Discrete Mathematics

Graph theory

Pham Quang Dung

Hanoi, 2012

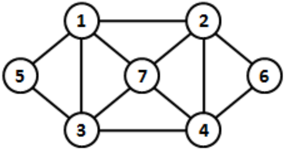
Outline

Introduction

- Many objects in our daily lives can be modeled by graphs
  - Internets, social networks (facebook), transportation networks, biological networks, etc.
- An graph G is a mathematical object consisting two finites sets, $G = (V, E)$
  - V is the set of vertices
  - E is the set of edges connecting these vertices
- Graphs have many types: directed, undirected, multigraphs, etc.

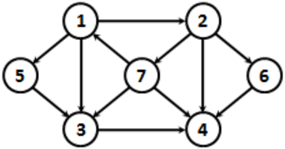
Definitions

- An undirected graph $G = (V, E)$
  - $V = (v_1, v_2, \ldots, v_n)$ is the set of vertices or nodes
  - $E \subseteq V \times V$ is the set of edges (also called undirected edges). E is the set of unordered pair (u, v) such that $u \neq v \in V$
  - $(u, v) \in E$ iff $(v, u) \in E$



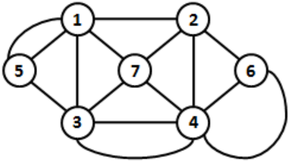
Definitions

- A directed graph $G = (V, E)$
  - $V = (v_1, v_2, \ldots, v_n)$ is the set of vertices or nodes
  - $E \subseteq V \times V$ is the set of arcs (also called directed edges). E is the set of ordered pair (u, v) such that $u \neq v \in V$



Multigraphs

- An undirected (directed) multigraph is a graph having multiples edges (arcs), i.e., edges (arcs) having the same endpoints
- Two vertices may be connected by more than one edges (arcs)



Definitions

- Given a graph $G = (V, E)$, for each $(u, v) \in E$, we say u and v are adjacent
- Given an undirected graph $G = (V, E)$
  - degree of a vertex v is the number of edges connecting it: $deg(v) = \sharp\{(u, v) \mid (u, v) \in E\}$
- Given a directed graph $G = V, E)$
  - An incoming arc of a vertex is an arc that enters it
  - An outgoing arc of a vertex is an arc that leaves it
  - indegree (outdegree) of a vertex v is the number of its incoming (outgoing) arcs

$$deg^{+}(v) = \sharp\{(v, u) \mid (v, u) \in E\}, deg^{-}(v) = \sharp\{(u, v) \mid (u, v) \in E\}$$

Definitions

Theorem

Given an undirected graph $G = (V, E)$, we have

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

Theorem

Given a directed graph $G = (V, E)$, we have

$$\sum_{v \in V} deg^{+}(v) = \sum_{v \in V} deg^{-}(v) = |E|$$

Definition - Paths, cycles

- Given a graph $G = (V, E)$, a path from vertex u to vertex v in G is a sequence $\langle u = x_0, x_1, \ldots, x_k = v\rangle$ in which $(x_i, x_{i+1}) \in E), \forall i = 0, 1, \ldots, k-1$
    - u: starting point (node)
    - v: terminating point
    - k is the length of the path (i.e., number of its edges)
- A cycle is a path such that the starting and terminating nodes are the same
- A path (cycle) is called simple if it contains no repeated edges (arcs)
- A path (cycle) is called elementary if it contains no repeated nodes

Connectivity

- Given an undirected graph $G = (V, E)$. G is called **connected** if for any pair (u, v) $(u, v \in V)$, there exists always a path from u to v in G
- Given a directed graph $G = (V, E)$, G is called
  - **weakly connected** if the corresponding undirected graph of G (i.e., by removing orientation on its arcs) is connected
  - **strongly connected** if for any pair (u, v) $(u, v \in V)$, there exists always a path from u to v in G
- Given an undirected graph $G = (V, E)$
  - an edge e is called **bridge** if removing e from G increases the number of connected components of G
  - a vertex v is called **articulation point** if removing it from G increases the number of connected components of G

Connectivity

Theorem

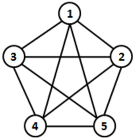
An undirected connected graph G can be oriented (each edge of G is oriented) to obtain a strongly connected graph iff each edge of G lies on at least one cycle

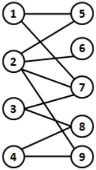
Proof.

