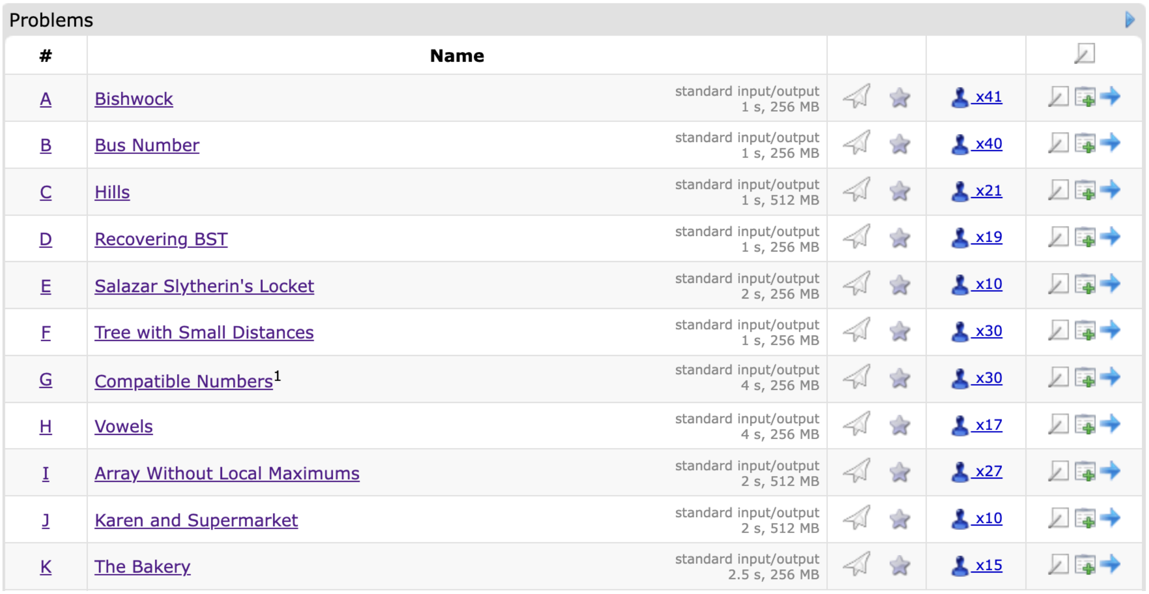
Lecture 2:
Matchings and Flows

Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794–4400

http://www.cs.stonybrook.edu/~skiena

Winner: Kanye East (11 problems, 890 minutes)



Problems

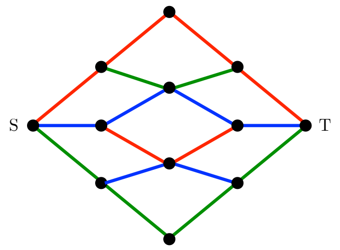
#	Name				
A	Bishwock	standard input/output 1 s, 256 MB		x41	
B	Bus Number	standard input/output 1 s, 256 MB		x40	
C	Hills	standard input/output 1 s, 512 MB		x21	
D	Recovering BST	standard input/output 1 s, 256 MB		x19	
E	Salazar Slytherin's Locket	standard input/output 2 s, 256 MB		x10	
F	Tree with Small Distances	standard input/output 1 s, 256 MB		x30	
G	Compatible Numbers[1]	standard input/output 4 s, 256 MB		x30	
H	Vowels	standard input/output 4 s, 256 MB		x17	
I	Array Without Local Maximums	standard input/output 2 s, 512 MB		x27	
J	Karen and Supermarket	standard input/output 2 s, 512 MB		x10	
K	The Bakery	standard input/output 2.5 s, 256 MB		x15	

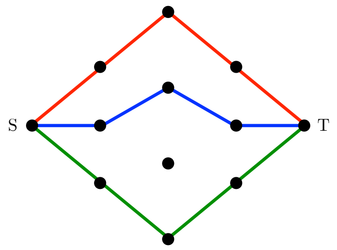
Topic: Network Flow Theory

- Network flow theory
- Matchings in Graphs
- Applications of Flows and Matchings
- Make Graphs, Not Algorithms

