Graph algorithms and applications

Pham Quang Dung and Do Phan Thuan

Computer Science Department, SoICT,
Hanoi University of Science and Technology.

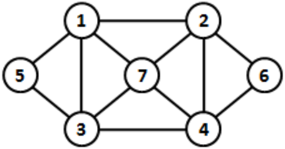
August 15, 2016

Recall Graphs and related concepts

- Many objects in our daily lives can be modelled 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.

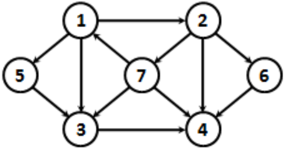
Definitions

- 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 $(v, u) \in E$



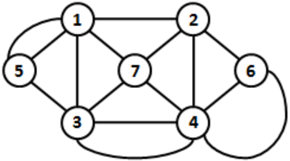
Definitions

- 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

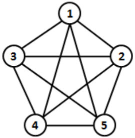
Connectivity

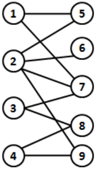
Theorem

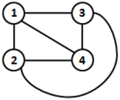
An undirected connected graph G can be oriented (each edge of G is oriented) to obtain a strongly connected graph iff each edge of G lies on at least one cycle

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 $m - n + 2$.

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*

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 $V = \{1, 2, \ldots, n\}$)

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

Graph representation

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

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

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

- Important information recorded during the DFS
  - $u.d$ is the discovery time: time point when the vertex u is first discovered
  - $u.f$ is the finishing time: time point when the search finishes examining adjacency list of the vertex u

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);
```

Depth-First Search (DFS)

For any two vertices u and v, exactly one of the following conditions holds:

- $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the DFS forest
- $[u.d, u.f]$ is contained entirely within $[v.d, v.f]$, and u is a descendant of v in the DFS forest
- $[v.d, v.f]$ is contained entirely within $[u.d, u.f]$, and v is a descendant of u in the DFS forest

Depth-First Search (DFS)

Edges classification

- **Tree edges**: (u, v) is a tree edge if v was first discovered by exploring edge (u, v)
- **Back edges**: (u, v) is a back edge if v is an ancestor of u in the DFS tree
- **Forward edges**: (u, v) is a forward edge if u is an ancestor of v in the DFS tree
- **Crossing edges**: remaining edges of the given graph

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
  - ...

Breadth-First Search (BFS)

Algorithm 3: BFS(G, s)

$s.color \leftarrow$ GRAY;
$s.d \leftarrow 0$;
$Q \leftarrow \oslash$;
Enqueue(Q, s);
while $Q \neq \oslash$ **do**
    $u \leftarrow$ Dequeue(Q);
    foreach $v \in G.Adj[u]$ **do**
        if $v.color = WHITE$ **then**
            $v.color \leftarrow$ GRAY;
            $v.d \leftarrow u.d + 1$;
            $v.p \leftarrow u$;
            Enqueue(Q, v);
    $u.color \leftarrow$ BLACK;

Breadth-First Search (BFS)

Algorithm 4: BFS(G)

foreach $u \in G.V$ **do**
- $u.color \leftarrow$ WHITE;
- $u.d \leftarrow \infty$;
- $u.p \leftarrow$ NULL;

foreach $u \in G.V$ **do**
- **if** $u.color = WHITE$ **then**
  - BFS(G, u);

Applications of DFS, BFS

- BFS and DFS: Compute connected components of a given graph
- BFS: Find shortest path (the length of a path is defined to be the number of edges of the path)
- BFS: Check if a given graph is a bipartite graph
- BFS and DFS: Detect cycle of an undirected graph
- DFS: compute strongly connected components of a given directed graph
- DFS: compute bridges and articulation points of an undirected connected graph
- DFS: topological sort on a directed acyclic graph (DAG)
- BFS and DFS: Find the longest path on a tree

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 ∈ G.V do
    u.color ← WHITE;
foreach u ∈ G.V do
    if u.color = WHITE then
        C ← new set;
        DFS-CC(G, u, C);
        output(C);
```

Compute Connected Components

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

Insert(C, u);
$u.color \leftarrow$ GRAY;
foreach $v \in G.Adj[u]$ **do**
    if $v.color = WHITE$ **then**
        DFS-CC(G, v, C);

Compute strongly connected components

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

1. Call DFS(G) to compute finishing time for all vertices V
2. Compute the residual graph $G^T = (V, E^T)$ of G:
   $E^T = \{(u, v) \mid (v, u) \in E\}$
3. Call DFS(G^T), but in the main LOOP, consider the vertices of V in a decreasing order of finishing time computed in line 1
4. Vertices of each tree in the DFS forest of line 3 form a strongly connected component of G

Check if a given graph is bipartite

- Call BFS from some vertex
- Color even-level vertices by "BLACK" and odd-level vertices by "WHITE"
- If there exists an edge such that both endpoints have the same color, then G is not bipartite

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: using DFS

- Call DFS(G) to compute finishing time for all vertices
- Whenever each vertex is finished, insert it onto the front of a linked list L
- Return the linked list L