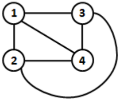
Exercise in class □

Special graphs

- Complete graphs K_n: undirected graph $G = (V, E)$ in which $|V| = n$ and $E = \{(u, v) \mid u, v \in V\}$
- Bipartie graphs $K_{n,m}$: undirected graph $G = (V, E)$ in which $V{=}X \cup Y$, $X \cap Y = \emptyset$, $|X| = n$, $|Y| = m$, $(u, v) \in E \Rightarrow u \in X \wedge v \in Y$
- Planar graphs: can be drawn on a plane in such a way that edges intersect only at their common vertices







Planar graphs - Euler Polyhedron Formula

Theorem

Given a connected planar graph having n vertices, m edges. The number of regions divided by G is $m - n + 2$.

Proof.

By induction on vertices (or edges) $\square$

Planar graphs - Kuratowski's theorem

Definition

A **subdivision** of a graph G is a new graph obtained by replacing some edges by paths using new vertices, edges (each edge is replaced by a path)

Theorem

Kuratowski *A graph G is planar iff it does not contain a subdivision of $K_{3,3}$ or K_5*

Proof.

$\square$

Outline

Graph representation

- Two standard ways to represent a graph $G = (V, E)$
  - Adjacency list
    - Appropriate with sparse graphs
    - $Adj[u] = \{v \mid (u, v) \in E\}, \forall u \in V$
  - Adjacency matrix
    - Appropriate with dense graphs
    - $A = (a_{ij})_{n \times n}$ such that (suppose $V = \{1, 2, \dots, n\}$)

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise} \end{cases}$$

- In some cases, we can use incidence matrix to represent a directed graph $G = (V, E)$

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i, \\ 1 & \text{if edge } j \text{ enters vertex } i, \\ 0 & \text{otherwise} \end{cases}$$

Outline

Depth-First Search (DFS)

- The DFS initially explore a selected vertex (called source)
- DFS explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it
- Once all of edges of v have been explored, the search **backtrack** to explore edges leaving the vertex from which v as discovered
- The process continues until all vertices reachable from the original source have been discovered
- If any undiscovered vertices remain, then DFS selects one of them as new source and start searching from it

Depth-First Search (DFS)

Algorithm 1: DFS-VISIT(G, u)

```
t ← t + 1;
u.d ← t;
u.color ← GRAY;
foreach v ∈ G.Adj[u] do
    if v.color = WHITE then
        v.p ← u;
        DFS-VISIT(G, v);
u.color ← BLACK;
t ← t + 1;
u.f ← t;
```

Depth-First Search (DFS)

Algorithm 2: DFS(G)

```
foreach u ∈ G.V do
    u.color ← WHITE;
    u.p ← NULL;
t ← 0;
foreach u ∈ G.V do
    if u.color = WHITE then
        DFS-VISIT(G, u);
```

Breadth-First Search (BFS)

- Given a graph $G = (V, E)$ and a source vertex s, the distance of a vertex v is defined to be the length (number of edges) of the shortest path from s to v
- BFS explores systematically vertices that are reachable from s
    - Explores vertices of distance 1, then
    - Explores vertices of distance 2, then
    - Explores vertices of distance 3, then
    - ...

Algorithm 3: BFS(G, s)

```
s.color ← GRAY;
s.d ← 0;
Q ← ∅;
Enqueue(Q, s);
while Q ≠ ∅ do
    u ← Dequeue(Q);
    foreach v ∈ G.Adj[u] do
        if v.color = WHITE then
            v.color ← GRAY;
            v.d ← u.d + 1;
            v.p ← u;
            Enqueue(Q, v);
    u.color ← BLACK;
```

Algorithm 4: BFS(G)

```
foreach u ∈ G.V do
    u.color ← WHITE;
    u.d ← ∞;
    u.p ← NULL;
foreach u ∈ G.V do
    if u.color=WHITE then
        BFS(G, u);
```

Compute Connected Components

- Given an undirected graph $G = (V, E)$, we want to compute all connected components of G
- Applying DFS (or BFS) for a given source vertex u will find all vertices of the same connected component of u

