目录

Weighted Graph Algorithms

(Chapter 20.1 21.1 21.2 21.3 Sedgewick)

Graphs so far have considered have edges that simply link two vertices.

Some applications have a **cost** or **weight** on edges. We'll use both terms interchangeably.

Weights:

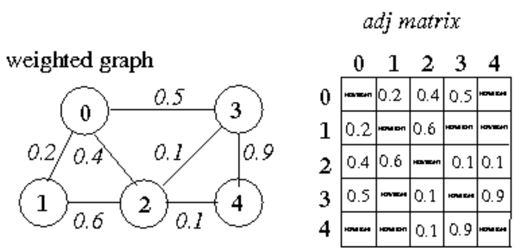
- sometimes there is a geometric interpretation of the term *weight*:
    - an edge with low weight is a "short" edge
    - an edge with high weight is a "long" edge
- other times it is simply a value
    - we'll use real numbers as edge weights
    - but, for convenience, we do not allow negative weights

An ADT for Weighted Graphs

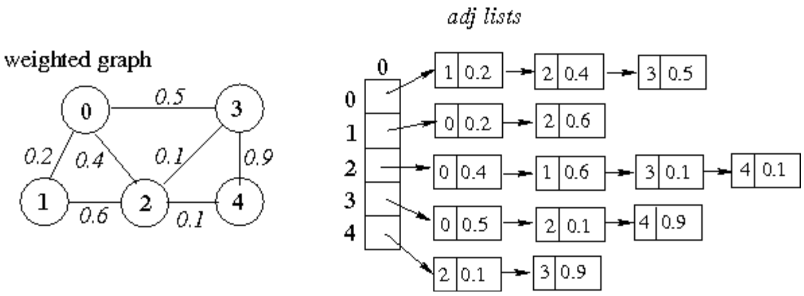
Weights can easily be added to:

- an adjacency matrix representation
    - change *bool* to *float*
    - need a special float constant to indicate *NOWEIGHT*, i.e. there is no edge
        - can’t use 0 anymore. That might be a valid weight.
        - most problems do not allow negative weights, so let NOWEIGHT = -1
- adjacency list representation
    - add float to each node in the list

Adjacency Matrix with weights:



Adjacency List with weights :



In both cases the interface of the ADT is defined as:

切换行号显示

```
// WeGraph.h: an interface for a weighted graph ADT
#include <math.h>

typedef float Weight;              // define a WEIGHT
#define NOWEIGHT  -1.0
#define MAXWEIGHT INFINITY

typedef int Vertex;                // define a VERTEX
#define UNVISITED -1
#define VISITED 1

typedef struct {
  Vertex v;
  Vertex w;
  Weight x;
} Edge;

typedef struct graphRep *Graph;    // define a GRAPH

Graph newGraph(int);
void  freeGraph(Graph);
void  showGraph(Graph);

void insertEdge(Edge, Graph);
void removeEdge(Edge, Graph);
void showEdge(Edge);
int  isEdge(Edge, Graph);
Edge newEdge(Vertex, Vertex, Weight);
Edge getEdge(Vertex, Vertex, Graph);
int  cmpEdge(Edge, Edge);

Weight getWeight(Graph, Vertex, Vertex);
```

程序内部尽量少出现数字
通过定义名称可读性更高

The implementation of the interface, which uses an adjacency matrix, can be found here

Weighted-Graph Problems

There are a huge number of applications of this data structure. We'll consider just two:

