CSE 417
Network Flows (pt 4)
Min Cost Flows

UNIVERSITY *of* WASHINGTON

Reminders

> HW6 is due Monday

Review of last three lectures

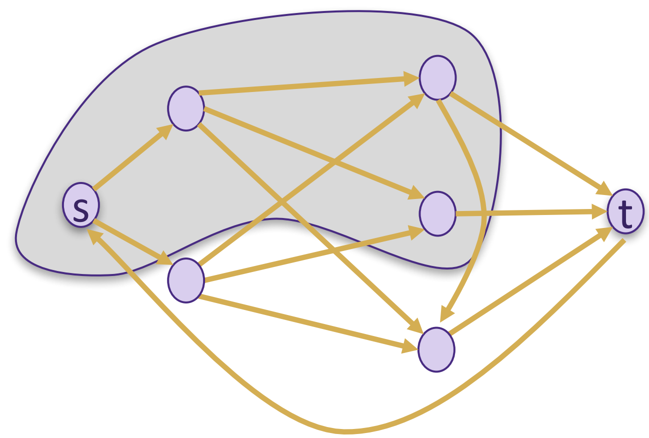
- Defined the maximum flow problem
  - find the feasible flow of maximum value
  - flow is feasible if it satisfies edge capacity and node balance constraints
- Described the Ford-Fulkerson algorithm
  - starts with a feasible flow (all zeros) and improves it (by augmentations)
  - essentially optimal if max capacity into t is O(1)
- Many, many other algorithms...

Review of last three lectures

- Modeling with matching, flows, & cuts
  - matching: allow multiple matches, restrict to groups
  - flows: node capacities, lower bounds, etc.
  - cuts: translate min to max, restrict allowed subsets using infinite capacities
    - one cut per subset of V \ {s, t}
- Many of those graphs have O(1) capacities, so F-F is fast

Review of last three lectures

- Theorem: value of max-flow = capacity of min-cut
  - any flow value ≤ any cut capacity
    - flow has to leave via those edges
  - F-F gives us a flow that matches cut value
    - flow value = flow leaving cut – flow entering cut
    - cut edges are saturated
    - backward edges have 0 flow



Review of last three lectures

> Techniques for *efficiently* solving problems defined over subsets:
  1. dynamic programming
  2. minimum cuts

> Cuts: define a graph where cut capacity = value
  – restrict allowed cuts using infinite capacities on edges
    > no min cut will ever include an infinite capacity edge
  – examples last time were maximization, so we had cut capacity = C – value
    > minimizing capacity is maximizing value when C is constant

Outline for Today

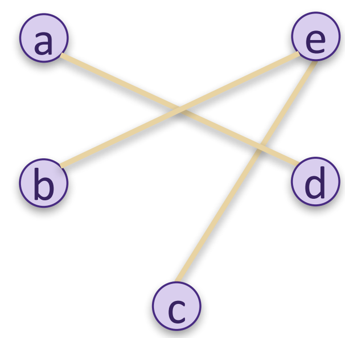
> Dynamic Programming over Subsets
> Minimum Cost Flows
> Cycle Canceling Algorithm
> Augmenting Path Algorithm
> Other Algorithms

Dynamic Programming Over Subsets

- Dynamic programming can be applied to *any* problem on subsets
  - opt solution on $\{a_1, \ldots, a_n\}$ = better of
    opt solution on $\{a_1, \ldots, a_{n-1}\}$ and
    (opt solution on $\{a_1, \ldots, a_{n-1}\}$ to which a_n can be legally added) + a_n
- BUT if problem is hard (e.g., NP-complete), it will be slow
  - in particular, there will be *too many* sub-problems
- <u>Key point</u>: don't have to guess if DP will work
  - just count the number of sub-problems you get
  - if it's small, the technique works

DP Over Subsets: Non-Example

> **Problem** (Independent Set): Given a graph, find the largest subset of nodes such that no two are connected by an edge.
  – (sort of opposite of a matching problem)



