Graph I

一、Basic knowledge and terminology 术语

1、G(V,E)

V={1,2,3,4} |V|=4

E={(1,2),(2,3),(3,4),(2,4)} |E|=4

2、Adjacency 邻接 list 小表 / matrix 矩

(1) Adj[u]={v:(u,v)∈E}   (2) matrix   若是无向图

1: 2
2: 3, 4
3: 4
4: ∅

u\v	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	1
4	0	0	0	0

list: Adj[u][i] = Adj[i][u]

matrix: 对称矩阵

O(V+E) ⇒ sparse 稀疏 graph

O(V²) ⇒ dense 密集 graph

3、weight

1: (2:1),
2: (3:4), (4:5)
3: (4:2)

u\v	1	2	3	4
1	∞	1	∞	∞
2	∞	∞	4	5
3	∞	∞	∞	2
4	∞	∞	∞	∞

4、BFS 与 DFS，以及它们的应用 复习原理+代码+应用

4(1) undirected   (2) directed

推论1, $\sum_{v\in V}\deg(v)=2|E|$ 边数×2=总度数

推论2, 奇度顶点数量为偶

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

ex: $\deg^-(4)=2$, $\deg^+(4)=0$

Neighborhood of 2: {1,3,4} set

5、subgraph

G'=(V',E') V'⊆V

G=(V,E) E'⊆E

H是G子图   也是子图

6、P15   8、应用 P20

7、special graph

(1) complete graph 任意2点间都有边   4色理论

(2) Cycle

(3) d-dimensional graph 数字仅为0,1，位数由点个数决定；相邻2个点，恰有1位不同；$d=\lceil\log_2 N\rceil$

(4) Bipartite graph 二分图 2-colorable

1个图分为2个set: V_1, V_2

每个集合内部的点均无边

$V\in V_1$，V必与V_2中1个及以上的点相连 → $V\in V_2$ ... 相联

是二分图   非二分图

推论 ⇒ 含奇数长度环的图，不是二分图

二、Graph isomorphism 图同构 P24 use adjacency matrix

① 判断 点与边数量是否相同

② 判断 2个图边的对应关系

$G_1: a\to b$

$G_2: F(a)\to F(b)$ ⇒ one-to-one 双射

→ 也可用矩阵判断，可画出相同矩阵

G_1, G_2 相同构

③ 判断 degree(点的度数)：G_1有 deg(x)=1，但 G_2 没有

三、Connectivity 连通性

1、connected graph: 只要从1个点到另1个点有path就行

G_1, G_2 都是连通图

A可通过 A→B, B→C，到达C

2、connected component 分量

连通分量也叫最大连通子图

① 必须是子图

② 子图也必须是个连通图

③ vertex 为极大数量(非最大)

原图 非连通图   子图1   子图2   子图3   都是极大连通子图(连通分量)

3、(1) cut vertices 割点, edges and (2) vertex cut 顶点割 (vertex connectivity 顶点连通性)

(1) 移除该点/边后，图不再连通

(2) 移除某1(些)点后，图不再连通 ⇒ set 集合   割点 ∈ 顶点割

(3) 顶点连通性 ⇒ 得出使图不连通的最少点的个数

ex: 1: {a},{b},{c},{d},{e}: 不行

2: {a,b} 就可以 ⇒ 2 但因为是最少

3: {a,b,c} 也可以 所以为2

若每1点都至少有2个连接，那么顶点连通性一定≥2 (vertex connectivity)

DFS Algorithm

DFS-Main (G, s) 初始化 初始化状态

1 for each vertex u ∈ G.V
2 u.depth ← ∞
3 u.p ← NIL
4 s.depth ← 0
5 DFS (G,s)

DFS (G,u) 排除已经遍历过的结点

1 for each v ← G.Adj[u]
2 if v.depth = ∞
3 v.depth ← u.depth + 1
4 v.p ← u
5 DFS (G,v)

Idea of the algorithm

- Each vertex u has two attributes
  - u.p : the parent of u
  - u.depth : the depth of u
  - Unseen vertex has depth as ∞
- Recursion
  - v is an adjacent vertex of u
  - Base case: adjacent vertex v has depth ≠ ∞, thus v has been seen
  - Recursive case: update the depth of adjacent vertex v to u.depth + 1, and then call DFS recursively at v

BFS Algorithm

层序遍历

BFS(G, s)

