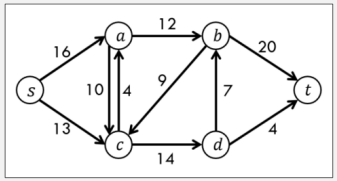
Network Flow

- a *flow network* consists of –
  - a directed graph
  - a source *s*
    - *s* has only outgoing edges
  - a sink *t*
    - *t* has only incoming edges
  - capacities on each edge



- assign *flow* to each edge

Feasible Flows

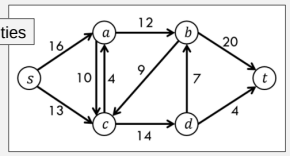
- a *feasible* flow must satisfy two conditions
  - *capacity constraint* – the flow on an edge cannot exceed its capacity

$$0 \le f(u,v) \le c(u,v)$$

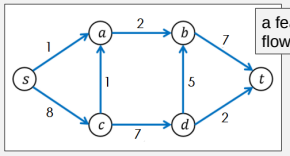
  - *flow conservation* – for every vertex except *s* and *t*, the incoming flow must equal the outgoing flow

$$\sum f(u,v) = \sum f(v,w)$$

capacities



a feasible flow

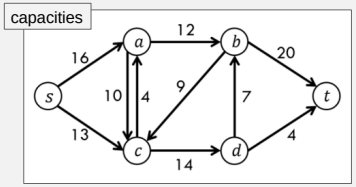


Maximum Flow

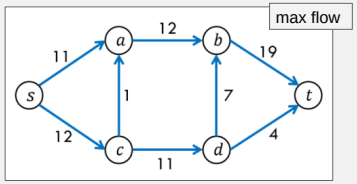
- assign *flow* so as to maximize the flow from *s* to *t*
  - flow from *s* to *t* = total flow = sum of the flow on the outgoing edges of *s*

$$|f| = \sum_v f(s,v)$$

capacities



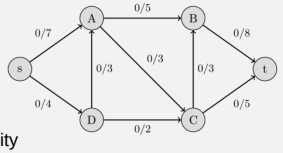
max flow

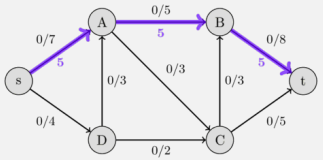


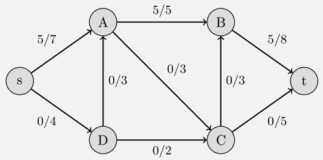
Ford-Fulkerson Method

- to find max flow –
  - start with zero flow – $f(u,v) = 0$ for all edges $(u,v) \in E$
  - repeat
    - find a path from *s* to *t* with unused capacity
    - send as much flow as possible along that path
    - update the remaining capacities

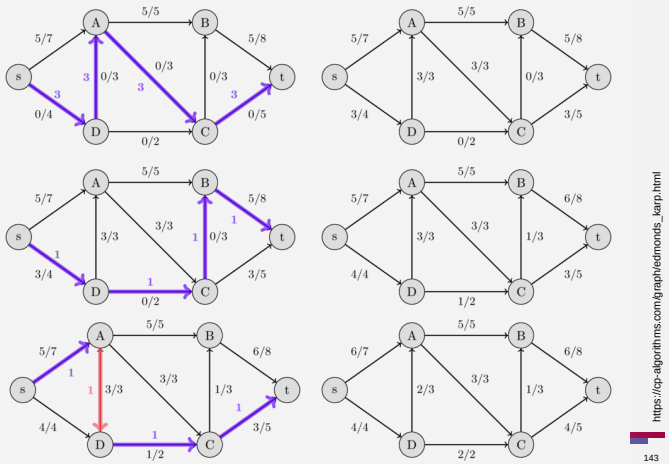
  until there aren't any more augmenting paths







