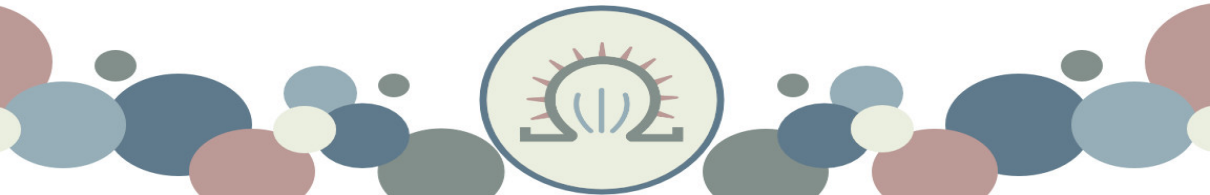
The Math Citadel

Graph Algorithms 1:
Tree, Path, Center

Rachel Traylor, Ph.D.
www.themathcitadel.com



Content

Introduction

Expectations and Rewards

Rewards

- ability to read new literature
- exposure to familiar algorithms in a different light
- exposure to new algorithms
- abstract one level up to identify types of problems
- look beyond implementation for commonalities

Expectations

- watched all previous graph lectures
- prepared to not understand everything immediately
- will finish/work out examples/exercises for yourself
- will keep your specific applications in mind

Maximum Branching

Maximum Branching Algorithms

- directed graphs
- Ideas due to Edmonds (1968)
- Variations on a "rumor monger" theme

Arborescence

a tree in a directed graph in which no two arcs are directed into the same vertex

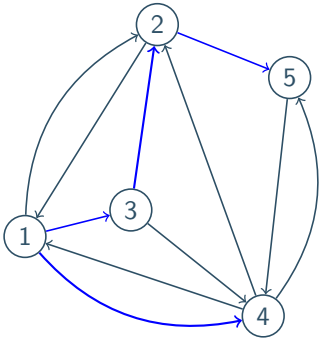


Figure: An aborescence

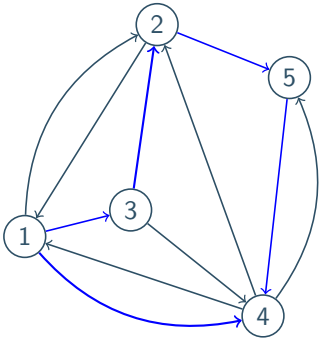


Figure: Tree, but not an aborescence

Arborescence

- For arc (i, j), we call i the "tail", and j the "head". (Some books reverse this.)
- Arborescences can have arcs that share the same tail, but not ones that share the same head.
- the *root* of an arborescence is the unique vertex that has no arcs directed into it.

Spanning Arborescence

an arborescence that is also a spanning tree

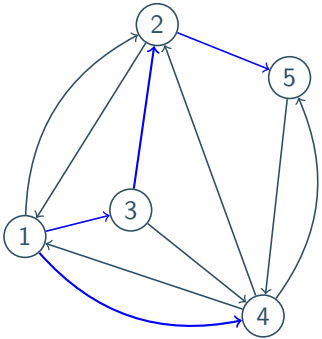


Figure: A spanning aborescence. Can you find another?

Branching

A *branching* is a forest (set of trees not necessarily connected) in which each tree is an arborescence. A *spanning branching* is a branching that includes all vertices.

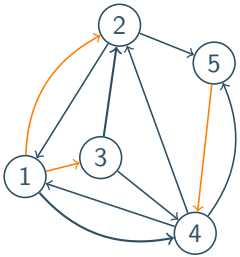


Figure: A spanning branching. All vertices belong to an arborescence, but this isn't an arborescence because this isn't one tree.

MBA: Basic Idea

- Greedy algorithm
- Searches for cheapest/most expensive arc into a node.
- Shifts costs onto receiving nodes and discounts all arcs accordingly.
- Collapses 0-cost cycles into a "black hole"
- Repeats on the subgraphs until a spanning aborescence is found or we end up with one black hole.
- Expands the black holes and deletes an appropriate edge of the cycle to keep a spanning arborescence.

MBA: Rationale

- the algorithm is greedy
- Collapsing cycles helps us "get it out of the way" while we work on dealing with other nodes
- We don't care about any cycles besides ones of cost 0, so the cost reduction/shifting helps us focus on only the arcs that will cause issues or should be included.

We'll go through a specific example, time permitting.

MBA: Uses

- new node comes up and wants to notify everyone of its presence
- node is faulty and needs to notify all other nodes
- you want to spread a rumor throughout the office as fast as possible, without someone hearing it twice

MBA: Variations

- For a maximum branching or spanning arborescence, look for the largest cost arc into each node instead of minimum cost node.
- If we want to force the arborescence to be rooted at a particular node, delete all arcs into the desired root and proceed.

Path Algorithms

Motivation

- Everyone is familiar with the shortest path problem between two nodes
- There are many variations on the "optimal path" problem
- There are pros and cons to different algorithms

Classes of Path Problems

Type of Problem	Problem Solved	
Existence	Connectivity	
Enumeration	⋆ Elementary Paths	⋆ Multicriteria Problems
Optimization	⋆ Path of Max Capacity	⋆ Path with min (max) arcs
	⋆ Shortest Path	⋆ Longest Path
	⋆ Path of Max Reliability	⋆ Reliability of a Network
	⋆ k shortest paths	⋆ η-optimal paths
Counting	counting paths	

Shortest Path Problems

- Given two specified vertices, find a shortest path between them.
- Given a specified vertex, find a shortest path between it and all other vertices (pairwise)
- Find a shortest path between all pairs of vertices.

We don't implement all these the same way.

Find a shortest path from source s.

General Algorithm Comparison

	Dijkstra ('59)	Ford ('56)/Bellman ('58)/Moore('57)
Notes	⋆ positive weights only	⋆ negative weights allowed
		⋆ no negative circuits
	⋆ digraphs or graphs	⋆ digraphs or graphs

Dijkstra's Algorithm: s to all

(1) All arcs and vertices are uncolored. Assign $d(x)$ to each vertex to denote the length of the shortest $s - x$ path that uses only colored vertices as hops.

