University of Technology Sydney
Department of Computer Science
Faculty of Engineering & Information Technology

ASSIGNMENT COVER SHEET

CS301 — Advanced Algorithms
Problem Set 3: Graph Algorithms
Semester 2, 2026

Student Name	Emily Chen
Student ID	2024-07-156
Program	BSc Computer Science (Honours)
Instructor	Dr. Robert Kim
Due Date	15 July 2026
Word Count	2,450 words

Assignment ID: CS301-PS3-2026-07-156

1 Problem 1: Dijkstra's Algorithm — Runtime Analysis

Given a directed graph $G = (V, E)$ with non-negative edge weights, Dijkstra's algorithm finds the shortest path from a source vertex s to every other vertex. The standard implementation uses a binary min-heap priority queue, achieving worst-case time complexity $O\big((|V| + |E|)\,\log |V|\big)$.

1.1 Setup

All benchmarks were run on an Corex Core i7-13700K with 32 GB RAM under Linux 6.1. Graphs were generated using the Erdős–Rényi model with varying edge probabilities to control density. Each data point represents the median of 50 independent runs; variance across runs was below 2% in all cases.

1.2 Empirical Results

Table 1: Runtime measurements across graph densities and sizes. All times in milliseconds.

Graph Class	$\lvert V\rvert$	$\lvert E\rvert$	Time (ms)	Mem. (MB)
Small sparse	100	198	0.41	0.8
Small dense	100	2 475	1.89	1.4
Medium sparse	1 000	1 998	5.72	6.1
Medium dense	1 000	249 683	148.3	22.7
Large sparse	10 000	19 996	81.40	58.3
Large dense	10 000	24.9 M	18 244	3 210

1.3 Discussion

Three patterns emerge from Table 1:

1. **Sparse graphs** scale near-linearly with $|V|$, consistent with the $O(|V| \log |V|)$ bound when $|E| = \Theta(|V|)$. Doubling $|V|$ roughly doubles the runtime.
2. **Dense graphs** exhibit quadratic behaviour as the $|E| \log |V|$ term dominates. The jump from medium dense to large dense confirms $O(|V|^2 \log |V|)$; a $10\times$ increase in vertices produces a $\sim 125\times$ runtime increase.
3. **Memory** tracks $O(|V| + |E|)$ storage for the adjacency list and distance array, with heap overhead adding roughly $2\times$ on dense inputs.

A Fibonacci heap would reduce the dense-case bound to $O(|E| + |V| \log |V|)$, but its constant factors make binary heaps faster on graphs with fewer than roughly 10^5 vertices in practice.

2 Problem 2: Priority Queue Implementation

The benchmark harness uses the Python implementation shown in Listing 1. The key optimisation is *lazy deletion*: rather than implementing a full `decrease_key` operation, stale heap entries are silently skipped when popped.

Listing 1: Dijkstra's algorithm with binary heap and lazy deletion.

```
import heapq
from collections.abc import Mapping

def dijkstra(graph: Mapping, source: str) -> dict[str, float]:
    """Compute single-source shortest paths via binary heap.

    Args:
```

```
        graph: adjacency map {vertex: [(neighbor, weight), ...]}
        source: starting vertex label

    Returns:
        distance map {vertex: shortest_path_length}
    """
    dist = {v: float('inf') for v in graph}
    dist[source] = 0.0
    pq: list[tuple[float, str]] = [(0.0, source)]

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:           # stale entry -- skip
            continue
        for v, w in graph[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(pq, (nd, v))
    return dist
```

2.1 Correctness

The algorithm maintains the invariant that when vertex u is popped with distance d, that distance is optimal. Proof sketch: all edge weights are non-negative, so any alternative path through an unvisited vertex v (with $d(v) \geq d$) would have length at least d. Lazy deletion on line 21 correctly handles vertices that appear multiple times with superseded distances — only the smallest-distance entry for each vertex is ever processed.

3 Problem 3: Algorithmic Alternatives

For specific graph families, algorithms other than general-purpose Dijkstra offer better constant factors or tighter asymptotic bounds. Table 2 summarises the trade-offs.

Table 2: Shortest-path algorithms compared by graph characteristics.

Algorithm	**Complexity**	**Suitable For**	**Limitation**
BFS	$O(V+E)$	Unweighted graphs	Fails with weights
Dial's algorithm	$O(V+E)$	Small integer weights	Bucket count $\propto$ max weight
0–1 BFS (deque)	$O(V+E)$	Edge weights $\in \{0,1\}$	Binary weights only
Bellman–Ford	$O(VE)$	Negative edges allowed	Slower; detects cycles
A* search	$O((V+E)\log V)$	Goal-directed, heuristic	Needs admissible heuristic
Dijkstra (binary heap)	$O((V+E)\log V)$	General, $w \geq 0$	Baseline choice
Dijkstra (Fibonacci heap)	$O(E + V\log V)$	Dense, $w \geq 0$	High constant factor

For the problem instances in this assignment, the binary-heap Dijkstra implementation in Listing 1 provides the best balance of simplicity, correctness guarantees, and empirical performance. The implementation passes all provided test cases and handles edge cases including disconnected components and zero-weight edges correctly.