Priority queues and Dijkstra’s algorithm

Ashley Montanaro
ashley@cs.bris.ac.uk

Department of Computer Science, University of Bristol
Bristol, UK

29 October 2013

Introduction

- In this lecture we will discuss Dijkstra’s algorithm, a more efficient way of solving the single-source shortest path problem.

- This algorithm requires the input graph to have no negative-weight edges.

- The algorithm is based on the abstract data structure called a priority queue, which can be implemented using a binary heap.

Priority queues

A priority queue $Q$ stores a set of distinct elements. Each element $x$ has an associated key $x.key$.

A priority queue supports the following operations:

- Insert$(x)$: insert the element $x$ into the queue.
- DecreaseKey$(x, k)$: decreases the value of $x$’s key to $k$, where $k \leq x.key$.
- ExtractMin(): removes and returns the element of $Q$ with the smallest key.

(Technically, this is a min-priority queue, as we extract the element with the minimal key each time; max-priority queues are similar.)

Example

Imagine we have a set of people Alice, Bob and Charlie, with initial keys 3, 2 and 1 respectively.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Returns</th>
<th>Queue contents afterwards</th>
</tr>
</thead>
<tbody>
<tr>
<td>(start)</td>
<td></td>
<td>(empty)</td>
</tr>
<tr>
<td>Insert(Alice)</td>
<td></td>
<td>{ (Alice,3) }</td>
</tr>
<tr>
<td>Insert(Charlie)</td>
<td></td>
<td>{ (Alice,3), (Charlie,1) }</td>
</tr>
<tr>
<td>ExtractMin()</td>
<td>Charlie</td>
<td>{ (Alice,3) }</td>
</tr>
<tr>
<td>Insert(Bob)</td>
<td></td>
<td>{ (Alice,3), (Bob,2) }</td>
</tr>
<tr>
<td>DecreaseKey(Alice,1)</td>
<td></td>
<td>{ (Alice,1), (Bob,2) }</td>
</tr>
<tr>
<td>ExtractMin()</td>
<td>Alice</td>
<td>{ (Bob,2) }</td>
</tr>
</tbody>
</table>
Priority queues

Priority queues can be implemented in a number of ways.

- Let $n$ be the maximal number of elements ever stored in the queue; we would like to minimise the complexities of various operations in terms of $n$.
- A simple implementation would be as an unsorted linked list.

Implementing Insert is very efficient: we just prepend the new element, with cost $O(1)$.

However, DecreaseKey and ExtractMin each might require time $\Theta(n)$ to find an element.

These complexities can be improved using a binary heap.

Remainder: Binary heaps

- A binary heap is an “almost complete” binary tree, where every level is full except (possibly) the lowest, which is full from left to right.
- It also satisfies the heap property: each element is less than or equal to each of its children.

A binary heap can be implemented efficiently using an array $A$:

$$
\begin{array}{cccccc}
2 & 3 & 5 & 3 & 4 \\
\end{array}
$$

We can move around the tree using
- $\text{Parent}(i) = \lfloor i/2 \rfloor$,
- $\text{Left}(i) = 2i$,
- $\text{Right}(i) = 2i + 1$.

(NB: the first element in $A$ is $A[1]$)

Building a heap from an array

We can use Heapify repeatedly to build a heap from an arbitrary array $A$.

$$
\begin{align*}
\text{BuildHeap}(A) & \\
1. & \text{heapsize} \leftarrow A.\text{length} \\
2. & \text{for } i = \lfloor A.\text{length}/2 \rfloor \text{ downto } 1 \\
3. & \quad \text{Heapify}(i)
\end{align*}
$$

- If $A.\text{length} = n$, each call to Heapify uses time $O(\log n)$.

- Claim: BuildHeap actually runs in time $O(n)$ (see COMS11600 or CLRS §6.3 for the proof).
Heaps and priority queues
We can use a heap to implement a priority queue.

- We need to modify the array $A$ so that it stores information about the elements in the queue, as well as their keys.
- In practice $A$ would often store pointers to information kept elsewhere.
- Each element $x$ also needs to store its position in the heap (e.g. as an integer $x.i$).

