Lecture 5 - Graph Algorithms

Lan Peng, Ph.D.

School of Management, Shanghai University, Shanghai, China

"You can't visit all seven bridges in Königsberg without repetition."

1 Preliminaries

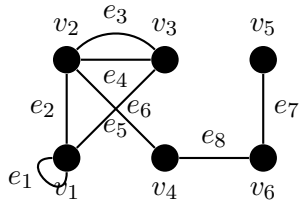
1.1 Graphs and Subgraphs

Graphs

Definition 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

$$V = V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$
$$E = E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$$
$$e_1 = v_1v_2, \quad e_2 = v_2v_4, \quad \dots$$



Definition 1.2 (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.

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	$O(n)$	$O(d_v)$

Subgraphs

Definition 1.3 (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 (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'.

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

Definition 1.5 (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 $G = (V, E)$*

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

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. □

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

Proof.

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

□

Special graphs

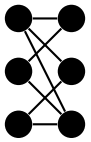
Definition 1.6 (Complete Graph)**.** A **complete** graph $K_n (n \geq 1)$ is a simple graph with n vertices and with exactly one edge between each pair of distinct vertices.

Definition 1.7 (Cycle Graph)**.** 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 1.8 (Wheel Graph)**.** 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

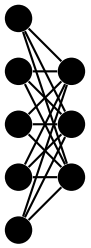
Definition 1.9 (Bipartite Graph)**.** A simple graph is said to be **bipartite** if the vertex set can be expressed as the union of two disjoint non-empty subsets V_1 and V_2 such that every edges has one end in V_1 and another end in V_2

Example. Here is an example for bipartite graph



Definition 1.10 (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 vertex 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.*

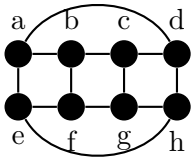
Proof. ($\Rightarrow$) If the graph G is bipartite, by definition, the vertices of graph can be partition into two groups, that within the group there is no connection between vertices. Therefore, for each cycle, the odd index of vertices and even index of vertices has to be choose alternatively from each groups. Therefore the cycle has to be even.

($\Leftarrow$) Prove by contradiction. A graph can be connected or not connected.

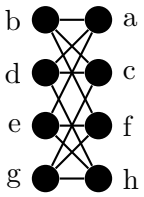
If G is connected and has at least two vertices, for an arbitrary vertex $v \in V(G)$, we can calculate the minimum number of edges between the other vertices v' and v (i.e., length, denoted by $l(v', v)$), for all the vertices that has odd length to v, assign them to set V_1, for the rest of vertices (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.

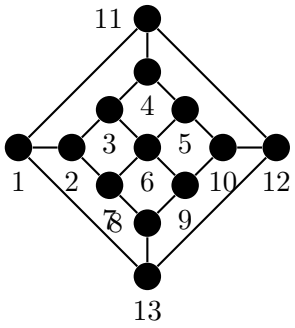


We can rearrange the graph to be more clear as following



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 $c = v_1v_{11}v_4v_3v_2$ have odd number of vertices.

1.2 Walk, Path and Cycle

Walk, trail, path

Definition 1.11 (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.

$$w = v_2e_4v_3e_4v_2e_5v_3$$

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

Definition 1.12 (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.

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

(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 concatenating w and w'. The new walk's origin is the same as terminal.

Definition 1.13 (trail)**.** A **trail** is a walk with no repeating edges. e.g., $v_3e_4v_2e_5v_3$

Definition 1.14 (path)**.** A **path** is a trail with no repeating vertices. e.g., $v_3e_4v_2$

Paths $\subseteq$ Trails $\subseteq$ Walks

Cycle

Definition 1.15 (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 1.16 (even cycle, odd cycle)**.** A cycle is **even** if it has a even number of edges otherwise it is **odd**.

Problem 1.1. 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

$$C_1 = v_{11}e_{11}v_{12}e_{12}v_{13}e_{13}...v_{1n}e_{1n}v_{11}$$
$$C_2 = v_{21}e_{21}v_{22}e_{22}v_{23}e_{23}...v_{2k}e_{2k}v_{21}$$

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: $K_1, K_2, ...K_m$
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: concatenate (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 $i = n$)

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, $K_i \cap K_j = \emptyset, \forall i \neq j$. □

1.3 Some warm-up algorithms

Component

Definition 1.17 (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 1.18 (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.

