CSC373: Algorithm Design, Analysis and Complexity
Fall 2017

Allan Borodin

October 11, 2017

Week 5 : Annoucements

Term test 1 will be held October 12, 5-6. This is a common test with the daytime section. Note: Since the room we were given yesterday for the common room is so large, all students taking the test 5:10-6 PM will take the test in the Exam Center, Room 200. Please be on time as the test ends exactly at 6PM.
The term test will be "mostly" based on some questions occuring in Assignment 1. More specifically, there will be three questions: one divide and conquer, one greedy and one dynamic programming.
Question 1 (the divide and conquer question) takes time to read and might be the most troublesome so you might do well to first proceed to the other two questions.
Aids allowed: one 8 by 11 sheet of paper, two sides, hand written.

Note about Markus: As some students found out, Markus is very strict as to the due date and time. Do not submit assignments late. You can submit early and then replace if needed. fits on the space allowed.

Next assignments I expect to post the start of Assignment 2 tomorrow.

Unicef Fund Raiser

The student volunteer will be here to collect next week. If you contribute enough, I wll be in costime the following week.

Dear Professor,

I am writing to you on behalf of the University of Toronto UNICEF group to request your assistance for our upcoming yearly event "Prof-in-a-Prop," which is a part of our National UNICEF Day campaign.

This year, "Prof-in-a-Prop" will run from October 1st to October 31st. All of the proceeds from our Prof-in-a-Prop campaign will go directly to UNICEF Canada and thus directly to children in need in the developing world.

The main purpose of the event is to raise funds for UNICEF and to create awareness for UNICEF Canada's campaigns worldwide. We are requesting the participation of numerous professors on campus to help us make this event a success.

The "Prof-in-a-Prop" event sends U of T UNICEF volunteers to stop by lecture halls to collect donations from students for UNICEF's campaigns. If the class donates enough money, the professor will wear a Halloween costume – such as bunny ears, a banana or a hotdog suit – for an entire lecture. This year, we have set our target goal at $1 per student in each participating class.

In the past, we have found that students are very receptive to this idea, and the event has consistently been our major fundraiser for the year. It also allows us to engage large numbers of students on campus, giving us the opportunity to educate them about various global issues.

As National UNICEF Day is October 31st, we always appreciate the opportunity to spread the word about UNICEF's work in over 190 countries worldwide and how students can contribute to the cause.

Therefore, we at U of T UNICEF ask if you would be interested in participating in our "Prof-in-a-Prop" campaign. If you would be willing to take part in our event, please contact me at the email address or phone number listed below.

If you have any questions or concerns, I would be more than happy to speak with you at length.

Thank you for your time and consideration.

Sincerely,

Tea Cimini
Co-President
University of Toronto UNICEF
tea.cimini@mail.utoronto.ca

Todays agenda

- Questions on the assignment or test?
- Some review of basic graph concepts. See texts: CLRS, chapter 22; KT, chapter 3, DPV chapter3. **Note:** We have already been doing topics in chapters 23 (MST) and 24 in CLRS (Shortest Paths) and some of the more basic graph trheoy concepts (including breadth first and depth first search) is a topic in CSC263 or CSC265.
- Finish up some discussion and applications of max flow.
- Begin complexity theory and NP-completeness. **Note:** In the winter/spring 2017 term, Dr. Pankratov first did LP theory and then NP completeness. I am inverting the order and doing LP theory and applications following complexity theory and NP completeness.

Review of basic graph theory concepts

Here follow some basic graph theory definitions and concepts. I again assume that much of this is familiar from say CSC263 or CSC265 but will entertain questions if anything does not look familiar or understandable. I also again note that this material is discussed in all the texts.

- Graphs vs directed graphs (digraphs)
- edge and vertex weighted graphs
- Special graphs: Bi-partite graphs; trees; rooted trees, DAGs
- Connectivity, connected components in graphs and strong connectivity in directed graphs.
- Breadth first search and depth first search.
- Complement of a graph.
- Graph problems: shortest paths, MST, matchings; vertex graph coloring, max independent sets

Review of Flow Networks, Max flow-Min Cut

A flow network is a tuple $F = (G, s, t, c)$ where

- $G = (V, E)$ is a "bidirectional graph"
- the source s and the terminal t are nodes in V
- the capacity $c : E \to \mathbb{R}^{\geq 0}$

A flow is a function $f : E \to \mathbb{R}$ satisfying the following properties:

1. Capacity constraint: for all $(u, v) \in E$,

$$f(u, v) \leq c(u, v)$$

2. Skew symmetry: for all $(u, v) \in E$,

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

3. Flow conservation: for all nodes u (except for s and t),

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

A flow f and its residual graph

- Given any flow f for a flow network $F = (G, s, t, c)$, we define the residual graph $G_f = (V, E_f)$, where
  - V is the set of vertices of the original flow network F
  - E_f is the set of all edges e having positive residual capacity

$$c_f(e) = c(e) - f(e) > 0.$$

- Note that $c(e) - f(e) \geq 0$ for all edges by the capacity constraint.

