasible solution to LP to exist, the sum of all bs must be zero. Since the summation of rows of N is zero. The interpretation of the LP is as follows.
The variables are defined on the edges of the digraph and that x_e denote the amount of flow of some commodity from the tail of e to the head of e
Each constraints

$$\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = b_i \tag{8.4}$$

represents consequential of flow of all edges into k vertex that have a demand of $b_i > 0$, or a supply of $b_i < 0$. If $b_i = 0$ we call that vertex a transshipment vertex.

8.2 Network Simplex Method

Lemma 8.1. *Let C_1 and C_2 be distinct cycles in a graph G and let $e \in C_1 \cup C_2$. Then $(C_1 \cup C_2) \setminus e$ contains a cycle.*

Proof. Case 1: $C_1 \cap C_2 = \emptyset$. Trivia.
Case 2: $C_1 \cap C_2 \neq \emptyset$. Let $e \in C_2$ and $f = uv \in C_1 \setminus C_2$. Starting at v traverse C_1 in the direction away from u until the first vertex of C_2, say x. Denote the (v, x)-path as P. Starting at u traverse C_1 in the direction away from v until the first vertex of C_2, say y. Denote the (u, y)-path as Q. C_2 is a cycle, there are two (x, y)-path in C_2. Denote the (x, y)-path without e as R. Then $vPxRyQ^{-1}uf$ is a cycle. □

Theorem 8.2. *Let T be a spanning tree of G. And let $e \in E \setminus T$ then $T + e$ contains a unique cycle C and for any edge $f \in C$, $T + e - f$ is a spanning tree of G*

Let (D, b, w) be a transshipment problem. A feasible solutions x is a **feasible tree solution** if there is a spanning tree T such that $||x|| = \{e \in A, x_e \neq 0\} \subseteq T$.
The strategy of network simplex algorithm is to generate negative cycles, if negative cycle exists, it means the solution can be improved.
For any tree T of D and for $e \in A \setminus T$, it follows from above theorem that $T + e$ contains a unique cycle. Denote that cycle $C(T, e)$ and orient it in the direction of e, define

$$\begin{aligned} w(T, e) = & \sum \{w_e : e \text{ forward in } C(T, e)\} \\ & - \sum \{w_e : e \text{ reverse in } C(T, e)\} \end{aligned} \tag{8.5}$$

We think of $w(T,e)$ as the weight of $C(T,e)$.

8.2.1 Network Simplex Method

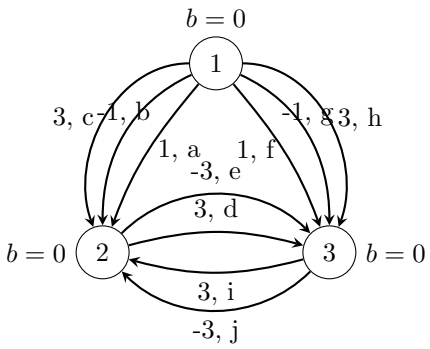
Algorithm 23 Network Simplex Method Algorithm

Ensure: An optimal solution or the conclusion that (D,b,w) is unbounded
Require: A transshipment problem (D,b,w) and a feasible tree solution x containing to a spanning tree T
  while $\exists e \in A \setminus T, w(T,e) < 0$ **do**
    let $e \in A \setminus T$ be such that $w(T,e) < 0$.
    if $C(T,e)$ has no reverse arcs **then**
      Return unboundedness
    else
      Set $\theta = \min\{x_f : f \text{ reverse in } C(T,e)\}$ and set $f = \{f \in C(T,e) : f \text{ reverse in } C(T,e), x_f = \theta\}$
      if f forward in $C(T,e)$ **then**
        $x_f \leftarrow x_f + \theta$
      else
        $x_f \leftarrow x_f - \theta$
      Let $f \in F$ and $T \leftarrow T + e - f$
  Return x as optimal

