Greedy Algorithms

- Make choice based on immediate rewards rather than looking ahead to see the optimum
- In many cases this is effective as the look ahead variation can require exponential time as the number of possible factors combine
  - Best way to get to a destination?
  - Without other knowledge, going down the road that heads in the direction of the destination is probably a reasonable choice
    - This is the greedy approach and can do well in certain cases
    - It also makes the decision much easier than considering all alternatives
  - May not be the optimal decision, in contrast to considering all possible alternatives and then subsequent alternatives, etc.
Next Move in Chess

- What move should you do next
- Could do move which leaves you in the best material position after one move
  - Greedy since it takes best now without considering the ramifications of what could occur later
- Could do a 2nd order greedy approach
  - Look two moves ahead
  - More time consuming
  - Better?
- $N^{th}$ order greedy? – until game decided
  - No longer greedy since consider full situation
  - Exponential time required
Coins Problem

- Given:
  - An unbounded supply of coins of specific denominations
  - An integer $c$
- Find: Minimal number of coins that add up to $c$
- What is your greedy algorithm?
Coins Algorithm

Repeat until sum = c
  Add the largest coin which does not cause sum to exceed c

Does this work and is it Optimal?
  - Assume denominations are 50¢, 20¢, 3¢, 2¢
  - Try it with goal of 75¢, 60¢
  - Now assume denominations are 50¢, 20¢, 3¢, 1¢

Greedy Philosophy

- Build up a solution piece by piece
- Always choose the next piece that offers the most obvious and immediate benefit
- Without violating given constraints
Given a connected undirected graph we would like to find the “cheapest” connected version of that graph.

Remove all extra edges leaving just enough to be connected – it will be a tree.

Find the tree that has the smallest sum of edge lengths.

Given $G = (V, E)$ and edge weights $w_e$, find the tree $T = (V, E')$ where $E' \subseteq E$ and which also minimizes $\sum_{e \in E'} w_e$.

This is the Minimum Spanning Tree.

Not necessarily unique.
MST – Minimum Spanning Tree

What greedy algorithm might we use assuming we start with the entire graph
What greedy algorithm might we use assuming we start with the entire graph

Iteratively take away the biggest remaining link in the graph which does not disconnect the graph
  - Is it a greedy approach?

How do we prove if it works or not
  - Counterexamples – natural first attempt
  - If no easily found counterexamples, we then seek a more formal proof
Sometimes greedy algorithms can also be optimal

Simple greedy optimal algorithm for MST

1. Start with an empty graph
2. Repeatedly add the next smallest edge from $E$ that does not produce a cycle

How might we test for cycles and what would the complexity be?
Kruskal's Algorithm

procedure kruskal(G, w)
Input: A connected undirected graph G = (V, E) with edge weights w_e
Output: A minimum spanning tree defined by the edges X

for all u ∈ V:
    makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
    if find(u) ≠ find(v):
        add edge {u, v} to X
        union(u, v)

Represents nodes in disjoint sets
makeset(u): create a singleton set containing just u
find(u): to which set does u belong?
union(u,v): merge the sets containing u and v
find(u) = find(v) if u and v are in the same set, which means they are in the same connected component
Why not union nodes that are already in the same set?
for all $u \in V$:
    makeset($u$)

$X = \{\}$

Sort the edges $E$ by weight

Make a disjoint set for each vertex

{1} {2} {3} {4} {5} {6} {7}
Kruskal’s Algorithm

Sort edges by weight

1: \{1,2\}  \{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\}
2: \{2,3\}
3: \{4,5\}
3: \{6,7\}
4: \{1,4\}
4: \{2,5\}
4: \{4,7\}
5: \{3,5\}

\ldots
for all edges \( \{u, v\} \in E \), in increasing order of weight:
if \( \text{find}(u) \neq \text{find}(v) \):
    add edge \( \{u, v\} \) to \( X \)
union\( (u, v) \)

