Advanced Algorithm Design and Analysis
CSc 140

Final Exam Review

Instructor: Hady Ahmady Phoulady

General Advice for Study

- **Understand** how the algorithms are working
  - Work through the examples we did in class
  - "Narrate" for yourselves the main steps of the algorithms in a few sentences
- Know **when** or **for what problems** the algorithms are applicable
- **Do not memorize** algorithms

Dynamic Programing

Dynamic Programming

- Used for **optimization problems**
  - A set of choices must be made to get an optimal solution
  - Find a solution with the optimal value (minimum or maximum)
  - There may be many solutions that lead to an optimal value
  - Our goal: **find an optimal solution**

Dynamic Programming Algorithm

1. **Characterize** the structure of an optimal solution
2. **Recursively** define the value of an optimal solution
3. **Compute** the value of an optimal solution in a bottom-up fashion
4. **Construct** an optimal solution from computed information (not always necessary)

Elements of Dynamic Programming

- Optimal Substructure
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is build in a bottom-up manner from optimal solutions to subproblems
- Overlapping Subproblems
  - If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems

Optimal Substructure

- Optimal substructure varies across problem domains:
  - *How many subproblems* are used in an optimal solution.
  - *How many choices* in determining which subproblem(s) to use.
- Informally, running time depends on (# of subproblems overall) $\times$ (# of choices).
- Dynamic programming uses optimal substructure **bottom up**.
  - *First* find optimal solutions to subproblems.
  - *Then* choose which to use in optimal solution to the problem.

Optimal Substucture

- Does optimal substructure apply to all optimization problems? No.
  - Applies to determining the **shortest path** but NOT the **longest simple path** of an unweighted directed graph.
- Why?
  - Shortest path has independent subproblems.
  - Solution to one subproblem does not affect solution to another subproblem of the same problem.
  - Subproblems are not independent in longest simple path.
    - Solution to one subproblem affects the solutions to other subproblems.

Overlapping Subproblems

- The space of subproblems must be “small”.
- The total number of distinct subproblems should be polynomial in the input size.
  - A recursive algorithm is usually exponential because it solves the same problems repeatedly.
  - However, in dynamic programming each problem solved will be brand new.

Memoization

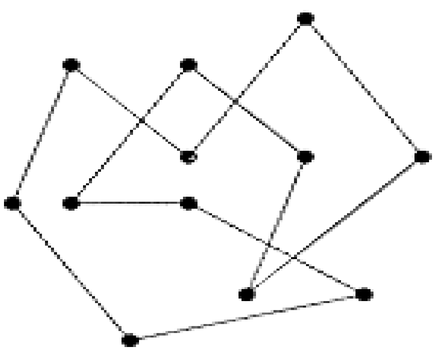
- Top-down approach with the efficiency of typical dynamic programming approach
- Maintaining an entry in a table for the solution to each subproblem
  - **memoize** the inefficient recursive algorithm
- When a subproblem is first encountered its solution is computed and stored in that table
- Subsequent “calls” to the subproblem simply look up that value

Dynamic Progamming vs. Memoization

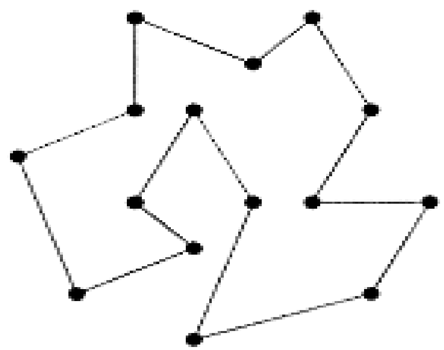
- Advantages of dynamic programming vs. memoized algorithms
  - No overhead for recursion, less overhead for maintaining the table
  - The regular pattern of table accesses may be used to reduce time or space requirements
- Advantages of memoized algorithms vs. dynamic programming
  - Some subproblems do not need to be solved

Problem 1: Minimum Triangulation

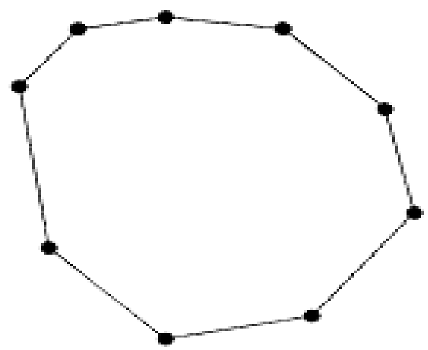
- A **polygon** is a piecewise linear closed curve in the plane, consisting of **sides** and **vertices**.
- A polygon is **simple** if it does not cross itself, and it is **convex** if given any two points on its boundary, the line segment between them lies entirely in the union of the polygon and its interior.



