Week 08: Graph Algorithms 2

Weighted Graphs

Weighted Graphs

Graphs so far have considered

- edge = an association between two vertices/nodes
- may be a precedence in the association (directed)

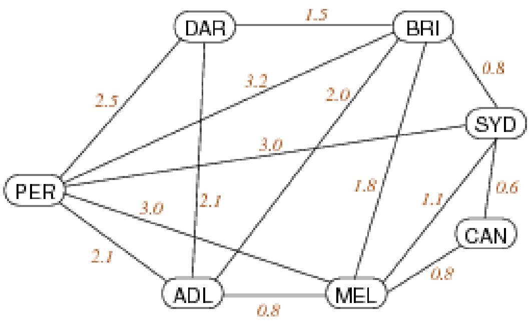
Some applications require us to consider

- a *cost* or *weight* of an association
- modelled by assigning values to edges (e.g. positive reals)

Weights can be used in both directed and undirected graphs.

... Weighted Graphs

Example: major airline flight routes in Australia



Representation: edge = direct flight; weight = approx flying time (hours)

... Weighted Graphs

Weights lead to minimisation-type questions, e.g.

1. Cheapest way to connect all vertices?

- a.k.a. *minimum spanning tree* problem
- assumes: edges are weighted and undirected

2. Cheapest way to get from *A* to *B*?

- a.k.a *shortest path* problem
- assumes: edge weights positive, directed or undirected

Exercise #1: Implementing a Route Finder

If we represent a street map as a graph

- what are the vertices?

- what are the edges?
- are edges directional?
- what are the weights?
- are the weights fixed?

What kind of algorithm would …

- help us find the "quickest" way to get from A to B?

Weighted Graph Representation

6/61

Weights can easily be added to:

- adjacency matrix representation   (0/1 → int or float)
- adjacency lists representation   (add int/float to list node)

An alternative representation useful in this context:

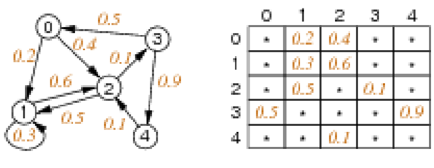
- edge list representation   (list of *(s,t,w)* triples)

All representations work whether edges are directed or not.

... Weighted Graph Representation

7/61

Adjacency matrix representation with weights:



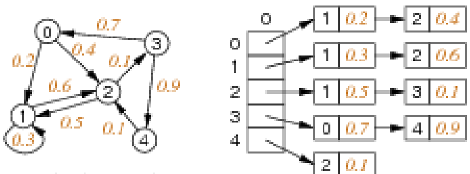
Weighted Digraph   *Adjacency Matrix*

Note: need distinguished value to indicate "no edge".

... Weighted Graph Representation

8/61

Adjacency lists representation with weights:



Weighted Digraph   *Adjacency Lists*

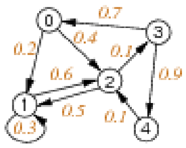
Note: if undirected, each edge appears twice with same weight

---

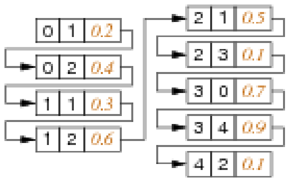
... Weighted Graph Representation

9/61

Edge array / edge list representation with weights:



Weighted Digraph



Edge List

Note: not very efficient for use in processing algorithms, but does
give a possible representation for min spanning trees or shortest paths

---

... Weighted Graph Representation

10/61

Sample adjacency matrix implementation in C requires minimal changes to previous Graph ADT:

WGraph.h

```
// edges are pairs of vertices (end-points) plus positive weight
typedef struct Edge {
   Vertex v;
   Vertex w;
   int    weight;
} Edge;

// returns weight, or 0 if vertices not adjacent
int adjacent(Graph, Vertex, Vertex);
```