(2) Set $d(s) = 0$, $d(x) = \infty$ for $x \neq s$. Set y as the most recently colored vertex.

(3) Color s. Set $y = s$.

(4) For each uncolored vertex x, redefine $d(x) = \min\{d(x), d(y) + w_{yx}\}$. Color the x with the smallest d. Set $y = x$.

(5) When all vertices have been colored, stop.

Note: If we wanted a specific $s - t$ path, we could stop as soon as the vertex t is colored.

Example: Dijkstra's algorithm

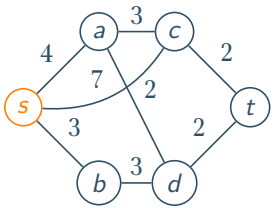


Figure: G

Initialize

Initially, $y = s$.

Vertex	Colored?	$d(x)$
s	C	0
a		∞
b		∞
c		∞
d		∞
t		∞

Example: Dijkstra

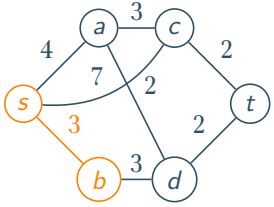


Figure: G

Step 1

$y = s$. $d(x) = \min\{\infty, d(s) + w_{sx}\}$

Vertex	Colored?	$d(x)$
s	C	0
a		$\min\{\infty, 0+4\} = 4$
b	C	$\min\{\infty, 0+3\} = 3$
c		$\min\{\infty, 0+7\} = 7$
d		$\min\{\infty, 0+\infty\} = \infty$
t		$\min\{\infty, 0+\infty\} = \infty$

Color b and arc (s, b) that produced it. Set $y = b$.

Example: Dijkstra

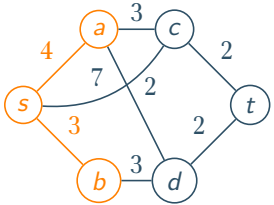


Figure: G

Step 2

$y = b$. $d(x) = \min\{d(x), d(b) + w_{bx}\}$

Vertex	Colored?	$d(x)$
s	C	0
a	C	$\min\{4, 3+\infty\} = 4$
b	C	3
c		$\min\{7, 3+\infty\} = 7$
d		$\min\{\infty, 3+3\} = 6$
t		$\min\{\infty, 3+\infty\} = \infty$

$d(a)$ is the minimum here. Color a and the arc that produced this length, which is (s, a). Set $y = a$.

Example: Dijkstra

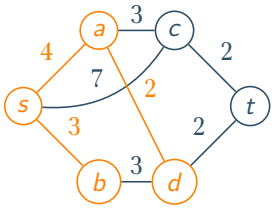


Figure: G

Step 3

$y = a$. $d(x) = \min\{d(x), d(b) + w_{ax}\}$

Vertex	Colored?	$d(x)$
s	C	0
a	C	4
b	C	3
c		$\min\{7, 4+3\} = 7$
d	C	$\min\{6, 4+2\} = 6$
t		$\min\{\infty, 4+\infty\} = \infty$

$d(d)$ is the minimum here. Color d and the arc that produced this length. Since both of arcs (b, d) and (a, d) produced it, choose one arbitrarily to color. Set $y = d$.

Example: Dijkstra

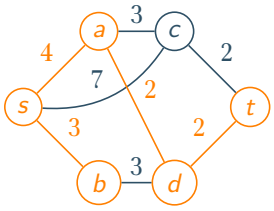


Figure: G

Step 4

$y = d$. $d(x) = \min\{d(x), d(d) + w_{dx}\}$

Vertex	Colored?	$d(x)$
s	C	0
a	C	4
b	C	3
c		$\min\{7, 6+\infty\} = 7$
d	C	6
t	C	$\min\{\infty, 6+2\} = 8$

$d(t)$ is the minimum here. Color t and the arc (d, t) that produced this length. If we only wanted an $S - t$ path, we could stop here. Otherwise, set $y = t$.

Example: Dijkstra

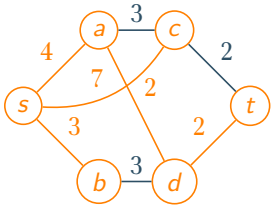


Figure: G

Step 5

$y = d$. $d(x) = \min\{d(x), d(d) + w_{dx}\}$

Vertex	Colored?	$d(x)$
s	C	0
a	C	4
b	C	3
c	C	$\min\{7, 8+2\} = 7$
d	C	6
t	C	$\min\{\infty, 6+2\} = 8$

Color c and arc (s, c) that produced this weight.

Since all vertices have been colored, the resulting stored arcs show us all the shortest paths from s to every other vertex.

Bellman-Moore-Ford: allowing negative weights

- If we allow negative weights on the arcs of a graph, then Dijkstra's algorithm is no longer guaranteed to produce a shortest path.
- Main difference between this and Dijkstra: we examine all vertices at each state, not just uncolored ones.
- It's possible for a colored vertex to have its "label" d decreased. If this happens, we uncolor the incident arc.
- The algorithm terminates when all vertices are colored and no labels d are decreased.

BMF: Example

We'll illustrate this a slightly different way than coloring – through the notion of "relaxing" and searching edges. Let $G = (V, E)$. Set $v.d$ as the distance from source vertex s to $v \in V$, $v.p$ as the predecessor for vertex v, w_{uv} is the weight of arc $(u, v) \in E$.The pseudocode for this algorithm is as follows:

```
for v ∈ V do
    v.d = ∞
    v.p = NULL
end for
s.d = 0
for i = 1,...,|V|-1 do
    for (u, v) ∈ E do
        if v.d > u.d + w_uv then
            v.d = u.d + w_uv
            v.p = u
        end if
    end for
end for
```

BMF: Example

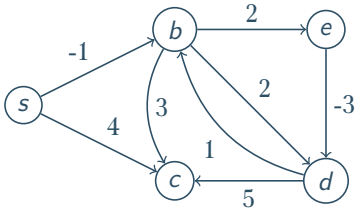


Figure: G

1: (s,b)	5: (b,e)
2: (s,c)	6: (d,c)
3: (b,c)	7: (d,b)
4: (b,d)	8: (e,d)

