Weighted Graph Algorithms

June 17, 2020

To represent weighted graphs, our adjacency list structure consists of an array of linked lists, such that the outgoing edges from vertex $ x $ appears in the $ edges[x]$

```
typedef struct {
    edgenode *edges[MAXV+1]; /* adjacency info */
    int degree[MAXV+1]; /* outdegree of each vertex */
    int nvertices; /* number of vertices in graph */
    int nedges; /* number of edges in graph */
    int directed; /* is the graph directed? */
} graph;
```

Each `edgenode` is a record containing three feilds, the first describing the second endpoint of the edge $ y $, the second enabling us to anotate the edge with a (*weight*) and the thrid pointing to the next edg in the list (*next*)

```
typedef struct {
    int y; /* adjacency info */
    int weight; /* edge weight, if any */
    struct edgenode *next; /* next edge in list */
} edgenode;
```

1 Minimum Spanning Trees

A *spanning tree* of a graph $ G = (V, E) $ is a subset of edges from $ E $ forming a tree connecting all vertices of $ V $.

For edge-weighted graphs, we are particularly intrested in the *minimum spanning tree* - the spanning trees whos sum of edge weights is as small as possible

Any tree is the smallest possible connected graph in terms of number of edges, while minimum spanning tree is the smallest conencted graph in terms of edge weight,

A minimum spanning tree minimizes the total lenght over all possible spanning trees.

All spanning trees of an unweighted graph $ G $ are minimum spanning trees, since each contains exactly $ n - 1 $ equal-weight edges.

1.1 Prim's Algorithm

Prim's minimum spanning tree algorithm starts form one vertex and grows the res of the tree one edge at a time untill all vertices are included

Greedy algorithms make the descision of what to do next by selecting the best local option from all available choices without regard to the global structure

The natural greedy algorithm for minimum spanning tree repeatedly selects the smallest weight edge that will enlarge the number of verices in the tree

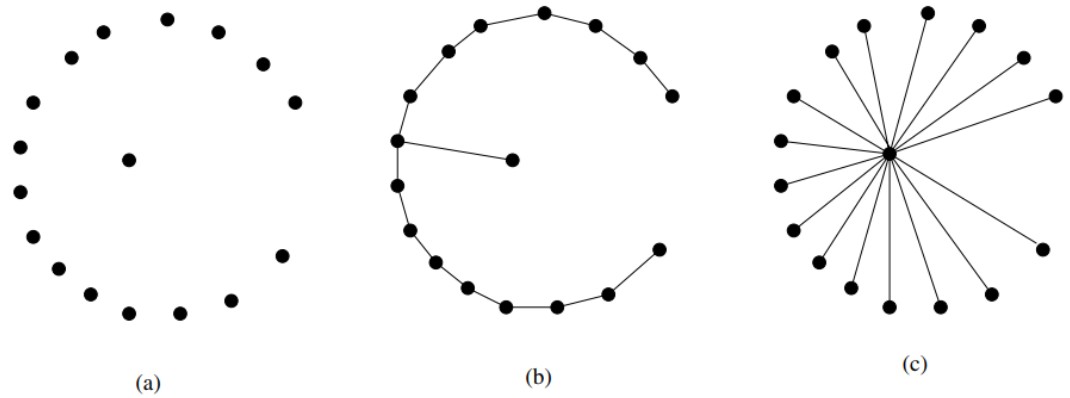


Figure 6: (a) Two spanning trees of point set; (b) the minimum spanning tree, and (c) the shortest path from center tree

```
Prim-MST(G)
    Select an arbitrary vertex s to start the tree from.
    While (there are still nontree vertices)
        Select the edge of minimum weight between a tree and nontree vertex
        Add the selected edge and vertex to the tree T_prim
```

Prim's algorithm clearly creates a spanning tree, because no cycle can be introduced by adding edges between tree and non-tree vertices

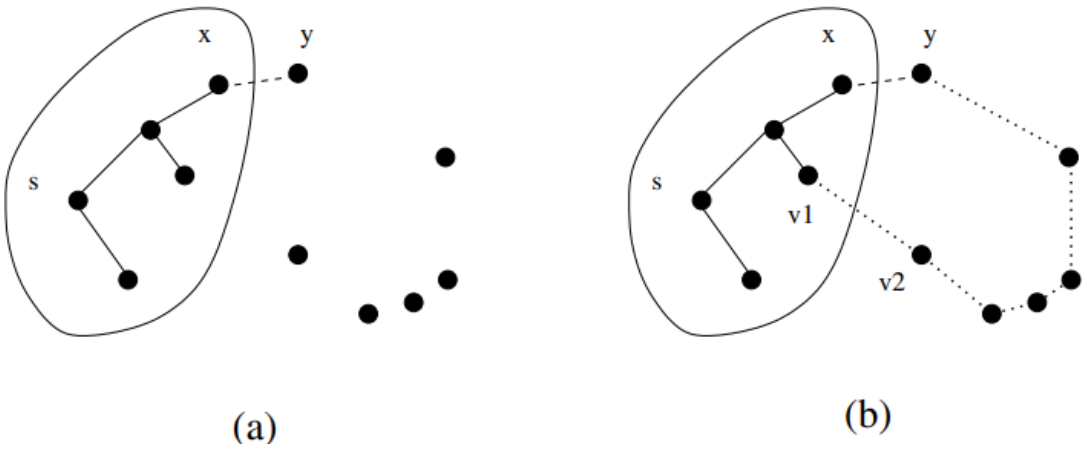


Figure 2: Where Prim's Algorithm goes bad? No, because $ d(v_1, v_2) >= d(x, y) $

We prove the MST with Prim's algorithm because suppose that instead of $ (a) $, the algorhtm chose $ (b)$, meaning it selected $ v_1 $ to $ v_2 $.

