数据结构与算法分析

华中科技大学软件学院

2014年秋

大纲

1. 图的表示
2. 拓扑排序与结点访问
3. 最短路径
4. 最小生成树

课程计划

- 已经学习了
  - 排序算法的重要性
  - 比较交换相邻元素的排序
  - 基于比较的最优排序
  - 排序算法的分析

课程计划

- 已经学习了
  - 排序算法的重要性
  - 比较交换相邻元素的排序
  - 基于比较的最优排序
  - 排序算法的分析
- 即将学习算法设计思想
  - 图的表示
  - 图的结点访问
  - 最小生成树
  - 最短路径

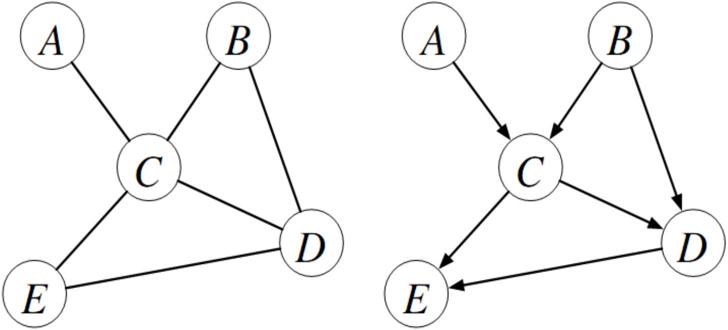
Roadmap

1. 图的表示
2. 拓扑排序与结点访问
3. 最短路径
4. 最小生成树

Graph Theory

- Graph $G = (V, E)$: $V = \{v_i : 1 \leq i \leq n\}$, set of vertices (nodes), E is a subset of $V \times V$ = set of edges (arcs)
- Can use graph to represent any relation: each node is an item, edge between 2 nodes if items are related
  - Directed graph ("digraph"), edges have directions
  - Regular graph ("bi-directional"), no directions
- Each edge can have a weight, say, distances/ costs between cities

Graphs

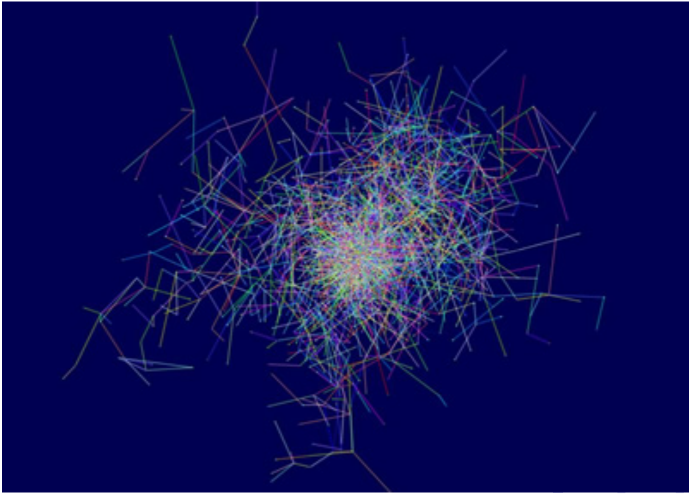


Graph Applications

- What’ s cheapest path from A to B? What's shortest path (in number of edges) from A to B? Where should direct flights be added?
- Other applications:
  - Modeling ground traffic:
  - Where are bottlenecks?
  - Neural networks
  - Markov Chains
  - The Web graph

Erdös Collaboration Graph

A random subgraph of the Erdös' 2nd neighborhood Collaboration graph

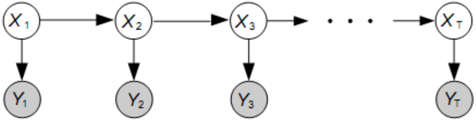


Erdös Numbers

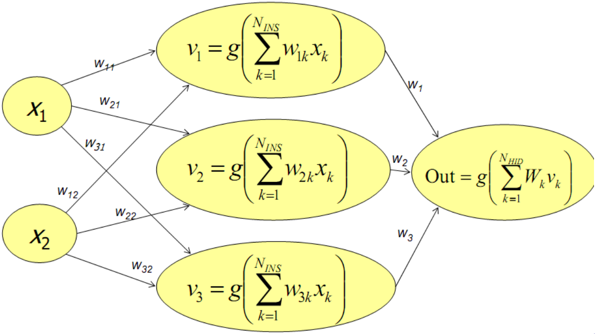
- Thus the median Erdös number is 5; the mean is 4.65, and the standard deviation is 1.21
  - Erdös number 1 - 504 people
  - Erdös number 2 - 6593 people
  - Erdös number 3 - 33605 people
  - Erdös number 4 - 83642 people
  - Erdös number 5 - 87760 people
  - Erdös number 6 - 40014 people
  - Erdös number 7 - 11591 people
  - Erdös number 8 - 3146 people
  - Erdös number 9 - 819 people
  - Erdös number 10 - 244 people
  - Erdös number 11 - 68 people
  - Erdös number 12 - 23 people
  - Erdös number 13 - 5 people

