Improving the Bottleneck for Dijkstra’s Algorithm

Matt Williamson¹

¹Lane Department of Computer Science and Electrical Engineering
West Virginia University
Outline

1. Introduction
2. Statement of Problem
3. Bottleneck of Dijkstra’s Algorithm
4. Improving Dijkstra’s Algorithm for Few Distinct Edge Lengths
5. Empirical Study
   - Setup
   - Results
6. Ongoing Work
We are concerned with Dijkstra’s Algorithm for the Single Source Shortest Path Problem.

Specifically, we want to identify the bottleneck of the algorithm and explore different methods for improving the bottleneck.

Just like in software, improving the bottleneck of an algorithm can improve the running time, which will also improve the performance when implemented.

We also propose an algorithm that improves the bottleneck even further in the case the graph contains few distinct edge lengths.

We want to see how well all of the approaches perform against each other through experimentation.
Statement of Problem

Definitions

1. We are given a graph \( G = (V, E) \), where \( V \) is the set of \( n \) vertices, and \( E \) is the set of \( m \) edges.
2. Let \( c_{ij} \geq 0 \) be the weight of an edge \((i, j) \in E\).
3. Let \( L = \{l_1, \ldots, l_K\} \) be the set of \( K \) distinct edge lengths.
4. For each vertex \( \nu \in V \), we let \( \text{Adj}(\nu) \) be the set of edges outgoing from \( \nu \).
5. Let \( s \in V \) be the source of the graph.
6. We also let \( \delta(\nu) \) be the length of the shortest path from \( s \) to \( \nu \) in the graph.

Problem to Solve

1. The goal of the Single Source Shortest Path Problem is to find the shortest paths from \( s \) to all of the other reachable vertices in \( G \).
2. For this project, we want to compare all of the approaches for improving the bottleneck of Dijkstra’s Algorithm [3] to see which one performs the best, in terms of execution time.
Dijkstra’s Algorithm

Summary of Algorithm

1. We initialize the set $S$ of permanently labeled vertices to $\emptyset$ and store all of the vertices in a min-priority queue $Q$.
2. Let $d(v) = \infty$, $\forall v \in Q$, except for vertex $s$ which is $d[s] = 0$.
3. Extract the vertex $u$ with the minimum $d(u)$ from $Q$, and add $u$ to $S$.
4. Update the vertices $v \in Adj(v)$.
5. Repeat the process of removing the minimum vertex and updating the adjacent vertices until $Q$ is empty.

Running Time

1. The priority queue has three operations: INSERT, EXTRACTMIN, DECREASEKEY.
2. Inserting all of the vertices in $Q$ requires $n$ INSERT operations.
3. Removing all of the vertices in $Q$ requires $n$ EXTRACTMIN operations.
4. Updating all of the adjacent vertices for each vertex implies each edge is visited exactly once, meaning we require $m$ DECREASEKEY operations.
5. The running time is expressed as $T(n, m) = n \times \text{ExtractMin}() + m \times \text{DecreaseKey}()$.

Therefore, the bottleneck would be how the priority queue is implemented! [1]
### Data Structures

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Each entry in the array corresponds to one of the $n$ vertices in the graph.</td>
</tr>
<tr>
<td>2</td>
<td>For each vertex $v$, we store $d(v)$ in the $v$th position.</td>
</tr>
<tr>
<td>3</td>
<td><strong>DECREASE</strong>\textbf{KEY} takes $O(1)$ time since we update a single vertex.</td>
</tr>
<tr>
<td>4</td>
<td><strong>EXTRACT</strong>\textbf{MIN} takes $O(n)$ time since we have to search the entire array.</td>
</tr>
<tr>
<td>5</td>
<td>This means the running time of Dijkstra’s Algorithm would be $O(n^2 + m) = \mathcal{O}(n^2)$ time.</td>
</tr>
</tbody>
</table>
Each node in the heap corresponds to one of the \( n \) vertices in the graph.

The heap is structured such that we start with a root node, each node has at most two children, and the value of the parent node is less than the values of its children.

\texttt{DECREASEKEY} takes \( O(\log n) \) time since updating a value could force the heap to reorganize.

\texttt{EXTRACTMIN} takes \( O(\log n) \) time since we have to reorganize the heap after removing the root node.

This means the running time of Dijkstra’s Algorithm would be \( O((m + n) \log n) = O(m \log n) \) time.
Improving the Bottleneck for Dijkstra’s Algorithm

Bottleneck of Dijkstra’s Algorithm

Data Structures

Fibonacci Heap

1. The heap is structured as several min-heap, rooted, unordered trees that can be merged when nodes are added and removed.

2. The children of a given parent node are connected using a circular, doubly linked list. This is also true for the roots of each tree.