Polygon



Simple polygon

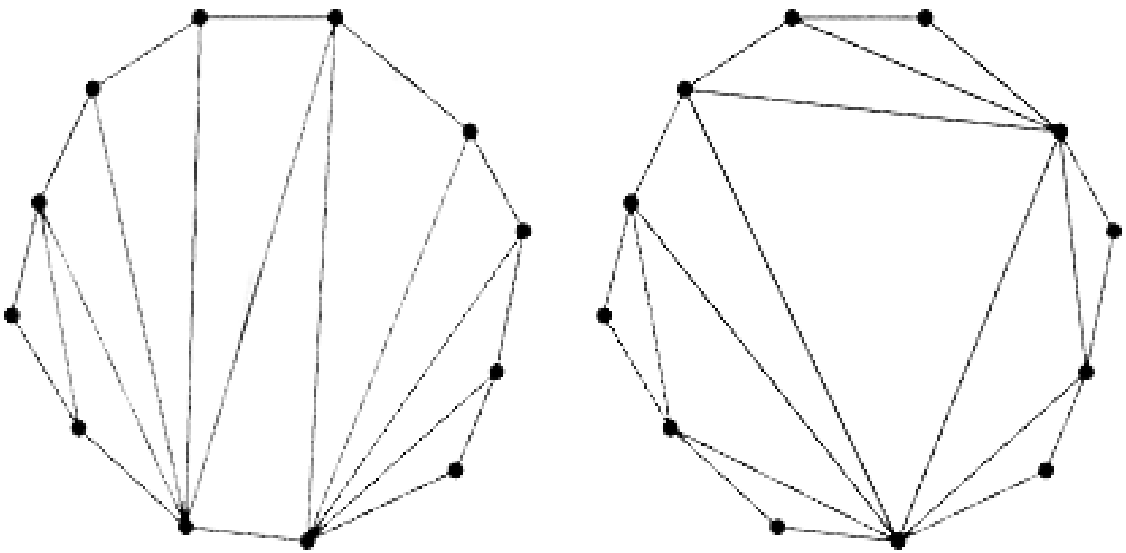


Convex polygon

Triangulations

- Given a convex polygon, assume that its vertices are labeled in counterclockwise order $P = \langle v_0, \ldots, v_{n-1} \rangle$. Assume that indexing of vertices is done modulo n, so $v_0 = v_n$. This polygon has n sides, (v_{i-1}, v_i).
- A **triangulation** of a convex polygon is a maximal set T of chords (line segments (v_i, v_j) such that $|i - j| > 1$) that do not intersect with each other. Every chord that is not in T intersects the interior of some chord in T. Such a set of chords subdivides interior of a polygon into set of triangles.

