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:

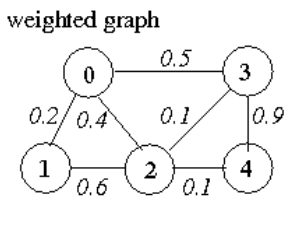
- sometimes there is a geometric interpretation of the term *weight*:
    - an edge with low weight is a "short" edge
    - an edge with high weight is a "long" edge
- other times it is simply a value
    - we'll use real numbers as edge weights
    - but, for convenience, we do not allow negative weights

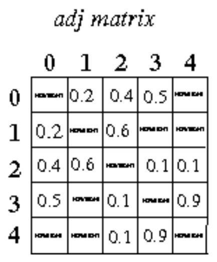
An ADT for Weighted Graphs

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

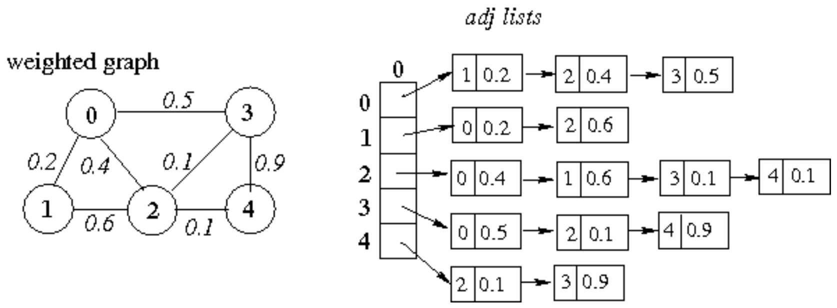
Adjacency Matrix with weights:




adj matrix

	0	1	2	3	4
0	NOWEIGHT	0.2	0.4	0.5	NOWEIGHT
1	0.2	NOWEIGHT	0.6	NOWEIGHT	NOWEIGHT
2	0.4	0.6	NOWEIGHT	0.1	0.1
3	0.5	NOWEIGHT	0.1	NOWEIGHT	0.9
4	NOWEIGHT	NOWEIGHT	0.1	0.9	NOWEIGHT

