Greedy Algorithms
Optimization Problems

- **Shortest path** is an example of an optimization problem: we wish to find the path with lowest weight.
- What is the general character of an optimization problem?
Optimization Problems

Ingredients:

• **Instances**: The possible inputs to the problem.

• **Solutions for Instance**: Each instance has an exponentially large set of valid solutions.

• **Cost of Solution**: Each solution has an easy-to-compute cost or value.

Specification

• **Preconditions**: The input is one instance.

• **Postconditions**: A valid solution with optimal cost. (minimum or maximum)
Greedy Solutions to Optimization Problems

Every two-year-old knows the greedy algorithm.

In order to get what you want, just start grabbing what looks best.

Surprisingly, many important and practical optimization problems can be solved this way.
Example 1: Making Change

**Problem:** Find the minimum # of quarters, dimes, nickels, and pennies that total to a given amount.
The Greedy Choice

Commit to the object that looks the "best"

Must prove that this locally greedy choice does not have negative global consequences.
Making Change Example

**Instance:** A drawer full of coins and an amount of change to return

Amount = 92¢

10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢
5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

**Solutions for Instance:**
A subset of the coins in the drawer that total the amount
Making Change Example

Instance: A drawer full of coins and an amount of change to return.

Amount = 92¢

Cost of Solution: The number of coins in the solution = 14

Goal: Find an optimal valid solution.
Making Change Example

Instance: A drawer full of coins and an amount of change to return

Amount = 92¢

10¢  10¢  10¢  10¢  10¢  10¢  10¢  10¢
  5¢  5¢  5¢  5¢  5¢  5¢  5¢  5¢
  1¢  1¢  1¢  1¢  1¢  1¢  1¢  1¢

Greedy Choice:
Start by grabbing quarters until exceeds amount, then dimes, then nickels, then pennies.

Does this lead to an optimal # of coins?

Cost of Solution: 7
Hard Making Change Example

Problem: Find the minimum # of 4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4-cent coin.

Consequences:
4+1+1 = 6  mistake
3+3=6      better

Greedy Algorithm does not work!
When Does It Work?

• Greedy Algorithms: Easy to understand and to code, but do they work?
• For most optimization problems, all greedy algorithms tried do not work (i.e. yield sub-optimal solutions)
• But some problems can be solved optimally by a greedy algorithm.
• The proof that they work, however, is subtle.
• As with all iterative algorithms, we use loop invariants.
## Designing an Algorithm

<table>
<thead>
<tr>
<th>Define Problem</th>
<th>Define Loop Invariants</th>
<th>Define Measure of Progress</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Illustration" /></td>
<td><img src="image2.png" alt="Illustration" /></td>
<td><img src="image3.png" alt="Illustration" /></td>
</tr>
<tr>
<td><strong>79 km to school</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Define Step</th>
<th>Define Exit Condition</th>
<th>Maintain Loop Inv</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image4.png" alt="Illustration" /></td>
<td><img src="image5.png" alt="Illustration" /></td>
<td><img src="image6.png" alt="Illustration" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Make Progress</th>
<th>Initial Conditions</th>
<th>Ending</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image7.png" alt="Illustration" /></td>
<td><img src="image8.png" alt="Illustration" /></td>
<td><img src="image9.png" alt="Illustration" /></td>
</tr>
</tbody>
</table>

- **Define Problem**
- **Define Loop Invariants**
- **Define Measure of Progress**
- **Define Step**
- **Define Exit Condition**
- **Maintain Loop Inv**
- **Make Progress**
- **Initial Conditions**
- **Ending**
The algorithm chooses the “best” object from amongst those not considered so far and either commits to it or rejects it.

Another object considered

All objects have been considered
Designing a Greedy Algorithm

< pre-condition >
CodeA
loop
< loop-invariant >
while → exit condition
    CodeB
end loop
CodeC
< post-condition >
Loop Invariant

We have not gone wrong. There is at least one optimal solution consistent with the choices made so far.
Initially no choices have been made and hence all optimal solutions are consistent with these choices.
Maintaining Loop Invariant

Must show that $\langle\text{loop-invariant}\rangle + \text{CodeB} \rightarrow \langle\text{loop-invariant}\rangle$

$\langle LI \rangle: \exists$ optimal solution $\text{OptS}_{LI}$ consistent with choices so far

$\text{CodeB}$: Commit to or reject next object

$\langle LI \rangle: \exists$ optimal soln $\text{OptS}_{Ours}$ consistent with prev objects + new object

*Note*: $\text{OptS}_{Ours}$ may or may not be the same as $\text{OptS}_{LI}$!