Network Flow

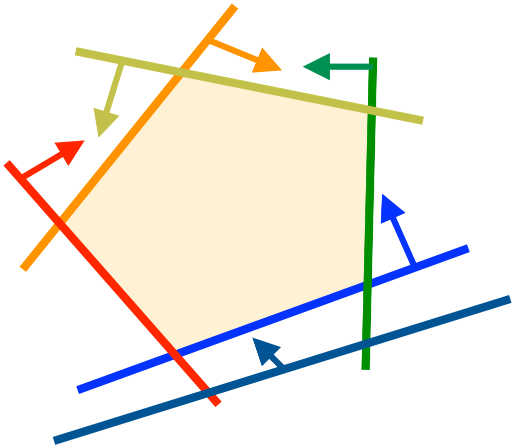
An edge-weighted graph can model a network of pipes, where the weight of edge (i, j) defines the *capacity* of the pipe.

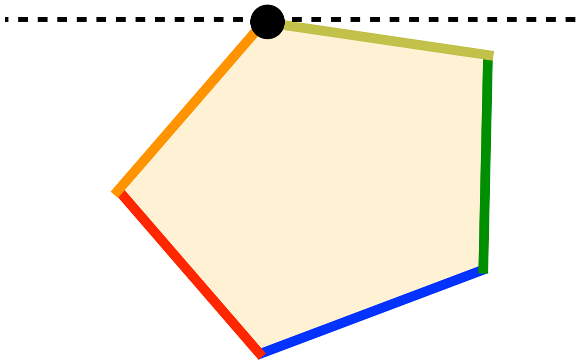




Given a weighted graph G and vertices s and t, the *network flow problem* seeks the maximum flow which can be sent from s to t respecting the maximum capacities of each pipe.

Linear Programming





Optimization people vs. algorithm people. . .

Flow Constraints as a Linear Program

Except for the source and the sink, each node must satisfy the constraint that flow in equals flow out:

$$\sum_{k\|(j,k)\in E} f_{jk} = \sum_{i\|(i,j)\in E} f_{ij}$$

The source and sink may have a capacity b, such that:

$$\sum_{k\|(s,k)\in E} f_{sk} = b = \sum_{i\|(i,t)\in E} f_{it}$$

The flow on each edge is bounded by its upper bound capacity u:

$$0 \leq f_{ij} \leq u_{ij}$$

Objective Functions

- The MaxFlow or maximum flow problem seeks to maximize the amount of flow from s to t. So b should be large (infinite) while we maximize:

$$\sum_{i \| (i,t) \in E} f_{it}$$

- If there is a cost c for each unit of flow, the minimum cost flow problem seeks to minimize:

$$\sum_{(i,j) \in E} c_{ij} \cdot f_{i,j}$$

Why is MaxFlow Important?

Although the network flow problem is of independent interest, its primary importance is that of being able to solve other important graph problems.

Classic examples are bipartite matching and edge/vertex connectivity testing.

But it is important as perhaps the most powerful linear programming problem which can be solved by combinatorial algorithms (not linear programming).

Linear Programming and Integrality

Linear programming is very powerful for optimization, but can produce fractional solutions: $\max y$ subject to the constraints $x = y$ and $x + y = 1$.

Forcing a linear program to only work with integers creates an NP-complete problem, integer programming.

But linear programmng on integer totally unimodular matrices as capacities *always* produce integer solutions.

The node-arc incidence matrix of a directed graph is totally unimodular. Hence network flow solutions with integer capacities always have integral solutions.

Thus flows on edges of capacity between 0 and 1 act as boolean variables.

Augmenting Paths

Traditional network flow algorithms are based on the idea of *augmenting paths*: finding a path of positive capacity from s to t and adding it to the flow.

It can be shown that the flow through a network is optimal iff it contains no augmenting path.

Since each augmentation increases the flow, by repeating the process until no such path remains we must eventually find the global maximum.