1 for each vertex u ∈ G.V − {s}
2 u.d ← ∞ ; u.p ← NIL
3 s.d ← 0 ; s.p ← NIL
4 Q ← create a FIFO queue
5 ENQUEUE(Q, s)
6 while Q ≠ ∅
7 u ← DEQUEUE(Q)
8 for each v ∈ G.Adj[u]
9 if v.d = ∞
10 v.d ← u.d + 1
11 v.p ← u
12 ENQUEUE(Q, v)

Idea of the algorithm

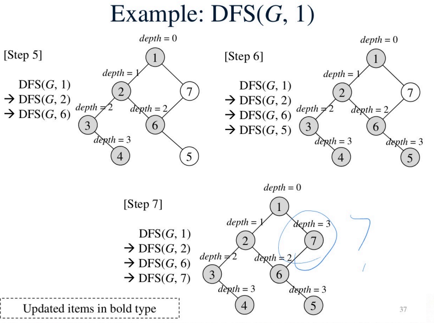
- Each vertex u has two attributes
  - u.p : the parent of u
  - u.d : the "distance" from s to u
  - Unseen vertex has infinite distance
- Manage vertices in a first-in-first-out (FIFO) queue Q
  - Each vertex u (of Q) has been seen and has finite distance u.d
  - Any unseen adjacent vertex v has the distance: u.d + 1
- Property: distances of dequeued entries are in ascending order

Add/remove vertices/edges of a graph

- Let G=(V, E) be a given graph
- When we add/remove vertices/edges of G, how to update the graph model?

Operation	The resulting vertex set	The resulting edge set
Add an edge e	V	E∪{e}
Remove an edge e	V	E−{e}
Add a vertex v	V∪{v}	E
Remove a vertex v	V−{v}	the set of edges of E that do not contain v

Example: DFS(G, 1)



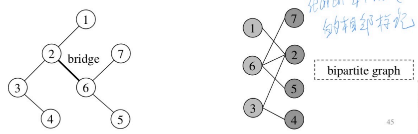
Updated items in bold type

Applications of Graph Traversals

BFS/DFS can be applied to solve the following problems

- (I) Decide whether we can reach a vertex t from a vertex s
- (II) Find an edge that serves as a bridge (connection) in a graph
- (III) Check whether a graph is a bipartite graph
  - Partition the vertices into two parts such that there are no edges among vertices with the same "part"

search 每个node 的相邻结点



Graph II

一、Euler 欧拉 and Hamilton 哈密顿 path/circuit

(一) Euler 想办法不重复地走过所有的边，可重复走过顶点

1、Euler Circuit：从起点出发又回到起点 → 走过所有边，

2、Euler Path：从起点出发不用回到起点 → 可重复顶点，不可重复边

3、寻找 Circuit 与 Path

i 先用度数连接图中2个度数为奇的顶点

ii 再用找circuit的方法

iii 去掉虚线边

abea, edce, abedcea 去掉所有走过的边

adca, aba, bceb ⇒ adcebca

4、定理

(1) 对于 Euler Circuit：每个顶点的度数均为偶数 → 不存在此闭合路

(2) 对于 Euler Path：恰好有2个顶点的度数为奇数 → 奇数度顶点数为0且2

(二) Hamilton 想办法不重复地走过所有点，也没办法重复走过边

1、Hamilton Circuit   2、Path 同上

3、Ore's theorem 不是 Hamilton

→ 当 vertex 数量 ≥3 时，若图中任意一对不相邻顶点 u,v，满足 deg(u)+deg(v) ≥ n，则存在 Hamilton Circuit

Hamilton Path，不存在 2个以上的节点度数为1

⇕

a→b→c→d 是 Hamilton Circuit，但不满足 Ore's theorem

结论：shortest path tree：连接到的结点，通过连接点来更新(不更新最短路径)

二、Shortest Path P18

(1) Dijkstra Algorithm P24

S: {- - - -} 更新后任意放入S(已求最短)

G\S: {- - - -} 取待确定的放入

① 找Heap中 v.d min 的作为u ② 遍历u的邻v，并更新v.d(更近的有的)

