Notes for Operations Research & More

Lan Peng, PhD Student

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

September 10, 2019

September 10, 2019

Contents

Part I

Graph and Network Theory

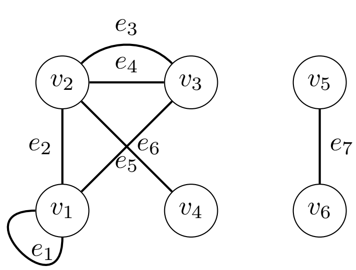
Chapter 1

Basic concepts

1.1 Graph

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

For example, the following graph



can be represented as

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

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

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

1.2 Subgraph

Definition 1.2.1 (subgraph)**.** Given two graphs G and H, H is a **subgraph** of G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ and an edge has the same ends in H as it does in G. Furthermore, if $E(H) \neq E(G)$ then H is a proper subgraph.

Definition 1.2.2 (spanning)**.** A subgraph H on G is **spanning** if $V(H) = V(G)$

Definition 1.2.3 (vertex-induced, edge-induced)**.** For a subset $V^{'} \subseteq V(G)$ we define an **vertex-induced** subgraph $G[V^{'}]$ to be the subgraph with vertices $V^{'}$ and those edges of G having both ends in $V^{'}$. The **edge-induced** subgraph $G[E^{'}]$ has edges $E^{'}$ and those vertices of G that are ends to edges in $E^{'}$.

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

Definition 1.2.4 (degree)**.** Let $v \in V(G)$, then the **degree** of $v \in V(G)$ denote by $d_G(v)$ is defines to be the number of edges incident of v. Loops counted twice.

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

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

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

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

Solution. The maximum number is 12.

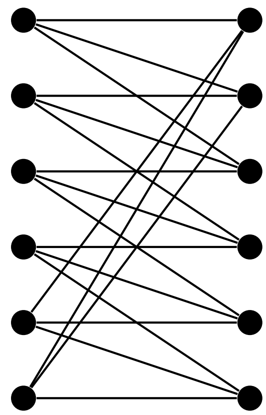
Proof. First we prove 12 vertices is possible, then we prove 13 vertices is not possible

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

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

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

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



□

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

Proof.

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

□

Chapter 2

Paths, Trees, and Cycles

2.1 Walk

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

Example.

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

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

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

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

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

(The reverse walk is guaranteed to exist because it is an undirected graph)

Given two walks w and w' we can create a third walk denoted by ww' by concating w and w'. The new walk's origin is the same as terminal.

2.2 Path and Cycle

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

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

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

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

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

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

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

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

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

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

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

- If G is connected, i.e., for all vertices, the degree is greater than 0. Also the graph is simple, for a graph with $|N|$ vertices, the degree of each vertex is less or equal to $|N| - 1$ (cannot have loop or parallel edge). For $|N|$ vertices, to make sure there is no two vertices that has same degree, it will need $|N|$ options for degrees, however, we only have $|N| - 1$ option. According to pigeon in holes principle, there has to be at least two vertices with the same degree.
- If G is not connected, i.e., the graph has more than one component. One of the following situation will happen:
  - For all components, each component contains only one vertex. Since we have at least two vertices, which means there are at least two component that has only one vertex. For those vertices, at least two vertices has the same degree as 0.

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

□

2.3 Tree and forest

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

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

Problem 2.3. Prove that if T is a tree, then T has exactly one more vertex than it has edges.

Proof. 1. First we prove for any tree T that has at least two vertices, there has to be at least one leaf, i.e., now we prove that we can find u with degree of 1. Proof by constructing algorithm. (In fact we can prove that there are at least two leaves.)

Algorithm 1 Find one leaf in a tree

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

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

2. Then, if we remove one leaf in the tree, i.e., we remove an edge and a vertex, where that vertex only connects to the edge we removed. One of the following situations will happen:

   (a) Situation 1: The remaining of T is one vertex. In this case, T has two vertices an one edge. (Exactly one more vertex than it has edges)

   (b) Situation 2: The remaining of T is another tree $T^{'}$ (removal of edges will not change acyclic), where $|V(T)| = |V(T^{'})| + 1$ and $|E(T)| = |E(V^{'}| + 1$. (one edge and one vertex has been removed)

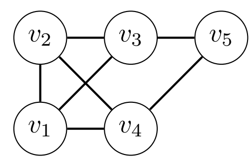
3. Do the leaf removal process recursively to $T^{'}$ if Situation 2 happens until Situation 1 happens.