3. `DECREASEKEY` takes $O(1)$ time since cutting vertices and merging trees are constant time operations.

4. `EXTRACTMIN` takes $O(\log n)$ time since will need to organize and possibly merge some of the trees.

5. This means the running time of Dijkstra’s Algorithm would be $O(m + n \log n)$ time [4].
**Strategy**

1. From [5], we maintain the sets $S$ and $T$, where $S$ is the set of permanently labeled vertices and $T = V - S$ is the set of temporarily labeled vertices.

2. Let $d(j)$ be the distance label of vertex $j$ (i.e. if $j \in S$, then $d(j) = \delta(j)$).

3. Let $L = \{l_1, l_2\}$ be the set of distinct edge lengths. For each item $l_t$ in $L$, we maintain a linked list $E_t(S) = \{(i, j) \in E : i \in S, c_{ij} = l_t\}$.

4. We also let $CurrentEdge(t)$ be the first edge $(i, j)$ in $E_t(S)$ such that $j \in T$. If no such edge exists then $CurrentEdge(t) = \emptyset$. If $CurrentEdge(t) = (i, j)$, then we let $f(t) = d(i) + l_t$. 

**K-Color Algorithm**
Given our new structures, we can find the vertex in $T$ with the minimum distance label by obtaining $\arg\min\{f(t) : 1 \leq t \leq 2\}$.

This gives us a constant time operation when implemented naively, which is an improvement when the number of distinct edge lengths is small.

We also have the subroutine $\text{UPDATE}(t)$ that changes $\text{CurrentEdge}(t)$ either to point to the first edge $(i, j) \in E_t(S)$ where $j \in T$ or sets it to $\emptyset$.

If $\text{CurrentEdge}(t) = (i, j)$, we let $f(t) = d(i) + c_{ij}$. Otherwise, we let $f(t) = \infty$. 
Function **INITIALIZE()**

1: $S := \{s\}; \ T := V - \{s\}$.
2: $d(s) := 0; \ pred(s) := \emptyset$.
3: for (each vertex $v \in T$) do
4: \hspace{0.5cm} $d(v) = \infty; \ pred(v) = \emptyset$.
5: end for
6: for ($t = 1$ to $K$) do
7: \hspace{0.5cm} $E_t(S) := \emptyset$.
8: \hspace{0.5cm} CurrentEdge($t$) := NIL.
9: end for
10: for each edge $(s,j)$ do
11: \hspace{0.5cm} Add $(s,j)$ to the end of the list $E_t(S)$, where $l_t = c_{sj}$.
12: \hspace{0.5cm} if (CurrentEdge($t$) = NIL) then
13: \hspace{0.5cm} \hspace{0.5cm} CurrentEdge($t$) := $(s,j)$
14: \hspace{0.5cm} end if
15: end for
16: for ($t = 1$ to $K$) do
17: \hspace{0.5cm} UPDATE($t$)
18: end for

**Algorithm 5.1:** The Initialization Procedure
**K-Color Algorithm**

```
Function NEW-DIJKSTRA()
1: INITIALIZE()
2: while (T ≠ ∅) do
3:   let r = argmin \{f(t) : 1 ≤ t ≤ K\}.
4:   let (i, j) = CurrentEdge(r).
5:   d(j) := d(i) + lr; pred(j) := i.
6:   S = S ∪ {j}; T := T − {j}.
7:   for (each edge (j, k) ∈ Adj(j)) do
8:     Add the edge to the end of the list E_t(S), where l_t = c_{jk}.
9:     if (CurrentEdge(t) = NIL) then
10:        CurrentEdge(t) := (j, k)
11:   end if
12:  end for
13:  for (t = 1 to K) do
14:    UPDATE(t).
15:  end for
16: end while
```

**Algorithm 5.2:** Dijkstra’s Algorithm with Few Distinct Edge Lengths
Function \texttt{UPDATE}(t)
1: Let \((i, j) = \text{CurrentEdge}(t)\).
2: if \((j \in T)\) then
3: \(f(t) = d(i) + c_{ij}\).
4: return
5: end if
6: while \((j \not\in T)\) \textbf{and} \((\text{CurrentEdge}(t).\text{next} \neq \text{NIL})\) do
7: Let \((i, j) = \text{CurrentEdge}(t).\text{next}\).
8: \(\text{CurrentEdge}(t) = (i, j)\).
9: end while
10: if \((j \in T)\) then
11: \(f(t) = d(i) + c_{ij}\).
12: else
13: Set \(\text{CurrentEdge}(t)\) to \(\emptyset\).
14: \(f(t) = \infty\).
15: end if

\textbf{Algorithm 5.3:} The Update Procedure
Improving Dijkstra’s Algorithm for Few Distinct Edge Lengths

Resource Analysis

Running Time of $\text{FINDMIN}()$

1. The time to find $r = \arg\min\{f(t) : 1 \leq t \leq K\}$ is $O(K)$ time per iteration.
2. Since we run this $n$ times, we get a total time of $O(nK)$.