Note

With our convention of negative flows, even a zero capacity edge (in G) can have residual capacity.

- The basic concept underlying the Ford-Fulkerson algorithm is an **augmenting path** which is an s-t path in G_f.
  - Such a path can be used to augment the current flow f to derive a better flow f'.

The residual capacity of an augmenting path

- Given an augmenting path π in G_f, we define its residual capacity $c_f(\pi)$ to be the
$$\min_e \{c_f(e) \mid e \in \pi\}$$

- **Note:** the residual capacity of an augmenting path is itself is greater than 0 since every edge in the path has positive residual capacity.

- We can think of an augmenting path as defining a flow f_π (in the "residual network"):
$$f_\pi(u,v) = \begin{cases} c_f(\pi) & \text{if } (u,v) \in \pi \\ -c_f(\pi) & \text{if } (v,u) \in \pi \\ 0 & \text{otherwise} \end{cases}$$

The Ford-Fulkerson scheme

The Ford-Fulkerson scheme

/* *Initialize* */
$f := 0$; $G_f := G$
WHILE there is an augmenting path π in G_f
  $f := f + f_\pi$
    /* *Note this also changes* G_f */
ENDWHILE

Note

I call this a "scheme" rather than an algorithm since we haven't said how one chooses an augmenting path (as there can be many such paths)

Issues concerning the Ford-Fulkerson scheme

- Does it matter how we choose an augmenting path for termination and speed of termination?

- That is, in the terminology of the local search paradigm, does it matter how we are choosing the $S' \in Nbhd(S)$?
  - **Answer:** YES, it matters but there are good ways to choose augmenting paths so that the algorithm is poly time.
  - Note that the $Nbhd(S)$ here can be of exponential size but that is not a problem as long as we can efficiently search for solutions in the local neighbourhood.

- Upon termination how good is the flow?
  - **Answer:** The flow is an optimal flow. This will be proved by the max flow - min cut theorem.
  - Note that this is unusual in that for most local search applications a local optimum is usually not a global optimum.

The max-flow min-cut theorem

- A cut (really an s-t cut) in a flow network is a partition (S, T) of the nodes such that $s \in S$ and $t \in T$.
- We define the capacity $c(S, T)$ of a cut as

$$\sum_{u \in S \text{ and } v \in T} c(u, v)$$

- We define the flow $f(S, T)$ across a cut as

$$\sum_{u \in S \text{ and } v \in T} f(u, v)$$

- It should be clear is that

$$f(S, T) \leq c(S, T)$$

for all cuts (S, T) (by the capacity constraint for each edge).

The max-flow min-cut theorem

The following three statements are equivalent:

1. f is a max-flow
2. There are no augmenting paths w.r.t. flow f (i.e. no s-t path in G_f)
3. There exists some cut (S, T) satisfying $val(f) = c(S, T)$
   - Hence this cut (S, T) must be a min (capacity) cut since $val(f) \leq c(S, T)$ for all cuts.

Note

The name follows from the fact that the value of a max-flow $=$ the capacity of a min-cut

The proof

Last week we proved the theorem by showing

1. ❶ $\Rightarrow$ ❷ If there is an augmenting path (w.r.t. f), then f can be increased and hence not optimal.

The proof

Last week we proved the theorem by showing

1. ⇒ 2 If there is an augmenting path (w.r.t. f), then f can be increased and hence not optimal.

2. ⇒ 3 Consider the set S of all the nodes reachable from s in the residual graph G_f.
   - Note that t cannot be in S and hence $(S, T) = (S, V - S)$ is a cut.
   - We also have $c(S, T) = val(f)$ since $f(u, v) = c(u, v)$ for all edges (u, v) with $u \in S$ and $v \in T$.

The proof

Last week we proved the theorem by showing

1. ⇒ 2 If there is an augmenting path (w.r.t. f), then f can be increased and hence not optimal.

2. ⇒ 3 Consider the set S of all the nodes reachable from s in the residual graph G_f.
   - Note that t cannot be in S and hence $(S, T) = (S, V - S)$ is a cut.
   - We also have $c(S, T) = val(f)$ since $f(u, v) = c(u, v)$ for all edges (u, v) with $u \in S$ and $v \in T$.

3. ⇒ 1 Let f' be an arbitrary flow. We know $val(f') \leq c(S, T)$ for any cut (S, T) and hence $val(f') \leq val(f)$ for the cut constructed in 2.

Note: There can be many min cuts with regard to a given max flow.

Termination and the runtime of Ford-Fulkerson

Observation with regard to integral capacities

Each augmenting path has residual capacity at least one.

- If all capacities are integral, the max-flow min-cut theorem along with the above observation ensures that with integral capacities, Ford-Fulkerson must always terminate and the number of iterations is at most:

$$C = \text{the sum of edge capacities leaving } s.$$

Notes

- There are bad ways to choose augmenting paths such that with irrational capacities, the Ford-Fulkerson algorithm will not terminate.
- However, even with integral capacities, there are bad ways to choose augmenting paths so that the Ford-Fulkerson algorithm does not terminate in polynomial time.