Proof must massage $\text{optS}_{LI}$ into $\text{optS}_{ours}$ and prove that $\text{optS}_{ours}$:

- is a valid solution
- is consistent both with previous and new choices.
- is optimal
Algorithm:
commits to
or rejects
next best
object

Prover:
Proves LI is
maintained.
His actions are
not part of the
algorithm

Fairy God Mother:
Holds the hypothetical
optimal sol $\text{optS}_{LI}$.
The algorithm
and prover do not
know $\text{optS}_{LI}$. 

Three Players
$\text{optS}_{LI}$
Proving the Loop Invariant is Maintained

• We need to show that the action taken by the algorithm maintains the loop invariant.

• There are 2 possible actions:
  – Case 1. Commit to current object
  – Case 2. Reject current object
Case 1. Committing to Current Object
Massaging $\text{optS}_L$ into $\text{optS}_o$

I hold $\text{optS}_o$ witnessing that there is an opt sol consistent with previous & new choices.

I commit to keeping another $25\,\text{¢}$

I instruct how to massage $\text{optS}_L$ into $\text{optS}_o$ so that it is consistent with previous & new choice.
As Time Goes On

I always hold an opt sol but one that keeps changing.

I keep making more choices.

I know that her optSLI is consistent with these choices. Hence, I know more and more of optSLI. In the end, I know it all.
Case 1A.
The object we commit to is already part of $\text{optS}_{LI}$
If it happens to be the case that the new object selected is consistent with the solution held by the fairy godmother, then we are done.
Case 1B.
The object we commit to is **not** part of $\text{optS}_{LI}$

\[ \text{optS}_{LI} \quad \text{partial solution} \quad \text{new object} \]
Case 1B. The object we commit to is **not** part of $\text{optSL}_L$

- This means that our partial solution is not consistent with $\text{optSL}_L$.
- The Prover must show that there is a new optimal solution $\text{optS}_{\text{ours}}$ that is consistent with our partial solution.
- This has two parts
  - All objects previously committed to must be part of $\text{optS}_{\text{ours}}$.
  - The new object must be part of $\text{optS}_{\text{ours}}$. 
Case 1B. The object we commit to is **not** part of $\text{optS}_{LI}$

- **Strategy of proof:** construct a consistent $\text{optS}_{ours}$ by replacing one or more objects in $\text{optS}_{LI}$ (but not in the partial solution) with another set of objects that includes the current object.

- We must show that the resulting $\text{optS}_{ours}$ is still
  - Valid
  - Consistent
  - Optimal
Case 1B. The object we commit to is **not** part of $\text{optS}_{\text{LL}}$

- **Strategy of proof:** construct a consistent $\text{optS}_{\text{ours}}$ by replacing one or more objects in $\text{optS}_{\text{LL}}$ (but not in the partial solution) with another set of objects that includes the current object.

- We must show that the resulting $\text{optS}_{\text{ours}}$ is still
  - Valid
  - Consistent
  - Optimal
Massaging $\text{optSLI}_L$ into $\text{optS}_{\text{ours}}$

Amount = 92¢

10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢
5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

Replace

• A different 25¢

With

• Alg’s 25¢
Massaging $\text{optS}_{\text{Ll}}$ into $\text{optS}_{\text{ours}}$

Amount $= 92¢$

10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢
5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

Replace

• A different 25¢

• $3 \times 10¢$

With

• Alg’s 25¢

• Alg’s $25¢ + 5¢$
Massaging $\text{optS}_L$ into $\text{optS}_o$

Amount $= 92\,\text{c}

25\,\text{c} \quad 25\,\text{c} \quad 25\,\text{c} \quad 25\,\text{c} \quad 25\,\text{c} \quad 25\,\text{c} \quad 25\,\text{c} \quad 25\,\text{c} \quad 25\,\text{c} \quad 25\,\text{c}

10\,\text{c} \quad 10\,\text{c} \quad 10\,\text{c} \quad 10\,\text{c} \quad 10\,\text{c} \quad 10\,\text{c} \quad 10\,\text{c} \quad 10\,\text{c} \quad 10\,\text{c} \quad 10\,\text{c}