Running Time of $\text{UPDATE}(t)$

Each time we call $\text{UPDATE}(t)$, one of two things occur.

1. $\text{CurrentEdge}(t)$ does not change. The running time of a single iteration is constant. Since we run $\text{UPDATE}(t)$ $O(n)$ times, we get $O(n)$ time.
2. $\text{CurrentEdge}(t)$ does change. Notice that since each edge is scanned sequentially in our lists, once we scan one edge, we cannot scan it again. In other words, each edge is scanned at most once. This gives us a running time of $O(m)$.

Therefore, the total running time is $O(m + nK)$. 
Outline

1. Introduction
2. Statement of Problem
3. Bottleneck of Dijkstra’s Algorithm
4. Improving Dijkstra’s Algorithm for Few Distinct Edge Lengths
5. Empirical Study
   - Setup
   - Results
6. Ongoing Work
Variables

Table: Experimental Factors

<table>
<thead>
<tr>
<th>Factor</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
<th>Level 5</th>
<th>Level 6</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm</strong></td>
<td>Array</td>
<td>Binary Heap</td>
<td>Fibonacci Heap</td>
<td>K-Color</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Graph Type</strong></td>
<td>Sparse Random</td>
<td>Dense Random</td>
<td>Long Mesh</td>
<td>Square Mesh</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Number of Vertices</strong></td>
<td>100</td>
<td>500</td>
<td>1000</td>
<td>5000</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td><strong>Distinct Edge Lengths</strong></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
</tbody>
</table>

Additional Variables

1. Independent Variables
   - 1. Graph Generators - 9th DIMACS Shortest Path Implementation Challenge [2]
   - 2. Coded in C Language
   - 3. Intel C Compiler
   - 4. Tested on 2.0 GHz 32-bit Intel Core 2 Duo machine with 4GB memory, 2MB cache and running Ubuntu version 8.10.

2. Dependent Variable
   - 1. Execution time for running each algorithm given a specific graph type, graph size, and number of distinct edge lengths
   - 2. We run the respective algorithm five times and compute the mean execution time
Outline

1. Introduction
2. Statement of Problem
3. Bottleneck of Dijkstra’s Algorithm
4. Improving Dijkstra’s Algorithm for Few Distinct Edge Lengths
5. **Empirical Study**
   - Setup
   - Results
6. Ongoing Work
Empirical Study

Results

Table: Various Graph Families Studied

<table>
<thead>
<tr>
<th>ID</th>
<th>Sparse and Mesh Graph Size</th>
<th>Dense Graph Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100 vertices, 400 edges</td>
<td>100 vertices, 9000 edges</td>
</tr>
<tr>
<td>2</td>
<td>500 vertices, 2000 edges</td>
<td>500 vertices, 45000 edges</td>
</tr>
<tr>
<td>3</td>
<td>1000 vertices, 4000 edges</td>
<td>1000 vertices, 90000 edges</td>
</tr>
<tr>
<td>4</td>
<td>5000 vertices, 20000 edges</td>
<td>5000 vertices, 450000 edges</td>
</tr>
<tr>
<td>5</td>
<td>10000 vertices, 40000 edges</td>
<td>10000 vertices, 900000 edges</td>
</tr>
</tbody>
</table>

Figure: Performance of all Dijkstra’s Algorithm implementations for four graph families, as the problem size is varied.
Empirical Study

Results

Empirical Results for Sparse Random Graphs

Figure: Mean performance for sparse random graphs as the values of $n$ and $K$ are varied.
Empirical Results for Dense Random Graphs

Figure: Mean performance for dense random graphs as the values of $n$ and $K$ are varied.
### Empirical Results for Long Mesh Graphs

<table>
<thead>
<tr>
<th>K</th>
<th>Array</th>
<th>Binary Heap</th>
<th>Fibonacci Heap</th>
<th>K-Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value of K, the number of distinct edge lengths</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSSP execution time normalized to BFS time</td>
</tr>
</tbody>
</table>

Figure: Mean performance for long mesh graphs as the values of $n$ and $K$ are varied.
Empirical Results for Square Mesh Graphs

Figure: Mean performance for square mesh graphs as the values of \( n \) and \( K \) are varied.
Open problems

(i) Can we improve the FINDMIN operation beyond the Fibonacci Heap?
(ii) Apply the $K$-Color algorithm for other shortest path problems such as graphs with negative edges and the All Pairs Shortest Paths Problem.
References