Graph Models

- Hidden Markov models

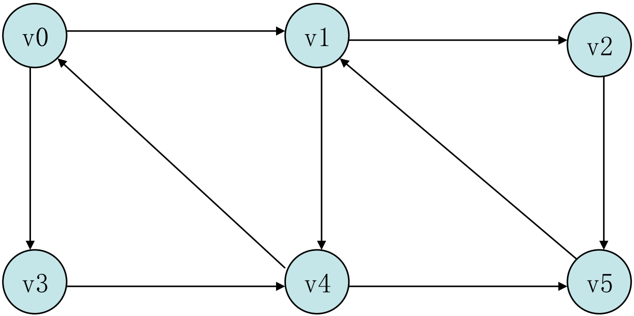


- Artificial neural networks

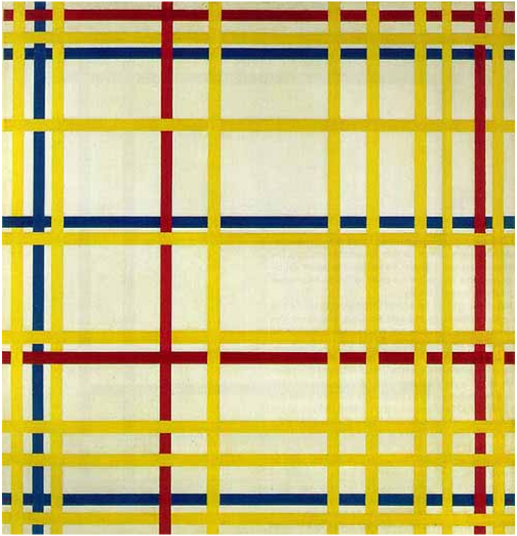


Adjacency Matrix

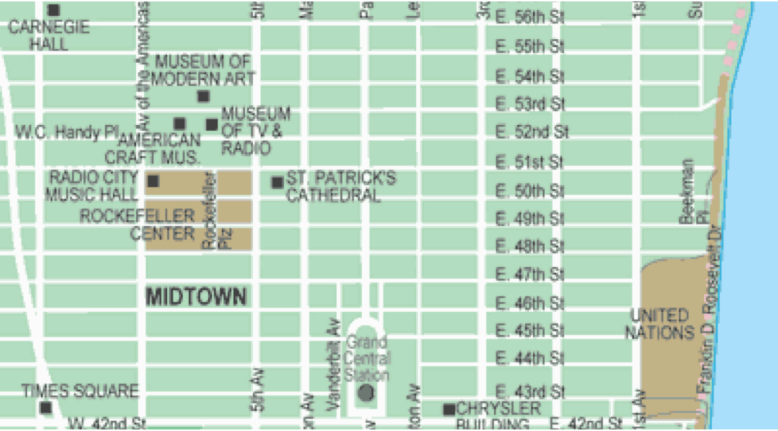
Natural way: adjacency matrix, $|V| \times |V|$ matrix A[v1][v2] = 1 (or put the edge cost) if and only if v_1, v_2 adjacent



Representation of Graphs



Representation of Graphs



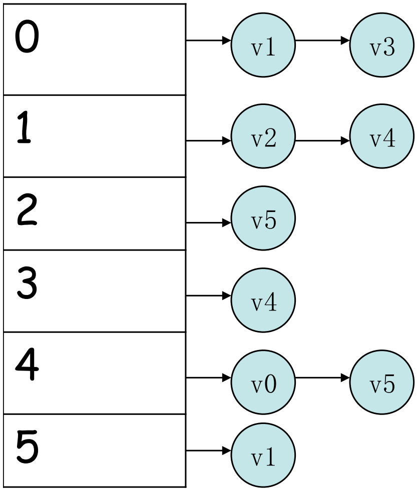
Dense/Sparse Graph Matrices

- Graph is dense if $|E| = \Theta(|V|^2)$, there is an edge between (almost) every node pair
- Create node for every intersection in Manhattan
  - Create edge for every street unit connecting 2 intersections
  - Suppose 3000 4-way intersections, 2 in, 2 out $\rightarrow$ 2*3000 = 6000 edges, 3000^2 = 9,000,000 entries in adjacent matrix

Adjacency lists

- For sparse graphs, better to use another implementation
- For each node: create linked list of adjacent nodes
- For digraph, have 1 entry for each edge
- For regular graph, have 2 entries for each
- Either way: $O(|E| + |V|)$

Adjacency List



Which is better?

