CS 161: Design and Analysis of Algorithms
Greedy Algorithms 1: Shortest Paths In Weighted Graphs

- Greedy Algorithm
- BFS as a greedy algorithm
- Shortest Paths in Weighted Graphs
- Dijsktra’s Algorithm
Greedy Algorithms

• Build up solution, making most obvious decision at each step

• Example: Making change
  – How can I make x cents using the fewest number of coins/bills?
Making Change

• General problem:
  – Given integer coin values $v_1, \ldots, v_n$, and a target amount $W$, find a set of coins (allowing repetition) whose total value is $w$ that minimizes the total number of coins

• Greedy Algorithm:
  – If $x = 0$, give nothing and stop
  – Otherwise, find the largest $v_i \leq W$, add $v_1$ to the set, and subtract $v_1$ from $W$. Repeat
Making Change

• For U.S. currency, we have $v_1 = 1$, $v_2 = 5$, $v_3 = 10$, $v_4 = 25$ (ignore higher values)
• Does greedy algorithm give best solution?
Greedy Algorithm Optimal?

• Claim: greedy is optimal for U.S. coins
• Equivalent to showing the following: For any amount \( W \), there is an optimal solution using the largest \( v_i \) that is at most \( W \)
Greedy Algorithm Optimal

• Proof of equivalence:
  – If claim true, then greedy is optimal, and greedy uses largest $v_i$ that is at most $W$
  – Other direction: Assume true for all $W' < W$. Say greedy uses $g$ coins, optimal uses $p$ coins
  – Let $P$ be an optimal solution for $W$ that uses largest $v_i$
  – Greedy first picks $v_i$, then solves $W - v_i$
  – By induction, greedy optimal on $W - v_i$, $g - 1$ coins
  – $P$ without $v_i$ is a solution for $W - v_i$ with $p - 1$ coins
  – Therefore, $p - 1 \geq g - 1$, so $p \geq g$
Greedy Algorithm Optimal?

• Claim: greedy is optimal for U.S. coins
• Proof:
  – Must have at most 2 dimes (otherwise can replace 3 dimes with quarter and nickel)
  – If 2 dimes, no nickels (otherwise can replace 2 dimes and 1 nickel with a quarter)
  – At most 1 nickel (otherwise can replace 2 nickels with a dime)
  – At most 4 pennies (otherwise can replace 5 pennies with a nickel)
Greedy Algorithm Optimal?

• In optimal solutions:
  – Total value of pennies: < 5 cents
  – Total value of pennies and nickels: < 10 cents
  – Total value of non-quarters: < 25 cents

• Therefore we always use the largest coin, so greedy optimal
Making Change

• Is greedy algorithm optimal in general?
  – Say we have 20 cent pieces as well
  – What if \( w = 40 \)?
  – Optimal: two 20 cent pieces (2 coins)
  – Greedy: first picks quarter, then dime, then nickel (3 coins)

• In general, greedy algorithm does not give optimal solution for the Making Change Problem!
Making Change

• What properties of coin values lets greedy be optimal?
• Let \( r_t = \text{Ceiling}(v_{i+1}/v_i) \), \( s_t = r_t v_t \)
• Theorem: If, for each \( t = 1,\ldots,n-1 \), greedy algorithm outputs fewer than \( r_t \) coins for value \( W_t = s_t - v_{t+1} \), then greedy always optimal
Example: U.S. Currency

<table>
<thead>
<tr>
<th>t</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_t$</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>$r_t$</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>N/A</td>
</tr>
<tr>
<td>$s_t$</td>
<td>5</td>
<td>10</td>
<td>30</td>
<td>N/A</td>
</tr>
<tr>
<td>$W_t$</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>N/A</td>
</tr>
<tr>
<td>Greedy($W_t$)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Example: U.S. Currency with 20 Cent Coin

<table>
<thead>
<tr>
<th>t</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_t$</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>$r_t$</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>$s_t$</td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>40</td>
<td>N/A</td>
</tr>
<tr>
<td>$W_t$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>N/A</td>
</tr>
<tr>
<td>Greedy($W_t$)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Greedy Algorithms

• General Goal: give a simple greedy algorithm, prove that it gives optimal solution
• Proving optimality is usually the hard part
BFS as Greedy Algorithm

• Problem: Given a source node \( v \), compute the distances to nodes in graph

• Approach: Set all distances to \( \infty \), update greedily
  
  – Set distance\((u) = \infty \) for \( u \neq v \)
  
  – Set distance\((v) = 0 \)

  – Repeatedly process nodes:
    
    • Find closest node \( u \) that hasn’t been processed
    
    • For each edge \( (u,w) \), update distance to \( w \)
BFS as Greedy Algorithm

• When at a node $u$, to update distance to $w$:
  – $\text{distance}(w) = \text{Min}(\text{distance}(w), \text{distance}(u) + 1)$

• Find closest unprocessed node by keeping queue
  – queue contains unprocessed nodes that have distance $< \infty$
  – Works because we always add nodes to queue in increasing distance, distances never updated once in queue
Why Greedy?

