ALGORITHMS - DIJKSTRA WITH HEAPS

Lent 2010 FRWMM

In lectures, it was stated that the runtime of Dijkstra's algorithm could be "improved" from $O(n^2)$ to $O((n+m)\log n)$ (noting that this is actually worse if the graph is dense). Here we show how to do this.

1 Priority Queues

The new data structure we will need is the **min-priority queue**. Informally, this is a gadget that we keep throwing new objects into, and at any point we can ask it to spit out the smallest one. By "smallest" we mean that every object comes with a **key** k, and these keys can be compared to determine the objects' size ordering.

More formally, we expect the gadget to support the following operations:–

- EmptyQ() – returns an empty Queue
- Insert(Q, x, k) – inserts object x with key k into Q
- ExtractMin(Q) – returns the minimal object in Q and removes it from Q
- DecreaseKey(Q, x, k) – adjusts object x's key to k, which is smaller than its previous value

It's not too hard to think of a data structure that does the job; for instance, a list of (x, k) pairs sorted by k. Then the operations Insert and DecreaseKey run in $O(n)$ time, where n is the number of objects in the queue. This may be good enough for some things, but for our purposes we want all these operations to run in $O(\log n)$ time. So we must find another data structure.

2 Binary Heaps

2.1 Definition

A **binary tree** is a rooted tree where each node has at most two children. An **almost complete** binary tree is one where depths of leaves differ by at most one, and the deepest leaves are all on the left. (Hence there is a uniquely shaped binary tree of each size.)

A **min binary heap** is an almost complete binary tree with keys and objects stored at the nodes, such that a node's key is greater than or equal to its parent's.

The name "heap" is appropriate: we have described a sort of spreading mound of objects with the smallest ones at the top. Note also that the maximum depth of the heap is $O(\log_2 n)$, so provided our operations just move "up and down" the tree, they should run in $O(\log n)$ time as required.

2.2 Representing Binary Heaps

In practice, we represent a binary heap by numbering the nodes $1..n$ in a top-down, left-to-right fashion, and thereby storing the nodes in an array. Then the following operations allow us to navigate the tree:–

$$\begin{aligned} \text{Parent}(m) &= \left\lfloor \frac{m}{2} \right\rfloor \\ \text{LeftChild}(m) &= 2m \\ \text{RightChild}(m) &= 2m+1 \end{aligned}$$

If the left/right child doesn't exist, you get a value greater than n. We assume that the array is always big enough, and that we have the variable n stored somewhere sensible.

2.3 Implementing Priority Queues

It turns out we can indeed use heaps to implement an efficient priority queue. EmptyQ() is trivial:–

```
PROCEDURE EmptyQ():
   n := 0
   RETURN a new array
```

DecreaseKey turns out not to be too bad:–

```
PROCEDURE DecreaseKey(Q, x, k):
   SET x's key to k
   WHILE Parent(x) != 0 AND k < Parent(x)'s key:
      SWAP x and Parent(x)
```

where SWAP means swap keys and objects. You can check that this works; i.e. given that the heap conditions are satisfied beforehand, they will be afterwards as well.

We can now do Insert with a cunning trick. We can just put the new node at the end (i.e. bottom right) of the tree, and run DecreaseKey until it's in the right place.

```
PROCEDURE Insert(Q, x, k):
   n := n+1
   Q[n] := (x, k)
   DecreaseKey(Q, x, k)
```

We do something similar to implement ExtractMin: first we write an IncreaseKey procedure, which is slightly more fiddly than DecreaseKey.

```
PROCEDURE IncreaseKey(Q, x, k):
   SET x's key to k
   WHILE x is not a leaf:
      c := a child of x with minimal key
      SWAP x and c
```

Then, we can do the same trick in reverse for ExtractMin:–

```
PROCEDURE ExtractMin(Q):
   (x, k) := Q[1]
   SWAP Q[1] and Q[n]
   n := n-1
   IncreaseKey(Q, 1, Q[1]'s key)
   RETURN (x, k)
```

Again, you can check these implementations work and run in worst-case $O(\log n)$ time.

3 Dijkstra using Heaps

How can we use this to speed up Dijkstra? The slow bit is when we iterate over all nodes to find a node that minimizes the magic function f_k. So if we keep a queue of all nodes keyed by their value of f_k, we can hopefully do this in $O(\log n)$ time. Of course, we'll have to keep updating the queue as f_k changes, which will take some time as well.

Here goes:–

Algorithm 1 (Fast Dijkstra).

```
Q := EmptyQ()

Insert(Q, node 0, 0)
FOR i from 1 to n-1:
   Insert(Q, node i, infinity)

FOR k from 1 to n:
   (i, d) := ExtractMin(Q)
   D[i] := d
   FOR all edges ij:
      IF d + w(i,j) < j's key
         DecreaseKey(Q, node j, d + w(i,j))
```

Time complexity: we do n ExtractMin's and m DecreaseKey's, so $O((n+m)\log n)$ as claimed.

As an aside, we can actually speed this up further using **Fibonacci heaps**, which can do the DecreaseKey operation in constant time on average. That reduces the runtime to $O(n \log n + m)$. You can actually prove that this is optimal for a Dijkstra implementation: as we visit the nodes in order, you can use Dijkstra to solve the sorting problem. (The details are an exercise.)