... Weighted Graph Representation

11/61

WGraph.c

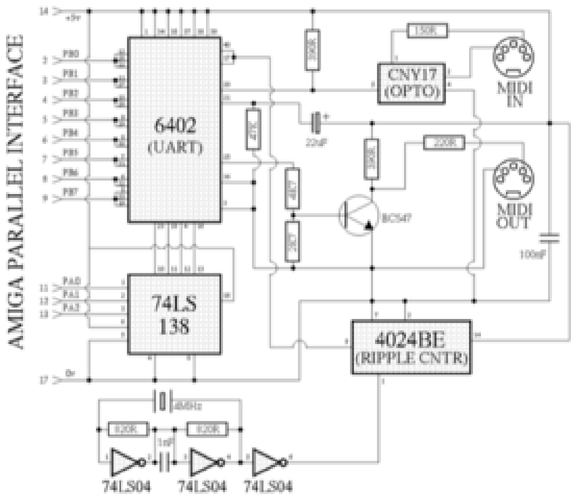
```
typedef struct GraphRep {
   int **edges;  // adjacency matrix storing positive weights
                 // 0 if nodes not adjacent
   int   nV;     // #vertices
   int   nE;     // #edges
} GraphRep;


void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));
   if (g->edges[e.v][e.w] == 0) {  // edge e not in graph
      g->edges[e.v][e.w] = e.weight;
      g->nE++;
   }
}

int adjacent(Graph g, Vertex v, Vertex w) {
   assert(g != NULL && validV(g,v) && validV(g,w));
   return g->edges[v][w];
}
```

Minimum Spanning Trees

Exercise #2: Minimising Wires in Circuits

13/61

Electronic curcuit designs often need to make the pins of several components electrically equivalent by wiring them together.



To interconnect a set of *n* pins we can use an arrangement of *n-1* wires each connecting two pins.

What kind of algorithm would …

- help us find the arrangement with the least amount of wire?

Minimum Spanning Trees

Reminder: *Spanning tree ST* of graph *G=(V,E)*

- *spanning* = all vertices, *tree* = no cycles
- *ST* is a subgraph of *G* (*G'=(V,E')* where *E'* $\subseteq$ *E*)
- *ST* is *connected* and *acyclic*

Minimum spanning tree MST of graph *G*

- *MST* is a spanning tree of *G*
- sum of edge weights is no larger than any other ST

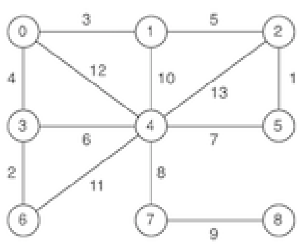
Applications: Computer networks, Electrical grids, Transportation networks …

Problem: how to (efficiently) find MST for graph *G*?

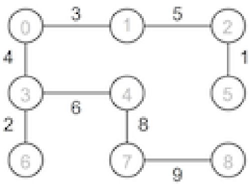
NB: MST may not be unique (e.g. all edges have same weight ⇒ every ST is MST)

... Minimum Spanning Trees

Example:



An MST …



... Minimum Spanning Trees

Brute force solution:

```
findMST(G):
|  Input  graph G
|  Output a minimum spanning tree of G
|
|  bestCost=∞
|  for all spanning trees t of G do
|  |  if cost(t)<bestCost then
|  |     bestTree=t
|  |     bestCost=cost(t)
|  |  end if
|  end for
|  return bestTree
```

Example of *generate-and-test* algorithm.

Not useful because #spanning trees is potentially large (e.g. n^{n-2} for a complete graph with n vertices)

... Minimum Spanning Trees

Simplifying assumption:

- edges in *G* are not directed (MST for digraphs is harder)

Kruskal's Algorithm

One approach to computing MST for graph *G* with *V* nodes:

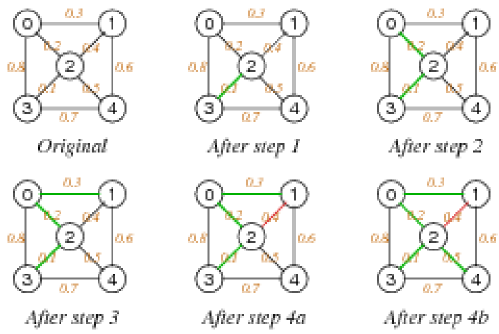
1. start with empty MST
2. consider edges in increasing weight order
   - add edge if it does not form a cycle in MST
3. repeat until *V-1* edges are added

Critical operations:

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

... Kruskal's Algorithm

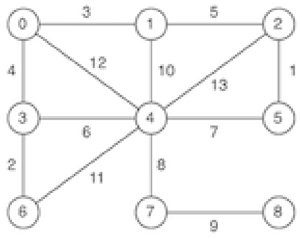
Execution trace of Kruskal's algorithm:



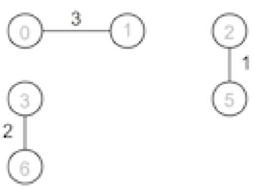
Exercise #3: Kruskal's Algorithm

20/61

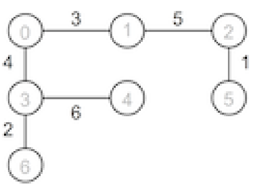
Show how Kruskal's algorithm produces an MST on:



