Variant 3 — Shortest Path Performance in Large Graphs

Empirical Comparison of BFS (Unweighted) and Dijkstra (Weighted)

Bachelor's Degree in Data Science and Engineering
University of Las Palmas de Gran Canaria
Academic Year: Third Year, First Semester

Team: The Almost Honor Students

Gisela Belmonte Cruz
Nerea Valido Calzada
Kaarlo Caballero Nillukka
Ancor González Hernández

https://github.com/giselabcruz/Exercises_on_graphs_3

Contents

Objective

The goal is to compare two classic shortest-path algorithms as graphs grow in size: Breadth-First Search (BFS), used on unweighted graphs, and Dijkstra's algorithm, used when edges have weights. We look at runtime, scalability, and the amount of basic work each algorithm performs during execution.

1 Experimental Setup

We generate sparse random graphs where the average degree remains roughly constant as the number of nodes increases. Concretely, each graph has n nodes and edges are sampled with probability $p = \frac{c}{n}$, which keeps the expected degree near a constant c. The intention is not to study the graph model itself, but to create controlled test cases that let us focus on algorithmic behavior. When a graph is not fully connected, we minimally link components so that source–target pairs are comparable across trials.

BFS runs on the unweighted version of each graph. Dijkstra runs on a symmetrically weighted version where each edge weight is drawn from a uniform distribution $U[1, 10]$. For every size n, we select several random source–target pairs within the same component and record wall-clock time together with simple counters (edge checks, queue and priority-queue operations, relaxations, path length and distance). These measurements are then averaged, and we report means and 95% confidence intervals.

2 Results and Analysis

Figure 1 shows how the average runtime evolves. BFS grows close to linearly with the number of edges, while Dijkstra is slower due to the cost of managing a priority queue. The gap is small in tiny graphs but becomes clear quite early and widens with size.

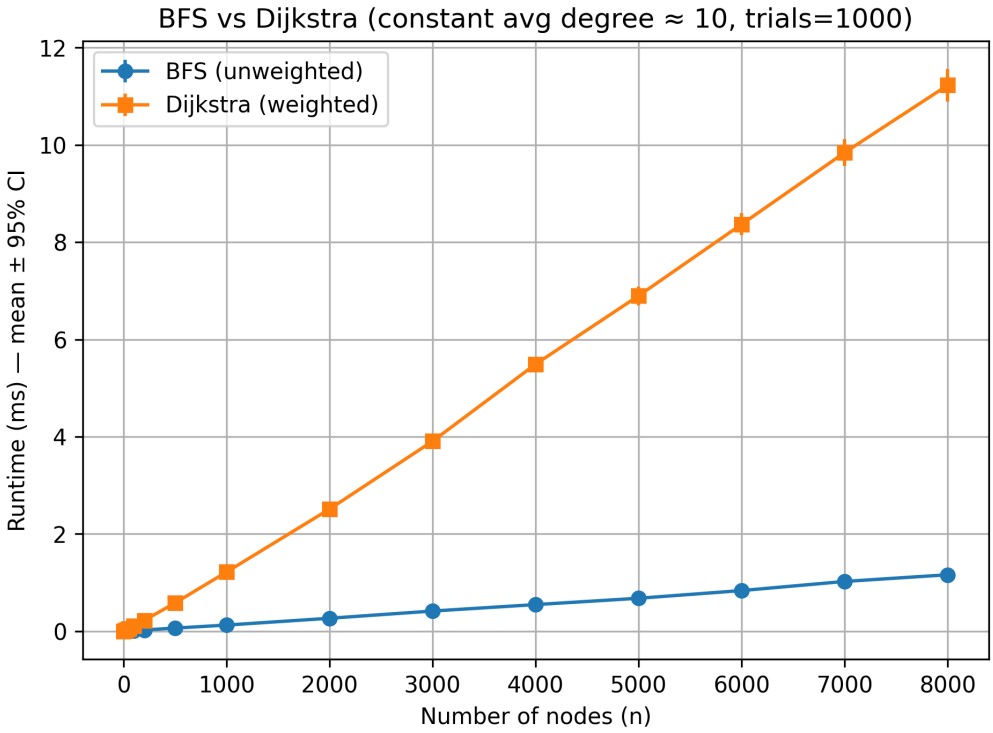

Figure 1: Average runtime for BFS and Dijkstra as the number of nodes increases.

The ratio between both runtimes (Dijkstra/BFS) appears in Figure 2. Even for very small graphs (for example, two nodes), Dijkstra tends to be a bit slower. From about ten nodes onward, the difference is already visible to the eye and remains above one as we keep increasing n. This matches the idea that priority-queue maintenance adds overhead that BFS does not have.