Example: Polygon Triangulation



A triangulation Another triangulation

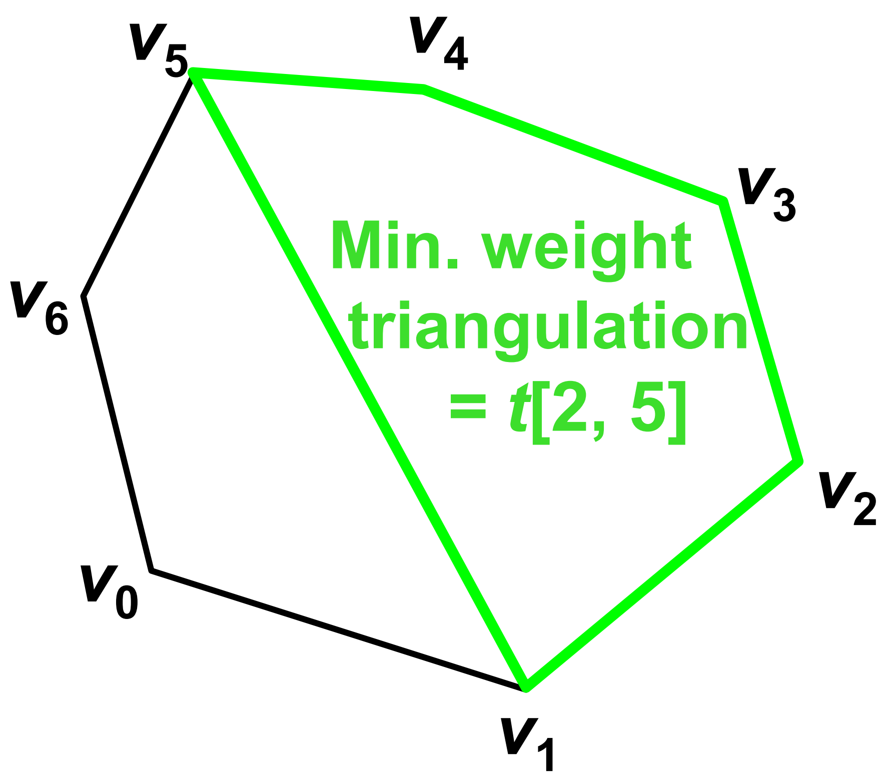
- The number of possible triangulations is exponential in n, the number of sides. The “best” triangulation depends on the applications.

Minimum-Weight Convex Polygon Triangulation

- Given three distinct vertices, v_i, v_j and v_k, we define the **weight** of the associated triangle by the weight function
$$w(v_i, v_j, v_k) = |v_i v_j| + |v_j v_k| + |v_k v_i|,$$
where $|v_i v_j|$ denotes length of the line segment (v_i, v_j).
- Define the weight of a triangulation as the sum of the weights of all its triangles.
- **The problem**: Given a convex polygon, determine the triangulation that has the minimum weight.

DP Solution

- For $0 \le i < j \le n$, let $t[i, j]$ denote the minimum weight triangulation for the subpolygon $<v_i, v_{i+1}, \ldots, v_j>$.



DP Solution (cont.)

- Observe: if we can compute $t[i, j]$ for all i and j ($0 \leq i \leq j \leq n$), then the weight of minimum weight triangulation of the entire polygon will be $t[0, n]$.
- For the basis case, the weight of the trivial 2-sided polygon is zero, implying that $t[i, i+1] = 0$ (line (v_i, v_{i+1})).

DP Solution (cont.)

- In general, to compute $t[i, j]$, consider the subpolygon $<v_i, v_i, \ldots, v_j>$, where $i < j$. One of the chords of this polygon is the side (v_i, v_j). We may split this subpolygon by introducing a triangle whose base is this chord, and whose third vertex is any vertex v_k, where $i < k < j$. This subdivides the polygon into 2 subpolygons $<v_i,\ldots v_k>$ & $<v_k,\ldots v_j>$, whose minimum weights are $t[i, k]$ and $t[k, j]$.
- We have following recursive rule for computing $t[i, j]$:

$$\boldsymbol{t[i, i+1] = 0}$$
$$\boldsymbol{t[i, j] = \min_{i<k<j} (t[i, k] + t[k, j] + w(v_i v_k v_j))}$$

Weighted-Polygon-Triangulation(*V*)

```
n ← length[V] - 1                                  // V = <v_0, v_1, ..., v_n>
for i ← 0 to n - 1                                  // initialization: O(n) time
    do t[i, i+1] ← 0
for L ← 2 to n                                      // L = length of sub-chain
    do for i ← 0 to n-L
        do j ← i + L
            t[i, j] ← ∞
            for k ← i + 1 to j - 1
                do q ← t[i, k] + t[k, j] + w(v_i, v_k, v_j)
                    if q < t[i, j]
                        then t[i, j] ← q
                             s[i, j] ← k
return t and s
```

Problem 2: String Edit Distance

- If we are to deal with inexact string matching, we must first define a cost function telling us how far apart two strings are (a distance measure).
- Hence, to detect and suggest corrections for misspellings or perform approximate string matching we often want to find the minimum edit distance between two strings.
- A reasonable distance measure minimizes the cost of the changes which must be made to convert *source* string to *target*.

Edit Operations

- There are three natural types of changes:
  - **Substitution:** Change a single character from *s* to a different character in *t*, such as changing “shot” to “spot”.
  - **Insertion**: Insert a single character from *s* to help it match target *t*, such as changing “ago” to “agog”.
  - **Deletion**: Delete a single character from source *s* to help it match target *t*, such as changing “hour” to “our”.