Table: Order of Edge Examination

BMF: Example

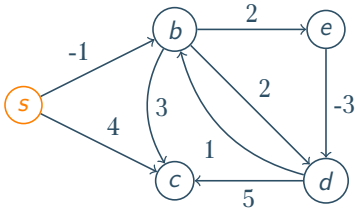


Figure: *G*

1: (s,b)	5: (b,e)
2: (s,c)	6: (d,c)
3: (b,c)	7: (d,b)
4: (b,d)	8: (e,d)

Table: Order of Edge Examination

i	S		B		C		D		E	
	a.d	*a.p*	*b.d*	*b.p*	*c.d*	*c.p*	*d.d*	*d.p*	*e.d*	*e.p*
0	0	N	∞	N	∞	N	∞	N	∞	N

BMF: Example

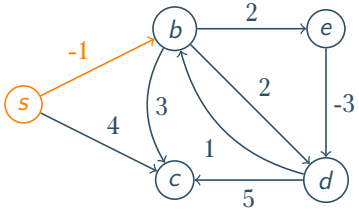


Figure: G

1: (s,b)	5: (b,e)
2: (s,c)	6: (d,c)
3: (b,c)	7: (d,b)
4: (b,d)	8: (e,d)

Table: Order of Edge Examination

$i = 1$ Examine (s, b).

$\infty = b.d > s.d + w_{sb} = 0 + -1 = -1$ TRUE
Set $b.d = -1$, $b.p = S$.

BMF: Example

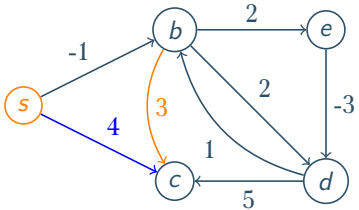


Figure: G

1: (s,b)	5: (b,e)
2: (s,c)	6: (d,c)
3: (b,c)	7: (d,b)
4: (b,d)	8: (e,d)

Table: Order of Edge Examination

Examine (s,c) and (b,c)

(s, c) yields $c.d = 4$, $c.p = S$. However, next we have for (b, c):
$4 = c.d > b.d + w_{bc} = -1 + 3 = 2$ (TRUE). So we update $c.d = 2$ and $c.p = b$.

BMF: Example

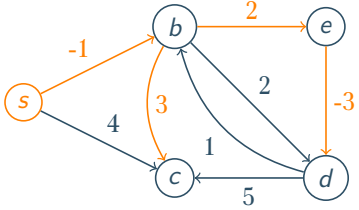


Figure: *G*

1: (s,b)	5: (b,e)
2: (s,c)	6: (d,c)
3: (b,c)	7: (d,b)
4: (b,d)	8: (e,d)

Table: Order of Edge Examination

i	S		B		C		D		E	
	a.d	*a.p*	*b.d*	*b.p*	*c.d*	*c.p*	*d.d*	*d.p*	*e.d*	*e.p*
0	0	N	∞	N	∞	N	∞	N	∞	N
1	0	N	-1	A	2	B	-2	E	1	B

BMF: Example

We continue in this fashion until we hit $i = 4$, which is our stopping point.

- You can also put in a break if all variables remain unchanged from i to $i + 1$, since the algorithm has converged early.
- Finish this out and note that $i = 2$ shows a convergence, so you could stop. Carry it out to $i = 4$ to convince yourself.

BMF: Uses and Remarks

- Better than Dijkstra for distributed systems
- Can be used to detect negative circuits (loops)
- Used in distance-vector routing protocols
- Scales poorly. There are updated versions.
- Topology changes not reflected quickly

Shortest Paths Pairwise

- If we want to find the shortest path between all pairs, this is a "bigger problem"
- Dijkstra and Bellman/Ford/Moore can be used repeatedly, but this is inefficient.
- Floyd (1962) and Dantzig (1967) exploit matrices.

Basic Ideas

- Number your vertices $1, \ldots, N$.
- Let d_{ij}^m be a shortest $i - j$ path length with only the first m vertices as possible intermediates. If none exists, $d_{ij}^m = \infty$.
- $d_{ij}^0 = w_{ij}$, and $d_{ii}^0 = 0$.
- d_{ij}^N is the length of a shortest $i - j$ path.
- Make a matrix $D^m = [d_{ij}^m]_{N \times N}$. We want D^N.
- We can calculate D^m from D^{m-1}:

$$d_{ij}^m = \min\{d_{im}^{m-1} + d_{mj}^{m-1}, d_{ij}^{m-1}\}$$

- The diagonals will always be 0.

Basic Ideas

$$d_{ij}^{m} = \min\{d_{im}^{m-1} + d_{mj}^{m-1}, d_{ij}^{m-1}\}$$

Why does this work?

We take our current known shortest $i - j$ path using the first $m - 1$ vertices, and see if adding vertex m to the path decreases the length. If so, add it. If not, don't.

Floyd's Shortest Path: Example

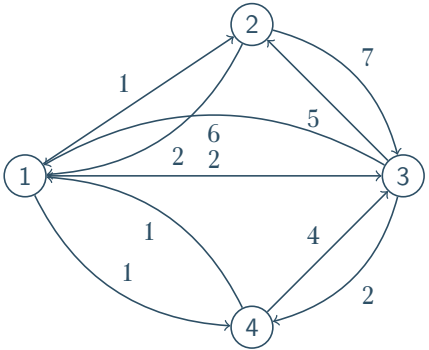


Figure: G

$$D^0 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 7 & \infty \\ 6 & 5 & 0 & 2 \\ 1 & \infty & 4 & 0 \end{bmatrix}$$

Floyd's Shortest Path: Example

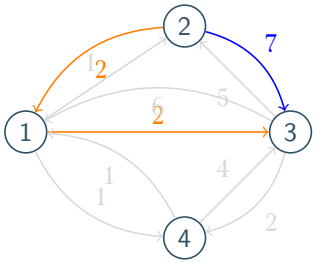


Figure: G

$$D^0 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 7 & \infty \\ 6 & 5 & 0 & 2 \\ 1 & \infty & 4 & 0 \end{bmatrix}$$

$d_{ij}^1 = \min\{d_{i1}^0 + d_{1j}^0, d_{ij}^0\}$

$d_{23}^1 = \min\{d_{21}^0 + d_{13}^0, d_{23}^0\} = \min\{2+2, 7\} = 4$ Therefore, it's cheaper to get from 2 to 3 by going through 1.