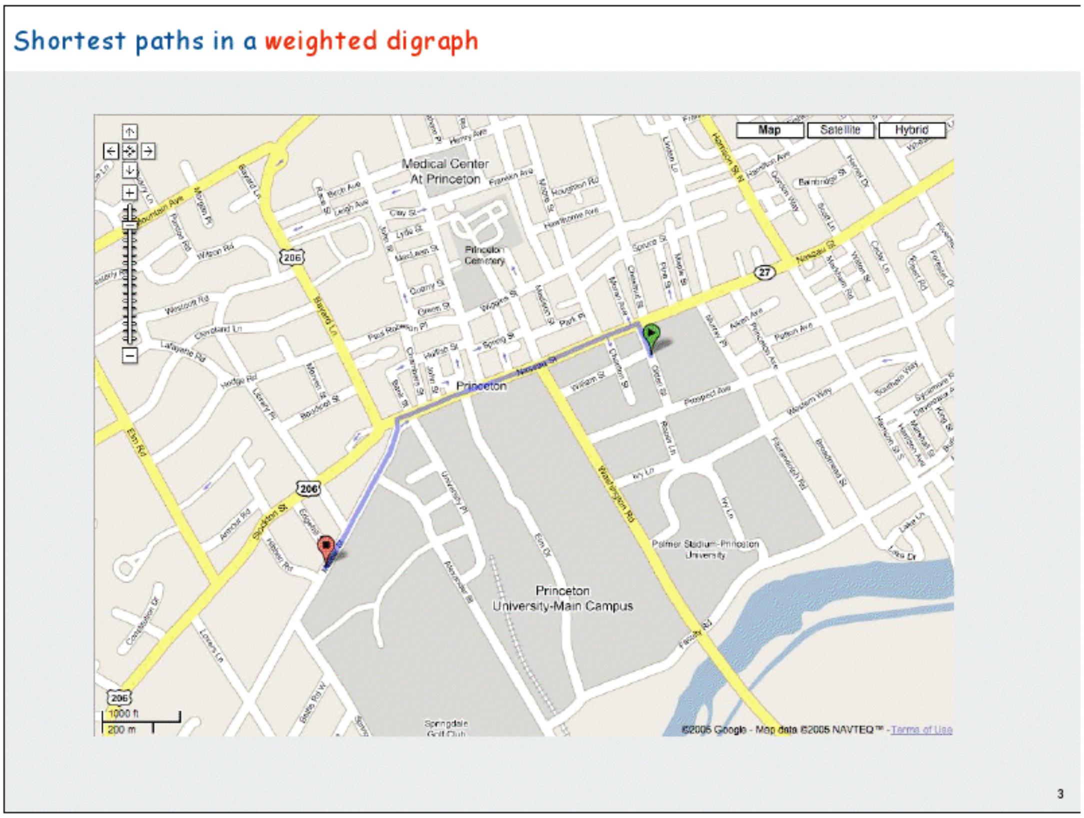
最短路径问题
最小生成树问题

- shortest-path problems:
  - *what is the least-cost path from* A *to* B*?*
- minimum spanning tree problems:
  - *what is the least-cost way to connect all vertices in a graph*
  - i.e. *what set of edges covers the graph with least-cost?*

Shortest Path Tree

Finding the shortest path in a graph is a very common problem:

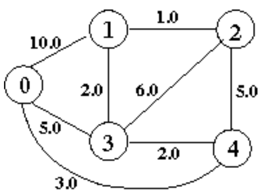
- Google maps:



- Robot navigation.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.

Example: Shortest Path

Consider the following weighted graph:



Assume we want to know a shortest path from vertex 0 to vertex 1. Which of the following is it?

- *0→1*
- *0→3→1*
- *0→3→2→1*
- *0→3→4→2→1*
- *0→4→2→1*
- *0→4→3→1*
- *0→4→3→2→1*
- *0→3→4→2→3→1* (...contains a cycle!)
- and so on

We also want to know a shortest path from vertex 0 to vertex 2. Which is it?

- *0→1→2*
- *0→1→3→2*
- *0→1→3→4→2*
- *0→3→1→2*

- *0→3→2*
- *0→3→4→2*
- *0→4→2*
- *0→4→3→1→2*
- *0→4→3→2*
- *0→1→3→0→4→2* (... another cycle)
- and so on

We also want the shortest path:

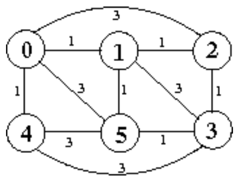
- from *0* to *3*
- from *0* to *4*

and also:

- from *1* to every other vertex
- from *2* to every other vertex
- from *3* to every other vertex
- from *4* to every other vertex

Tree representation of a shortest path

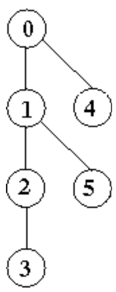
Consider the graph:



Assume that we start at vertex *0*. By observation, we can see that the shortest path:

- to vertex *1* is:
  - *0→1*
- to vertex *4* is:
  - *0→4*
- to vertex *2* is:
  - *0→1→2*
- to vertex *5* is:
  - *0→1→5*
- to vertex *3* is (here we have a choice, we select arbitrarily):
  - *0→1→2→3*

We can draw all these paths as a single tree, called the *Shortest Path Tree* (SPT). Here this would look like:



It is a tree because every vertex has a single parent

- **Note, we assume that we want a shortest path, not all shortest paths**

Each edge in the SPT corresponds to an edge in the original graph (of course).

You can represent a tree as an array: 每一个顶点只有一个parent

- we'll call the array **parent[ ]** 索引值对应的parent，与上周求最短路径的方法一致
  - the index corresponds to the vertex label
  - the value is its parent vertex label root没有parent
- note, the root of the tree, the start vertex *0*, has no parent of course (represented by 'NONE')

For example, the SPT above could be represented as **parent = [NONE, 0, 1, 2, 0, 1]**

The array **parent[ ]** tells us what the paths are, but not the costs of the paths. To store the path costs:

- we use an array **pacost[ ]** pacost:path cost
  - the index corresponds to the vertex label
  - the value is the sum of the weights on the edges from *start* to this vertex
- note, the cost of the path from the start vertex *0* to itself is 0.

For example, for the graph above, we have **pacost[0, 1, 2, 3, 1, 2]** 0到其他顶点的最短距离

There are actually different 'flavours' of shortest-path algorithms:

1. **single-source, single-target** (from *s* to *t*)
2. **single-source** (from *s* to all other vertices in the graph)
3. **all-pairs** (for all pairs of edges)

Here we are interested in a *single-source* shortest-path algorithm, so to all other vertices in the graph.

