# CSE 331 Final Exam Preparation

This is in no way a substitute for exam preperation, mearly a compilation of all the key talking points.

Chandra Neppalli

May 4, 2021

# Counter Example

Best proof to use to *disprove* universally true propositions.

# Counter Example

Best proof to use to *disprove* universally true propositions.

Ex: Every day is a Wednesday, where a counter example would be Monday is not Wednesday.

# Contradiction

Best proof to use if you want to assert something is true.

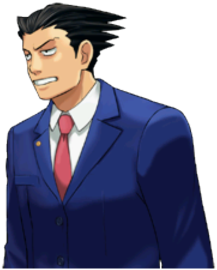

Figure: Sourced from Google

# Contradiction

Best proof to use if you want to assert something is true.

Assume what you want to prove is false, then show this leads to a contradiction.

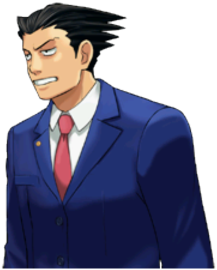

Figure: Sourced from Google

# Contradiction

Best proof to use if you want to assert something is true.

Assume what you want to prove is false, then show this leads to a contradiction.

Therefore, the original assumption has to be true.

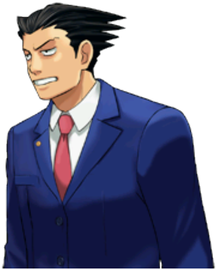

Figure: Sourced from Google

# Contraposition

Best proof for proving causality. Define two propositions E and F.

# Contraposition

Best proof for proving causality. Define two propositions E and F.

If you want to prove that $E \rightarrow F$, it might be more doable to prove $\neg F \rightarrow \neg E$, as they are both logically equivalent.

# Contraposition

Best proof for proving causality. Define two propositions E and F.

If you want to prove that $E \rightarrow F$, it might be more doable to prove $\neg F \rightarrow \neg E$, as they are both logically equivalent.

This is especially useful if the **scope** of F is smaller than the scope of E.

# Direct Proof

If the proof is *simple*, consider directly proving it.

# Direct Proof

If the proof is *simple*, consider directly proving it.

Remember though, that you must maintain $W.L.O.G$, that your proof can never be too specific and must be arbitrary.

# Proof by Induction

Proof by Induction is a really nice proof technique when you reduce your proof to a known correct base case.

# Proof by Induction

Proof by Induction is a really nice proof technique when you reduce your proof to a known correct base case.

If proof needs to be correct for all numbers $\in \mathbb{N}$, and each step is dependant on the previous step, then *every* step can be reduced to a definitive base case that is easy to directly prove.

# Extra: Progress Measure

This is useful for proving an algorithm with a loop terminates.

# Extra: Progress Measure

This is useful for proving an algorithm with a loop terminates.

Let $P(i)$ denote an integer such that:

# Extra: Progress Measure

This is useful for proving an algorithm with a loop terminates.

Let $P(i)$ denote an integer such that:

- $P(0) = I$

# Extra: Progress Measure

This is useful for proving an algorithm with a loop terminates.

Let $P(i)$ denote an integer such that:

- $P(0) = I$
- $P(i)$ is an accumulator. This means that $P(i+1) > P(i)$

# Extra: Progress Measure

This is useful for proving an algorithm with a loop terminates.

Let $P(i)$ denote an integer such that:

- $P(0) = l$
- $P(i)$ is an accumulator. This means that $P(i+1) > P(i)$
- $\forall i, \quad P(i) \leq k$

# Extra: Progress Measure

This is useful for proving an algorithm with a loop terminates.

Let $P(i)$ denote an integer such that:

- $P(0) = l$
- $P(i)$ is an accumulator. This means that $P(i+1) > P(i)$
- $\forall i, \quad P(i) \leq k$

From these 3 properties, the number of iterations is bounded by $k - l + 1$

# Extra: Progress Measure

This is useful for proving an algorithm with a loop terminates.

Let $P(i)$ denote an integer such that:

- $P(0) = l$
- $P(i)$ is an accumulator. This means that $P(i+1) > P(i)$
- $\forall i, \quad P(i) \leq k$

From these 3 properties, the number of iterations is bounded by $k - l + 1$

Note: This isn't a runtime analysis, rather a proof that the algorithm terminates.

# Greedy Stays Ahead

This technique is used to prove that a greedy algorithm returns an optimal solution.

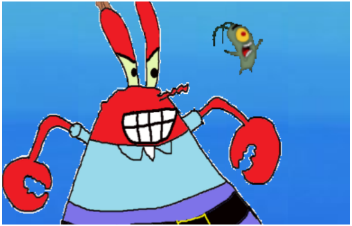

Figure: Sourced from New Grounds

# Greedy Stays Ahead

This technique is used to prove that a greedy algorithm returns an optimal solution.