Some ways to achieve polynomial time

- Choose an augmenting path having shortest distance: This is the Edmonds-Karp method and can be found in CLRS. It has running time $O(nm^2)$, where $n = |V|$ and $m = |E|$.
- There is a "weakly polynomial time" algorithm in KT
  - Here the number of arithmetic operations depends on the length of the integral capacities.
  - It follows that always choosing the largest capacity augmenting path is at least weakly polynomial time.
- There is a history of max flow algorithms leading to a recent $O(mn)$ time algorithm (see http://tinyurl.com/bczkdfz).
- Although not the fastest, Dinitz's $O(n^2m)$ time algorithm is interesting.
  - A shortest augmenting-path method based on the concept of a blocking flow in the leveled graph.
  - Has an additional advantage (i.e. an improved bipartite matching bound) beyond the somewhat better running time of Edmonds-Karp.

Dinitz's algorithm

Definition

- Define $level(u)$ = length of shortest path from s to u in G_f.
- Let the "leveled graph" w.r.t the residual graph G_f be the graph $L_f = (\hat{V}, \hat{E})$ where
  - $\hat{V} = \{v \mid v \text{ reachable from } s\}$
  - $(u, v) \in \hat{E}$ if and only if $level(v) = level(u) + 1$ in G_f.
- A blocking flow $\tilde{f}$ is a flow such that every s-t path in L_f has a saturated edge (i.e. an edge e such that $\tilde{f}(e) = c_f(e)$).

Dinitz's algorithm

1: Initialize $f(e) = 0$ for all $e \in E$.
2: **while** t is reachable from s in G_f **do**
3: Construct L_f from G_f
4: Find a blocking flow $\tilde{f}$ w.r.t. L_f and set $f := f + \tilde{f}$
5: **end while**
6: % There's no more augmenting path

Proof Sketch

Claims

1. The algorithm halts in at most $n - 2$ iterations.
2. A blocking flow in the leveled graph can be found in time $O(mn)$.

Proof.

Let f be a flow. Let f' be the updated flow after one iteration of Dinitz's algorithm, and let $level'$ be the updated level w.r.t. the graph $G_{f'}$.

1. The proof of this claim rests on two facts:
   - For every node $v \in L_{f'}$, $level'(v) \geq level(v)$ since every edge in $L_{f'}$ is either an edge in G_f or the reverse of an edge in L_f.
   - Since f' was a blocking flow, $level'(t) > level(t)$.

Proof Sketch

Claims

1. The algorithm halts in at most $n - 2$ iterations.
2. A blocking flow in the leveled graph can be found in time $O(mn)$.

Proof.

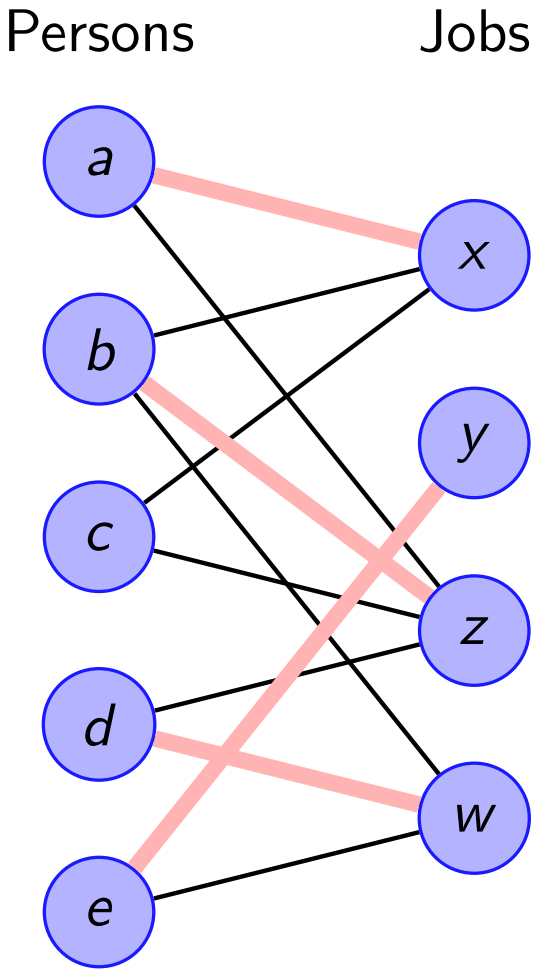
Let f be a flow. Let f' be the updated flow after one iteration of Dinitz's algorithm, and let $level'$ be the updated level w.r.t. the graph $G_{f'}$.

1. The proof of this claim rests on two facts:
   - For every node $v \in L_{f'}$, $level'(v) \geq level(v)$ since every edge in $L_{f'}$ is either an edge in G_f or the reverse of an edge in L_f.
   - Since f' was a blocking flow, $level'(t) > level(t)$.
2. The leveled graph can be computed in $O(m)$. And using depth first search we can compute a blocking path in time $O(mn)$.

