Applied Algorithm Design
Lecture 1

Pietro Michiardi

Eurecom

Introduction

As an opening topic we will look at an algorithmic problem that nicely illustrates many of the themes we will address in this course.
This problem is motivated by some very practical concerns and our goal in the reminder of this lecture will be to:

- Define clearly the problem
- Find an algorithm to solve the problem
- Prove that the algorithm is correct
- Study the amount of time it takes to come up with a solution

The Stable Matching problem: historical note

The stable matching problem originated from two mathematical economists, D. Gale and L. Shapley, that asked the following question:

Job hunting:

Could one design a job recruiting process or college admission process that is *self-enforcing*?

The Job recruiting process

Informally, the essence of the application process is the interplay between *two* parties: the **company** and the **the applicants**.

Each applicant has a preference ordering on companies, and each company, once the applications come in, forms a preference ordering on its applicants.

Based on these preferences, companies extend offers to some applicants and applicants chose which offer to accept.

What could go wrong?

The applicants could “misbheave”:

What could go wrong?

The applicants could “misbheave”:

- Alice accepts a summer job at BayTelecom

What could go wrong?

The applicants could “misbheave”:

- Alice accepts a summer job at BayTelecom
- Few days later a small company, WebJoy, offers Alice a better deal and Alice turns down BayTelecom

What could go wrong?

The applicants could “misbheave”:

- Alice accepts a summer job at BayTelecom
- Few days later a small company, WebJoy, offers Alice a better deal and Alice turns down BayTelecom
- BayTelecom then contacts Bob, who was on a waiting list, and offers him a job

What could go wrong?

The applicants could “misbheave”:

- Alice accepts a summer job at BayTelecom
- Few days later a small company, WebJoy, offers Alice a better deal and Alice turns down BayTelecom
- BayTelecom then contacts Bob, who was on a waiting list, and offers him a job
- Bob promptly reacts, turns down the offer from BubbleSoft and accept the offer from BayTelecom

What could go wrong?

The applicants could “misbheave”:

- Alice accepts a summer job at BayTelecom
- Few days later a small company, WebJoy, offers Alice a better deal and Alice turns down BayTelecom
- BayTelecom then contacts Bob, who was on a waiting list, and offers him a job
- Bob promptly reacts, turns down the offer from BubbleSoft and accept the offer from BayTelecom

And the situation begins to spiral out of control...

What could go wrong?

The companies could “misbheave”:

What could go wrong?

The companies could “misbheave”:

- Suppose Chelsea, destined to BubbleSoft, hears about Alice being employed at WebJoy

What could go wrong?

The companies could “misbheave”:

- Suppose Chelsea, destined to BubbleSoft, hears about Alice being employed at WebJoy
- Nothing prevents Chelsea to call up WebJoy and send her application

What could go wrong?

The companies could “misbheave”:

- Suppose Chelsea, destined to BubbleSoft, hears about Alice being employed at WebJoy
- Nothing prevents Chelsea to call up WebJoy and send her application
- WebJoy, on looking at Chelsea application just realize they would have preferred to have her for the summer

What could go wrong?

The companies could “misbheave”:

- Suppose Chelsea, destined to BubbleSoft, hears about Alice being employed at WebJoy
- Nothing prevents Chelsea to call up WebJoy and send her application
- WebJoy, on looking at Chelsea application just realize they would have preferred to have her for the summer
- ... and thus turn down Alice, that now would have no summer job at all (she lost her initial one at BayTelecom)

What could go wrong?

The companies could “misbheave”:

- Suppose Chelsea, destined to BubbleSoft, hears about Alice being employed at WebJoy
- Nothing prevents Chelsea to call up WebJoy and send her application
- WebJoy, on looking at Chelsea application just realize they would have preferred to have her for the summer
- ... and thus turn down Alice, that now would have no summer job at all (she lost her initial one at BayTelecom)

Situations like those rapidly generate chaos: if people are allowed to act in their self-interest, then everything could break down!

What could go wrong?

Question:

Is there a way to formulate the problem such that self-interest itself prevents offers from being retracted and redirected?