Ford-Fulkerson Method

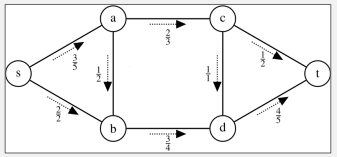


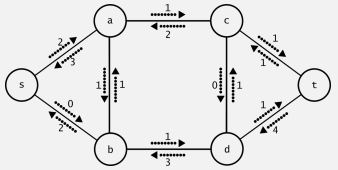
https://cp-algorithms.com/graph/edmonds_karp.html

Edmonds-Karp Algorithm

- Ford-Fulkerson is a *method* – it doesn't specify how the augmenting paths are found
- Edmonds-Karp uses BFS to find augmenting paths
  - $O(VE^2)$
  - makes use of a *residual graph* to keep track of the remaining capacities
    - contains remaining forward capacity + reverse edges to allow cancellation of flow

$$c_f(u,v) = c(u,v) - f(u,v)$$

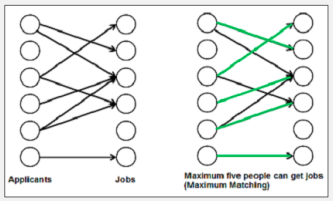




https://en.wikipedia.org/wiki/Flow_network

Bipartite Matching

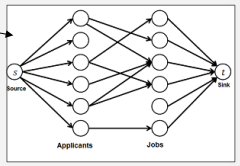
- a *matching* is a set of edges $E' \subset E$ such that no two edges of E' share a vertex
- a *bipartite graph* divides V into two sets L, R such that for all edges $(u,v) \in E$, u in L and v in R
- the *maximum cardinality bipartite matching* is the largest set of edges that form a matching in a bipartite graph

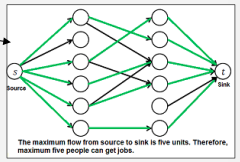


https://www.geeksforgeeks.org/dsa/maximum-bipartite-matching/

Bipartite Matching

- maximum cardinality bipartite matching can be solved by a reduction to max flow
  - build a flow network
    - create a source *s* and a sink *t*
    - connect *s* to all vertices in *L*
    - connect all vertices in R to *t*
    - for each edge $(u,v) \in E$, direct the edge from the vertex in *L* to the vertex in *R*
    - all edges have capacity 1
  - compute the max flow
- observations
  - the value of the max flow is the cardinality of the matching
    - capacity 1 means there can be at most one edge in the matching for each vertex
  - the edges that end up with flow 1 constitute the matching





https://www.geeksforgeeks.org/dsa/maximum-bipartite-matching/

Minimum Cost Flow

- edges have both capacity and cost d_{ij}
- find the cheapest way to send a certain amount of flow f from s to t
  - minimize $\sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij} \cdot x_{ij}$ subject to $\sum_{i=1}^{n} x_{it} = f$
    - x_{ij} = flow on edge ij
- methods and algorithms – find a library implementation
  - cycle-canceling, similar to Ford-Fulkerson
    - start with an arbitrary feasible flow, then iteratively improve the cost
  - linear programming
  - others
- variation: *min cost max flow* – find the cheapest max flow

Summary

- bipartite matching
  - goal: match elements of two sets so that no vertex is used more than once
  - can be solved by converting graph into a flow network and using max flow
- max flow
  - each edge has a capacity
  - goal: send as much flow as possible from a source s to a sink t respecting capacities and conservation of flow
  - generalizes matching to arbitrary networks
- min cost flow
  - each edge has a capacity and a cost
  - goal: send flow as cheaply as possible
  - generalizes max flow, solves assignment and transportation problems

Applications – Bipartite Matching

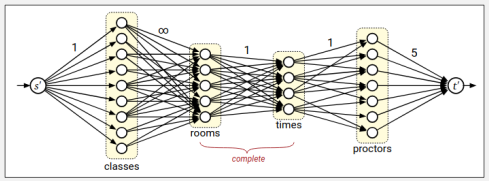
- models problems where elements from two sets are paired
  - each element can participate in at most one pair
  - edges represent compatible matches between the two sets
- one-to-one assignment
  - resource allocation – assigning workers to tasks, jobs to machines, applicants to jobs, …