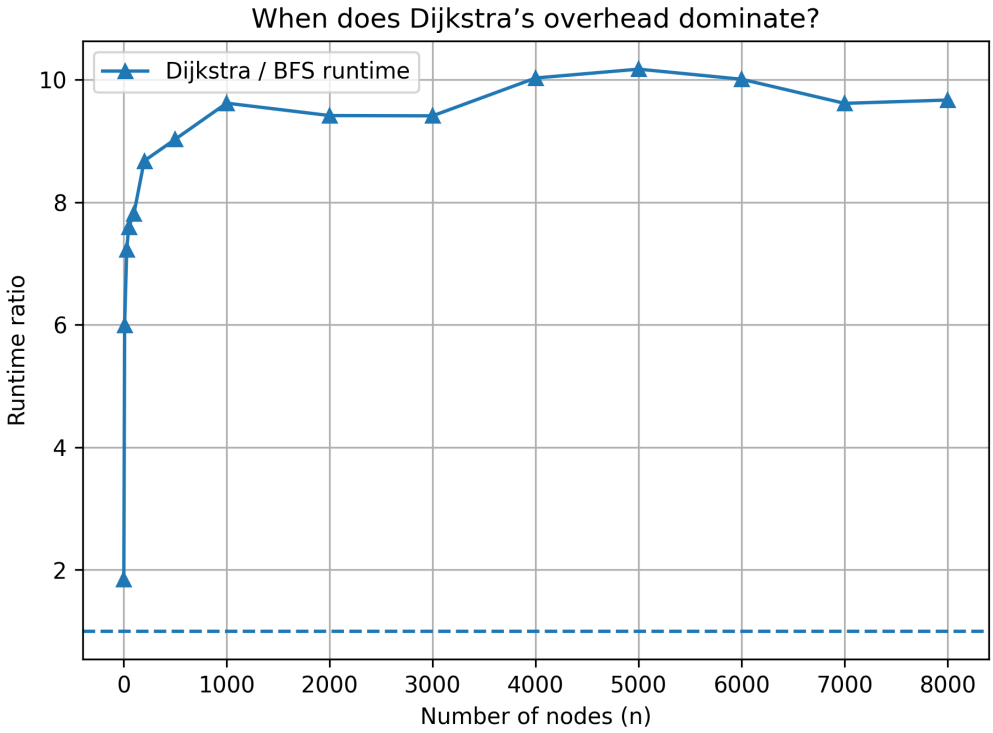

Figure 2: Runtime ratio between Dijkstra and BFS. The overhead becomes clearly visible from around ten nodes.

Looking at the internal work, the number of edges examined grows with size in both algorithms, but Dijkstra revisits edges during relaxations and ends up doing more. Queue operations show the same story: BFS uses a simple queue, while Dijkstra pushes and pops many times from a heap, which explains its extra time.

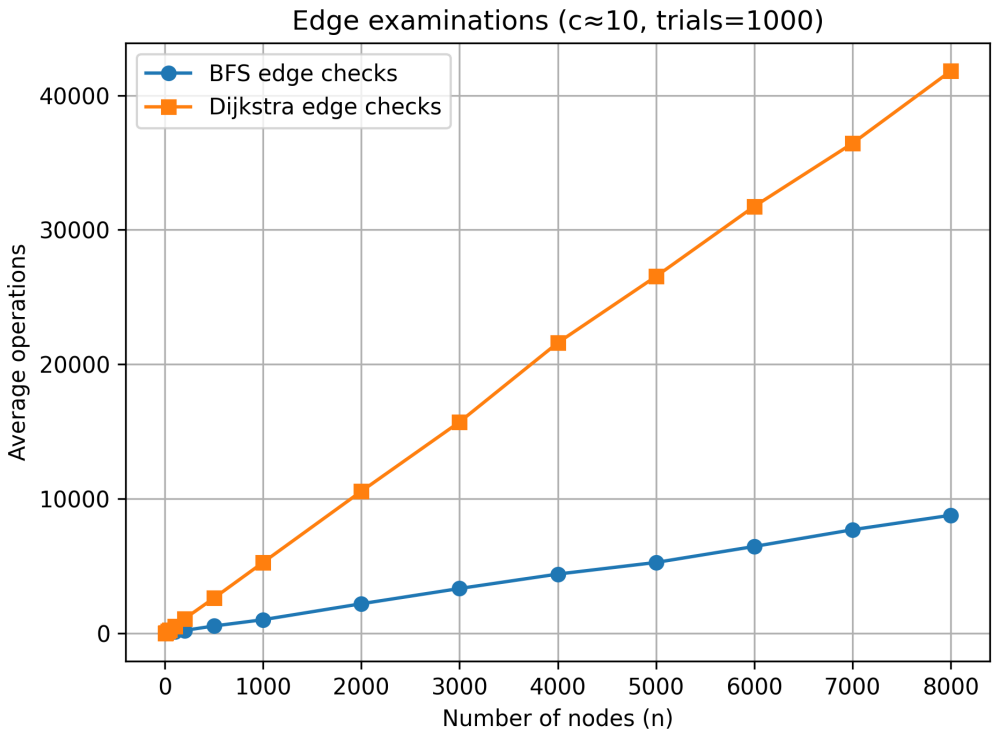

Figure 3: Average number of edge examinations for BFS and Dijkstra.