This is the question Gale and Shapley really asked

Question:

Given a set of preferences among employers and applicants, can we assign applicants to employers so that for every employer E, and every applicant A who is not scheduled to work for E, at least on of the following two things is the case?

i) E prefers every one of its accepted applicants to A; or

ii) A prefers her current situation over working for employer E

If this holds, the outcome is stable: individual self-interest will prevent any applicant/employer deal from being made behind the scenes.

Formulating the problem

In this course, we're not only interested in taking a formal definition of a problem from a text book and trying to find an algorithmic solution to it.

*What is really important here is: how can we **clearly** formulate a problem?*

Difficulties

The world of companies and applicants contains some *distracting asymmetries*.

- Each applicant is looking for a single company, but each company is looking for many applicants
- There may be more applicants than there are available slots for summer jobs
- Each applicant does not typically apply to every company

Question:

Can we eliminate these complications and arrive at a more "bare-bones" version of the problem?
Can we do so while *preserving* the fundamental issues of the initial problem, so that any extensions apply to the general case as well?

Assumptions

We assume here that each of n applicants applies to each of n companies, and each company wants to accept a *single* applicant.

Stable marriage problem

The problem can be viewed as the problem of devising a system by which each of n men and n women can end up getting married: our problem has a natural analogue of two "genders" - the applicants and the companies - and in the case we are considering, everyone is seeking to be paired with exactly one individual of the opposite gender.

Definitions: matchings

Consider a set $M = \{m_1, ..., m_n\}$ of n men, and a set $W = \{w_1, ..., w_n\}$ of n women. Let $M \times W$ denote the set of all possible ordered pairs of the form (m, w), where $m \in M$ and $w \in W$.

Matching

A *matching* S is a set of ordered pairs, each from $M \times W$, with the property that each member of M and each member of W appears in at most one pair in S.

Perfect Matching

A *perfect matching* S' is a matching with the property that each member of M and each member of W appears in *exactly* one pair in S'.

Definitions: preference lists

We now formally define the notion of *preferences* to the Stable Marriage problem setting. Each man $m \in M$ ranks all the women; we say that m prefers w to w' if m ranks w higher than w'. We will refer to the ordered ranking of m as his *preference list*. Analogously, each woman ranks all the men.

Important: we do not allow ties in the ranking[1].

[1] What would happen in this case?

Definition: stability and instability

Given a *perfect matching* S, what can go wrong?
There are two pairs (m, w) and (m', w') in S with the property that m prefers w' to w, and w' prefers m to m'. There's nothing to prevent m and w' to get together and abandon their current partners.

Unstable matching

We say that (m, w') is an *instability* with respect to S: (m, w') does not belong to S, but each of m and w' prefers the other to their partners in S.

Stable matching

We say that a matching S is *stable* if: i) it is *perfect*, and ii) there is no instability with respect to S.

Example 1: agreement

Suppose we have a set of two men $\{m, m'\}$ and a set of two women $\{w, w'\}$. The preference lists are as follows:

- m prefers w to w'
- m' prefers w to w'
- w prefers m to m'
- w' prefers m to m'

Questions:

Is there any *perfect* matching?

Is there any *stable* matching?

Example 2: clash

Suppose again we have a set of two men $\{m, m'\}$ and a set of two women $\{w, w'\}$. The preference lists are as follows:

- m prefers w to w'
- m' prefers w' to w
- w prefers m' to m
- w' prefers m to m'

Questions:

Is there any *perfect* matching?

Is there any *stable* matching?

Examples: solutions

Agreement example:

- (m, w) and $(m', w') \rightarrow$ Perfect Matching and Stable
- (m', w) and $(m, w') \rightarrow$ Perfect Matching

Clash example:

- (m, w) and $(m', w') \rightarrow$ Perfect Matching and Stable
- (m', w) and $(m, w') \rightarrow$ Perfect Matching and Stable

Key questions:

With all these definitions at hand, after discussing the previous two examples here are the key questions we will address now:

Questions:

- Does there exist a *stable matching* for every set of preference list?
- Given a set of preference lists, can we efficiently construct a stable matching if there is one?

The algorithm (part 1)

We now show that there exists a stable matching for every set of preference lists among the men and women and we will give an efficient algorithm that takes the preference lists and constructs a stable matching.