5\,\text{c} \quad 5\,\text{c} \quad 5\,\text{c} \quad 5\,\text{c} \quad 5\,\text{c} \quad 5\,\text{c} \quad 5\,\text{c} \quad 5\,\text{c} \quad 5\,\text{c} \quad 5\,\text{c}

1\,\text{c} \quad 1\,\text{c} \quad 1\,\text{c} \quad 1\,\text{c} \quad 1\,\text{c} \quad 1\,\text{c} \quad 1\,\text{c} \quad 1\,\text{c} \quad 1\,\text{c} \quad 1\,\text{c}

Replace

• A different 25\,\text{c}
• $3 \times 10\,\text{c}$
• $2 \times 10\,\text{c} + 1 \times 5\,\text{c}$

With

• Alg’s 25\,\text{c}
• Alg’s 25\,\text{c} + 5\,\text{c}
• Alg’s 25\,\text{c}
Massaging $\text{optS}_{\text{Li}}$ into $\text{optS}_{\text{ours}}$

Amount = 92¢

$\begin{align*}
&25¢ \quad 25¢ \quad \boxed{25¢} \quad 25¢ \quad 25¢ \quad 25¢ \quad 25¢ \quad \boxed{25¢} \\
&10¢ \quad 10¢ \quad \boxed{10¢} \quad 10¢ \quad 10¢ \quad 10¢ \quad 10¢ \quad \boxed{10¢} \\
&\boxed{5¢} \quad \boxed{5¢} \quad 5¢ \quad 5¢ \quad 5¢ \quad 5¢ \quad 5¢ \quad \boxed{5¢} \\
&\boxed{1¢} \quad \boxed{1¢} \quad 1¢ \quad 1¢ \quad 1¢ \quad 1¢ \quad 1¢ \quad \boxed{1¢}
\end{align*}$

Replace

• A different 25¢
• $3 \times 10¢$
• $2 \times 10¢ + 1 \times 5¢$
• $1 \times 10¢ + 3 \times 5¢$

With

• Alg’s 25¢
• Alg’s 25¢ + 5¢
• Alg’s 25¢
• Alg’s 25¢
Massaging $\text{optS}_{\text{Li}}$ into $\text{optS}_{\text{ours}}$

Amount = 92¢

10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢ 10¢
5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢ 5¢
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

Replace

- A different 25¢
- $3 \times 10¢$
- $2 \times 10¢ + 1 \times 5¢$
- $1 \times 10¢ + 3 \times 5¢$
- ?? + $5 \times 1¢$

With

- Alg’s 25¢
- Alg’s 25¢ + 5¢
- Alg’s 25¢
- Alg’s 25¢
- Alg’s 25¢
## Must Consider All Cases

<table>
<thead>
<tr>
<th>optS_{Li}</th>
<th>#Coins</th>
<th>optS_{Ours}</th>
<th>#Coins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Q</td>
<td>1</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>3D</td>
<td>3</td>
<td>1Q 1N</td>
<td>2</td>
</tr>
<tr>
<td>2D 1N</td>
<td>3</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>2D 5P</td>
<td>7</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>1D 3N</td>
<td>4</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>1D 2N 5P</td>
<td>8</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>1D 1N 10P</td>
<td>12</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>1D 15P</td>
<td>16</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>5N</td>
<td>5</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>4N 5P</td>
<td>9</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>3N 10P</td>
<td>13</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>2N 15P</td>
<td>17</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>1N 20P</td>
<td>21</td>
<td>1Q</td>
<td>1</td>
</tr>
<tr>
<td>25P</td>
<td>25</td>
<td>1Q</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Note that in all cases our new solution** \( \text{optS}_{ours} \) **is:**
  - **Valid:** the sum is still correct
  - **Consistent** with our previous choices (we do not alter these).
  - **Optimal:** we never add more coins to the solution than we delete
Massaging $\text{optS}_{LI}$ into $\text{optS}_{ours}$

She now has something. We must prove that it is what we want.
Massaging $\text{optS}_{\text{LI}}$ into $\text{optS}_{\text{ours}}$

$\text{optS}_{\text{ours}}$ is valid

$\text{optS}_{\text{LI}}$ was valid and we introduced no new conflicts.
Total remains unchanged.