For example, imagine we want to store elements A-F, each with a key. The heap might look like:

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{F} & \text{E} & \text{C} \\
   5 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} & \text{B} & \text{D} \\
   2 & 2 & 3 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{B} \\
   2 \\
\end{array}
\]

\[
\begin{array}{c}
   \text{A} \\
   2 \\
\end{array}
\]
Priority queue operations

**DecreaseKey**(x, k)

1. if \( k > A[x.i].key \)
2. error("new key is larger than current key")
3. \( A[x.i].key \leftarrow k \)
4. while \( x.i > 1 \) and \( A[\text{Parent}(x.i)].key > A[x.i].key \)
5. swap \( A[x.i] \) and \( A[\text{Parent}(x.i)] \)
6. \( x.i \leftarrow \text{Parent}(x.i) \)

Example:

```
  1   2
  B   D
 F  1  A  C
 5  2  3  4
```

**Insert**(x)

1. \( \text{heapsize} \leftarrow \text{heapsize} + 1 \)
2. \( x.i \leftarrow \text{heapsize} \)
3. \( A[\text{heapsize}] \leftarrow x \)
4. \( \text{DecreaseKey}(x, x.key) \)

Example:

```
  1   2
  B   D
 F  2  A  C
 5  2  3  4
```

**Insert**(G, 2)

```
  1   2
  B   D
 F  2  A  C
 5  2  3  4
```
### Priority queue operations

**Insert(x)**

1. `heapsize ← heapsize + 1`
2. `x.i ← heapsize`
3. `A[heapsize] ← x`
4. `DecreaseKey(x, x.key)`

#### Example:

- `Insert(G, 2)`

![Diagram showing Insert(G, 2)](image)

### Priority queue operations

**ExtractMin()**

1. if `heapsize < 1`
2. `error("Heap underflow")`
3. `min ← A[1]`
5. `heapsize ← heapsize − 1`
6. `Heapify(1)`
7. return `min`

#### Example:

- `ExtractMin()`

![Diagram showing ExtractMin()](image)
Priority queue operations

ExtractMin()

1. if heapsize < 1
2. error("Heap underflow")
3. min ← A[1]
5. heapsize ← heapsize − 1
6. Heapify(1)
7. return min

ExtractMin()

1. if heapsize < 1
2. error("Heap underflow")
3. min ← A[1]
5. heapsize ← heapsize − 1
6. Heapify(1)
7. return min

ExtractMin()

1. if heapsize < 1
2. error("Heap underflow")
3. min ← A[1]
5. heapsize ← heapsize − 1
6. Heapify(1)
7. return min

ExtractMin()

1. if heapsize < 1
2. error("Heap underflow")
3. min ← A[1]
5. heapsize ← heapsize − 1
6. Heapify(1)
7. return min
Priority queue operations

What are the time complexities of these operations?

▶ DecreaseKey uses time \( O(\log n) \) as there can be at most \( O(\log n) \) levels in a tree containing \( n \) elements.

▶ So Insert also uses time \( O(\log n) \).

▶ The complexity of ExtractMin is dominated by the complexity of Heapify, which is also \( O(\log n) \).

All of these complexities are actually tight, i.e. there are sequences of operations which need this time complexity (optional exercise . . . ).

Priority queue complexities

So we have the following summary.

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>DecreaseKey</th>
<th>ExtractMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked list</td>
<td>( \Theta(1) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>Binary heap</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>

Can we do better still? This is an area of current research! One structure which achieves better bounds is the Fibonacci heap:

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>DecreaseKey</th>
<th>ExtractMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci heap</td>
<td>( \Theta(1)^* )</td>
<td>( \Theta(1)^* )</td>
<td>( O(\log n)^* )</td>
</tr>
</tbody>
</table>

▶ The stars are because the bounds are amortised – that is, the bound given is the average complexity per operation, obtained by averaging over the entire set of operations performed.

▶ Although the Fibonacci heap offers good theoretical performance, it is a complicated data structure and in practice the constant factors are prohibitive.

Dijkstra’s algorithm

Let \( Q \) be a priority queue.