Algorithm 5: COMPUTE-CC(G)

foreach $u \in G.V$ **do**
    $u.color \leftarrow$ WHITE;

foreach $u \in G.V$ **do**
    if $u.color = WHITE$ **then**
        $C \leftarrow$ new set;
        DFS-CC(G, u, C);
        output(C);

Compute Connected Components

Algorithm 6: DFS-CC(G, u, C)

```
Insert(C, u);
u.color ← GRAY;
foreach v ∈ G.Adj[u] do
    if v.color = WHITE then
        DFS-CC(G, v, C);
```

Outline

Topological sort

- Given a directed acyclic graph (dag) $G = (V, E)$
- Order the vertices of G such that if (u, v) is an arc of G then u appears before v in the ordering

Topological sort

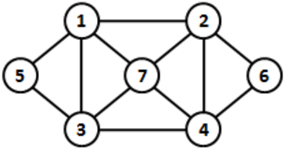
Algorithm 7: TOPO-SORT(G)

```
Compute in-degree d(v) of every vertex v of G;
Q ← ∅;
foreach v ∈ G.V do
    if d(v) = 0 then
        Enqueue(Q, v);
while Q ≠ ∅ do
    v ← Dequeue(Q);
    output(v);
    foreach u ∈ G.Adj[v] do
        d(u) ← d(u) − 1;
        if d(u) = 0 then
            Enqueue(Q, u);
```

Outline

Definition

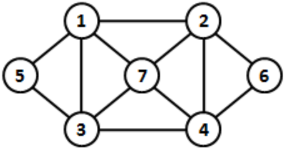
- A simple cycle (path) that visits each edge of an undirected graph $G = (V, E)$ exactly once is called **Eulerian cycle (path)** of G
- Graphs contain Eulerian cycles are called **Eulerian graphs**



Euler cycle is $1, 5, 3, 1, 7, 3, 4, 7, 2, 4, 6, 2, 1$

Definition

- A simple cycle (path) that visits each node of an undirected graph $G = (V, E)$ exactly once is called **Hamiltonian cycle (path)** of G
- Graphs contain Hamiltonian cycles are called **Hamiltonian graphs**



Hamilton cycle is $1, 2, 6, 4, 7, 3, 5, 1$

Theorem

An undirected connected graph $G = (V, E)$ is Eulerian iff each vertex of G has even degree

Proof.

By induction on edges $\square$

Algorithm for finding Euler cycles

Algorithm 8: EULER-CYCLE(G)

```
Stack S ← ∅;
Stack CE ← ∅;
u ← select a vertex of G.V;
Push(S, u);
while S ≠ ∅ do
    x ← Top(S);
    if G.Adj[x] ≠ ∅ then
        y ← select a vertex of G.Adj[x];
        Push(S, y);
        Remove (x, y) from G;
    else
        x ← Pop(S); Push(CE, x);
while CE ≠ ∅ do
    v ← Pop(CE);
    output(v);
```

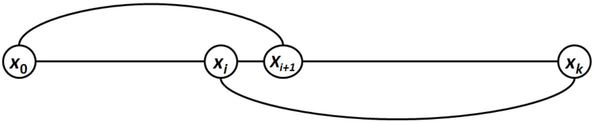
Dirak Theorem

Theorem

(Dirak 1952) *An undirected graph $G = (V, E)$ in which the degree of each vertex is greater or equal to $\frac{|V|}{2}$ is Hamiltonian*

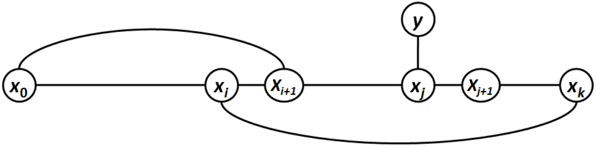
Dirak Theorem - proof

- G is connected, since otherwise the degree of any vertex in the smallest connected component C of G would be at most $|C| - 1 < \frac{|V|}{2}$ (contradiction)
- Let $P = x_0, x_1, \ldots, x_k$ be the longest elementary path of G
- All neighbors of x_0 and x_k lie on P because P cannot be extended to a longer path
- Pigeonhole principle: there exists some vertex x_i $(0 \leq i \leq k-1)$ such that $(x_0, x_{i+1}), (x_i, x_k) \in E$



