lecture29: Shortest Path Algorithms

Sean Massung

Largely based on slides by Cinda Heeren
CS 225 UIUC

30th July, 2013
Outline

1. Announcements
2. MST Review
3. Dijkstra’s
4. $A^*$
Announcements

- lab_graphs due Thursday, 8/1
- final exam this Friday (8/2), 10:30am-12:30pm in this room
Why?

```java
if(boolean_statement_or_condition())
    return true;
else
    return false;
```
Why?

```cpp
bool ret;

if(boolean_statement_or_condition())
    ret = true;
else
    ret = false;

return ret;
```
Outline

1. Announcements
2. MST Review
3. Dijkstra's
4. A*
Kruskal’s Pseudocode

1. Initialize our output graph, $T = (V', E')$, $V' = V$, $E' = {}$
2. Initialize a disjoint sets structure $S$, where each vertex represents a set (all in own set to begin with)
3. Initialize a priority queue, $P$, holding all the edges in the original graph
4. $e = P.removeMin()$
   - If $e$ connects two vertices from different sets, add $e$ to $E'$, and union the two vertices from $e$ in $S$
   - If $e$ connects two vertices from the same set, do nothing (this would create a cycle)
5. Repeat step 4 until $|E'| = n - 1$
Kruskal’s Analysis

Here’s the outline of Kruskal’s:

- Initialize disjoint sets: $O(m)$
- Initialize priority queue
- Call removeMin $n - 1$ times:
  - Call union if necessary: $\sim O(1)$

<table>
<thead>
<tr>
<th></th>
<th>Binary Heap</th>
<th>Sorted Array</th>
<th>AVL Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>each removeMin</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kruskal’s Analysis II

We call build once and removeMin $n$ times. DisjointSets operations are effectively constant.

Using all this information, we can build the final running time for Kruskal’s based on the following:

$$O(build + n \cdot removeMin)$$

You should be able to plug in running times (and simplify) using any PQ structure. For Kruskal’s, the graph implementation doesn’t matter! This is assuming we start with all the edges. Otherwise, we need to do a traversal to get them. Don’t forget:

$$n - 1 \leq m \leq n^2$$
Prim’s Pseudocode

1. Initialize a priority queue $P$, to hold vertices based on a “cost” (initially all $\infty$); change a start vertex’s cost to 0
2. Initialize an empty dictionary $D$: vertex $\rightarrow$ parent
3. $v = P.removeMin()$, label $v$ as visited
   $\forall w \in adjacent(v)$:
   - if $w$ is unvisited and $cost(v, w) < P[w]$
     - $P.decreaseKey(w, cost(v, w))$
     - $D[w] = v$
4. Repeat step 3 for $n$ times (until $P$ is empty), then create $T$ by using the parents in $D$
Prim’s Analysis

Here’s an outline of Prim’s (assuming $D$ is a hash table):

- Initialize priority queue
- Call `removeMin` $n$ times:
  - Call `getAdjacent`
  - Call `decreaseKey` if necessary

<table>
<thead>
<tr>
<th></th>
<th>Binary Heap</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td></td>
<td>$O(n)$</td>
</tr>
<tr>
<td>each <code>removeMin</code></td>
<td></td>
<td>$O(\log n)^*$</td>
</tr>
<tr>
<td>each <code>decreaseKey</code></td>
<td></td>
<td>$O(1)^*$</td>
</tr>
</tbody>
</table>

But how many times do we call `decreaseKey`?

$$\sum_{v \in V} \deg(v) = 2m = O(m)$$
We call build once, removeMin \( n \) times, getAdjacent \( n \) times, and decreaseKey \( 2m \) times.

Using all this information, we can build the final running time for Prim’s based on the following:

\[
O(build + n \cdot (removeMin + getAdjacent) + m \cdot decreaseKey)
\]

You should be able to plug in running times (and simplify) using any PQ structure and any graph implementation! Also remember for connected graphs:

\[
n - 1 \leq m \leq n^2
\]
Outline

1. Announcements
2. MST Review
3. Dijkstra’s
4. A*
Single Source Shortest Path

- Input: directed graph with non-negative edge weights
- Output: a subgraph consisting of the shortest path (minimum total cost) from the start vertex to every other vertex in the graph
- Dijkstra’s algorithm is a solution to this problem (1956)
- $A^*$ is another solution (1968), and is an extension to Dijkstra’s algorithm
Example graph
Dijkstra’s Pseudocode

1. Initialize a priority queue $P$, to hold vertices based on a “cost” (initially all $\infty$); change a start vertex’s cost to 0
2. Initialize an empty dictionary $D$: vertex $\rightarrow$ parent
3. $v = P.removeMin()$, label $v$ as visited
   $\forall w \in adjacent(v)$:
   - if $w$ is unvisited and $P[v] + cost(v, w) < P[w]$
     - $P.decreaseKey(w, P[v] + cost(v, w))$
     - $D[w] = v$
4. Repeat step 3 for $n$ times (until $P$ is empty), then create $T$ by using the parents in $D$
It looks just like Prim’s!