What were to happen if we deleted $ v_1, v_2 $ connection and reselected $ x, y $ connection?

Nothing, the distance would be the same. So by contradicition, Prim's algo must constuct a MST

1.1.1 Implementation

Prim's algo grows the MST in stages. Starting from a given vertex, At each iteration, we add one new vertex into the spanning tree. A greedy algo works, we always add the lowest-weighted edge linking a vertex in the tree to a vertex on the outside

The simplest implementation of this idea would assign each vertex a Boolean variable denoting wheter it is already in the tree (the array **intree** in the code below), and then searches all edges at each iteration to find the minimum weight edge with exactly one **intree** vertex

We cam make the inplementation smarter. If we keep track of the cheapest edge linking every nontree vertex in the tree. The cheapest such edge over all remaining non-tree vertices gets added in each iteration. We must update the costs of getting to the non-tree vertices after each insertion. However, since the most redently inserted vertex is the only change in the tree, all possible edge-weight updates must come from its outgoing edges

```
prim(graph *g, int start)
{
    int i; /* counter */
    edgenode *p; /* temporary pointer */
    bool intree[MAXV+1]; /* is the vertex in the tree yet? */
    int distance[MAXV+1]; /* cost of adding to tree */
    int v; /* current vertex to process */
    int w; /* candidate next vertex */
    int weight; /* edge weight */
    int dist; /* best current distance from start */

    for (i=1; i<=g->nvertices; i++) {
        intree[i] = FALSE;
        distance[i] = MAXINT;
        parent[i] = -1;
}

distance[start] = 0;
v = start;

while (intree[v] == FALSE) {
    intree[v] = TRUE;
    p = g->edges[v];
    while (p != NULL) {
        w = p->y;
        weight = p->weight;
        if ((distance[w] > weight) && (intree[w] == FALSE)) {
            distance[w] = weight;
            parent[w] = v;
        }
        p = p->next;
    }
```

```
    v = 1;
    dist = MAXINT;
    for (i=1; i<=g->nvertices; i++)
        if ((intree[i] == FALSE) && (dist > distance[i])) {
            dist = distance[i];
            v = i;
        }
}
```

1.1.2 Analysis

Prim's algortithm makes $ n $ iterations sweeping through all $ m $ edges on each iteration– yielding an $ O(mn) $ algorithm

But our implementation avoids the need to test all $ m $ edges on each pass. It only considers $ <= n $ cheapest known edges represented in the **parent** array and the $ <= n $ edges out of new tree vertex $ v $ to update **parent**.

By maintaining a Boolean flag along with each vertex to denote whether it is in the tree or not, we test whether the current edge joins a tree with a non-tree vertex in constant time

The result is an $ O(n^2) $ implementation of Prim's algorithm

A more sophisticated priority-queue data structure leads to an $ O(m + n lg n) $ implementation, by making it faster to dind the minimum cost edge to expand the tree at each iteration

1.2 Kruskal's Algorithm

Kruskal's algortihm is an alternative to prims and works more efficient on sparse graphs. Like Prim's, Kruskal's algorithm is greedy. Unlike Prim's, it does not start with a particular vertex

Kruskal's algorithm builds up conencted components of vertices culminating in the MST.

Initiall, each vertex froms its own separate compnent in the tree-to-be. The algorithm repeatedly considers the lightest remaining edge and tests whether its two endpoints lie within the same connected component. If so, this edge will be discarded because adding it would create a cycel in the tree-to-be. If the endpoints are in different components, we insert rhe edge and merge the two components into one. Since conencted components is always a tree, we need never explicitly test for cycles

```
Kruskal-MST(G)
    Put the edges in a priority queue ordered by weight.
    count = 0

    while (count < n - 1) do
        get next edge (v,w)
        if (component (v) != component(w))
            add to T_kruskal
            merge component(v) and component(w)
```

the algorithm adds $ n - 1 $ without creating a cycle, so it clearly creates a spanning tree for any connected graph.

Sorting the $ m $ edges takes $ O (m lg m)$ time. The `for` loop makes $ m $ iterations, each testing the connectivity of two trees plus an edge. In the most simple-mind approach, this can be implemented by breadth-first or depth-first search in a sparse graph with at most $ n $ edges and $ n $ vertices, this yielding an $ O (mn) $ algorithm

Using the more clever datastructure called *union-find* , we can support such queries in $ O (lg n) time.