Add first edge to \( X \) if no cycle created

1: \{1,2\} \quad \{1\} \quad \{2\} \quad \{3\} \quad \{4\} \quad \{5\} \quad \{6\} \quad \{7\}
2: \{2,3\}
3: \{4,5\}
3: \{6,7\}
4: \{1,4\}
4: \{2,5\}
4: \{4,7\}
5: \{3,5\}
\ldots
\ldots
for all edges \( \{u, v\} \in E \), in increasing order of weight:

if \( \text{find}(u) \neq \text{find}(v) \):
    add edge \( \{u, v\} \) to \( X \)
union(u, v)
Kruskal’s Algorithm

Process each edge in order

1: {1,2}  {1,2} {3} {4} {5} {6} {7}
2: {2,3}  {1,2,3} {4} {5} {6} {7}
3: {4,5}
3: {6,7}
4: {1,4}
4: {2,5}
4: {4,7}
5: {3,5}
Note that each set is a connected component of $G$
Kruskal’s Algorithm

1: \{1,2\}  \{1,2\} \{3\} \{4\} \{5\} \{6\} \{7\}
2: \{2,3\}  \{1,2,3\} \{4\} \{5\} \{6\} \{7\}
3: \{4,5\}  \{1,2,3\} \{4,5\} \{6\} \{7\}
3: \{6,7\}  \{1,2,3\} \{4,5\} \{6,7\}
4: \{1,4\}
4: \{2,5\}
4: \{4,7\}
5: \{3,5\}
\ldots
Kruskal’s Algorithm

1: \{1,2\} \{1,2\} \{3\} \{4\} \{5\} \{6\} \{7\}
2: \{2,3\} \{1,2,3\} \{4\} \{5\} \{6\} \{7\}
3: \{4,5\} \{1,2,3\} \{4,5\} \{6\} \{7\}
3: \{6,7\} \{1,2,3\} \{4,5\} \{6,7\}
4: \{1,4\} \{1,2,3,4,5\} \{6,7\}
4: \{2,5\}
4: \{4,7\}
5: \{3,5\}

\[\ldots\]
for all edges \( \{u,v\} \in E \), in increasing order of weight:

if \( \text{find}(u) \neq \text{find}(v) \):

add edge \( \{u,v\} \) to \( X \)

union\((u,v)\)

Must join separate components

1: \{1,2\} \{1,2\} \{3\} \{4\} \{5\} \{6\} \{7\}
2: \{2,3\} \{1,2,3\} \{4\} \{5\} \{6\} \{7\}
3: \{4,5\} \{1,2,3\} \{4,5\} \{6\} \{7\}
3: \{6,7\} \{1,2,3\} \{4,5\} \{6,7\}
4: \{1,4\} \{1,2,3,4,5\} \{6,7\}
4: \{2,5\} rejected
4: \{4,7\}
5: \{3,5\}

...
Kruskal’s Algorithm

Stop when all vertices connected

1: \{1,2\} \{1,2\} \{3\} \{4\} \{5\} \{6\} \{7\}
2: \{2,3\} \{1,2,3\} \{4\} \{5\} \{6\} \{7\}
3: \{4,5\} \{1,2,3\} \{4,5\} \{6\} \{7\}
3: \{4,5\} \{1,2,3\} \{4,5\} \{6,7\}
4: \{1,4\} \{1,2,3,4,5\} \{6,7\}
4: \{2,5\} rejected
4: \{4,7\} \{1,2,3,4,5,6,7\} done
5: \{3,5\}
  
  ...
Kruskal's Algorithm – Is it correct?