- We can also simply **match** two characters if they are the same.

Example: Change IAGO to AGOG

Convert IAGO to AGOG

Solution 1					Solution 2			
I	A	G	O	-	I	A	G	O
D	M	M	M	I	S	S	S	S
-	A	G	O	G	A	G	O	G

$\text{Cost}(\text{Solution 1}) = 2$
$\text{Cost}(\text{Solution 2}) = 4$

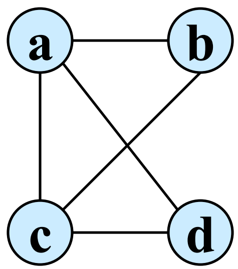
DP Solution

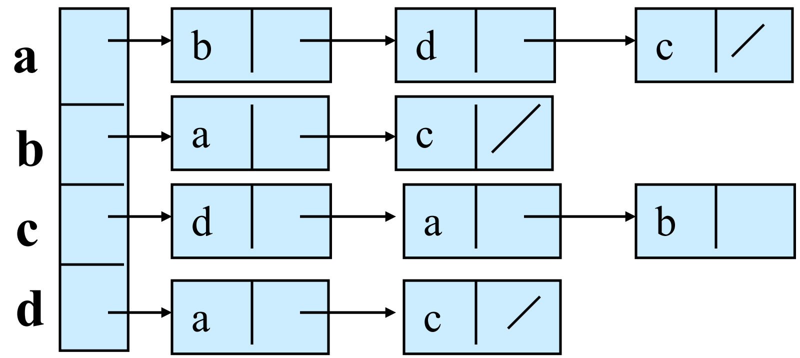
- Find the dynamic programming solution to the edit distance problem (to find the minimum edit distance of two given strings).
- To do this, first find the recursive formula using the observation that the last character in the string must either be matched, substituted, inserted, or deleted.
- Moreover, answer the typical questions about your approach that we answered in other examples we solved in class.

Graphs Representation

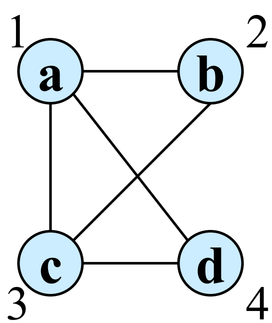
Representation of Graphs

- Two standard ways.
  - Adjacency Lists.





  - Adjacency Matrix.



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

Problem 1

- **(Exercise 22.1-7, page 593)** The ***incidence matrix*** of a directed graph G=(V,E) with no self-loops is a $|V| \times |E|$ matrix $B=(b_{ij})$ such that

$$b_{ij} = \begin{cases} -1 & \text{if edge j leaves vertex i} \\ 1 & \text{if edge j enters vertex i} \\ 0 & \text{otherwise} \end{cases}$$

Describe what the entries of the matrix product BB^T represent, where B^T is the transpose of B.

Minimum Spanning Trees (and Greedy Algorithms)

Greedy Algorithms Overview

- Like dynamic programming, used to solve optimization problems.
- When we have a choice to make, make the one that looks best *right now*.
  - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.
- Problems solvable via Greedy algorithms:

1. exhibit **optimal substructure** (like DP).
2. exhibit the **greedy-choice** property.
   - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

Typical Steps

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.

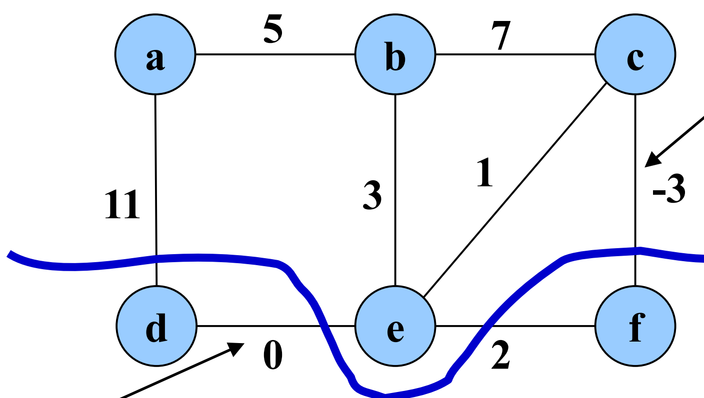
2. **Prove that there's always an optimal solution that makes the greedy choice**, so that the greedy choice is always safe.
   – Show that greedy choice and optimal solution to subproblem $\Rightarrow$ optimal solution to the problem.