8.2.2 Example for cycling

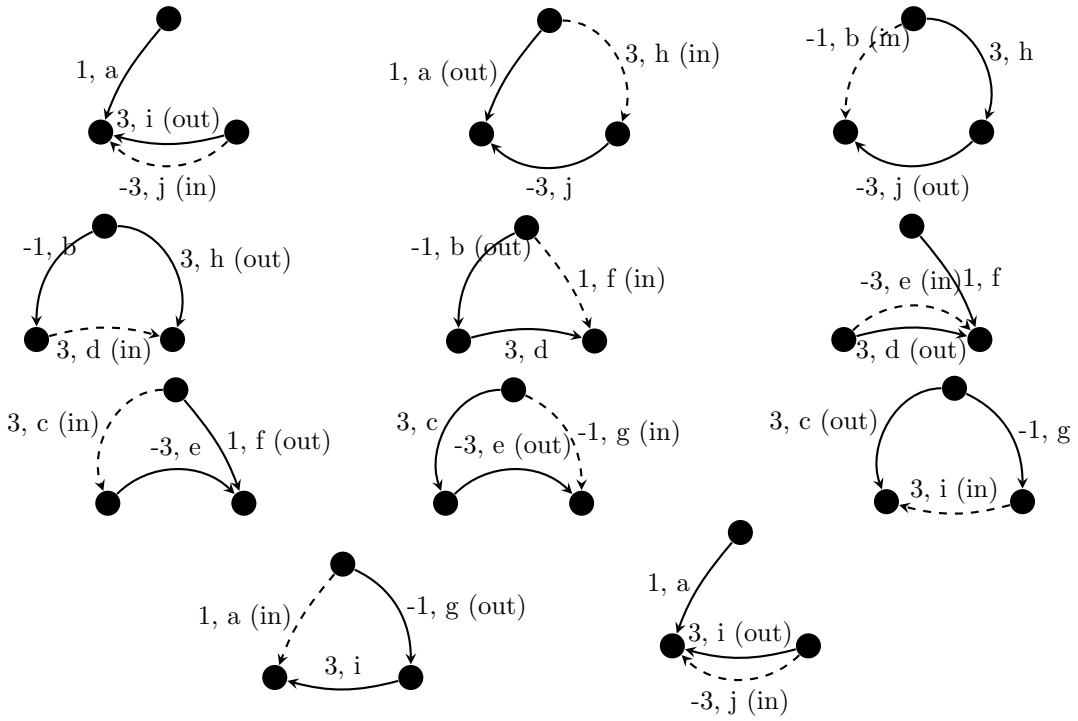
Notice: Similar to Simplex Method in LP, even though in worst case may be inefficient. In most cases it is simple and empirically efficient. Also, similarily, there will be cycling problems.

The following is an example of cycling



Then for the following steps we can detect cycling:
$w(T,j) = w_j - w_i = -3 - 3 = -6$, therefore j in entering basis, i is leaving basis.
$w(T,h) = w_h + w_j - w_a = 3 - 3 - 1 = -1$, therefore h is entering basis, a is leaving basis.
$w(T,b) = w_b - w_j - w_h = -1 + 3 - 3 = -1$, therefore b is entering basis, j is leaving basis.
$w(T,d) = w_d - w_h + w_b = 3 - 3 - 1 = -1$, therefore d is entering basis, h is leaving basis.
$w(T,f) = w_f - w_d - w_b = 1 - 3 + 1 = -1$, therefore f is entering basis, b is leaving basis.
$w(T,e) = w_e - w_d = -3 - 3 = -6$, therefore e is entering basis, d is leaving basis.
$w(T,c) = w_c + w_e - w_f = 3 - 3 - 1 = -1$, therefore c is entering basis, f is leaving basis.
$w(T,g) = w_g - w_e - w_c = -1 + 3 - 3 = -1$, therefore g is entering basis, e is leaving basis.
$w(T,i) = w_i - w_c + w_g = 3 - 3 - 1 = -1$, therefore i is entering basis, c is leaving basis.
$w(T,a) = w_a - w_i - w_g = 1 - 3 + 1 = -1$, therefore a is entering basis, g is leaving basis.

The last graph is the same as the first graph, i.e., cycling detected.



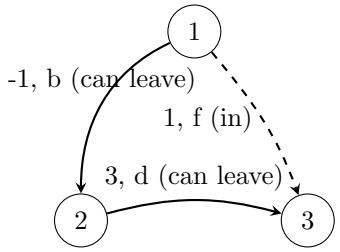
8.2.3 Cycling prevention

To Avoid cycling we will introduce the Modified Network Simplex Method. Let T be a **rooted** spanning tree. Let f be an arc in T, we say f is **away** from the root r if $t(f)$ is the component of $T - f$. Otherwise we say f is **towards** r.
Let x be a feasible tree solution associated with T, then we say T is a **strong feasible tree** if for every arc $f \in T$ with $x_f = 0$ then f is away from $r \in T$.
Modification to NSM:

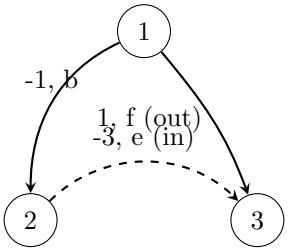
- The algorithm is initialed with a strong feasible tree.
- f in pivot phase is chosen to be the first reverse arc of $C(T, e)$ having $x_f = \theta$. By "first", we mean the first arc encountered in traversing $C(T, e)$ in the direction of e, starting at the vertex i of $C(T, e)$ that minimizes the number of arcs in the unique (r, i)-path in T.

Notice: In the second rule above, r could also be in the cycle, in that case, i is r.

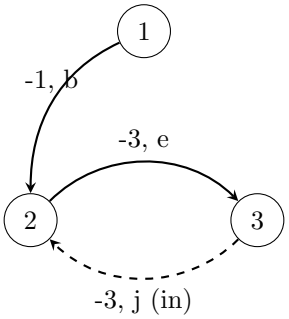
Continue the previous example. Now should how we can avoid cycling:
The first few (four) steps are the same as previous example, starting from
$w(T, f) = w_f - w_d - w_b = 1 - 3 + 1 = -1$. f is entering basis, both b and d can leave the basis, according to



the modified pivot rule, we choose the "first" arc encountered in traversing $C(T, e)$, which is d to leave the basis, instead of b.
$w(T, e) = w_e - w_f + w_b = -5$, e is entering basis, f is leaving basis. Now the only arc to enter basis and maintain