> Apply dynamic programming...
  – opt solution on $\{a_1, \ldots, a_n\}$ = better of
    opt solution on $\{a_1, \ldots, a_{n-1}\}$ and
    (opt solution on $\{a_1, \ldots, a_{n-1}\}$ to which a_n can be legally added) + a_n
  – latter is subsets of $\{a_1, \ldots, a_{n-1}\}$ with no neighbors a_n

DP Over Subsets: Non-Example

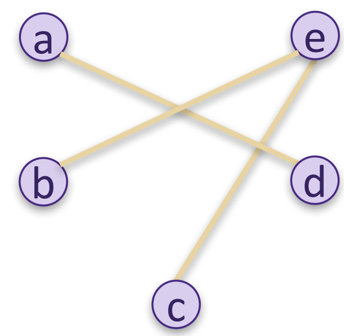
find opt solution on a, b, c, d, e

    find opt solution on a, b, c, d

        ...

    find opt solution on a, b, c, d not adjacent to e

        ...



DP Over Subsets: Non-Example

find opt solution on a, b, c, d, e

    find opt solution on a, b, c, d

        find opt solution on a, b, c

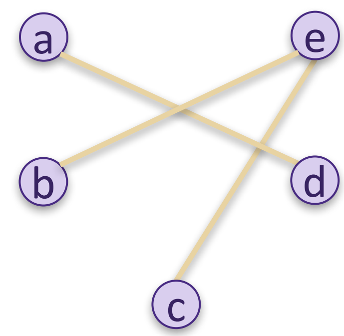
            ...

        find opt solution on a, b, c not adjacent to d

            ...

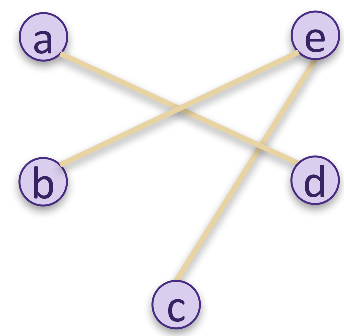
    find opt solution on a, b, c, d not adjacent to e

        ...

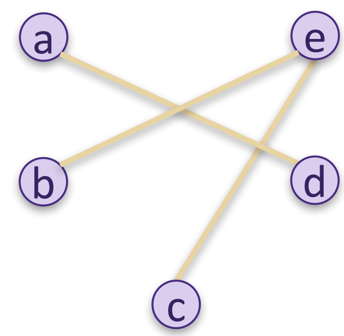


DP Over Subsets: Non-Example

find opt solution on a, b, c, d, e
    find opt solution on a, b, c, d
        …
    find opt solution on a, b, c, d not adjacent to e
        find opt solution on a, b, c not adjacent to e
            …
        find opt solution on a, b, c not adjacent to {d, e}
            …



DP Over Subsets: Non-Example



find opt solution on a, b, c, d, e

    find opt solution on a, b, c, d

        …

    find opt solution on a, b, c, d not adjacent to e

        find opt solution on a, b, c not adjacent to e

            find opt solution on a, b not adjacent to e

                …

            find opt solution on a, b not adjacent to {c, e}

                …

        find opt solution on a, b, c not adjacent to {d, e}

            …

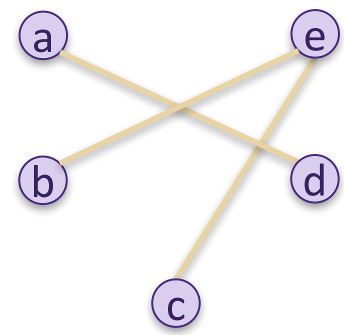
DP Over Subsets: Non-Example

- In general, sub-problems are:

  find opt solution on $a_1, \ldots, a_j$ not adjacent to S,
  where S is some subset of $\{a_{j+1}, \ldots, a_n\}$

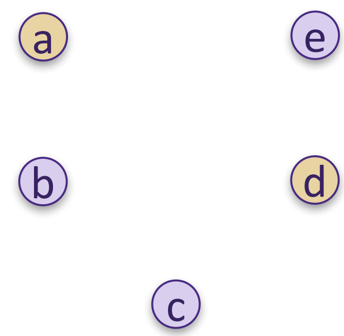
  – there are exponentially many such sub-problems
    - (none of them repeat)