Problems	Adj matrices	Adj lists
Adj(x, y)?	O(1)	deg(x) or deg(y)
Find deg(x)	$\lvert V \rvert$	deg(x)
Sparse	$\lvert V \rvert^2$	$\lvert V \rvert + \lvert E \rvert$
Dense	$\lvert V \rvert^2$	$\lvert V \rvert^2$
Add/del edge	O(1)	$\lvert V \rvert$
Traverse graph	$\lvert V \rvert^2$	$\lvert V \rvert + \lvert E \rvert$

Adjacent lists usually considered better

Roadmap

1 图的表示

2 拓扑排序与结点访问

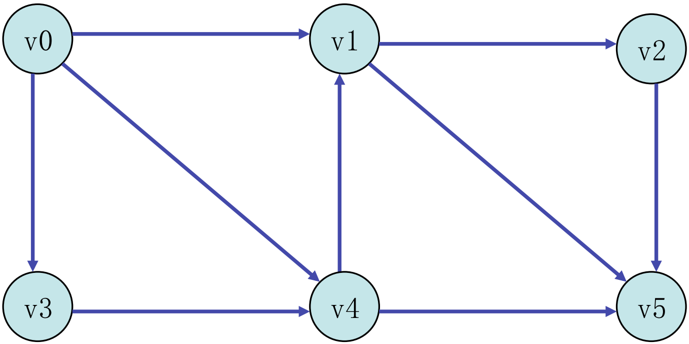
3 最短路径

4 最小生成树

Topological sort

- Given a directed acyclic graph, can ask: in what order can we visit the nodes?
- Top sort: list of nodes s.t. if there is a path from v1 to v2, then v1 appears after v2
    - In general there may be many possible top sorts of graph
    - If the graph has cycles, not well defined
- Which comes first?
    - Eg: each course has prerequisites, find ordering of all classes that is legal
    - Serious application: management of tasks

Example



Top-sort

```
void topSort (Graph G)
{
    int ctr;
    Vertex v, w;

    for (ctr = 0; ctr < NUM_VERTS; ctrl++)
    {
        v = findInDeg0Vert();
        if (v == NULL)
        {
            printf ("A␣cycle␣found\n");
            break;
        }

        topNum[v] = ctr;
        for each w adjacent to v
            indegree[w]--;
    }
}
```

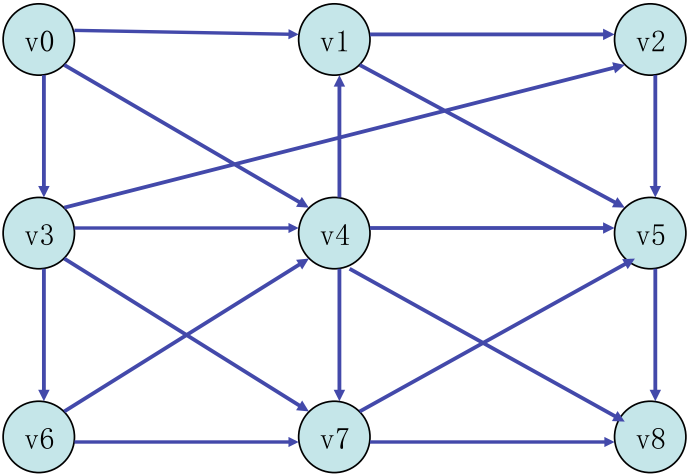
Top Sort Algorithm

- findInDeg0Vert() subroutine
  - Walk through array of vertices
  - O(|V|) each $\rightarrow O(|V|^2)$
  - Okay for dense graphs
- Better: keep indegree-0 nodes in a box
  - Each time decrease indegree of a node, if it's now 0, put it in the box
  - Box is a stack or queue

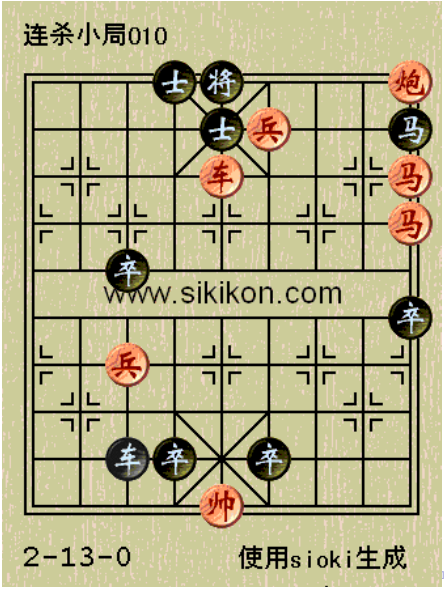
Top-sort with Queue

```
 void topsort()
{
    Queue q;
    int ctr = 0;
    Vertex v,w;

    q = createQueue (NumVertex); MakeEmpty (Q);
    for each vert v
        if (indegree[v] == 0)
            enqueue (v, q);

    while (!Isempty (q))
    {
        v = dequeue (q);
        topNum[v] = ++ctr;
        for each w adj to v
            if (--indegree[w] == 0)
                enqueue (w, q);
    }

    if (ctr != NUM_VERTS)
        printf ("A␣cycle␣is␣found\n");
    disposeQueue (q);
}
```

Exercise



Visiting Graph Nodes



Search Algorithms

- BFS: breadth first search = level-order traversal
- DFS: depth first search = pre-order traversal