Basically, the **SPT** is:

```
Given:
 * a weighted graph G and   从s到达所有顶点的最短路径
 * a source vertex s
the Shortest Path Tree represent the shortest paths from s to all other vertices
```

Edsger Dijkstra is credited with discovering the algorithm to do this.

- Remember him from the first lecture:
  E. W. Dijkstra [1930-2002]

- ... remember ... *the father of structured programming*
  - *Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.*

Dijkstra's formulation

We use the two arrays **parent[ ]** and **pacost[ ]** to store the SPT.

- the parent array represents the tree structure
- the cost array the lengths of the paths to each vertex

单源最短路径

The basic algorithm is:

```
set pacost[v] = INFINITY for all vertices 首先，设置所有pacost为无穷大
set parent[v] = NONE for all vertices          设置所有parent为NONE
set all vertices to unvisited                  设置所有顶点未访问过
set pacost[start] = 0                     然后，while循环遍历
while there are still unvisited vertices
   let s be the unvisited vertex that has the smallest cost in pacost[]
   set s to visited
   for each vertex t adjacent to s
      if ((t is unvisited) && (pacost[t] > pacost[s] + cost_of_edge(s, t)))
         set pacost[t] = pacost[s] + cost_of_edge(s, t)
         set parent[t] to s
```

Note:

- in each *while-loop* iteration, one vertex is visited

- when a vertex is visited, it is added to the SPT
- the 'closest/smallest' unvisited vertex is always selected to be next
  - *remember, we want the minimum path cost to a vertex* s的cost+edge的权值
    - the path cost is *pacost[s]* plus the cost of the edge from *s* to the vertex
    - where *pacost[s]* is the path cost to an already visited vertex *s*

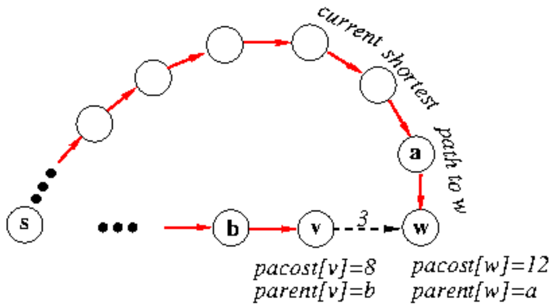
Dijkstra's algorithm demonstrated (3 mins 33 secs)

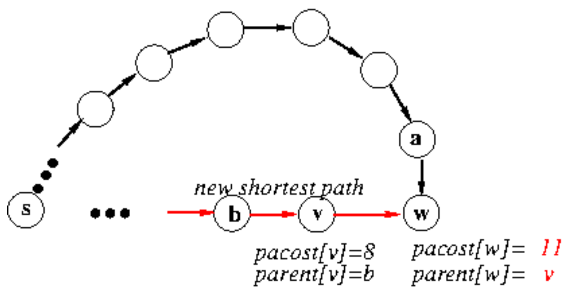
Edge relaxation

This means simply that if a vertex is re-visited and a lower cost path is found, the cost of the vertex is updated.

For example: an example of a relaxation along edge *v->w*.

- Assume we are in vertex **v**, and **w** is adjacent
- *pacost[v]* is length of current shortest path from *s→...→v*
- *pacost[w]* is length of current shortest path from *s→...→w*





In essence, it could be coded as:

切换行号显示

```
if (pacost[w] > pacost[v] + getWeight(g, v, w) {
    pacost[w] = pacost[v] + getWeight(g, v, w);
    parent[w] = v;
}
```

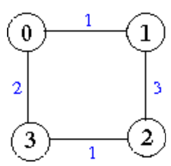
The edge relaxing above is *non-trivial* (the finite path cost is reduced) 有限具体减少，non-trival

- a *trivial* relaxation happens when the initial infinite path cost is assigned a value

trivial relaxation：初始化的无穷大被分配一个值

An example of Dijkstra relaxation

Consider the graph:



Apply Dijkstra's algorithm

- initially
  - set *pacost[0, INFINITY, INFINITY, INFINITY]*
  - set *parent[NONE, NONE, NONE, NONE]*
- step 1
  - vertex *s = 0* is minimal
    - set *0* to visited
    - vertex *t = 1* is adjacent but unvisited

- **trivial relax** *pacost[1]*: INFINITY => 1.00 trivial：无意义的
- *parent[1] = 0*
- vertex *t = 3* is adjacent but unvisited
  - **trivial relax** *pacost[3]*: INFINITY => 2.00
  - *parent[3] = 0*

- step 2
  - vertex *s = 1* is minimal
    - set *1* to visited
    - vertex *t = 0* is adjacent and visited
    - vertex *t = 2* is adjacent but unvisited
      - **trivial relax** *pacost[2]*: INFINITY => 4.00
      - *parent[2] = 1*
- step 3
  - vertex *s = 3* is minimal
    - set *3* to visited
      - vertices 0 is adjacent and visited non-trival：有意义的
      - vertices 2 is adjacent but unvisited
      - **non-trivial relax** *pacost[2]*: **4.00 => 3.00**
      - *parent[2] = 3*
- step 4
  - vertex *s = 2* is minimal
    - set *2* to visited
      - vertices 1 and 3 are adjacent and visited

The final contents of the 2 arrays will be:

- *parent[NONE, 0, 3, 0]*
- *pacost[0.00, 1.00, 3.00, 2.00]*

I hope that you can easily draw the SPT represented by parent[ ].