The following algorithm finds components in a given graph $G = (V, E)$.

Algorithm 2 Find components

```
Let Q be the set of components
for i ∈ V do
    Q ← {i}
for e = (i, j) ∈ E do
    for p ∈ Q do
        for q ∈ Q do
            if i ∈ p and j ∈ q then
                Q ← Q \ p \ q ∪ (p ∪ Q)
return Q
```

We will revisit this procedure when we discuss the subtour elimination constraints for TSP. Such procedure can be improved by introducing disjoint-set forest.

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 3 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 4 DFS(w, v)

```
1: Label v as "visited"
2: if number of v 's neighbor is 1 then
3:     return null
4: else
5:     for all neighbor u in linked-list of v excepts w do
6:         if u is labeled as "visited" then
7:             return "Cycle is found"
8:         else
9:             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 $O(n + \sum_{v \in N} d_v) = O(n + m)$

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 convenient 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 5 Test Bipartiteness

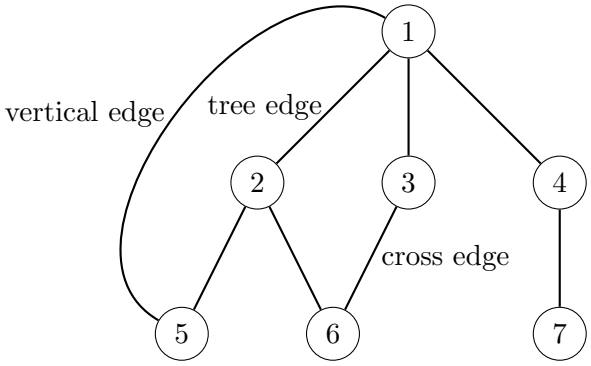
```
1: Initialize, head ← 1, tail ← 1, queue[1] ← s, mark all vertices as "unvisited"
2: Mark s as "visited"
3: color[s] ← 0
4: while head ≥ tail do
5:     v ← queue[tail], tail ← tail + 1
6:     for ∀u ∈ d_v do
7:         if u is "unvisited" then
8:             head ← head + 1, queue[head] = u
9:  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

Bridge

Definition 1.19 (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 1.20 (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 6 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)
```

Algorithm 7 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)
```

2 Matching

2.1 Matching

Definition 2.1 (matching, M-saturated, M-unsaturated)**.** 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**, if some edge of M is incident with v. Otherwise, v is **M-unsaturated**.

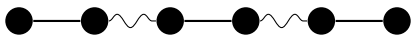
Definition 2.2 (perfect matching, maximum matching)**.** If every vertex of G is M-saturated, then the matching is said to be **perfect matching**, for perfect matching, we have $|M| = \frac{|V(G)|}{2}$. M is a **maximum matching** if G has no matching M' with $|M'| > |M|$. Every perfect matching is maximum. The maximum matching does not necessarily to be perfect. Perfect matching and maximum matching may not be unique.

Definition 2.3 (M-alternating)**.** An **M-alternating** path in G is a path whose edges are alternately in $E \setminus M$ and M.

Definition 2.4 (M-augmenting)**.** An **M-augmenting** path in G is an M-alternating path whose origin and terminus are M-unsaturated.

Lemma 2.1. *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 2.2 (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 2.5 (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 2.3. *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 2.4 (König Theorem)**.** *If G is bipartite, the cardinality of the maximum matching is equal to the cardinality of the minimum vertex cover.*

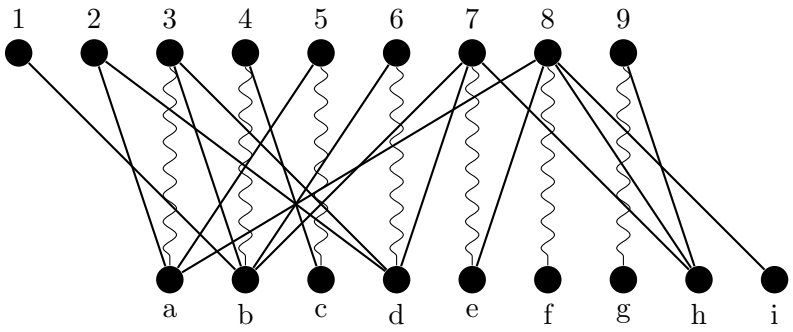
Proof. Let G be a bipartite graph, $G = (V, E)$ where $V = X \cup Y$ as X and Y are two disjointed sets of vertices. Let M be a maximum matching on G. For each edge in M, denoted by $e_i = a_i b_i$ where $e_i \in M$, $a_i \in A$ and $b_i \in B$ and $A = \{a_i : e_i \in M\} \subseteq X$, and $B = \{b_i : e_i \in M\} \subseteq Y$. 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.