With union-find, Kruskal's algo runs in $ O (m lg m)$ time

1.2.1 Implementation

```
kruskal(graph *g)
{
    int i; /* counter */
    set_union s; /* set union data structure */
    edge_pair e[MAXV+1]; /* array of edges data structure */
    bool weight_compare();

    set_union_init(&s, g->nvertices);

    to_edge_array(g, e); /* sort edges by increasing cost */
    qsort(&e,g->nedges,sizeof(edge_pair),weight_compare);

    for (i=0; i<(g->nedges); i++) {
        if (!same_component(s,e[i].x,e[i].y)) {
            printf("edge (%d,%d) in MST\n",e[i].x,e[i].y);
            union_sets(&s,e[i].x,e[i].y);
        }
    }
}
```

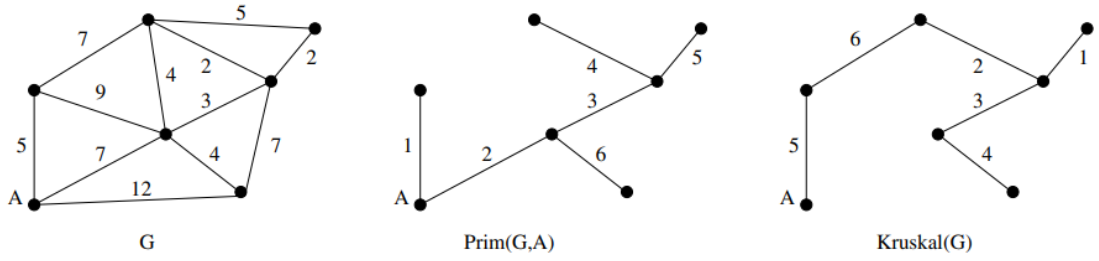


Figure 3: A graph $ G(1) $ with minimum spanning trees produced by Prim's $ (m)$ and Kruskal's $ (r) $ algorithms. The number on the trees denote the order of insertion; tries are broken arbitrarily

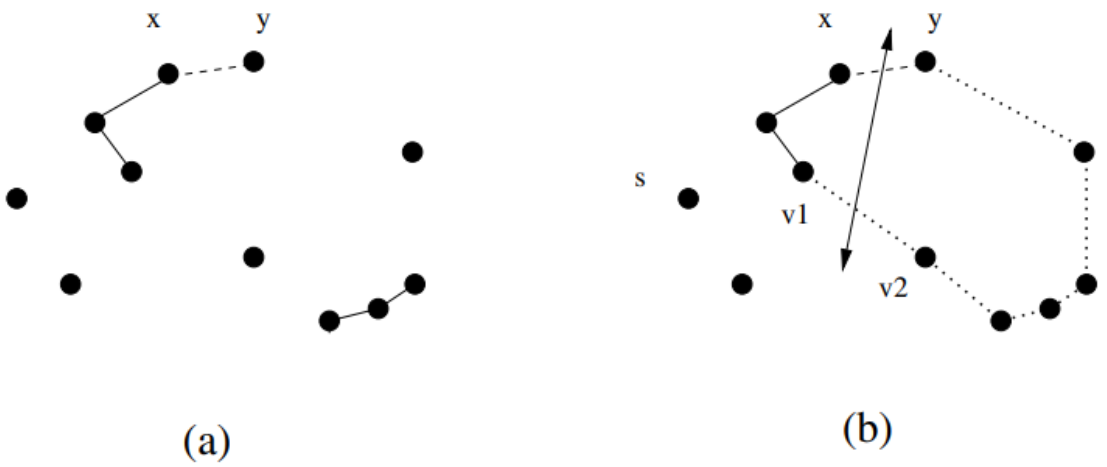


Figure 4: Where Kruskal's algorithm goes bad? No, because $ d(v_1, v_2) >= d(x, y) $

1.3 The Union-Find Data Structure

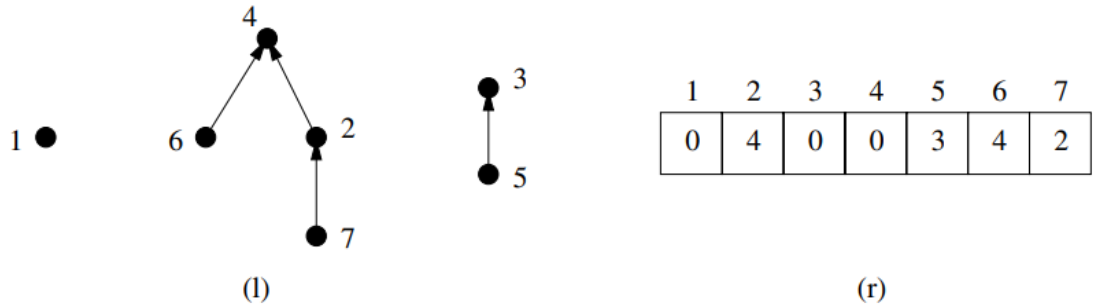


Figure 5: Union-find example: structure represented as trees $(1) $ and array (r)

A *set partion* is a partitioning of the elements of some universal set (say the integer $ 1 $ to $ n $) into a collection of disjointed subsets. Thus, each element must be in exactly one subset. Set partitions naturally arise in graph problems such as conencted compoennts (each vertex is in exactly one connected component) and vertex coloring (aperson may be male or female, but not both or neither)

For Kruskal's algortihm to run efficiently, we need a data structure that eddciently supports the following operations

- Same component(v_1, v_2)- basically find. Do vertices $ v_1 $ and $ v_2 $ occur in the same connected component of the current graph?
- Merge compoennt($ C_1, C_2$)- Merge the given pair of connected components into one component in respons eto an edge betwene them

Explicitly labeling each element with its component number enables the *same component* test to be performed in constant time, but updating the component nunber after a merger would require linear time.