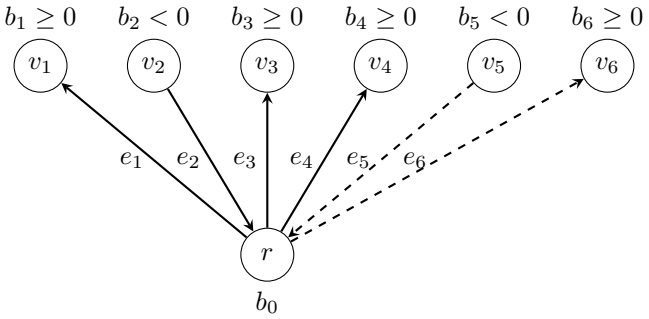


negative w is j.
$w(T, j) = w_j + w_e = -6$, but in $C(T, j)$ there is no reversing arc, therefore we detect unboundedness.



8.2.4 Finding Initial Strong Feasible Tree

Pick a vertex in D to be root r. The tree T has an arc e with the $t(e) = r$ and $h(e) = v$. For each $v \in V \setminus r$ with $b_v \geq 0$ and has an arc e with $h(e) = r$ and $t(e) = v$ for each $v \in V \setminus r$ for which $b_v < 0$. Wherever possible the arcs of T are chosen from A, where an appropriate arc doesn't exist. We create an **artificial arc** and give its weight $|V|(\max\{w_e : e \in A\}) + 1$. This is similar to Big-M method and if optimal solution contains artificial arcs ongoing arc problem is infeasible.
Here is an example after adding artificial arcs:
Where e_5 and e_6 are artificial arcs, the weight of those arcs are $|V|(\max\{w_e : e \in \mathcal{A}\}) + 1$. And the above tree is



a basic feasible solution.
We need to prove that such artificial arc has sufficiently large weight to guarantee

- It will leave the basis, and
- It will not enter the basis again (for this, just delete the artificial arc after it leaves the basis, then it will never enter the basis again)

Proof. Now prove that such arcs will always leave the basis. Before the prove we give some notation.

- Define set E as the set of arcs which is not artificial arc, in the above example, $E = \{e_1, e_2, e_3, e_4\}$.
- Define set A as the set of arcs which are artificial arcs, in the above example, $A = \{e_5, e_6\}$. Noticed that $E \cap A = \emptyset$.

- Define set M as the vertices in the spanning tree that is reachable from r by E, in the above example, $M = \{v_1, v_2, v_3, v_4\}$.
- Define $M' = (V \setminus r) \setminus M$ in the tree that can only be reached from r by A, i.e., artificial arcs, in the above example, $M' = \{v_5, v_6\}$.

Then the initial basic feasible solution is a graph

$$G_0 =< M \cup M' \cup \{r\}, E \cup A > \tag{8.6}$$

Denote the origin graph

$$G =< V, \mathcal{A} > \tag{8.7}$$

Notice that with the artificial arcs, G_0 is not a subgraph of G.
Let $(M \cup \{r\}, M')$ be a cut in the origin graph G. For the vertices in M', one of the following cases will happen:

- case 1: $\sum_{v \in M'} b_v \geq 0$
- case 2: $\sum_{v \in M'} b_v < 0$

For case 1, we claim that at least one of the vertices $v_{M'} \in M'$ with $b_{v_{M'}} \geq 0$ linked by an arc, say f, such that $h(f) = v_{M'}$ and $t(f) = v_M \in M$. Otherwise the balance of flow cannot hold in the origin graph G. Furthermore, denote the artificial arc from r to $v_{M'}$ by $e_{rv_{M'}}$.
Notice that for v_M there is not necessarily be an arc between r and v_M, but there must exists an (r, v_M)-path denoted by P, for M is the set of vertices that reachable from r by arcs in E.
Take that arc f as entering arc to the basis. Then

$$C(T, f) = re_{rv_{M'}} v_{M'} f v_M P r \tag{8.8}$$

For

$$w(T, f) = w_f - w_{e_{rv_{M'}}} + \sum_{e \in P} d_e w_e \tag{8.9}$$

where $d_e = 1$ if w_e is forward in P and $d_e = -1$ otherwise.
Now that $w_{e_{rv_{M'}}} = |V|(\max\{w_e : e \in \mathcal{A}\}) + 1$, it guarantees that

$$\begin{aligned} w(T, f) &= w_f + \sum_{e \in P} d_e w_e - w_{e_{rv_{M'}}} && (8.10)\\ &\leq w_f + \sum_{e \in P} w_e - w_{e_{rv_{M'}}} && (8.11)\\ &\leq \sum_{e \in \mathcal{A}} w_e - w_{e_{rv_{M'}}} && (8.12)\\ &\leq |V|(\max\{w_e : e \in \mathcal{A}\}) - w_{e_{rv_{M'}}} && (8.13)\\ &\leq -1 < 0 && (8.14) \end{aligned}$$

