CSCE 411
Design and Analysis of Algorithms

Set 8: Greedy Algorithms
Slides by Prof. Jennifer Welch
Spring 2014
Greedy Algorithm Paradigm

- Characteristics of greedy algorithms:
  - make a sequence of choices
  - each choice is the one that seems best so far, only depends on what's been done so far
  - choice produces a smaller problem to be solved

- In order for greedy heuristic to solve the problem, it must be that the optimal solution to the big problem contains optimal solutions to subproblems
Designing a Greedy Algorithm

- Cast the problem so that we make a greedy (locally optimal) choice and are left with one subproblem
- Prove there is always a (globally) optimal solution to the original problem that makes the greedy choice
- Show that the choice together with an optimal solution to the subproblem gives an optimal solution to the original problem
Some Greedy Algorithms

- fractional knapsack algorithm
- Huffman codes
- Kruskal's MST algorithm
- Prim's MST algorithm
- Dijkstra's SSSP algorithm
- ...

CSCE 411, Spring 2014: Set 8
Knapsack Problem

- There are \( n \) different items in a store
- Item \( i \):
  - weighs \( w_i \) pounds
  - worth $\( v_i \)
- A thief breaks in
- Can carry up to \( W \) pounds in his knapsack
- What should he take to maximize the value of his haul?
0-1 vs. Fractional Knapsack

- **0-1 Knapsack Problem:**
  - the items cannot be divided
  - thief must take entire item or leave it behind

- **Fractional Knapsack Problem:**
  - thief can take partial items
  - for instance, items are liquids or powders
  - solvable with a greedy algorithm...
Greedy Fractional Knapsack Algorithm

- Sort items in decreasing order of value per pound
- While still room in the knapsack (limit of $W$ pounds) do
  - consider next item in sorted list
  - take as much as possible (all there is or as much as will fit)
- $O(n \log n)$ running time (for the sort)
Greedy 0-1 Knapsack Alg?

- 3 items:
  - item 1 weighs 10 lbs, worth $60 ($6/lb)
  - item 2 weighs 20 lbs, worth $100 ($5/lb)
  - item 3 weighs 30 lbs, worth $120 ($4/lb)

- knapsack can hold 50 lbs

- greedy strategy:
  - take item 1
  - take item 2
  - no room for item 3
0-1 Knapsack Problem

- Taking item 1 is a big mistake globally although looks good locally
- Use dynamic programming to solve this in pseudo-polynomial time
Finding Optimal Code

- **Input:**
  - data file of characters and
  - number of occurrences of each character

- **Output:**
  - a binary encoding of each character so that the data file can be represented as efficiently as possible
  - "optimal code"
### Huffman Code

- **Idea:** use short codes for more frequent characters and long codes for less frequent

<table>
<thead>
<tr>
<th>char</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>total bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>fixed</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>300</td>
</tr>
<tr>
<td>variable</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
<td>224</td>
</tr>
</tbody>
</table>

- The number of bits used for each character is calculated based on the frequency of the character.
How to Decode?

- With fixed length code, easy:
  - break up into 3's, for instance

- For variable length code, ensure that no character's code is the prefix of another
  - no ambiguity

```
101111110100
  b  d  e  a  a
```
Binary Tree Representation

fixed length code

cost of code is sum, over all chars c, of number of occurrences of c times depth of c in the tree
Binary Tree Representation

variable length code

cost of code is sum, over all chars c, of number of occurrences of c times depth of c in the tree
Algorithm to Construct Tree Representing Huffman Code

- Given set $C$ of $n$ chars, $c$ occurs $f[c]$ times
- insert each $c$ into priority queue $Q$ using $f[c]$ as key
- for $i := 1$ to $n-1$ do
  - $x := \text{extract-min}(Q)$
  - $y := \text{extract-min}(Q)$
  - make a new node $z$ w/ left child $x$ (label edge 0), right child $y$ (label edge 1), and $f[z] = f[x] + f[y]$
  - insert $z$ into $Q$
<board work>
Given a connected undirected graph with edge weights, find subset of edges that spans all the nodes, creates no cycle, and minimizes sum of weights.
Facts About MSTs

- There can be many spanning trees of a graph.
- In fact, there can be many *minimum* spanning trees of a graph.
- But if every edge has a unique weight, then there is a unique MST.
Uniqueness of MST

Suppose in contradiction there are 2 MSTs, $M_1$ and $M_2$. Let $e$ be edge with minimum weight that is in one MST but not the other (e.g., orange or blue but not both)

- in the example it is the edge with weight 4
- WLOG, assume $e$ is in $M_1$ (e.g., the orange MST)
Uniqueness of MST

If $e$ is added to $M_2$ (e.g., the blue MST), a cycle is formed.