③ 将u放入S ④ 将邻 v.d 放回 Heap，并重新更新 Heap

三、Planar graph 平面图

1、定义：每条边之间没有交叉 → 可能画成这样

判断是否为平面图 P33

3、定理 在平面图G中

$r=e-v+2$ 区域数=边数−顶点数+2

(1) if $v\ge3$, then $e\le3v-6$

证：1条边属于2个区域，1个区域至少有3条边

2e是每个区域的边数和的总计算次数，而每个边会被每个区域计算2次，所以所有区域边总计算次数为2e；所有区域的证明思路

⇒ 边的计算总次数：$2e\ge3r$ ⇒ $r\le\frac{2e}{3}$ Σ(每个区域边数)=2e

(2) There exits a vertex in G, so its degree is at most 5

证：反证法 ∀x, degree ≥6，$2e\ge6v$ Σ(每个顶点degree)=2e

(3) If $v\ge3$ and there is no circuit of length 3, then $e\le2v-4$

证：$2e\ge4r$ → 每个区域至少由4条边围成

四、Graph Coloring

1、dual graph: map → graph 对偶图

色数 2、χ(G) = 图着色需的最少颜色数 chromatic number 色数

3、环图：若 n=odd，χ(G)=3；若 n=even，χ(G)=2

4、The 4 color theorem (4色定理)：任意平面图的 χ(G) ≤ 4

→ 只需4种颜色就能完成"相邻区域颜色不同"的着色

→ 针对 planar graph 平面图

Dijkstra's Algorithm

initial all 顶点的 v 和 p

DIJKSTRA(G, s)

1 INIT(G, s)
2 S ← ∅ // set of seen vertices
3 BUILD-HEAP(Q, G.V)
4 while Q ≠ ∅
5 u ← EXTRACT-MIN(Q)
6 S ← S ∪ {u}
7 for each vertex v ∈ G.Adj[u]
8 RELAX(u, v)
9 if v.d is updated
10 DECREASE-KEY(Q,v)

已确定最短path集合；构建heap；找 v.d min 值；取出 Umin；加入S；优化 v.d；更新堆中 v.d；遍历u的相邻节点V

Q就是对v.d进行排序；当需要u的邻居时，可以找最小 v.d

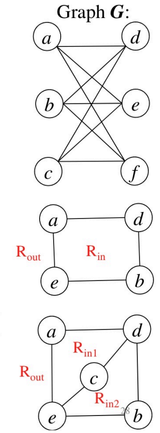
Why this number?

Time complexity

- Lines 1-3: O(|V|) time
- Each vertex leads to exactly 1 iteration
- EXTRACT-MIN operations: |V|
- DECREASE-KEY operations: at most |E|
- Each operation on Q takes O(log |V|) time
- Total time: O((|V|+|E|) log |V|)

Planar graphs

- G is not a planar graph
- Explanation
  - a, e, b, d form a cycle, which is a closed curve in a planar representation. It divides the plane into two regions R_{in} and R_{out}
- Case 1: if c is in R_{in}
  - Since c is adjacent to both d and e, they divide R_{in} into two regions R_{in1} and R_{in2}
  - Case 1.1 if f is in R_{out}, edge (c,f) must have crossing
  - Case 1.2 if f is in R_{in1}, edge (b,f) must have crossing
  - Case 1.3 if f is in R_{in2}, edge (a,f) must have crossing
- Case 2: if c is in R_{out}
  - [Exercise] Try to complete this part by yourself



Ford-Fulkerson Algorithm

Ford-Fulkerson(G, s, t)

```
1  for each edge (u, v) ∈ G.E
2      f(u, v) ← 0
3  while there exists a path p from s to t
       in the residual network G_f
4      c_f(p) ← min{ c_f(u, v) : (u, v) is on p }
5      for each edge (u, v) on p
6          if (u, v) ∈ G.E
7              f(u, v) ← f(u, v) + c_f(p)
8          else
9              f(v, u) ← f(v, u) − c_f(p)
```

Time complexity

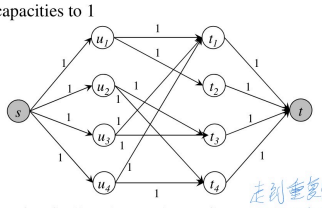
- To find a path p by graph traversal, it takes $O(|V| + |E|) = O(|E|)$ time
- Each outer loop (Lines 3-9) increases the flow value by at least 1
- Let $|f^*|$ be the maximum flow value
- Total time: $O(|E||f^*|)$

Matching: using maximum flow

- Consider a prime pair matching problem
  - Given a set of distinct numbers: {4, 5, 7, 14, 17, 20}
  - Divide this set into pairs so that each pair sum to a prime number
    - < 4, 7>
    - < 5, 14 >
    - < 17, 20>
- Use a bipartite graph to model this problem
  - [*Question*] Which vertices should be placed on the same side?
- Then solve the problem by maximum flow

Matching: using maximum flow

- Model it as a bipartite graph (vertices on same side not adjacent)
  - Link workers (on left-side) to tasks (on right-side)
  - Link s to workers, link tasks to t
  - Fix all capacities to 1



- [*Question*] After finding the maximum flow, how to extract the matching from the residual network?