Floyd's Shortest Path: Example

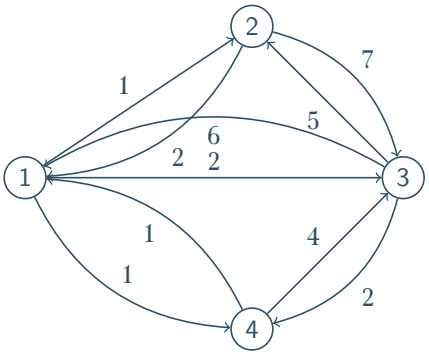


Figure: G

$$D^0 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 7 & \infty \\ 6 & 5 & 0 & 2 \\ 1 & \infty & 4 & 0 \end{bmatrix}$$

$$D^1 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 4 & 3 \\ 6 & 5 & 0 & 2 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

$$P^1 = \begin{bmatrix} X & (1,2) & (1,3) & (1,4) \\ (2,1) & X & (2,1,3) & (2,1,4) \\ (3,1) & (3,2) & X & (3,4) \\ (4,1) & (4,1,2) & (4,1,3) & X \end{bmatrix}$$

Computing D^2, D^3, and D^4 will yield the same results (with the corresponding path matrices P), so we converged quickly. We're done at D^4.

Notes and Remarks

- To determine the actual paths, there are more efficient methods than storing the results in a matrix.
- Negative weights are allowed.
- Largest pairwise paths: Switch min to max.
- Dantzig's version: Grow matrices by one row/column until you hit $N \times N$ matrix.
- Floyd & Dantzig perform the same number of calculations.

Uses for Shortest Path Problems

- Static Routing (OSPF, BGP, distance-vector routing)
- min-delay path

Variations of Shortest Path Problems

Bottleneck Problem

- **bottleneck**: smallest arc weight in an $i - j$ path
- Find a path between two vertices with the largest bottleneck.

How do we do it?

- Floyd's algorithm with modifications.
- If no arc (i, j) exists, $d_{ij}^0 = -\infty$, not ∞
- Substitute min for $+$, and max for min. Then

$$d_{ij}^m = \max(\min(d_{im}^{m-1}, d_{mj}^{m-1}), d_{ij}^{m-1})$$

- Does rerouting through vertex m increase our bottleneck size? If so, route through. If not, don't.

Floyd's Algorithm: Bottleneck

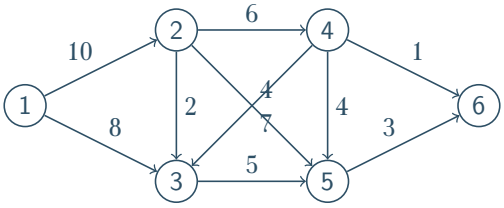


Figure: G

$$D^0 = \begin{bmatrix} 0 & 10 & 8 & -\infty & -\infty & -\infty \\ -\infty & 0 & 2 & 6 & 4 & -\infty \\ -\infty & -\infty & 0 & -\infty & 5 & -\infty \\ -\infty & -\infty & 7 & 0 & 4 & 1 \\ -\infty & -\infty & -\infty & -\infty & 0 & 3 \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix} = D^1$$

Floyd's Algorithm: Bottleneck

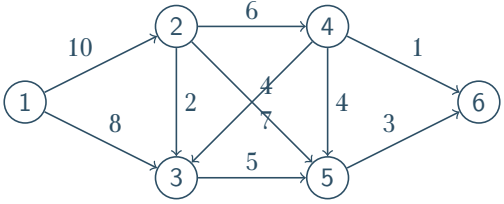


Figure: G

$$D^2 = \begin{bmatrix} 0 & 10 & 8 & 6 & 4 & -\infty \\ -\infty & 0 & 2 & 6 & 4 & -\infty \\ -\infty & -\infty & 0 & -\infty & 5 & -\infty \\ -\infty & -\infty & 7 & 0 & 4 & 1 \\ -\infty & -\infty & -\infty & -\infty & 0 & 3 \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix}$$

$$d^2_{14} = \max(\min(d^1_{12}, d^1_{24}), d^1_{14}) = \max(\min(10, 6), -\infty) = 6$$

Floyd's Algorithm: Bottleneck

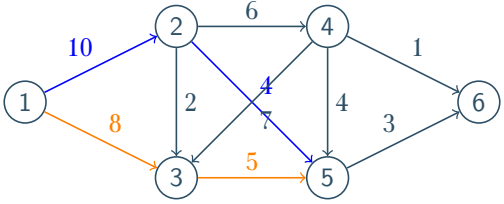


Figure: G

$$D^3 = \begin{bmatrix} 0 & 10 & 8 & 6 & 5 & -\infty \\ -\infty & 0 & 2 & 6 & 4 & -\infty \\ -\infty & -\infty & 0 & -\infty & 5 & -\infty \\ -\infty & -\infty & 7 & 0 & 5 & 1 \\ -\infty & -\infty & -\infty & -\infty & 0 & 3 \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix}$$

$$d^3_{15} = \max(\min(d^2_{13}, d^2_{35}), d^2_{15}) = \max(\min(8, 5), 4) = 5$$

Floyd's Algorithm: Bottleneck

Continuing to D^6 in a similar fashion,

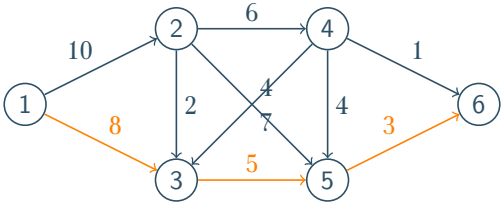


Figure: G

$$D^6 = \begin{bmatrix} 0 & 10 & 8 & 6 & 5 & 3 \\ -\infty & 0 & 6 & 6 & 4 & 3 \\ -\infty & -\infty & 0 & -\infty & 5 & 3 \\ -\infty & -\infty & 7 & 0 & 5 & 3 \\ -\infty & -\infty & -\infty & -\infty & 0 & 3 \\ -\infty & -\infty & -\infty & -\infty & -\infty & 0 \end{bmatrix}$$

k-shortest paths

- We now want to know the *k* shortest paths, perhaps for alternatives.
- We can generalize Floyd's algorithm to return the lengths of the *k* shortest paths.
- Double-sweep algorithm (Shier, 1974/76), where each entry of the matrix is a vector of length *k* tracking the *k* shortest paths
- Used for routing in optical mesh networks, or for finding alternate paths
- Example will be published separately for those interested.