- We update neighbors of closest nodes first
Why Correct?

• Let $d(u)$ be correct distance to $u$
• Claim 1: All nodes have $\text{distance}(u) \geq d(u)$
Proof

• True at beginning. Inductively assume true for first $i-1$ updates
• Let $\text{distance}_i(u)$ be distance at step $i$
• For step $i$,
  \[ \text{distance}_i(w) = \text{Min}(\text{distance}_{i-1}(w), \text{distance}_{i-1}(u) + 1) \]
• By induction,
  \[ d(w) \leq \text{distance}_{i-1}(w) \text{ and } d(u) \leq \text{distance}_{i-1}(u) \]
• $d(w) \leq d(u) + 1 \leq \text{distance}_{i-1}(u) + 1$
• Therefore, $d(w) \leq \text{distance}_i(w)$
Why Correct?

- Let $d(u)$ be correct distance to $u$.
- Claim 1: All nodes have $\text{distance}(u) \geq d(u)$.
- Claim 2: After processing a node $w$, all processed nodes $u$ have $\text{distance}(u) = d(u) \leq d(w)$, all unprocessed nodes $u'$ have $d(u') \geq d(w)$, and for all $u$, $\text{distance}(u)$ is the length of the shortest path from $v$ to $u$ where intermediate nodes are constrained to be processed.
Proof

- True at beginning. Inductively assume true for the first i-1 nodes we process. Now process w
- If distance(w) > d(w), then the shortest path to w goes contains nodes that haven’t been processed
- Let u be the first such node
- All on shortest path to u have been processed
- distance(u) = d(u) \leq d(w) < distance(w)
- Therefore, u would have been processed instead of w
- Would have set distance(w) \leq d(u)+1 = d(w)
Proof

• Say we are processing w and update w’
• Let dist be length of shortest path to w’ through processed nodes, u be last node on this path
• If w is on path, u = w (otherwise, since \(d(u) \leq d(w)\), we can bypass w without increasing distance)
  – \(\text{distance}(w’) = \text{distance}(w) + 1\)
• If w is not on path, that distance(w’) was already set correctly
• Thus distance(w’) = Min of these two cases
Why Correct?

• Let $d(u)$ be correct distance to $u$
• Claim 1: All nodes have $\text{distance}(u) \geq d(u)$
• Claim 2: After processing a node $w$, all processed nodes $u$ have $\text{distance}(u) = d(u) \leq d(w)$, all unprocessed nodes $u'$ have $d(u') \geq d(w)$, and for all $u$, $\text{distance}(u)$ is the length of the shortest path from $v$ to $u$ where intermediate nodes are constrained to be processed
• Therefore, once all nodes are processed, distances correct
Weighted Edges
Weighted Edges

• Length of path = sum of weights of edges
• Shortest path = path with shortest length
• Distance from v to u = length of shortest path from v to u
Shortest Path with Positive Integer Weights

• Idea: replace edge of length k with k-1 nodes and k unweighted edges
Shortest Path with Positive Integer Weights

- Idea: replace edge of length $k$ with $k-1$ nodes and $k$ unweighted edges
Running Time?

• Let $W$ be total weight

• $|V'|$:
  - $|V| +$ extra nodes
  - Edge of weight $w$ gets $w-1$ extra nodes
  - Sum over all edges: $W-|E|$
  - $|V'| = |V|-|E|+W$

• $|E'| = W$

• $O(|V'| + |E'|) = O(|V|+W)$
Problem

• Can’t handle non-integer weights
• W can be very large – poor performance
Solution: Dijsktra’s Algorithm

• Recall BFS high level idea:
  – Set distance(u) = ∞ for u ≠ v
  – Set distance(v) = 0
  – Repeatedly process nodes:
    • Find closest node u that hasn’t been processed
    • For each edge (u,w), update distance to w

• Works for general (non-negative) edge weights!
  – update sets
    distance(w)=Min(distance(w),distance(u)+weight(u,w))

• Issue: now distances might be updated multiple times, so can’t use queue
Why Correct?