- **Tree Properties**
  - Tree of $n$ nodes has $n-1$ edges
  - Any connected undirected graph with $|E| = |V|-1$ is a tree
  - An undirected graph is a tree iff there is a unique path between any pair of nodes

- **Kruskal's**
  - If we add $|V|-1$ edges with no cycles then we will have a tree by the above properties
  - But how do we know if it is minimal?
  - Make sure you review the proof in the book
Kruskal's Algorithm: Inductive Proof

- **Theorem:** *Kruskal’s Algorithm finds a minimum spanning tree*
- **Basis:** $X_0 = \emptyset$ and $G$ is connected so a solution must exist
  - Is this a correct partial solution?
- **Assumption:** At any moment edges $X_t$ are part of an MST for $G$
- **Inductive step is the Cut Property**
- **Cut Property:** Assume edges $X$ are part of an MST for $G=(V,E)$. Pick any subset $S$ for which $X$ does not cross between $S$ and $V-S$, and let $e$ be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST.
Cut Property: Assume edges $X$ are part of an MST for $G=(V,E)$. Pick any subset $S$ for which $X$ does not cross between $S$ and $V-S$, and let $e$ be the lightest edge across this partition. Then $X \cup \{e\}$ is part of some MST. – Why?

1. Assume edges $X$ are part of a partial MST $T$ (Inductive hypothesis)
2. If $e$ is a part of $T$ then done, so consider case where $e$ is not part of $T$.
3. Now add $e$ to $T$, creating a cycle, meaning there must be another edge $e'$ across the cut $(S, V-S)$ (note that $\text{weight}(e') \geq \text{weight}(e)$)
4. Create $T'$ by replacing $e'$ with $e$: $T' = T \cup \{e\} - \{e'\}$
5. $T'$ is a tree since it is connected and still has $|V|-1$ edges.
6. $T'$ is an MST since:
   1. $\text{weight}(T') = \text{weight}(T) + w(e) - w(e')$
   2. both $e$ and $e'$ cross the cut $(S, V-S)$
   3. by cut property $e$ was the lightest edge across the cut
   4. Therefore, $w(e') = w(e)$ and $T'$ is an MST

Thus, any (and only a) lightest edge across a cut will lead to an MST.
Which edge is \( e' \)?
procedure kruskal\((G, w)\)
Input: A connected undirected graph \(G = (V, E)\) with edge weights \(w_e\)
Output: A minimum spanning tree defined by the edges \(X\)

for all \(u \in V:\)
   makeset\((u)\)

\(X = {}\)
Sort the edges \(E\) by weight
for all edges \(\{u, v\} \in E\), in increasing order of weight:
   if \(\text{find}(u) \neq \text{find}(v)\):
      add edge \(\{u, v\}\) to \(X\)
      union\((u, v)\)
procedure _kruskal_\((G, w)\)
Input: A connected undirected graph \(G = (V, E)\) with edge weights \(w_e\)
Output: A minimum spanning tree defined by the edges \(X\)

for all \(u \in V\):
    makeset\((u)\)

\(X = \{\}\)  
Sort the edges \(E\) by weight  
for all edges \(\{u, v\} \in E\), in increasing order of weight:
    if \(\text{find}(u) \neq \text{find}(v)\):
        add edge \(\{u, v\}\) to \(X\)
    union\((u, v)\)

Data structure will represent the state as a collection of disjoint sets where each set represents a connected component (sub-tree) of \(G\)
- \(\text{makeset}(u)\): create a singleton set containing just \(u\): \(|V|\) times
- Sorting edges is \(O(|E| \log |E|)\) (or \(O(|E| \log |V|)\) since differs by a constant factor)
- \(\text{find}(u)\): to which set does \(u\) belong?: \(2|E|\) times
- \(\text{union}(u, v)\): merge the sets containing \(u\) and \(v\): \(|V|-1\) times (why?)
Complexity is \(O(|E| \log |E| + |E| \times \text{find\_complexity} + |V| \times \text{union\_complexity})\)
Directed Tree Representation of Disjoint Sets