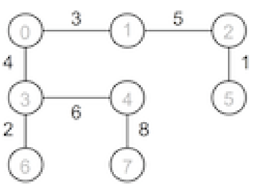
After 3rd iteration:



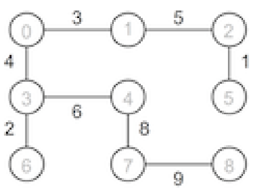
After 6th iteration:



After 7th iteration:



After 8th iteration (*V*-1=8 edges added):



... Kruskal's Algorithm

22/61

Pseudocode:

```
KruskalMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  sort edges(G) by weight
|  for each e∈sortedEdgeList do
|  |  MST = MST ∪ {e}
|  |  if MST has a cyle then
|  |     MST = MST \ {e}
|  |  end if
|  |  if MST has n-1 edges then
|  |     return MST
|  |  end if
|  end for
```

... Kruskal's Algorithm

Rough time complexity analysis …

- sorting edge list is $O(E \cdot log\ E)$
- at least V iterations over sorted edges
- on each iteration …
  - getting next lowest cost edge is $O(1)$
  - checking whether adding it forms a cycle: cost = ??

Possibilities for cycle checking:

- use DFS … too expensive?
- could use *Union-Find data structure* (see Sedgewick Ch.1)

Prim's Algorithm

Another approach to computing MST for graph $G=(V,E)$:

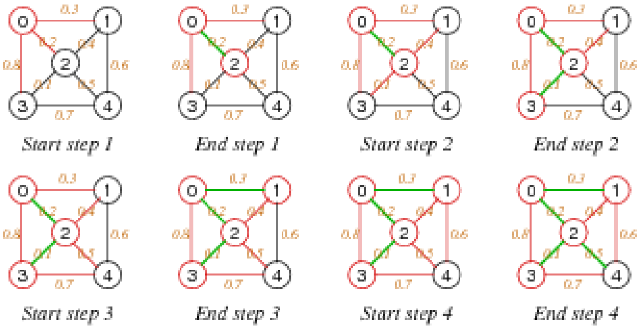
1. start from any vertex v and empty MST
2. choose edge not already in MST to add to MST
   - must be incident on a vertex s already connected to v in MST
   - must be incident on a vertex t not already connected to v in MST
   - must have minimal weight of all such edges
3. repeat until MST covers all vertices

Critical operations:

- checking for vertex being connected in a graph
- finding min weight edge in a set of edges

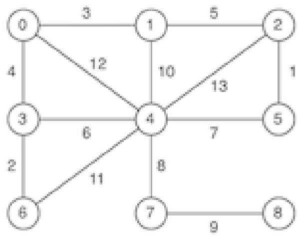
... Prim's Algorithm

Execution trace of Prim's algorithm (starting at s=0):



Exercise #4: Prim's Algorithm

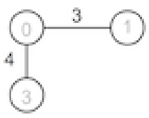
Show how Prim's algorithm produces an MST on:



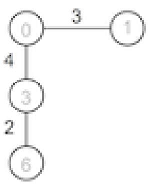
Start from vertex 0

After 1st iteration:

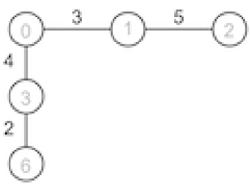
After 2nd iteration:



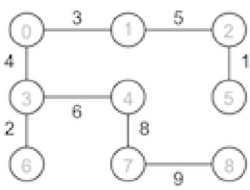
After 3rd iteration:



After 4th iteration:



After 8th iteration (all vertices covered):



... Prim's Algorithm

Pseudocode:

```
PrimMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  usedV={0}
|  unusedE=edges(g)
|  while |usedV|<n do
|  |  find e=(s,t,w)∈unusedE such that {
|  |     s∈usedV, t∉usedV and w is min weight of all such edges
|  |  }
|  |  MST = MST ∪ {e}
|  |  usedV = usedV ∪ {t}
|  |  unusedE = unusedE \ {e}
|  end while
|  return MST
```

Critical operation: finding best edge

... Prim's Algorithm

Rough time complexity analysis …

- *V* iterations of outer loop
- in each iteration …
  - find min edge with set of edges is $O(E) \Rightarrow O(V{\cdot}E)$ overall
  - find min edge with *priority queue* is $O(\log E) \Rightarrow O(V{\cdot}\log E)$ overall

Sidetrack: Priority Queues

Some applications of queues require

- items processed in order of "priority"
- rather than in order of entry (FIFO — first in, first out)