Dirak Theorem - proof

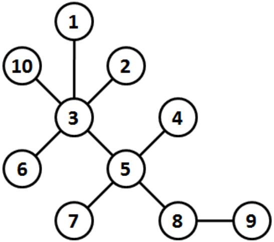
- Claim: the cycle $\mathcal{C} = x_0 x_{i+1} x_{i+2} \ldots x_{k-1} x_k x_i x_{i-1} \ldots x_1 x_0$ is Hamilton cycle of G,
- Otherwise
  - Since G is connected, there would be some vertex x_j of $\mathcal{C}$ s.t. $(x_j, y) \in E \wedge y \notin \mathcal{C}$
  - Take an elementary path P' containing k edges of $\mathcal{C}$ starting at x_j ($\mathcal{C}$ has $k+1$ edges)
  - Then, we could attach (x_j, y) to P' to obtain a longer path than P (contradiction)



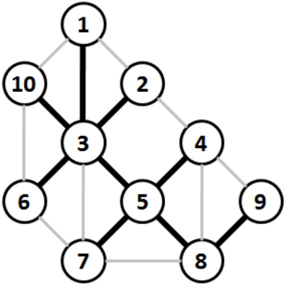
Outline

Tree and spanning trees

- A tree is an undirected connected graph containing no cycles
- A spanning tree of an undirected connected graph $G = (V, E)$ is a tree $T = (V, F)$ where $F \subseteq E$



a. Tree



b. Spanning tree (bold edges)

Theorem

Given an undirected graph $T = (V, E)$. We have

- *If T is a tree then T does not have any cycle and contains $|V| - 1$ edges*
- *If T does not have any cycle and contains $|V| - 1$ edges then T is connected*
- *If T is connected and contains $|V| - 1$ edges then each edge of T is a bridge*
- *If T is connected and each edge is a bridge then for each pair $u, v \in V$, there exists a unique path in T connected them*
- *If for each pair $u, v \in V$ there exists a unique path in T connected them, then T contains no cycle and a cycle will be created if we add an edge connecting any pair of its nodes*

Minimum Spanning Tree (MST)

- Given an undirected weighted graph $G = (V, E)$, each edge $e \in E$ is associated with a weight $w(e)$
- The weight of a spanning tree T is defined to be

$$w(T) = \sum_{e \in E_T} w(e)$$

where E_T is the set of edges of T
- Find a spanning tree of G such that the total weights on edges is minimal

Theorem

For any graph G having distinct weights on edges, the **MST** $\mathcal{T}$ *of* G *satisfies the following properties*

- *Cut property: For any cut $(X, \overline{X})$ of G, $\mathcal{T}$ must contain shortest edges crossing the cut*
- *Cycle property: Let C be a cycle in G, $\mathcal{T}$ does not contain the longest edges in C*

Minimum Spanning Tree - Proof of Cut property

Proof.

- **Proof by contradiction**
- Denote (x, y)=$\textbf{argMin}_{u \in X \wedge v \in \overline{X}}\{w(u, v)\}$
- Suppose that $\mathcal{T}$ does not contain (x, y)
- There exists a path $\mathcal{P}$ from x to y in $\mathcal{T}$ because $\mathcal{T}$ is connected
- $\mathcal{P}$ must contain an edge (u, v) such that $u \in X \wedge v \in \overline{X}$ $(w(u, v) > \epsilon = w(x, y))$
- Denote $\mathcal{T}'$ another spanning tree of G by replacing (u, v) of $\mathcal{T}$ by (x, y)
- Clear $w(\mathcal{T})' < w(\mathcal{T})$ because $w(x, y) < w(u, v)$ (**contradiction** with the hypothesis that $\mathcal{T}$ is one **MST** of G)

$\square$

Minimum Spanning Tree - Proof of Cycle property

Proof.