Replace
- A different 25¢
- 3×10¢
- 2×10¢ + 1×5¢
- 1×10¢ + 3×5¢
- ?? + 5×1¢

With
- Alg’s 25¢
- Alg’s 25¢ + 5¢
- Alg’s 25¢
- Alg’s 25¢
Massaging $\text{optS}_{\text{LI}}$ into $\text{optS}_{\text{ours}}$

$\text{optS}_{\text{ours}}$ is consistent with previous choices and we made it consistent with new.
Massaging $\text{optS}_{LI}$ into $\text{optS}_{ours}$

$\text{optS}_{ours}$ is optimal
We do not even know the cost of an optimal solution.

$\text{optS}_{LI}$ was optimal and $\text{optS}_{ours}$ cost (# of coins) is not bigger.

Replace
- A different $25¢$
- $3 \times 10¢$
- $2 \times 10¢ + 1 \times 5¢$
- $1 \times 10¢ + 3 \times 5¢$
- $?? + 5 \times 1¢$

With
- Alg’s $25¢$
- Alg’s $25¢ + 5¢$
- Alg’s $25¢$
- Alg’s $25¢$
Committing to Other Coins

- Similarly, we must show that when the algorithm selects a dime, nickel or penny, there is still an optimal solution consistent with this choice.

\[
\begin{align*}
\text{opt} S_{LI} + \text{dime} &\rightarrow \text{opt} S_{Ours} \\
\text{opt} S_{LI} + \text{nickel} &\rightarrow \text{opt} S_{Ours} \\
\text{opt} S_{LI} + \text{penny} &\rightarrow \text{opt} S_{Ours}
\end{align*}
\]
Example: Dimes

- We only commit to a dime when less than 25¢ is unaccounted for.
- Therefore the coins in $\text{optS}_{\text{Li}}$ that this dime replaces have to be dimes, nickels or pennies.

<table>
<thead>
<tr>
<th>$\text{optS}_{\text{Li}}$</th>
<th>#Coins</th>
<th>$\text{optS}_{\text{Ours}}$</th>
<th>#Coins</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D</td>
<td>1</td>
<td>1D</td>
<td>1</td>
</tr>
<tr>
<td>2N</td>
<td>2</td>
<td>1D</td>
<td>1</td>
</tr>
<tr>
<td>1N 5P</td>
<td>6</td>
<td>1D</td>
<td>1</td>
</tr>
<tr>
<td>10P</td>
<td>10</td>
<td>1D</td>
<td>1</td>
</tr>
</tbody>
</table>
Committing to Other Coins

• We must consider all possible coins we might select:
  – **Quarter**: Swap for another quarter, 3 dimes (with a nickel) etc.
  – **Dime**: Swap for another dime, 2 nickels, 1 nickel + 5 pennies etc.
  – **Nickel**: Swap for another nickel or 5 pennies.
  – **Penny**: Swap for another penny.
Massaging $\text{optS}_{LI}$ into $\text{optS}_{ours}$

- $\text{optS}_{ours}$ is valid
- $\text{optS}_{ours}$ is consistent
- $\text{optS}_{ours}$ is optimal

Maintaining Loop Invariant

- $\neg\langle\text{exit Cond}\rangle$
- $\text{codeB}$
Case 2. Rejecting the Current Object
Rejecting the Current Object

Strategy of Proof:

1. There is at least one optimal solution $\text{optS}_L$ consistent with previous choices.
2. Any optimal solution consistent with previous choices cannot include current object.
3. Therefore $\text{optS}_L$ cannot include current object.
Rejecting an Object

• Making Change Example:
  – We only reject an object when including it would make us exceed the total.
  – Thus optS_{Li} cannot include the object either.
I hold \( \text{optSLI} \) witnessing that there is an opt sol consistent with previous choices.

I must make sure that what the Fairy God Mother has is consistent with this new choice.

I reject the next \( 25\)¢

Amount = 92¢
The Algorithm has

\[ 92\text{¢} - 75\text{¢} = 17\text{¢} < 25\text{¢} \] unchoosen.

Fairy God Mother must have \(< 25\text{¢}\) that I don’t know about.

\(\text{optSLI} \) does not contain the \(25\text{¢}\) either.
Clean up loose ends

Alg has committed to or rejected each object. Has yielded a solution $S$.

$\exists$ opt sol consistent with these choices. $S$ must be optimal.