Let $e'$ be an edge in the cycle that is not in $M_1$.
- In the example, the only possibility for $e'$ is the edge with weight 7, since the edges with weights 3 and 5 are in $M_1$ (the orange MST).

By choice of $e$, weight of $e'$ must be $> \text{weight of } e$.
- In the example, $7 > 4$. 

![Diagram showing the uniqueness of MST with edges and weights labeled](image-url)
Uniqueness of MST

- Replacing $e$ with $e'$ in $M_2$ creates a new MST $M_3$ whose weight is less than that of $M_2$
  - In the example, replace edge with weight 7 in blue MST by edge with weight 4

Result is a new spanning tree, $M_3$, whose weight is less than that of $M_2$. **contradiction**
Generic MST Algorithm

- **input:** weighted undirected graph
  \[ G = (V,E,w) \]
- **T := empty set**
- **while** T is not yet a spanning tree of G
  - find **an edge** \( e \) in E s.t. \( T \cup \{e\} \) is a subgraph of some MST of G
  - add \( e \) to T
- return T (as MST of G)
Kruskal's MST algorithm

Consider the edges in increasing order of weight, add in an edge iff it does not cause a cycle.
Kruskal's Algorithm as a Special Case of Generic Algorithm

- Consider edges in increasing order of weight
- Add the next edge iff it doesn't cause a cycle
- At any point, T is a forest (set of trees); eventually T is a single tree
Why is Kruskal's Greedy?

- Algorithm manages a set of edges s.t.
  - these edges are a subset of some MST

- At each iteration:
  - choose an edge so that the MST-subset property remains true
  - subproblem left is to do the same with the remaining edges

- Always try to add cheapest available edge that will not violate the tree property
  - locally optimal choice
Correctness of Kruskal's Alg.

- Let $e_1, e_2, \ldots, e_{n-1}$ be sequence of edges chosen
- Clearly they form a spanning tree
- Suppose it is not minimum weight
- Let $e_i$ be the edge where the algorithm goes wrong
  - $\{e_1, \ldots, e_{i-1}\}$ is part of some MST $M$
  - but $\{e_1, \ldots, e_i\}$ is not part of any MST
Correctness of Kruskal's Alg.

gray edges are part of MST M, which contains $e_1$ to $e_{i-1}$, but not $e_i$

$e_i$, forms a cycle in M

$e^*$: min wt. edge in cycle not in $e_1$ to $e_{i-1}$

$wt(e^*) > wt(e_i)$

replacing $e^*$ w/ $e_i$ forms a spanning tree with smaller weight than M, contradiction!
Note on Correctness Proof

- Argument on previous slide works for case when every edge has a unique weight.
- Algorithm also works when edge weights are not necessarily unique.
- Modify proof on previous slide: contradiction is reached to assumption that $e_i$ is not part of any MST.
Implementing Kruskal's Alg.

- Sort edges by weight
  - efficient algorithms known
- How to test quickly if adding in the next edge would cause a cycle?
  - use disjoint set data structure!
Running Time of Kruskal's Algorithm

- $|V|$ Make-Sets, one per node
- $2|E|$ Find-Sets, two per edge
- $|V| - 1$ Unions, since spanning tree has $|V| - 1$ edges in it
- So sequence of $O(E)$ operations, $|V|$ of which are Make-Sets
- Time for Disjoint Sets ops is $O(E \log^* V)$
- Dominated by time to sort the edges, which is $O(E \log E) = O(E \log V)$. 
Another Greedy MST Alg.

- Kruskal's algorithm maintains a forest that grows until it forms a spanning tree
- Alternative idea is keep just one tree and grow it until it spans all the nodes
  - Prim's algorithm
- At each iteration, choose the minimum weight outgoing edge to add
  - greedy!
Idea of Prim's Algorithm

- Instead of growing the MST as possibly multiple trees that eventually all merge, grow the MST from a single vertex, so that there is only one tree at any point.

- Also a special case of the generic algorithm: at each step, add the minimum weight edge that goes out from the tree constructed so far.
Prim's Algorithm

- input: weighted undirected graph $G = (V,E,w)$
- $T :=$ empty set
- $S := \{\text{any vertex in } V\}$
- while $|T| < |V| - 1$ do
  - let $(u,v)$ be a min wt. outgoing edge ($u$ in $S$, $v$ not in $S$)
  - add $(u,v)$ to $T$
  - add $v$ to $S$
- return $(S,T)$ (as MST of $G$)
Prim's Algorithm Example
Correctness of Prim's Algorithm

Let $T_i$ be the tree represented by $(S,T)$ at the end of iteration $i$.

Show by induction on $i$ that $T_i$ is a subtree of some MST of $G$.