- **Proof by contradiction**
- Denote $\Delta = \max\{w(e) \mid e \in C\}$
- Suppose that $\mathcal{T}$ contains an edge $(x, y) \in C$ such that $w(x, y) = \Delta$
- Remove (x, y) from $\mathcal{T}$, we obtain two connected subtrees $\mathcal{T}_1$ containing x and $\mathcal{T}_2$ containing y
- C must contain an edge (u, v) such that u is in $\mathcal{T}_1$ and v is in $\mathcal{T}_2$ $(w(u, v) < w(x, y))$
- Replacing (x, y) of $\mathcal{T}$ by (u, v) yielding a strictly smaller spanning tree than $\mathcal{T}$ (**contradiction** with the hypothesis that $\mathcal{T}$ is a **MST** of G)

$\square$

Kruskal algorithm

Algorithm 9: KRUSKAL($G = (V, E)$)

$C \leftarrow$ set of edges of G;
$E_T \leftarrow \emptyset$;
$V_T \leftarrow \emptyset$;
while $|V_T| < |V|$ **do**
    $(u, v) \leftarrow$ a shortest edge of C;
    $C \leftarrow C \setminus \{(u, v)\}$;
    if $E_T \cup \{(u, v)\}$ *does not introduce any cycle* **then**
        $E_T \leftarrow E_T \cup \{(u, v)\}$;
        $V_T \leftarrow V_T \cup \{u, v\}$;
return (V_T, E_T);

Proof of the correctness of the Kruskal algorithm

Proof by contradiction

- Suppose that the tree T constructed by the KRUSKAL algorithm which is not a **MST** of the given graph G
- Suppose T^* is a **MST** of G having most edges in common with T
- Denote e_i the edge selected by the KRUSKAL algorithm at step $i, \forall i = 1, \ldots, n-1$
- Let e_k be the first edge of T (during the construction) that is not in T^* $(e_1, e_2, \ldots, e_{k-1} \in T^*)$
- Adding e_k to T^* creates a graph G^1 containing a cycle C
- There exists an edge $e' \neq e_k$ of C that is not in T, since otherwise T contains all edges of C and T is thus not a tree
- e_k is in T and not in T^*
- e' is in T^* and not in T

Proof of the correctness of the Kruskal algorithm

Proof by contradiction (continue)

- **Case 1**: If $w(e') < w(e)$:
  - $e_1, e_2, \ldots, e_{k-1}$ and e' are in T^*. Thus adding e' to the set $\{e_1, \ldots, e_{k-1}\}$ does not create any cycle
  - Hence, the KRUSKAL must select e' instead of e_k at the k^{th} step of the KRUSKAL during the construction of T (contradiction, case 1 does not happen)
- **Case 2**: $w(e') \geq w(e_k)$:
  - Replacing e' by e_k in T^* creates a new tree T^{**} having $w(T^{**}) \leq w(T^*)$. Clearly T^{**} is a **MST** and has one more edge (i.e., the edge e_k) in common with T than T^* (contradiction)

Prim algorithm

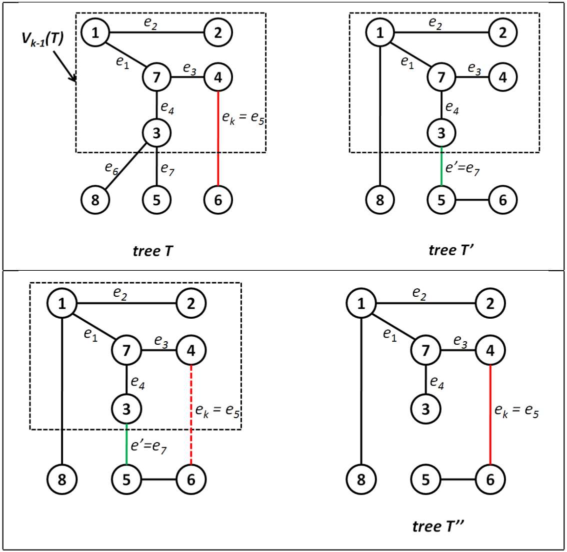
Algorithm 10: PRIM$(G = (V, E))$

$s \leftarrow$ select a vertex of V;
$S \leftarrow V \setminus \{s\}$;
$V_T \leftarrow \{s\}$;
$E_T \leftarrow \emptyset$;
foreach $v \in V$ **do**
    $d(v) \leftarrow w(s, v)$;
    $near(v) \leftarrow s$;
while $|V_T| < |V|$ **do**
    $v \leftarrow \text{argMin}_{u \in S} d(u)$;
    $S \leftarrow S \setminus \{v\}$;
    $V_T \leftarrow V_T \cup \{v\}$;
    $E_T \leftarrow E_T \cup \{(v, near(v))\}$;
    foreach $u \in S$ **do**
        if $d(u) > w(u, v)$ **then**
            $d(u) \leftarrow w(u, v)$;
            $near(u) \leftarrow v$;
return (V_T, E_T);