Alg returns $S$.
Making Change Example

Problem: Find the minimum # of quarters, dimes, nickels, and pennies that total to a given amount.

Greedy Choice: Start by grabbing quarters until exceeds amount, then dimes, then nickels, then pennies.

Does this lead to an optimal # of coins?

Yes
Hard Making Change Example

Problem: Find the minimum # of 4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4 coin.
I will now instruct how to massage $\text{optSLI}$ into $\text{optS_ours}$ so that it is consistent with previous & new choice.

Amount = 6¢

4¢ 4¢ 4¢ 4¢ 4¢ 4¢ 4¢ 4¢ 4¢ 4¢
3¢ 3¢ 3¢ 3¢ 3¢ 3¢ 3¢ 3¢ 3¢ 3¢
1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢ 1¢

I hold $\text{optSLI}$.

I commit to keeping a 4¢

I will now instruct how to massage $\text{optSLI}$ into $\text{optS_ours}$ so that it is consistent with previous & new choice.

Oops!
Hard Making Change Example

Problem: Find the minimum # of 4, 3, and 1 cent coins to make up 6 cents.

Greedy Choice: Start by grabbing a 4 coin.

Consequences:

4+1+1 = 6   mistake
3+3=6      better

Greedy Algorithm does not work!
**Analysing Arbitrary Systems of Denominations**

- Suppose we are given a system of coin denominations. How do we decide whether the greedy algorithm is optimal?
- It turns out that this problem can be solved in $O(D^3)$ time, where $D =$ number of denominations (e.g., $D=6$ in Canada) (Pearson 1994).
Designing Optimal Systems of Denominations

In Canada, we use a 6 coin system:
1 cent, 5 cents, 10 cents, 25 cents, 100 cents and 200 cents.

Assuming that $N$, the change to be made, is uniformly distributed over \{1, \ldots, 499\}, the expected number of coins per transaction is 5.9.

The optimal (but non-greedy) 6-coin systems are \((1,6,14,62,99,140)\) and \((1,8,13,69,110,160)\), each of which give an expected 4.67 coins per transaction.

The optimal greedy 6-coin systems are \((1,3,8,26,64,\{202 \text{ or } 203 \text{ or } 204\})\) and \((1,3,10,25,79,\{195 \text{ or } 196 \text{ or } 197\})\) with an expected cost of 5.036 coins per transaction.
Summary

- We must prove that every coin chosen or rejected in greedy fashion still leaves us with a solution that is
  - Valid
  - Consistent
  - Optimal

- We prove this using an inductive ‘cut and paste’ method.

- We know from the previous iteration we have a partial solution $S_{part}$ that is part of some complete optimal solution $\text{opt } S_{LI}$. 
Summary

• **Selecting a coin**: we show that we can replace a subset of the coins in $\text{optS}_{LI} \setminus S_{part}$ with the selected coin (+ perhaps some additional coins).
  
  – **Valid** because we ensure that the trade is fair (sums are equal)
  
  – **Consistent** because we have not touched $S_{part}$
  
  – **Optimal** because the number of the new coin(s) is no greater than the number of coins they replace.

• **Rejecting a coin**: we show that we only reject a coin when it could not be part of $\text{optS}_{LI}$.
Example 2: Job/Event Scheduling
The Job/Event Scheduling Problem

Ingredients:

• **Instances**: Events with starting and finishing times

  
  \[<<s_1,f_1>,<<s_2,f_2>,\ldots,<<s_n,f_n>>.\]

• **Solutions**: A set of events that do not overlap.

• **Value of Solution**: The number of events scheduled.

• **Goal**: Given a set of events, schedule as many as possible.

• **Example**: Scheduling lectures in a lecture hall.
Possible Criteria for Defining “Best”

- - - - - - - -  Optimal

- - - - - - - - - -

Greedy Criterion: The Shortest Event

Motivation: Does not book the room for a long period of time.

- Schedule first
- Optimal

Counter Example
Possible Criteria for Defining “Best”

Greedy Criterion: The Earliest Starting Time

Motivation: Gets room in use as early as possible

Counter Example
Possible Criteria for Defining “Best”

Optimal

Greedy Criterion:
Conflicting with the Fewest Other Events

Motivation: Leaves many that can still be scheduled.