The union-find data structure represents each subset as a "backwards" tree, with pointer from a node to its parent. Each node of this tree contains a set element, and the *name* of the set is taken

from the key at the root. We will also maintain the number of elements in the subtree rooted in each vertex $ v $

```
typedef struct {
    int p[SET_SIZE+1]; /* parent element */
    int size[SET_SIZE+1]; /* number of elements in subtree i */
    int n; /* number of elements in set */
} set_union;
```

Operations: - $ Find(i) $- find the root of tree containing element $ i $, by walking up the parent pointers untill there is no where to go. Return the lable of the root - $ Union(i, j) $- Link the root of one of the trees (say containing $ i $) to the root of the tree containing the other (say $ j $) so $ find(i) $ now equals $ find(j) $

We try to minimize the time it takes to execute *any* sequence of unions and finds. Tree structures can be very unbalanced, so we must limit the height of our trees

To minimize the tree height, it is beter to makr the smaller tree the subtree of the bigger one

This is because the height of all the nodes in the root subtree stay the same, while the height of the nodes merged into this tree all increase by one. This merging in the smaller tree leaves the height unchanged on the larger set of vertices

1.3.1 Implementation

```
set_union_init(set_union *s, int n)
{
    int i; /* counter */

    for (i=1; i<=n; i++) {
        s->p[i] = i;
        s->size[i] = 1;
    }

    s->n = n;

}

int find(set_union *s, int x)
{
    if (s->p[x] == x)
        return(x);
    else
            return( find(s,s->p[x]) );
}

int union_sets(set_union *s, int s1, int s2)
{
    int r1, r2; /* roots of sets */
```

```
    r1 = find(s,s1);
    r2 = find(s,s2);

    if (r1 == r2) return; /* already in same set */

    if (s->size[r1] >= s->size[r2]) {
        s->size[r1] = s->size[r1] + s->size[r2];
    s->p[ r2 ] = r1;
    }
    else {
        s->size[r2] = s->size[r1] + s->size[r2];
        s->p[ r1 ] = r2;
    }
}

bool same_component(set_union *s, int s1, int s2)
{
return ( find(s,s1) == find(s,s2) );
}
```

1.3.2 Analysis

On each union, the tree with fewer nodes becomes the child. But how tall can such a tree get as a function of the nodes in it?

We must double the number of nodes in the tree to get an extra unit of height, How many doublings can we do before we use up all $ n $ nodes? At most $ lg_2 n $ doublings can be performed. Thus, we can do both unions and finds in $ O (log n) $

1.4 Variations on MST

- Max Spanning Tree
  - suppose an evil cable company is paid by how much cable they put, this company want to put as much as possible. This is solved by negating the weights. The most negative tree in the negated graph is the max spanning tree
- Min Product Spanning Tree
  - Suppose we seek the spanning tree that minimizes the product of edge weights, assuming all edge weight are psotive. Since $ lg(a*b) = lg(a) + lg(b) $, the minimum spanning tree on a graph whos weights are replaced with their logarithms gives the minimum product spannign tree on the orginal graph
- Minimum Bottleneck Spanning Tree
  - What if we want to minimize the maximum edge weight over all such trees. Every minimum spanning tree has this property. To test this, delete all "heavy" edges from graph and ask wheter the result is still conencted. This can be done with a $ BFS/DFS $

2 Shortest Path

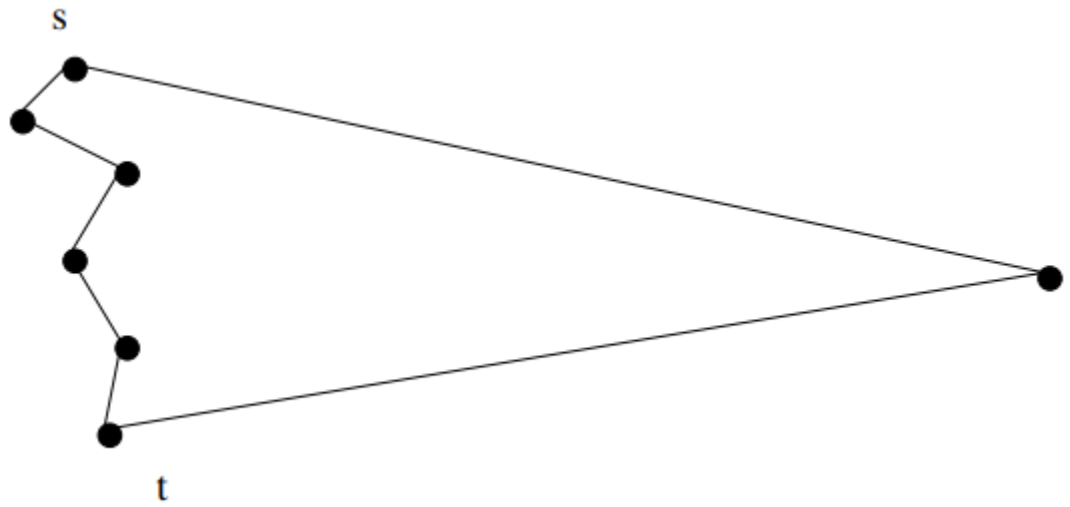


Figure 6: The shortest path from $ s $ to $ t $ may pass through many intermediate vertices