Proof of the correctness of the Prim algorithm

- Let T be the spanning tree computed by the Prim algorithm
- Let e_k is the edge selected and $V_k(T)$ be the set of vertices of T computed at the k^{th} iteration
- Suppose $T^{(0)}$ is a MST of G. If $T \neq T^{(0)}$, take the following action $\mathcal{A}$ on $T^{(0)}$ for generating $T^{(1)}$:
  - Let $e_k = (u, v)$ be the first edge chosen by Prim algorithm at k^{th} iteration which is not in $T^{(0)}$ ($u \in V_{k-1}(T)$ and $v \notin V_{k-1}(T)$, $e_1, e_2, ..., e_{k-1}$ are in $T^{(0)}$ and e_k is not in $T^{(0)}$)
  - Let P be the path from v to u in $T^{(0)}$ and (e') be the first edge when traversing along P from v to u s.t. one endpoint is in $V_{k-1}(T)$ and the other endpoint is not
  - Clearly $w(e_k) \leq w(e')$ by Prim algorithm selection
  - Replacing e' of $T^{(0)}$ by e_k yielding a spanning tree $T^{(1)}$ having weight less than or equal to the weight of $T^{(0)}$. Thus $T^{(1)}$ is also a MST where k edges $e_1, e_2, ..., e_k$ are included in $T^{(1)}$
- We continue the action $\mathcal{A}$ on $T^{(1)}$ if $T \neq T^{(1)}$, etc. The sequence of actions $\mathcal{A}$ finishes when obtaining T. Hence T is also a MST

Proof of the correctness of the Prim algorithm



Outline

Shortest path problem

- Given a graph $G = (V, E)$, each edge e is associated with a weight $w(e)$.
  - **Single-source shortest paths problem** Find the shortest paths from a given source node s to all other nodes of G
  - **All-pairs shortest paths problem** Find shortest paths between every pairs of vertices u, v in G

Bellman-Ford algorithms

- Graph without negative cycles

Algorithm 11: Bellman-Ford$(G = (V, E), s)$

foreach $v \in V$ **do**
    $d(v) \leftarrow w(s, v)$;
    $p(v) \leftarrow s$;
$d(s) \leftarrow 0$;
foreach $k = 1, \ldots, n-2$ **do**
    foreach $v \in V \setminus \{s\}$ **do**
        foreach $u \in V$ **do**
            if $d(v) > d(u) + w(u, v)$ **then**
                $d(v) \leftarrow d(u) + w(u, v)$;
                $p(v) \leftarrow u$;

Shortest path problem on directed acyclic graphs (DAG)

- Given a DAG $G = (V, E)$ and a source node $s \in V$. Find shortest paths from s to all other nodes of G

Algorithm 12: ShortestPathAlgoDAG($G = (V, E), s$)

$L \leftarrow$ Topological sort vertices of G;
foreach $v \in V$ **do**
    $d(v) \leftarrow w(s, v)$;
$d(s) \leftarrow 0$;
foreach $v \in L$ **do**
    foreach $u \in G.Adj[v]$ **do**
        $d(u) \leftarrow \min(d(u), d(v) + w(v, u))$;

Dijkstra algorithm

- Graph without negative edge weights

Algorithm 13: Dijkstra($G = (V, E), s$)

foreach $x \in V$ **do**
  $d(x) \leftarrow w(s, x)$;
  $pred(x) \leftarrow s$;