3. Make the greedy choice and **solve top-down**.
   – May have to **preprocess** input to put it into greedy order.

Definitions

no edge in the set crosses the cut

cut **respects** the edge set {(a, b), (b, c)}



a **light** edge crossing cut (could be more than one)

⇐**cut** partitions vertices into disjoint sets, S and $V - S$.

this edge **crosses** the cut

one endpoint is in S and the other is in $V - S$.

Finding Safe Edges

- Suppose A is subset of some MST.
- Edge is **safe** if it can be added to A without destroying this invariant.

Theorem 23.1: Let (S, V-S) be any cut that respects A, and let (u, v) be a light edge crossing (S, V-S). Then, (u, v) is safe for A.

Corollary: If (u, v) is a light edge connecting one CC in (V, A) to another CC in (V, A), then (u, v) is safe for A.

Corollary

In general, A will consist of several connected components.

Corollary: If (u, v) is a light edge connecting one CC in (V, A) to another CC in (V, A), then (u, v) is safe for A.

MST Algorithms

- Kruskal's Algorithm
  - Starts with each vertex in its own component.
  - Repeatedly merges two components into one by choosing a light edge that connects them (i.e., a light edge crossing the cut between them).
  - Scans the set of edges in monotonically increasing order by weight.
  - Uses a disjoint-set data structure to determine whether an edge connects vertices in different components.

- Prim's Algorithm
  - Builds **one tree**, so A is always a tree.
  - Starts from an arbitrary "root" r .
  - At each step, **adds a light edge** crossing cut $(V_A, V - V_A)$ to A.
    - V_A = vertices that A is incident on.

Problem 1: Bottleneck Spanning Tree

- **(Exercise 23.3-7, page 640)** A **bottleneck spanning tree** T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G. We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in T .

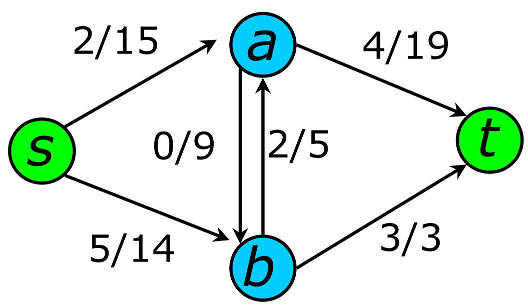
- Argue that a minimum spanning tree is a bottleneck spanning tree.
  - Prove it by contradiction…

Max Flow

Formal Max Flow Problem

- Graph $G=(V,E)$ – a **flow network**
  - Directed, each edge has **capacity** $c(u,v) \geq 0$
  - Two special vertices: ***source*** s, and ***sink*** t
  - For any other vertex v, there is a path $s \rightarrow \ldots \rightarrow v \rightarrow \ldots \rightarrow t$
- **Flow** – a function $f : V \times V \rightarrow \boldsymbol{R}$
  - *Capacity constraint*: For all $u, v \in V$: $f(u,v) \leq c(u,v)$
  - *Skew symmetry*: For all $u, v \in V$: $f(u,v) = -f(v,u)$
  - *Flow conservation:* For all $u \in V - \{s, t\}$: $\sum_{v \in V} f(u,v) = f(u,V) = 0$, or

$$\sum_{v \in V} f(v,u) = f(V,u) = 0$$



We want to find a flow of maximum value from the source to the sink (Denoted by |f|)

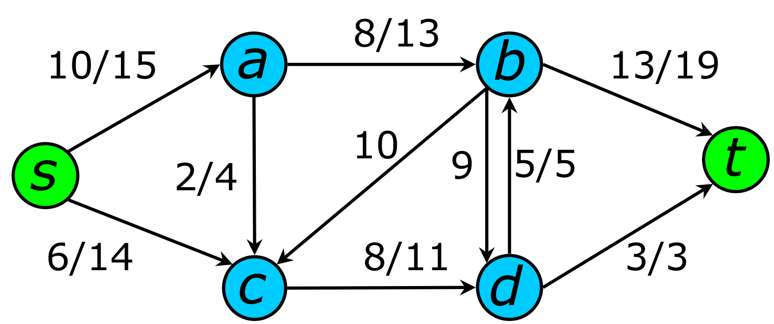
Ford-Fulkerson method

- Contains several algorithms:
  - Residue networks
  - Augmenting paths
    - Find a path p from s to t (**augmenting path**), such that there is some value $x > 0$, and for each edge (u,v) in p we can add x units of flow
      - $f(u,v) + x \leq c(u,v)$

Augmenting Path?