- So dynamic programming is not useful here…
  – (we don't have enough memory to memoize / build table)



DP Over Subsets: Example

- **Problem**: Given a list of items of two colors, purple and gold, find the subset of maximum value that does **not** have 2+ purples or 2+ golds
  - (previous problem with golds connected to purples)
- Easy to solve this directly
  - max(0, max value of a purple) + max(0, max value of a gold)
- Dynamic programming will do the same thing….



DP Over Subsets: Example

> Apply dynamic programming....
  - opt solution on $\{a_1, \ldots, a_n\}$ = better of
    opt solution on $\{a_1, \ldots, a_{n-1}\}$ and
    (opt solution on $\{a_1, \ldots, a_{n-1}\}$ to which a_n can be legally added) + a_n

find opt solution on a, b, c, d, e
  find opt solution on a, b, c, d
    ...
  find opt solution on a, b, c, d using no purples
    ...

DP Over Subsets: Example

find opt solution on a, b, c, d, e

    find opt solution on a, b, c, d

        ...

    find opt solution on a, b, c, d using no purples

        ...

DP Over Subsets: Example

find opt solution on a, b, c, d, e

find opt solution on a, b, c, d

find opt solution on a, b, c

...

find opt solution on a, b, c using no golds

...

find opt solution on a, b, c, d using no purples

find opt solution on a, b, c using no purples

...

find opt solution on a, b, c using no purples & no golds

...

DP Over Subsets: Example

find opt solution on a, b, c, d, e
find opt solution on a, b, c, d
find opt solution on a, b, c
...
find opt solution on a, b, c using no golds
find opt solution on a, b using no golds
...
find opt solution on a, b using no purples and no golds
...
find opt solution on a, b, c, d using no purples
find opt solution on a, b, c using no purples
...
find opt solution on a, b, c using no purples & no golds
...

DP Over Subsets: Example

- In general, sub-problems are:

  opt solution on $a_1, \ldots, a_i$
  with (no purple / no gold / no purple or gold / any)

- Dynamic programming works well here
  - only 4n sub-problems
    - (means we are getting many, many repeats in the recursion)
  - O(n) just like the direct solution

Dynamic Programming over Subsets

> Dynamic programming can be applied to *any* problem on subsets
  – opt solution on $\{a_1, \ldots, a_n\}$ = better of
    opt solution on $\{a_1, \ldots, a_{n-1}\}$ and
    (opt solution on $\{a_1, \ldots, a_{n-1}\}$ to which a_n can be legally added) + a_n

> BUT if problem is hard (e.g., NP-complete), it will be slow
  – in particular, there will be *too many* sub-problems

> Key point: don't have to guess if DP will work
  – just count the number of sub-problems you get
  – if it's small, the technique works

Dynamic Programming over Subsets

- Dynamic programming can be applied to *any* problem on subsets
  - opt solution on $\{a_1, \ldots, a_n\}$ = better of
    opt solution on $\{a_1, \ldots, a_{n-1}\}$ and
    (opt solution on $\{a_1, \ldots, a_{n-1}\}$ to which a_n can be legally added) + a_n
- BUT if problem is hard (e.g., NP-complete), it will be slow
  - in particular, there will be *too many* sub-problems
- HW6 problem 1 shows the good case (stays small)
- HW6 problem 3 shows the bad / normal case

Outline for Today

> Dynamic Programming over Subsets
> Minimum Cost Flows
> Cycle Canceling Algorithm
> Augmenting Path Algorithm
> Other Algorithms

Minimum Cost Flow Problem

> **Problem**: Given a graph G, two nodes s and t, a *flow value* v, and both a capacity, u_e, and a cost, c_e, for each edge e, find the feasible flow of value v with least cost.
  - (note: changing capacity from c_e to u_e… c_e is now cost of edge e)
  - <u>flow cost</u> is defined as sum of f_e c_e over all edges