An application of max-flow: the maximum bipartite matching problem

The maximum bipartite matching problem

- Given a bipartite graph $G = (V, E)$ where
  - $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \varnothing$
  - $E \subseteq V_1 \times V_2$
- **Goal:** Find a maximum size matching.



- We do not know any efficient DP or greedy optimal algorithm for solving this problem.
- But we can efficiently reduce this problem to the max-flow problem.

The reduction

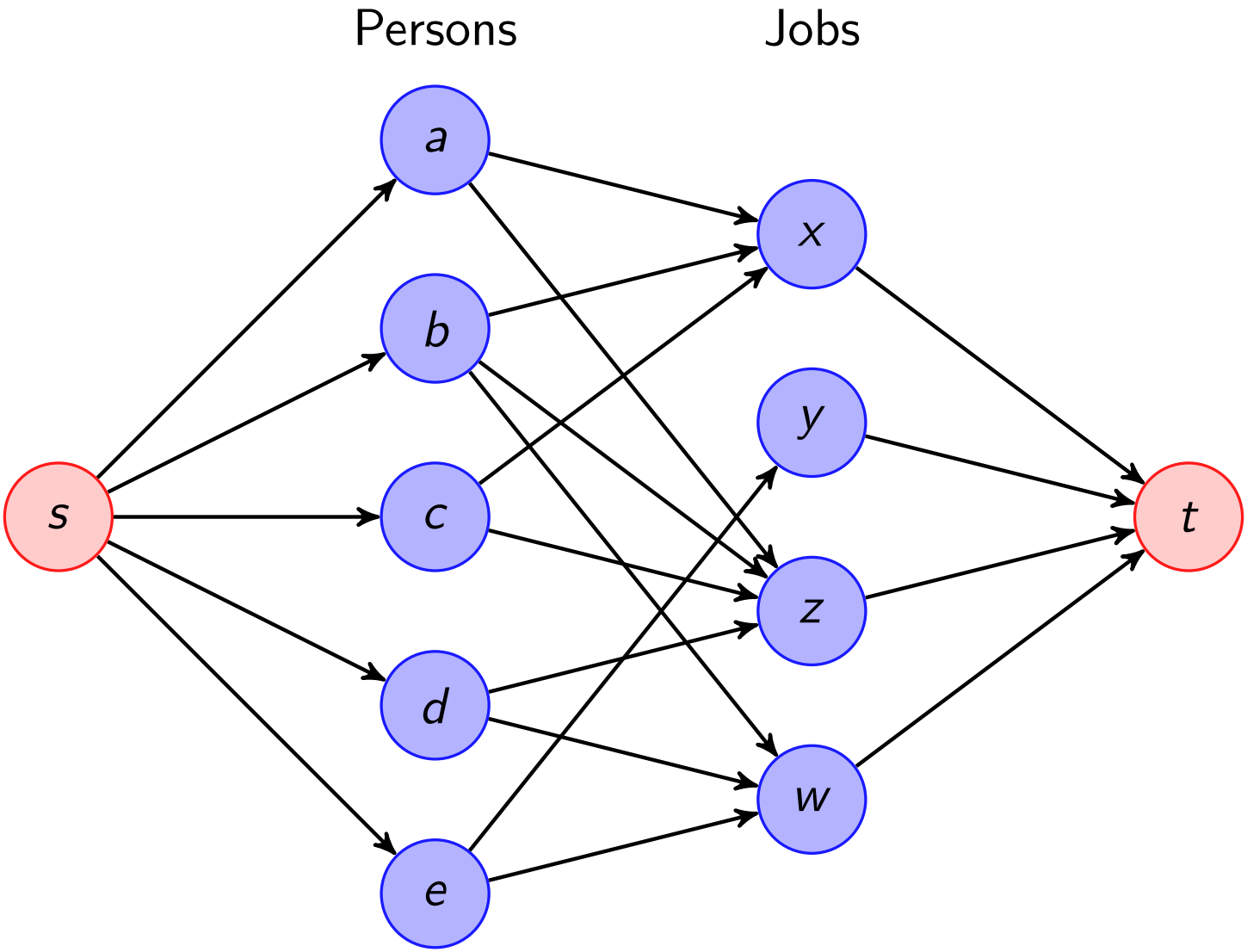


Figure: Assign every edge of the network flow a capacity 1.

The reduction preserves solutions

Claims

1. Every matching M in G gives rise to an integral flow f_M in the newly constructed flow network F_G with $val(f_M) = |M|$
2. Conversely every integral flow f in F_G gives rise to a matching M_f in G with $|M_f| = val(f)$.

Let $m = |E|, n = |V|$

- Time complexity: $O(mn)$ using any Ford Fulkerson algorithm since the max flow is at most n and $C = n$ since all edge capacities are integral and set to 1.
- Dinitz's algorithm can be used to obtain a runtime $O(m\sqrt{n})$.