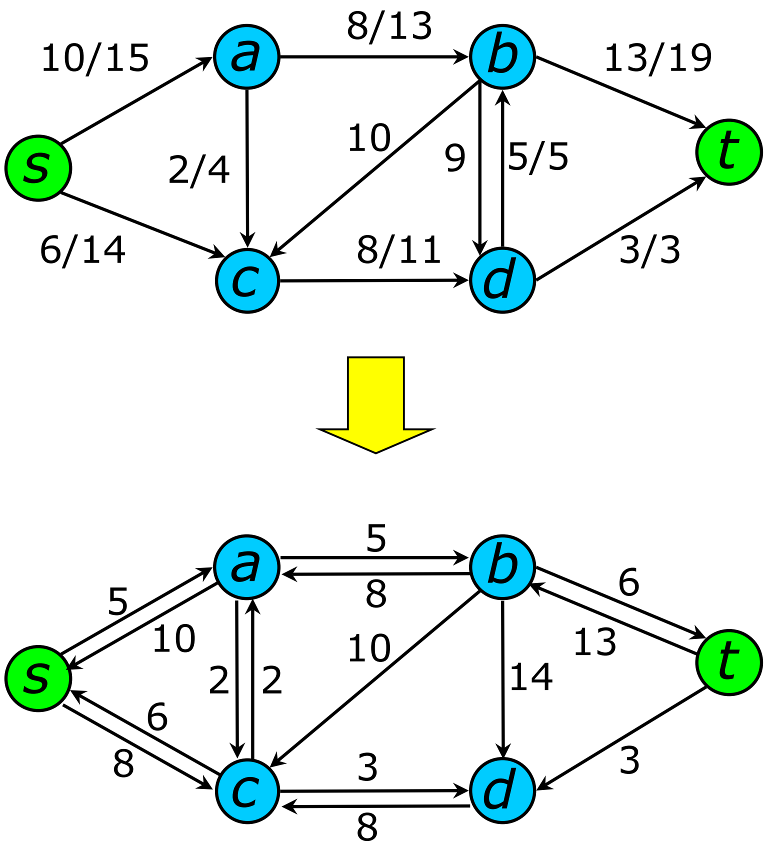
FORD-FULKERSON-METHOD(G, s, t)

initialize flow f to 0
while there exists an augmenting path p
    do augment flow f along p
return f



Residual Graph

- Compute the residual graph of the graph with the following flow:



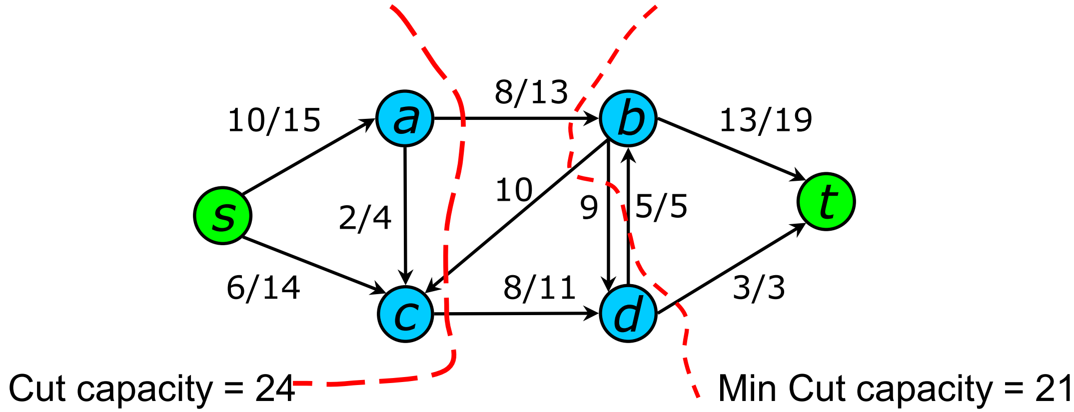
Residual Capacity and Augmenting Path

- Finding an Augmenting Path
  - Find a path from s to t in the residual graph
  - The *residual capacity* of a path p in G_f:
    $c_f(p) = \min\{c_f(u,v): (u,v) \text{ is in } p\}$
    - i.e. find the minimum capacity along p
- Doing augmentation: for all (u,v) in p, we add $c_f(p)$ to $f(u,v)$ (and subtract it from $f(v,u)$)

Resulting flow is a valid flow with a larger value

Min Cut

- **Min Cut** – a cut with the smallest capacity of all cuts.