The shortest path from $ s $ to $ t $ in an unweighted graph can be constructed using a $ BFS $ search from $ s $

$ BFS $ shortest path does not work on a weighted graph because the shortest path may take a large number of edges

2.1 Dijkstra's Algorithm

Dijkstra's algorithm proceeds in a series of rounds, where each rough establishes the shortest path from $ s $ to *some* new vertex. Specifically, $ x $ is the vertex that minimizes $ dist(s, v_i) + w(v_i, x) $ over all unfinished $ 1 <= i <= n $, where $ w(i, j) $ is the lenght of the edge from $ i $ to $ j $, and $ dist(i, j) $ is the lenght of the shrotest path between them

This is a dynamic programming-like stragety. The shortest pah from $ s $ to itself is trival unless there are negative weight edges, so $ dist(s, s) = 0 $. If $ (s, y) $ is the lightest edge incicdent to $ s $, then this implies that $ dist(s, y) = w(s,)$. Once we determine the shortest path to a node $ x $, we check all the outgoing edges of $ x $ to see whether there is a better path from $ s $ to some unknown vertex through $ x $

```
ShortestPath-Dijkstra(G,s,t)
    known = {s}
    for i = 1 to n, dist[i] = ϖ
    for each edge (s,v), dist[v] = w(s,v)
    last = s
    while (last != t)
        select vnext, the unknown vertex minimizing dist[v]
        for each edge (vnext,x), dist[x] = min[dist[x],dist[vnext] + w(vnext,x)]
        last = vnext
        known = known  {vnext}
```

The basic ides is simmilar to Prim's algorithm. In each iteration, we add exactly one vertex to the

tree of vertices for which we *know* the shortest path from $ s $. As in Prim's, we keep track of the best path seen to date for all vertices outside the tree and insert them in order of increasing cost

The difference between Dijkstra's and Prim's algortihm is how they rate the desirability of each outside verted. In sortest path we simply select the next closest vertex to $ s $, not just the one with the smallest weight

2.1.1 Implementation

```
dijkstra(graph *g, int start) /* WAS prim(g,start) */
{
    int i; /* counter */
    edgenode *p; /* temporary pointer */
    bool intree[MAXV+1]; /* is the vertex in the tree yet? */
    int distance[MAXV+1]; /* distance vertex is from start */
    int v; /* current vertex to process */
    int w; /* candidate next vertex */
    int weight; /* edge weight */
    int dist; /* best current distance from start */

    for (i=1; i<=g->nvertices; i++) {
        intree[i] = FALSE;
        distance[i] = MAXINT;
        parent[i] = -1;
    }

    distance[start] = 0;
    v = start;

    while (intree[v] == FALSE) {
        intree[v] = TRUE;
        p = g->edges[v];
        while (p != NULL) {
            w = p->y;
            weight = p->weight;
    /* CHANGED */ if (distance[w] > (distance[v]+weight)) {
    /* CHANGED */ distance[w] = distance[v]+weight;
    /* CHANGED */ parent[w] = v;
        }
        p = p->next;
    }

   v = 1;
    dist = MAXINT;
    for (i=1; i<=g->nvertices; i++)
        if ((intree[i] == FALSE) && (dist > distance[i])) {
            dist = distance[i];
            v = i;
```

```
        }
    }
}
```

Dijkstra's algo finds the shortest path form $ s $ to all other vertices. This defines a shortest path spanning tree rooted in $ s $

2.1.2 Analysis

The complexity of it as implemented above is $ O(n^2) $

Dijkstra's works correctly on graphs without negative-cost edges. The reson is that midway through the execution we may enounter an edge with wieght so negative that it canges the cheapest way to get from $ s $ to some other vertex already in the tree

The most cost-effective way to get frm your house to your next-door neighbor would be repeatedly though the lobby of any bank offering you enough money to make the detour worthwile

2.2 Shortest Path With Node Costs

- Problem: Suppose we are given a graph whose weights are on the vertices, instead of the edges. Thus, the cost of a path from $ x $ to $ y $ is the sum of the weights of allvertices on the path.