• Let $d(u)$ be correct distance to $u$
• Claim 1: All nodes have $\text{distance}(u) \geq d(u)$
Proof

• True at beginning. Inductively assume true for first i-1 updates
• Let \(\text{distance}_i(u)\) be distance at step i
• For step i,
  \[\text{distance}_i(w) = \min(\text{distance}_{i-1}(w),\text{distance}_{i-1}(u) + 1)\]
• By induction,
  \[d(w) \leq \text{distance}_{i-1}(w)\] and \[d(u) \leq \text{distance}_{i-1}(u)\]
• \[d(w) \leq d(u) + \text{weight}(u,w) \leq \text{distance}_{i-1}(u) + \text{weight}(u,w)\]
• Therefore, \[d(w) \leq \text{distance}_i(w)\]
Why Correct?

• Let $d(u)$ be correct distance to $u$
• Claim 1: All nodes have $\text{distance}(u) \geq d(u)$
• Claim 2: After processing a node $w$, all processed nodes $u$ have $\text{distance}(u) = d(u) \leq d(w)$, all unprocessed nodes $u'$ have $d(u') \geq d(w)$, and for all $u$, distance($u$) is the length of the shortest path from $v$ to $u$ where intermediate nodes are constrained to be processed
Proof

• True at beginning. Inductively assume true for the first i-1 nodes we process. Now process w
• If \text{distance}(w) > d(w), then the shortest path to w goes contains nodes that haven’t been processed
• Let u be the first such node
• All on shortest path to u have been processed
• distance(u) = d(u) ≤ d(w) < distance(w)
• Therefore, u would have been processed instead of w
• Would have set distance(w) ≤ d(u) + \text{weight}(u,w) = d(w)
Proof

• Say we are processing w and update w’
• Let dist be length of shortest path to w’ through processed nodes, u be last node on this path
• If w is on path, u = w (otherwise, since d(u)≤d(w), we can bypass bypass w without increasing distance)
  – distance(w’) = distance(w)+weight(w,w’)
• If w is not on path, that distance(w’) was already set correctly
• Thus distance(w’) = Min of these two cases
Why Correct?

• Let $d(u)$ be correct distance to $u$
• Claim 1: All nodes have $\text{distance}(u) \geq d(u)$
• Claim 2: After processing a node $w$, all processed nodes $u$ have $\text{distance}(u) = d(u) \leq d(w)$, all unprocessed nodes $u'$ have $d(u') \geq d(w)$, and for all $u$, $\text{distance}(u)$ is the length of the shortest path from $v$ to $u$ where intermediate nodes are constrained to be processed
• Therefore, once all nodes are processed, distances correct
How to Allow Multiple Updates

• Queue no longer works
• Instead, use a heap!
• Heap contains all unprocessed nodes
  – Ordered by distance
• To find closest, deletemin()
• To update:
  – distance(w) =
    Min(distance(w), distance(u) + weight(u, w))
  – decreasekey(w)
Solution: Dijsktra’s Algorithm

• Set $\text{distance}(v) = 0$, $\text{distance}(u) = \infty$ for $u \neq v$
• Create heap $q$ with all nodes, ordered by distance
• While $q$ is not empty:
  – Let $u = q.\text{delete}\text{amin}()$
  – For each edge $(u, w)$ in $E$:
    • $\text{distance}(w) = \text{Min}(\text{distance}(w), \text{distance}(u) + \text{weight}(u, w))$
    • $\text{decrease}\text{key}(w)$
Running Time?

- $|V|$ deletemin operations
- $|V|+|E|$ insert/decreasekey operations ($|V|+2|E|$ in undirected graphs)
- Recall heap operations:
  - deletemin: $O(\log |V|)$
  - insert/decreasekey: $O(\log |V|)$
- Total time: $O( (|V|+|E|)\log |V| )$
Potential Optimization: Lists

• Implement heap operations using a linked list
  – Insert: add beginning of list $O(1)$
  – decreasekey: do nothing $O(1)$
  – deletemin: search whole list for minimum $O(|V|)$

• Total time: $O(|V|^2)$
  – Better for dense graphs $|E|=O(|V|^2)$
  – Worse for sparse graphs
Best of Both Worlds: \(d\)-ary Heaps