- Nodes are stored in an array (easy access) and have a pointer and a rank value
- $\pi(x)$ is a pointer to parent
- If $\pi(x)$ points to itself it is the root/name of the disjoint set
- $\text{find}(x)$ returns the unique root/name of the set
- $\text{union}(x, y)$ merges sets to which $x$ and $y$ belong and keeps the tree balanced so that the maximum depth of the tree representing the disjoint set is $\log |V|$.
- $\text{rank}(x)$ is the height of the sub-tree rooted at node $x$
- $\text{makeset}$ is $O(1)$
- $\text{find}$ and $\text{union}$ complexity?

```plaintext
procedure makeset(x)
    $\pi(x) = x$
    rank(x) = 0

function find(x)
    while $x \neq \pi(x)$:  $x = \pi(x)$
    return $x$

procedure union(x, y)
    $r_x = \text{find}(x)$
    $r_y = \text{find}(y)$
    if $r_x = r_y$:  return
    if rank($r_x$) > rank($r_y$):
        $\pi(r_y) = r_x$
    else:
        $\pi(r_x) = r_y$
    if rank($r_x$) = rank($r_y$):  rank($r_y$) = rank($r_y$) + 1
```
After `makeset(A), makeset(B), ..., makeset(G)`:

```
A^0  B^0  C^0  D^0  E^0  F^0  G^0
```

After `union(A, D), union(B, E), union(C, F)`:

```
D^1  E^1  F^1  G^0

A^0  B^0  C^0
```

After `union(C, G), union(E, A)`:

```
P^2

D^1

A^0

E^1

B^0

F^1

C^0

G^0
```

After `union(B, G)`:

```
D^2

E^1

F^1

A^0

B^0

C^0

G^0
```
Kruskal Algorithm Complexity

- \(O(|E|\log|V|)\) for initially sorting the edges
  - Sort is actually \(O(|E|\log|E|)\)
  - Note that for a dense graph \(|E| \approx |V|^2\)
  - But remember that \(\log n^2 = 2\log n\) so they only differ by a constant factor and thus \(\Theta|E|\log|V| = \Theta(|E|\log|E|)\)

- \(O(|V|)\) for the initial makesets

- \(O(|E|\log|V|)\) for the iteration body including the find and union (merge) operations

- Total complexity: \(O(|E|\log|V|)\)
Prim's Algorithm

- Any algorithm which follows the Cut Property can discover an MST for a graph $G$
- Prim's algorithm differs from Kruskal's by growing $S$ as a single tree
  - Intermediate set of edges $X$ always forms one sub-tree
  - $S$ is chosen to be the set of $X$’s vertices (the partial MST)
  - On each iteration, $X$ grows by one edge
    - namely the lightest edge between a vertex in $S$ and a vertex outside $S$.
- The algorithm is basically Dijkstra's algorithm except that the key value for a node is the lightest incoming edge from $S$.
Prims's Algorithm

procedure prim(G, w)
Input: A connected undirected graph $G = (V, E)$ with edge weights $w_e$
Output: A minimum spanning tree defined by the array prev

for all $u \in V$:
    cost($u$) = $\infty$
    prev($u$) = nil
Pick any initial node $u_0$
    cost($u_0$) = 0

$H = \text{makequeue}(V)$  \hspace{1cm} (priority queue, using cost-values as keys)
while $H$ is not empty:
    $v = \text{deletemin}(H)$
    for each $\{v, z\} \in E$:
        if cost($z$) > $w(v, z)$:
            cost($z$) = $w(v, z)$
            prev($z$) = $v$
            decreasekey($H, z$)

- Decreasekey section does not sum path length, but just updates the key
  with the edge cost into the node
- Same complexity values as Dijkstra's Algorithm
**Prim’s Algorithm**

Choose arbitrary starting vertex

\[ S = \{5\} \]

while \( H \) is not empty:
  \( v = \text{deletemin}(H) \)
  for each \( \{v, z\} \in E \):
    if \( \text{cost}(z) > w(v, z) \):
      \( \text{cost}(z) = w(v, z) \)
      \( \text{prev}(z) = v \)
      \( \text{decreasekey}(H, z) \)