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);
```

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

Weighted-Graph Problems

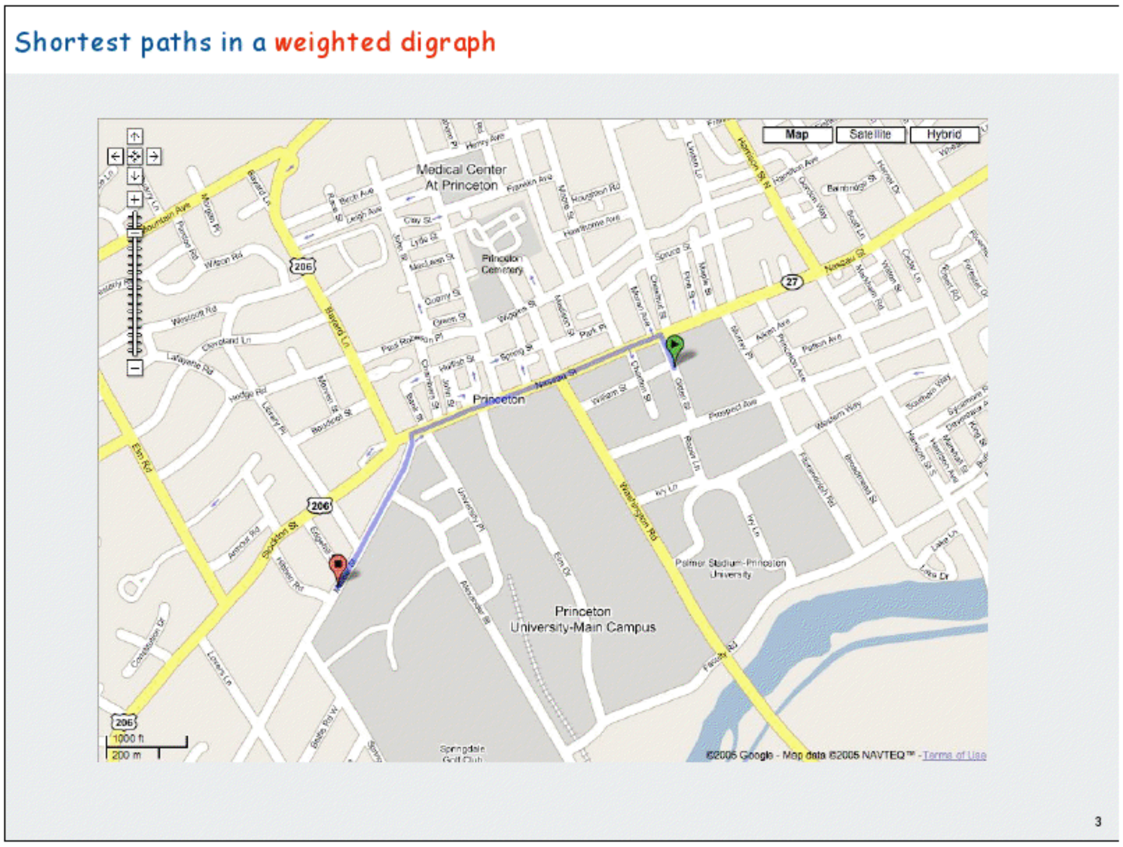
There are a huge number of applications of this data structure. We'll consider just two:

- 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:

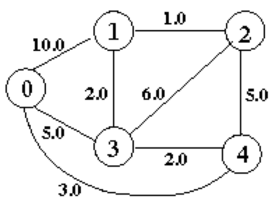
- Google maps:


Shortest paths in a weighted digraph
Map
Satellite
Hybrid
Medical Center At Princeton
Princeton
Princeton University-Main Campus
Palmer Stadium-Princeton University
3

- 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:

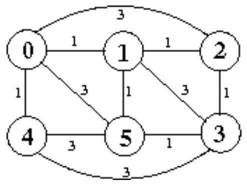
- from *0* to *3*
- from *0* to *4*

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

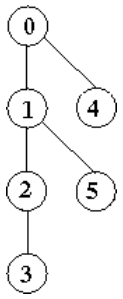
Consider the graph:



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 <u>tree</u> because every vertex has a single parent

- **Note, we assume that we want <u>a</u> shortest path, not <u>all</u> 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:

- we'll call the array **parent[ ]**
  - the index corresponds to the vertex label
  - the value is its parent vertex label
- 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[ ]**
  - 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]**

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

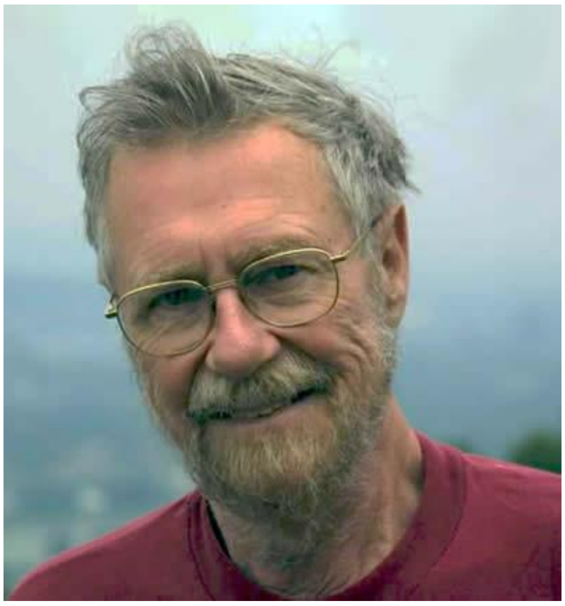
1. **single-source, single-target** (from *s* to *t*)
2. **single-source** (from *s* to <u>all</u> other vertices in the graph)
3. **all-pairs** (for all pairs of edges)

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
 * 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

The basic algorithm is:

```
set pacost[v] = INFINITY for all vertices
set parent[v] = NONE for all vertices
set all vertices to unvisited
set pacost[start] = 0
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*
      - 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*

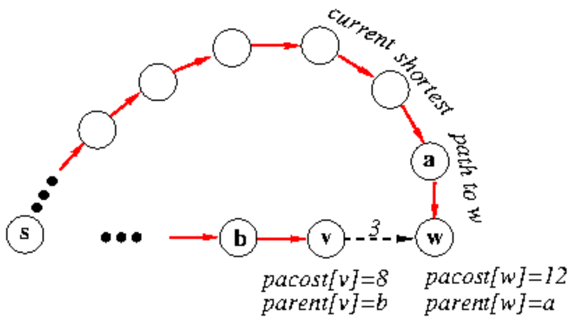
Dijkstra's algorithm demonstrated (3 mins 33 secs)