Priority Queues (*PQueues*) provide this via:

- `join`: insert item into PQueue with an associated priority (replacing `enqueue`)
- `leave`: remove item with highest priority (replacing `dequeue`)

Time complexity for naive implementation of a PQueue containing *N* items …

- $O(1)$ for `join` $O(N)$ for `leave`

Most efficient implementation ("heap") …

- $O(\log N)$ for `join`, `leave`

Other MST Algorithms

Boruvka's algorithm … complexity $O(E{\cdot}\log V)$

- the oldest MST algorithm
- start with *V* separate components
- join components using min cost links
- continue until only a single component

Karger, Klein, and Tarjan … complexity *O(E)*

- based on Boruvka, but non-deterministic
- randomly selects subset of edges to consider
- for the keen, here's the paper describing the algorithm

Shortest Path

Shortest Path

33/61

Path = sequence of edges in graph G $p = (v_0,v_1), (v_1,v_2), \ldots, (v_{m-1},v_m)$

cost(path) = sum of edge weights along path

Shortest path between vertices *s* and *t*

- a simple path *p(s,t)* where *s = first(p)*, *t = last(p)*
- no other simple path *q(s,t)* has *cost(q) < cost(p)*

Assumptions: weighted digraph, no negative weights.

Finding shortest path between two given nodes known as *source-target* SP problem

Variations: *single-source* SP, *all-pairs* SP

Applications: navigation, routing in data networks, …

Single-source Shortest Path (SSSP)

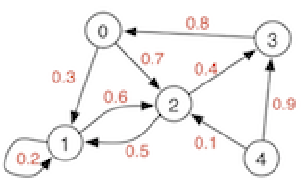
34/61

Given: weighted digraph *G*, source vertex *s*

Result: shortest paths from *s* to all other vertices

- `dist[]` *V*-indexed array of cost of shortest path from *s*
- `pred[]` *V*-indexed array of predecessor in shortest path from *s*

Example:



	0	1	2	3	4
dist	0	0.3	0.7	1.1	inf
pred	-	0	0	2	-

Shortest paths from s=0

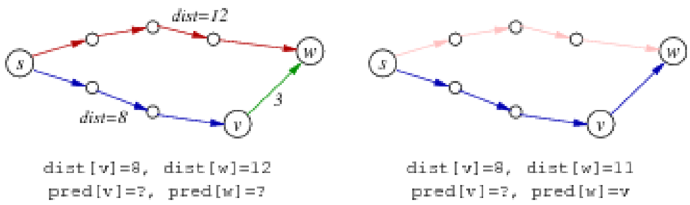
Edge Relaxation

35/61

Assume: `dist[]` and `pred[]` as above   (but containing data for shortest paths *discovered so far*)

`dist[v]` is length of shortest known path from *s* to *v*
`dist[w]` is length of shortest known path from *s* to *w*

Relaxation updates data for *w* if we find a shorter path from *s* to *w*:



Relaxation along edge *e*=`(v,w,weight)`:

- if **`dist[v]+weight < dist[w]`** then
  update `dist[w]:=dist[v]+weight` and `pred[w]:=v`

Dijkstra's Algorithm

One approach to solving single-source shortest path problem …

Data: *G*, *s*, `dist[]`, `pred[]` and

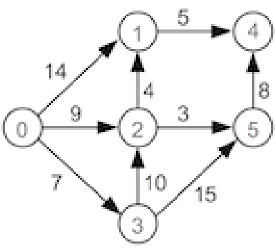
- *vSet*: set of vertices whose shortest path from *s* is unknown

Algorithm:

```
dist[]  // array of cost of shortest path from s
pred[]  // array of predecessor in shortest path from s

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise dist[] to all ∞, except dist[source]=0
|  initialise pred[] to all -1
|  vSet=all vertices of G
|  while vSet≠∅ do
|  |  find s∈vSet with minimum dist[s]
|  |  for each (s,t,w)∈edges(G) do
|  |     relax along (s,t,w)
|  |  end for
|  |  vSet=vSet\{s}
|  end while
```

Exercise #5: Dijkstra's Algorithm

Show how Dijkstra's algorithm runs on (source node = 0):



---

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–

dist	0	14	9	7	∞	∞
pred	–	0	0	0	–	–

dist	0	14	9	7	∞	22
pred	–	0	0	0	–	3

dist	0	13	9	7	∞	12
pred	–	2	0	0	–	2

dist	0	13	9	7	20	12
pred	–	2	0	0	5	2

dist	0	13	9	7	18	12
pred	–	2	0	0	1	2

... Dijkstra's Algorithm

Why Dijkstra's algorithm is correct:

Hypothesis.
(a) For visited *s* … *dist[s]* is shortest distance from source
(b) For unvisited *t* … *dist[t]* is shortest distance from source *via visited nodes*

Proof.
Base case: no visited nodes, *dist[source]=0*, *dist[s]=∞* for all other nodes

Induction step:

1. If *s* is unvisited node with minimum *dist[s]*, then *dist[s]* is shortest distance from source to *s*:
    - if ∃ shorter path via only visited nodes, then *dist[s]* would have been updated when processing the predecessor of *s* on this path
    - if ∃ shorter path via an unvisited node *u*, then *dist[u]<dist[s]*, which is impossible if *s* has min distance of all unvisited nodes
2. This implies that (a) holds for *s* after processing *s*
3. (b) still holds for all unvisited nodes *t* after processing *s*:
    - if ∃ shorter path via *s* we would have just updated *dist[t]*
    - if ∃ shorter path without *s* we would have found it previously

---

... Dijkstra's Algorithm

Time complexity analysis …

Each edge needs to be considered once ⇒ *O(E)*.

Outer loop has *O(V)* iterations.

Implementing "`find s∈vSet with minimum dist[s]`"

1. try all `s∈vSet` ⇒ cost = *O(V)* ⇒ overall cost = $O(E + V^2) = O(V^2)$
2. using a PQueue to implement extracting minimum
    - can improve overall cost to *O(E + V·log V)* (for best-known implementation)

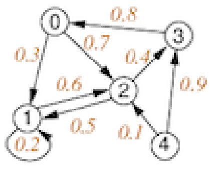
All-pair Shortest Path (APSP)

Given: weighted digraph G

Result: shortest paths between all pairs of vertices

- `dist[][]` $V \times V$-indexed matrix of cost of shortest path from v_{row} to v_{col}
- `path[][]` $V \times V$-indexed matrix of next node in shortest path from v_{row} to v_{col}

Example:



Weighted Digraph

dist

V	0	1	2	3	4
0	0	0.3	0.7	1.1	inf
1	1.8	0	0.6	1.0	inf
2	1.2	0.5	0	0.4	inf
3	0.8	1.1	1.5	0	inf
4	1.3	0.6	0.1	0.5	0

path

V	0	1	2	3	4
0	–	1	2	2	–
1	2	–	2	2	–
2	3	1	–	3	–
3	0	0	0	–	–
4	2	2	2	2	–

Shortest paths between all vertices

Floyd's Algorithm

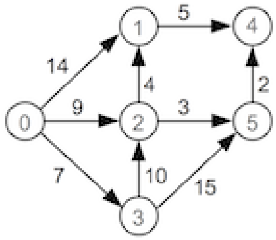
One approach to solving all-pair shortest path problem…

Data: G, `dist[][]`, `path[][]` Algorithm:

```
dist[][]  // array of cost of shortest path from s to t
path[][]  // array of next node after s on shortest path from s to t

floydAPSP(G):
|  Input graph G
|
|  initialise dist[s][t]=0 for each s=t
|                       =w for each (s,t,w)∈edges(G)
|                       =∞ otherwise
|  initialise path[s][t]=t for each (s,t,w)∈edges(G), otherwise to -1
|  for all i∈vertices(G) do
|     for all s∈vertices(G) do
|        for all t∈vertices(G) do
|           if dist[s][i]+dist[i][t] < dist[s][t] then
|              dist[s][t]=dist[s][i]+dist[i][t]
|              path[s][t]=path[s][i]
|           end if
|        end for
|     end for
|  end for
```

Exercise #6: Floyd's Algorithm

Show how Floyd's algorithm runs on:



After 1st iteration `i=0`: unchanged

After 2nd iteration `i=1`:

dist	[0]	[1]	[2]	[3]	[4]	[5]	path	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	14	9	7	∞	∞	[0]	–	1	2	3	–	–
[1]	∞	0	∞	∞	5	∞	[1]	–	–	–	–	4	–
[2]	∞	4	0	∞	9	3	[2]	–	1	–	–	1	5
[3]	∞	∞	10	0	∞	15	[3]	–	–	2	–	–	5
[4]	∞	∞	∞	∞	0	∞	[4]	–	–	–	–	–	–
[5]	∞	∞	∞	∞	2	0	[5]	–	–	–	–	4	–