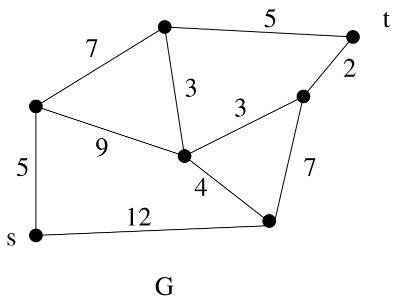
The Residual Flow Graph

The key structure is the *residual flow graph*, denoted as $R(G, f)$, where G is the input graph whose weights are the capacities, and f is array of flows through G.
For each edge (i, j) in G with capacity $c(i, j)$ and flow $f(i, j)$, $R(G, f)$ may contain two positive-weight edges:

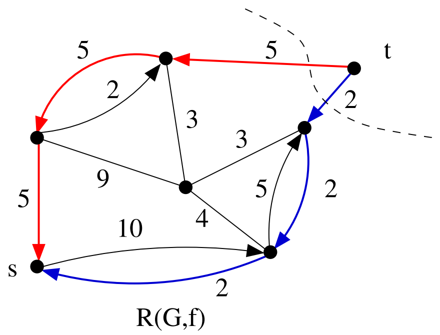
(i) an edge (i, j) with remaining capacity $c(i, j) - f(i, j)$, and

(ii) an edge (j, i) with weight $f(i, j)$, if $f(i, j) > 0$.

Example: Residual Flow Graph



G



R(G,f)

The maximum s–t flow in graph G is 7. Such a flow is revealed by the two directed t to s paths in the residual graph $R(G)$, of flows 2 and 5 respectively. These flows completely saturate capacity, so no augmenting path remains.

Edge-weights in the Residual Flow Graph

The weight of the edge (i, j) in the residual graph gives the exact amount of extra flow that can be pushed from i to j. A path in the residual flow graph from s to t implies that more flow can be pushed from s to t.
The smallest edge weight on this path defines the amount of extra flow that can be pushed along it.
Initial flows are all set to 0. The initial residual flow of (i, j) is set to the capacity of e, while the initial residual flow of (j, i) is set to 0.

MaxFlow Implmementatiopn

Add augmenting paths until none can be found.
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 accommodate the extra fields:

```
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;
```

Finding an Augmenting Path

We use a breadth-first search to look for any path from source to sink that increases the total flow, and use it to augment the total. We terminate with the optimal flow when no such *augmenting* path exists.

```
void 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);

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

The Volume of a Path

The flow through an augmenting a path is limited by the path edge with the smallest amount of residual capacity:

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

    if (parent[end] == -1) {
        return(0);
    }

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

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

Updating the Residual Graph

Sending an additional unit of flow along directed edge (i, j) reduces the residual capacity of edge (i, j) but *increases* the residual capacity of edge (j, i).

```
void augment_path(flow_graph *g, int start, int end, int volume) {
    edgenode *e;     /* edge in question */

    if (start == end) {
        return;
    }

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

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

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

The Edmonds-Karp Algorithm

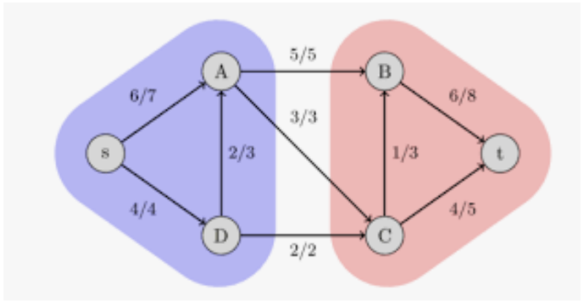
The augmenting path algorithm above eventually converges to the optimal solution. However, each augmenting path may add only a little to the total flow: perhaps as little as 1 flow unit per path if the weights are integral.
So‘in principle, the algorithm might take an arbitrarily long time to converge.
But Edmonds and Karp proved that using BFS to select the *shortest* augmenting path in the residual graph guarantees that $O(n^3)$ augmentations suffice for optimization.
Note this is independent of the capacities of the edges.

Algorithms for Minimum Cost Flow

Edmonds-Karp solved MaxFlow by repeatedly augmenting along the shortest unweighted path.
It similarly solves MinCost Flow by repeatedly augmenting along the shorted cost-weighted path.
Thus Dijkstra's algorithm should be used instead of BFS for MinCost flow.

