CS 2500: Algorithms

Lecture 22: Greedy Algorithms: Dijkstra's Algorithm and Optimal Merge Pattern

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

October 31, 2024

Problem

- **Task:**
  - A delivery company needs to find the shortest route from a central warehouse in city S to other cities $x_1, x_2, x_3, \ldots, x_n$ in the region.
  - Let d_{ij} be the distance or cost to travel directly from city x_i to city x_j.
- **Objective:** Find the shortest path from the central warehouse (source node) to every other city.
  - **Constraint:** Only direct connections (edges) between cities can be used, with each connection having a non-negative cost.
- Problems like this can be modeled using graphs and solved using **Dijkstra's Algorithm** for shortest paths.

Single Source Shortest Path Problem: Definition

- Consider a directed graph $G = (N, A)$ where:
  - N is the set of nodes.
  - A is the set of directed edges.
- Each edge has a nonnegative length.
- One node is designated as the **source** node.
- **Goal:** Find the length of the shortest path from the source to every other node in the graph.
- We could also interpret the length of an edge as its **cost** and seek the cheapest path from the source to each node.

Single Source Shortest Path: Dijkstra's Algorithm

- Solved by a greedy algorithm called **Dijkstra's algorithm**.
- The algorithm uses two categories of nodes:
  - Nodes with **PERM** status: Nodes for which the shortest path from the source is known.
  - Nodes with **TEMP** status: Nodes whose shortest distance from the source is not yet known.

Initialization and Process

- Initially, the source node has a **PERM** status with a distance of zero.
- All other nodes have a **TEMP** status, as their distances from the source are not yet determined.
- At each step, the algorithm:
  1. Chooses the node with the smallest temporary distance from the source.
  2. Changes this node's status to **PERM**, marking its distance as final.

Updating Distances. For each node with **PERM** status:

- Update the distances of all its neighbors with **TEMP** status.
- For each neighbor v of a node u ($u \to v$):

$$\text{path}[v] = \min(\text{path}[v], \text{path}[u] + \text{weight}(u, v))$$

- If the new path to v is shorter, update the distance and set the predecessor of v to u.

'**Example:** Given the directed graph with edge weights below, find the shortest path from vertex 1 to every other vertex in the graph using Dijkstra's Algorithm.

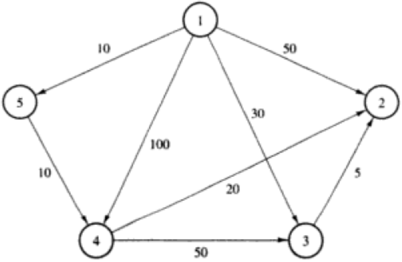


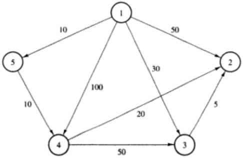
Figure: Example graph for Dijkstra's Algorithm

Initial Setup

For the source node (node 1), initialize:

- `path[1] = 0` (distance to itself)
- All other nodes have `path = infinity` (unreachable initially)

Vertex	Path (Distance)	Predecessor	Status
1	0	NIL	PERM
2	∞	NIL	TEMP
3	∞	NIL	TEMP
4	∞	NIL	TEMP
5	∞	NIL	TEMP



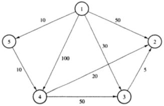
Iteration 1: Selecting Node 1 (Distance = 0)

Current Node: Vertex 1

Neighbors: Vertices 2, 3, 4, and 5

- For $1 \to 2$: $\text{path}[2] = \min(\infty, 0 + 50) = 50$
- For $1 \to 3$: $\text{path}[3] = \min(\infty, 0 + 30) = 30$
- For $1 \to 4$: $\text{path}[4] = \min(\infty, 0 + 100) = 100$
- For $1 \to 5$: $\text{path}[5] = \min(\infty, 0 + 10) = 10$

Vertex	Path (Distance)	Predecessor	Status
1	0	NIL	PERM
2	50	1	TEMP
3	30	1	TEMP
4	100	1	TEMP
5	10	1	TEMP

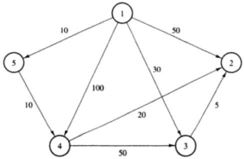


Iteration 2: Selecting Node 5 (Distance = 10)

Current Node: Vertex 5
Neighbor: Vertex 4

- For $5 \to 4$: $\text{path}[4] = \min(100, 10 + 10) = 20$

Vertex	Path (Distance)	Predecessor	Status
1	0	NIL	PERM
2	50	1	TEMP
3	30	1	TEMP
4	20	5	TEMP
5	10	1	PERM



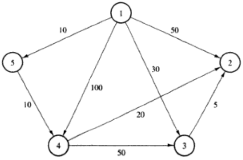
Iteration 3: Selecting Node 4 (Distance = 20)

Current Node: Vertex 4

Neighbors: Vertices 2 and 3

- For $4 \to 2$: $\text{path}[2] = \min(50, 20 + 20) = 40$
- For $4 \to 3$: No update since $\min(30, 20 + 50) = 30$