Center Problems

Why do we want to find a "center"?

- Typically we wish to locate a new node of some kind in the network

Minimize the Max Distance

We could minimize the maximum distance from any other node to this new node.
Example: We want to locate a new school that is no more than 20 miles away from the most distant house in the district.

Minimize the Sum of Distances

We could minimize the total distance from all nodes to this new node.
Example: We want to locate a new node in a network such that the total travel delay from this node to all other nodes is as small as possible.

These two objectives may not yield the same result.

Other considerations

- Must the "center" be an existing node?
- May we locate the center on some point along an arc?
- May we locate the center anywhere in the enclosed area of the graph? (won't cover)

Center of a graph

vertex-vertex distance

the length of the shortest path between two vertices

- Find by: Any shortest path algorithm

center of a graph

Any existing vertex $x \in V(G)$ such that the maximum distance from x to any other vertex j is as small as possible.

$$\max_{j \in V(G)} (d(x,j)) = \min_{i \in V(G)} \left(\max_{j \in V(G)} (d(i,j)) \right)$$

Center of a Graph: Example

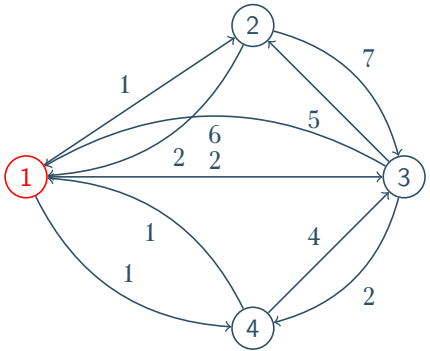


Figure: G

$$D^1 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 4 & 3 \\ 6 & 5 & 0 & 2 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

Find the row with the smallest maximum.

Median of a Graph

Median

An existing vertex $x \in V(G)$ such that the sum of the distances from x to all other vertices is as small as possible.

$$\sum_{j \in V(G)} d(x,j) = \min_{i \in V(G)} \{ \sum_{j \in V(G)} d(i,j) \}$$

- Find by: Any shortest path algorithm

Median of a Graph: Example

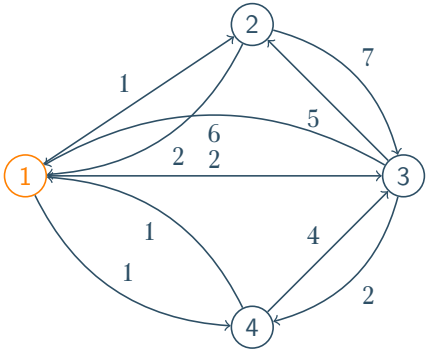


Figure: G

$$D^4 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 4 & 3 \\ 6 & 5 & 0 & 2 \\ 1 & 2 & 3 & 0 \end{bmatrix} \quad S = \begin{bmatrix} 4 \\ 9 \\ 13 \\ 6 \end{bmatrix}$$

Vertex whose row has the smallest sum.

General Center

f-point or point of arc (i, j)

a location on (i, j) that is fraction f from vertex i and $1 - f$ from vertex j, for $0 \leq f \leq 1$

interior point

point on an arc that is not a vertex

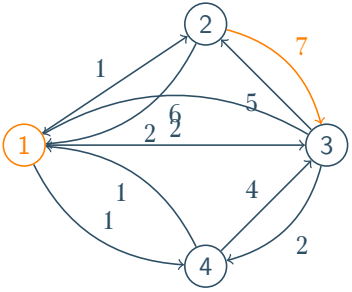
Example: If (i, j) has weight 10 (perhaps delineating miles), then the 0.4-point is located at the 4 mile mark, 40% of the way along the arc.

General Center

vertex-arc distance

the maximum of the shortest distance from a vertex j to a point on arc (r, s)

$$d'(j,(r,s)) = \begin{cases} \frac{d(j,r)+d(j,s)+w_{rs}}{2}, \text{undirected} \\ d(j,r) + w_{rs}, \text{directed} \end{cases}$$



$$D^4 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 2 & 0 & 4 & 3 \\ 6 & 5 & 0 & 2 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

$$d'(1,(2,3)) = d(1,2) + w_{23} = 1 + 7 = 8$$

(since $(2,3)$ is directed)

vertex-arc distance

$$d'(j,(r,s)) = \begin{cases} \frac{d(j,r)+d(j,s)+w_{rs}}{2}, & \text{(r,s) undirected} \\ d(j,r) + w_{rs}, & \text{(r,s) directed} \end{cases}$$

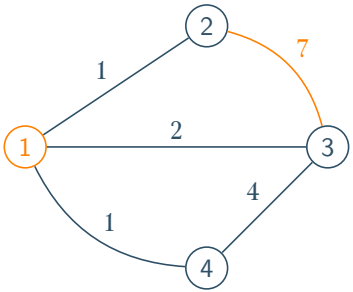


Figure: G

$$D^4 = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 1 & 0 & 3 & 2 \\ 2 & 3 & 0 & 4 \\ 1 & 2 & 4 & 0 \end{bmatrix}$$

$$d'(1,(2,3)) = \frac{d(1,2) + d(1,3) + w_{23}}{2}$$
$$= \frac{1+2+7}{2} = 5$$

(since $(2,3)$ is *un*directed)

General Center: Example

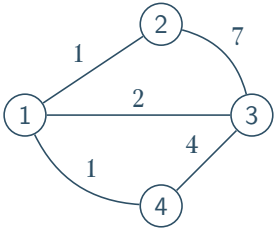


Figure: G

Index	Arc	Index	Arc
1	(1,2)	4	(2,3)
2	(1,3)	5	(3,4)
3	(1,4)		

Make a matrix D' with rows as vertex numbers, and columns as arc numbers. Each entry is the vertex-arc distance.

$$D' = \begin{bmatrix} 1 & 2 & 1 & 5 & 3.5 \\ 1 & 3 & 2 & 5 & 4.5 \\ 3 & 2 & 3.5 & 5 & 4 \\ 2 & 3.5 & 1 & 6.5 & 4 \end{bmatrix}$$

We can have more than one general center. Here it's vertices 1,2, and 3.