- Initially, everyone is unmarried.

The algorithm (part 1)

We now show that there exists a stable matching for every set of preference lists among the men and women and we will give an efficient algorithm that takes the preference lists and constructs a stable matching.

- Initially, everyone is unmarried.
- An unmarried man *m* choses the woman *w* who ranks highest on his preference list and *proposes* to her

The algorithm (part 1)

We now show that there exists a stable matching for every set of preference lists among the men and women and we will give an efficient algorithm that takes the preference lists and constructs a stable matching.

- Initially, everyone is unmarried.
- An unmarried man m choses the woman w who ranks highest on his preference list and *proposes* to her

$\rightarrow$ Can we determine immediately that (m, w) will be one of the pairs in our final stable matching?

The algorithm (part 2)

- Suppose we are now in a state in which there are some free man and women, and some of them are engaged. An arbitrary free man *m* proposes to the highest-ranked woman *w* to whom he has **not yet** proposed.

The algorithm (part 2)

- Suppose we are now in a state in which there are some free man and women, and some of them are engaged. An arbitrary free man m proposes to the highest-ranked woman w to whom he has **not yet** proposed.
- If w is also free, then m and w become engaged. Otherwise w is already engaged to some other man m'. In this case she determines which of m or m' ranks higher on her preference list; this man becomes engaged to w and the other becomes free

The algorithm (part 2)

- Suppose we are now in a state in which there are some free man and women, and some of them are engaged. An arbitrary free man m proposes to the highest-ranked woman w to whom he has **not yet** proposed.
- If w is also free, then m and w become engaged. Otherwise w is already engaged to some other man m'. In this case she determines which of m or m' ranks higher on her preference list; this man becomes engaged to w and the other becomes free
- The algorithm terminates when no one is free: the resulting perfect matching is returned

Some useful observations

First we consider the view of a woman *w* during the execution of the algorithm. For a while, no one has proposed to her, and she is free. Then a man *m* may propose to her, and she becomes engaged. As time goes on, she may receive additional proposals, accepting those that increase the rank of her partner. Hence we have:

Observation

w remains engaged from the point at which she receives her first proposal, and the sequence of partners to which she is engaged gets better and better (in terms of her preference list)

Some useful observations

The view of a man during the execution of the algorithm is different. He is free until he proposes to the highest-ranked woman on his list; at this point he may or may not become engaged. As time goes on, he may alternate being free and being engaged. Hence we have:

Observation

The sequence of women to whom m proposes gets worse and worse (in terms of his preference list)

A bound on the execution time

Lemma

The G-S algorithm terminates after at most n^2 iterations.

Proof.

Each iteration consists of some man proposing, **for the only time**, to a woman he has never proposed before. Let $P(t)$ denote the set of pairs (m, w) such that m has proposed to w by the end of iteration t. We see that

$$|P(t+1)| > |P(t)| \forall t$$

Since there are only n^2 possible pairs of men and woman in total, $|P(t)|$ can increase at most n^2 times over the course of the algorithm. It follows that there can be at most n^2 iterations. □

What did we do?

A useful strategy for *upper-bounding* the running time of an algorithm, is to find a measure of *progress*. Not all the quantities to measure the progress of the algorithm would have worked. For example, the number of free individuals could remain constant from one iteration to the next, as could the number of engaged pairs. These quantities need not to strictly increase in each iteration.

Now, is it obvious that the set S returned at the end of the algorithm is a perfect matching?
Essentially, we have to show that no man can "fall off" the end of its preference list; the only way for the algorithm to terminate is for there to be no free man. In this case, the set of engaged couples would indeed be a perfect matching.

Understanding better the algorithm

Lemma

If m is free at some time in the execution of the algorithm, then there is a woman to whom he has not yet proposed.

Proof.

By contradiction. Suppose there comes a point when m is free but has already proposed to every woman. Then, by Obs. 1, each of the n women is engaged at this point in time. Since the set of engaged pairs forms a matching, there must also be n engaged men. But there are only n men in total and m is not engaged, so this is a contradiction. □

Is S a perfect matching?