Vertex	Path (Distance)	Predecessor	Status
1	0	NIL	PERM
2	40	4	TEMP
3	30	1	TEMP
4	20	5	PERM
5	10	1	PERM

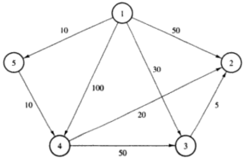


Iteration 4: Selecting Node 3 (Distance = 30)

Current Node: Vertex 3
Neighbor: Vertex 2

- For $3 \to 2$: $\text{path}[2] = \min(40, 30 + 5) = 35$

Vertex	Path (Distance)	Predecessor	Status
1	0	NIL	PERM
2	35	3	TEMP
3	30	1	PERM
4	20	5	PERM
5	10	1	PERM

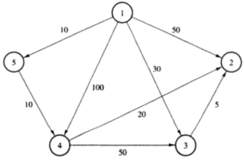


Final Table and Shortest Paths

Vertex	Path (Distance)	Predecessor	Status
1	0	NIL	PERM
2	35	3	PERM
3	30	1	PERM
4	20	5	PERM
5	10	1	PERM

Final Shortest Paths:

- Vertex 2: Path $1 \to 3 \to 2$ with distance 35.
- Vertex 3: Path $1 \to 3$ with distance 30.
- Vertex 4: Path $1 \to 5 \to 4$ with distance 20.
- Vertex 5: Path $1 \to 5$ with distance 10.



Single Source Shortest Path: Dijkstra's Algorithm

Algorithm 1 DIJKSTRA (Graph G, Adjacency Matrix adj, Source Node s)

```
Initialize path[] with infinity for all nodes except the source
Initialize status[] with TEMP for all nodes
Initialize pred[] with NIL for all nodes
Set path[s] = 0 (distance from source to itself is zero)
while True do
    Set current = node with minimum temporary distance from source
    if current == NIL then
        return {All reachable nodes processed; terminate algorithm}
    end if
    Set status[current] = PERM {Mark current node as permanent}
    for each node i in the graph do
        if adj[current][i] != 0 and status[i] == TEMP then
            if path[current] + adj[current][i] < path[i] then
                Update path[i] = path[current] + adj[current][i]
                Set pred[i] = current
            end if
        end if
    end for
end while
```

Dijkstra's Algorithm: Proof of Correctness

Theorem: Dijkstra's algorithm finds the shortest paths from a source node to all other nodes in a graph with nonnegative edge weights.

Goal: Prove by induction that:

- (a) If a node `i` is marked as `PERM`, then `path[i]` is the length of the shortest path from the source `s` to `i`.
- (b) If a node `i` is marked as `TEMP`, then `path[i]` represents the length of the shortest special path from `s` to `i`.

Special path: A path where all intermediate nodes are also marked as `PERM`.

Basis Step

Initially:

- Only the source node `s` is marked `PERM`, and `path[s] = 0`, which represents the shortest path to itself.
- For all other nodes, `path[i]` is set to `infinity`, as no paths are defined yet, aligning with the algorithm's initialization.

Thus, both conditions (a) and (b) hold at the start of the algorithm.

Induction Hypothesis

Assume that:

- Condition (a): For every node currently marked as `PERM`, `path[i]` is the shortest path from `s` to `i`.
- Condition (b): For every node currently marked as `TEMP`, `path[i]` represents the shortest special path from `s` to `i`.

This is true just before a new node `v` is added to the set of `PERM` nodes.

Induction Step for Condition (a)

Objective: Show that when a new node `v` is marked as `PERM`, `path[v]` is the shortest path from `s` to `v`.

- When `v` is chosen, it has the smallest `path` value among all nodes marked as `TEMP`.
- According to the induction hypothesis, `path[v]` is the shortest special path to `v`.

Contradiction Proof:

- Suppose there's a shorter path to `v` passing through a node `x` marked as `TEMP`.
- Since all edge weights are nonnegative, this alternative path cannot be shorter, as `v` was chosen with the smallest `path` value.

This contradiction confirms that `path[v]` is indeed the shortest path.

Induction Step for Condition (b)

Objective: Prove that after adding `v` to `PERM`, `path[i]` for any node `i` still marked `TEMP` represents the shortest special path from `s` to `i`.

- Consider a node `w` still marked `TEMP` after `v` is added to `PERM`.
- Two cases for the shortest special path from `s` to `w`:
  - Path does not pass through `v`: `path[w]` remains unchanged.
  - Path passes through `v`: length is `path[v] + adj[v][w]`.
- The algorithm updates `path[w]` only if `path[v] + adj[v][w]` is shorter, ensuring `path[w]` is the shortest special path.

Thus, condition (b) remains true.

Conclusion of the Proof

When Dijkstra's algorithm completes:

- All nodes are marked as `PERM`, meaning each `path[i]` is the shortest path from `s` to `i`.

Consequently, `path[]` contains the shortest distances from the source to each node.
This concludes the proof that Dijkstra's algorithm correctly finds the shortest paths.

