Examining Different Implementations of Dijkstra's Shortest Path Algorithm

Colin Furey

MATH 7825

2024

Overview

- Recap of Dijkstra's algorithm for finding shortest paths.
- Examination of Dial's implementation of Dijkstra's algorithm.
- Definition of d-heap data structures and brief look at some of their capabilities.
- Implementation of Dijkstra's algorithm using a d-heap data structure.
- Brief look at how the algorithms perform in practice.

Dijkstra's Algorithm

- Given a directed graph $G(V, E)$ with nonnegative arc lengths and source node s, Dijkstra's algorithm computes the shortest path between the source and all other nodes.
- In the original implementation, this is accomplished by maintaining an array of distance labels $d(i)$ for each node $i \in V$.
- Each iteration partitions V into two sets: S and $\bar{S}$. The former contains nodes whose labels correspond to the shortest path and the latter contains temporarily labeled nodes whose labels are upper bounds on the shortest path.
- On each iteration the node with minimum label in $\bar{S}$ is permanently labeled and we fan out from this node to relabel its neighbors using the update formula

Bottlenecks In Dijkstra's Algorithm

- The first is node selection (red). In order to determine which temporary label is minimum, the algorithm must scan all temporary labels. This operation must be performed n times and so has time complexity

$$n + (n-1) + \cdots + 1 = O(n^2)$$

- The second is distance updates (blue). Each update requires $O(1)$ time and so the time complexity here is

$$m \leq \binom{n}{2} = O(n^2).$$

- Putting everything together, the original implementation is $O(n^2)$.

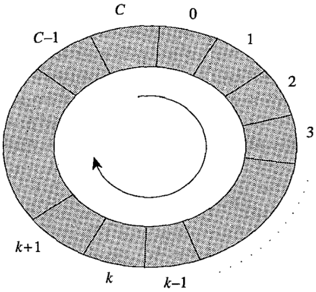
```
algorithm Dijkstra;
begin
    S : = ∅; S̄ : = N;
    d(i) : = ∞ for each node i ∈ N;
    d(s) : = 0 and pred(s) : = 0;
    while |S| < n do
    begin
        let i ∈ S̄ be a node for which d(i) = min{d(j) : j ∈ S̄};
        S : = S ∪ {i};
        S̄ : = S̄ − {i};
        for each (i, j) ∈ A(i) do
            if d(j) > d(i) + c_ij then d(j) : = d(i) + c_ij and pred(j) : = i;
    end;
end;
```

Dialing Down The Complexity: Dial's Implementation

- In effect, the bottleneck in the original implementation is that on each iteration the temporary labels in $\bar{S}$ need to be sorted in order to identify the smallest label and designate it as permanent.
- Dial's implementation of Dijkstra's algorithm addresses this bottleneck by storing temporary distance labels in a sorted fashion, in $nC + 1$ "buckets", thereby bypassing the need to sort all temporary labels at each iteration.
- The distance labels designated as permanent throughout the algorithm form a nondecreasing sequence.
- In the node selection step we need not examine all temporary labels to find the minimum but can instead scan the buckets $0, 1, \ldots$ until we find the first nonempty bucket.

Complexity of Dial's Implementation

- If the kth bucket is the first nonempty bucket, then all nodes in this bucket have the same temporary label k and they're all minimal.
- We can then designate each node in the bucket with the permanent label k and update the temporary labels of all its adjacent nodes.
- At the next iteration, we need only scan buckets $k+1, k+2, \ldots$ since all of the updated labels are at least k.
- Checking whether a bucket is empty or not, deleting a node from a bucket, adding a node to a bucket and distance updates are all $O(1)$ operations.
- It follows that the distance updates require $O(m)$ time and the scanning of the $nC+1$ buckets is $O(nC)$, implying Dial's implementation is $O(m+nC)$.



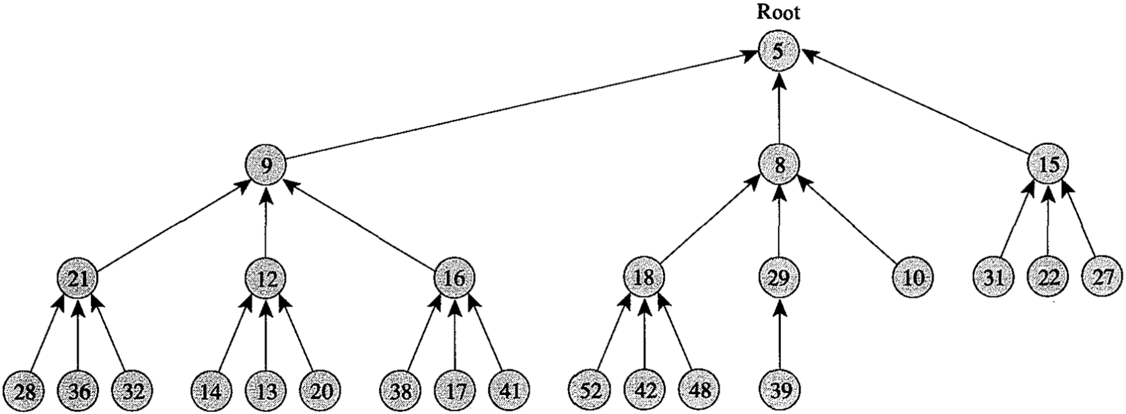
Drawbacks of Dial's Implementation

- Although Dial's implementation can be an improvement in many scenarios, there are a few drawbacks.
- When C is large there are many buckets and the memory footprint of the array of buckets can become prohibitively large.
- This implementation has a pseudopolynomial running time. So if, for example, $C = n^3$ the algortithm runs in $O(n^4)$ time.
- However, in most practical applications C is modest and so the algorithm's running time is often better than its worst case complexity indicates.