How to use properties of trees?

- *Question*
  - A student starts a chain email. Each student who receives the email is asked to send it to 3 other students (who have not received that email). Some students may follow the above instruction, but others do not send any email.
  - Suppose that 51 students have received that email but not sent it out. How many students sent out that email?
- *Solution*
  - We model the above process by a **full 3-ary tree**, where internal vertices denote students who sent out that email, and leaves denote students who received that email but not sent it out.
  - Let i be the number of internal vertices
  - The number of leaves $l = 51$
  - We have: $(i + 51) = 3\,i + 1$
  - The answer is: $i = 25$

Delete (T, z)

1. if $z.left$ = NIL
2. Transplant(T, z, $z.right$)
3. elseif $z.right$ = NIL
4. Transplant(T, z, $z.left$)
5. else
6. $y \leftarrow$ Minimum($z.right$)
7. Delete (T, y)
8. replace z by y

Search (x, k)

1. if x = NIL or $k = x.key$
2. return x
3. if $k < x.key$
4. Search($x.left$, k)
5. else
6. Search($x.right$, k)

Insert (T, z)

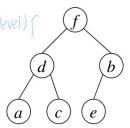
1. $y \leftarrow$ NIL; $x \leftarrow T.root$
2. while $x \neq$ NIL
3. $y \leftarrow x$
4. if $z.key < x.key$
5. $x \leftarrow x.left$
6. else
7. $x \leftarrow x.right$
8. $z.p \leftarrow y$
9. if y = NIL
10. $T.root \leftarrow z$
11. elseif $z.key < y.key$
12. $y.left \leftarrow z$
13. else
14. $y.right \leftarrow z$

- Write an algorithm to count the number of nodes in a tree

```
count t = countNode(n.left);
count t = countNode(n.right);
return count +1;
```

- Write an algorithm to find the level of each node in a tree

```
void nodeLevel (Node n, int level){
(n==null) return;
print ( n.label + level)
nodeLevel (n.left , level+1);
nodeLevel (n.right; level); }
```



Preorder(node x)

1. print $x.label$
2. if $x.nc > 0$
3. for $i \leftarrow 1$ to $x.nc$
4. Preorder($x.c[i]$)

Postorder(node x)

1. if $x.nc > 0$
2. for $i \leftarrow 1$ to $x.nc$
3. Postorder($x.c[i]$)
4. print $x.label$

Inorder(node x)

1. if $x.nc > 0$
2. Inorder($x.c[1]$)
3. print $x.label$
4. if $x.nc > 1$
5. for $i \leftarrow 2$ to $x.nc$
6. Inorder($x.c[i]$)

Prim's algorithm

Prim(G: weighted undirected graph)

1. $n \leftarrow$ number of vertices of G
2. let T be an empty graph
3. insert the minimum-weight edge of G into T
4. for $i \leftarrow 1$ to $n-2$
5. let E' be the set of edges that such edge is incident to a vertex in T and it does not form any cycle in T if added to T
6. insert the minimum-weight edge of E' to T
7. return T

Kruskal's algorithm

Kruskal($G(V,E)$: weighted undirected graph)

1. sort edges of E in increasing order of weight
2. let T be an empty graph
3. while T contains less than $|V|-1$ edges
4. $e_{next} \leftarrow$ the next minimum-weight edge of E
5. if e_{next} does not form a cycle in T
6. insert e_{next} into T
7. return T

Prefix Codes: Huffman Algorithm

Huffman-Code ($a[1..n]$: array of symbols, $freq[1..n]$: frequencies of symbols)

1. $F \leftarrow$ a empty forest
2. for each $i \leftarrow 1$ to n
3. create a tree T_i with vertex $a[i]$; $w(T_i) \leftarrow freq[i]$
4. insert T_i into F
5. while F is not a tree
6. find two trees T and T' of smallest weights from F
7. create a tree T_{new} with a new root, T as left subtree, and T' as right subtree
8. $w(T_{new}) \leftarrow w(T) + w(T')$
9. remove T and T' from F; insert T_{new} into F

Derive the scores of nodes **from the bottom up**