Notes for Operations Research & More

Lan Peng, PhD Student

Department of Industrial and Systems Engineering
University at Buffalo, SUNY
lanpeng@buffalo.edu

October 27, 2019

October 27, 2019

Contents

Part I

Graph and Network Theory

Chapter 1

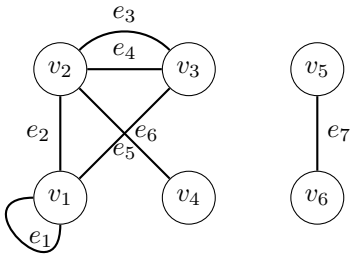
Graphs and Subgraphs

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

Figure 1.1: An example of graph



can be represented as

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

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 Graph Isomorphism

1.3 The Adjacency and Incidence Matrices

1.4 Subgraph

Definition 1.4.1 (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 1.4.2 (spanning)**.** A subgraph H on G is **spanning** if $V(H) = V(G)$

Definition 1.4.3 (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.

1.5 Degree

Definition 1.5.1 (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* $G = (V, E)$

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

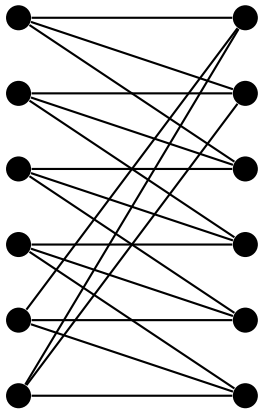
Proof. $\forall$ edge $e = \mu v$ with $\mu \neq v$, e is and counted once for μ and once for v, a total of two altogether. If $e = \mu\mu$, a loop, then it is counted twice for μ $\square$

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.

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

□

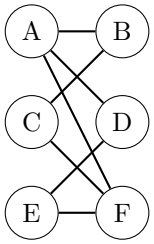
1.6 Special Graphs

Definition 1.6.1 (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 1.6.2 (Cycle)**.** A **cycle** graph $C_n (n \geq 3)$ consists of n vertices $v_1, ...v_n$ and n edges $\{v_1, v_2\}, \{v_2, v_3\}, ...\{v_{n-1}, v_n\}$

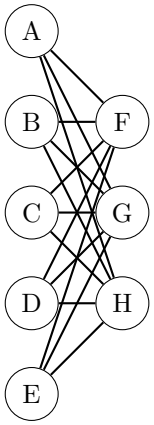
Definition 1.6.3 (Wheel)**.** A **wheel** graph $W_n (n \geq 3)$ is a simple graph obtains by adding one vertex to the cycle graph C_n, and connecting this new vertex to all vertices of C_n

Definition 1.6.4 (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



Definition 1.6.5 (complete bipartite)**.** 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. *(König Theorem) A graph G is bipartite iff every cycle is even.*

Proof. Hereby we prove the ⇒ and ⇐

- (⇒) 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.

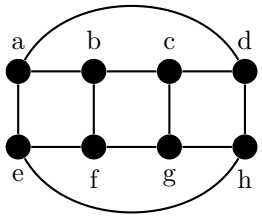
- (⇐) Prove by contradiction. A graph can be connected or not connected.

– 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 subgraph 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.

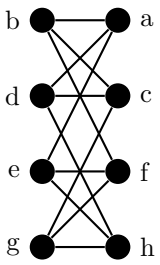


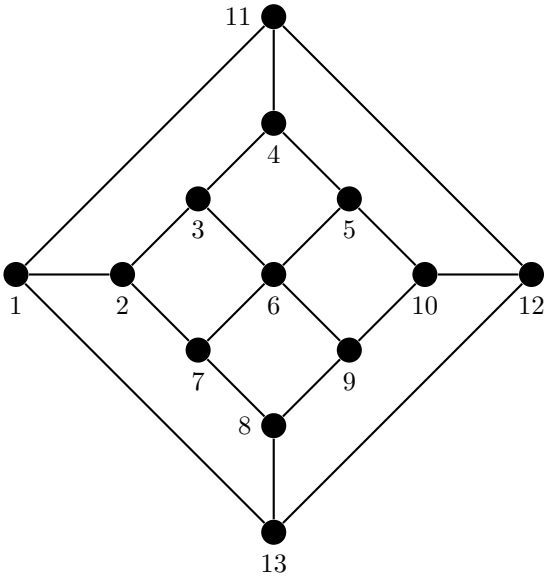
We can rearrange the graph to be more clear as following

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 $c = v_1v_11v_4v_3v_2$ have odd number of vertices.