DFS

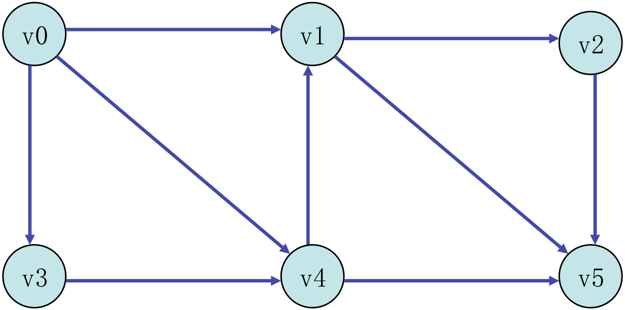
```
void dfs(vert v)
{
    visited[v] = TRUE;

    for each w adjacent to v
        if (!visited[w])
            dfs(w);
}
```

BFS

```
void bfs (vert v)
{
    queue Q;
    vert w;

    makeEmptyQueue (Q) ;
    visited[v] = TRUE;
    enqueue (Q, v) ;

    while (!isEmpty (Q ))
    {
        v = dequeue (Q);
        for all w adjacent to v
            if (!visited [w])
            {
                visited (w) = TRUE;
                enqueue (Q, w);
            }
    }

    disposeQueue (Q);
}
```

Starting from V0

Try DFS and BFS



Roadmap

1. 图的表示
2. 拓扑排序与结点访问
3. 最短路径
4. 最小生成树

Shortest-path Problems

- Single-source shortest-path problem: given a weighted graph G, and one vertex s find shortest weighted paths from s to all nodes
- Start with unweighted version
- Do Breadth-First-Search

Shortest Path

Theorem

A sub-path of a shortest path is a shortest path

- Triangle inequality
- $d(s,t) = \min_{(v,t)\in E}(d(s,v) + w(v,t))$
- Bellman Ford Algorithm works for negative weights, O(|E||V|)

Stortest Path

```
for v != s
    initialize d[v][0] = INFTY;

for all i
    d[t][i]=0;

for i=1 to n-1
    for each v != s
        d[v][i] = min ((v,x) in E (len(v,x)
         + d[x][i-1]))

for each v
    output d[v][n-1].
```

Unweighted Algorithm Code

```
void unweighted (Vertex s)
{
    int currDist;
    Vertex v, w;
    dist[s] = 0;

    for (currDist = 0; currDist < NUM_VERTS; currDist++)
        for each vert v
            if (!known[v] && dist[v] == currDist)
            {
                known[v] = TRUE;
                for each w adjacent to v
                    if (dist[w] == INFTY)
                    {
                        dist[w] = currDist+1;
                        path[w] = v;
                    }
             }
}
```

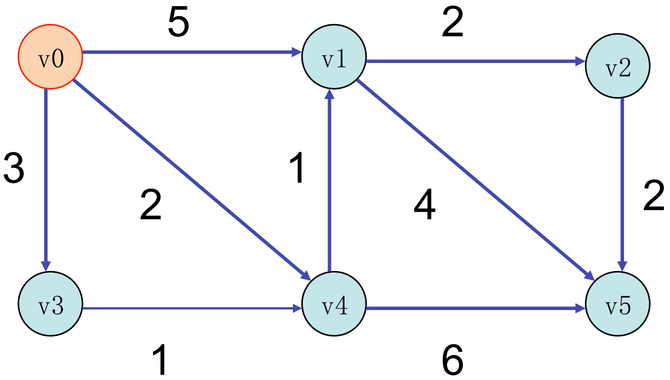
With the help of Queue

```
void unweighted (Vertex s)
{
    Queue q;
    Vwrtex v, w;

    createQueue (q); makeEmpty (q);
    enqueue (s, q);

    while (!isEmpty (q))
    {
        v = dequeue (q);
        known[v] = TRUE;

        for each w adjacent to v
            if (dist[w] == INF)
            {
                dist[w] = dist[v] + 1;
                path[w] = v;
                enqueue (w);
            }
    }
    disposeQueue (q);
}
```

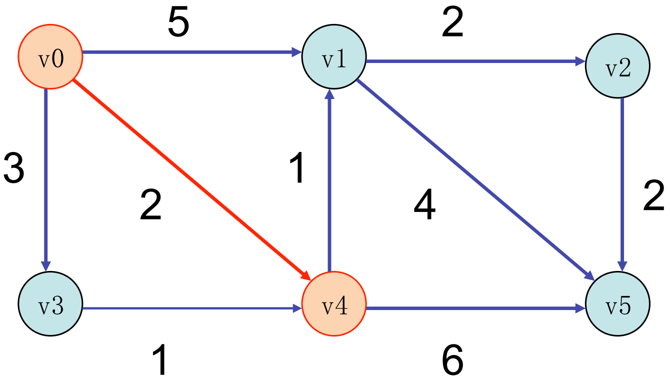
Dijkstra's Algorithm

- Weighted shortest paths for positive weights
- Complexity $O(|E|\log|V|)$
- Greedy algorithm: always choose shortest edge
- Idea: at each iteration, select unknown node with lowest distance
- 3 pieces of information for each node
  - Known, boolean, whether the shortest distance is determined
  - d_v shortest distance so far
  - p_v previous node