After 3rd iteration `i=2`:

dist	[0]	[1]	[2]	[3]	[4]	[5]	path	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	13	9	7	18	12	[0]	–	2	2	3	2	2
[1]	∞	0	∞	∞	5	∞	[1]	–	–	–	–	4	–
[2]	∞	4	0	∞	9	3	[2]	–	1	–	–	1	5
[3]	∞	14	10	0	19	13	[3]	–	2	2	–	2	2
[4]	∞	∞	∞	∞	0	∞	[4]	–	–	–	–	–	–
[5]	∞	∞	∞	∞	2	0	[5]	–	–	–	–	4	–

After 4th iteration `i=3`: unchanged

After 5th iteration `i=4`: unchanged

After 6th iteration `i=5`:

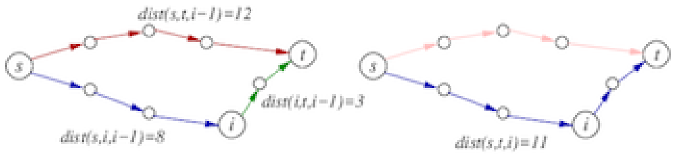
dist	[0]	[1]	[2]	[3]	[4]	[5]	path	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	13	9	7	14	12	[0]	–	2	2	3	2	2
[1]	∞	0	∞	∞	5	∞	[1]	–	–	–	–	4	–
[2]	∞	4	0	∞	5	3	[2]	–	1	–	–	5	5
[3]	∞	14	10	0	15	13	[3]	–	2	2	–	2	2
[4]	∞	∞	∞	∞	0	∞	[4]	–	–	–	–	–	–
[5]	∞	∞	∞	∞	2	0	[5]	–	–	–	–	4	–

... Floyd's Algorithm

Why Floyd's algorithm is correct:

A shortest path from *s* to *t* using only nodes from {0,…,*i*} is the shorter of

- a shortest path from *s* to *t* using only nodes from {0,…,*i-1*}
- a shortest path from *s* to *i* using only nodes from {0,…,*i-1*}
  plus a shortest path from *i* to *t* using only nodes from {0,…,*i-1*}



Also known as Floyd-Warshall algorithm (can you see why?)

... Floyd's Algorithm

Cost analysis …

- initialising `dist[][],path[][]` $\Rightarrow O(E)$
- V iterations to update `dist[][],path[][]` $\Rightarrow O(V^3)$

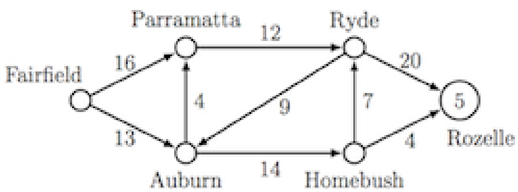
Time complexity of Floyd's algorithm: $O(V^3)$ (same as Warshall's algorithm for transitive closure)

Network Flow

Exercise #7: Merchandise Distribution

Lucky Cricket Company produces cricket balls in Fairfield and has a warehouse in Rozelle that stocks them.

To ship the cricket balls from the factory to the warehouse, Lucky Cricket leases space on trucks that have limited capacity:



What kind of algorithm would …

- help us find the maximum number of crates that can be shipped from Fairfield to Rozelle per day?

Flow Networks

Flow network …

- weighetd graph $G=(V,E)$
- distinct nodes $s\in V$ (*source*), $t\in V$ (*sink*)

Edge weights denote *capacities*

Applications:

- Distribution networks, e.g.
    - source: oil field
    - sink: refinery
    - edges: pipes
- Traffic flow

... Flow Networks

50/61

Flow in a network $G=(V,E)$ … nonnegative $f(v,w)$ for all vertices $v,w \in V$ such that

- $f(v,w) \le capacity$ for each edge $e=(v,w,capacity) \in E$
- $f(v,w)=0$ if no edge between v and w
- total flow *into* a vertex = total flow *out of* a vertex:

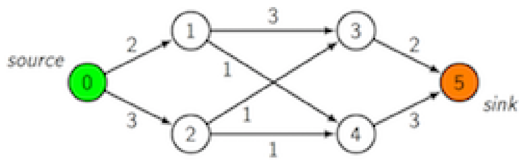
$$\sum_{x \in V} f(x,v) = \sum_{y \in V} f(v,y) \quad \text{for all } v \in V \setminus \{s,t\}$$

Maximum flow … no other flow from s to t has larger value

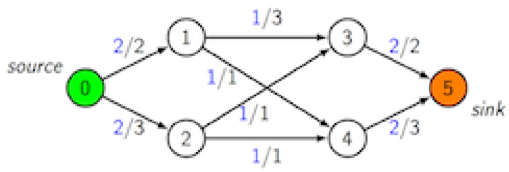
... Flow Networks

51/61

Example:



A (maximum) flow …



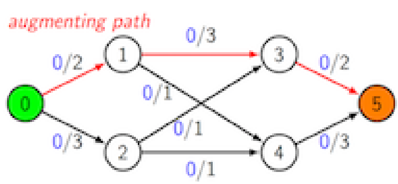
Augmenting Paths

52/61

Assume … $f(v,w)$ contains current flow

Augmenting path: any path from source s to sink t that can currently take more flow

Example:



Residual Network

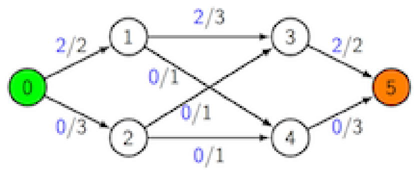
53/61

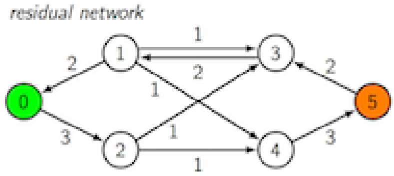
Assume … flow network $G=(V,E)$ and flow $f(v,w)$

Residual network (V,E'):

- same vertex set V
- for each edge $v \to^{c} w \in E$ …
  - $f(v,w) < c \quad \Rightarrow$ add edge $(v \to^{c\text{-}f(v,w)} w)$ to E'
  - $f(v,w) > 0 \quad \Rightarrow$ add edge $(v \leftarrow^{f(v,w)} w)$ to E'

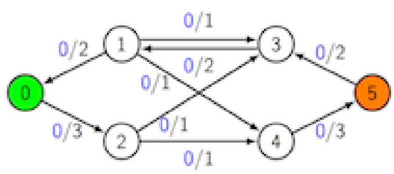
Example:





Exercise #8: Augmenting Paths and Residual Networks

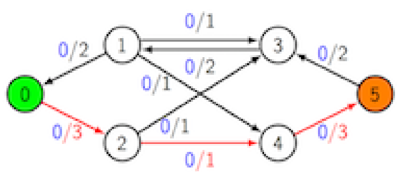
Find an augmenting path in:



and show the residual network after augmenting the flow

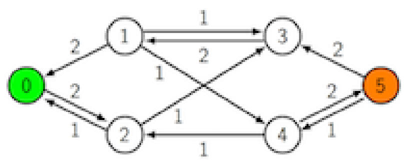
---

1. Augmenting path:



   maximum additional flow = 1

2. Residual network:



Can you find a further augmenting path in the new residual network?

Edmonds-Karp Algorithm

One approach to solving maximum flow problem …

maxflow(G):

1. Find a shortest augmenting path
2. Update `flow[][]` so as to represent residual graph
3. Repeat until no augmenting path can be found

... Edmonds-Karp Algorithm

Algorithm:

```
flow[][]  // V×V array of current flow
visited[] /* array of predecessor nodes on shortest path
             from source to sink in residual network */

maxflow(G):
|  Input  flow network G with source s and sink t
|  Output maximum flow value
|
|  initialise flow[v][w]=0 for all vertices v, w
|  maxflow=0
|  while ∃shortest augmenting path visited[] from s to t do
|     df = maximum additional flow via visited[]
|     // adjust flow so as to represent residual graph
|     v=t
|     while v≠s do
|        flow[visited[v]][v] = flow[visited[v]][v] + df;
|        flow[v][visited[v]] = flow[v][visited[v]] - df;
|        v=visited[v]
|     end while
|     maxflow=maxflow+df
|  end while
|  return maxflow
```

Shortest augmenting path can be found by standard BFS

... Edmonds-Karp Algorithm

Time complexity analysis ...

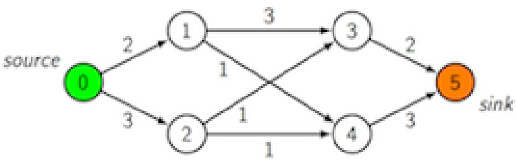
- *Theorem*. The number of augmenting paths needed is at most $V{\cdot}E/2$.
  ⇒ Outer loop has $O(V{\cdot}E)$ iterations.
- Finding augmenting path ⇒ $O(E)$.