General Median

general median

an existing vertex $x \in V(G)$ such that the sum of all vertex-arc distances is as small as possible.

$$\sum_{\text{all arcs}} d'(x,(r,s)) = \min_{i \in V(G)} \left(\sum_{\text{all arcs}} d'(i,(r,s)) \right)$$

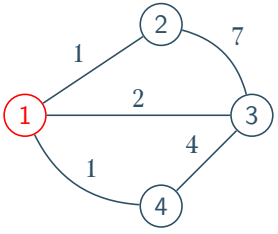


Figure: G

$$D' = \begin{bmatrix} 1 & 2 & 1 & 5 & 3.5 \\ 1 & 3 & 2 & 5 & 4.5 \\ 3 & 2 & 3.5 & 5 & 4 \\ 2 & 3.5 & 1 & 6.5 & 4 \end{bmatrix} \quad S = \begin{bmatrix} 12.5 \\ 15.5 \\ 17.5 \\ 17 \end{bmatrix}$$

Absolute Center

point-vertex distance

$$\tilde{d}(f-(r,s),j) = \begin{cases} \min\{fw_{rs} + d(r,j), (1-f)w_{rs} + d(s,j)\}, & \text{(r,s) undirected} \\ (1-f)w_{rs} + d(s,j), & \text{(r,s) directed} \end{cases}$$

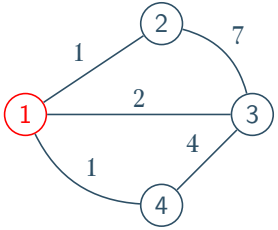


Figure: G

The point-vertex distance from the 0.5-point of arc (2,3) to vertex 4 is

$$\begin{aligned} \tilde{d}(0.5-(2,3),4) &= \min\{0.5 \cdot 7 + d(2,4), 0.5 \cdot 7 + d(3,4)\} \\ &= \min\{3.5+2, 3.5+4\} \\ &= 5.5 \end{aligned}$$

Absolute Center

absolute center

any f-point on any existing arc $(r, s) \in E(G)$ whose furthest vertex is as close as possible measured by point-vertex distance

Useful fact

No interior point of a directed arc can be an absolute center.

How we find it

- Hakimi (1964) - pictorial method
- Cuninghame-Green (1984) - global minimization of piecewise linear non-convex functions $\rightarrow$ combinatorial problem
  - Mathematical methods are beyond this talk

Absolute Median

absolute median

any $f-$point on any arc (r, s) in the graph such that the sum of the point-vertex distances to all vertices is as small as possible

How to Find: Minieka (1977), Hakimi (1964)

General Absolute Center & Median

- The general absolute center is a point whose maximum distance to any other point is as small as possible
- We define a point-point distance. (Not covered)
- The general absolute median is a point whose sum of point-point distances is as small as possible.
- Much more difficult problem

Summary

Location	Type	Distance Metric	Criteria
Center	vertex	vertex-vertex	smallest maximum
Median	Vertex	vertex-vertex	smallest sum
General Center	Vertex	vertex-arc	smallest maximum
General Median	Vertex	vertex-arc	smallest sum
Absolute Center	f-point	point-vertex	smallest maximum
Absolute Median	f-point	point-vertex	smallest sum
General Absolute Center	f-point	point-point	smallest maximum
General Absolute Median	f-point	point-point	smallest sum

Uses

Location	Example Use
Center	Put a post office in an existing town with the max distance to other towns as small as possible
Median	Put a post office in an existing town that minimizes the total mileage driven from the post office to the other towns.
General center	Open a tow-truck facility in an existing town such that the max distance driven to anywhere on the graph is as small as possible
General median	Locate a warehouse in an existing town such that the total delivery distance driven to anywhere on the graph is as small as possible

Uses

Location	Example Use
Absolute center	Locate a highway patrol station anywhere such that the max distance from the station to any town jail is as small as possible.
Absolute median	Locate highway patrol station anywhere such that the max total patrol distance out to each town is as small as possible.
General Absolute Center	Open a tow-truck facility anywhere so max distance driven to anywhere on the graph is as small as possible
General Absolute Median	Locate warehouse anywhere so total delivery distance driven to anywhere on the graph is as small as possible

Conclusion

- We've opened up a world of more general ideas about graph algorithms.
- Begin trying to see your day-to-day tasks/challenges in networking in terms of graph problems rather than specific protocols or branded acronyms.

Selected References

- *Optimization Algorithms for Networks and Graphs*, Minieka (1984)
- `https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04DemoEdmondsBranching.pdf`
- *Graphs and Algorithms*, Gondran and Minoux (1984)
- *Computer Networks*, Tannenbaum (1981)
- *Network Routing: Algorithms, Protocols, and Architectures*, Medhi and Ramasamy (2007)

For the Curious

Finding Spanning Trees By Coloring

Basic Idea

- Examine edges in some arbitrary sequence and choose to either include or exclude (coloring)
- Blue: included, Orange: excluded
- An edge is excluded if it forms a cycle with existing included edges. Otherwise, it's included
- The algorithm terminates when either
  - $n-1$ edges are colored blue, where n is the number of vertices
  - all edges are colored, but $k < n-1$ are colored blue (no spanning tree)

Finding Spanning Trees By Coloring

Algorithm

Initialize: All edges are uncolored, and all buckets are empty.

(1) Select a non-loop edge, color it blue, and place its endpoints in a bucket.