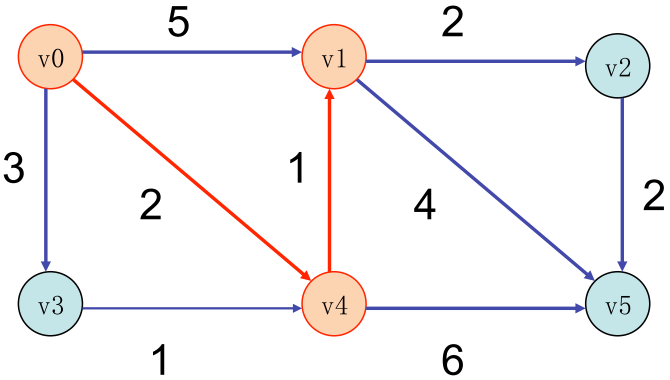
Example



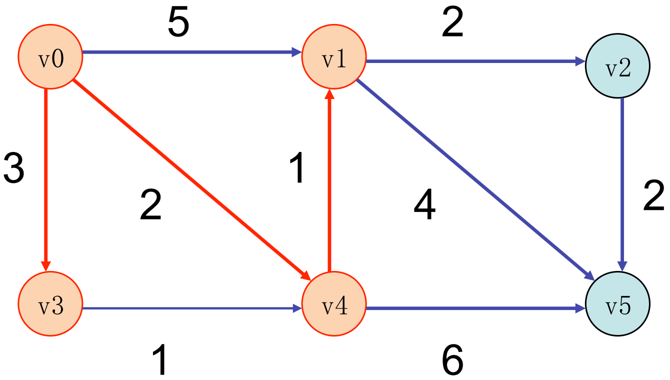
Example



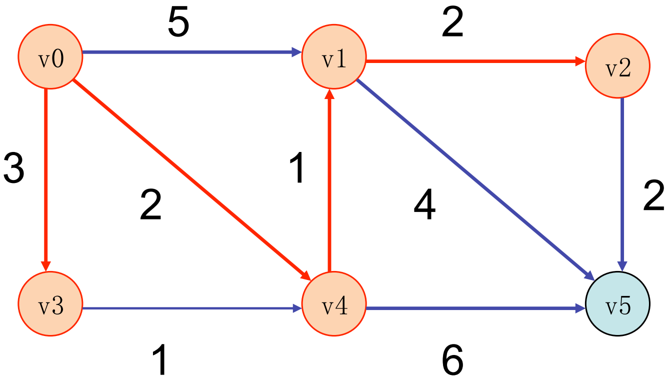
Example



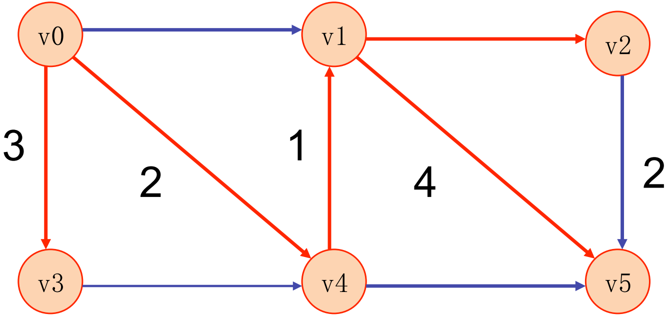
Example



Example



Example



Dijkstra Code

```
typedef struct TableEntry
{
    List header;
    boolean known;
    DistType dist;
    Vertex path;
}

void initTable (Veterx s, Graph G, Table T)
{
    int i;

    readGraph (G, T);
    for (i = 0; i < NUM_VERTS; i++)
    {
        T[i].known = FALSE;
        T[i].dist = INF;
        T[i].path = NULL;
    }
    T[s].dist = 0;
}
```

Dijkstra Code

```
void printPath (Vertex v, Table T)
{
    Vertex v, w;

    if (T[v].path != NotAVertex)
    {
        printPath (T[v].path, T);
        printf ("␣to␣");
    }

    printf ("%d", v);
}
```

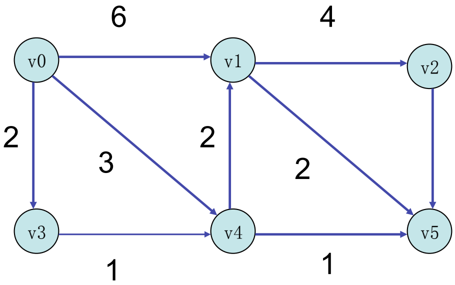
Dijkstra Code

```
void dijkstra (Vertex s, Table T)
{
    Vertex v, w;
    T[s].dist = 0;

    while (TRUE)
    {
        v = smallest-dist unknown vertex;
        if (v == NotAVertex) break;
        T[v].known = TRUE;

        for each w adjacent to v
            if (!T[w].known && T[v].dist + Cvw < T[w].dist)
            {
                T[w].dist = T[v].dist + Cvw;
                T[w].path = v;
            }
      }
}
```