Lemma

The set S returned at termination is a perfect matching

Proof.

By contradiction. The set of engaged pairs always forms a matching. Let us suppose that the algorithm terminates with a free man m. At termination, it must be the case that m already proposed to every woman, for otherwise the algorithm could not have terminated. But this contradicts Lemma 5, which says there cannot be a free man who has never proposed to every woman. □

Is S a stable matching?

Lemma

Consider an execution of the G-S algorithm that returns a set of pairs S. The set S is a stable matching.

Proof.

Assume there is an instability with respect to S. By definition, an instability occur when two pairs, (m, w) and (m', w') in S have the preference list: m prefers w' to w and w' prefers m to m'. In the execution that produced S, m's last proposal was to w, by definition. Did m propose to w' at some earlier point in time? If he didn't, then w must occur higher on m's preference list than w', contradicting our assumption that m prefers w' to w. If he did, then he was rejected by w' in favor of m''. Since m' is the final partner of w', either $m'' = m'$ or, by Obs. 1 w' prefers m' to m'': either way, this contradicts our initial assumption.
It follows that S is a stable matching. □

Is that all?

The story so far...

- We began by defining the notion of a stable matching
- We have just proven that the G-S algorithm actually constructs one

Let's now consider some further questions about the behavior of the G-S algorithm. One of the reasons to do that is: all these Lemmas and Proofs look nice, but what if we really implement the G-S algorithm?

Illustrative example

- Recall the “clash” example of before: we have seen that there could be multiple stable matchings
- If we execute the G-S algorithm in the “clash” case we would end up with the couples (m, w) and (m', w')
- The *other* stable matching (m', w) and (m, w') cannot be attained, when men propose first.
- However, if we let women propose first, the other matching is attained.

This is unfair!!

The previous example shows that in the G-S algorithm men are favored: there's a certain "unfairness"!

Explanation:

- If men's preferences mesh perfectly (they prefer different women) then in all run of the G-S they all end up matched with their first choice, **independent** of the preferences of the women.
- If women's preferences clash completely with the men's, then the resulting stable matching is as bad as possible for the woman.

To summarize: someone is destined to be unhappy. Women are unhappy if men propose first and men are unhappy if women propose first.

Watch the spec!

The G-S appears to be *under-specified*: as long as there is a free men we are allowed to chose *any* free man to make the next proposal.

Question:

Do all executions of the G-S algorithm yield the same matching?

This is a *very important* question in many settings of computer science: we have an algorithm that runs **asynchronously**, with different independent components performing actions that can be interleaved in complex ways and we want to know how much variability this asynchrony causes in the final outcome[2].

[2] Hey, what about randomized settings?

Answer:

All executions of the G-S algorithm yield the same matching

All executions of the G-S algorithm yield the same matching

- How can we prove such a statement?
- There are several ways to do that, but not all of them are easy
- One way to proceed is to *uniquely characterize* the matching obtained by one execution
- ... and then show somehow that all executions result in the matching with this same characterization

All executions of the G-S algorithm yield the same matching

What is this characterization? Let's start with men.

Definition: valid partner

A woman w is a valid partner of a man m if there is *a* stable matching that contains the pair (m, w)

Definition: best valid partner

A woman w is the best valid partner of m if w is a valid partner of m and no woman whom m ranks higher than w is a valid partner of his. We will use the notation $best(m)$ to denote the best valid partner of m.

All executions of the G-S algorithm yield the same matching

Let S^* denote the set of pairs $\{(m, best(m)) : m \in M\}$

Lemma

Every execution of the G-S algorithm results in the set S^*

Before delving into the proof: this statement is quite surprising! First of all, there is no reason to believe S^* to be a matching at all, nor a stable one! After all, could it happen that two men have the same best valid partner? Secondly, this lemma answers the question we discussed before: the order of proposals in the G-S algorithm has absolutely no effect on the final outcome.

Proving the Lemma - part 1 of 3

- Assume the following: some execution $\mathcal{E}$ results in a matching S in which a man is paired with a woman who is not his best valid partner.
- Since men propose in decreasing order of preference, it means that some man is rejected by a valid partner during the execution $\mathcal{E}$. Consider the first moment when this happens: m is rejected by a valid partner w. Then w must be m's best valid partner $best(m)$.