Dijkstra Algorithm (and Prim)

An implementation of Dijkstra's algorithm is the following:

切换行号显示

起点的顶点 算法类型：d，p

NONE = -1

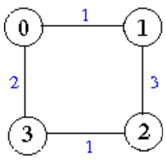
```
void DijkstraPrim(Graph g, int nV, int nE, Vertex src, char alg) {
// the last parameter arg is set by main, and is:
// 'd' for Dijkstra or
// 'p' for Prim
                        主要就是分配memory的一个函数
   int *visited = mallocArray(nV);   // initialised to UNVISITED
   int *parent = mallocArray(nV);    // initialised to UNVISITED
   float *pacost = mallocFArray(nV); // floats: initialised to INFINITY
                                  默认都是正无穷
   pacost[src] = 0.0;  本身的cost为0.0
   for (int step = 1; step <= nV; step++) { 进行nV次的循环遍历
      Vertex minw = NONE;  找到最小的w值（未访问过的）    接下来访问这个minw顶点
      for (Vertex w = 0; w < nV; w++) {        // find minimum cost vertex
         if ((visited[w] == UNVISITED) &&
             (minw == NONE || pacost[w] < pacost[minw])) { 第一次遍历就是
            minw = w;           如果minw=NONE，则遇到w就为其赋值    找到的src
         }                      否则需要w的pacost更小才可以
      }                                          首先，关注的点是未访问的，否则不会执行
                                                 操作，遇到第一个未访问的顶点，这时候
                                                 minw == NONE 肯定成立，因此 minw 赋值了
      visited[minw] = VISITED;  找到顶点后标记已访问  一个数据，然后继续遍历，比较所有未访问
                                                 顶点的最小 pacost 对应的 w 就就是 minw

      for (Vertex w = 0; w < nV; w++) {遍历邻接点找到最小的权值
         Weight minCost = getWeight(g, minw, w);// if minw == w, minCost = NOWEIGHT
         // minCost is cost of the minimum crossing edge
         if (minCost != NOWEIGHT) {NOWEIGHT表示不相连
            if (alg == 'd') {                 // if DIJKSTRA ...
               minCost = minCost + pacost[minw];// add in the path cost
            }                                前面的值+当前的pacost
            if ((visited[w] != VISITED) &&
                (minCost < pacost[w])) {        pacost记录了每一个顶点到达目标顶
                   pacost[w] = minCost;         点的权值之和
                   parent[w] = minw;
            }      如果w没有被访问，并且minCost还更小，因此需要替换值
         }         pacost替换为minCost
      }            parent则替换为当前的minw
   }
   showArray("visited", visited, nV);
   showArray("parent", parent, nV);
```

```
    showFArray("pacost", pacost, nV);
    free(visited);
    free(parent);
    free(pacost);
    return;
}
```

- The weighted graph ADT (*weGraph.c*) must of course be compiled with the main program that calls this function.
- The output generated by this function is the following:

```
visited: {1, 1, 1, 1}
parent: {-1, 0, 3, 0}
pacost: {0.00, 1.00, 3.00, 2.00}
```

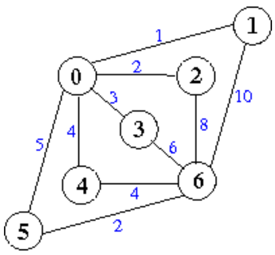
Here is the original graph again:



Make sure you understand what *parent[ ]* and *pacost[ ]* are telling you.

A more relaxing example of Dijkstra

Consider the more complicated example:



Dijkstra's algorithm visits vertex '6' 5 times, each time it does a relax:

- step = 1, minw = 0, pacost[0] = 0.00 第一步都是trivial relax

```
        trivial relax: pacost[1] = inf  ==> 1.00
        trivial relax: pacost[2] = inf  ==> 2.00
        trivial relax: pacost[3] = inf  ==> 3.00
        trivial relax: pacost[4] = inf  ==> 4.00
        trivial relax: pacost[5] = inf  ==> 5.00
```

- step = 2, minw = 1, pacost[1] = 1.00 只有具体数字的减少是 non-trival relaxation

```
        relax: pacost[6] = inf  ==> 11.00
```

- step = 3, minw = 2, pacost[2] = 2.00

```
        non-trivial relax: pacost[6] = 11.00  ==> 10.00
```

- step = 4, minw = 3, pacost[3] = 3.00

```
        non-trivial relax: pacost[6] = 10.00  ==> 9.00
```

- step = 5, minw = 4, pacost[4] = 4.00

```
        non-trivial relax: pacost[6] = 9.00  ==> 8.00
```

- step = 6, minw = 5, pacost[5] = 5.00

```
        non-trivial relax: pacost[6] = 8.00  ==> 7.00
```

- step = 7, minw = 6, pacost[6] = 7.00

The result is:

```
visited: {1, 1, 1, 1, 1, 1, 1}
```

```
parent: {-1, 0, 0, 0, 0, 0, 5}
pacost: {0.00, 1.00, 2.00, 3.00, 4.00, 5.00, 7.00}
```

Can you draw the SPT?