Dijkstra's Complexity

- How to find smallest-distance unknown vertex?
- If do linear search, time $\Theta(|V|)$ for each $\Theta(|E| + |V|^2) = \Theta(|V|^2)$ total
- Fine for dense graphs, bad for sparse
- Better: put unknown nodes in minQueue Select v. known == true $\rightarrow$ delMin, $\log|V|$ each
- What about changing dist[w]?
- Becomes a decreaseKey operation
- Assuming have way of find element, or store location, $\log|V|$ each, Total: $\Theta(|E|\log|V| + |V|\log|V|) = \Theta(|E|\log|V|)$

Exercise



Trivial case: dist (V0, V0) = 0
Recursive step: dist (V0, Vi) = min (dist (V0, Vi), dist (V0, Vj) + Cji), for all edges j->i

Initial Values

	Known	path	dist	0
V0	True	Null	0	0
V1	False	Null		INF
V2	False	Null		INF
V3	False	Null		INF
V4	False	Null		INF
V5	False	Null		INF

Round 1

	Known	path	dist	0	1
V0	True	Null	0	0	0
V1	False	V0		INF	6
V2	False	Null		INF	INF
V3	False	V0	2	INF	2
V4	False	V0		INF	3
V5	False	Null		INF	INF

Round 1

	Known	path	dist	0	1
V0	True	Null	0	0	0
V1	False	V0		INF	6
V2	False	Null		INF	INF
V3	True	V0	2	INF	2
V4	False	V0		INF	3
V5	False	Null		INF	INF

Round 2

	Known	path	dist	0	1	2
V0	True	Null	0	0	0	0
V1	False	V0		INF	6	6
V2	False	Null		INF	INF	INF
V3	True	V0	2	INF	2	2
V4	False	V0	3	INF	3	3
V5	False	Null		INF	INF	INF

Round 2

	Known	path	dist	0	1	2
V0	True	Null	0	0	0	0
V1	False	V0		INF	6	6
V2	False	Null		INF	INF	INF
V3	True	V0	2	INF	2	2
V4	True	V0	3	INF	3	3
V5	False	Null		INF	INF	INF

Round 3

	Known	path	dist	0	1	2	3
V0	True	Null	0	0	0	0	0
V1	False	V0		INF	6	6	5
V2	False	Null		INF	INF	INF	INF
V3	True	V0	2	INF	2	2	2
V4	True	V0	3	INF	3	3	3
V5	False	V4	4	INF	INF	INF	4

Round 3

	Known	path	dist	0	1	2	3
V0	True	Null	0	0	0	0	0
V1	False	V0		INF	6	6	5
V2	False	Null		INF	INF	INF	INF
V3	True	V0	2	INF	2	2	2
V4	True	V0	3	INF	3	3	3
V5	True	V4	4	INF	INF	INF	4

Round 4

	Known	path	dist	0	1	2	3	4
V0	True	Null	0	0	0	0	0	0
V1	False	V4	5	INF	6	6	5	5
V2	True	Null		INF	INF	INF	INF	INF
V3	True	V0	2	INF	2	2	2	2
V4	True	V0	3	INF	3	3	3	3
V5	True	V4	4	INF	INF	INF	4	4

Round 4

	Known	path	dist	0	1	2	3	4
V0	True	Null	0	0	0	0	0	0
V1	True	V4	5	INF	6	6	5	5
V2	True	Null		INF	INF	INF	INF	INF
V3	True	V0	2	INF	2	2	2	2
V4	True	V0	3	INF	3	3	3	3
V5	True	V4	4	INF	INF	INF	4	4

Final Round

	Known	path	dist	0	1	2	3	4	5
V0	True	Null	0	0	0	0	0	0	0
V1	True	V4	5	INF	6	6	5	5	5
V2	False	V1	9	INF	INF	INF	INF	INF	9
V3	True	V0	2	INF	2	2	2	2	2
V4	True	V0	3	INF	3	3	3	3	3
V5	True	V4	4	INF	INF	INF	4	4	4

Finally

	Known	path	dist	0	1	2	3	4	5
V0	True	Null	0	0	0	0	0	0	0
V1	True	V4	5	INF	6	6	5	5	5
V2	True	V1	9	INF	INF	INF	INF	INF	9
V3	True	V0	2	INF	2	2	2	2	2
V4	True	V0	3	INF	3	3	3	3	3
V5	True	V4	4	INF	INF	INF	4	4	4