□

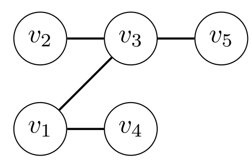
2.4 Spanning tree

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

Example. In the following graph



This is a spanning tree



Problem 2.4. Prove that if T_1 and T_2 are spanning trees of G and $e \in E(T_1)$, then there exists a $f \in E(T_2)$, such that $T_1 - e + f$ and $T_2 + e - f$ are both spanning trees of G.

Proof. □

Theorem 2.1. *Every connected graph has a spanning tree.*

Proof. Prove by constructing algorithm: □

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

For the complexity of spanning tree algorithm:

1. First we need to input the data, create an array such that the first and the second entries are the ends of e_1, third and fourth are the ends of e_2, and so on.

2. The amount of storage needs in $2|E|$, which is $O(|E|)$

Algorithm 2 Find a spanning tree for connected graph

Require: a connected graph G and an enumeration $e_1, ...e_m$ of the edges of G
Ensure: a spanning tree T of G

```
1: Let T be the spanning subgraph of G with V(T) = V(G) and E(T) = ∅
2: i ← 1
3: while i ≤ |E| do
4:   if T + e_i is acyclic then
5:     T ← T + e_i
6:     i ← i + 1
7:   end if
8: end while
```

3. The main work involved in the algorithm is for each edges e_i and the current T, to determine if $T + e_i$ creates a cycle.

4. At every stage T has certain components $V_1, ...V_t$, (every time we add an edge, the number of components minus 1)

5. So at the beginning $t = |V|$ with $|V_i| = 1 \forall i$

6. At the end, $t = 1$

7. suppose we keep each component V_i by keeping for each vertex a pointer from the vertex to the name of the component containing it. Thus if $\mu \in V_3$, there will be a pointer from μ to integer 3.

8. Then when edge $e_i = \mu v$ is encountered in Step 2, we see that $T + e_i$ contains a cycle if and only if μ and v point to same integer which means they are in the same component

9. If they are not in the same component, we want to add the edge which means then I have to update the pointers.

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

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

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

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

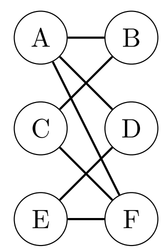
2.5 Special Graphs

Definition 2.5.1. 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 2.5.2. A **cycle** graph $C_n (n \geq 3)$ consists of n vertices $v_1, ...v_n$ and n edges $\{v_1, v_2\}, \{v_2, v_3\}, ...\{v_{n-1}, v_n\}$

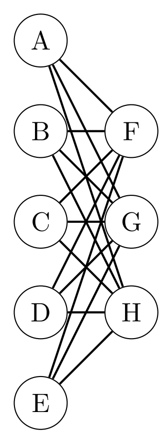
Definition 2.5.3. A **wheel** graph $W_n (n \geq 3)$ is a simple graph obtains by adding one vertex to the cycle graph C_n, and connecting this new vertex to all vertices of C_n

Definition 2.5.4. A simple graph is said to be **bipartite** if the vertex set can be expressed as the union of two disjoint non-empty subsets V_1 and V_2 such that every edges has one end in V_1 and another end in V_2



Definition 2.5.5 (complete bipartite)**.** The **complete bipartite** graph K_{mn} is the bipartite graph V_1 containing m vertices and V_2 containing n vertices such that each vertiex in V_1 is adjacent to every vertex in V_2

Example. Here is an example for K_{53}



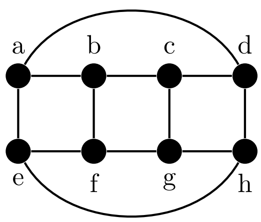
Theorem 2.2. *(König) A graph G is bipartite iff every cycle is even.*

Proof. Hereby we prove the $\Rightarrow$ and $\Leftarrow$

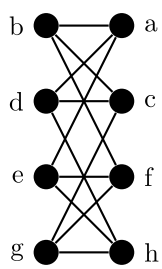
- (⇒) If the graph G is bipartite, by definition, the vertices of graph can be partition into two groups, that within the group there is no connection between vertices. Therefore, for each cycle, the odd index of vertices and even index of vertices has to be choose alternative from each groups.
- (⇐)

□

Example. The following graph is bipartite, it only contains even cycles.