A few more comments on this reduction

- When we get to our next big topic (NP completeness), we will be focusing on decision problems and as a decision problem we have $|M| \geq k$ iff $val(f_M) \geq k$.
- The reduction we are using is very efficient (linear time in the representation of the graph) and it is a special type of polynomial time reduction which we will call a polynomial time transformation.

Alternating and augmenting paths in graphs

There is a graph theoretic concept of an augmenting path relative to a matching (in an arbitrary graph).

- An alternating path π relative to a matching M is one whose edges alternate between edges in M and edges not in M.
- An augmenting path is an alternating path that starts and ends with an edge not in M.

- The reduction provides a 1-1 correspondence between augmenting paths in the bipartite G wrt. M_f and augmenting paths in G_{f_M}.

Can this reduction be extended to a maximum edge weighted matching

- In the weighted bipartite matching bipartite matching problem, we are given an edge weighted bipartite graph $G = (V, E)$ with $V = V_1 \cup V_2$ (a disjoint partition) and with say integral weights $w : E \to \mathbb{N}$
- Goal: Compute a matching M so as to maximize $\sum_{e \in M} w(e)$.
- A "reaasonable" idea is to extend the unweighted reduction by again forming a flow network with distinguished source s and terminal t nodes.
- We could then set the capacities of the edges as follows:
  - $c(x, y) = w(x, y)$ for all $(x, y) \in E$
  - $c(s, x) = \max_y \{w(x, y) : x \in V_1\}$
  - $c(y, t) = \max_x \{w(x, y) : y \in V_2\}$
- Why doesn't this work?

Edge disjoint paths: another similar max flow application

- A problem of interest in fault tolerant networks is to ensure that there are sufficiently many edge disjoint paths between any two given nodes.
- Given a directed graph $G = (V, E)$ with distinguished source s and terminal t nodes, the **goal** is to compute the the maximum number of edge disjoint paths from s to t.
- Similar to the bipartite matching transformation, we view G as a flow network $\mathcal{F}_G$ by setting the capacity of all edges equal to 1.
- Once again, because of integrality and unit capacities, we can argue that there are k edge-disjoint paths in G iff $\mathcal{F}_G$ has max (integral) flow k.
- The max flow-min cut theorem implies Menger's theorem which states that the maximum number of edge-disjoint $s - t$ paths in a directed graph is equal to the minimum number of edges in an $s - t$ cut.
- The same theorem holds for undirected graphs.

The $\{0,1\}$ metric labelling problem.

We consider one more application of max flow-min cut, the $\{0,1\}$ metric labelliing problem, discussed in sections 12.6 and 7.10 of the KT text.

- This problem is a special case of the following more general metric labelling problem defined as follows:
- The input is an edge weighted graph $G = (V, E)$, a set of labels $L = \{a_1, \ldots, a_r\}$ in a metric space with distance metric d, and functions $w : E \to \mathbb{R}^{\geq 0}$ and $p : V \times L \to \mathbb{R}^{\geq 0}$.
- $\beta(u, a_j)$ is the benefit of giving label a_j to node u.
- Goal: Find a labelling $\lambda : V \to L$ of the nodes so as to maximize

$$\sum_u \beta(u, \lambda(u)) - \sum_{(u,v)\in E} w(u, v) \cdot d((\lambda(u), \lambda(v))$$

The $\{0,1\}$ metric labelling problem.

We consider one more application of max flow-min cut, the $\{0,1\}$ metric labelliing problem, discussed in sections 12.6 and 7.10 of the KT text.

- This problem is a special case of the following more general metric labelling problem defined as follows:
- The input is an edge weighted graph $G = (V, E)$, a set of labels $L = \{a_1, \ldots, a_r\}$ in a metric space with distance metric d, and functions $w : E \to \mathbb{R}^{\geq 0}$ and $p : V \times L \to \mathbb{R}^{\geq 0}$.
- $\beta(u, a_j)$ is the benefit of giving label a_j to node u.
- Goal: Find a labelling $\lambda : V \to L$ of the nodes so as to maximize

$$\sum_u \beta(u, \lambda(u)) - \sum_{(u,v)\in E} w(u, v) \cdot d((\lambda(u), \lambda(v))$$

- For example, the nodes might represent documents, the labels are topics, and the edges are links between documents weighted by the importance of the link.
- When there are 3 or more labels, the problem is NP-hard even for the case of the $\{0,1\}$ metric d where $d(a_i, a_j) = 1$ for $a_i \neq a_j$.

The labelling problem with 2 labels

- When there are only 2 labels, the only metric is the $\{0,1\}$ metric.
- While the labelling problem is NP-hard for 3 or more labels (even for the $\{0,1\}$ metric), it is solvable in polynomial time for 2 labels by reducing the problem to the min cut problem. This is what is being done in Section 7.10 of the KT text for a special graph relating to pixels in an image.
- Later we may see an approximation algorithm for the $\{0,1\}$ metric with 3 or more labels that uses local search and a reduction to min cut.
- Informally, the idea is that we can construct a flow network such that the nodes on the side of the source node s will correspond to (say) nodes labled a and the nodes on the side of the terminal node t will correspond to the nodes labeled b.
- We will place capacities between the source s and other nodes to reflect the cost of a "mislabel" and similarly for the terminal t.
- The min cut will then correspond to a min cost labelling.