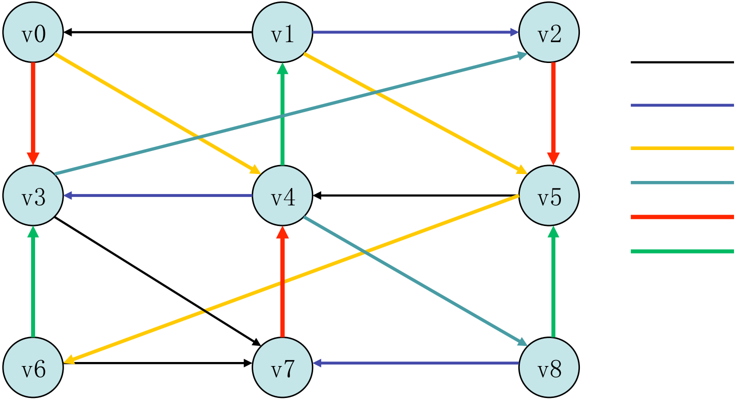
Weighted Negative

```
void weightedNegative (Vertex s, Table T)
{
    Queue = q;
    Vertex v, w;
    q = createQueue (NUM_VERTS); makeEmpty (q);
    enqueue (s q);

    while (!isEmpty (q))
    {
        v = dequeue (q);
        if have seen v |V|+1 times, break;
        for each w adjacent to v
            if (T[v].dist + Cvw < T[w].dist)
            {
                T[w].dist = T[v].dist + cvw;
                T[w].path = v;
                if (! contains(q, w))
                    enqueue (w, q);
            }
    }

    disposeQueue (q);
}
```

Acyclic Graphs

- Dijkstra easier if graph is acyclic
- Change order in which vertices are known
- Select vertices in topology order, in one pass
- When v selected, its distance dv can't be lowered
- By topology order rule, it has no unknown nodes point to it
- Constant time selection $\rightarrow$ no priority Q $\rightarrow \Theta(|E| + |V|)$

Example



Roadmap

1 图的表示

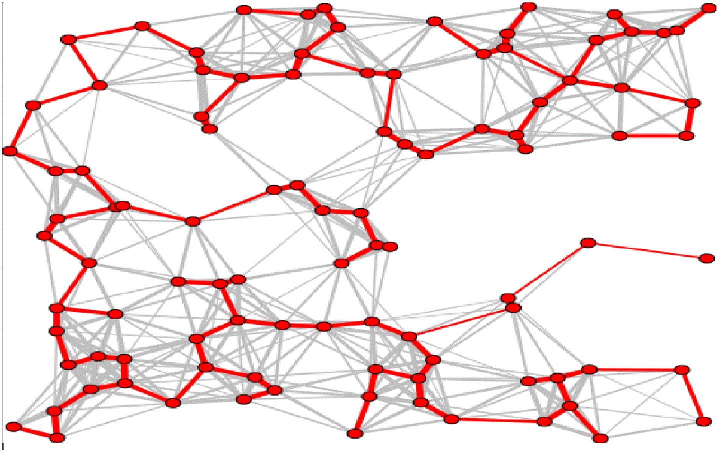
2 拓扑排序与结点访问

3 最短路径

4 最小生成树

Minimum Spanning Tree

To connect all nodes with the lowest cost



Greedy Algorithm

- Idea: given a (weighted) graph, produce tree containing all nodes whose sum of weights is smallest
- Eg: given building with TVs in different rooms, find way to connect all TVs using minimal total length of cable
- How many edges in MST? $|V| - 1$
- Greedy algorithm: generate a spanning tree edge by edge, always adding min-cost edge (that avoids cycle)

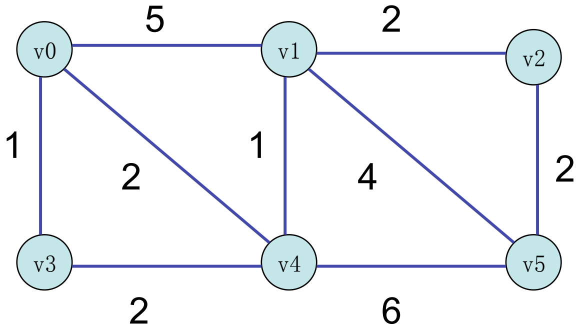
2 Main Algorithms

- Prim’s algorithm: create single MST
- Kruskal’s algorithm: create a forest of MSTs that connect

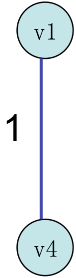
Prim’s Algorithm

- Same as Dijkstra’s algorithm, except dv is the weight of the shortest edge connecting v to a known vertex
- Update rule: for each unknown vertex w adjacent to v, $d_w = \min(d_w, c_{wv})$
- Time: $O(|V|^2)$ without heaps, O(|E|log|V|) with heaps