Schedule first
Optimal

Counter Example
Possible Criteria for Defining “Best”

Greedy Criterion: Earliest Finishing Time

Motivation: Schedule the event that will free up your room for someone else as soon as possible.
The Greedy Algorithm

algorithm Scheduling((s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n))

⟨pre-cond⟩: The input consists of a set of events.
⟨post-cond⟩: The output consists of a schedule that maximizes the number of events scheduled.

begin
    Sort the events based on their finishing times f_i
    Commit = ∅ % The set of events committed to be in the schedule
    loop i = 1 \ldots n % Consider the events in sorted order.
        if (event i does not conflict with an event in Commit) then
            Commit = Commit \cup \{i\}
        end if
    end loop
    return(Commit)
end algorithm
Massaging $\text{optS}_{\text{LJ}}$ into $\text{optS}_{\text{ours}}$

Start by adding new event $i$.
Delete events conflicting with job $i$. 

Massaging $\text{optS}_{LI}$ into $\text{optS}_{ours}$ was valid and we removed any new conflicts.
Massaging $\text{optS}_\text{LI}$ into $\text{optS}_\text{ours}$ was consistent with our prior choices. We added event $i$. Events in Commit don’t conflict with event $i$ and hence were not deleted. $\text{optS}_\text{ours}$ is consistent with our choices.
Massaging $\text{optS}_{LI}$ into $\text{optS}_{ours}$

$\text{optS}_{LI}$ was optimal.

If we delete at most one event then $\text{optS}_{ours}$ is optimal too.
Massaging $\text{optS}_{\text{LI}}$ into $\text{optS}_{\text{ours}}$

Deleted at most one event $j$

$i < j \Rightarrow f_i \leq f_j$

$[j \text{ conflicts with } i] \Rightarrow s_j \leq f_i$

$\Rightarrow j$ runs at time $f_i$.

Two such $j$ conflict with each other.

Only one in $\text{optS}_{\text{LI}}$. 
Massaging $\text{optS}_\text{LI}$ into $\text{optS}_\text{ours}$

- $\text{optS}_\text{ours}$ is valid
- $\text{optS}_\text{ours}$ is consistent
- $\text{optS}_\text{ours}$ is optimal

Maintaining Loop Invariant

- $\neg$<exit Cond> $\Rightarrow$ codeB

Case 1
Massaging $\text{optS}_{\text{LI}}$ into $\text{optS}_{\text{ours}}$

I hold $\text{optS}_{\text{LI}}$ witnessing that there is an opt sol consistent with previous choices.

I reject next event i.

Event i conflicts with events committed to so it can’t be in $\text{optS}_{\text{LI}}$ either.
Massaging $\text{optS}_{\text{Li}}$ into $\text{optS}_{\text{ours}}$

Maintaining Loop Invariant

$\langle \text{LI} \rangle$
$\neg$ $\langle \text{exit Cond} \rangle$
$\text{codeB}$

$\langle \text{LI} \rangle$
Clean up loose ends

<loop-invariant>
<exit Cond>  codeC
<postCond>

<exit Cond>  Alg commits to or reject each event. Has a solution \( S \).

<LI>  \( \exists \) opt sol consistent with these choices. \( S \) must be optimal.

codeC  Alg returns \( \text{optS} \).
<postCond>
Running Time

Greedy algorithms are very fast because they only consider each object once.