Max Flow = Min Cut

A set of edges whose deletion separates 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 always possible.



Min cut is an important problem which can be solved by network flow, although there also more specialized randomized algorithms.

Questions?

Topic: Matchings in Graphs

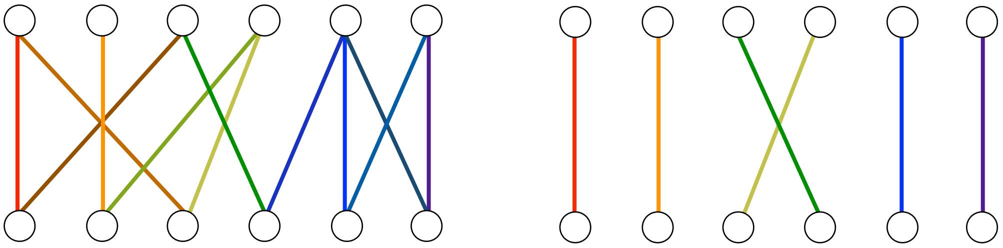
- Network flow theory
- Matchings in Graphs
- Applications of Flows and Matchings
- Make Graphs, Not Algorithms

Bipartite Graphs

Graph G is *bipartite* or *two-colorable* if the vertices can be divided into two sets, say, L and R, such that all edges in G have one vertex in L and one vertex in R.
Many naturally defined graphs are bipartite. For example, suppose certain vertices represent jobs to be done and the remaining vertices people who can potentially do them. The existence of edge (j, p) means that job j can potentially done by person p. Or let certain vertices represent boys and certain vertices girls, with edges representing compatible pairs.

Bipartite Matching

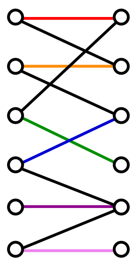
A *matching* in a graph $G = (V, E)$ is a subset of edges $E' \subset E$ such that no two edges in E' share a vertex. Thus every vertex is in at most one such matching edge.

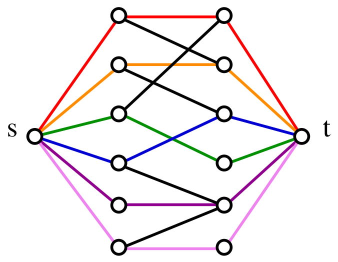


Matchings in bipartitie graphs have natural interpretations as job assignments or as marriages.

Bipartite Matching through Network Flow

The largest possible bipartite matching can be found using network flow, where each edge has capacity 1:

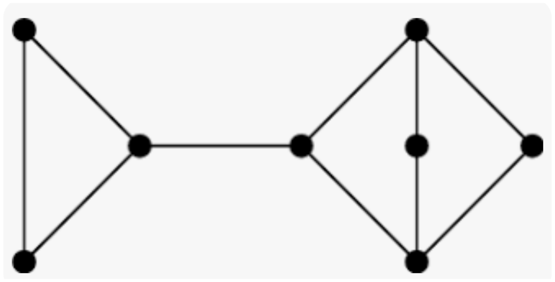




The maximum weighted bipartite matching can be found using the Hungarian algorithm. See textbook for details.

Matching in General Graphs

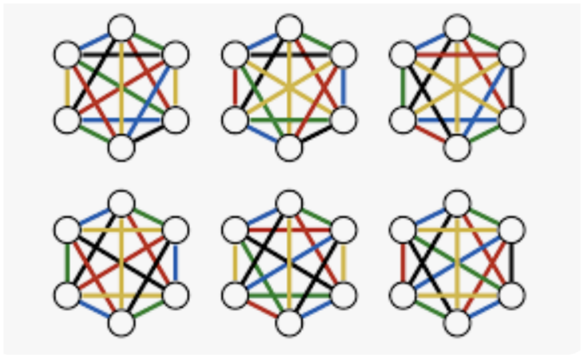
Maximum cardinality and minimum weight perfect matchings can also be found in general (non-bipartite graphs) using the blossum algorithm.



It uses augmenting paths, with special cases of odd-length cycles.

1-Factorizations

The edges of certain regular graphs can be partitioned into edge-disjoint matchings called *1-factorizations*:



Such 1-factorizations define *edge colorings*, where no two edges of the same color touch.