Greed can pay ...

Greedy algorithms make locally optimum choices at each step (when there is choice of course)

- ... but the choices may not result in a globally optimum solution

For instance, consider the *Traveling Salesman Problem*, which asks the question:

```
Given a list of cities and the distances between each pair of cities,
what is the shortest possible route that visits each city and returns to the origin city?
```

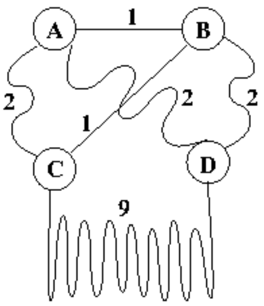
It is an NP-hard problem to solve for large *N* (number of cities)

- basically means that every algorithm will have exponential complexity or worse

A greedy algorithm to 'solve' this problem would be

- at each stage visit the nearest unvisited city to the current city

For example: consider the following 'map' of cities:



and a travelling salesman who needs to visit every city and be home for dinner.

If he's 'in a real rush' he'll take the best option whenever he leaves a city

- this will result in the route **A--B--C--D--A**, with a total cost of 1+1+9+2 = 13

If he planned ahead

- the route **A--C--B--D--A**, with cost 2+2+1+2 = 7 is almost 50% better

Greedy algorithms usually fail because they don't look at enough data to make a decision

- ... but Dijkstra's algorithm is a greedy algorithm that is globally optimal
  - the algorithms below, Prim's and Kruskal's MST algorithms, are also greedy, and also globally optimal
- so *greed does pay ... sometimes*

The difference lies in the problem:

- to plan a route, you need a lot of information
- to find a spanning graph, or shortest path tree, you need little

Complexity

Time complexity is dependent mainly upon how you

- implement the 'search for the closest vertex' operation

As described abovei, **Dijkstra's algorithm is O(V^2)**, where *V* is the number of vertices:

- *why?*
  - for each of the *V* vertices:
    - it must search up to *V-1* adjacent vertices for a minimum
    - it does this search in *linear* time, i.e. *O(V)* time
  - hence complexity is $O(V^2)$

So the algorithm takes $O(V^2)$ time to generate the SPT for a particular vertex.

- remember, Dijkstra's algorithm is an example of a *single-source* algorithm

If you want to generate the SPT for every vertex

- this would make the algorithm an *all-pairs* algorithm
  - i.e. find the shortest path from every vertex to every other vertex
- time complexity is $V * O(V^2) = O(V^3)$

To improve the performance, you need to 'search for the closest vertex' faster.

- use a *priority queue* to store the adjacent vertices and their costs
- search is then *O(log(V))*
- if you are building the SPT for a single source
  - complexity is then *O(Elog(V))*

Minimum Spanning Tree

(Chapter 20.1–20.4 Sedgewick)

Discovered by *Otakar Boruvka*, electrical engineer in 1926

- most economical construction of electric power network

Networks are everywhere:

- roads
- electrical wires
- water
- gas pipes
- computers are built of networks at many levels:
  - microscopic connections between transistors in a chip
  - ...
  - cables and satellites that link the internet around the world

Networks can be very complicated, but you may only want to know what are the minimum set of connections that will reach every vertex

- e.g. the minimum number of connections to broadcast a message to every node
- e.g. the minimum number of kilometers required to visit every city

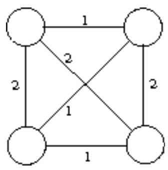
A *spanning tree* is a subgraph of a graph *G* that:

- consists of the same set of vertices as *G*
- consists of a subset of the edges of *G*
- is a tree

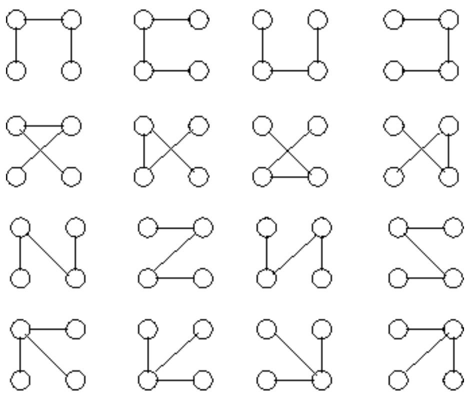
A minimum *spanning tree* (**MST**) of a weighted graph *G* is a spanning tree of *G* whose edges sum to minimum weight.