Overall cost of Edmonds-Karp algorithm: $O(V{\cdot}E^2)$

Note: Edmonds-Karp algorithm is an implementation of general *Ford-Fulkerson method*

Exercise #9: Edmonds-Karp Algorithm

Show how Edmonds-Karp algorithm runs on:



---

flow	[0]	[1]	[2]	[3]	[4]	[5]	c-f	[0]	[1]	[2]	[3]	[4]	[5]
[0]	**0**	**0**	**0**	**0**	**0**	**0**	[0]	**–**	**2**	**3**	**–**	**–**	**–**
[1]	**0**	**0**	**0**	**0**	**0**	**0**	[1]	**–**	**–**	**–**	**3**	**1**	**–**
[2]	**0**	**0**	**0**	**0**	**0**	**0**	[2]	**–**	**–**	**–**	**1**	**1**	**–**
[3]	**0**	**0**	**0**	**0**	**0**	**0**	[3]	**–**	**–**	**–**	**–**	**–**	**2**
[4]	**0**	**0**	**0**	**0**	**0**	**0**	[4]	**–**	**–**	**–**	**–**	**–**	**3**
[5]	**0**	**0**	**0**	**0**	**0**	**0**	[5]	**–**	**–**	**–**	**–**	**–**	**–**

augmenting path: 0-1-3-5, `df`: 2

flow	[0]	[1]	[2]	[3]	[4]	[5]	c-f	[0]	[1]	[2]	[3]	[4]	[5]
[0]	**0**	**2**	**0**	**0**	**0**	**0**	[0]	**–**	**0**	**3**	**–**	**–**	**–**
[1]	**-2**	**0**	**0**	**2**	**0**	**0**	[1]	**2**	**–**	**–**	**1**	**1**	**–**
[2]	**0**	**0**	**0**	**0**	**0**	**0**	[2]	**–**	**–**	**–**	**1**	**1**	**–**
[3]	**0**	**-2**	**0**	**0**	**0**	**2**	[3]	**–**	**2**	**–**	**–**	**–**	**0**
[4]	**0**	**0**	**0**	**0**	**0**	**0**	[4]	**–**	**–**	**–**	**–**	**–**	**3**
[5]	**0**	**0**	**0**	**-2**	**0**	**0**	[5]	**–**	**–**	**–**	**2**	**–**	**–**

augmenting path: 0-2-4-5, `df`: 1

flow	[0]	[1]	[2]	[3]	[4]	[5]	c-f	[0]	[1]	[2]	[3]	[4]	[5]
[0]	**0**	**2**	**1**	**0**	**0**	**0**	[0]	**–**	**0**	**2**	**–**	**–**	**–**
[1]	**-2**	**0**	**0**	**2**	**0**	**0**	[1]	**2**	**–**	**–**	**1**	**1**	**–**
[2]	**-1**	**0**	**0**	**0**	**1**	**0**	[2]	**1**	**–**	**–**	**1**	**0**	**–**
[3]	**0**	**-2**	**0**	**0**	**0**	**2**	[3]	**–**	**2**	**–**	**–**	**–**	**0**
[4]	**0**	**0**	**-1**	**0**	**0**	**1**	[4]	**–**	**–**	**1**	**–**	**–**	**2**
[5]	**0**	**0**	**0**	**-2**	**-1**	**0**	[5]	**–**	**–**	**–**	**2**	**1**	**–**

augmenting path: 0-2-3-1-4-5, `df`: 1

flow	[0]	[1]	[2]	[3]	[4]	[5]	c-f	[0]	[1]	[2]	[3]	[4]	[5]
[0]	**0**	**2**	**2**	**0**	**0**	**0**	[0]	**–**	**0**	**1**	**–**	**–**	**–**
[1]	**-2**	**0**	**0**	**1**	**1**	**0**	[1]	**2**	**–**	**–**	**1**	**0**	**–**
[2]	**-2**	**0**	**0**	**1**	**1**	**0**	[2]	**2**	**–**	**–**	**0**	**0**	**–**
[3]	**0**	**-1**	**-1**	**0**	**0**	**2**	[3]	**–**	**1**	**1**	**–**	**–**	**0**
[4]	**0**	**-1**	**-1**	**0**	**0**	**2**	[4]	**–**	**1**	**1**	**–**	**–**	**1**
[5]	**0**	**0**	**0**	**-2**	**-2**	**0**	[5]	**–**	**–**	**–**	**2**	**2**	**–**

Summary

- Weighted graph representations
- Minimum Spanning Tree (MST)
  - Kruskal, Prim
- Shortest path problems
  - Dijkstra (single source SPP)
  - Floyd (all-pair SSP)
- Flow networks
  - Edmonds-Karp (maximum flow)

- Suggested reading (Sedgewick):
  - MST … Ch.20-20.4
  - SSP … Ch.21-21.3
  - Flow … Ch.22.1-22.2