> (As before, a feasible flow is one that satisfies both
  - flow balance constraint: excess(n) = 0 for each n ≠ s, t
  - capacity constraint: $f_e \leq c_e$ for each edge e)

Minimum Cost Flow Problem

- Can generalize to include lower bounds and demands
  - same constructions given for feasible flow apply to min cost flow
    - remove lower bounds by subtracting them out
    - remove demands by adding a new source and sink
  - removing lower bounds changes value of the flow but not which is minimum
- Min cost flow many useful applications...
  - more examples next lecture
  - start with the two premier examples

Assignment Problem

- **Problem**: Given two equal-length lists of objects, A and B, and a cost, $c_{a,b}$, for each pair (a, b), find the perfect matching of minimum total cost.
  - a perfect matching is one that matches every a in A and every b in B
  - cost of the matching is the sum of the costs of each match

- Saw (maximum) bipartite matching previously...
  this is minimum cost bipartite matching

Assignment Problem Example

B

A		1'	2'	3'	4'	5'
	1	3	8	9	15	10
	2	4	10	7	16	14
A	3	9	13	11	19	10
	4	8	13	12	20	13
	5	1	7	5	11	9

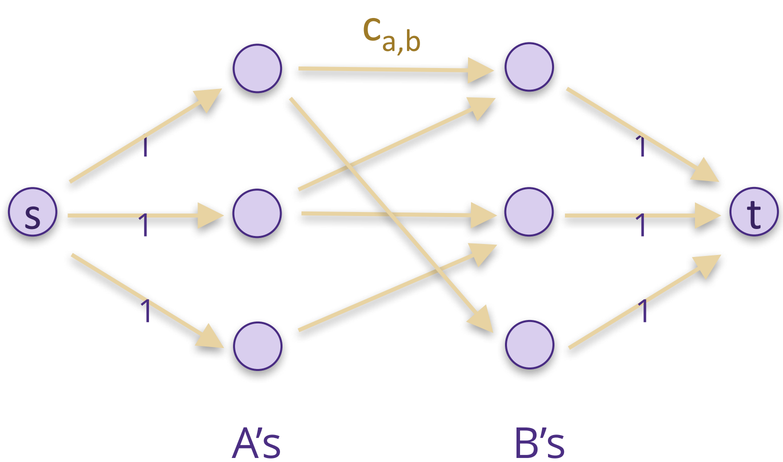
Min cost perfect matching

M = { 1-2', 2-3', 3-5', 4-1', 5-4' }

cost(M) = 8 + 7 + 10 + 8 + 11 = 44

Assignment Problem

> **Solution**: model as a min cost flow problem
  – start with the same modeling as for (maximal) bipartite matching
  – set edge (a, b) to have cost $c_{a,b}$



gold = cost
purple = capacity

Assignment Problem

- **Solution**: model as a min cost flow problem
  - start with the same modeling as for (maximal) bipartite matching
    - create a graph whose nodes are the As and Bs
    - source s has edge to each a in A with capacity 1
    - target t has edge from each b in B with capacity 1
  - set edge (a,b) to have cost $c_{a,b}$
  - find min cost flow of value $|A|$

- (As mentioned before, bipartite cases are not really special cases… any flow graph can be made bipartite through a transformation)
  - textbook only talks about this problem

Transportation Problem

> **Problem**: Given two equal-length lists of objects, A and B, amounts to be supplied by each a in A, amounts required by each b in B, and a cost, $c_{a,b}$, for sending units from a to b, find the least cost way to meet the required demands.
>
> – sum of the demands should equal the sum of the supplies

> **Application**: Find the cheapest way to ship products from warehouses (sources) to stores (sinks).