(2) Select the next uncolored non-loop edge, and do one of the following:

- (a) Both endpoints of the new edge are in the same bucket: Excluded
- (b) One endpoint in the existing bucket, other in no bucket: Included. Assign bucketless endpoint to existing bucket.
- (c) neither endpoint in a bucket: Included. Assign both endpoints to a new bucket.
- (d) each endpoint in a different bucket: Included. Combine buckets.

Finding Spanning Trees By Coloring

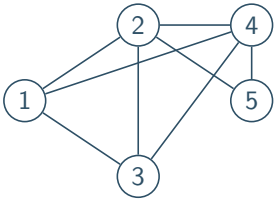


Figure: G

Step	Edge	Color	Bucket 1	Bucket 2
0			Empty	Empty

Finding Spanning Trees By Coloring

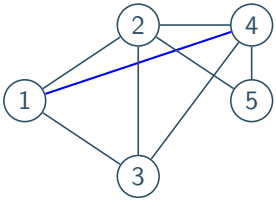


Figure: G

Step	Edge	Color	Bucket 1	Bucket 2
0			Empty	Empty
1	(1,4)	B	1,4	Empty

Finding Spanning Trees By Coloring

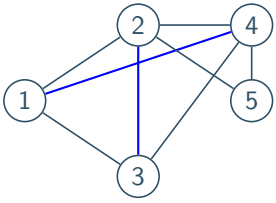


Figure: G

Step	Edge	Color	Bucket 1	Bucket 2
0			Empty	Empty
1	(1,4)	B	1,4	Empty
2	(2,3)	B	1,4	2,3

Finding Spanning Trees By Coloring

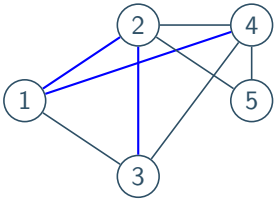


Figure: G

Step	Edge	Color	Bucket 1	Bucket 2
0			Empty	Empty
1	(1,4)	B	1,4	Empty
2	(2,3)	B	1,4	2,3
3	(1,2)	B	1,2,3,4	Empty

Finding Spanning Trees By Coloring

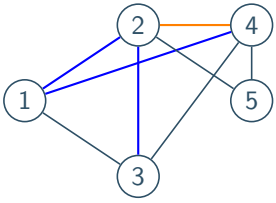


Figure: *G*

Step	Edge	Color	Bucket 1	Bucket 2
0			Empty	Empty
1	(1,4)	B	1,4	Empty
2	(2,3)	B	1,4	2,3
3	(1,2)	B	1,2,3,4	Empty
4	(2,4)	O	1,2,3,4	Empty

Finding Spanning Trees By Coloring

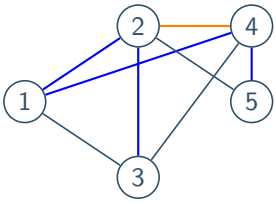


Figure: G

Step	Edge	Color	Bucket 1	Bucket 2
0			Empty	Empty
1	(1,4)	B	1,4	Empty
2	(2,3)	B	1,4	2,3
3	(1,2)	B	1,2,3,4	Empty
4	(2,4)	O	1,2,3,4	Empty
5	(4,5)	B	1,2,3,4,5	Empty

Terminate Algorithm. 4 edges have been colored blue.

Finding Spanning Trees By Coloring

Rationale

As edges are colored, they will form clumps of connected components. The vertices in a connected component belong to a single bucket.

- If we try to add an edge whose endpoints are already in the same bucket, we're adding a cycle.
- If an edge has one endpoint in one bucket, and one endpoint in another bucket, then we've linked two separate connected components into one big connected component, and can dump the two buckets together.
- If an edge has one endpoint in a bucket, and another endpoint lost and bucketless, then we've added a dangling endpoint to that connected component.

Modification: Minimum and Maximum Spanning Trees

- If all the edges are weighted equally, the ordering of edge examination is arbitrary.
- Ordering the edges for inspection in ascending order: Minimum Spanning Tree
- Ordering the edges for inspection in descending order: Maximum Spanning Tree.
- This is Kruskal's Algorithm.
- Prim's algorithm builds the spanning tree by purposely selecting edges incident with currently blue ones, but basically works the same way.

Example: Spanning Tree of Maximum Reliability

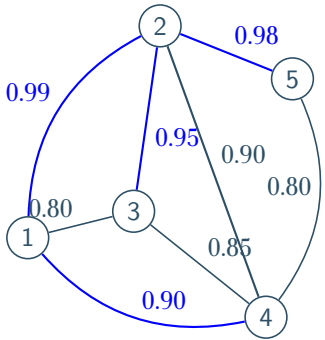


Figure: G

Step	Edge	Color	Bucket 1	Bucket 2
0			Empty	Empty
1	(1,2)	B	1,2	Empty
2	(2,5)	B	1,2,5	Empty
3	(2,3)	B	1,2,3,5	Empty
4	(1,4)	B	1,2,3,4,5	Empty

Terminate Algorithm. 4 edges have been colored blue.

Maximum Branching Algorithm

We'll examine this one through an example, then look at what makes it work. Assume the arc weights are the costs in terms of delay of packet transmission, measured in ms. We want the smallest spanning arborescence for this graph, where the total delay in ms of the arborescence is lowest.

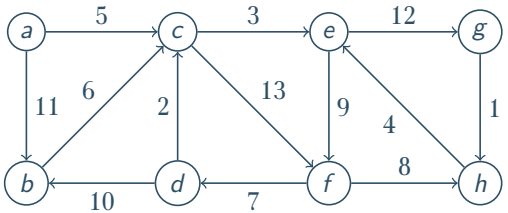


Figure: Our sample graph G

Step 1: Find the cheapest edge *entering* each node

Node	Entering Edge	Cost	Node	Entering Edge	Cost
a	$\emptyset$	0	e	(c,e)	3
b	(d,b)	10	f	(e,f)	9
c	(d,c)	2	g	(e,g)	12
d	(f,d)	7	h	(g,h)	1