$NonFixed \leftarrow V \setminus \{s\}$;
$Fixed \leftarrow \{s\}$;
while $NonFixed \neq \emptyset$ **do**
  (*get the vertex v of $NonFixed$ such that $d(v)$ is minimal*);
  $v \leftarrow \text{argMin}_{u \in NonFixed} d(u)$;
  $NonFixed \leftarrow NonFixed \setminus \{v\}$;
  $Fixed \leftarrow Fixed \cup \{v\}$;
  foreach $x \in NonFixed$ **do**
    if $d(x) > d(v) + w(v, x)$ **then**
      $d(x) \leftarrow d(v) + w(v, x)$;
      $pred(x) \leftarrow v$;

Proof of the correctness of the Dijkstra algorithm

Proof by induction on $|Fixed|$

- Invariant: At any step, we have $d(v) = \min_{x \in Fixed}\{d(x) + w(x, v)\}, \forall v \notin Fixed$ (1)
- Base case $|Fixed| = 1$, trivial
- Inductive hypothesis (step k): $|Fixed| \leq k$, $d(x)$ is the shortest path distance from s to $x, \forall x \in Fixed$
- At step $k + 1$, we select a node $v \notin Fixed$ having smallest value of $d(v)$
  - $d(v) = d(x) + w(x, v)$ with $x \in Fixed$
  - Suppose that P is the shortest path from s to v and z is the first node when traversing from s to v on P that is not in $Fixed$: $P = s, x_1, x_2, \ldots, x_k, z, \ldots, v$ where $x_1, x_2, \ldots, x_k \in Fixed$
  - At this time, we have $d(z) \leq d(x_k) + w(x_k, z)$ because of invariant (1)
  - $w(P) = d(x_k) + w(x_k, z) + L \geq d(z)$ because L is the weight of a path from z to v on P which is greater or equal to 0
  - If $d(v)$ is not the shortest path distance from s to v, then $d(v) > w(P) \geq d(z)$ (contradiction with the hypothesis that $d(v)$=$\min_{x \notin Fixed} d(x) \leq d(z)$)

All-pairs shortest path - Floyd-Warshall algorithm

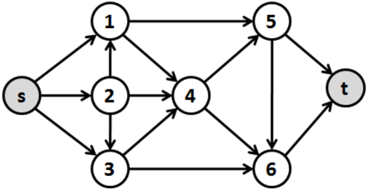
Algorithm 14: Floyd-Warshall($G = (V, E)$)

foreach $u \in V$ **do**
    foreach $v \in V$ **do**
        $d(u, v) \leftarrow w(u, v)$;
        $p(u, v) \leftarrow u$;

foreach $z \in V$ **do**
    foreach $u \in V$ **do**
        foreach $v \in V$ **do**
            if $d(u, v) > d(u, z) + d(z, v)$ **then**
                $d(u, v) \leftarrow d(u, z) + d(z, v)$;
                $p(u, v) \leftarrow p(z, v)$;

Outline

- **Capacitated Networks**
  - $G = (V, E, s, t)$ where V and E are respectively the set fo vertices and the set of arcs
  - Each arc $(u, v) \in E$ is associated with a nonnegative capacity $c(u, v)$
  - A source node s: no incoming arcs
  - A sink node t: no outgoing arcs
  - $A^-(v) = \{u \mid (u, v) \in E\}, \forall v \in V \setminus \{s\}$
  - $A^+(v) = \{u \mid (v, u) \in E\}, \forall v \in V \setminus \{t\}$



Network Flow

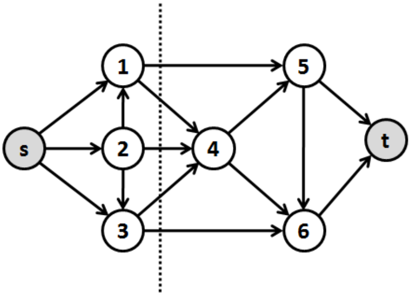
- A **flow** x on G: each arc $(u, v) \in E$ has a flow $x(u, v)$ traversing along it
  - $0 \leq x(u, v) \leq c(u, v), \forall (u, v) \in E$
  - Flow conservation

$$\sum_{v \in A^-(v)} x(v, u) = \sum_{v \in A^+(u)} x(u, v), \forall u \in V \setminus \{s, t\}$$

  - Flow value $f(x) = \sum_{v \in A^+(s)} x(s, v) = \sum_{v \in A^-(t)} x(v, t)$
- **Objective**: find a flow x on G such that $f(x)$ is **maximal**