Transportation Problem

> Just a special case of min cost flow with demands where the graph happens to be bipartite
  - left side = supply nodes
  - right side = sink nodes

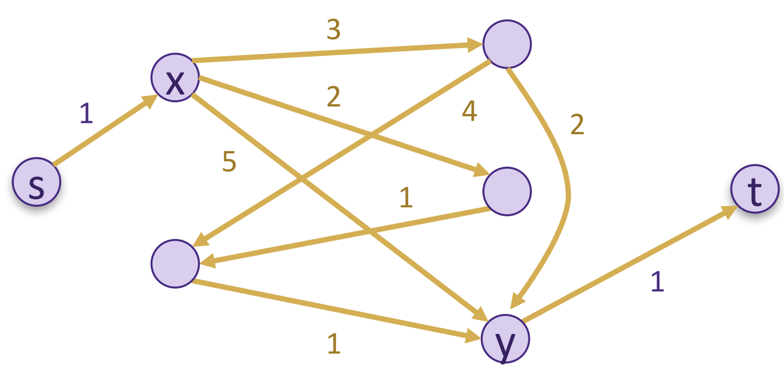
> **Solution**: model as a min cost flow problem with demands
  - apply the transformation to remove demands

Relation to Other Flow Problems

- Max flow / feasible flow is a special case:
  - setting all the costs to zero makes any flow of that value a solution
  - (can use binary search to find the maximum flow value)
- Shortest path is a special case...
  - (this includes negative cost edges, so no Dijkstra's algorithm)

Relation to Other Flow Problems

- Shortest path from x to y is a special case:
  - given a graph with costs (gold below) but no capacities (purple)
  - add a source with a capacity-1 edge to x
  - add a sink with a capacity-1 edge from y
  - any 0/1 flow is a path — cost of the flow is the cost of that path



Relation to Other Flow Problems

> Max flow / Feasible flow is essentially a special case ← no costs

> Shortest path is a special case ← no capacities

> Most general flow problem we will see

> Most useful flow problem for modeling

> Best solutions to both problems take $\Omega(nm)$ time.
  – we should expect algorithms slower than $O(nm)$

Why is this harder? (out of scope)

> Not harder because minimizing rather than maximizing
  - as we saw with cuts, we can often turn minimization into maximization
  - we could equivalently talk about max-cost flow: just negate all costs

> Key issue is the introduction of a new measure: costs
  - max flow directly maximizes what is being constrained (flow values)
  - min cost flow introduces a separate metric (costs) that needs to be minimized and do not appear in the normal flow constraints
  - look for this to see if you want to model with min cost flow vs max flow

Outline for Today

> Dynamic Programming over Subsets
> Minimum Cost Flows
> Cycle Canceling Algorithm
> Augmenting Path Algorithm
> Other Algorithms

Cycle Canceling Algorithm

- One simple algorithm:
  - start with any feasible flow
  - repeat as long as possible:
    - find a negative cost cycle in G(f)
    - let δ be minimum capacity along the cycle
    - push δ more flow along cycle

e.g., use max flow algorithm

e.g., use shortest path algorithm
(Bellman-Ford can detect negative cycles)

- Correctness: pushing flow along a cycle preserves balance
  - every node gets δ more incoming and δ more outgoing
  - hence, the flow remains feasible until termination

Cycle Canceling Algorithm

- One simple algorithm:
  - start with any feasible flow
  - repeatedly push flow along a negative cost cycle in G(f) until none exists
- Correctness: algorithm exits when flow is optimal
  - i.e., feasible flow is optimal iff there are no negative cost cycles in G(f)
  - if f' were optimal, then f – f' would be a *circulation* of positive cost
    - i.e., if f and f' both have excess d_u at u, then f – f' has excess 0 at u
  - circulation decomposes into a collection of cycles
  - each cycle has non-negative cost
    - if any had negative cost, f' could be improved further

Cycle Canceling Algorithm

> One simple algorithm…
  – start with any feasible flow
  – repeatedly push flow along a negative cost cycle in G(f) until none exists

> Can use Bellman-Ford to find a negative cycle in O(nm) time
  – (can actually use Dijkstra instead for this… see textbook)

> Total running time is $O(nm^2CU)$
  – where C is maximum cost and U is maximum capacity
  – can prove this is $O(n^2 m^3 \log n)$ by choosing appropriate cycles
    > use cycle with minimum *average* cost (see earlier lecture)