1: \( \infty \)
2: \( \infty \)
3: \( \infty \)
4: \( \infty \)
5: 0
6: \( \infty \)
7: \( \infty \)
while $H$ is not empty:
   $v = \text{deletemin}(H)$
   for each $\{v, z\} \in E$:
     if $\text{cost}(z) > w(v, z)$:
       $\text{cost}(z) = w(v, z)$
       $\text{prev}(z) = v$
       $\text{decreasekey}(H, z)$

Choose arbitrary starting vertex

$X : \quad S = \{5\}$

1: $\infty$
2: 4
3: 5
4: 3
6: 8
7: 8
Prim’s Algorithm

Pick shortest edge to leave $S$

$$X : \{4, 5\}$$

$S = \{5\}$

1: $\infty$
2: 4
3: 5
4: 3
6: 8
7: 8
Prim’s Algorithm

Add connected vertex to $S$

$X : \begin{array}{c}
1: \infty \\
2: 4 \\
3: 5 \\
4: 3 \\
5: \{4,5\} \\
6: 8 \\
7: 8
\end{array}$

$S = \{5\}$

$S = \{4,5\}$
Prims’s Algorithm

while $H$ is not empty:
  $v = \text{deletemin}(H)$
  for each $\{v, z\} \in E$:
    if $\text{cost}(z) > w(v, z)$:
      $\text{cost}(z) = w(v, z)$
      $\text{prev}(z) = v$
      $\text{decreasekey}(H, z)$

Now choose shortest path from any node in $S$ – one on front of queue

$X : \{4, 5\}$
$S = \{5\}$

1: 4
2: 4
3: 5
4: 4
5: 3
6: 8
7: 4

$X : \{4, 5\}$
$S = \{4, 5\}$
Prim’s Algorithm

Repeat until $S = V$

$X :$

$S = \{5\}$

$2: 1$

$\{4,5\}$

$3: 5$

$\{1,4\}$

$S = \{4,5\}$

$6: 8$

$\{1,4,5\}$

$7: 4$
Prim’s Algorithm

Repeat until $S = V$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$S$</th>
<th>$V$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:2</td>
<td>{5}</td>
<td>3:2</td>
</tr>
<tr>
<td>6:8</td>
<td>{4,5}</td>
<td>6:8</td>
</tr>
<tr>
<td>7:4</td>
<td>{1,4,5}</td>
<td>7:4</td>
</tr>
<tr>
<td></td>
<td>{1,2,4,5}</td>
<td></td>
</tr>
</tbody>
</table>
Prim’s Algorithm

Repeat until $S = V$

$$X :$$

<table>
<thead>
<tr>
<th>$S$</th>
<th>6: 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>${4,5}$</td>
<td></td>
</tr>
<tr>
<td>${1,4}$</td>
<td>7: 4</td>
</tr>
<tr>
<td>${1,2}$</td>
<td></td>
</tr>
<tr>
<td>${2,3}$</td>
<td></td>
</tr>
<tr>
<td>${1,2,3,4,5}$</td>
<td></td>
</tr>
</tbody>
</table>

$$S = \{5\}$$

$$S = \{4,5\}$$

$$S = \{1,4,5\}$$

$$S = \{1,2,4,5\}$$

$$S = \{1,2,3,4,5\}$$
Repeat until $S = V$

<table>
<thead>
<tr>
<th>$X$</th>
<th>$S$</th>
<th>$S$</th>
<th>$6: 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>4,5</td>
<td>4,5</td>
</tr>
<tr>
<td>2</td>
<td>1,4,5</td>
<td>1,4,5</td>
<td>1,4,5</td>
</tr>
<tr>
<td>3</td>
<td>1,2,4,5</td>
<td>1,2,4,5</td>
<td>1,2,4,5</td>
</tr>
<tr>
<td>4</td>
<td>1,2,3,4,5</td>
<td>1,2,3,4,5</td>
<td>1,2,3,4,5,7</td>
</tr>
<tr>
<td>5</td>
<td>2,3</td>
<td>2,3</td>
<td>2,3</td>
</tr>
<tr>
<td>6</td>
<td>4,7</td>
<td>4,7</td>
<td>4,7</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Prim’s Algorithm