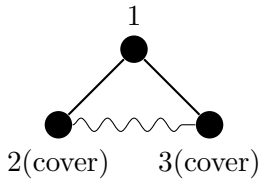
$U \subset X$ connects to vertices in B_1 by definition. If vertices in $W \subset Y$ is connected to vertices in A_1, then we will have M-augmenting path which is contradicted to the assumption that M is maximum matching. □

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.

2.2 Blossom Algorithm

First, for the bipartite graphs, there is no odd cycles in the graph, the maximum matching can be derived by repeatedly extend the existing augmenting paths. For the non-bipartite algorithm, we use the Blossom algorithm to find maximum matching.

Blossom A blossom is a set of nodes and edges starting at an unsaturated vertex, with a stem of even number of edges, and link to an odd cycle of edges.

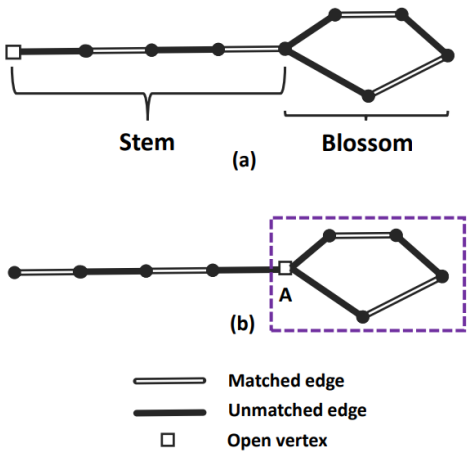


Figure 1: An example of blossom

Pseudo codes

Algorithm 8 Blossom algorithm

```
Let F be the set of unsaturated nodes
while F ≠ ∅ do
    Let r ∈ F be an unsaturated node
    queue.push(r)
    T ← ∅
    T.add(r)
    while queue ≠ ∅ do
        v ← queue.pop()
        for All neighbor w of v do
            if w ∉ T and w is saturated then
                T.add(w)
                T.add(mate(w)), mate(w) is another neighbor of w
                queue.push(mate(w))
            else if w ∈ T and odd cycle detected then
                Contract cycle
            else if w ∈ F then
                Expand all contract cycles
                Reconstruct augmenting path
                Augment path by inverting edges
```

3 Maximum Flow Problem

3.1 Maximum Flow Problem

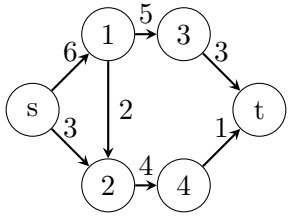
Let $D = (V, A)$ be a strict digraph with distinguished vertices s and t. We call s the source and t the sink, let $u = \{u_e : e \in A\}$ be a nonnegative integer-valued capacity function defined on the arcs of D. The maximum flow problem on (D, s, t, u) is the following Linear program.

$$\begin{aligned} \max \quad & v \\ \text{s.t.} \quad & \sum_{h(e)=i} x_e - \sum_{t(e)=i} x_e = \begin{cases} -v, & \text{if } i = s \\ v, & \text{if } i = t \\ 0, & \text{otherwise} \end{cases} \\ & 0 \le x_e \le u_e, \quad \forall e \in A \end{aligned}$$

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.

Theorem 3.1. *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 $\delta^-(S) = \{e \in A : t(e) \in S \text{ and } h(e) \in \bar{S}\}$*

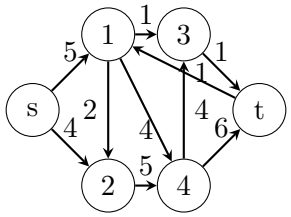
Example. For the following graph:



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

Definition 3.1. If $(S, \bar{S})$ has minimum capacity of all (s, t)-cuts, then it is called **minimum cut**.