So f can enter the basis, and the artificial variable $e_{rv_{M'}}$ will leave the basis, for it is the most violated reverse arc in the $C(T, f)$. When we put f into the basis, update G_0, such that $M \leftarrow M \cup \{v_{M'}\}$ and $M' \leftarrow M' \setminus \{v_{M'}\}$.
For case 2, it is similar. At least one of the vertices $v_{M'} \in M'$ with $b_{v_{M'}} < 0$ linked by an arc, say $f\prime$, such that $t(f') = v_{M'}$ and $h(f') = v_M \in M$. Otherwise the balance of flow cannot hold in the origin graph G. Furthermore, denote the artificial arc from $v_{M'}$ to r by $e_{v_{M'}r}$.
Similarly we can find a cycle $C(T, f') = rP'v_M f' v_{M'} e_{v_{M'}r} r$. $w(T, f') = w_{f'} - w_{e_{rv_{M'}}} + \sum_{e \in P'} d_e w_e$, where $d_e = 1$ if w_e is forward in $P\prime$ and $d_e = -1$. We can prove $w(T, f') \leq -1 < 0$. That that f' as entering arc to the basis, similarly move $v_{M'}$ form set M' to M.
The above case can be dealt with iteratively until set M' become $\emptyset$, at which stage there is no artificial arc in the basic feasible solution. Which means all the artificial variable can leave the basis. □

Notice: This algorithm can be really bad, its mimic of Simplex Method of LP, which means we can run into exponential operations

8.3 Transshipment Problem and Circulation Problem

Definition 8.3.1. The minimum weight circulation problem is defined as follows:

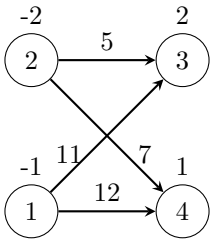
$$\min \quad wx \tag{8.15}$$
$$\text{s.t.} \quad Nx = 0 \tag{8.16}$$
$$l \leq x \leq u \tag{8.17}$$

It turns out that the circulation problem is equivalent with transshipment problem.

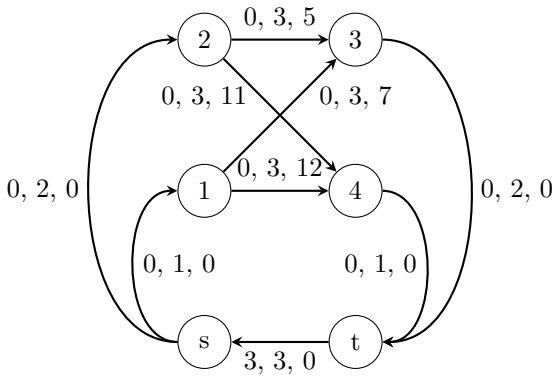
We will show how to transform any transshipment into circulation.
Let (D, b, w) be a transshipment problem and define two new vertices s and t.

- For each supply vertex x add the arc (s, x) to D with $l_x = 0, u_x = -b_x, w_x = 0$.
- Similarly, for each demand vertex x, add the arc (x, t) to D with $l_x = 0, u_x = b_x, w_x = 0$.
- Finally, add an arc (t, s) having $w_{ts} = 0, l_{ts} = u_{ts} = \sum\{b_x : \forall x, x \text{ is demand vertex}\}$.
- Each original arc is given a $l_x = 0, u_x = \sum\{b_x : \forall x, x \text{ is a demand vertex}\}$, w_x remains unchanged.

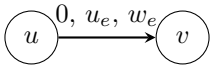
The following is a graph for transshipment problem.



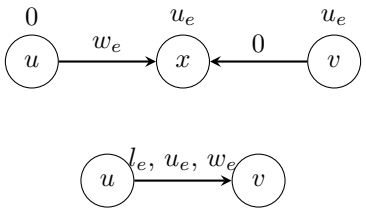
After the above procedures, it is now transformed into a circulation problem.



Then, we will show how to transform any circulation problem into transshipment problem.
If the lower bound of the arc is zero, i.e., $l_e = 0$, then for each such arc (u, v), introduce a vertex in between u and v, replace the arc $e = (u, v)$ by $e_1 = (u, x)$ and $e_2 = (x, v)$. Both arcs are uncapacitated. Let $w_1(u, x) = w(u, v)$ and $w_2(x, v) = 0$ be the new weights for the arcs. Let u_e be the demands of newly added vertex x and add u_e to the supplies of vertex v (in v the supplies is the summation from all arcs that go to v).

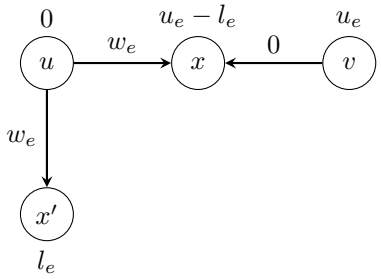


Will be transform into
If the lower bound of the arc is not zero, i.e., $l_e \neq 0$, then for each such arc (u, v), introduce two new vertices, one vertex is in between u and v, similar to the previous case, the difference is the demands of this new vertex is



$u_e - l_e$, the others stays the same. Then add the other vertex, this vertex, denoted as x' is added along with an arc between u and x', the weight of this arc is $w(u, v)$, the demands on this new vertex is l_e.

Will be transform into



Perform the above procedures to all the arcs in a circulation problem. In $O(E)$ (polynomially times) transformation, such problem can be transformed into transshipment problem.