A good idea would be to replace any refrence to the weight with a refrence to an edge with the weight of the destination vertex. But better would be to concentrate on consturcting an edge-weighted graph on which Dijkstra's algorithm will give the answer. Set the weight of each directed edge $ (i, j)$ in the input graph to the cost of vertex $ j $. Diklstra's algortihm now does the job

2.3 All Pair Shortest Path

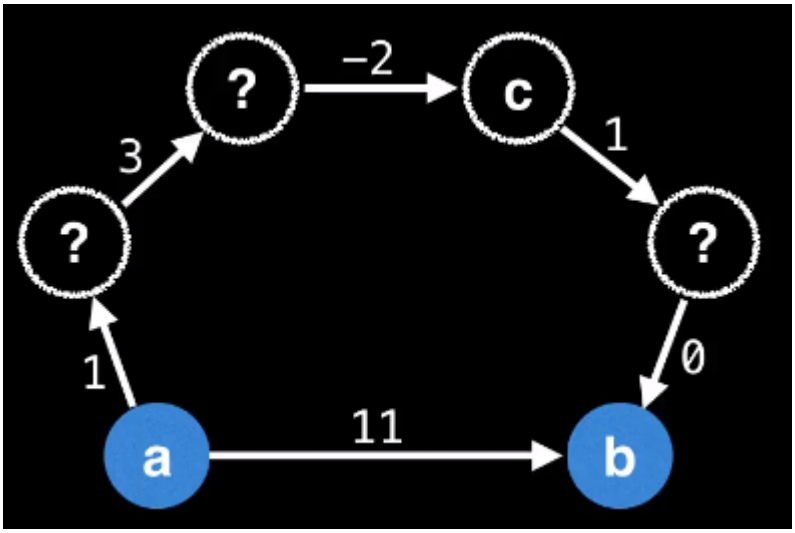
Suppose you want to find the "center" vertex in a graph- the one that minimizes the longest or average distance to all other nodes. This might be the best place to start a new business.

Or you want to know a graph's diameter- the longest shortest path distance over all pairs of vertices. This might correspond to the longest possible time it takes a letter or network packet to be delivered.

We could solve the *all-pair shortest path* by calling Dijkstra's algorithm from each of the $ n $ possible starting vertices

But Floyd's all-pair shortest-path algortihm is a slick way to construct this $ n x n $ distance matrix from the orginal weight matrix on the graph

Imagine that the shortest path from school to home is a stright line (1 vertex to another), but if you were to take multiple shortcuts, you could end up there faster. The cost howerver is going though more nodes



Floyd's algorithm is best employed on an adjaceny matrix data structure, which is no extravagance since we must store all $ n^2 $ paiewise distances anyways

```
typedef struct {
    int weight[MAXV+1][MAXV+1]; /* adjacency/weight info */
    int nvertices; /* number of vertices in graph */
} adjacency_matrix;
```

Normally Edges are $ 1 $ or $ 0 $. But this is wrong, we want ot initialize each edge to $ MAXINT $. This way we can both test whether it is present and automatically ignore it in shortest-path computations, since only real edges will be used, provided $ MAXINT $ is less than the diameter of your graph

The Floyd-Warshall algorithm starts by numbering the verties of the graph from $ 1 $ to $ n $. We use these number not to label the vertices but to ordet them

Define $ W[i, j]^k $ to be the lenght of the shortest path from $ i$ to $ j $ using only vertices numbered from $ 1, 2, ..., k $ as pssoble intermediate vertices

When $ k = 0 $, we are allowed no intermediate vertices, so the only allowed paths are the orginal edges in the graph.

Thus the initial all-pair shortest-path matrix consists of the initial adjacency matrix. We will perfrom $ n $ iterations, where the kth iteration allows only the first $ k $ vertices as possible form $ n $ iterations, where the $ kth $ iteration allows only the frist $ k $ vertices as possible intermediate steps on the path betweem each pair of vertices $ x $ and $ y $

At each iteration we allow a richer set of possible shortest paths by adding a new vertex as a possible intermediary. Allowing the $ kth $ vertex as a stop helps only if there is a short path that goes though $ k $, so

$$W[i,j]^k = min(W[i,j]^{k-1} + W[k,j]^{k-1})$$

The correctness of this is somewhat subtle, and i encourage you to convince youself of it. But there is nothing subtle about how simple the implementation is

```
floyd(adjacency_matrix *g)
{
    int i,j; /* dimension counters */
    int k; /* intermediate vertex counter */
    int through_k; /* distance through vertex k */

    for (k=1; k<=g->nvertices; k++)
        for (i=1; i<=g->nvertices; i++)
            for (j=1; j<=g->nvertices; j++) {
                through_k = g->weight[i][k]+g->weight[k][j];
                if (through_k < g->weight[i][j])
                    g->weight[i][j] = through_k;
            }
}
```

2.3.1 Analysis

The Floyd-Warshal all-pairs shortest path runs in $ O(n^3) $ time, which is asymptotically no better than $ n $ calls to Dijsktra's algorithm.

One of the rare algortihms that work on adjacency matrixs over adjacency lists.

Do not use on large graphs

2.3.2 Transitive Closure

Floyd's algortihm has another important application, that is computing *teansitive clousre* .