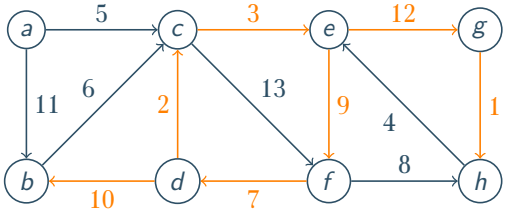


Figure: Our sample graph G

Step 2: Assign this cost to the nodes

Node	Entering Edge	Cost	Node	Entering Edge	Cost
a	$\emptyset$	0	e	(c,e)	3
b	(d,b)	10	f	(e,f)	9
c	(d,c)	2	g	(e,g)	12
d	(f,d)	7	h	(g,h)	1

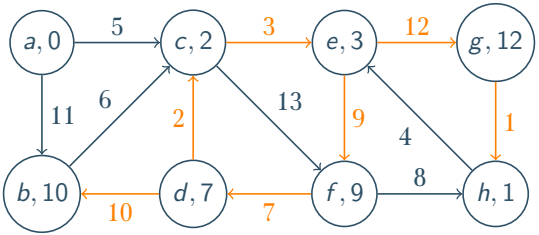


Figure: Our sample graph, G

Step 3: Subtract the node cost from all arcs directed into each node

Example: Arc (a, b) reduces cost from 11 to 1, because 10 of that cost is shifted to node b.

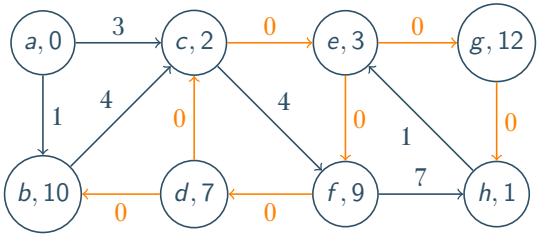


Figure: Our sample graph G

Step 4: Look for a 0 cost cycle and collapse

(c,e,f,d) is a cycle of 0 cost. This is bad.

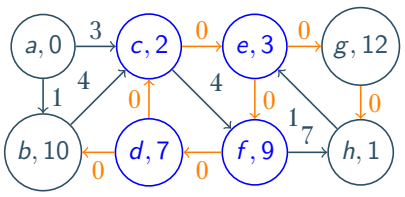


Figure: Our sample graph G

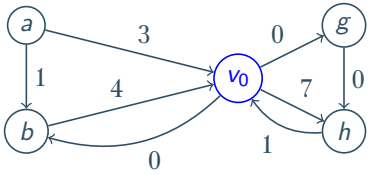


Figure: Contracted graph. The cycle collapses into a vertex, (c,f) disappears, and all other edges are directed as we expect.

MBA: What to do with the retracted graph

We now perform Steps 1-4 on our contracted graph G'

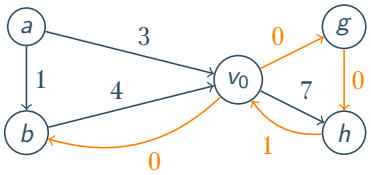


Figure: The lowest cost arcs entering each node are in orange.

MBA: What to do with the retracted graph

We now perform Steps 1-4 on our contracted graph G'

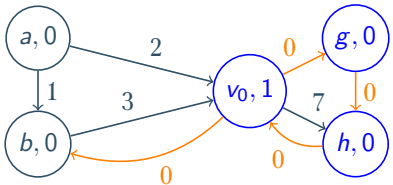


Figure: We reduced the cost of the arcs after shifting. We see a 0 cost cycle again, so we have to contract one more time.

MBA: And retract yet again

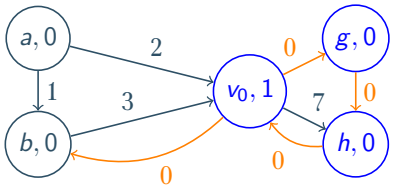


Figure: G'

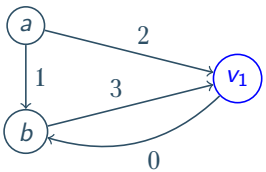


Figure: The second contraction, G''

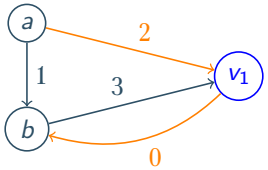


Figure: Oh hey, after finding the cheapest edges, we found an arborescence. We can stop contracting!

Now what? We have to expand the graph back, and get rid of edges that cause cycles in an intelligent way.

Uncontract v_1 back into its cycle

But, we get rid of the edge in the cycle that will result in more than one arc directed into a node. In this case, (h, v_0) is eliminated from being "put back".

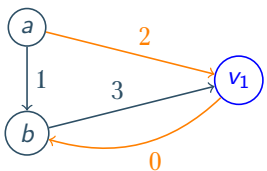


Figure: G''

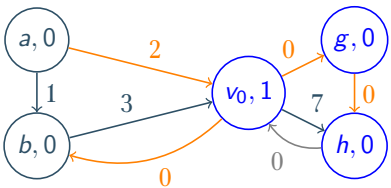


Figure: We don't take (h, v_0).

Uncontract v_0 back into its cycle

But, we get rid of the edge in the cycle that will result in more than one arc directed into a node. In this case, (d, c) is eliminated from being "put back".

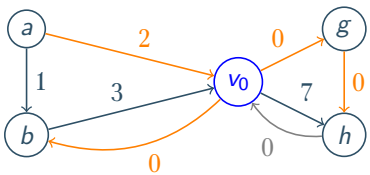


Figure: We don't take (h, v_0).

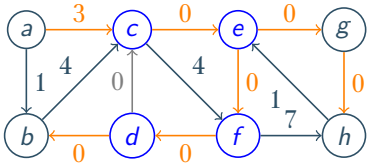


Figure: Keep all the original edges of the cycle (c,d,e,f) except (d,c), because we already have (a,c).

MBA: Put the original costs back

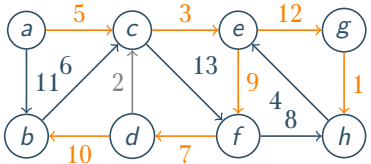


Figure: Our minimum spanning arborescence is in orange, with total cost as $C = 5 + 3 + 12 + 1 + 9 + 7 + 10 = 47$