Repeat until $S = V$

$X : \quad S = \{5\}$

$\{4,5\} \quad S = \{4,5\}$

$\{1,4\} \quad S = \{1,4,5\}$

$\{1,2\} \quad S = \{1,2,4,5\}$

$\{2,3\} \quad S = \{1,2,3,4,5\}$

$\{4,7\} \quad S = \{1,2,3,4,5,7\}$

$\{6,7\} \quad S = \{1,2,3,4,5,6,7\}$
Complexity

- **Dense Graph**: $|E| = O(|V|^2)$
  - Kruskal’s: $\Theta(|E| \log |V|) = \Theta(|V|^2 \log |V|)$
  - Prim’s (w/ Array PQ): $\Theta(|V|^2)$
  - Prim’s (w/ Binary Heap PQ): $\Theta(|E| \log |V|) = \Theta(|V|^2 \log |V|)$

- **Sparse Graph**: $|E| = O(|V|)$
  - Kruskal’s: $\Theta(|E| \log |V|) = \Theta(|V| \log |V|)$
  - Prim’s (w/ Array PQ): $\Theta(|V|^2)$
  - Prim’s (w/ Binary Heap PQ): $\Theta(|E| \log |V|) = \Theta(|V| \log |V|)$

- **Punch-lines**?
  - Prefer Prim’s with Array for a dense graph
  - Prefer Kruskal’s or Prim’s with Heap PQ for a sparse graph

---

**Dijkstra’s Complexity**:

| Implementation | $\text{deletemin}$ | $\text{insert/decreasekey}$ | $|V| \times \text{deletemin} + (|V| + |E|) \times \text{insert}$ |
|----------------|---------------------|-----------------------------|-----------------------------------------------------------------|
| Array          | $O(|V|)$            | $O(1)$                     | $O(|V|^2)$                                                      |
| Binary heap    | $O(\log |V|)$       | $O(\log |V|)$               | $O((|V| + |E|) \log |V|)$                                       |
Huffman Encoding

- Used in MP3 audio compression which works as follows
- Digitize the analog signal by sampling at regular intervals
  - Yields real numbers $s_1, s_2, \ldots, s_T$
  - Current high fidelity is 44,100 samples per second
  - Thus a 50 minute symphony would have $T = 50 \cdot 60 \cdot 44,100 \approx 130$ million samples
- Quantize each sample $s_t$ into a finite alphabet $\Gamma$
  - These are called codewords or symbols
  - e.g. Quantize into 16 bit numbers
  - Sufficient that close codewords are indistinguishable to human ear
- Encode the quantized values into binary labels
  - Huffman coding (compression) can give savings in this step
Assume for previous example of 130 million samples, that the alphabet has 4 codewords: A, B, C, D
  - Thus all measurements are quantized into one of 4 values

Encode these into binary
  - A: 00
  - B: 01
  - C: 10
  - D: 11

Total memory would be 260 million bits (2 bits/sample)

Can we do better?
Huffman Encoding

- Consider the frequency (count) of the symbols: $A, B, C, D$
  - $A$: 70 million
  - $B$: 3 million
  - $C$: 20 million
  - $D$: 37 million
- Could use a *variable length encoding* where more common codewords are represented with less bits
  - $A$: 0
  - $B$: 001
  - $C$: 01
  - $D$: 10
- Total number of bits would now be less due to frequency of $A$
- However, how would you distinguish $AC$ from $B$?
Prefix-Free Property

- Prefix-Free Property: Binary representation of codewords such that no codeword can be a prefix of another codeword
- Any full binary tree (all nodes have 0 or two children) with the #leaves equal to the #codewords will always give a valid prefix free encoding
  - Any codeword can arbitrarily be at any leaf
- Encoding below allows us to store our example with 213 Mbits
  - Why? (17% improvement)
- But how do we find the optimal encoding tree?