The reduction for the two label case

For the two label case (labels {a,b}), we can let $a_u = \beta(u, a)$ and $b_u = \beta(u, b)$.
The goal is to maximize $\sum_{u \in A} a_u + \sum_{v \in B} b_v - \sum_{(u,v) \in A \times B} w(u, v)$
Leting $Q = \sum_{u \in V} a_u + b_u$, the goal is then equivalent to maximizing
$Q - \sum_{u \in A} b_u - \sum_{v \in B} a_v - \sum_{(u,v) \in A \times B} w(u, v)$
Equivalently, to minimizing
$\sum_{u \in A} b_u + \sum_{v \in B} a_v + \sum_{(u,v) \in A \times B} w(u, v)$

- We transform this problem to a min cut problem as follows: construct the flow network $\mathcal{F} = (G', s, t, c)$ such that $G' = (V', E')$
- $V' = V \cup \{s, t\}$
- $E' = \{(u, v) | u \neq v \in V\} \cup \{(s, u) | u \in V\} \cup \{(u, t) | u \in V\}$
- $c(i, j) = c(j, i) = p_{i,j}; c(s, i) = a_i; c(i, t) = b_i$

Claim:

For any partition $V = A \cup B$, the capacity of the cut $c(A, B) = \sum_{u \in A} b_i + \sum_{v \in B} a_j + \sum_{(u,v) \in A \times B} w_{u,v}$.

New topic: complexity, theory, the class NP and NP completeness

We are now going to begin our discussion of a topic I have been alluding to all term.

Inrtroducing complexity theory and NP completeness

- We begin complexity theory and *NP* completeness
- Warning: today's lecture starts with many definitions and conventions and hence may seem boring. I apologize in advance but we need to set the stage.
- I am going to use some slides from a recent talk by Stephen Cook to hopefully better motivate the topic.

What is computable?

- One of the greatest ideas in computer science is that we basically all agree on what it means for a (discrete) computational problem to be computable.
- Namely, we equate the intuitive concept of "computable" with the mathematically precisely defined concept of "Turing computable"
  - Turing computable = computable by a precise model called a Turing machine, named for Alan Turing.
- This is the so-called Church-Turing thesis (from 1936) as both Church and Turing independently established formal definitions for the concept of a computable function.
  - These formulations turned out to be equivalent.
- Although the Church-Turing thesis can never be proved, it is almost universally believed. **Why and why?**
- Using diagonalization (the same proof technique used to show that there are more reals than rationals), Turing showed that there are (explicitly defined) functions that are not (Turing) computable.

Turing Machines

- In 1936, Alan Turing ("The Father of Computer Science") published

  "On Computable Numbers with an Application to the Entscheidnungsproblem"

- He introduced a mathematical model of a computer (before modern computers had been invented), which we call a "Turing Machine".
- All *present day* computers can be simulated by Turing machines. The hardware gets faster, but they are not fundamentally different.
- There may be fundamental limitations on how fast and large computers can be.
- Say the limit is one bit per electron, with the computer clock limited by the speed of light between atoms.
- With such limits, even with the whole earth as a computer, a brute force search through all 300-bit strings would take universe lifetimes.

What does it mean to be efficiently computable?

- Let n denote a measure of the input/output "size" of the problem. One can also use diagonalization to show that

For any time bound $T(n)$, there are computable functions not computable within time T(n) for problems of size n.

- Following Cobham and Edmonds (circa 1965), we will equate the intuitive concept of "efficiently computable" with computable in polynomial time (i.e. time bounded by a polynomial function of the encoded length of the inputs and outputs).
  - This has sometimes been called the Extended Church-Turing Thesis.
  - This hypothesis is not literally believed in contrast to the Church-Turing Thesis. **Why?**
  - But it is an abstraction that has led to great progress in computing.
- **Informal claim:** Any function (polynomial time) computable is (polynomial time) computable by a Turing machine.
- For the time being we will not define a precise computational model such as the Turing machine model.

Encoding of inputs and outputs

- We are always assuming that inputs and outputs are encoded as strings over some finite alphabet S with at least 2 symbols.
  - S^n denotes the set of all finite strings of length n over the alphabet S.
  - The empty string is the only string of length 0.
  - $S^* = \bigcup_{n \geq 0} S^n$ denotes the set of all finite strings over the alphabet S
- We can use as many symbols as we want but 2 suffices for our purpose. (**Note:** finitely many symbols on a keyboard.)
- We can also encode in unary, but that causes an exponential blowup in representation.
- We can encode a set of inputs or outputs $w_1, \ldots, w_n$ by having a special symbol (say #) to separate the inputs but again this can all be encoded back into 2 symbols.
- If we need to distinguish components of an input, we can denote such an encoding of many inputs (or outputs) as $\langle w_1, \ldots, w_n \rangle$.
  - We will usually not need to be so careful about the distinction between an object G and its encoding $\langle G \rangle$.