**Basis:** $i = 0$ (before first iteration). $T_0$ contains just a single vertex, and thus is a subtree of every MST of $G$. 
Correctness of Prim's Algorithm

- **Induction:** Assume $T_i$ is a subtree of some MST $M$. We must show $T_{i+1}$ is a subtree of some MST.

- Let $(u,v)$ be the edge added in iteration $i+1$.

**Case 1:** $(u,v)$ is in $M$. Then $T_{i+1}$ is also a subtree of $M$. 
Correctness of Prim's Algorithm

**Case 2:** \((u,v)\) is not in \(M\).

- There is a path \(P\) in \(M\) from \(u\) to \(v\), since \(M\) spans \(G\).
- Let \((x,y)\) be the first edge in \(P\) with one endpoint in \(T_i\) and the other not in \(T_i\).
Correctness of Prim's Algorithm

- Let $M' = M - \{(x,y)\} \cup \{(u,v)\}$
- $M'$ is also a spanning tree of $G$.
- $w(M') = w(M) - w(x,y) + w(u,v) \leq w(M)$ since $(u,v)$ is min wt outgoing edge
- So $M'$ is also an MST and $T_{i+1}$ is a subtree of $M'$
Implementing Prim's Algorithm

- How do we find minimum weight outgoing edge?
- First cut: scan all adjacency lists at each iteration.
- Results in $O(VE)$ time.
- Try to do better.
Implementing Prim's Algorithm

**Idea:** have each vertex not yet in the tree keep track of its best (cheapest) edge to the tree constructed so far.

To find min wt. outgoing edge, find minimum among these values

- use a priority queue to store the best edge info (insert and extract-min operations)
Implementing Prim's Algorithm

- When a vertex \( v \) is added to \( T \), some other vertices might have their best edges affected, but only neighbors of \( v \)
  - add decrease-key operation to the priority queue
Details on Prim's Algorithm

Associate with each vertex v two fields:

- **best-wt[v]**: if v is not yet in the tree, then it holds the min. wt. of all edges from v to a vertex in the tree. Initially infinity.

- **best-node[v]**: if v is not yet in the tree, then it holds the name of the vertex (node) u in the tree s.t. w(v,u) is v's best-wt. Initially nil.
Details on Prim's Algorithm

- **input:** $G = (V,E,w)$
- **// initialization**
- Initialize priority queue $Q$ to contain all vertices, using best-wt values as keys
- Let $v_0$ be any vertex in $V$
- `decrease-key(Q,v_0,0)`
- **// last line means change best-wt[$v_0$] to 0 and adjust Q accordingly**
Details on Prim's Algorithm

while Q is not empty do
  u := extract-min(Q)  // vertex w/ smallest best-wt
  if u is not \( v_0 \) then add (u,best-node[u]) to T
  for each neighbor v of u do
    if v is in Q and \( w(u,v) < \) best-wt[v] then
      best-node[v] := u
      decrease-key(Q,v,w(u,v))

return (V,T)  // as MST of G
Running Time of Prim's Algorithm

Depends on priority queue implementation. Let

- $T_{ins}$ be time for insert
- $T_{dec}$ be time for decrease-key
- $T_{ex}$ be time for extract-min

Then we have

- $|V|$ inserts and one decrease-key in the initialization: $O(V \cdot T_{ins} + T_{dec})$
- $|V|$ iterations of while
  - one extract-min per iteration: $O(V \cdot T_{ex})$ total
Running Time of Prim's Algorithm

- Each iteration of while includes a for loop.
- Number of iterations of for loop varies, depending on how many neighbors the current vertex has.
- Total number of iterations of for loop is \( O(E) \).
- Each iteration of for loop:
  - one decrease key, so \( O(E \cdot T_{\text{dec}}) \) total
Running Time of Prim's Algorithm

- $O(V(T_{ins} + T_{ex}) + E \cdot T_{dec})$

- If priority queue is implemented with a binary heap, then
  - $T_{ins} = T_{ex} = T_{dec} = O(\log V)$
  - total time is $O(E \log V)$

- (Think about how to implement decrease-key in $O(\log V)$ time.)
Shortest Paths in a Graph

- We’ve already seen one single-source shortest path algorithm
  - Bellman-Ford algorithm
  - based on dynamic programming
- Now let’s review a greedy one:
  - Dijkstra’s algorithm
Dijkstra's SSSP Algorithm

- Assumes all edge weights are nonnegative
- Similar to Prim's MST algorithm
- Start with source vertex s and iteratively construct a tree rooted at s
- Each vertex keeps track of tree vertex that provides cheapest path \textbf{from s} (not just cheapest path from any tree vertex)
- At each iteration, include the vertex whose cheapest path from s is the overall cheapest
Prim's vs. Dijkstra's

**Prim's MST**

- s
- 6
- 5
- 4
- 1

**Dijkstra's SSSP**

- s
- 6
- 5
- 4
- 1
Implementing Dijkstra's Alg.