1.7 Directed Graph

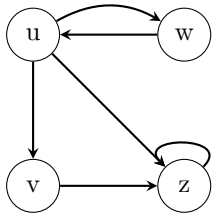
Definition 1.7.1. A graph $G = (V, E)$ is called directed if for each edge $e \in E$, there is a **head** $h(e) \in V$ and a **tail** $t(e) \in V$ and the edges of e are precisely $h(e)$ and $t(e)$, denoted $e = (t(e), h(e))$

Definition 1.7.2. We call directed graphs **digraphs**, we call edges in a digraph are called **arcs**, and vertices in a digraph **nodes**

Definition 1.7.3. Similar as in the undirected case we have walks, traces, paths and cycles in digraphs.

Definition 1.7.4. 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 1.7.5. A digraph is strongly connected if every pair of vertices are strongly connected.



Definition 1.7.6. A digraph is **strict** if it has no loops and whenever e and f are parallel, $h(e) = t(f)$

Definition 1.7.7. 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

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

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

1.8 Sperner's Lemma

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.

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: $K_1, K_2, ...K_m$

```
Initial, K ← ∅, j = 1
Set temporary storage units, v_o ← v_11, v_t ← ∅
for i = 1, 2, ..., n do
    if e_1i ∈ C_2 then
        if v_o ≠ v_1i then
            v_t ← v_1i
            concate (v_o, v_t)-path ⊂ C_1 and (v_o, v_t)-path
            ⊂ C_2 to create a new K_j
            Append K with K_j, K ← K ∪ K_j
            Reset temporary storage unit. v_o ← v_1(i+1)
            (or v_11 if i = n), v_t ← ∅
        else
            v_o ← v_1(i+1) (or v_11 if i = n)
        end if
    end if
end for
```

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 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.)

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: $p \leftarrow p \cup e$
9: **end if**
10: **end while**

The above algorithm is guaranteed to have an end because a tree is acyclic by definition

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)

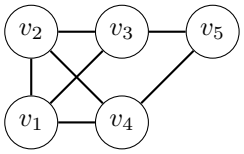
3. Do the leaf removal process recursively to $T^{'}$ if Situation 2 happens until Situation 1 happens.

□

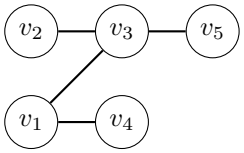
2.4 Spanning tree

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

```
Let T be the spanning subgraph of G with V(T) = V(G) and E(T) = ∅
i ← 1
while i ≤ |E| do
    if T + e_i is acyclic then
        T ← T + e_i
        i ← i + 1
    end if
end while
```

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

Chapter 3

Connectivity

3.1 Connectivity

3.2 Blocks

Chapter 4

Euler Tours and Hamilton Cycles

4.1 Euler Tours

4.2 Hamilton Cycles

Chapter 5

Planarity

Chapter 6

Minimum Spanning Tree Problem

6.1 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 6.1.1 (Cut vertex)**.** A vertex v of a connected graph G is a **cut vertex** if $G \setminus v$ is not connected.

Definition 6.1.2 (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 6.1. *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 6.1.3 (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.

6.2 Kroskal's Algorithm

Algorithm 4 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.
  end while

- Kroskal start with a forest that contains all vertices, Prim start with a tree that only contain one vertex.

Algorithm 5 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.
  end while

- 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.

Definition 6.2.1 (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 6.2.2. We also define $\gamma(A) = \{e \in E : \text{both ends of } e \text{ are in } A\}$

Theorem 6.2. *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 6.2.3. Let us call a subset $A \in E$ **extendsible** to a minimum spanning tree problem if A is contained in the edge set of some MST of G

Theorem 6.3. *Suppose $B \subseteq E$, that B is extendsible 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.*

6.3 Prim's Algorithm

6.4 Comparison between Kroskal's and Prim's Algorithm

6.5 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

$$X_e = \begin{cases} 1, \text{if edge } e \text{ is in the optimal solution} \\ 0, \text{otherwise} \end{cases} \tag{6.1}$$