Vizing's theorem: every graph can be edge colored with Δ or $\Delta + 1$ colors, where Δ is the largest vertex degree.

Questions?

Topic: Applications of Flows and Matching

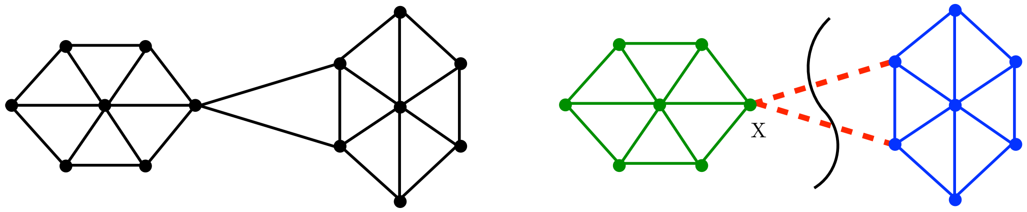
- Network flow theory
- Matchings in Graphs
- Applications of Flows and Matchings
- Make Graphs, Not Algorithms

Applications of Flows

I find that I think more naturally of bipartite matchings than network flows, and many flow applications are really just matching problems.
To learn to recognize flow problems, it is useful to look at as many examples as you can.

Edge Connectivity

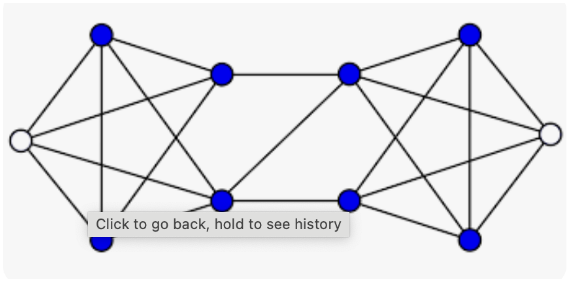
Finding the minimum cut between vertex v and all other vertices defines the edge-connectivity of G.



This means running $n-1$ min cut/max flows between v and each other vertex.

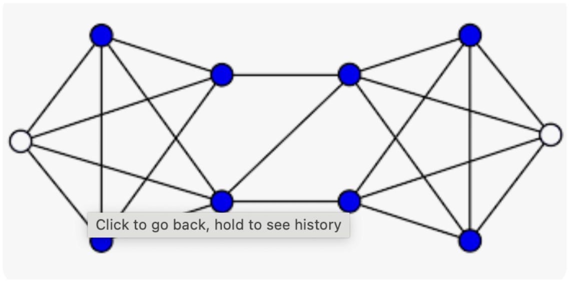
Menger’s Theorem and Edge-Disjoint Paths

The size of the minimum edge cut between s and t equals the maximum number of edge-disjoint paths from s to t.
Think flow on a graph with edge capacities 1.



Vertex Connectivity and Vertex-Disjoint Paths

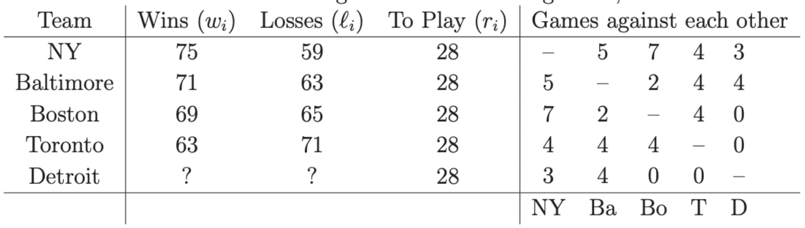
The size of the minimum vertex cut between s and t equals the maximum number of vertex-disjoint paths from s to t.



Think flow on a graph with edge capacities 1 and split verticies: $v_i(in)$ and $v_i(out)$

The Baseball Elimination Program

Table 1: Standings of AL East on August 30, 1996.