Heap Data Structures

- As Dial's implementation shows, improvements in the running time can be achieved via clever data structures for storing of the nodes in $\bar{S}$ and their corresponding temporary distance labels.
- One such example is the heap data structure. These are data structures capable of efficiently storing and manipulating a collection H of objects where each object $i \in H$ has an associated real number key, denoted $key(i)$.
- In most applications of heaps to problems in network flows, the elements of H are the nodes and their corresponding keys are some kind of label associated with a node.
- d-heaps are tree-based data structures where for some $d \geq 2$, each node in the has up to d children. In the context of Dijkstra's algorithm, the elements of H are those nodes with a finite temporary distance label and their corresponding keys are their current label.

d-heap Example



$$\mathrm{pred}(i) = \left\lceil \frac{i-1}{d} \right\rceil, \mathrm{suc}(i) = id - d + 2, ..., id + 1$$

Heap Operations

- Create Heap: Creates an empty heap.
- find-min(i, H): Return the object $i \in H$ of minimum key.
- insert(i, H): add a new object i to H with predefined key.
- delete(i, H): delete object i from the heap.
- decrease-key(i, H): Reduce the key of object $i \in H$ from current value to v. Only defined when current value is greater than v.
- increase-key(i, H): increase the key of object i from current value to v. Only defined when current value is less than v.
- delete-min(i, H): Delete the object i with minimum key.

- Assuming a H has n objects, the operation find-min runs in $O(1)$ time, the insert and decrease-key operations run in $O(log_d n)$ time and the delete, delete-min and increase-key operations run in $O(dlog_d n)$ time.

Pseudocode for Dijkstra's Algorithm Using a d-heap

- Main while loop runs n times as each node needs to be processed at least once. So we perform the find-min operations n times, yielding a time complexity of $O(n)$.
- We need to perform the delete-min operation n times, yielding time complexity $O(nd \log_d n)$.
- We need to perform the operation decrease-key for each neighbor of the current vertex, so the total time complexity for processing all edges is $O(mlog_d n)$.
- We also need to delete perform the delete-min operation n times with time complexity $O(nd \log_d n)$.
- Thus, it follows that the total time complexity of the algorithm is $O(mlog_d n + nd \log_d n)$.

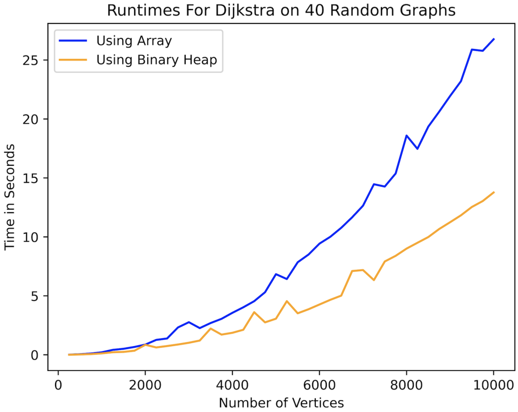
```
algorithm heap-Dijkstra;
begin
    create-heap(H);
    d(j) : = ∞ for all j ∈ N;
    d(s) : = 0 and pred(s) : = 0;
    insert(s, H);
    while H ≠ Ø do
    begin
        find-min(i, H);
        delete-min(i, H);
        for each (i, j) ∈ A(i) do
        begin
            value : = d(i) + c_ij;
            if d(j) > value then
                if d(j) = ∞ then d(j) : = value, pred(j) : = i, and insert (j, H)
                else set d(j) : = value, pred(j) : = i, and decrease-key(value, i, H);
        end;
    end;
end;
```

Complexity of d-Heap Implementation

- To determine the optimal value of d for a given network, we set the two terms in the sum equal and solve for d.
- Doing so implies $d = \min\{2, \lceil \frac{m}{n} \rceil\}$.
- Using this choice of d, the running time reduces to $O(mlog_d n)$.
- For sparse networks where $m = O(n)$, the running time of Dijkstra is reduced to $O(n \log n)$.
- For non-sparse networks where $m = O(n^{1+\epsilon})$ for some $\epsilon > 0$, the running time can be shown to be $O(m)$, which is optimal.

Comparison of Dijkstra Using an array vs Heap

- The most common choice of d for heaps used in Dijkstra's algorithm is $d = 2$, which is referred to as a binary heap.
- Both algorithms were implemented using python, which has a built in module, heapq, for binary heaps.
- Comparisons in the running time were made for 40 random graphs with between 10 and 1000 vertices.



References

- Dijkstra, E.W. A note on two problems in connexion with graphs. Numer. Math. 1, 269–271 (1959)
- R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows: Theory, algorithms, and applications. Prentice Hall Inc., Englewood Cliffs, NJ, 1993.
- Dial, Robert B. Algorithm 360: Shortest-path forest with topological ordering. Communications of the ACM. 12 (11): 632–633 (1969).