What are the objects of study?

- Since it suffices to encode all inputs as finite strings over the alphabet $S = \{0, 1\} = \{\text{false}, \text{true}\}$, we often refer to the complexity issues of such computations as Boolean complexity theory.
  - This is in contrast to, say, arithmetic complexity theory, or the theory of real valued computation.
- Our general objects of study (for Boolean complexity theory) are the computation of:
  1. Functions $f : S^* \to S^*$
  2. Search problems
     - ★ Let $R(x, y) \subseteq S^* \times S^*$ be a relation.
     - ★ Given x, need to find a y satisfying $R(x, y)$ or say that no such y exists.
  3. Optimization problems
     - ★ Let $R(x, y) \subseteq S^* \times S^*$ be a relation, and $c : S^* \times S^* \to \Re$ be an objective function.
     - ★ Given x, we need to find a y satsifying $R(x, y)$ so as to minimize (or maximize) $c(x, y)$ or say that no such y exists.
  4. Decision problems
     - ★ Let $R(x, y) \subseteq S^* \times S^*$ be a relation.
     - ★ Given x, determine if there exists a y satisfying $R(x, y)$.

Polynomial time computable functions

- We say that a function $f : S^* \to S^*$ is computable in time $T(\cdot)$ if:
  1. There is an algorithm (to be precise a Turing machine or an idealized RAM program with an appropriate instruction set)
  2. For all inputs $w \in S^*$, the algorithm halts using at most $T(n)$ "time" where $n = |w| + |f(w)|$.
- We can define polynomial time search problems, optimization problems and decision problems similarly.
- We will generally not be dealing with functions where $|f(w)| > |w|$. So n will be the length of the input.
- In particular, for decision problems, n will always refer to the length of the encoded input.
- We let P denote the class of decision problems that are solvable in polynomial time. We sometimes abuse notation and also use P to denote any problem computable in polynomial time.

Computational Complexity Theory

- Classifies problems according to their computational difficulty.
- The class P (Polynomial Time) [Cobham, Edmonds, 1965]
- P consists of all problems that have an efficient (e.g. $n, n^2 \ldots$) algorithm. (n is the input length)

Examples in P

- Addition, Multiplication, Square Roots
- Shortest Path (Google Maps)
- Network flows (Internet Routing)
- Pattern matching (Spell Checking, Text Processing)
- Fast Fourier Transform (Audio and Image Processing, Oil Exploration)
- Recognizing Prime Numbers [Agrawal-Kayal-Saxena 2002]
- ...

Polynomial time continued

- Until we have a precise computation model, we cannot define time.
- But we can proceed informally by assuming that we all have a good intuition as to our concept of computational time.
- One of the nice properties of a Turing machine model is that the concept of a computational step and hence time is well defined.
- **Warning:** If say a RAM has a multiplication operation, then by repeatedly squaring we can compute 2^{2^n} in n operations.
  - We argue that we cannot assume such an operation only costs 1 time unit since the operands will have 2^n bits and hence we might be encoding an exponential time computation within such operations.
- In particular, it can be shown that if we do not account properly for the cost of RAM operations, then we can factor integers (and thus be able to break encryption schemes such as RSA) in polynomial time but such a computational algorithm would not be considered realistic.
- One way to deal with such RAM models is to say that any operation on operands of length m requires time m.

Classical vs quantum computation

- Quantum computation takes advantage of principles of quantum mechanics (such as "entangled states") which allow a quantum state involving n 'qubits' to be a "superposition" of up to 2^n possible bit strings.
- In 1994 Peter Shor proved that a quantum computer can factor numbers into primes in polynomial time.
- So if large quantum computers can be built, they could crack the RSA encryption scheme. (There are other ways that RSA could be broken (side channel attacks.)
- **The Catch:** Despite substantial effort, physicists have so far been unable to build an actual quantum computer large enough to process more than a dozen or so bits of information.
- **Caveat:** Quantum cryptography provides a promising approach to secure communication in which security (rather than complexity) depends on principles of quantum mechanics.

Note

Quantum computation does NOT change the Church-Turing thesis, that is, what is computable. But it does seem to change what is computable in polynomial time.

What have we been doing so far in this course

- So far, we have almost entirely been presenting polynomial time algorithms.
- As one exception, we did consider the pseudo polynomial time DP for the knapsack problem.
- Many times we didn't care if the operands (say in interval scheduling) were real numbers or integers.
- We just assumed that we could do basic arithmetic operations and comparisons in one step.
- This was not a problem because we didn't use algorithms that would build up large integers. (Note that if we only use addition, then repeated doubling can only produce a number as large as 2^n in n steps).