1. Initialize a priority queue $P$, to hold vertices based on a “cost” (initially all $\infty$); change a start vertex’s cost to 0
2. Initialize an empty dictionary $D$: vertex $\rightarrow$ parent
3. $v = P\.removeMin()$, label $v$ as visited
   $\forall w \in adjacent(v)$:
   - if $w$ is unvisited and $cost(v, w) < P[w]$
     - $P\.decreaseKey(w, cost(v, w))$
     - $D[w] = v$
4. Repeat step 3 for $n$ times (until $P$ is empty), then create $T$ by using the parents in $D$
Running Dijkstra’s
Dijkstra’s Analysis: the same as Prim’s

We call build once, removeMin \( n \) times, getAdjacent \( n \) times, and decreaseKey \( 2m \) times.

Using all this information, we can build the final running time for Dijkstra’s based on the following:

\[
O(build + n \cdot (removeMin + getAdjacent) + m \cdot decreaseKey)
\]

Of course, this just ends up being the same as Prim’s analysis. You should know how to calculate the final running times for a variety of data structure combinations.
Outline

1. Announcements
2. MST Review
3. Dijkstra’s
4. A*
Searching for a particular node

1. Initialize a priority queue $P$, to hold vertices based on a “cost” (initially all $\infty$); change the start vertex’s cost to 0
2. Initialize an empty dictionary $D$: vertex $\rightarrow$ parent
3. $v = P.removeMin()$, label $v$ as visited
   - If $v$ is the node we’re searching for, stop
   - Otherwise, $\forall w \in adjacent(v)$: if $w$ is unvisited and $P[v] + cost(v, w) < P[w]$
     - $P.decreaseKey(w, P[v] + cost(v, w))$
     - $D[w] = v$
4. If $P$ is empty, the destination was not found. Otherwise, go to step 3.
Searching on a grid
PathFinding.js

Compare BFS, Dijkstra’s, and $A^*$:

http://qiao.github.io/PathFinding.js/visual/

Then some elevation map searches (for maps with non-uniform edge weights).
Heuristics

- A heuristic is an approximation or a guess
- In the case of A*, a heuristic is used to guess how far from the a given node the destination is
- It’s only possible to guess for certain types of graphs—specifically, graphs on grids give an easy way to guess a distance:
  1. Manhattan Distance
  2. Euclidean Distance
  3. Chebyshev Distance
- So, instead of only comparing the actual cost so far, a heuristic search algorithm also takes into account the estimated future cost
The heuristics

- **Manhattan Distance**

\[ \text{dist}(p, q) = \sum_{i=0}^{n} |p_i - q_i| \]

- **Euclidean Distance**

\[ \text{dist}(p, q) = \sqrt{\sum_{i=0}^{n} (p_i - q_i)^2} \]

- **Chebyshev Distance**

\[ \max_{i} (|p_i - q_i|) \]
A* Outline

- Use a heuristic distance measurement to explore in the best-looking direction first
- Like Dijkstra’s, store the cost so far at each node, but also include the estimated future cost in this number
- That way, assuming our heuristics are good, we can explore fewer nodes before we reach the target
- This is also known as a “best-first search”
A* Data Structures

- A priority queue $P$, to hold the actual + estimated costs to the goal
- A dictionary $D_A$, to map actual costs to explored nodes
- A dictionary $D_P$, to map nodes to parents (to reconstruct the path)
- A function $\text{cost}(v_i, v_j)$ just as before; these are the edge weights
- A function $h(v_i, v_k)$, which is one of the heuristic distance measures, so that $v_i$ does not have to be adjacent to $v_k$ to estimate the cost
A* Pseudocode

1. Initialize $P$, to hold vertices (initially all $\infty$)
2. $D_A[start] = 0, P.decreaseKey(start, D_A[start] + h(start, goal))$
3. $D_P = \{\}$
4. $v = P.removeMin()$. If $v = goal$, stop
5. $\forall w \in adjacent(v)$:
   - if $D_A[v] + cost(v, w) < D_A[w]$
     - $D_A[w] = D_A[v] + cost(v, w)$
     - $P.decreaseKey(w, D_A[w] + h(w, goal))$
     - $D[w] = v$
6. If $P$ is empty, the destination was not found. Otherwise, go to step 4.
Related problems

- Dijkstra’s is Prim’s with a modified cost function
- BFS is Dijkstra’s on graphs with identical edge weights; in this case, the priority queue turns into a FIFO queue
- Dijkstra’s is $A^*$ without a heuristic
- $A^*$ is BFS with no heuristic and uniform edge weights