Dijkstra(G, s)

1. for each vertex \( v \in G \): \( v.d \leftarrow \infty \), \( v.\pi \leftarrow \text{nil} \)
2. \( s.d \leftarrow 0 \)
3. add every vertex in \( G \) to \( Q \)
4. while \( Q \) not empty
5. \( u \leftarrow \text{ExtractMin}(Q) \)
6. for each vertex \( v \) such that \( u \rightarrow v \)
7. \( \text{Relax}(u, v) \)

Here adding vertices to \( Q \) uses Insert and Relax uses DecreaseKey.

Dijkstra’s algorithm

The Bellman-Ford algorithm solves the single-source shortest paths problem in time \( O(VE) \). Can we do better?

▶ Dijkstra’s algorithm achieves a time complexity as low as \( O(E + V \log V) \) but requires the weights in the graph to be non-negative.

▶ The algorithm also illustrates the effect of the choice of data structure on runtime.

▶ It is based on a priority queue. In the queue, we store the vertices whose distances from the source are yet to be settled, keyed on their current distance from the source.
Example

Imagine we want to find shortest paths from vertex A in the following graph:

At the start of the algorithm:

First A is extracted from the queue:

Then B is extracted:

Vertex colours: Blue: current vertex, green: settled vertices.
Example
Then C is extracted:

\[
\begin{array}{c}
\text{A} \\
0, \text{nil} \\
\text{B} \\
1, \text{A} \\
1, 1, \infty, \text{nil} \\
\text{C} \\
0, \text{nil} \\
2, 2, 4, \text{C} \\
3, \text{B} \\
\text{D} \\
2, 1, 4, \text{C} \\
\text{E} \\
2, 5, 4, \text{C} \\
\text{F} \\
\infty, \text{nil} \\
\text{G} \\
8, \text{C} \\
\end{array}
\]

- Vertex colours: Blue: current vertex, green: settled vertices.

Example
Then D is extracted:

\[
\begin{array}{c}
\text{A} \\
0, \text{nil} \\
\text{B} \\
1, \text{A} \\
1, 1, 6, \text{D} \\
\text{C} \\
3, \text{B} \\
\text{D} \\
2, 1, 4, \text{C} \\
\text{E} \\
2, 5, 4, \text{C} \\
\text{F} \\
6, \text{D} \\
\text{G} \\
8, \text{C} \\
\end{array}
\]

- Vertex colours: Blue: current vertex, green: settled vertices.

Example
Then either E or F is extracted (here, assume F):

\[
\begin{array}{c}
\text{A} \\
0, \text{nil} \\
\text{B} \\
1, \text{A} \\
1, 1, 6, \text{D} \\
\text{C} \\
3, \text{B} \\
\text{D} \\
2, 1, 4, \text{C} \\
\text{E} \\
2, 5, 4, \text{C} \\
\text{F} \\
\infty, \text{nil} \\
\text{G} \\
7, \text{F} \\
\end{array}
\]

- Vertex colours: Blue: current vertex, green: settled vertices.

Example
Then E is extracted:

\[
\begin{array}{c}
\text{A} \\
0, \text{nil} \\
\text{B} \\
1, \text{A} \\
1, 1, 6, \text{D} \\
\text{C} \\
3, \text{B} \\
\text{D} \\
2, 1, 4, \text{C} \\
\text{E} \\
2, 5, 4, \text{C} \\
\text{F} \\
\infty, \text{nil} \\
\text{G} \\
7, \text{F} \\
\end{array}
\]

- Vertex colours: Blue: current vertex, green: settled vertices.
Example

Finally, G is extracted and the algorithm is complete:

- So we see that the shortest path from A to G has weight 7.

Proof of correctness

Claim

If G is a weighted, directed graph with non-negative weights, Dijkstra’s algorithm terminates with \( v.d = \delta(s, v) \) for all vertices v.

Proof

- Sufficient to show that, when each vertex v is extracted, \( v.d = \delta(s, v) \).
- Towards a contradiction, let v be the first vertex such that \( v.d \neq \delta(s, v) \) when v is extracted.
- \( v \neq s \) because s is the first vertex extracted and \( s.d = \delta(s, s) = 0 \).
- There must be a path from s to v, because otherwise \( v.d = \delta(s, v) = \infty \).
- So let p be a shortest path from s to v.