The formulation is as following

$$\min \quad \sum_{e \in E} c_e x_e \tag{6.2}$$

$$\text{s.t.} \quad \sum_{e \in E} x_e = |V| - 1 \tag{6.3}$$

$$x_e \geq 0 \tag{6.4}$$

$$e \in E \tag{6.5}$$

$$\sum_{e \in E(S)} x_e = |S| - 1, \forall S \subseteq V, S \neq \emptyset \tag{6.6}$$

$$\tag{6.7}$$

Chapter 7

Shortest-Path Problem

7.1 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 7.1.1 (feasible potential)**.** We call $y = (y_v : v \in V)$ a **feasible potential** if it satisfies

$$y_v + c_{vw} \geq y_w \quad \forall (v, w) \in E \tag{7.1}$$

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

$$C(P) = \sum_{i=1}^{k} C_{e_i} \geq \sum_{i=1}^{k} (y_{v_i} - y_{v_{i-1}}) = y_{v_k} - y_{v_0} \tag{7.2}$$

□

7.2 Breadth-First Search Algorithm

7.3 Ford's "algorithm"

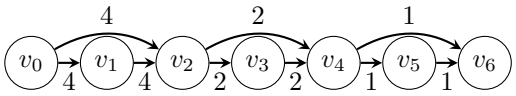
Define a predecessor function $P(w), \forall w \in V$ and set $P(w)$ to v whenever y_w is set to $y_v + c_{vw}$

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.

Algorithm 6 Ford's Algorithm

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)
    end if
  end while



7.4 Ford-Bellman Algorithm

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.

Algorithm 7 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)
      end if
    end for
  end for
  for $\forall e = (v, w) \in E$ **do**
    if $y_v + c_{vw} < y_w$ (incorrect) **then**
      Return error, negative cycle
    end if
  end for

7.5 SPFA Algorithm

7.6 Dijkstra Algorithm

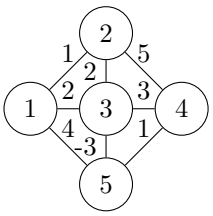
Algorithm 8 Dijkstra Algorithm

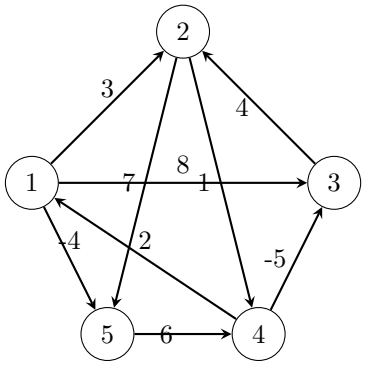
Ensure: Shortest Paths from r to all other nodes in V
Require: A digraph with arc costs,starting node r
  Initialize y and p
  $S \leftarrow V$
  while $S \neq \emptyset$ **do**
    Choose $v \in S$ with minimum y_v
    $S \leftarrow S \setminus v$
    for $\forall w, (v, w) \in E$ **do**
      if $y_v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

7.7 A* Algorithm

7.8 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 $O(n^3)$





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

$$d_{ij}^0 = \begin{cases} c_{ij} & \text{distance from } i \text{ to } j \text{ if } (i,j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \notin E \end{cases} \tag{7.3}$$

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

$$D^0 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{7.4}$$

$$\Pi^0 = \begin{bmatrix} & 1 & 1 & & 1 \\ & & & 2 & 2 \\ & 3 & & & \\ 4 & & 4 & & \\ & & & 5 & \end{bmatrix} \tag{7.5}$$

$$D^1 = \begin{bmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \mathbf{5} & -5 & 0 & \mathbf{-2} \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{7.6}$$

$$\Pi^1 = \begin{bmatrix} & 1 & 1 & & 1 \\ & & & 2 & 2 \\ & 3 & & & \\ 4 & \mathbf{1} & 4 & & \mathbf{1} \\ & & & 5 & \end{bmatrix} \tag{7.7}$$

$$D^2 = \begin{bmatrix} 0 & 3 & 8 & \mathbf{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \mathbf{5} & \mathbf{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{7.8}$$

$$\Pi^2 = \begin{bmatrix} & 1 & 1 & \mathbf{2} & 1 \\ & & & 2 & 2 \\ & 3 & & \mathbf{2} & \mathbf{2} \\ 4 & 1 & 4 & & 1 \\ & & & 5 & \end{bmatrix} \tag{7.9}$$

$$D^3 = \begin{bmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & \mathbf{-1} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{bmatrix} \tag{7.10}$$