- There can be more than one minimum spanning tree in a graph:

For example: the following 'complete' graph on 4 vertices:



has 16 spanning trees:



Notice:

- each diagram is a tree
- each spans all the vertices
  - but only 1 happens to be a minimum spanning tree

    which one is it?

Assumptions:

- edges in *G* are not directed (MST for directed graphs is hard)
- no edge weights are negative
- graph is connected

Prim's formulation

Prim's method of generating a Minimum Spanning Tree is as follows:

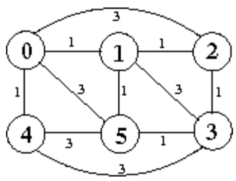
1. initially
   - partition all vertices into *mst* and *rest* sets
   - where
     - *mst* contains some *start* vertex
     - *rest* contains the rest of the vertices
2. select a minimum crossing edge
   - *crossing edges* are edges joining vertices in the *mst* and *rest* sets
3. move that edge and the adjacent vertex from *rest* to *mst*
4. repeatedly select and move until the *mst* set contains all vertices

'Choice' in the algorithm:

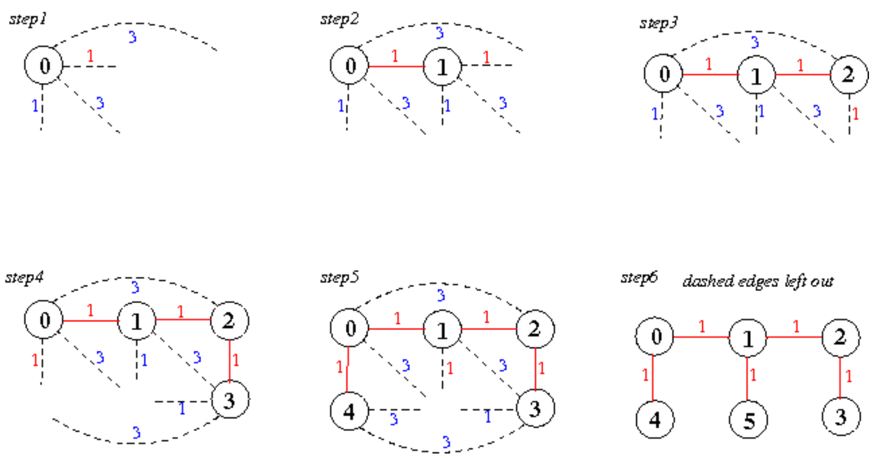
- we arbitrarily start with vertex *0*
- when selecting a minimum (crossing) edge, there may be more than one:
  - we arbitrarily select the edge that goes to the lowest vertex

Example 1 of Prim's method

Consider the graph:



Prim's algorithm generates a MST as follows:



Notice that:

- the vertices shown comprise the set *mst*
  - (the vertices in the set *rest* are not shown)

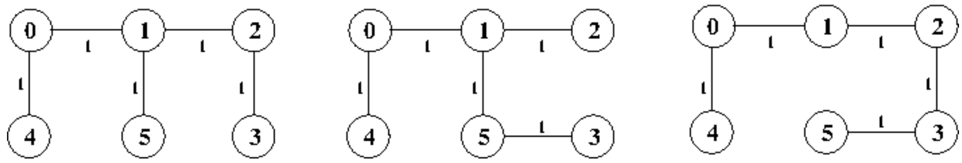
The red edges leaving the *mst* vertices are the crossing edges

- at each step, the minimum crossing edge is selected
  - that edge and the vertex adjacent are added to *mst*

By construction, the final *mst* will contain:

- all the vertices
- the minimum-weight set of edges that connect them

There are in fact 3 possible MSTs for the above graph:



- selecting different (equally minimum) crossing edges will generate a different MST:
- the sum of the edge weights in each is 5

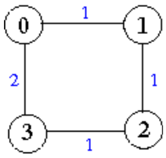
We saw the above graph before with Dijkstra's algorithm:

- the MST on the left is the same as the SPT!
- in fact, the MST on the right is also a SPT!

So, are MSTs the same as SPTs?

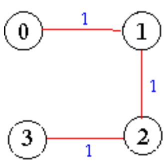
Example 2 of Prim's method

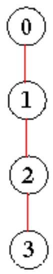
Consider the graph:



Prim's algorithm generates only 1 possible MST: It has a cost of 3.

MST *MST redrawn*



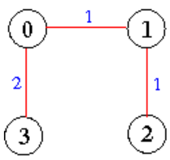


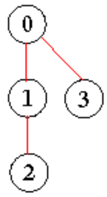
整体距离最小值

集体主义，整体最好

What is the SPT of the graph? The result of Dijkstra's algorithm is:

SPT *SPT redrawn*





某一顶点到达其他所有点的最小值

个人主义，每个人都最好

Notice that the sum of the weights in the SPT is 4, which is greater than the cost of the MST.