Topological sort: using queues

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);
```

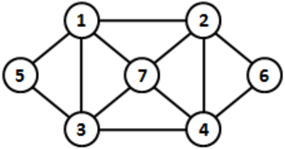
Find the longest path on a tree

1. Apply a BFS (or DFS) from a node s to find the furthest node v from s
2. Apply a BFS (or DFS) from the node v (found in step 1) to find the furthest node u from v
3. The unique path from u to v is the longest path on the given tree

Euler and Hamilton cycles

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**

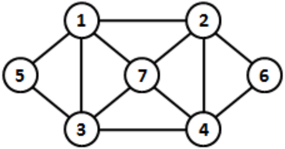


Euler cycle is $1, 5, 3, 1, 7, 3, 4, 7, 2, 4, 6, 2, 1$

Euler and Hamilton cycles

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

Euler and Hamilton cycles

- G is connected and degree of each node is even. Hence, the degree of each node is greater or equal to 2
- $\Rightarrow$ there exists a cycle $C = v_1, v_2, ..., v_k, v_1$ on G
- Remove all edges of C, we obtain a graph G' which is divided into connected components $G_1, ..., G_q$.
- Each G_i is connected and the degree of each node of G_i is even.
- $\Rightarrow$, there exists an euler cycle C_i on G_i
- We construct the euler cycle of G as follows:
  - Start from v_1, we traverse along the euler cycle of the connected component containing v_1 and terminate at v_1
  - Go to v_2. If the connected component containing v_2 has not been visited, then we go along the euler cycle of this connected component from v_2 and terminate at v_2
  - Go to v_3. If the connected component containing v_2 has not been visited, then we go along the euler cycle of this connected component from v_3 and terminate at v_3
  - ...
  - Go back to v_1

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);
```

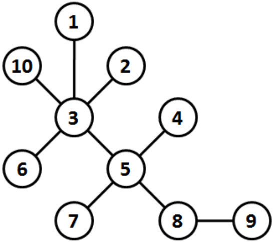
Dirak Theorem

Theorem

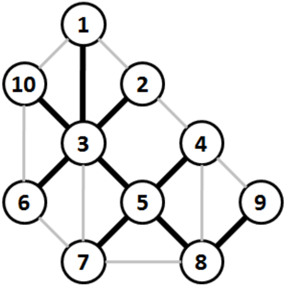
(Dirak 1952) *An undirected graph* $G = (V, E)$ *in which the degree of each vertex is greater or equal to* $\frac{|V|}{2}$ *is Hamiltonian*

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*

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);

Prim algorithm

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

```
s ← select a vertex of V;
S ← V \ {s};
V_T ← {s};
E_T ← ∅;
foreach v ∈ V do
    d(v) ← w(s, v);
    near(v) ← s;
while |V_T| < |V| do
    v ← argMin_{u∈S} d(u);
    S ← S \ {v};
    V_T ← V_T ∪ {v};
    E_T ← E_T ∪ {(v, near(v))};
    foreach u ∈ S do
        if d(u) > w(u, v) then
            d(u) ← w(u, v);
            near(u) ← v;
return (V_T, E_T);
```

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 ∈ V do
    d(v) ← w(s, v);
    p(v) ← s;
d(s) ← 0;
foreach k = 1, ..., n − 2 do
    foreach v ∈ V \ {s} do
        foreach u ∈ V do
            if d(v) > d(u) + w(u, v) then
                d(v) ← d(u) + w(u, v);
                p(v) ← 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**
      $d(x) \leftarrow d(v) + w(v, x)$;
      $pred(x) \leftarrow v$;

All-pairs shortest path - Floyd-Warshall algorithm

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

```
foreach u ∈ V do
    foreach v ∈ V do
        d(u, v) ← w(u, v);
        p(u, v) ← u;
foreach z ∈ V do
    foreach u ∈ V do
        foreach v ∈ V do
            if d(u, v) > d(u, z) + d(z, v) then
                d(u, v) ← d(u, z) + d(z, v);
                p(u, v) ← p(z, v);
```

Two Disjoint Shortest Paths - Suurballe algorithm

SAMSUNG

Algorithm 15: Suurballe(G, s, t)

Input: $G = (V, E, w)$, $s, t \in V$
Output: Set of arcs of Pair Disjoint Shortest Paths from s to t in G

```
Apply the Dijkstra algorithm on G;
P1 ← Shortest path from s to t in G;
foreach v ∈ V do
    dG(s, v) ← shortest distance from s to v in G;
foreach (u, v) ∈ E do
    w'(u, v) ← w(u, v) − dG(s, v) + dG(s, u);
E' ← {};
foreach arc (u, v) ∈ E do
    if (u, v) ∈ P1 then
        if (v, u) ∉ E then
            E' ← E' ∪ {(v, u)};
        w'(v, u) ← 0;
    else
        E' ← E' ∪ {(u, v)};
G' ← (V, E', w');
Apply the Dijkstra algorithm on G';
P2 ← the shortest path from s to t in G';
P ← {(u, v) | (u, v) ∈ P1 ∧ (v, u) ∈ P2 ∨ (u, v) ∈ P2 ∧ (v, u) ∈ P1};
return set of arcs of P1 and P2 excluding P;
```