- Recall \(d\)-ary heap operations:
  - \(\text{deletemin}: O(d \log |V|/\log d)\)
  - \(\text{decreasekey}: O(\log |V|/\log d)\)
- Total time:
  \[O(|V| \cdot d \log |V|/\log d + (|V| + |E|) \log |V|/\log d)\]
  \[= O( (d \cdot |V| + |E|) \log |V|/\log d)\]
- What \(d\) minimizes this quantity?
Choosing $d$

- Minimize $O((d |V| + |E|) \log |V|/\log d)$
- Difficult to minimize exactly
- Possible to show that $d = O(|E|/|V|)$ is optimal
- Some cases:
  - Sparse: $|E| = O(|V|)$, $d = O(1)$, time = $O(|V| \log |V|)$
    - Same as with binary heaps
  - Dense: $|E| = O(|V|^2)$, $d = O(|V|)$, time = $O(|V|^2)$
    - Same as with linked lists
  - Intermediate: $|E| = O(|V|^{1+c})$, $d = O(|V|^c)$, time = $O(|V|^{1+c})$
    - Linear, better than both binary heaps and linked lists
Even Better: Fibonacci Heap

- Complicated data structure
  - deletemin: $O(\log |V|)$
  - insert/decreasekey: $O(1)$ amortized

- Total running time: $O(|E| + |V|\log |V|)$
  - Only asymptotically better when $|E|$ is close to $|V|$, but not $O(|V|)$
  - Example: $|E| = O(|V|\log |V|)$
Finding Actual Path

• So far, we only compute distance from v to other nodes
• How do we compute the actual shortest path?
• If processing node u causes last change to distance(w), some shortest path to w passes through u
• Keep track of prev(w), the previous node in shortest path
Finding Actual Path

• Set distance(v) = 0, distance(u) = ∞ for u ≠ v
• Create heap q with all nodes, ordered by distance
• While q is not empty:
  – Let u = q.delete_min()
  – For each edge (u,w) in E:
    • If(distance(u) + weight(u,w) < distance(w)):
      – distance(w) = distance(w) + weight(u,w)
      – prev(w) = u
Finding Actual Path

• To find path from v to w, follow prev pointers form w back to v
Negative Edges

• If we have negative edges, Dijkstra’s algorithm fails
Negative Edges

- If we have negative edges, Dijkstra’s algorithm fails
Negative Edges

- If we have negative edges, Dijsktra’s algorithm fails
Negative Edges

• If we have negative edges, Dijkstra’s algorithm fails
Negative Edges

- If we have negative edges, Dijkstra’s algorithm fails
Negative Edges

- If we have negative edges, Dijsktra’s algorithm fails
Negative Edges

• If we have negative edges, Dijkstra’s algorithm fails

• Distance from V to C is 3, not 4!
Why Fail on Negative Edges?

• Let $d(u)$ be correct distance to $u$
• Claim 1: All nodes have $\text{distance}(u) \geq d(u)$
• Claim 2: After processing a node $w$, all processed nodes $u$ have $\text{distance}(u) = d(u) \leq d(w)$, all unprocessed nodes $u'$ have $d(u') \geq d(w)$, and for all $u$, $\text{distance}(u)$ is the length of the shortest path from $v$ to $u$ where intermediate nodes are constrained to be processed
Why Fail on Negative Edges

• True at beginning. Inductively assume true for the first $i-1$ nodes we process. Now process $w$
• If $\text{distance}(w) > d(w)$, then the shortest path to $w$ contains nodes that haven’t been processed
• Let $u$ be the first such node
• All on shortest path to $u$ have been processed
• $\text{distance}(u) = d(u) \leq d(w) < \text{distance}(w)$
• Therefore, $u$ would have been processed instead of $w$
• Would have set $\text{distance}(w) \leq d(u) + \text{weight}(u, w) = d(w)$
Bigger Problem: Negative Cycles

- In the presence of a negative cycle, the shortest path problem is undefined

Path VAC has length 2
Path VABAC has length 1
Path VABABAC has length 0
Path VABABABAC has length -1 ...
Negative Cycles

• If negative cycles, no shortest path
• Possible solution: add constant value to every edge
  – Does not compute shortest paths!
• Possible solution: shortest simple path
  – Finite number of simple paths, so shortest exists
  – Turns out to be very hard in presence of negative cycles