Conclusion:

- there can be more than 1 MST for a given graph
- there can be more than 1 SPT for a given graph

两者都可能不止一种

- both MSTs and SPTs ***span*** a graph 两者都是树
  - i.e. they include every vertex, and a minimum number of edges
- a MST is guaranteed to span a graph with minimal total edge weight
  - a SPT may be 'lucky' and be a MST, or the sum of its edge weights may be greater
    - it can never be less

SPT的权值≥MST的权值

个人主义需要的资源更多呗~

Why is a SPT sometimes worse than a MST?

- different criteria are used to include a vertex in Dijkstra's and Prim's algorithms
- Prim selects the vertex with minimum crossing-edge weight
  - ... and does not use path costs
- Dijkstra selects the vertex with the minimum distance to the start vertex
  - ... in other words, the path cost, which of course includes the edge cost to the vertex

Both

- **both are optimal algorithms**
- Prim guarantees to find a minimum MST

- Dijkstra guarantees to find a minimum SPT
- **both are greedy algorithms**

If the algorithms are so similar, can they be programmed in a similar way?

- *yes: in fact almost identically*
- the implementation above of Dijkstra's SPT algorithm, also does Prim's MST algorithm
  - the last parameter in the function is *alg* which is either **d** or **p**
  - **d**: Dijkstra, the next vertex to be included is determined by the cost of the path to the vertex
  - **p**: Prim, the next vertex to be included is determined by cost of the crossing edge alone

Complexity

A naive implementation based on an adjacency matrix is:

- $O(V^2)$.

If the edges are stored in a priority queue then searching for the next edge is faster. The complexity is:

- $O(Elog(V))$

Kruskal's algorithm

Informal algorithm: to compute the MST for graph $G(V,E)$:

- start with empty MST
- consider edges in increasing weight order
  - add edge to MST if it does not form a cycle
  - repeat until $V\text{-}1$ edges are added

Kruskal's algorithm for minimum spanning tree works by including edges in order of increasing cost

- works for connected graphs only

Imagine you have 128 U.S. cities, and wanted to span-tree them!

Kruskal demo on U.S. cities

Critical operations:

- iterating over edges in weight or cost order
- checking for cycles in a graph

Implementation of Kruskal's algorithm

The implementation of Dijkstra's (and Prim's) algorithms was done using arrays

- a linear search was used to find the next edge to visit
  - (the for-loop on line 13 that checks every vertex in the graph)
- the complexity of this algorithm is $O(V^2)$

We could have implementated them using priority queues ($O(Elog(V))$)

- the lecture notes on priority queues are here
- but the heap used in the priority queue was a ***max-heap*** ...
  - and we want the ***minimum spanning tree***

So, we need a ***min-heap*** implementation of the priority queue

- the priority queue is used to store edges
- *delMin* can remove an edge from the 'array' in $log(V)$ time
- must also check that a new edge will not cause a cycle

Here is an interface, which is a *Weighted Graph* version of a priority queue:

切换行号显示 Weighted Graph Priority Queue

```
// WeGPQ.h                      // different name
#include <stdio.h>
#include <stdlib.h>

typedef struct pqRep *PQ;

PQ   createPQ(int);             // same
void insertPQ(Edge, PQ);        // an Edge is inserted, not an int
Edge delMinPQ(PQ);              // 'min' not 'max plus returns Edge
int  isEmptyPQ(PQ);             // same
```

So we need 2 ADTs for Kruskal

切换行号显示

```
#include "WeGraph.h"
#include "WeGPQ.h"

```

Here is the Kruskal code:

切换行号显示

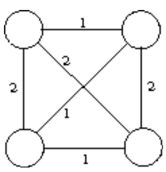
```
 1 Graph kruskal(Graph g, int nV, int nE) {
 2    Graph mst = newGraph(nV);
 3    Edge e;                        新建一个graph
 4    PQ p = createPQ(nE+1);         新建一个PQ
 5    Vertex v1;
 6    for (v1 = 0; v1 < nV; v1++) { // insert all the edges into the PQ
 7       Vertex v2;
 8       for (v2 = v1; v2 < nV; v2++) {
 9          e = getEdge(v1, v2, g);
10          if (isEdge(e, g)) {          将边按照小根堆输入到PQ中去
11             insertPQ(e, p);
12          }
13       }
14    }
15    int n = 0;  // counts the number of edges
16    Weight cost = 0.0;
17    while (!isEmptyPQ(p) && n < nV-1) {
18       e = delMinPQ(p); // delete an edge, smallest first
19       int *visited = mallocArray(nV);
20       int order = 0;
21       if (isPath(mst, e.v, e.w, nV, &order, visited) == 0) {
22          insertEdge(e, mst);
23          printf("Accept ");
24          showEdge(e);              删除PQ中的最短边，但是需要判断是否有循环，
25          putchar('\n');            因此把删除的 edge 插入到新的 mst graph 中去
26          cost += e.x;              判断是否存在循环，如果没有循环，则接收，
27          n++;                      存在循环则拒绝
28       }
29       else {
30          printf("Reject ");
31          showEdge(e);
32          putchar('\n');
33       }
34       free(visited);
35    }
36    printf("\tTotal cost = %f\n", cost);
37    return mst;
38 }
```