$$\Pi^3 = \begin{bmatrix} & 1 & 1 & 2 & 1 \\ & & & 2 & 2 \\ & 3 & & 2 & 2 \\ 4 & \mathbf{3} & 4 & & 1 \\ & & & 5 & \end{bmatrix} \tag{7.11}$$

$$D^4 = \begin{bmatrix} 0 & 3 & \mathbf{-1} & 4 & -4 \\ \mathbf{3} & 0 & \mathbf{-4} & 1 & \mathbf{-1} \\ \mathbf{7} & 4 & 0 & 5 & \mathbf{3} \\ 2 & -1 & -5 & 0 & -2 \\ \mathbf{8} & \mathbf{5} & \mathbf{1} & 6 & 0 \end{bmatrix} \tag{7.12}$$

$$\Pi^4 = \begin{bmatrix} & 1 & \mathbf{4} & 2 & 1 \\ \mathbf{4} & & \mathbf{4} & 2 & \mathbf{1} \\ \mathbf{4} & 3 & & 2 & \mathbf{1} \\ 4 & 3 & 4 & & 1 \\ \mathbf{4} & \mathbf{3} & \mathbf{4} & 5 & \end{bmatrix} \tag{7.13}$$

$$D^5 = \begin{bmatrix} 0 & \mathbf{1} & \mathbf{-3} & \mathbf{2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix} \tag{7.14}$$

$$\Pi^5 = \begin{bmatrix} & \mathbf{3} & 4 & \mathbf{5} & 1 \\ 4 & & 4 & 2 & 1 \\ 4 & 3 & & 2 & 1 \\ 4 & 3 & 4 & & 1 \\ 4 & 3 & 4 & 5 & \end{bmatrix} \tag{7.15}$$

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.

7.9 Johnson's Algorithm

Chapter 8

Maximum Flow Problem

8.1 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.

$$\max \quad v \tag{8.1}$$

$$\text{s.t.} \quad \sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = \begin{cases} -v, & \text{if } i = s \\ v, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \tag{8.2}$$

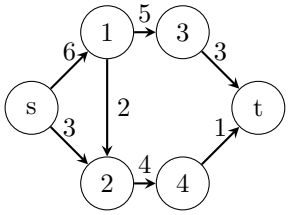
$$0 \le x_e \le u_e, \quad \forall e \in A \tag{8.3}$$

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.

8.2 The main theorem

Theorem 8.1. *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 $u(S, \bar{S})$ is $\sum\{u_e : e \in \delta^-(S)\}$ where $\delta^-(S) = \{e \in A : t(e) \in S \text{ and } h(e) \in \bar{S}\}$*

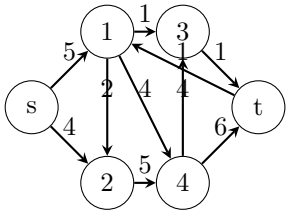
Example. For the following graph:



Let $S = \{1, 2, 3, s\}$, $\bar{S} = \{4, t\}$
then $\delta^-(S) = \{(2, 4), (3, t)\} \Rightarrow u(S, \bar{S}) = 7$

Definition 8.2.1. If $(S, \bar{S})$ has minimum capacity of all (s, t)-cuts, then it is called **minimum cut**.

Definition 8.2.2. Let $\delta^+(S) = \delta^-(V - S)$



Example. Let $S = \{s, 1, 2, 3\}$, $\bar{S} = \{4, t\}$, $u(S, \bar{S}) = u_{14} + u_{24} + u3t = 10$, $\delta^-(S) = \{(1, 4), (2, 4), (3, t)\}$, $\delta^+ S = \{(t, 1)\}$

Lemma 8.2. *If x is a (s, t) flow of value v and $(S, \bar{S})$ is a (s, t)-cut, then*

$$v = \sum_{e \in \delta^-(S)} x_e - \sum_{e \in \delta^+(S)} x_e \tag{8.4}$$

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

$$\sum_{i \in S} \Big(\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e \Big) = -v \tag{8.5}$$

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

$$\sum_{e \in \delta^+(S)} x_e - \sum_{e \in \delta^-(S)} x_e = -v \tag{8.6}$$

□

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

Corollary 8.2.1. *If x is a feasible flow of value v, and $(S, \bar{S})$ is an (s, t)-cut, then*

$$v \le u(S, \bar{S}) \quad \textit{(Weak duality)} \tag{8.7}$$

Definition 8.2.3. Define an arc e to be **saturated** if $x_e = u_e$, and to be **flowless** if $x_e = 0$