At every step of a greedy algorithm, it will stay *at least* as far as the optimal solution at that step.

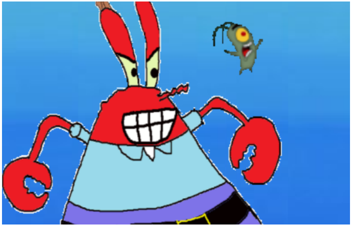

Figure: Sourced from New Grounds

# Greedy Stays Ahead

This technique is used to prove that a greedy algorithm returns an optimal solution.

At every step of a greedy algorithm, it will stay *at least* as far as the optimal solution at that step.

HW4 "Attack on Alarms" and Interval Scheduling are examples of problems with greedy solutions.

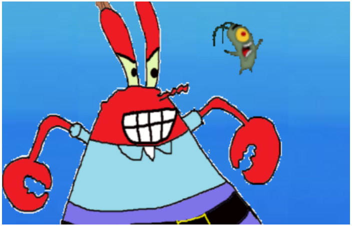

Figure: Sourced from New Grounds

# Introduction

Let's say there are two groups: Group A and Group B.

# Introduction

Let's say there are two groups: Group A and Group B.

How do we generate a **stable** matching between each member of the two groups efficiently?

# Introduction

Let's say there are two groups: Group A and Group B.

How do we generate a **stable** matching between each member of the two groups efficiently?

Moreover, what is a **stable** matching?

# Perfect Matchings

A **perfect matching** is a bijective matching between A and B.

# Perfect Matchings

A **perfect matching** is a bijective matching between A and B.

**Every member** in group A is matched with exactly one member in group B.

# Perfect Matchings

A **perfect matching** is a bijective matching between A and B.

**Every member** in group A is matched with exactly one member in group B.

Conversely, every member in group B is matched with **exactly** one member in group A.

# Perfect Matchings

A **perfect matching** is a bijective matching between A and B.

**Every member** in group A is matched with exactly one member in group B.

Conversely, every member in group B is matched with **exactly** one member in group A.

With $n$ members in each group, there are $n!$ perfect matchings.

# Instability

For a particular matching, define a member $m$ from group A and $n$ from group B such that $(m, n)$ is not in the matching.

# Instability

For a particular matching, define a member $m$ from group A and $n$ from group B such that $(m, n)$ is not in the matching.

If $m$ prefers $n$ over their current matching **and** $n$ prefers $m$ over their current matching, then $(m, n)$ is an instability to the perfect matching.

# Stable Matching

A stable matching **is** a perfect matching with no instabilities.

# Stable Matching

A stable matching **is** a perfect matching with no instabilities.

It therefore follows that the number of stable matchings is *at most* the number of perfect matchings, or $n!$

# Stable Matching

A stable matching **is** a perfect matching with no instabilities.

It therefore follows that the number of stable matchings is *at most* the number of perfect matchings, or $n!$

The Gale Shapely Algorithm is an $O(n^3)$ time algorithm that can output a stable matching.

# Stable Matching

A stable matching **is** a perfect matching with no instabilities.

It therefore follows that the number of stable matchings is *at most* the number of perfect matchings, or $n$!

The Gale Shapely Algorithm is an $O(n^3)$ time algorithm that can output a stable matching.

With the right data structures, the runtime can be reduced to $O(n^2)$.

# Stable Matching

A stable matching **is** a perfect matching with no instabilities.

It therefore follows that the number of stable matchings is *at most* the number of perfect matchings, or $n!$

The Gale Shapely Algorithm is an $O(n^3)$ time algorithm that can output a stable matching.

With the right data structures, the runtime can be reduced to $O(n^2)$.

Even though the runtime isn't linear, because the input size is $2n^2 \to \Theta(n^2)$ [1], the runtime **with respect** to the input size is $O(N)$, or linear time.

[1] This comes from $n$ Group A members and $n$ Group B members with their $2n$ preference lists

# Stable Matching

Code:

---

```
Initially all m ∈ M and w ∈ W are free
While there is a man m who is free and hasn't proposed to every
woman
      Choose such a man m
      Let w be the highest-ranked woman in m's
            preference list to whom m has not yet proposed
      If w is free then
            (m, w) become engaged
      Else w is currently engaged to m'
            If w prefers m' to m then
                  m remains free
            Else w prefers m to m'
                  (m, w) become engaged
                  m' becomes free
            Endif
      Endif
Endwhile
Return the set S of engaged pairs.
```

# What are graphs?

Graphs are a way to represent relations between data points.

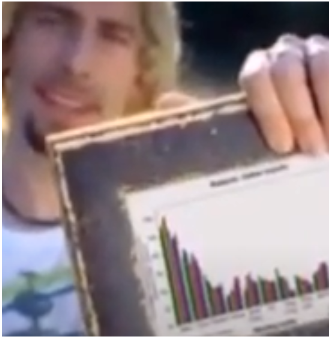

Figure: Sourced from Youtube

# What are graphs?

Graphs are a way to represent relations between data points.