The function does the following:

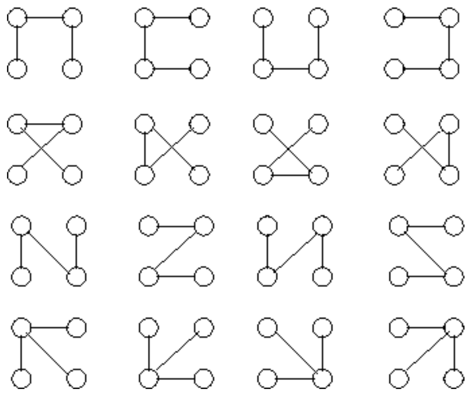
- creates a graph called *mst* in line 2
  - this is really unusual:
    - *mst* is a 'graph' created to hold the MST
    - it is separate to the input graph *g*
- creates a priority queue called *p* in line 4
- inserts the edges of the input graph *g* into the queue (line 11)
- in a loop, delMin's each of the *nV-1* edges from the queue (line 18)
  - if no cycle exists (line 21) then inserts that edge (line 22) into *mst*
  - otherwise ignore the edge
- returns *mst* to the calling program

Example 1 of Kruskal's algorithm

Remember the 'complete' graph on 4 vertices we saw earlier:



which has 16 spanning trees:



The input file for this graph is:

```
4
0 1:1    0 2:2    0 3:2
1 2:2    1 3:1
2 3:1
```

The output of Kruskal (using *showGraph(mst)* where *mst* is returned by function *kruskal*) is:

Accept 0-1: 1.00
Accept 1-3: 1.00
Accept 2-3: 1.00
Total cost = 3.000000
Minimum spanning tree:
V=4, E=3
0 1:1.00
1 0:1.00 **1 3**:1.00
2 3:1.00
3 1:1.00 3 2:1.00

Notice that no edges are 'rejected':

- *why is that?*

Example 2 of Kruskal's algorithm

The input file is:

```
4
0 1:2   0 2:2
1 2:1   1 3:1   2 3:1
```

The output is:

Accept 1-2: 1.00
Accept 2-3: 1.00
Reject 1-3: 1.00
Accept 0-1:2.00
Total cost = 4.000000
Minimum spanning tree:
V=4, E=3
0 1:2.00
1 0:2.00 **1 2**:1.00
2 1:1.00 **2 3**:1.00
3 2:1.00

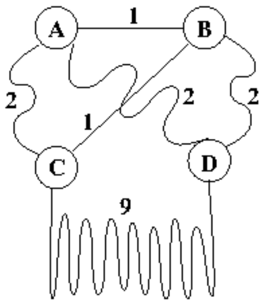
Example 2 of Kruskal's algorithm

The previous example was 'boring' because Kruskal didn't reject any edges

Let's consider the *Travelling Salesman's Problem*

- we stated before it's NP-hard (something like *N factorial!*)

Here it is again:



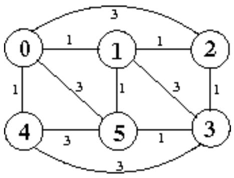
Here are the action carried out by Kruskal:

Accept A-B: 1.00
Accept A-C: 1.00
Reject B-C: 2.00
Accept A-D: 2.00
Total cost = 5.000000
Minimum spanning tree:
V=4, E=3
A B:1.00
A C:2.00 **A D**:2.00
B A:1.00
C A:2.00
D A:2.00

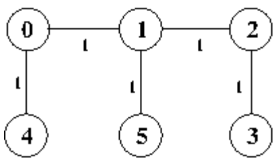
Example 3 of Kruskal's algorithm

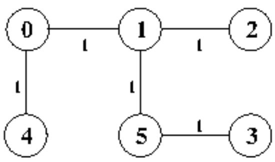
Here we compare the MSTs of Prim and Kruskal.

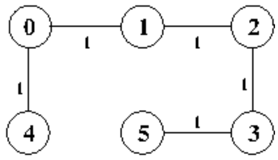
Consider the graph we saw earlier with Prim:



We saw that it generated the MST on the far left below:







Using Kruskal's algorithm, the MST on the far right is generated

- note the heapifying data structure used in this implementation make predicting which minimum edge gets chosen difficult

Comparison of Prim and Kruskal

- Prim and Kruskal both build an MST one edge at a time, but ...
- Kruskal focusses on edges
  - it builds a forest of minimum trees that merge
  - it adds the cheapest edge from anywhere in the graph to the MST
  - it is better for sparse graphs
- Prim focusses on vertices
  - it builds a single minimum tree that grows one edge at a time
  - it adds the cheapest 'crossing' edge that is connected to the MST
  - it is better for dense graphs

WeightedGraphAlgorithms (2019-08-01 13:27:43由AlbertNymeyer编辑)