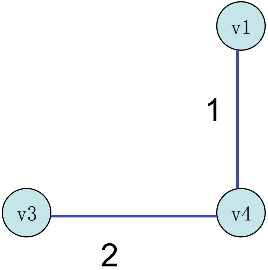
Prim’s Algorithm



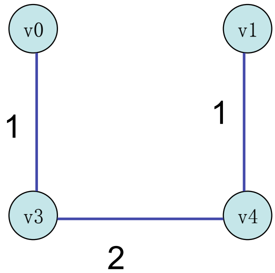
Prim’s Algorithm



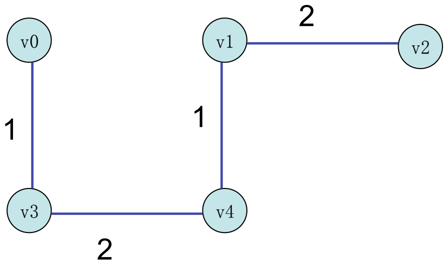
Prim’s Algorithm



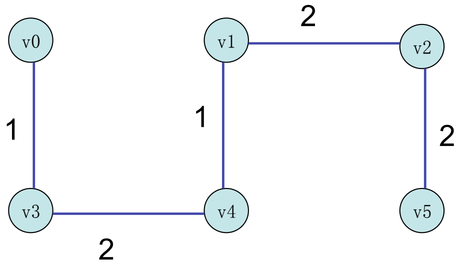
Prim’s Algorithm



Prim’s Algorithm



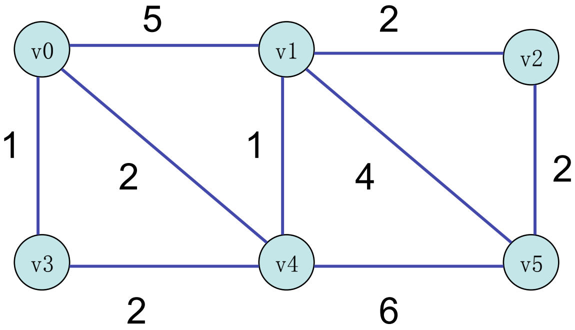
Prim’s Algorithm



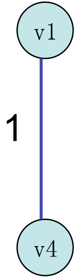
Kruskal’s Algorithm

- Choose edge with smallest weight. If it doesn’t cause a cycle, add to graph, iterate until have added $|V|-1$ edges
- How to select min edge? Could sort, but that’s $|E|\log|E|$. Better: build edge priority Q in $|E|$ time, then extract min edges
- How to tell if will causes a cycle? Put all connected edges in a set together, both ends of edge are in set $\rightarrow$ will cause cycle
- Time: $O(|E|\log|E|) = O(|E|\log|V|)$. In practice, Kruskal’s faster than Prim’s

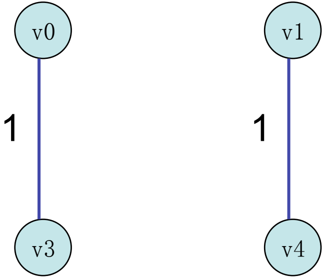
Kruskal’s Algorithm



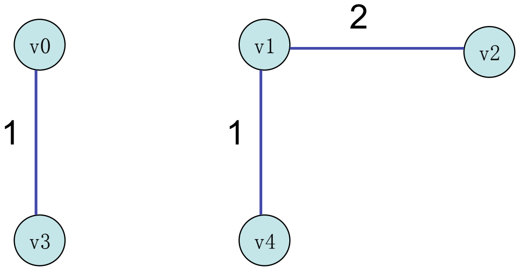
Kruskal’s Algorithm



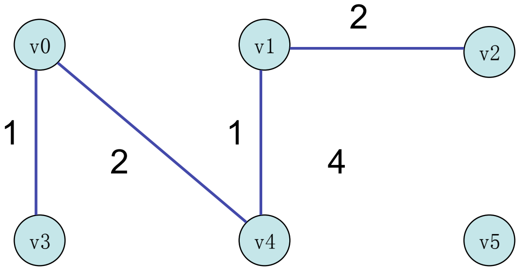
Kruskal’s Algorithm



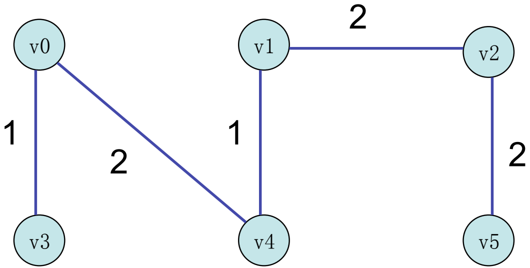
Kruskal’s Algorithm



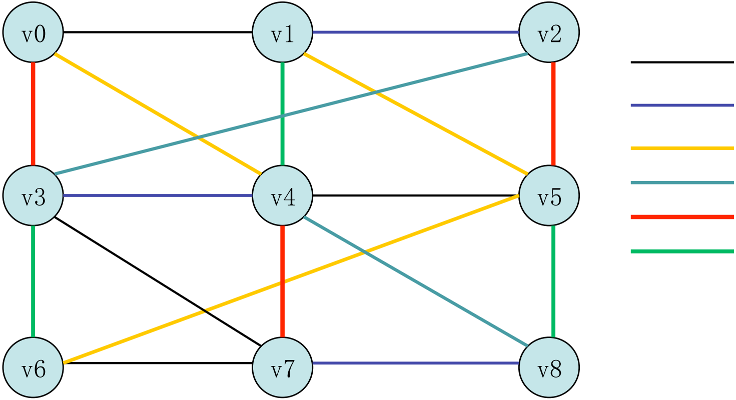
Kruskal's Algorithm



Kruskal’s Algorithm



Exercise



小结

- 图可以用邻接矩阵或邻接链表表示
- 图的结点可以通过深度优先或广度优先的算法实现
- 拓扑排序将有向无环图中的偏序关系转换为线性关系
- 贪婪算法：Dijkstra算法计算最短路径，Prim和Kruskal算法寻找最小生成树