Definition 3.2. 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 3.2. *If x is a (s, t) flow of value v and $(S, \bar{S})$ is a (s, t)-cut, then*

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

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

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

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

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

□

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

Corollary 3.2.1. *If x is a feasible flow of value v, and $(S, \bar{S})$ is an (s, t)-cut, then*

$$v \leq u(S, \bar{S}) \quad \textit{(Weak duality)}$$

Definition 3.3. Define an arc e to be **saturated** if $x_e = u_e$, and to be **flowless** if $x_e = 0$

Corollary 3.2.2. *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

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

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

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

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

3.2 Prime and Dual of Maximum Network Flow Problem

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

$$\begin{aligned}
\max \quad & f = \begin{bmatrix}0 & 0 & \cdots & 0 & 1\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ f\end{bmatrix}\\
\text{s.t.} \quad & \begin{bmatrix}\mathbf{A} & \mathbf{F}\end{bmatrix}\begin{bmatrix}\mathbf{x}\\ f\end{bmatrix} = \mathbf{0}\\
& \mathbf{I}\mathbf{x} \le \mathbf{u}\\
& \begin{bmatrix}\mathbf{x}\\ f\end{bmatrix} \ge 0
\end{aligned}$$

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.

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

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

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

Then, we take the dual of LP

$$\begin{aligned}
\min \quad & \mathbf{u}\mathbf{w}_\mathbf{E}\\
\text{s.t.} \quad & \begin{bmatrix}\mathbf{w}_\mathbf{V} & \mathbf{w}_\mathbf{E}\end{bmatrix}\begin{bmatrix}\mathbf{A}\\ \mathbf{I}\end{bmatrix} \ge 0\\
& \begin{bmatrix}\mathbf{w}_\mathbf{V} & \mathbf{w}_\mathbf{E}\end{bmatrix}\begin{bmatrix}\mathbf{F}\\ \mathbf{0}\end{bmatrix} = 1\\
& \mathbf{w}_\mathbf{V} \quad \text{unrestricted}\\
& \mathbf{w}_\mathbf{E} \ge \mathbf{0}
\end{aligned}$$

In which $\mathbf{w}_\mathbf{V}$ is "whether or not" vertex v is in S where $(S,\bar{S})$ represents a cut, $\mathbf{w}_\mathbf{E}$ is "whether

or not" an arc in in $\delta^+(S)$. $\mathbf{u}, \mathbf{E}, \mathbf{F}$ have the same meaning as in prime.

$$\mathbf{w_V} = \begin{bmatrix} w_1 & w_2 & \cdots & w_{|V|} \end{bmatrix}^\top$$
$$\mathbf{w_E} = \begin{bmatrix} w_{|V|+1} & w_{|V|+2} & \cdots & w_{|V|+|E|} \end{bmatrix}^\top$$

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

$$\begin{aligned} \min \quad & \sum_{e \in E} u_e w_e \\ \text{s.t.} \quad & w_i - w_j + w_{|V|+e} \geq 0, \forall e = (i,j) \in E \\ & - w_1 + w_{|V|} = 1 \\ & \mathbf{w_V} \quad \text{unrestricted} \\ & \mathbf{w_E} \geq \mathbf{0} \end{aligned}$$

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.

3.3 Maximum Flow Minimum Cut Theorem

Definition 3.4. Let P be a path, (not necessarily a dipath), P is called **unsaturated** if every **forward** arc is unsaturated ($x_e < u_e$) and every **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 3.3. *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):

$$\begin{aligned} D_1 &= \min\{u_e - x_e : e \text{ forward in } P\} \\ D_2 &= \min\{x_e : e \text{ backward in } P\} \\ D &= \min\{D_1, D_2\} \end{aligned}$$

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

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

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 3.4. *(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*

3.4 Ford-Fulkerson Method

Augmenting path An augmenting path is a path of edges in the residual graph with unused capacity greater than zero from the source s to target t. Every augmenting path has a bottleneck, which limits the flow that can go through this path.