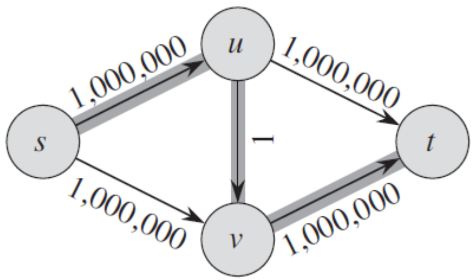


- Lemma: If (S,T) is a min cut, then the max flow = f(S,T)
  - i.e. the value of a max flow is equal to the capacity of a min cut.

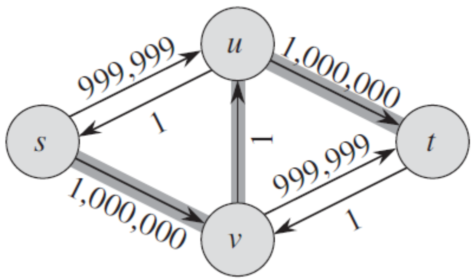
- Max Flow / Min Cut Theorem: These conditions are equivalent
  1. $|f| = c(S,T)$ for some cut (S, T)
  2. f is a maximum flow in G
  3. The residual network G_f contains no augmenting paths.

Worst Case Running Time

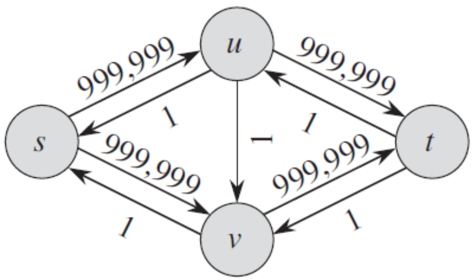
- Assuming integer flow
- Each augmentation increases the value of the flow by some positive amount.
- Augmentation can be done in O(E).
- Total worst-case running time O(E|f*|), where f* is the max-flow found by the algorithm.
- Example of worst case:



Augmenting path of 1



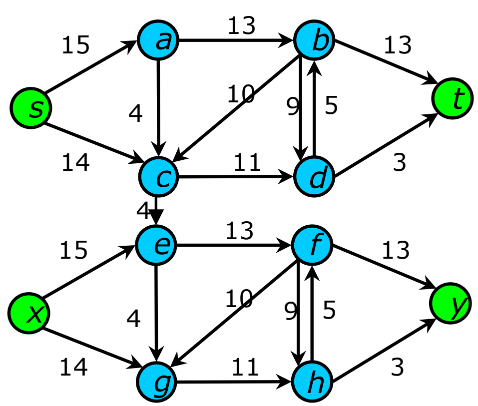
Resulting Residual Network

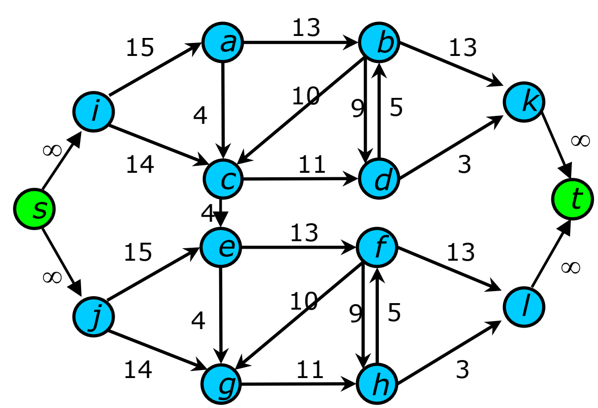


Resulting Residual Network

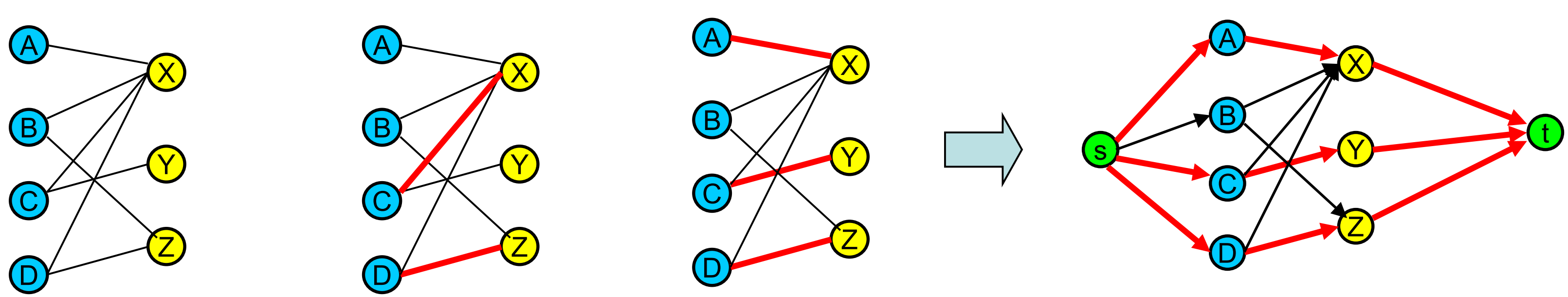
Applications

- Multiple Sources or Sinks





- Bipartite Matching: Maximum Matching



Men Women A maximal matching Optimal matching

Problem

- **(Exercise 26.3-3, page 735)** Let G=(V,E) be a bipartite graph with vertex partition V=L ∪ R, and let G' be its corresponding flow network (you can imagine that the source is connected to all the vertices in L and every vertex in R is connected to the sink). Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

NP Completeness

Summary of Definitions

- **Intractable**: algorithms running longer than polynomial time.
- **Decision Problems**: problems with solutions of yes/no.
- **P**: set of problems solvable in polynomial time.
- **NP**: set of problems with verifiable solutions in polynomial time.
- **NP-Complete**: a problem in NP that any problem in NP is polynomial-time reducible to it.
  NP-Hard: a problem that any problem in NP is polynomial-time reducible to it (not necessarily in NP).

Reducibility

- The crux of NP-Completeness
- We say *P is reducible to Q* if we can transform any instance of P to an instance of Q such that the answer to those instances of problems are the same.
- We use the notation **$P \leq_p Q$** to show that problem P is reducible to problem Q in polynomial time.
  – In other words: *P is no harder than Q to solve*.

Why Prove NP-Completeness?

- Though nobody has proven that **P != NP**, if you prove a problem NP-Complete, most people accept that it is probably intractable

- Therefore it can be important to prove that a problem is NP-Complete
  - Don’t need to come up with an efficient algorithm
  - Can instead work on *approximation algorithms*

Sample Question about Definitions

- *What, intuitively, does it mean if we can reduce problem P to problem Q?*
  - P is “no harder than” Q
- *How do we reduce P to Q?*
  - Transform instances of P to instances of Q in polynomial time s.t. Q: “yes” iff P: “yes”
- *What does it mean if Q is NP-Hard?*
  - Every problem $P \in \mathbf{NP} \leq_p Q$
- *What does it mean if Q is NP-Complete?*
  - Q is NP-Hard and $Q \in \mathbf{NP}$

Questions in NP-Completeness

- You should be able to (including but not limited to!)...
  - answer questions regarding the definitions related to NP-Completeness
  - show that why a transformation (used to reduce a problem to another) is a valid transformation
  - prove that a specific problem is reducible to another one by designing a polynomial transformation algorithm and proving its correctness

What Does the Final Exam Include?

Final Exam

- The topics are as follows. The suggested points allocation is just an approximation and may vary in the final exam.
  - Asymptotic Analysis: 15 — from midterm 1
  - Recurrence: 15 — from midterm 1
  - Sorting/Hashing/Trees: 10 — from midterm 1/2
  - Dynamic Programming: 15 — from midterm 2
  - Graph Representation (Greedy): 10 — new material
  - Minimum Spanning Trees: 10 — new material
  - Max Flow: 10 — new material
  - NP-Completeness: 15 — new material
  - Extra credit… :-? 10 — who knows?