Maximum Flow problem - Cut

- Given a network $G = (V, E, s, t)$, a $s - t$ cut $C = (S, T)$ is a partition of V such that $s \in S \wedge t \in T$
- Capacity of $s - t$ cut is $C(S, T) = \sum_{(u,v) \in S \times T} c(u, v)$



Maximum Flow problem - Cut

Lemma

For any flow x and a $s-t$ cut (S,T) of a network $G=(V,E)$, we have $f(x) \leq C(S,T)$

Proof.

$$
\begin{aligned}
f(x) \quad & = \textstyle\sum_{v\in A^+(s)} x(s,v) - \sum_{v\in A^-(s)} x(v,s) \text{ (by definition)} \\
& = \textstyle\sum_{v\in S}(\sum_{w\in A^+(v)} x(v,w) - \sum_{u\in A^-(v)} x(u,v)) \\
& \text{(by flow conservation constraint)} \\
& = \textstyle\sum_{v\in S, u\in T} x(v,u) - \sum_{v\in S,u\in T} x(u,v) \\
& \text{(by removing duplicate arcs)} \\
& \leq \textstyle\sum_{v\in S,u\in T} c(v,u) = c(S,T) \text{ because} \\
& \textstyle\sum_{v\in S,u\in T} x(v,u) \leq \sum_{v\in S,u\in T} c(v,u) \text{ and } \sum_{v\in S,u\in T} x(u,v) \geq 0
\end{aligned}
$$

□

Residual graph

- Given a network $G = (V, E, s, t)$ and a flow x on G. The residual graph $G_x = (V, E_x)$ is defined as follows:
  - If $(v, w) \in E$ with $x(v, w) = 0$, then $(v, w) \in E_x$ with weight $c(v, w)$
  - If $(v, w) \in E$ with $x(v, w) = c(v, w)$, then $(w, v) \in E_x$ with weight $x(v, w)$
  - If $(v, w) \in E$ with $0 < x(v, w) < c(v, w)$, then $(v, w) \in E_x$ with weight $c(v, w) - x(v, w)$ and $(w, v) \in E_x$ with weight $x(v, w)$
- If we can find a path $P = s = v_0, v_1, \ldots, v_k = t$ on G_x and denote δ the minimum weight of edges on P, then we can construct a flow x' as follows:

$$x'(u, v) = \begin{cases} x(u, v) + \delta, & \text{if } (u, v) \in P \wedge (u, v) \in E \\ x(u, v) - \delta, & \text{if } (u, v) \in P \wedge (u, v) \notin E \\ x(u, v), & \text{if } (u, v) \notin P \end{cases}$$

AND we have $f(x') = f(x) + \delta > f(x)$

Maximum Flow prolem

Theorem

Given a network $G = (V, E, s, t)$ and a flow x on G. If we cannot find a path from s to t on the residual graph G_x, then there exists a cut (S, T) such that $f(x) = C(S, T)$

Maximum Flow prolem

Proof.

- Let S be the set of vertices of V that can be reachable from s on G_x and $T = V \setminus S$
- We claim that for $(v, w) \in E_x$ with $v \in T$ and $w \in V$, we have $x(v, w) = 0$, since otherwise ($x(v, w) > 0$), (w, v) must be included in E_x by construction rule of G_x, v can thus be reachable from s (contradiction with the hypothesis that v is in T which is not reachable from s on G_x)
- We have $f(x) = \sum_{v \in S, u \in T} x(v, u) - \sum_{v \in S, u \in T} x(u, v) = \sum_{v \in S, u \in T} x(v, u)$
- Since $(v, u) \notin E_x$, then $x(v, u) = c(v, u)$ (by construction rule of G_x)
- Hence $f(x) = C(S, T)$

$\square$

Algorithm 15: FordFulkerson($G = (V, E, s, t)$)

foreach $(u, v) \in E$ **do**
    $x(u, v) \leftarrow 0$;
while *true* **do**
    G_x is the residual graph of G w.r.t flow x;
    $P \leftarrow$ FindPath(G_x, s, t);
    if $P = NULL$ **then**
        break;
    $x \leftarrow$ augmentFlow(P, x, G_x);
return x;