Outline for Today

> Dynamic Programming over Subsets
> Minimum Cost Flows
> Cycle Canceling Algorithm
> Augmenting Path Algorithm
> Other Algorithms

Augmenting Path Algorithm

- Second simple algorithm:
  - start with zero flow
  - repeat until flow value is v:
    - find min cost s ~> t path in G(f)
    - push 1 more unit of flow along cycle
- Idea: preserves optimality rather than feasibility
  - only get feasibility upon termination
  - (note that this assumes no negative cost cycles in the graph so that zero flow is optimal)

Augmenting Path Algorithm

> Second simple algorithm:
  - start with zero flow
  - repeatedly push 1 unit of flow along min cost s ~> t path in G(f) until value is v

> Correctness: pushing flow along a path preserves balance
  - uses same augmentation process as used in max flow algorithm
  - all constraints are satisfied except the value of the flow equaling v

Augmenting Path Algorithm

- Second simple algorithm:
  - start with zero flow
  - repeatedly push 1 unit of flow along min cost s ~> t path in G(f) until value is v

- Correctness: augmentation preserves optimality
  - if pushing 1 flow produces non-optimal flow, G(f) has a negative cost cycle
    - (see discussion of earlier algorithm)
  - this can only happen because new edge (v, u) appears in G(f)
    - augmenting path is s ~> u → v ~> t
    - cycle includes edge (v, u): v → u ~> v
    - combination is shorter s ~> t path: s ~> u ~> v ~> t — contradiction

Augmenting Path Algorithm

- Second simple algorithm:
  - start with zero flow
  - repeatedly push 1 unit of flow along min cost s ~> t path in G(f) until value is v
- Number of augmentations is value of flow
  - as discussed with Ford-Fulkerson, value of flow ≤ nU
- Total running time is $O(n^2 m U)$

Consequences

> **Theorem**: If all the capacities are integers, then there is a min cost flow where each edge flow is *integral*.
  – note: no restriction on costs

> **Proof**:
  – our algorithms work via augmentations
  – as before, if all capacities are integers, we will increase flows by integer amounts on each iteration
  – hence, the flow upon termination will be integral

Primal-Dual Algorithm (out of scope)

- Max flow algorithm repeatedly solves reachability
- Min cost flow algorithm repeatedly to shortest path

- Common: solve problem by repeatedly solving easier problem

- This is not an accident…
  - both algorithms are special cases of the “primal dual algorithm” for LPs
  - very useful technique for algorithms problems
    - doesn’t always give optimal algorithms (as these examples show)
    - but usually gives an algorithm and very useful insights

Duality (out of scope)

- As with max flow, there are dual objects that give upper bounds
  - in max flow those were cuts, which bound the value of any flow
- For min cost flow, the dual objects are actually *distances*
  - can think of a cut as a special case: those in the cut are distance 0 from s and those outside the cut are distance infinity from s
  - min cost flow matches the upper bound given by shortest paths in G(f)
- This is another reason why both max / feasible flow & shortest path are required to understand min cost flow

Outline for Today

> Dynamic Programming over Subsets
> Minimum Cost Flows
> Cycle Canceling Algorithm
> Augmenting Path Algorithm
> Other Algorithms

Other Algorithms (out of scope)

> Not as many algorithms for min cost flow as max flow

> Most prominent (& useful) is the network simplex method
  - specialization of the simplex method for linear programming to problems defined on graphs
  - accommodates additional (linear) constraints very easily
  - seems to be very fast in practice
  - Tarjan proved $O^*(nm)$ bound in theory also

Other Algorithms (way out of scope)

- In theory, this problem should not be any harder than max flow
  - the space of feasible flows forms a convex set
    - same for flows of a particular value
  - min cost flow asks us to minimize a linear function over that set
  - under mild assumptions, if you can solve the feasibility problem on a convex set, then you can
    - (need a “separation oracle” for the set)
    - proof is to apply the Ellipsoid algorithm…