Checking whether next event $i$ conflicts with previously committed events requires only comparing it with the last such event.
Running Time

```
algorithm Scheduling ((\{s_1, f_1\}, \{s_2, f_2\}, \ldots, \{s_n, f_n\}))

\text{\textbf{pre-cond}}: \text{The input consists of a set of events.}

\text{\textbf{post-cond}}: \text{The output consists of a schedule that maximizes the number of events scheduled.}

\begin{verbatim}
begin
    Sort the events based on their finishing times \( f_i \)
    \text{Commit} = \emptyset \quad \% \text{The set of events committed to be in the schedule}
    \text{loop} i = 1 \ldots n \quad \% \text{Consider the events in sorted order.}
        \text{if (event} i \text{does not conflict with an event in Commit)}\text{ then}
            \text{Commit} = \text{Commit} \cup \{i\}
    end \text{loop}
    return(Commit)
end \text{algorithm}
\end{verbatim}

\[ \theta(n \log n) \]

\[ \theta(n) \]

\[ T(n) = \theta(n \log n) \]
Example 3: Minimum Spanning Trees
Minimum Spanning Trees

• Example Problem
  – You are planning a new terrestrial telecommunications network to connect a number of remote mountain villages in a developing country.
  – The cost of building a link between pairs of neighbouring villages \((u,v)\) has been estimated: \(w(u,v)\).
  – You seek the minimum cost design that ensures each village is connected to the network.
  – The solution is called a minimum spanning tree (MST).
Minimum Spanning Trees

The problem is defined for any undirected, connected, weighted graph.

The weight of a subset $T$ of a weighted graph is defined as:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

Thus the MST is the spanning tree $T$ that minimizes $w(T)$. 
Building the Minimum Spanning Tree

- Iteratively construct the set of edges $A$ in the MST.
- Initialize $A$ to $\emptyset$.
- As we add edges to $A$, maintain a Loop Invariant:
  - $A$ is a subset of some MST
- Maintain loop invariant and make progress by only adding **safe** edges.
- An edge $(u,v)$ is called **safe** for $A$ iff $A \cup \{u,v\}$ is also a subset of some MST.
Finding a safe edge

- Idea: Every 2 disjoint subsets of vertices must be connected by at least one edge.

- Which one should we choose?
Some definitions

• A cut \((S, V-S)\) is a partition of vertices into disjoint sets \(S\) and \(V-S\).

• Edge \((u,v) \in E\) crosses cut \((S, V-S)\) if one endpoint is in \(S\) and the other is in \(V-S\).

• A cut respects a set of edges \(A\) iff no edge in \(A\) crosses the cut.

• An edge is a light edge crossing a cut iff its weight is minimum over all edges crossing the cut.
Minimum Spanning Tree Theorem

- Let
  - $A$ be a subset of some MST
  - $(S, V-S)$ be a cut that respects $A$
  - $(u,v)$ be a light edge crossing $(S, V-S)$

- Then
  - $(u,v)$ is safe for $A$.

Basis for a greedy algorithm
Proof

- Let $G$ be a connected, undirected, weighted graph.
- Let $T$ be an MST that includes $A$.
- Let $(S, V-S)$ be a cut that respects $A$.
- Let $(u,v)$ be a light edge between $S$ and $V-S$.
- If $T$ contains $(u,v)$ then we’re done.

---

Edge $\in T$

---

Edge $\notin T$
• Suppose $T$ does not contain $(u,v)$
  
  - Can construct different MST $T'$ that includes $A \cup (u,v)$
  
  - The edge $(u,v)$ forms a cycle with the edges on the path $p$ from $u$ to $v$ in $T$.
  
  - There is at least one edge in $p$ that crosses the cut: let that edge be $(x,y)$
  
  - $(x,y)$ is not in $A$, since the cut $(S,V-S)$ respects $A$.
  
  - Form new spanning tree $T'$ by deleting $(x,y)$ from $T$ and adding $(u,v)$.
  
  - $w(T') \leq w(T)$, since $w(u,v) \leq w(x,y) \Rightarrow T'$ is an MST.
  
  - $A \subseteq T'$, since $A \subseteq T$ and $(x,y) \not\in A \Rightarrow A \cup (u,v) \subseteq T'$
  
  - Thus $(u,v)$ is safe for $A$. 
End of Lecture 17

Nov 13, 2007
Kruskal’s Algorithm for computing MST

- Starts with each vertex being its own component.
- Repeatedly merges two components into one by choosing the light edge that crosses the cut between them.
- Scans the set of edges in monotonically increasing order by weight (greedy).