We can rearrange the graph to be more clear as following



The vertices of graph G can be partition into two sets, $\{a, c, f, h\}$ and $\{b, d, e, g\}$

Example. The following graph is not bipartite

The cycle $c = v_1 v_1 1 v_4 v_3 v_2$ have odd number of vertices.

2.6 Complexity

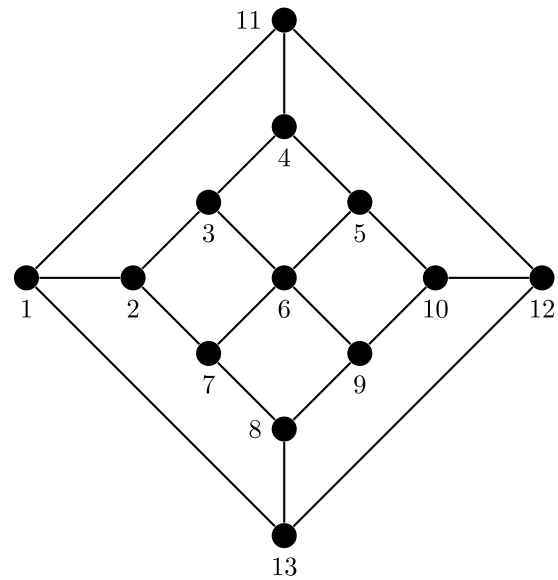
FIXME This part is going to be moved to Algorithm notes (or I will just delete this part because of duplication). We want to know guaranteed performances - "worse case" scenarios - for any algorithm working on any problem instance.

The following example is for addition of two matrices:

The "running time" of an algorithm is measured by the number of basic operational steps.

For so called "basic" steps, it includes

- $+$, $-$, $\times$, $\div$



Algorithm 3 Add two $m \times n$ matrices A, B to get matrix C

for $i = 1, 2, ..., m$ **do**
  for $j = 1, 2, ..., n$ **do**
    $C_{ij} = A_{ij} + B_{ij}$
  end for
end for

- assignments and storage of a variable
- comparisons

For the example above

- $c_1 mn$ for addition $C_{ij} = A_{ij} + B_{ij}$
- $c_2 mn$ for saving C_{ij}
- $c_3 mn$ for comparison and assignment for i and j

c_1, c_2, c_3 does not matter, the number of steps are $m \times n$, we say the algorithm runs $O(mn)$ (big O notation, the worse case)

Example. $f(n) = 3n^3 + 4n^2$
Claim: $f(n)$ is $O(n^3)$
Proof: Find a c for $cn^3 - 3n^3 - 4n^2$ that there exist a n_0, for $\forall n \geq n_0$, the inequality holds.

2.7 O notation, Omega notation, Theta notation

Definition 2.7.1 (O notation)**.** An algorithm is said to run in $O(f(n))$ time if for some constant $C(\geq 0)$, and n_0 the time takes algorithm is at most $Cf(n)$ for all $n \geq n_0$

Definition 2.7.2 (Ω notation)**.** An algorithm is said to run in $\Omega(f(n))$ time if for some constant $C^{'}(\geq 0)$, and n_0. For all instance $n \geq n_0$, the time takes algorithm is at least $C^{'} f(n)$ for some instances.

Definition 2.7.3 (Θ notation)**.** An algorithm is said to be $\Theta(f(n))$ if it is both $O(f(n))$ and $\Omega(f(n))$

Example. $\frac{1}{2}n^2 - 3n$, $\Omega(n^2) \leq \Theta(n^2) \leq O(n^2)$
Find $c^{'}$ and c (both ¿ 0) where $c^{'}n^2 \leq \frac{1}{2}n^2 - 3n \leq cn^2$ for a given n_0, $\forall n \geq n_0$ the inequality holds.
Let $c^{'} = \frac{1}{14}$ and $c = \frac{1}{2}$, $\forall n \geq n_0$, where $n_0 = 7$, the inequality always holds.