Graphs have a set of vertices which represent data points and a set of edges to model the **relation** between vertices.

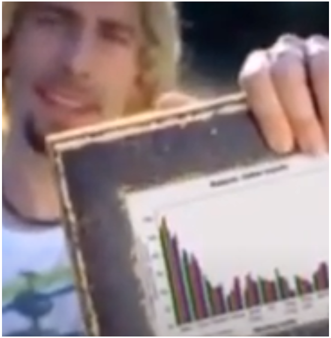

Figure: Sourced from Youtube

# What are graphs?

Graphs are a way to represent relations between data points.

Graphs have a set of vertices which represent data points and a set of edges to model the relation between vertices.

An edge exists between two or more vertices *iff* there is a connection between them.

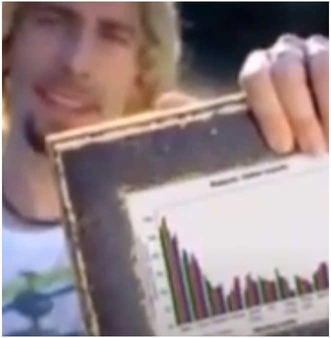

Figure: Sourced from Youtube

# Graph Representation

Adjacency List: Keep a $n$ length array. Vertices are indices of the array where each element of the array contains a pointer to a list of adjacent vertices to the index/vertex. This takes up $\Theta(n+m)$ space for $m$ edges.

Adjacency Matrix: Keep a $n$ $X$ $n$ matrix. Rows and columns are vertices. An edge $(u, v)$ exists if the matrix at row $u$ and column $v$ is not null This takes up $\Theta(n^2)$ space.

# Undirected vs. Directed Graphs

A graph is undirected if edges go both ways. If for any vertices $u$, $v$, vertex $u$ connects to $v$, $v$ connects to $u$.

# Undirected vs. Directed Graphs

A graph is undirected if edges go both ways. If for any vertices $u$, $v$, vertex $u$ connects to $v$, $v$ connects to $u$.

A graph is directed if there is an edge $(u, v)$ such that $u$ is connected to $v$, but $v$ is not connected to $u$.

# Undirected vs. Directed Graphs

A graph is undirected if edges go both ways. If for any vertices $u$, $v$, vertex $u$ connects to $v$, $v$ connects to $u$.

A graph is directed if there is an edge $(u, v)$ such that $u$ is connected to $v$, but $v$ is not connected to $u$.

A graph is a tree if it is connected and there are **no** cycles present.

# Undirected vs. Directed Graphs

A graph is undirected if edges go both ways. If for any vertices $u$, $v$, vertex $u$ connects to $v$, $v$ connects to $u$.

A graph is directed if there is an edge $(u, v)$ such that $u$ is connected to $v$, but $v$ is not connected to $u$.

A graph is a tree if it is connected and there are **no** cycles present.

Bus transportation routes could be classified as an undirected graph, while airports and flights could be modeled as a directed graph.

# Paths and Cycles

A path is a sequence of vertices and edges.

# Paths and Cycles

A path is a sequence of vertices and edges.

A cycle is a sequence of vertices and edges where the first and last vertex are **the same**.

# Paths and Cycles

A path is a sequence of vertices and edges.

A cycle is a sequence of vertices and edges where the first and last vertex are **the same**.

A **simple path** is a path that contains no cycles. A **simple cycle** is a cycle in which the only repeating vertices are the first and last.

# Paths and Cycles

A path is a sequence of vertices and edges.

A cycle is a sequence of vertices and edges where the first and last vertex are **the same**.

A **simple path** is a path that contains no cycles. A **simple cycle** is a cycle in which the only repeating vertices are the first and last.

A cycle needs **at least** 4 elements. The **maximum length** of a simple path for a graph with $n$ vertices is $n - 1$ [1].

[1] Pigeonhole Principle

# Connectivity

A graph is connected if there is a path between any two vertices.

# Connectivity

A graph is connected if there is a path between any two vertices.

A **directed** graph is *strongly connected iff* for any to vertices $u$ and $v$, if there is a path from $u$ to $v$, there must also be a path from $v$ to $u$.

# Connectivity

A graph is connected if there is a path between any two vertices.

A **directed** graph is *strongly connected iff* for any to vertices $u$ and $v$, if there is a path from $u$ to $v$, there must also be a path from $v$ to $u$.

A directed acyclic graph $\Leftrightarrow$ topological ordering, meaning the vertices can be ordered in a way that all edges point in the **same** direction.

# Connectivity

Breadth First Search (BFS) traverses a graph by layers and builds a BFS tree.