Ford-Fulkerson method The Ford-Fulkerson method is a greedy algorithm with the following procedures:

- Iteratively find a path P from $s \to t$ with $c_f(e) = c(e) - f(e)$, where f is the current flow, and c_f is the so called residual capacity where

$$\forall (u,v) \in E : c_f(u,v) = c(u,v) - f(u,v), \quad \text{if } f(u,v) \geq 0 : c_f(v,u) = f(u,v) \tag{1}$$

- Then, define $\delta(P) = \min_{e \in P} c_f(e)$ and

$$f_P(e) = \begin{cases} \delta(P), & \text{if } e \in P \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

  then we assign new values to the flow at each edge

$$f_{new} = f_{old} + f_P \tag{3}$$

  where both flows satisfy noon-negativity, capacity, and flow conservation constraints. And by the definition of the residual capacities, the original capacities constraints are never violated.

- The procedure terminates when no such path $s \to t$ can be found with $c_f(e) \geq 0$.

The Ford-Fulkerson method

Algorithm 9 Ford-Fulkerson method

```
Initialize, set f(e) = 0, ∀e ∈ E, G_f = G
while ∃P = s → t ∈ G_f such that ∀e ∈ E, c_f(e) ≥ 0 do
    δ(P) = min_{e∈P} c_f(e)
    for ∀(u,v) ∈ P do
        if f(u,v) > 0 then
            f(v,u) = f(v,u) − δ(P)
        else
            f(u,v) = f(u,v) + δ(P)
    Update G_f
```

Some implementations

- Edmonds-Karp algorithm. Runs in $O(VE^2)$. The augmenting path is the shortest path found by breadth-first search.

- Dinic's algorithm. Runs in $O(V^2E)$. The augmenting path is the shortest path found by combining BFS and DFS.
- Capacity scaling. Use heuristic to find the augmenting path that has the largest flow.

4 Minimum Cost Flow Problem

4.1 Transshipment Problem

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

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

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

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

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.

4.2 Network Simplex Method

Lemma 4.1. *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 4.2. *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

$$\begin{aligned} w(T, e) = & \sum \{w_e : e \text{ forward in } C(T, e)\} \\ & - \sum \{w_e : e \text{ reverse in } C(T, e)\} \end{aligned} \tag{4}$$

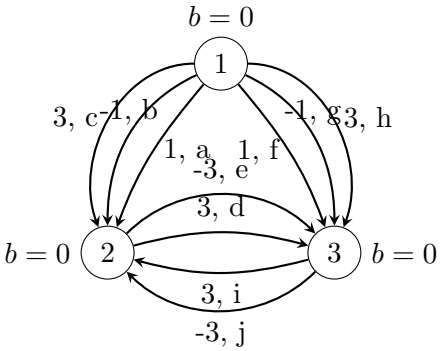
We think of $w(T, e)$ as the weight of $C(T, e)$.

Network Simplex Method The following algorithm describes the procedure of using simplex method to solve the transshipment problem

Algorithm 10 Network Simplex Method Algorithm

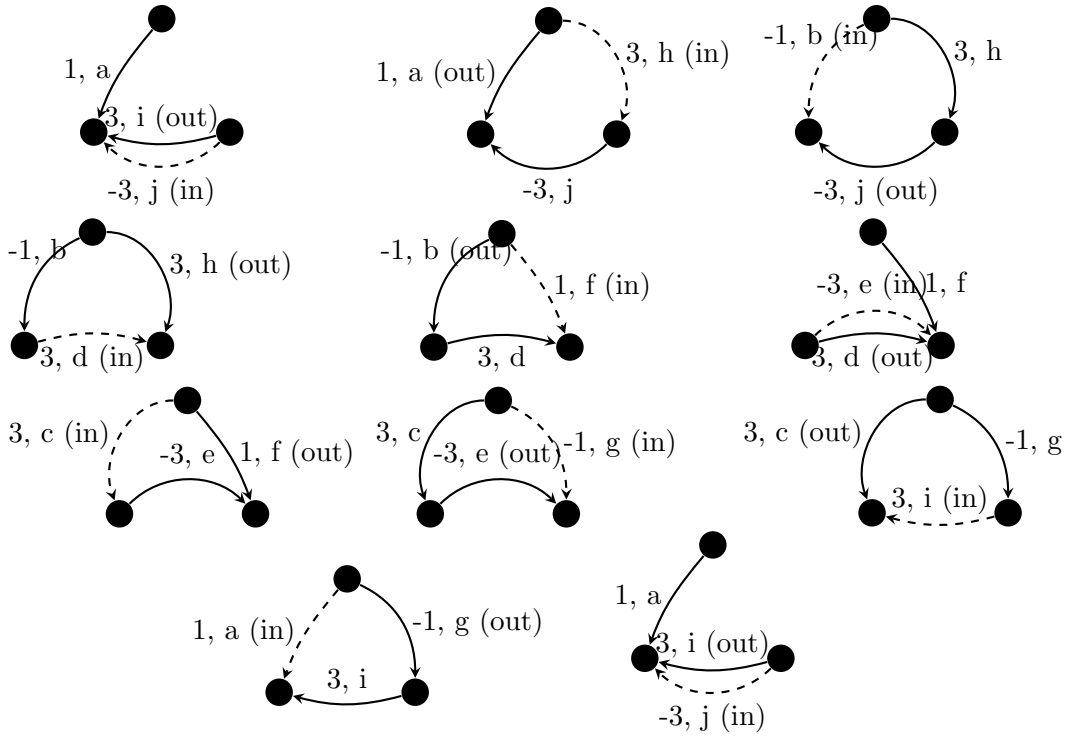
Ensure: An optimal solution or the conclusion that (D, b, w) is unbounded
Require: A transshipment problem (D, b, w) and a feasible tree solution x containing to a spanning tree T
while $\exists e \in A \setminus T, w(T, e) < 0$ **do**
let $e \in A \setminus T$ be such that $w(T, e) < 0$.
if $C(T, e)$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