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>0</td>
</tr>
<tr>
<td>$B$</td>
<td>100</td>
</tr>
<tr>
<td>$C$</td>
<td>101</td>
</tr>
<tr>
<td>$D$</td>
<td>11</td>
</tr>
</tbody>
</table>
Assume frequency (count) of codewords are $f_1, f_2, \ldots, f_{|\Gamma|}$

\[
\text{cost(tree)} = \sum_{i=1}^{|\Gamma|} f_i \cdot \text{depth}_\text{in}_\text{tree}(s_i)
\]

Tree cost = $70 \cdot 1 + 37 \cdot 2 + 3 \cdot 3 + 20 \cdot 3 = 213$

If we keep frequencies at internal nodes which are equal to the sum of their children, then cost(tree) is just the sum of all internal and leaf nodes (excluding the root node)
Huffman Algorithm

Greedy constructive algorithm

- Repeat until all symbols/codewords are used
- Find the two symbols with smallest frequencies, say $i$ and $j$
- Make them children of a new node with frequency $f_i + f_j$
- Pull $f_i$ and $f_j$ off the list of frequencies
- Insert $f_i + f_j$ on the list of frequencies
Although we insert array indexes (integers from 1 to 2n-1) into the queue, the sorted key value is $f(index)$
  - Note the array $f$ is assumed unsorted

Can implement priority queue just like in Dijkstra's algorithm

If we use a binary heap implementation of priority queue, then Huffman complexity is $O(?)$
Although we insert array indexes (integers from 1 to $2n-1$) into the queue, the sorted key value is $f(index)$
- Note the array $f$ is assumed unsorted

- Can implement priority queue just like in Dijkstra's algorithm
- If we use a binary heap implementation of priority queue, then Huffman complexity is $O(n \log n)$
Travelling Salesman Problem

- Given a graph of \( n \) cities with distances from one city to another:
  - Visit each city exactly once, and return to the initial city
  - Travelling the least possible distance
- We will do the directed graph version – can have asymmetric city distances
- TSP is a Hamiltonian Cycle, also called a Rudrata Cycle
  - Cycle through a connected graph which visits each node exactly once and for which the sum of the edge costs is minimized
TSP Complexity

- $n!$ possible paths
- What is the simplistic algorithm?
- To calibrate, there are about $10^{57}$ atoms in the solar system and $10^{80}$ atoms in the current known universe

Approximate TSP Time Complexity – big O

<table>
<thead>
<tr>
<th># of Cities</th>
<th>Brute force $O(n!)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$10^6$</td>
</tr>
<tr>
<td>15</td>
<td>$10^{12}$</td>
</tr>
<tr>
<td>20</td>
<td>$10^{18}$</td>
</tr>
<tr>
<td>50</td>
<td>$10^{64}$</td>
</tr>
<tr>
<td>100</td>
<td>$10^{159}$</td>
</tr>
<tr>
<td>1000</td>
<td>$10^{2567}$</td>
</tr>
</tbody>
</table>
How would Greedy TSP work?
Will this always lead to a legal path?
- In those cases when not, just try again
TSP Final Comparative Project

- Group Project (see syllabus for details), You will implement:
  - Random (just report cost of path using order of initial city generation) – This gives a baseline for initial comparison
  - Greedy approach
  - Branch and Bound approach (Project #5)
  - Your own fast approach

- You can start on all of them right away (except B&B)

- Your own approach can be original or you can use the best algorithm(s) you can find out there

- Your group will give an oral (and written) report about your approach and comparative results of your different algorithms on the last day of class

- Some extra credit for very nice results with own approach