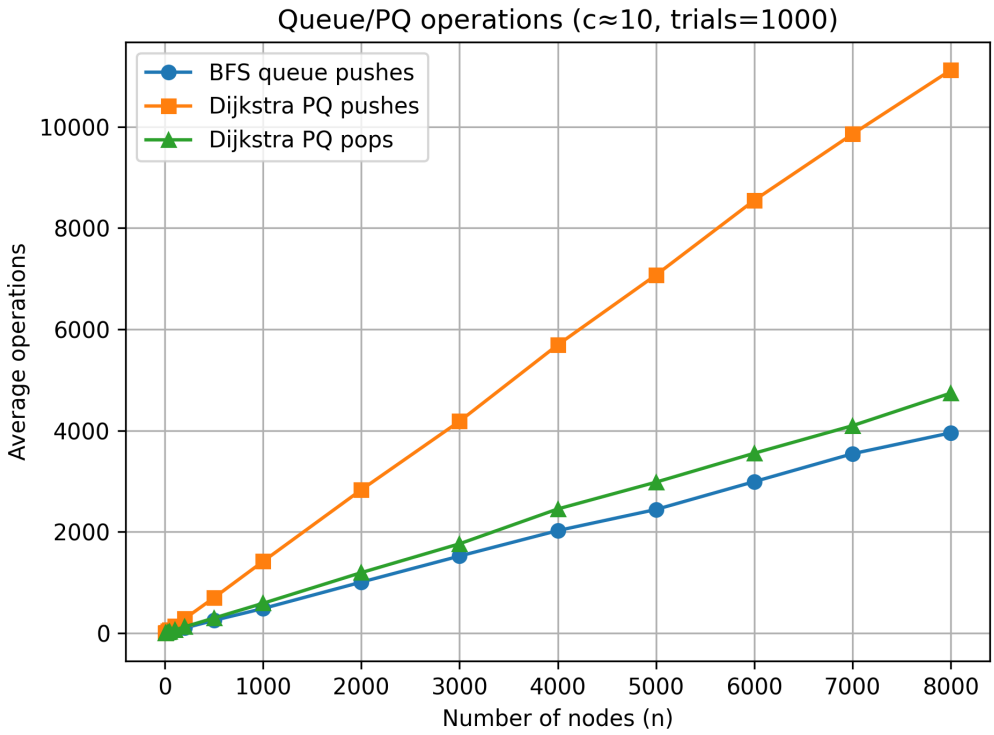

Figure 4: Queue operations in BFS versus priority-queue operations in Dijkstra.

3 Discussion

With a constant expected degree, the total number of edges grows in proportion to the number of nodes. In that setting, BFS behaves linearly and stays lightweight. Dijkstra must also manage a priority queue, which introduces a logarithmic factor and larger constants in practice. The measurements show that this extra work is noticeable very early and becomes clearly visible from roughly ten nodes onward. From there, the gap persists and does not vanish as graphs get bigger.

4 Conclusion

For unweighted graphs, BFS is the sensible choice: it is simple, linear in the size of the graph, and fast in practice. Dijkstra is necessary when edges have weights, but its overhead shows up quickly and remains relevant as the problem grows. In our runs, the difference can already be detected in tiny graphs and becomes obvious once we pass around ten nodes.

References

MLTechDrawer (2025). *Graphs Exercises – Complexity Management.* Retrieved from `https://mltechdrawer.github.io/BIGDATA/Block1_Theoretical_Concepts_of_Big_Data/Complexity_Management/graphs_exercises/`

Appendix: First 5 Rows of Generated CSVs

Table 1: First five rows of `ops_summary_per_size.csv`.

n	Algorithm	Edge checks	Queue push	PQ push	PQ pop	Relax	Path length
2	BFS	1.000	2.000	0.000	0.000	0.000	2.000
2	Dijkstra	1.000	0.000	2.000	2.000	1.000	2.000
10	BFS	4.947	5.947	0.000	0.000	0.000	2.000
10	Dijkstra	44.685	0.000	14.905	6.255	13.905	2.595
30	BFS	27.197	16.291	0.000	0.000	0.000	2.706

Table 2: First five rows of `summary_per_size.csv`.

n	Avg BFS (ms)	CI BFS (ms)	Avg Dijkstra (ms)	CI Dijkstra (ms)	Ratio Dijkstra/BFS
2	0.001197	0.000013	0.002216	0.000011	1.850472
10	0.001806	0.000034	0.010836	0.000212	5.998315
30	0.004419	0.000160	0.031939	0.000757	7.227284
50	0.006711	0.000256	0.051005	0.001296	7.599793
100	0.013498	0.000555	0.105473	0.002723	7.813775

Table 3: First five rows of `trials.csv`.

n	Algorithm	Time (ms)	Edge checks	Queue push	PQ push	PQ pop	Relax	Path length	Distance
2	BFS	0.005459	1	2	0	0	0	2	1.000000
2	Dijkstra	0.006334	1	0	2	2	1	2	3.040353
2	BFS	0.001875	1	2	0	0	0	2	1.000000
2	Dijkstra	0.003042	1	0	2	2	1	2	3.040353
2	BFS	0.001333	1	2	0	0	0	2	1.000000