8.4 Out-of-Kilter algorithm

This algorithm is a Primal-dual method and is applied to the minimum weight circulation problem.

For LP optimality conditions we need primal feasibility, dual feasibility and complementary slackness, i.e., KKT conditions. Primal and dual feasibility are obvious so we need to show complementary slackness through following theorem.

Theorem 8.3. *Let x be a feasible circulation flow for (D, l, u, w). And suppose there exists a real value vector $\{y_i : i \in V\}$ which we called **vertex-numbers**. For all edges $e \in A$*

$$y_{h(e)} - y_{t(e)} > w_e \text{ implies } x_e = u_e \tag{8.18}$$
$$y_{h(e)} - y_{t(e)} < w_e \text{ implies } x_e = l_e \tag{8.19}$$

Then x is optimal to the circulation problem.

Proof. For each $e \in A$ define

$$\gamma_e = \max\{y_{h(e)} - y_{t(e)} - w_e, 0\} \tag{8.20}$$
$$\mu_e = \max\{w_e - y_{h(e)} + y_{t(e)}, 0\} \tag{8.21}$$

Then

$$\gamma_e - \mu_e = y_{h(e)} - y_{t(e)} - w_e \tag{8.22}$$

Furthermore

$$\sum_{e\in A}(\mu_e l_e - \gamma_e u_e) \tag{8.23}$$

$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e) + \sum_{i\in V} y_i(\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e) \tag{8.24}$$

$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e + x_e(y_{h(e)} - y_{t(e)})) \tag{8.25}$$

$$= \sum_{e\in A}(\mu_e l_e - \gamma_e u_e + x_e(\gamma_e - \mu_e + w_e)) \tag{8.26}$$

$$= \sum_{e\in A}(\gamma_e(x_e - u_e) + \mu_e(l_e - x_e) + x_e w_e) \tag{8.27}$$

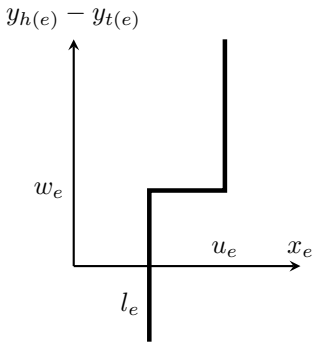
$$\leq \sum_{e\in A} x_e w_e \tag{8.28}$$

The last inequality will be satisfied as equality iff the first two hold. □

The following is the formulation of circulation problem

$$\begin{aligned}
&(\text{P}) \quad \min \quad && wx && (8.29)\\
&\qquad \text{s.t.} \quad && Nx = 0 \quad y && (8.30)\\
&\qquad && x \geq l \quad z^l && (8.31)\\
&\qquad && -x \leq -u \quad z^u && (8.32)\\
&(\text{D}) \quad \max \quad && lz^l - uz^u && (8.33)\\
&\qquad (\text{s.t.}) \quad && yN^{-1} + z^l - z^u \leq w && (8.34)\\
&\qquad && y \quad free && (8.35)\\
&\qquad && z^l, z^u \geq 0 && (8.36)\\
&\qquad (\text{CS}) \quad && y_{h(e)} - y_{t(e)} > w_e \Rightarrow x_e = u_e && (8.37)\\
&\qquad && y_{h(e)} - y_{t(e)} < w_e \Rightarrow x_e = l_e && (8.38)
\end{aligned}$$

There is an alternative way of circulation optimality for a circulation problem. We define a **kilter-diagram** as follows.
For every edge construct the following:

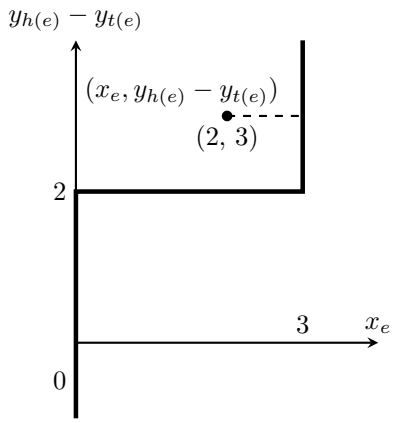


For each point $(x_e, y_{h(e)} - y_{t(e)})$ we define a **kilter-number** k_e, be the minimum positive distance change in x_e required to put in on the kilter line.

Example. For edge $e : w_e = 2, l_e = 0, u_e = 3$, assume $x_e = 2, y_{h(e)} - y_{t(e)} = 3$, then $k_e = 1$

Lemma 8.4. *If for every circulation x and vertex number y we have $\sum_{e\in A} k_e = 0$, then x is optimal.*

Proof. Since k_e is a nonnegative number, then the only way that $\sum_{e\in A} k_e = 0$ is $k_e = 0, \forall e \in A$, which means $\forall e \in A, l_e \leq x_e \leq u_e$. Furthermore, the complementary slackness are satisfied. □



General idea of algorithm follows. Suppose we are given a circulation x and vertex-numbers y (we do not require feasibility). Usually we pick $x = 0, y = 0$. If every edge is in kilter-line then we are optimal.
Otherwise there is at least one edge e^* that is out-of kilter. The algorithm consist of two phases, one called **flow-change** phase (horizontally), then other **number-change** phase (vertically).
In the flow-change phase, we want to find a new circulation for an out-of-kilter edge e^* say $\hat{e}$ such that we reduce the kilter number k_{e^*}, without increasing any other kilter number for other edges.
To do this, denote the edges of e^* to be s and t, where such that k_{e^*} will be decreased by increasing the flow from s to t on e^*.
If $e^* = (s, t)$ this will accomplished by increasing x_{e^*} and if $e = (t, s)$ it is accomplished by decreasing x_{e^*}.
To do this we look for an (s, t)-path p of the following edges.