In analyzing a directed graph, we are often intrested in which vertices are reachable from a given node

The vertices reachable from any single node can be computed suing $ BFS $ of $ DFS $. But the whole batch can be computed using an all-pairs shortest path

3 Network Flows and Bipartite Matching

Edge-weighted graphs can be interpreted as a network of pipes, where the weight of edge $ (i, j) $ determines the capacity of the pipe. A wide pipe might be able to carry $10 $ unites of flow in a given time, where as a narrower pipe might only carry $ 5 $ units

The *network flow problem* asks for the maximum amount of flow which can be sent from vertices $ s $ to $ t $ in a given weighted graph $ G $ while respecting the maximum capacities of each pipe

3.1 Biparite Matching

A *matching* in a graph $ G = (V, E) $ is a subset of edges $ E' E $, such that no two edges of $ E' $ share a vertex. A matching pairs off certain vertices such that every vertex is in, at most, one such pair as show in the figure below

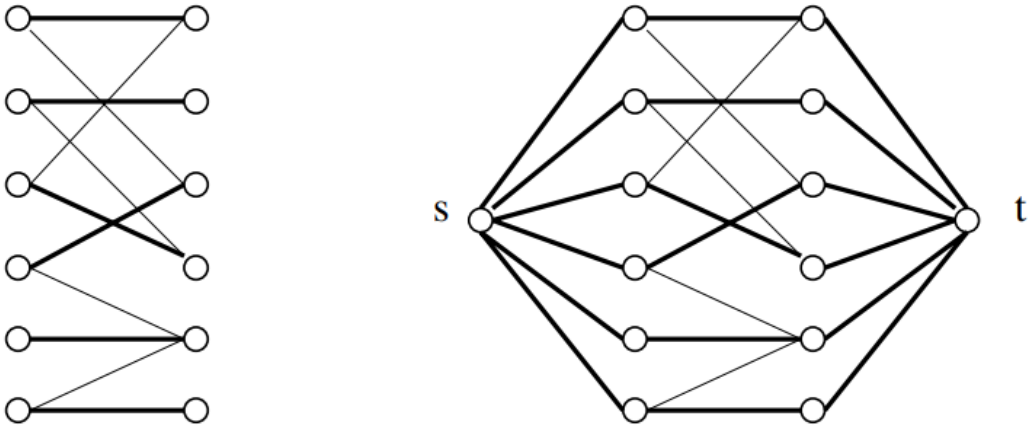


Figure 7: Bipartite graph with a mazimum matching highlighted (on left). The corresponding network flow instance highlighting the maximum $ s - t $ flow (on right)

Graph $ G $ is *bipartite* or *two-colorable* if the vertices can be divided into two sets $ L $ and $ R $, such that all edges in $ G $ have one vertex in $ L $ and one vertex in $ R $

Many graphs are bipartite, for example, jobs that can be done and people who can do them

The existance of edge $ (j, p) $ means that job $ j $ can be done by person $ p $.

The largest bipartite matching can be readily found using network flow. Create a *soruce node* $ s $ that is conencted to every vertex in $ L $ by an edge of weight $ 1 $. Create a *sink* node $ t $ and conenct it to every vertex in $ R $ by an edge of weight $ 1 $. Finally assign each edge in the bipartite graph $ G $ a weight of $ 1 $. Now the maximum possible flow from $ s $ to $ t $ defines the largest matching in $ G $. We cam find a flow as large as the matching by using only the matching edges and their source-to-sink conenctions..

3.2 Computing Network Flows

Network flow algorithms are based on the idea of *augmenting paths* and repeatedly finding a path of postive capacity from $ s $ to $ t $ and adding it ot the flow.

It can be shown that a flow through a network is optimal if and only if it contains no augmenting path. Since each augmentation adds to the flow, we must eventually find the global maximum

The key structure is the *residual flow graph* , denoted as $ R(G, f) $, where $ G $ is the input graph and $ f $ is the current flow through $ G $. This directed, edge-weighted $ R (G, f) $ contains the same verices as $ G $.

For each edge $ (i, j) $ in $ G $ with capacity $ c(i, j)$ and flow $ f(i, j $, $ R(G, f)$ may contain two edges: - $(i) $ an edge $ (i, j)$ with weight $ c(i, j)$ - $ f(i, j)$, if $ c(i, j)$ - $ f(i, j) < 0 $ and - (*ii*) an edge $ (j, i) $ with weight $ f(i, j) $, if $ f(i, j) > 0 $

The presence of edge $ (i, j) $ in the residual graph indicates that postive flow can be pushed from $ i $ to $ j $. The weight of the edge gives the exact amount that can be pushed

A path in the residual flow graph from $ s $ to $ t $ implies that more flow can be pushed from $ s $ to $ t $ and the minimum edge weight on the path defines the amount of extra flow that can be pused