Runtime analysis

Dijkstra(G, s)

1. for each vertex \( v \in G \): \( v.d \leftarrow \infty \), \( v.\pi \leftarrow \text{nil} \)
2. \( s.d \leftarrow 0 \)
3. add every vertex in G to Q
4. while Q not empty
5. \( u \leftarrow \text{ExtractMin}(Q) \)
6. for each vertex v such that \( u \rightarrow v \)
7. Relax(u, v)

- Relax is implemented using one call to DecreaseKey.
- So the runtime is \( O(V \cdot T_{\text{Insert}} + V \cdot T_{\text{ExtractMin}} + E \cdot T_{\text{DecreaseKey}}) \).
Runtime analysis

So we have the following complexities.

<table>
<thead>
<tr>
<th></th>
<th>Insert</th>
<th>DecreaseKey</th>
<th>ExtractMin</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary heap</td>
<td>$O(\log V)$</td>
<td>$O(\log V)$</td>
<td>$O(\log V)$</td>
<td>$O(E \log V)$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$\Theta(1)^*$</td>
<td>$\Theta(1)^*$</td>
<td>$O(\log V)^*$</td>
<td>$O(E + V \log V)$</td>
</tr>
</tbody>
</table>

Recall that the complexities for the Fibonacci heap are amortised.

Summary

- Dijkstra’s algorithm gives a more efficient way of solving the single-source shortest path problem than the Bellman-Ford algorithm.
- It requires the input graph to have non-negative weight edges.
- The algorithm uses a priority queue data structure which can be implemented in a number of different ways.
- If implemented using a binary heap, its runtime is $O(E \log V)$; if implemented using a Fibonacci heap, its runtime is $O(E + V \log V)$.
- The latter is smaller for fairly dense graphs (i.e. graphs where $V = \Theta(E)$), but in practice Fibonacci heaps are difficult to implement and have poor constant factors.

Coursework

- The first piece of coursework for this unit consists of two parts: a theory part about dynamic programming (which you will hear about next), and an implementation part about Dijkstra’s algorithm.
- The implementation part requires you to write a program in C to navigate a robot across a ruined city.
- It is worth 30 marks. 5 of the marks are competitive and awarded based on the speed of your algorithm.
- The whole coursework is worth 20% of the total mark for the unit and the deadline is Friday 6 December at 12 noon.
- Details online at https://www.cs.bris.ac.uk/Teaching/Resources/COMS21103/robot/, including test code you can download to check your algorithm against a few examples, view its output and benchmark its speed.

Further Reading

- Introduction to Algorithms
  T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein
  - Chapter 6 – Heaps
  - Chapter 10 – Elementary Data Structures
  - Chapter 19 – Fibonacci Heaps
  - Chapter 24 – Single-Source Shortest Paths
- Algorithms
  S. Dasgupta, C. H. Papadimitriou and U. V. Vazirani
  http://www.cs.ucsd.edu/users/dasgupta/mcgrawhill/
  - Chapter 4, Section 4.4 – Dijkstra’s algorithm
  - Chapter 4, Section 4.5 – Priority queue implementations
- Algorithms lecture notes, University of Illinois
  Jeff Erickson
  http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/
  - Lecture 19 – Single-source shortest paths
Biographical notes

Edsger W. Dijkstra (1930–2002)

- Many other contributions, including to distributed computing, programming language design and formal verification.
- Winner of the Turing Award in 1972.
- Also famous for his letter “Go To Statement Considered Harmful”, which marks the start of structured programming.
- Initially found it hard to get his shortest-path algorithm published...

Dijkstra quotes

- “What's the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path.”
- “The intellectual challenge of programming was greater than the intellectual challenge of theoretical physics, and as a result I chose programming.”
- “The quality of programmers is a decreasing function of the density of go to statements in the programs they produce.”
- “Computer science is no more about computers than astronomy is about telescopes.” (attr.)