- How can each vertex u keep track of its best path from s?
- Keep an estimate, d[u], of shortest path distance from s to u
- Use d as a key in a priority queue
- When u is added to the tree, check each of u's neighbors v to see if u provides v with a cheaper path from s:
  - compare d[v] to d[u] + w(u,v)
Dijkstra's Algorithm

- input: \( G = (V,E,w) \) and source vertex \( s \)

// initialization

- \( d[s] := 0 \)
- \( d[v] := \text{infinity} \) for all other vertices \( v \)
- initialize priority queue \( Q \) to contain all vertices using \( d \) values as keys
Dijkstra's Algorithm

while Q is not empty do
  u := extract-min(Q)
  for each neighbor v of u do
    if d[u] + w(u, v) < d[v] then
      d[v] := d[u] + w(u, v)
      decrease-key(Q, v, d[v])
      parent(v) := u
Dijkstra's Algorithm Example

source is vertex a

<table>
<thead>
<tr>
<th>iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>Ø</td>
</tr>
<tr>
<td>d[a]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d[b]</td>
<td>∞</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d[c]</td>
<td>∞</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>d[d]</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>16</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>d[e]</td>
<td>∞</td>
<td>∞</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
Correctness of Dijkstra's Alg.

- Let $T_i$ be the tree constructed after $i$-th iteration of while loop:
  - vertices not in $Q$
  - edges indicated by parent variables
- Show by induction on $i$ that the path in $T_i$ from $s$ to $u$ is a shortest path and has distance $d[u]$, for all $u$ in $T_i$ (i.e., show that $T_i$ is a correct shortest path tree).
- **Basis:** $i = 1$. $s$ is the only vertex in $T_1$ and $d[s] = 0$. 
Correctness of Dijkstra's Alg.

- **Induction:** Assume $T_i$ is a correct shortest path tree. Show that $T_{i+1}$ is a correct shortest path tree.
- Let $u$ be the vertex added in iteration $i$.
- Let $x = \text{parent}(u)$.

Need to show path in $T_{i+1}$ from $s$ to $u$ is a shortest path, and has distance $d[u]$. 
Correctness of Dijkstra's Alg

(a,b) is first edge in P' that leaves $T_i$

$P$, path in $T_{i+1}$ from s to u

$P'$, another path from s to u
Correctness of Dijkstra's Alg

Let $P_1$ be part of $P'$ before $(a,b)$.

Let $P_2$ be part of $P'$ after $(a,b)$.

$w(P') = w(P_1) + w(a,b) + w(P_2)$

$\geq w(P_1) + w(a,b)$ (nonneg wts)

$\geq w(s\rightarrow a \text{ path in } T_i) + w(a,b)$ (inductive hypothesis)

$\geq w(s\rightarrow x \text{ path in } T_i) + w(x,u)$ (alg chose $u$ in iteration $i$ and
d-values are accurate, by inductive hypothesis)

$= w(P)$.

So $P$ is a shortest path, and $d[u]$ is accurate after iteration $i+1$. 
Running Time of Dijkstra's Alg.

- initialization: insert each vertex once
  - $O(V T_{ins})$
- $O(V)$ iterations of while loop
  - one extract-min per iteration => $O(V T_{ex})$
  - for loop inside while loop has variable number of iterations...
- For loop has $O(E)$ iterations total
  - one decrease-key per iteration => $O(E T_{dec})$
- Total is $O(V (T_{ins} + T_{ex}) + E T_{dec})$
Using Different Heap Implementations

- \(O(V(T_{ins} + T_{ex}) + E \cdot T_{dec})\)

- If priority queue is implemented with a binary heap, then
  - \(T_{ins} = T_{ex} = T_{dec} = O(\log V)\)
  - total time is \(O(E \log V)\)

- There are fancier implementations of the priority queue, such as Fibonacci heap:
  - \(T_{ins} = O(1), T_{ex} = O(\log V), T_{dec} = O(1)\) (amortized)
  - total time is \(O(V \log V + E)\)
Using Simpler Heap Implementations

- $O(V(T_{ins} + T_{ex}) + E \cdot T_{dec})$
- If graph is dense, so that $|E| = \Theta(V^2)$, then it doesn't help to make $T_{ins}$ and $T_{ex}$ to be at most $O(V)$. Instead, focus on making $T_{dec}$ be small, say constant.
- Implement priority queue with an unsorted array:
  - $T_{ins} = O(1), T_{ex} = O(V), T_{dec} = O(1)$
  - total is $O(V^2)$