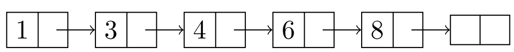
2.8 Representation of data

For set $\{1, 3, 4, 6, 8\}$ we have two ways to represent, array and list.

For array, the representation is

1	2	3	4	5	6	7	8
1	0	1	1	0	1	0	1

For list, the representation is



There is no "better" representation, one can be better than the other one in different cases.

Example. For $\emptyset$, array need to build a space with n cells of 0, complexity $O(n)$, list just need to build the first cell, $O(1)$

Example. To find if a number is in a set, array needs $O(1)$ (look at the address), list needs $O(n)$ (loop through list, worst case, entire list)

2.9 Size of a problem

Number of bits needed to store the problem

Example. Let $0 \leq A_{ij} \leq 2^{51}$

Representation by the bits of the number, $A_{ij} \leq 2^51$ can be represented by max of $O(log(A_{ij}))$, (51 bits)

For matrix addition that the total storage for addtion two matrix is $O(mnk)$, $k = O(log|A_{mn}|)$

Chapter 3

Shortest-Path Problem

Chapter 4

Minimum Spanning Tree Problem

Chapter 5

Maximum Flow Problem

Chapter 6

Minimum Cost Flow Problem

Chapter 7

Assignment and Matching Problem

Chapter 8

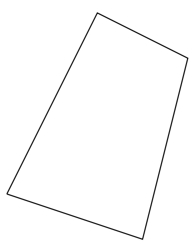
Graph Algorithms

Chapter 9

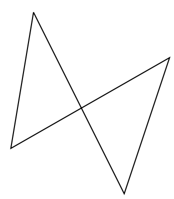
Polygon Triangulation

9.1 Types of Polygons

Definition 9.1.1 (simple polygon)**.** A **simple polygon** is a closed polygonal curve without self-intersection.



Simple Polygon

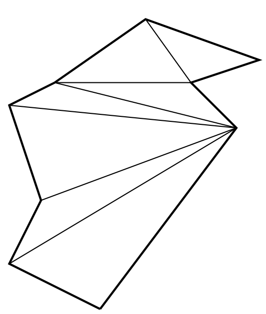


Non-simple Polygon

Polygons are basic building blocks in most geometric applications. It can model arbitrarily complex shapes, and apply simple algorithms and algebraic representation/manipulation.

9.2 Triangulation

Definition 9.2.1 (Triangulation)**.** **Triangulation** is to partition polygon P into non-overlapping triangles using diagonals only. It reduces complex shapes to collection of simpler shapes. Every simple n-gon admits a triangulation which has $n-2$ triangles.



Triangulation

Theorem 9.1. *Every polygon has a triangulation*

theorem 9.2. *Every polygon with more than three vertices has a diagonal.*

Proof. (by Meisters, 1975) Let P be a polygon with more than three vertices. Every vertex of a P is either *convex* or *concave*. W.L.O.G.(any polygon must has convex corner) Assume p is a convex vertex. Denote the neighbors of p as q and r. If $\bar{qr}$ is a diagonal, done, and we call $\triangle pqr$ is an *ear*. If $\triangle pqr$ is not an ear, it means at least one vertex is inside $\triangle pqr$, assume among those vertexes inside $\triangle pqr$, s is a vertex closest to p, then $\bar{ps}$ is a diagonal. □

9.3 Art Gallery Theorem

Theorem 9.3. *Every n-gon can be guarded with $\lfloor \frac{n}{3} \rfloor$ vertex guards*

theorem 9.4. *Triangulation graph can be 3-colored.*

Problem 9.1. The floor plan of an art gallery modeled as a simple polygon with n vertices, there are guards which is stationed at fixed positions with 360 degree vision but cannot see through the walls. How many guards does the art gallery need for the security? (Fun fact: This problem was posted to Vasek Chvatal by Victor Klee in 1973).

Proof. - P plus triangulation is a planar graph
- 3-coloring means there exist a 3-partition for vertices that no edge or diagonal has both endpoints within the same set of vertices.
- Proof by Induction:
  - Remove an ear (there will always exist ear)
  - Inductively 3-color the rest
  - Put ear back, coloring new vertex with the label not used by the boundary diagonal. □

9.4 Triangulation Algorithms

9.5 Shortest Path