Kruskal’s Algorithm: Loop Invariant

Let $A = \text{solution under construction}$. Let $E_i = \text{the subset of } i \text{ lowest-weight edges thus far considered}$

< loop-invariant >:

$\exists \text{MST } T:$

1) $A \in T,$

2) $\forall (u,v) \in E_i: (u,v) \in A \text{ or } (u,v) \not\in T$
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example
Kruskal’s Algorithm: Example

Finished!
Disjoint Set Data Structures

• Disjoint set data structures can be used to represent the disjoint connected components of a graph.

• Make-Set($x$) makes a new disjoint component containing only vertex $x$.

• Union($x,y$) merges the disjoint component containing vertex $x$ with the disjoint component containing vertex $y$.

• Find-Set($x$) returns a vertex that represents the disjoint component containing $x$. 
Disjoint Set Data Structures

• Most efficient representation represents each disjoint set (component) as a tree.

• Time complexity of a sequence of $m$ operations, $n$ of which are Make-Set operations, is:

$$O(m \times \alpha(n))$$

where $\alpha(n)$ is Ackerman's function, which grows extremely slowly.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\alpha(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>2047</td>
<td>3</td>
</tr>
<tr>
<td>$10^{80}$</td>
<td>4</td>
</tr>
</tbody>
</table>
Kruskal's Algorithm for computing MST

Kruskal(G, w)
A = ∅
for each vertex v ∈ V[G]
    Make-Set(v)
sort E[G] into nondecreasing order: E[1...n]
for i = 1: n
<loop-invariant>:
∃ MST T : 1) A ∈ T,
  2) ∀(u, v) ∈ E[1...i−1]: (u, v) ∈ A or (u, v) /∈ T
(u, v) = E[i]
if Find-Set(u) ≠ Find-Set(v)
    A = A ∪ {(u, v)}
    Union(u, v)
return A

Running Time = O(E log E)
= O(E log V)
Prim’s Algorithm for Computing MST

• Build one tree \( A \)
• Start from arbitrary root \( r \)
• At each step, add light edge connecting \( V_A \) to \( V - V_A \) (greedy)
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example
Prim’s Algorithm: Example

Finished!
Finding light edges quickly

- All vertices not in the partial MST formed by $A$ reside in a min-priority queue.
- Key($v$) is minimum weight of any edge $(u,v)$, $u \in V_A$.
- Priority queue can be implemented as a min heap on key($v$).
- Each vertex in queue knows its potential parent in partial MST by $\pi[v]$.
Prim's Algorithm

Let \( A = \{(v, \pi[v]) : v \in V - \{r\} - Q\} \)

Let \( V_A = V - Q \)

<loop-invariant>:
1. \( \exists \text{MST } T : A \in T \)
2. \( \forall v \in Q, \text{if } \pi[v] \neq \text{NIL} \)

then \( \text{key}[v] = \text{weight of light edge connecting } v \text{ to } V_A \)

\[
\text{PRIM}(V, E, w, r) \\
Q \leftarrow \emptyset \\
\text{for each } u \in V \\
\quad \text{do } \text{key}[u] \leftarrow \infty \\
\quad \pi[u] \leftarrow \text{NIL} \\
\quad \text{INSERT}(Q, u) \\
\text{DECREASE-KEY}(Q, r, 0) \\
\quad \text{▷ } \text{key}[r] \leftarrow 0 \\
\text{while } Q \neq \emptyset \\
\quad \text{do } u \leftarrow \text{EXTRACT-MIN}(Q) \\
\quad \text{for each } v \in \text{Adj}[u] \\
\quad \quad \text{do if } v \in Q \text{ and } w(u, v) < \text{key}[v] \\
\quad \quad \text{then } \pi[v] \leftarrow u \\
\quad \quad \text{DECREASE-KEY}(Q, v, w(u, v))
\]
Prim's Algorithm

\[ \text{PRIM}(V, E, w, r) \]
\[ Q \leftarrow \emptyset \]
\[ \text{for each } u \in V \]
\[ \text{do } key[u] \leftarrow \infty \]
\[ \pi[u] \leftarrow \text{NIL} \]
\[ \text{INSERT}(Q, u) \]
\[ \text{DECREASE-KEY}(Q, r, 0) \quad \triangleright \text{key}[r] \leftarrow 0 \]
\[ \text{while } Q \neq \emptyset \]
\[ \text{do } u \leftarrow \text{EXTRACT-MIN}(Q) \]
\[ \text{for each } v \in \text{Adj}[u] \]
\[ \text{do if } v \in Q \text{ and } w(u, v) < \text{key}[v] \]
\[ \text{then } \pi[v] \leftarrow u \]
\[ \text{DECREASE-KEY}(Q, v, w(u, v)) \]

\( O(V) \)

Executed \(|V|\) times

\( O(\log V) \)

Executed \(|E|\) times

\( O(\log V) \)

Running Time = \( O(E \log V) \)
Algorithm Comparison

- Both Kruskal’s and Prim’s algorithm are greedy.
  - Kruskal’s: Queue is static (constructed before loop)
  - Prim’s: Queue is dynamic (keys adjusted as edges are encountered)