## Connectivity

Breadth First Search (BFS) traverses a graph by layers and builds a BFS tree.

It runs in $\Theta(n+m)$ time and is linear with respect to its input size.

# Connectivity

Breadth First Search (BFS) traverses a graph by layers and builds a BFS tree.

It runs in $\Theta(n + m)$ time and is linear with respect to its input size.

Depth First Search (DFS) traverses a graph by picking a point and following it until a dead end.

# Connectivity

Breadth First Search (BFS) traverses a graph by layers and builds a BFS tree.

It runs in $\Theta(n + m)$ time and is linear with respect to its input size.

Depth First Search (DFS) traverses a graph by picking a point and following it until a dead end.

It runs in $\Theta(n + m)$ time and is linear with respect to its input size.

# Introduction

A greedy algorithm is an algorithm that at each step, produces a **locally** optimal solution. The goal is to achieve **polynomial time** for a given algorithmic problem.

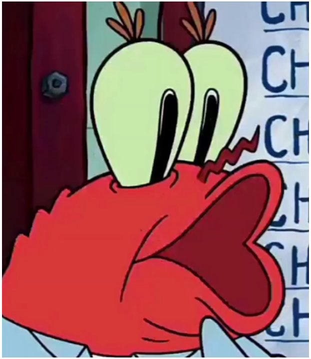

Figure: Sourced from Google

# Introduction

A greedy algorithm is an algorithm that at each step, produces a **locally** optimal solution. The goal is to achieve **polynomial time** for a given algorithmic problem.

It follows that at the end of the algorithm, the locally optimal solution returned can **approximate** a globally optimal solution. A greedy algorithm **never** backtracks.

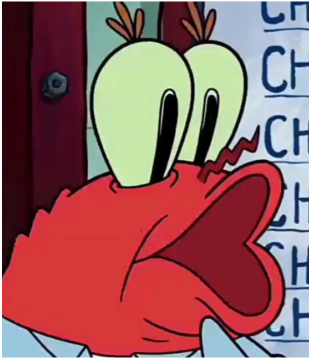

Figure: Sourced from Google

# Introduction

A greedy algorithm is an algorithm that at each step, produces a **locally** optimal solution. The goal is to achieve **polynomial time** for a given algorithmic problem.

It follows that at the end of the algorithm, the locally optimal solution returned can **approximate** a globally optimal solution. A greedy algorithm **never** backtracks.

See **the greedy stays ahead proof** for details on proving these types of algorithms.

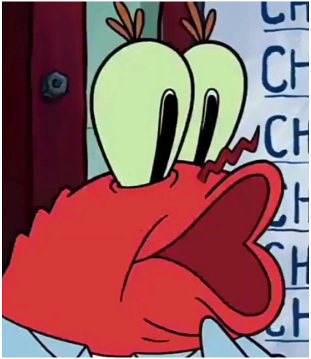

Figure: Sourced from Google

# Interval Scheduling Problem

Given a set of tasks to do over a period in time, how do you pick the **maximum** number of tasks that aren't in conflict?

# Interval Scheduling Problem

Given a set of tasks to do over a period in time, how do you pick the **maximum** number of tasks that aren't in conflict?

One greedy solution is to sort the set of tasks by their earliest $\rightarrow$ latest finishing time.

# Minimum Spanning Tree Problem

Given a **weighted**, graph, how do you construct a tree such that the accumulation of all the weights in the tree is minimized?

# Minimum Spanning Tree Problem

Given a **weighted**, graph, how do you construct a tree such that the accumulation of all the weights in the tree is minimized?

There are numerous greedy algorithms to generate an MST:

# Minimum Spanning Tree Problem

Given a **weighted**, graph, how do you construct a tree such that the accumulation of all the weights in the tree is minimized?

There are numerous greedy algorithms to generate an MST:

- Prim's Algorithm

# Minimum Spanning Tree Problem

Given a **weighted**, graph, how do you construct a tree such that the accumulation of all the weights in the tree is minimized?

There are numerous greedy algorithms to generate an MST:

- Prim's Algorithm
- Boruvka's Algorithm

# Minimum Spanning Tree Problem

Given a **weighted**, graph, how do you construct a tree such that the accumulation of all the weights in the tree is minimized?

There are numerous greedy algorithms to generate an MST:

- Prim's Algorithm
- Boruvka's Algorithm
- Kruskal's Algorithm

# Minimum Spanning Tree Problem

Given a **weighted**, graph, how do you construct a tree such that the accumulation of all the weights in the tree is minimized?

There are numerous greedy algorithms to generate an MST:

- Prim's Algorithm
- Boruvka's Algorithm
- Kruskal's Algorithm

Refer to the sources for proof of correctness and runtime analysis.

# Shortest Path Problem

Given a weighted graph with **no** negative weights, how do you find the shortest path?