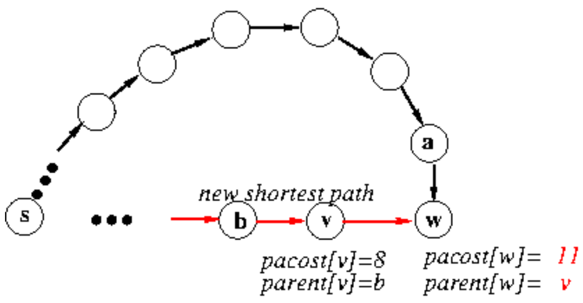
Edge relaxation

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

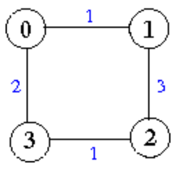
```
if (pacost[w] > pacost[v] + getWeight(g, v, w) {
    pacost[w] = pacost[v] + getWeight(g, v, w);
    parent[w] = v;
}
```

The edge relaxing above is *non-trivial* (the finite path cost is reduced)

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

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
      - *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
            - 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:

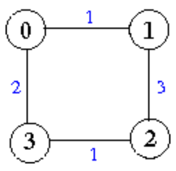
切换行号显示

```
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

   int *visited = mallocArray(nV);   // initialised to UNVISITED
   int *parent = mallocArray(nV);    // initialised to UNVISITED
   float *pacost = mallocFArray(nV); // floats: initialised to INFINITY

   pacost[src] = 0.0;
   for (int step = 1; step <= nV; step++) {
      Vertex minw = NONE;
      for (Vertex w = 0; w < nV; w++) {         // find minimum cost vertex
         if ((visited[w] == UNVISITED) &&
             (minw == NONE || pacost[w] < pacost[minw])) {
            minw = w;
         }
      }

      visited[minw] = VISITED;

      for (Vertex w = 0; w < nV; w++) {          //
         Weight minCost = getWeight(g, minw, w);// if minw == w, minCost = NOWEIGHT
         // minCost is cost of the minimum crossing edge
         if (minCost != NOWEIGHT) {
            if (alg == 'd') {                    // if DIJKSTRA ...
               minCost = minCost + pacost[minw];// add in the path cost
            }
            if ((visited[w] != VISITED) &&
                (minCost < pacost[w])) {
                   pacost[w] = minCost;
                   parent[w] = 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}
```

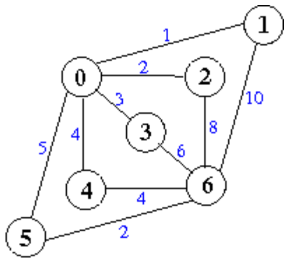
Here is the original graph again:



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

```
        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