Proving the Lemma - part 2 of 3

- The rejection of m by w could have happened: i) either because m proposed and was turned down in favor of w's existing partner; or ii) because w broke her engagement to m in favor of a better proposal
- In any case, w forms or continue an engagement with a man m' whom she prefers to m
- Since w is a valid partner of m there exists a stable matching S' containing the pair (m, w). Now we ask: who is m' paired with in this matching? Suppose a woman $w' \neq w$.

Proving the Lemma - part 3 of 3

- Since the rejection of m by w was the first rejection of a man by a valid partner in $\mathcal{E}$, it must be that m' had not been rejected by any valid partner at the point in $\mathcal{E}$ when he became engaged to w. Since m' proposed in decreasing order of preference, and since w' is clearly a valid partner of m', it must be that m' prefers w to w'.
- Hey! But we have already seen that w prefers m' to m, for in the execution $\mathcal{E}$ she rejected m in favor of m'. Since $(m', w) \notin S'$, it follows that (m', w) is an instability in S'.
- But this contradicts our claim that S' is stable and hence contradicts our initial assumption.

So for the men, the G-S algorithm is ideal: they always get their best partner!

Unfortunately, this is not the case for the women.

Homework 1:

Using similar arguments as for the men, prove the following Lemma:

In the stable matching S^*, each woman is paired with her worst valid partner.

Summarizing

With the results discussed before we have the following very general phenomenon :

Who proposes, men or women?

For any input, the side that does the proposing in the G-S algorithm ends up with the best possible stable matching (from their perspective), while the side that does not do the proposing correspondingly ends up with the worst possible stable matching.

What did you learn today about Algorithm Design?

The Stable Matching Problem provides a rich example of the process of algorithm design. For many problems, this process involves few significant steps:

- Formulating the problem with enough mathematical precision so that we can ask a concrete question and start thinking about algorithms to solve it;
- Designing an algorithm for the problem;
- Analyzing the algorithm by proving it is correct and giving a bound on the running time so as to establish the algorithm *efficiency*

This strategy is carried out in practice by re-conducting every new problem to a few fundamental design techniques.

Yeah, right ... but where are the applications?

Well, besides the fact that the Stable Matching Problem has many applications in hiring processes, in admissions of students to schools and universities, in admissions of people to hospitals, not to mention financial applications ...

Let's have an overview of five representative problems we can link to what we learned.

Five representative examples

- Interval Scheduling
- Weighted Interval Scheduling
- Bipartite matching
- Independent Set
- Competitive Facility Location

But first... a brief background on graph theory

Don't worry, next lecture will be devoted partly to a fairly detailed introduction to graphs.

- A graph G is a way of encoding pairwise relationships among a set of objects
- $G = (V, E)$ consists of a pair of sets: $\{V\}$, the set of nodes, and $\{E\}$ the set of edges
- *edges* connect two nodes, that is: $e \in E$, $e = \{u, v\}$ for some $u, v \in V$.
- u and v are also called the *ends* of the edge e

Interval Scheduling

Consider the following very simple scheduling problem.

- You have a resource - a slot on a computational grid, a lecture room or the uplink capacity of your peer - and many people request to use it for periods of time
- A *request* takes the form: Can I reserve the resource starting at time s, until time f?
- Assume the resource can be used by at most one person at a time
- A scheduler wants to accept a subset of all these requests, rejecting all others, so that the accepted requests do not overlap in time

The goal is to maximize the number of accepted requests.

Duhhh... what did we learn so far? Formalize the problem...

Interval Scheduling - formal definition

- We have n requests labelled $1, \cdots n$
- Each request i has a start time s_i and a finish time f_i
- We have $s_i < f_i \, \forall i$

Definition

Two requests i and j are *compatible* if $f_i \leq s_j$ or $f_j \leq s_i$.

In general, a subset A of requests is compatible if all pairs $i, j \in A, i \neq j$ are compatible.

The goal is to select a compatible subset of requests of maximum possible size.