- If e is forward in p, then increasing x_e does not increase k_e and
- If e is reversed in p, then decreasing x_e dose not increase k_e

In terms of kilter diagram, an arc satisfies "forward" if it is forward and in left side of kilter line, and it satisfies "reversed" if it is reverse and in right side of kilter line.
Suppose we can not find such a path. From s to t, let x be the vertices that can decrease by an augmenting path. Then either we can change the vertex numbers y so that $\sum_{e \in A} k_e$ does not increase but x does, or we can show that problem is infeasible.
INPUT a minimum circulation problem (D, l, u, w) a circulation x and vertex-numbers y
OUTPUT conclusion that (D, l, u, w) is infeasible or an minimum weighted flow.
Step 1: If every arc is in kilter ($k_e = 0, \forall e \in A$). Stop with x is optimal. Otherwise let e^* be an out-of-kilter arc. If increasing x_{e^*} decreases k_{e^*} set $s = h(e^*)$ and $t(e^*)$ otherwise set $s = t(e^*)$ and $t = h(e^*)$
Step 2: If there exists an (s, t) augmenting path p then goto Step 3, otherwise goto Step 4.
STEP 3: Set $y_e = y_{h(e)} - y_{t(e)}, e \in A$ Set $\Delta_1 = \min\{u_e - x_e : e \text{ is forward and } y_e \geq w_e\}$ Set $\Delta_2 = \min\{l_e - x_e : e \text{ is forward and } y_e < w_e\}$ Set $\Delta_3 = \min\{x_e - l_e : e \text{ is reverse and } y_e \leq w_e\}$ Set $\Delta_4 = \min\{x_e - u_e : e \text{ is reverse and } y_e > w_e\}$ $\Delta = \min\{\Delta_i, i = 1, 2, 3, 4\}$
Increase x_e by Δ on each forward arc in p, decrease x_e by Δ on each reverse arc in p.
If $e^* = (s, t)$ decrease x_{e^*} by Δ, otherwise increase x_{e^*} by Δ
If $k_{e^*} > 0$ goto Step 2. otherwise goto Step 1.
Step 4: Let X be the set of vertices reachable from s by augmenting paths, then $t \notin X$, if every arc e with $h(e) \in X$ has $x_e \leq l_e$ and every arc e with $t(e) \in X$ has $x_e \geq u_e$, and at least one of the above inequality is strict, then Stop with problem infeasible
Otherwise set $\delta_1 = \min\{w_e - y_e : t(e) \in X, y_e < w_e, x_e \leq u_e \neq l_e\}$ $\delta_2 = \min\{y_e - w_e : h(e) \in X, y_e > w_e, x_e \geq u_e \neq l_e\}$ $\delta = \min\{\delta_1, \delta_2\}$
Set $y_i = y_i + \delta$ for $i \notin X$
If $k_{e^*} > 0$, goto Step 2, otherwise goto Step 1.
Out-of-kilter takes $O(|E||V|K)$ where $K = \sum_{e \in A} k_e$. However, there is an algorithm called **scaling algorithm** that uses out-of-kilter as subroutine that runs in $O(R|E|^2|V|)$ where $R = \lceil \max\{\log_2 u_e : e \in A\} \rceil$

8.5 Complexity of Different Minimum Weighted Flow Algorithms

Let arc capacities between 1 and U, costs between $-C$ and C

Year	Discoverer	Method	Big O
1951	Dantzig	Network Simplex	$O(E^2V^2U)$
1960	Minty, Fulkerson	Out-of-Kilter	$O(EVU)$
1958	Jewell	Successive Shortest Path	$O(EVU)$
1962	Ford-Fulkerson	Primal Dual	$O(EV^2U)$
1967	Klein	Cycle Canceling	$O(E^2CU)$
1972	Edmonds-Karp, Dinitz	Capacity Scaling	$O(E^2 \log U)$
1973	Dinitz-Gabow	Improved Capacity Scaling	$O(EV \log U)$
1980	Rock, Bland-Jensen	Cost Scaling	$O(EV^2 \log C)$
1985	Tardos	ϵ-optimality	poly(E, V)
1988	Orlin	Enhanced Capacity Scaling	$O(E^2)$

Chapter 9

Matchings

9.1 Maximum Matching

Definition 9.1.1 (Matching)**.** Let $G = (V, E)$ be a graph, a **matching** is a subset of edges $M \subseteq E$ such that no two elements of M are adjacent. The two ends of an edge in M are said to be **matched under** M. A matching M saturates a vertex v, and v is said to be **M-saturated** or **M-covered**, if some edge of M is incident with v. Otherwise, v is **M-unsaturated** or **M-exposed**.