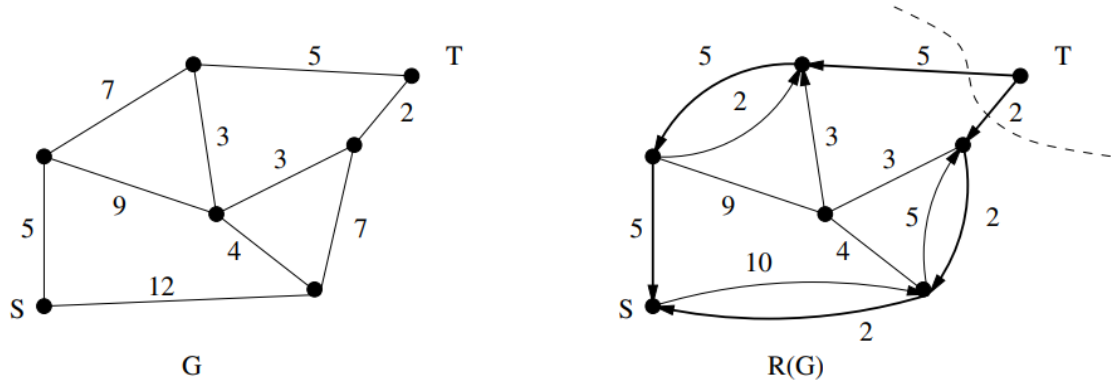


Figure 8: Maximum $ s - t $ flow in a graph $ G $ (on left) showing the associated residual graph $ R(G) $ and minimum $ s - t $ cut (dotted line near $ t $)

Figure 8 shows the maximum $ s - t $ flow in graph $ G $ is $ 7 $. Such a flow is revealed by the two directed $ t $ to $ s $ paths in residual graph $ R(G) $ of cpacities $ 2 + 5 $ respectively.

These flows completely saturate the capacity of the two edges inciden to vertex $ t $, so no augmentation path remains. Thus the flow is optimal.

A set of edges whos deletion seprates $ s $ from $ t $ (like the two edges incident to $ t $ is called an $ s-t $ cut.

Clearly, no $ s $ to $ t $ flow can exceed the weight of the minimum such cut. In fact, a flow equal to the minimum cut is alway possible

The maximum flow from $ s $ to $ t $ is always equals the wight of the minimum $ s-t $ cut. Thus, flow algorithms can be used to solve general edge and vertex connecctivity problems in graphs

3.2.1 Implementations

For each `edge` in the residual flow graph, we must keep track of both the amount of flow currently going through the edge, as well as its remaining *residual capacity*. Thus we must modify our `edge` structure to accomodate the extra feilds

```
typedef struct {
    int v; /* neighboring vertex */
    int capacity; /* capacity of edge */
    int flow; /* flow through edge */
    int residual; /* residual capacity of edge */
    struct edgenode *next; /* next edge in list */
} edgenode;
```

We use breadth-first earch to look for any path from source to sink that increases the total flow, and use it to augment the total

We teminate with the optimal flow when no such augmenting path exists

```
netflow(flow_graph *g, int source, int sink)
{
    int volume; /* weight of the augmenting path */
```

```
    add_residual_edges(g);

    initialize_search(g);
    bfs(g,source);

    volume = path_volume(g, source, sink, parent);

    while (volume > 0) {
        augment_path(g,source,sink,parent,volume);
        initialize_search(g);
        bfs(g,source);
        volume = path_volume(g, source, sink, parent);
    }
}
```

Any augmenting path from source to sink increases the flow, so use $ BFS $ to find such a path in the appropriate graph. We only consider network edges that have remaining capacity or, in other words, postive residual flow.

The predicate below helps $ BFS $ distinquish betwen saturated and unsaturated edges

```
bool valid_edge(edgenode *e)
{
    if (e->residual > 0) return (TRUE);
    else return(FALSE);
}
```

Augmenting a path transfer the maximum possible volume from the residual capacity into postive flow. The amount is limited by the path-edge with the smallest amount of residual capacity, just as the rate at which traffic can flow is limited by the most congested point

```
int path_volume(flow_graph *g, int start, int end, int parents[])
{
    edgenode *e; /* edge in question */
    edgenode *find_edge();

    if (parents[end] == -1) return(0);

    e = find_edge(g,parents[end],end);

    if (start == parents[end])
        return(e->residual);
    else
        return( min(path_volume(g,start,parents[end],parents),
            e->residual) );
}

edgenode *find_edge(flow_graph *g, int x, int y)
{
```

```
    edgenode *p; /* temporary pointer */

    p = g->edges[x];

    while (p != NULL) {
        if (p->v == y) return(p);
        p = p->next;
    }

    return(NULL);
}
```

Sending an additional unit of flow along directed edges $ (i, j) $ reduces the residual capacity of edge $ (i, j)$ but increases the residual capacity of edge $ (j, i)$. Thus, the act of augmenting a path requires modifying both forward and reverse edges for each link on the path

```
augment_path(flow_graph *g, int start, int end, int parents[], int volume)
{

    edgenode *e; /* edge in question */
    edgenode *find_edge();

    if (start == end) return;

    e = find_edge(g,parents[end],end);
    e->flow += volume;
    e->residual -= volume;

    e = find_edge(g,end,parents[end]);
    e->residual += volume;

    augment_path(g,start,parents[end],parents,volume);
}
```

Initializing the flow graph requires creating directed flow edges $ (i, j) $ and $ (j, i)$ for each network edge $ e = (i, j)$. Initial flows are all set to $ 0 $. The inital residual flow of $ (i, j) $ is set to the capacity of $ e $, while the initial residual flow of $ (j, i)$ is set to $ 0 $

The algorithm above is the **Edmonds-Karp** algorithm and the big $ O (n^3) $