Corollary 8.2.2. *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

$$\sum_{e \in \delta^-(S)} x_e = \sum_{e \in \delta^-(S)} u_e \tag{8.8}$$

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

$$\sum_{e \in \delta^+(S)} x_e = 0 \tag{8.9}$$

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

Definition 8.2.4. 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 8.3. *A feasible flow x in a digraph D is maximum iff D has 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):

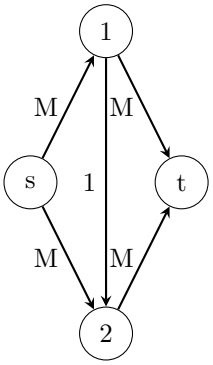
$$D_1 = \min\{u_e - x_e : e \text{ forward in } P\} \tag{8.10}$$
$$D_2 = \min\{x_e : e \text{ backward in } P\} \tag{8.11}$$
$$D = \min\{D_1, D_2\} \tag{8.12}$$

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

$$\hat{x_e} = \begin{cases} x_e + D & \text{If } e \text{ is forward in } P \\ x_e - D & \text{If } e \text{ is backward in } P \\ x_e & otherwise \end{cases} \tag{8.13}$$

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 8.4. *(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*



8.3 Maximum flow algorithm

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 FIXME

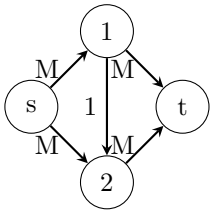
Algorithm 9 Labeling algorithm

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
  end while
  x is the maximum.

Labeling algorithm can be exponential, the following is an example

8.4 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 $V_{i+1} \leftarrow R \setminus \cup_{0 \le j \le i} V_j$, $l \leftarrow i + 1, l(x) = i$, goto Step 2. If $V_{i+1} = \emptyset$, set $l(x) = i$ and Stop.

Example. For the following graph

$$\text{Second iteration} \tag{8.14}$$
$$V_0 = \{s\}, i = 0, l(x) = 0 \tag{8.15}$$
$$R = \{1, 2\} \tag{8.16}$$
$$V_1 \leftarrow \{1, 2\}, i = 1, l(x) = 1 \tag{8.17}$$
$$R = \{3, 4, 5\} \tag{8.18}$$
$$V_2 \leftarrow \{3, 4\}, i = 2, l(x) = 2 \tag{8.19}$$
$$R = \{1, 5, 6, 3\} \tag{8.20}$$
$$V_3 \leftarrow \{5, 6\}, i = 3, l(x) = 3 \tag{8.21}$$
$$R = \{4, t\} \tag{8.22}$$
$$V_4 = \{t\} \tag{8.23}$$
$$A_1 = \{(s, 1), (s, 2)\} \tag{8.24}$$
$$A_2 = \{(1, 3), (2, 4)\} \tag{8.25}$$
$$A_3 = \{(3, 5), (4, 6)\} \tag{8.26}$$
$$A_4 = \{(5, t), (6, t)\} \tag{8.27}$$

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 $i, 1 \le i \le l$, 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

$$x'_e = x_e + \hat{x_e}, \quad \text{If } t(e) \in V_{i-1}, h(e) \in V_i \tag{8.28}$$
$$x'_e = x_e - \hat{x_e}, \quad \text{If } h(e) \in V_{i-1}, t(e) \in V_i \tag{8.29}$$

8.5 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 $i_{j-1} = T(i_j), 1 \le j \le k$. Set $D = \min\{\hat{u_{e_j}} - \hat{x_{e_j}} : i \le j \le k\}$, set $\hat{x_{e_j}} \leftarrow \hat{x_{e_j}} + D, i \le j \le k$. Delete from H all saturated arcs on this path, set $i \leftarrow 1$ and goto Step 2.

Theorem 8.5. *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 9

Minimum Weight Flow Problem

9.1 Transshipment Problem

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

$$\begin{aligned} \min \quad & wx && (9.1)\\ \text{s.t.} \quad & Nx = b && (9.2)\\ & x \geq 0 && (9.3) \end{aligned}$$

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

$$\sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = b_i \tag{9.4}$$

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.

Chapter 10

Matchings

Chapter 11

Colorings

Chapter 12

Independent Sets and Cliques

Chapter 13

Matroids