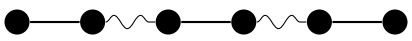
Definition 9.1.2 (Perfect matching, Maximum matching)**.** If every vertex of G is M-saturated, then the matching is said to be **perfect matching**. M is a **maximum matching** if G has no matching M' with $|M'| > |M|$. Every perfect matching is maximum. The maximum matching does not necessarily to be perfect. Perfect matching and maximum matching may not be unique.

Definition 9.1.3 (M-alternating)**.** An **M-alternating** path in G is a path whose edges are alternately in $E \setminus M$ and M.

Definition 9.1.4 (M-augmenting)**.** An **M-augmenting** path in G is an M-alternating path whose origin and terminus are M-unsaturated.

Lemma 9.1. *Every augmenting path P has property that let $M' = P\Delta M = (M \cup P) \setminus (M \cap P)$ then M' contains one more edge then M*

The following path is an M-augmenting path



The following path is $M' = P\Delta M(M \cup P) \setminus (M \cap P)$ and all the vertices are M-saturated.

Theorem 9.2 (Berge, 1957)**.** *A matching M in a graph G is maximum iff G has no M-augmenting path.*

Proof. ($\Rightarrow$) It is clear that if M is maximum, it has no augmenting paths since otherwise by problem claim we can increase by one.
($\Leftarrow$) Suppose M is not maximum and let M' be a bigger matching. Let $A = M\Delta M'$ now no vertex of G is incident to more than two members of A. For otherwise either two members of M or two members of M' would be adjacent. Contradict the definition of matching. It follows that every component of the edges incident subgraph $G[A]$ is either an even cycle with edge augmenting in $M\Delta M'$ or else A path with edges alternating between M and M'.
Since $|M'| \geq |M|$ then the even cycle cannot help because exchanging M and M' will have same cardinality.
The path case implies that p is alternating in M and since $|M'| > |M|$ the end arc exposed so that p is augmenting. □

Definition 9.1.5 (Vertex-cover)**.** The **vertex-cover** is a subset of vertices X such that every edge of G is incident to some member of X.

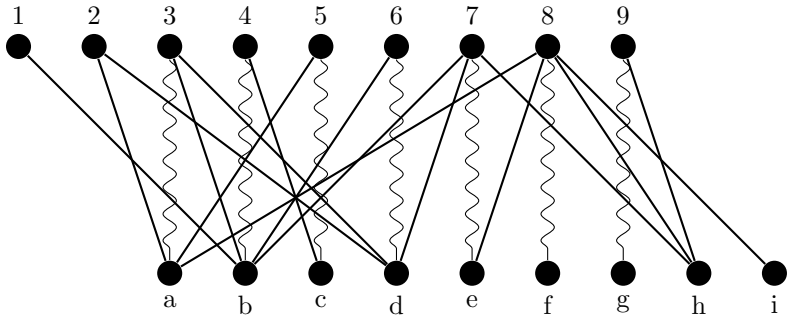
Lemma 9.3. *The cardinality of any matching is less than or equal to the cardinality of any vertex cover.*

Proof. Consider any matching. Any vertex cover must have nodes that at least incident to the edges in the matching. Since all the edges in the matching are disjointed, so for a single node can at most cover one edge in the matching. If the matching is not perfect, for the edges that not in the tree, they may or may not be possible to be covered by the nodes incident to the edges in the matching, with an easy triangle graph example, we can prove this lemma. □

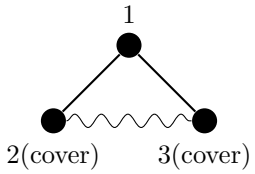
Theorem 9.4 (König Theorem)**.** *If G is bipartite, the cardinality of the maximum matching is equal to the cardinality of the minimum vertex cover.*

Proof. Let G be a bipartite graph, $G = (V, E)$ where $V = X \cup Y$ as X and Y are two disjointed sets of vertices. Let M be a maximum matching on G. For each edge in M, denoted by $e_i = a_i b_i$ where $e_i \in M$, $a_i \in A$ and $b_i \in B$ and $A = \{a_i : e_i \in M\} \subseteq X$, and $B = \{b_i : e_i \in M\} \subseteq Y$. Therefore, we can partition X by A and $U = X \setminus A$, partition Y by B and $W = Y \setminus B$.
We can further partition the matching M into M_1 and M_2. For all the edges in M_1 can be included into an M-alternating path starts from a vertex in U (which includes the edges directly linked to vertices in U), and $M_2 = M \setminus M_1$. For edges in M_1, we take the ends of edges in B in the vertex cover, denoted by B_1, take the ends of edges in A as a subset denoted by $A_1 \subseteq A$. For the edges in M_2, we take the ends of edges in A in the vertex cover, denoted by A_2, and the ends of edges in B as a subset denoted by $B_2 \subseteq B$.
We claim that all the vertices in U can only be connected to vertices in B_1 and vertices in W can only be connected to vertices in A_2.
$U \subset X$ connects to vertices in B_1 by definition. If vertices in $W \subset Y$ is connected to vertices in A_1, then we will have M-augmenting path which is contradicted to the assumption that M is maximum matching. □

The following is an example. Where the edge in the matching that accessible from members of $U = \{1, 2\}$ in an M-alternating path is edge $3a, 4b, 5c, 6d$.