Applications – Max Flow

- models problems involving capacity constraints
- transportation networks
  - cities are connected by roads with capacity limits
  - compute the maximum number of vehicles that can travel between two cities
- network routing
  - routers are connected by links with bandwidth limits
  - determine the maximum data throughput between two points

Applications – Max Flow

- tuple selection (name due to Jeff Erickson)
  - e.g. schedule a room, time slot, and proctor for the final exam for each class
    - one class per room per time slot
    - each proctor can oversee only one exam at a time and five exams total
    - each proctor is only available for certain time slots



Applications – Max Flow

- sports team elimination – when is a team out of contention for the overall title

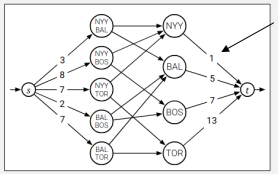
Team	Won–Lost	Left	NYY	BAL	BOS	TOR	DET
New York Yankees	75–59	28		3	8	7	3
Baltimore Orioles	71–63	28	3		2	7	4
Boston Red Sox	69–66	27	8	2		0	0
Toronto Blue Jays	63–72	27	7	7	0		0
Detroit Tigers	49–86	27	3	4	0	0	

Detroit is doing badly...but can they still win?

is it possible to select a winner for each of the remaining games so that team *n* wins the most games?

Detroit (49-86) has won 49 and has 27 games left, for a max of 76 wins possible – if NY (75-59) wins more than 76-75 = 1 more game, they will have 77+ wins and Detroit's hopes are dashed

number of games remaining between pairs of teams



flow network construction –
- a *game* vertex for each possible game remaining (pairs of teams) not involving team *n*
- a *team* vertex for the other *n*-1 teams (not team *n*)
- a source *s* and a sink *t*
- edges between each game vertex and the two teams involved, with capacity ∞
- an edge from the source to each game vertex, with capacity equal to the number of games remaining between those two teams
- an edge from each team vertex to the sink, with capacity equal to the max number of additional wins the team can have without beating team *n*

Theorem. *Team n can end the season in first place if and only if there is a feasible flow in this graph that saturates every edge leaving s.*

the flow corresponds to "winner" tokens – the capacity of the edges leaving *s* allow up to one winner token per game, the edges between game vertices and team vertices distribute the winner tokens amongst the teams playing in those games, and the capacity of the edges entering *t* limit each team's wins to a number that won't exceed what team *n* can achieve

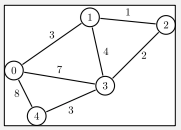
the total capacity of edges leaving *s* corresponds to the number of games remaining – if those edges are all saturated, winner tokens for all of the remaining games are accounted for, and since all flow starting at *s* must reach *t*, the only way for all edges leaving *s* to be saturated is if there is a scenario where no other team beats team *n* overall

Applications – Min Cost Flow

- transportation problem
  - cheapest way to transport items from point A to point B given some routes with limited capacity
- shipping and logistics
  - warehouses supply goods to stores
  - edges represent shipping routes with costs
  - find the cheapest way to satisfy demand
- airline scheduling
  - flights must be assigned aircraft
  - costs represent delays, repositioning, or operational costs
- assignment problem – minimum weight bipartite matching
  - matching problems where assignments have different costs

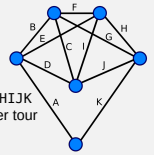
More Graph Algorithms

min cut – cheapest set of edges whose removal partitions the graph into two disjoint subsets



postman tour – min-length cycle traversing every edge at least once

euler cycle (tour) – cycle traversing every edge exactly once



ABCDEFGHIJK is an euler tour

maximum spanning tree – spanning tree that maximizes the sum of the edge weights

minimum bottleneck spanning tree – spanning tree with minimum max edge weight (of all the spanning trees, this one has the smallest max edge weight)

minimum product spanning tree – spanning tree that minimizes the product of the edge weights

https://hyperskill.org/learn/step/5645
https://en.wikipedia.org/wiki/Eulerian_path