And now we take a detour from efficient algorithms

- We have often been refering to NP-hardness when we considered computing optimal solutions to a number of optimization problems.
- We alluded to the widely held belief that such problems cannot be computed efficiently (for all inputs).
- This will be expressed as the conjecture (sometimes called Cook's Hypothesis) that $P \neq NP$.
- Hence we will later embarck on approximation algorithms for such problems.
- We now want to define the meaning of this conjecture and the evidence we have for believing in this conjecture.
  - Briefly stated, the main evidence is the extensive number of problems which can be shown to be "equivalent" in the sense that if any one of them can be computed efficiently (i.e. in P) then they all are.
  - These are problems that have been studied for many years (decades and in certain cases centuries) without anyone being able to find polynomial time algorithms.

What is *NP*?

- The class *NP* (Nondeterministic Polynomial Time)
- *NP* consists of all search problems for which a solution can be efficiently (i.e. in polynomial time) verified.
- More specifically, a set (or language) L is in the class *NP* if there is a polynomial time computable $R(x, y)$ and a polynomial time computable function f such that for all $x \in L$, there is a certificate y such that $|y| \leq f(|x|)$ and $R(x, y)$.

Examples in *NP* (besides everything in *P*)

- Given an integer x (in say binary of decimal representation), is it a composite number? (This is in fact a polynomial time computable decision problem.)
- Given a graph G, can it be vertex colored in 3 colors?
- Given a set $S = \{a_i\}$ of integers can it partitioned into two subsets S_1 and S_2 such that $\sum_{a_i \in S_1} a_i = \sum_{a_i \in S_2} a_i$?

P versus NP

- P: Problems for which solutions can be efficiently found
- NP: Problems for which solutions can be efficiently verified

Conjecture

$$P \neq NP$$

- Most computer scientists believe this conjecture.
- But is seems to be incredibly hard to prove.

Why is proving $P \neq NP$ difficult?

- One reason is that some search problems in NP (such as finding a square root) turn out to easy.
- Another easy example is the maximum bipartite matching problem introduced last week). More generally,

The matching problem for undirected graphs

Given a large group of people, we want to pair them up to work on projects. We know which pairs of people are compatible, and (if possible) we want to put them all in compatible pairs.

- If there are 50 or more people, a brute force approach of trying all possible pairings would take billions of years.
- However in 1965 Jack Edmonds found an ingenious efficient algorithm. So this problem is in P.
- How can we identify the hard NP problems?

NP-Complete Problems

- These are the hardest *NP* problems.
- A problem A is *p*-reducible to a problem B if an "oracle" for B can be used to efficiently solve A.
- If A is *p*-reducible to B, then any efficient procedure for solving B can be turned into an efficient procedure for A.
- If A is *p*-reducible to B and B is in P, then A is in P.

Definition

A problem B is *NP*-complete if B is in *NP* and every problem A in *NP* is *p*-reducible to B.

Theorem

If A is NP-complete and A is in P, then $P = NP$.

To show $P = NP$ you just need to find a fast (polynomial-time) algorithm for one *NP*-complete problem!!!

- A great many (thousands) of problems have been shown to be *NP*-complete.
- Most scheduling related problems (delivery trucks, exams etc) are *NP*-complete.
- The following simple exam scheduling problem is *NP*-complete:

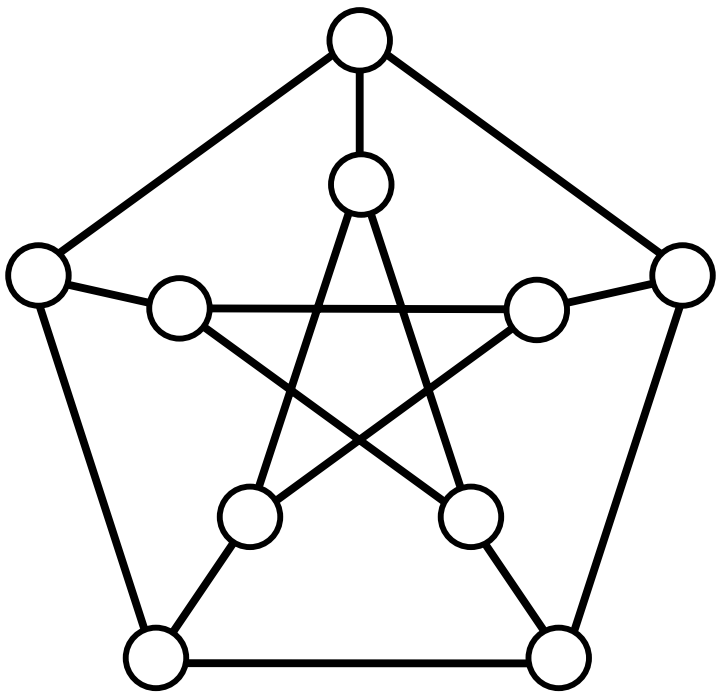
Example

- We need to schedule N examinations, and only three time slots are available.
- We are given a list of exam conflicts: A conflict is a pair of exams that cannot be offered at the same time, because some student needs to take both of them.
- **Problem:** Is there a way of assigning each exam to one of the time slots T_1, T_2, T_3, so that no two conflicting exams are assigned to the same time slot.
- This problem is also known as graph 3-colourability.

Graph 3-Colourability

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.



Graph 3-Colourability

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.