In which $U = \{1, 2\}$, $M_1 = \{3a, 4b, 5c, 6d\}$. $U = \{1, 2, \}, A_1 = \{3, 4, 5, 6\}, A_2 = \{7, 8, 9\}, W = \{h, i\}, B_1 = \{a, b, c, d\}, B_2 = \{e, f, g\}$. The vertex cover is $\{a, b, c, d, 7, 8, 9\}$.
The above theorem does not apply to non-bipartite graph. The following is an example



The maximum matching has one edge, where the minimum cover has two vertices.

9.2 Maximum Matching Algorithm

Definition 9.2.1 (M-alternating tree)**.** An **M-alternating tree** T is a rooted tree satisfied the following condition:

- The root r is M-unsaturated
- The unique path from r to any vertex T is M-alternating
- Every vertex in T, except r is incident to a matching edge of T

A vertex x of T is called **inner** if (r, s)-path in T has an odd number of edges. Otherwise x is called **outer**.

Lemma 9.5. *Let M be a matching in G and let T be an M-alternating tree with root r, then the following conclusion hold*

- *If $v \neq r$ is an outer vertex and p is the unique (r, v)-path in T then the edge of p incident to v is in M*
- *The number of inner vertices in T equals the number of matching edges in T.*

Definition 9.2.2 (Alternating forest)**.** An **alternating forest** is a forest of G where every components is an alternating tree. An alternating forest (F, e) is an alternating forest to be then with an edge $e = M, V$ where M and V are outer vertices contained in two distinct components of F.

Example (Hungarian forest)**.** A **Hungarian forest** F is an alternating forest containing all exposed vertices of G and such that the outer vertices of F are adjacent in G only to inner vertices of F

Definition 9.2.3 (Augmenting forest)**.** An **augmenting forest** (F, e) where F is an augmenting forest and $e = uv$ connects two outer vertices in distinct components of F.

The plan is to grow an alternating forest that eventually become augmenting or Hungarian. Augmenting forest will increase the cardinality of the match, Hungarian implies that you have found optimal maximum cardinality matching.

Theorem 9.6. *Let (F, e) be an augmenting forest. And let T_1 and T_2 be the two components of F containing an end of e. Let p_i be the unique path in T_i from the root to the end of e, then $p_1 e p_2^{-1}$ is an augmenting path.*

Theorem 9.7. *If F is a Hungarian forest for some matching M then M is a maximum match.*

The above theorems suggest a method for computing maximum matching. Let M be a matching of G and let F be an alternating forest in G made up of all M-exposing vertices. If F happens to be Hungarian, stop with the max matching M. If (F, e) for some e is augmenting then we increase our matching by 1 and start process.
Suppose F is neither Hungarian nor augmenting, by definition, there must be an edge e incident to an outer vertex of F to no inner vertex of F. But e cannot be incident to two outer vertices of F in distinct components, since (F, e) is not augmenting. Hence there are only two cases:
Case 1: $e = uv$ where u is outer in F and v is not outer in F. The only way its possible if v is covered. Augmenting F to M will increase the matching.
Case 2: $e = uv$ where u and v are outer vertices in F. Let r be the root of the component of F containing u and v, let p_u and p_v be (r, u)-path and (r, v)-path in F. Let b be the last vertex these two paths have in common and let p be the (u, v)-path in F, let p'_u be $(b - u)$-path and p'_v be the (b, v)-path respectively. Let n be the length of p_u, let m be the length of p_v, let k be the length of (r, b)-path. Let c be the cycle $(p'_u e {p'_v}^{-1})$. Number of vertices in c is $n + m - 2k + 1$, n, m are even, so c is always an odd length cycle. If G has no odd cycles, we call those graph bipartite. To this case can not happen in bipartite graph so algorithm without case 2 will solve bipartite matching. If a graph has no odd cycles, i.e., bipartite, then we have an algorithm using augmenting forest and Hungarian forest and case 1.
We now have to deal with odd cycles. The idea is to "shrink" add cycles to a super node
Let $S \subseteq E(G)$, denote by $G : S$ the subgraph with edge set S

$$G : S = G \setminus (E(G) - S) \tag{9.1}$$

The contraction of S to be the $G \setminus S$ with $E(G/S) = E(G) - S$. $V(G/S)$ to be the components of $G : S$ and if $e \in E(G/S)$ then the ends of e in G/S are in components of $G : S$ containing both ends in G
Let network to general case 2. b is outer vertex. Let B be the set of edges of cycle C, we call B a **blossom**. We propose to replace M by $M - B$, G by G/B and F by F/B

9.3 Edmonds's Blossom Algorithm

$O(|V|^4)$ - Non-bipartite matching is one of very few problems in P, for which LP relaxation will not provide optimal solution.

9.4 Hall's Marriage Theorem

9.5 Transversal Theory

9.6 Menger's Theorem

9.7 The Hungarian Algorithm

Chapter 10

Colorings