Optimal Merge Patterns

Problem: Suppose we have several sorted files and wish to merge them into one sorted file.

- If we merge two sorted files with n and m records, the result can be obtained in $O(n + m)$ time.
- When merging more than two sorted files, we need a strategy for repeatedly merging files in pairs.

Objective: Determine the optimal way to pairwise merge n sorted files to minimize the total number of comparisons.

Example of Merging Files

Suppose we have four files x_1, x_2, x_3, x_4 with varying sizes.

- We could merge x_1 and x_2 to get a new file y_1.
- Then merge y_1 with x_3 to get y_2.
- Finally, merge y_2 with x_4 to get the final merged file.

Alternative: We could merge x_1 and x_2 to get y_1, then x_3 and x_4 to get y_2, and finally merge y_1 and y_2.
Question: Which sequence of merges results in the fewest comparisons?

Optimal Merge Pattern: Example

Consider files x_1, x_2, x_3 with lengths 30, 20, and 10, respectively.

- Merging x_1 and x_2: 50 comparisons.
- Merging result with x_3: additional 60 comparisons.

Total comparisons $= 110$.

Alternatively:

- Merge x_2 and x_3: 30 comparisons.
- Merge result with x_1: additional 60 comparisons.

Total comparisons $= 90$.

Thus, the second merge pattern is more efficient.

Optimal Merge Pattern: Greedy Strategy

A greedy strategy can be used to determine the optimal merge pattern:

- At each step, merge the two files with the smallest sizes.
- Repeat this process until only one file remains.

This strategy minimizes the total number of comparisons by always selecting the least costly merge at each step.

Optimal Merge Pattern: Greedy Strategy

Example: Suppose we have five files x_1, x_2, x_3, x_4, x_5 with sizes 20, 30, 10, 5, and 30.

Following the greedy strategy:

- Merge x_4 and x_3: 15 comparisons.
- Merge x_1 and x_2: 35 comparisons.
- Merge the result with x_5: 60 comparisons.
- Merge the results to get the final merged file.

Total number of comparisons is minimized with this approach.

Two-Way Merge Pattern

Definition: A two-way merge pattern is one in which each merge step involves the merging of exactly two files.

- This can be represented by a binary merge tree.
- Each leaf node of the tree represents an individual file.
- Each internal node represents a merged result of its two child nodes.

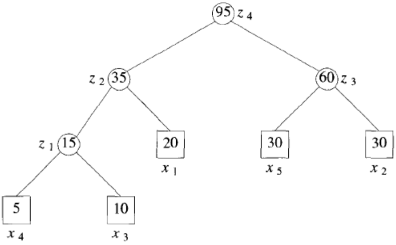


Figure: Binary merge tree representing a merge pattern.

Optimal Merge Pattern with Priority Queue

Objective: Merge multiple sorted files in an optimal way to minimize the total comparison cost.

- When merging more than two sorted files, the optimal pattern is achieved by merging the smallest files first.
- A **priority queue** (min-heap) allows us to efficiently retrieve and merge the smallest files at each step.

Optimal Merge Pattern with Priority Queue

1. Insert all file sizes into a min-priority queue.
2. While more than one file remains in the queue:
    - Remove the two smallest files.
    - Merge them, and add the merge cost to a running total.
    - Insert the merged file back into the priority queue.
3. When only one file remains, the total merge cost represents the minimum comparisons needed.

Optimal Merge Pattern: Algorithm

Algorithm 1 OptimalMerge(files)

```
Initialize a priority queue pq and insert all elements of files into pq.
total_cost ← 0
while pq contains more than one element do
    first ← extract_min(pq) {Extract the smallest file}
    second ← extract_min(pq) {Extract the next smallest file}
    merge_cost ← first + second
    total_cost ← total_cost + merge_cost
    insert(pq, merge_cost) {Insert merged file back into the queue}
end while
return total_cost
```

Optimal Merge Pattern: Algorithm

Example: Suppose we have file sizes `[20, 30, 10, 5, 30]`:

1. Insert files into priority queue: `[5, 20, 10, 30, 30]`
2. Step 1: Merge 5 and 10 (cost $= 15$) $\rightarrow$ Total cost $= 15$
3. Step 2: Merge 15 and 20 (cost $= 35$) $\rightarrow$ Total cost $= 50$
4. Step 3: Merge 30 and 30 (cost $= 60$) $\rightarrow$ Total cost $= 110$
5. Step 4: Merge 35 and 60 (cost $= 95$) $\rightarrow$ Total cost $= 205$

Final total cost $= 205$, which is the minimum merge cost.

Optimal Merge Pattern: Analysis

Time Complexity: $O(n \log n)$

- Inserting each file size into the priority queue takes $O(\log n)$.
- There are $n - 1$ merge operations, each involving $O(\log n)$ for extraction and insertion.
- Thus, the overall time complexity is $O(n \log n)$.

Space Complexity: $O(n)$

- The priority queue stores all file sizes, so space complexity is $O(n)$.