Interval Scheduling - example

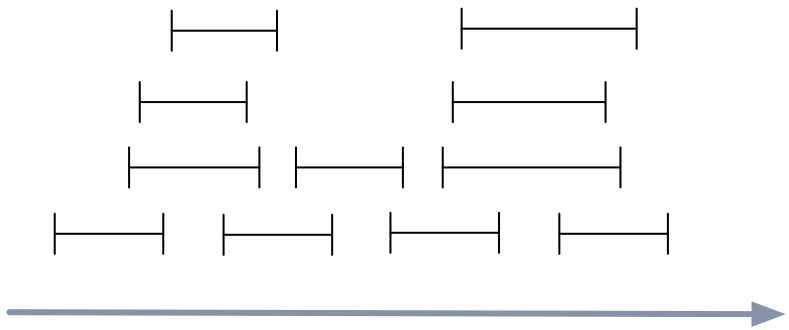

Figure: An instance of the Interval Scheduling Problem

Interval Scheduling

We will see in Lecture 3 that this (and similar) problem can be solved with a very simple approach that orders the set of requests according to a certain heuristic and then "greedily" processes them in one pass, selecting as large a compatible subset as it can.

Greedy algorithms

A set of myopic rules that process the input in one piece at a time with no apparent look-ahead.

When a greedy algorithm can be shown to find an optimal solution for all instances of a problem, it's often fairly surprising! This allows learning a lot about the structure of the underlying problem. In other cases, the solution provided by these algorithm will be not optimal, and the goal will be to study how close to optimal it is.

Weighted Interval Scheduling

In the problem discussed before we sought to maximize the *number* of requests that could be accommodated simultaneously.

Now, suppose more generally that each request interval i has an associated *value*, or *weight* $v_i > 0$; we can imagine this as the amount of money we will make from the i^{th} individual if we schedule his or her request.

The goal here is to find a compatible subset of intervals of maximum total value.

Weighted Interval Scheduling

Obviously, if we assume $v_i = 1 \, \forall i$, we're back to the Interval Scheduling problem.

However, assume there is a very rich person who values its interval a lot, e.g. $v_1 > \sum_j v_j \, \forall j$.
In this case the algorithm that solves the problem would have to understand the situation and schedule only one interval!

There appear to be no simple greedy rule that walks through the intervals one at time, making the correct decision in presence of arbitrary values.

Bipartite Matching

We just saw the Stable Matching Problem: similar concepts can be expressed more generally in terms of graphs.

Definition: Bipartite Graph

We say that a graph $G = (V, E)$ is *bipartite* if its node set V can be partitioned into sets X and Y such that every edge $e \in E$ has one end in X and the other end in Y

Definition: Matching in a Graph

A matching in a graph $G = (V, E)$ is a set of edges $M \subseteq E$ with the property that each node appears in at most one edge of M.

M is a *perfect matching* if every node appears in exactly one edge of M.

Bipartite Matching: where's the problem?

In this case we do not consider preferences: there is however a different source of complexity:

- There is not necessarily an edge from every $x \in X$ to every $y \in Y$
- This implies that the set of possible matchings has quite a complicated structure
- Take a look at the following example: there are many matchings, but there's only one that is perfect. Which one?

Bipartite Matching - example

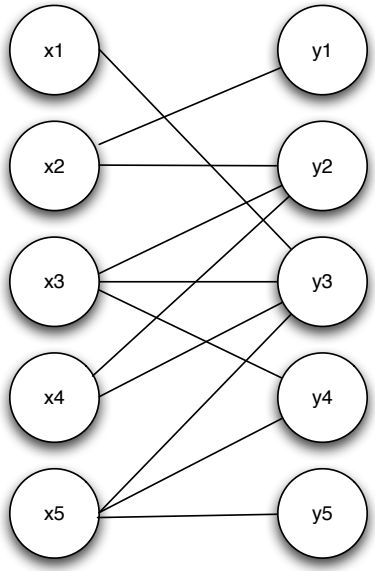

Figure: An instance of the Bipartite Matching Problem

Bipartite Matching Problem

The Bipartite Matching Problem

Given an arbitrary bipartite graph G, the problem is to find a matching of the maximum size.