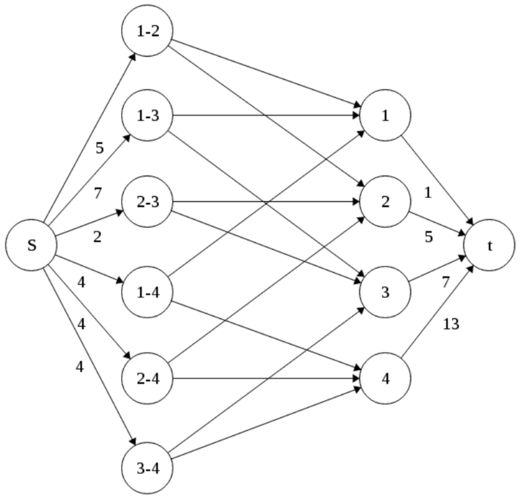
Team	Wins (w_i)	Losses (ℓ_i)	To Play (r_i)	Games against each other				
NY	75	59	28	–	5	7	4	3
Baltimore	71	63	28	5	–	2	4	4
Boston	69	65	28	7	2	–	4	0
Toronto	63	71	28	4	4	4	–	0
Detroit	?	?	28	3	4	0	0	–
				NY	Ba	Bo	T	D

Does Detroit have a chance if it has won 46 games to date? No, because $46 + 28 = 74 < 75$.
But what if it had won 47 games? 48? Is it officially eliminated, or might there be a pattern of outcomes where it beats all other teams?

Maxflow Solution

Build a bipartite network of:

- Team pairings (future games) weighted by how often team i plays j
- Teams, weighted by the maximum number of remaining games they can win while still below Detroit.



Is there a flow where all games can end with Detroit winning more total games?

Bipartite Matching Solution

The network is essentially bipartite.

- Have a vertex for each distinct game (not team pairing)
- Have w_i vertices for each team where w_i is the number of games team i is allowed to win

This formulation is less efficient if tean-pairs still must plan large numbers of games between each other. (one number w vs. w more vertices).

Realizing a Matrix from Row/Column Sums

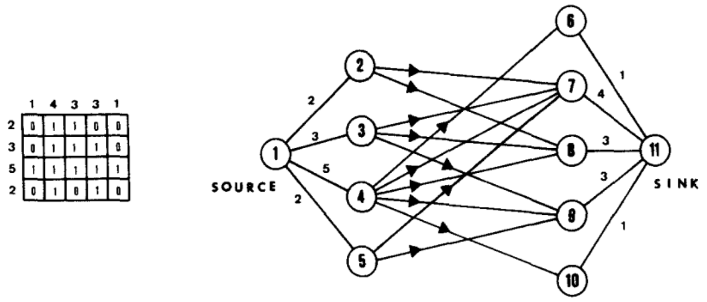
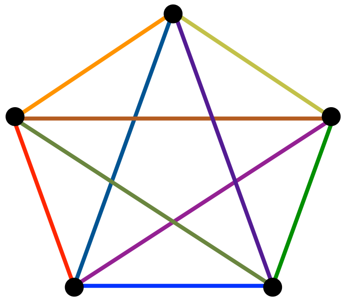


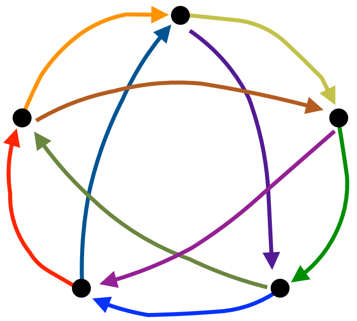
FIG. 1. The equivalence of reconstructing a binary matrix from row and column sums and finding a maximal flow in a capacitated directed network. The nodes are represented by circles and the arrows denote directed arcs. For simplicity, only those intermediate arcs are drawn that do transport a unit of flow.

There are generally many solutions: min cost flow can be used to add constraints on shape.
Again, a lot like matching with multiple copies of nodes.

Eulerian Cycles

An *Eulerian cycle* is a tour which visits every edge of the graph exactly once.





A mailman's route is ideally an Eulerian cycle, so he can visit every street (edge) in the neighborhood once before returning home.