Example for cycling Similar to Simplex Method in LP, there could 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.

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 $r \in T$.

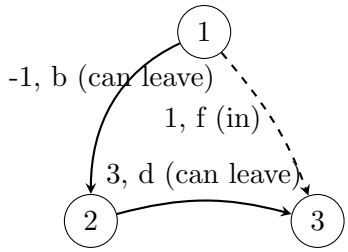
Modification to NSM:

- 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.

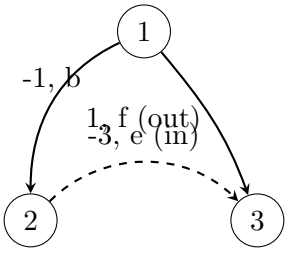
In the second rule above, r could also be in the cycle, in that case, i is r.

Continue the previous example. Now should how we can avoid cycling:

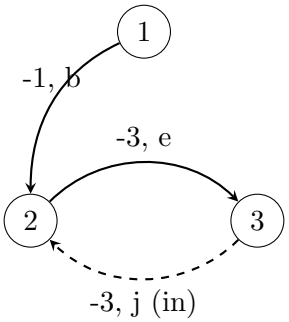
The first few (four) steps are the same as previous example, starting from



$w(T, f) = w_f - w_d - w_b = 1 - 3 + 1 = -1$. 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.



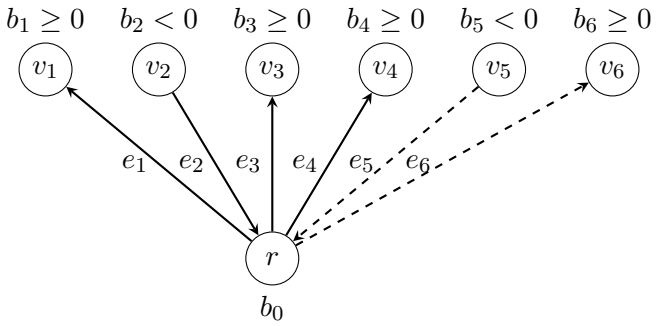
$w(T, e) = w_e - w_f + w_b = -5$, e is entering basis, f is leaving basis. Now the only arc to enter basis and maintain negative w is j.



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

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

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

Denote the origin graph

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

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

$$C(T, f) = re_{rv_{M'}}v_{M'}fv_MPr$$

For

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

where $d_e = 1$ if w_e is forward in P and $d_e = -1$ otherwise.

Now that $w_{e_{rv_{M'}}} = |V|(\max\{w_e : e \in \mathcal{A}\}) + 1$, it guarantees that

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

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 $C(T, f') = rP'v_Mf'v_{M'}e_{v_{M'}r}r$. $w(T, f') = w_{f'} - w_{e_{rv_{M'}}} + \sum_{e \in P'} d_e w_e$, 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. □