If $|X| = |Y| = n$, then there is a perfect matching if and only if the maximum matching has size n.

We will see, in Lecutre 4, that the algorithm we studied today does not seem to fit the problem. There is, however, a very elegant and efficient algorithm to find a maximum matching.
It inductively builds up larger and larger matchings, selectively backtracking along the way. This process is called an *augmentation* and it forms the central component in a large class of efficiently solvable problems called **network flow problems**.

The Independent Set Problem

Now let's talk about an extremely general problem, which includes most of the earlier problems we saw as special cases.

Definition: Independent Set

Given a graph $G = (V, E)$ a set of nodes $S \subseteq V$ is *independent* if no two nodes in S are joined by an edge.

The Independent Set Problem

Given a graph $G = (V, E)$, find an independent set that is as large as possible.

Independent Set - example

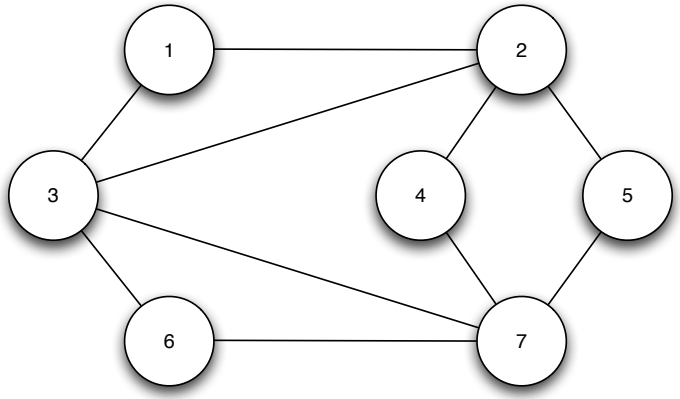

Figure: A graph whose largest independent set has size 4

The Independent Set Problem: why is it so important?

The Independent Set Problem *encodes* any situation in which you are trying to choose from among a collection of objects and there are pairwise *conflicts* among some of the objects.

Informal example:

Say you have n friends and some pairs of them don't get along. How large a group of your friends can you invite to dinner if you don't want any interpersonal tension? This is simply the largest independent set in the graph whose nodes are your friends, with an edge between each conflicting pair.

The Independent Set Problem: encoding previous problems

Interval Scheduling Problem

Define a graph $G = (V, E)$ in which nodes are the intervals and there is an edge between each pair of them that overlap; the independent sets in G are then just the compatible subsets of intervals, and you want to find the largest.

Bipartite Matching Problem

Given a bipartite graph $G' = (V', E')$ the objects being chosen are edges and the conflicts arise between two edges that share an end. Define a graph $G = (V, E)$ in which the node set V is equal to the edge set E' of G'. Define an edge between each pair of elements in V that correspond to edges of G' with a common end. The independent sets of G are precisely the matchings of G'.

Competitive Facility Location

If the previous problem was very difficult to solve efficiently, wait for this one!
This is a typical two-player game.

The Competitive Facility Location Problem

Consider two companies, C_1 and C_2, competing for market shares in a geographic area, which is broken into n zones: $1, 2, \cdots, n$. Each zone i has a value b_i, corresponding to the revenue of the companies if they opened a franchise there.

Certain pair of zones (i, j) are *adjacent* and local zoning laws prevent two adjacent zones from each containing a franchise, regardless of which company owns them.

Competitive Facility Location - continued

The Competitive Facility Location Problem - continued

We model these conflicts with a graph $G = (V, E)$ where V is the set of zones and (i, j) is an edge in E if zone i and j are adjacent. The zoning requirement says that the full set of franchises opened must form an independent set in G.

The game consists of two players, company C_1 and C_2 alternately selecting nodes in G, with C_1 moving first. At all times, the set of selected nodes must form an independent set in G. Suppose C_2 has a target bound B: is there a strategy for C_2 so that no matter how C_1 plays, C_2 will be able to select a set of nodes with a total value of at least B?

Homework 2:

The Gale-Shapley Algorithm

Implement[a] the Gale-Shapley algorithm in the python.

[a] Let's discuss what does "implement" mean...