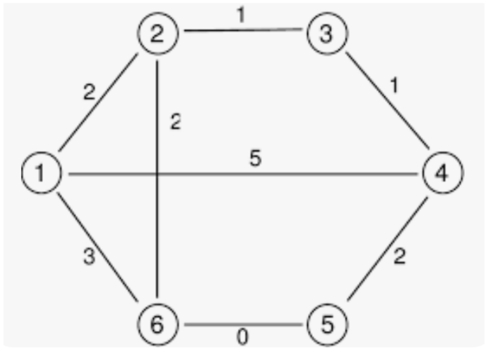
Degree Conditions and Eulerian Graphs

An undirected graph contains an Eulerian cycle if it is connected and every vertex is of even degree. Why? The circuit must enter and exit every vertex it encounters, implying that all degrees must be even.
A connected directed graph contains an Eulerian cycle if it is connected and every vertex has the equal in-degree and out-degree.

Chinese Postman

Given a weighted graph G, find the lowest-weight tour that visits every edge in G.

If it is Eulerian, the answer is an Eulerian cycle. If not, we must add edges between vertices odd or imbalanced degrees.



The optimal bipartite matching gives the least expensive way to make the graph Eulerian.

Finding an Eulerian Cycle

We can find an Eulerian cycle by building it one cycle at a time. We can find a simple cycle in the graph by finding a back edge using DFS. Deleting the edges on this cycle leaves each vertex with even degree. Once we have partitioned the edges into edge-disjoint cycles, we can merge these cycles arbitrarily at common vertices to build an Eulerian cycle.

A *Hamiltonian cycle* is a tour which visits every vertex of the graph exactly once. The traveling salesman problem asks for the shortest such tour on a weighted graph.

Unfortunately, no efficient algorithm exists for solving Hamiltonian cycle problems. If the graph is sufficiently small, it can be solved via backtracking.

Questions?

Topic: Make Graphs, Not Algorithms

- Network flow theory
- Matchings in Graphs
- Applications of Flows and Matchings
- Make Graphs, Not Algorithms

Make Graphs, Not Algorithms

Designing novel graph algorithms is very hard, so don’t do it. Instead, try to design graphs that enable you to use classical algorithms to model your problem.

This approach is consistent with the idea of a reduction between two problems, which is important in the theory of NP-completeness.

Shortest k-Link Path

Given a weighted graph G, find the shortest (lowest weight) path using exactly k links from s to t.

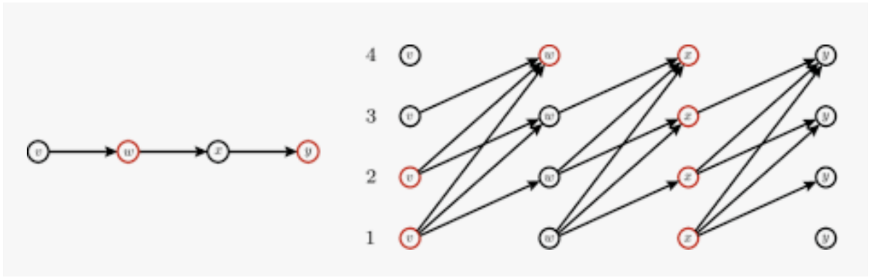
Yes, there is a dynamic programming solution:

$$Cost[s,t,k] = \min_{j} w(s,j) + Cost[j,t,k-1]$$

But is there another way?

Reduction to Shortest Path

Build a DAG with k copies of G, such that all edges go from the ith to $(i+1)$ copy.



Dijkstra's algorithm (or something simpler) can now find the shortest weighted path from s_1 to t_k.

Names in Collision

In porting code from Unix to DOS, I have to shorten several hundred file names down to at most eight characters each. I can’t just use the first eight characters from each name, because “filename1” and “filename2” would be assigned the exact same name. How can I meaningfully shorten the names while ensuring that they do not collide?

Disambiguating Names

Construct a bipartite graph with vertices corresponding to each original file name f_i for $1 \leq i \leq n$, as well as a collection of acceptable shortenings for each name $f_{i1}, \ldots, f_{ik}$. Add an edge between each original and shortened name. We now seek a set of n edges that have no vertices in common, so each file name is mapped to a distinct acceptable substitute. *Bipartite matching* is exactly this problem of finding an independent set of edges in a graph.

Questions?