```
        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?
```

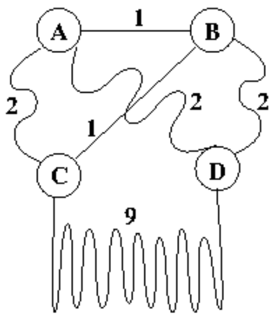
It is an NP-hard problem to solve for large *N* (number of cities)

- 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

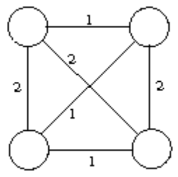
A *spanning tree* is a subgraph of a graph *G* that:

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

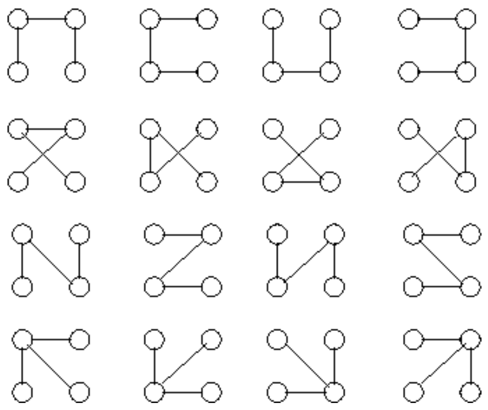
A minimum *spanning tree* (**MST**) of a weighted graph *G* is a spanning tree of *G* whose edges sum to minimum weight.

- 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:

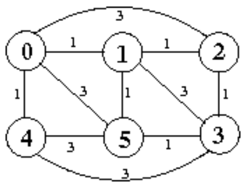
1. initially
   - partition all vertices into *mst* and *rest* sets
   - where
     - *mst* contains some *start* vertex
     - *rest* contains the rest of the vertices
2. select a minimum crossing edge
   - *crossing edges* are edges joining vertices in the *mst* and *rest* sets
3. move that edge and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

'Choice' in the algorithm:

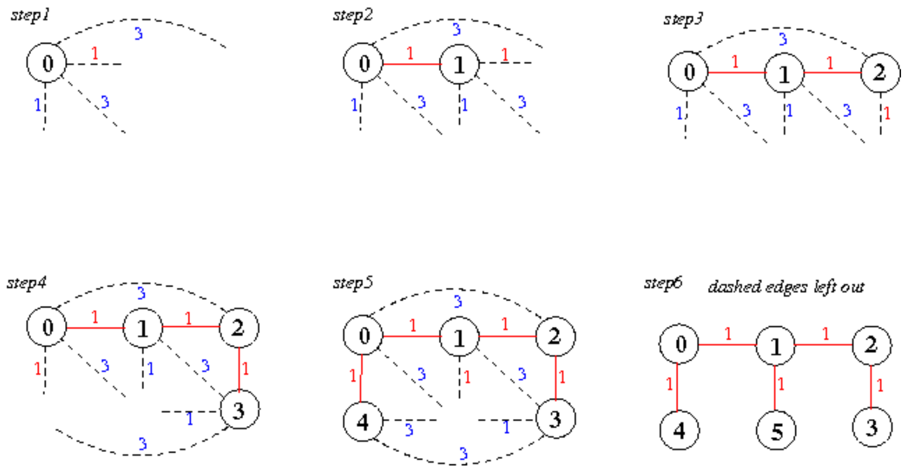
- we arbitrarily start with vertex *0*
- when selecting a minimum (crossing) edge, there may be more than one:
  - we arbitrarily select the edge that goes to the lowest vertex

Example: 6-vertex graph

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)

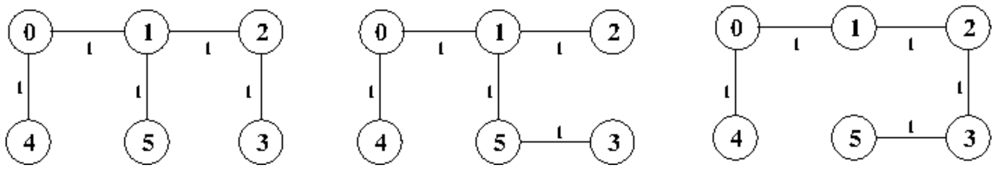
The red edges leaving the *mst* vertices are the crossing edges

- 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

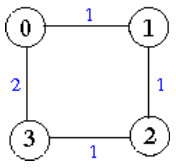
We saw the above graph before with Dijkstra's algorithm:

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

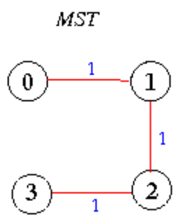
So, are MSTs the same as SPTs?

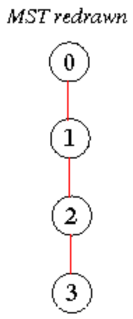
Example: 4-vertex graph

Consider the graph:

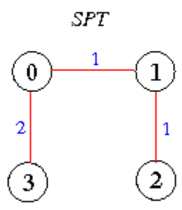


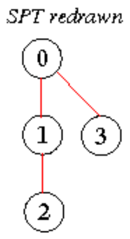
Prim's algorithm generates only 1 possible MST: It has a cost of 3.





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





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
- both MSTs and SPTs ***span*** a graph
  - 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

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:

- $O(E\log(V))$

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-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 ($O(E\log(V))$)

- 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

```
// 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

```
Graph kruskal(Graph g, int nV, int nE) {
   Graph mst = newGraph(nV);
   PQ p = createPQ(nE+1);
   Vertex v1;
   for (v1 = 0; v1 < nV; v1++) { // insert all the edges into the PQ
      Vertex v2;
      for (v2 = v1; v2 < nV; v2++) {
         Edge e = getEdge(v1, v2, g);
         if (isEdge(e, g)) {
            insertPQ(e, p);
         }
      }
   }
   int n = 0;  // counts the number of edges
   Weight cost = 0.0;
   while (!isEmptyPQ(p) && n < nV-1) {
      Edge e = delMinPQ(p); // delete an edge, smallest first
      int *visited = mallocArray(nV);
      int order = 0;
      if (isPath(mst, e.v, e.w, nV, &order, visited) == 0) {
         insertEdge(e, mst);
         printf("Accept ");
         showEdge(e);
         putchar('\n');
         cost += e.x;
         n++;
      }
      else {
         printf("Reject ");
         showEdge(e);
```

```
 31          putchar('\n');
 32       }
 33       free(visited);
 34    }
 35    printf("\tTotal cost = %f\n", cost);
 36    return mst;
 37 }
```

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

Crucial is the check for a cycle:

- *e* is the edge deleted from the priority queue in line 18
  - its endpoints are *e.v* and *e.w*
- if there is already a path between *e.v* and *e.w*
  - ... then adding this edge would result in a cycle

Checking for a path from *e.v* and *e.w* is done by:

切换行号显示

```
isPath(mst, e.v, e.w, nV, &order, visited)
```

The implementation of this recursive function is as follows:

切换行号显示

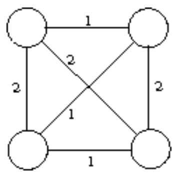
```
int isPath(Graph g, int curv, int goal, int numV, int *order, int *visited) {
// starting at 'curv', do a recursive DFS looking for vertex 'goal'
   int found = 0;
   visited[curv] = *order;
   *order = *order+1;
   if (curv == goal) {
      found = 1;
   }
   else {
      for (Vertex w=0; w < numV && !found; w++) {
         if (isEdge(newEdge(curv, w, 0.0), g)) {
            if (visited[w]==UNVISITED) {
               found = isPath(g, w, goal, numV, order, visited);
            }
         }
      }
   }
   return found;
}
```

- which returns true if any call finds *curv==goal*, otherwise false.

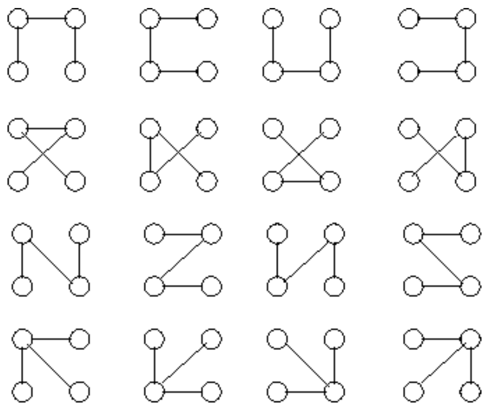
This in effect determines whether *goal* is *reachable* from *curv*.

Example: 4-vertex graph

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:

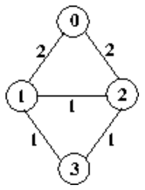
```
Accept 0-1: 1.00
Accept 1-3: 1.00
Accept 2-3: 1.00
Total cost = 3.000000
Minimum spanning tree:
V=4, E=3
0 1:1.00
1 0:1.00 1 3:1.00
2 3:1.00
3 1:1.00 3 2:1.00
```

Notice that no edges are 'rejected':

- *why is that?*

Example: 4-vertex graph with rejection

The previous example was 'boring' because Kruskal didn't reject any edges. Consider the following graph:



Obviously Kruskal must reject an edge here as picking the three lowest-weight vertices results in a cycle.

The corresponding input file is:

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

The output is:

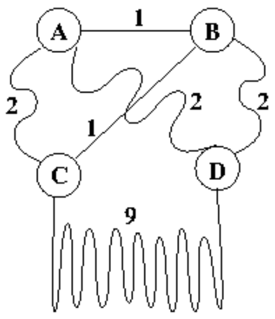
```
Accept 1-2: 1.00
Accept 2-3: 1.00
Reject 1-3: 1.00
Accept 0-1:2.00
Total cost = 4.000000
Minimum spanning tree:
V=4, E=3
0 1:2.00
1 0:2.00 1 2:1.00
2 1:1.00 2 3:1.00
3 2:1.00
```

Example: Travelling Salesman's Problem

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 actions carried out by Kruskal:

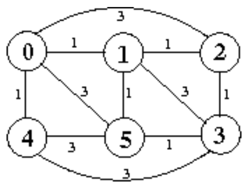
```
Accept A-B: 1.00
Accept B-C: 1.00
Accept B-D: 2.00
Total cost = 4.000000
Minimum spanning tree:
V=4, E=3
A B:1.00
B A:1.00 B C:1.00 B D:2.00
C B:1.00
D B:2.00
```

- Notice Kruskal selects the two weight-1 edges, plus B-D.
- Prim always goes for the lowest-weight vertices so if we used Prim's algorithm it would select the two weight-1 edges, plus A-D.
- both are MSTs of course

Example: 6-vertex graph comparison with Prim

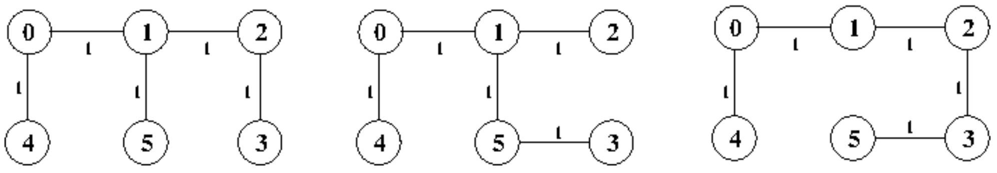
Here we compare the MSTs of Prim and Kruskal.

Consider the graph we saw earlier with Prim:



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

- Prim selects the nodes with lowest values first



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

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

General 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 (2019-08-02 16:54:27由AlbertNymeyer编辑)