18YZALG — Tutorial 8 Assignments

Tutorial 8 — Graph Mini-Projects

Choose **one** mini-project per group and present your solution in the tutorial session.

Quick facts

Item	Description
Who	Student groups (recommended 2-3 students)
What	Short live demo (target: about 10-15 minutes per group)
We practice	Graph modelling, BFS, DFS, path reconstruction, Dijkstra, A-star, and small measurements
Grading	Credit to the continuous assessment (tutorial part of the course)

Submission format

- No files need to be submitted beforehand.
- The only deliverable is the **live demonstration**.
- Use slides, a notebook, terminal output, printed examples, or a small visualization - anything is fine as long as the required content is covered.

Live demonstration checklist (structure)

Aim for a concise demo. You can split speaking roles inside the group.

Suggested structure that fits graph mini-projects well:

- story and formal problem statement,
- graph model: vertices, edges, weights, directedness,
- algorithms: baseline and main method,
- path reconstruction or final output,
- correctness and edge cases,
- small measurement and interpretation,
- final recommendation.

Assessment (20 points)

Criterion	Points	What we look for
Graph model and correctness	0-5	Vertices, edges, weights, and edge cases are clear; solution actually works
Algorithm choice and explanation	0-5	The chosen method matches the problem; queue, stack, or heap is explained
Tests and measurement	0-5	Tests are shown; measurement is fair and interpreted honestly
Demo clarity	0-5	Clear structure, readable output, good pacing, and a practical conclusion

Mini-projects

Pick **exactly one**. Send me an email with what your group has picked - first come, first served.

1. City metro stops: shortest route with BFS

Goal. Imagine a small city metro network. Stations are vertices and direct train connections are edges. Given a start station and a target station, output a route with the fewest station-to-station moves. For example, from `A` to `F`, output a path such as `A -> C -> D -> F` and the distance 3.

Required content

- Define the metro graph as unweighted and usually undirected.
- Implement BFS with a queue and a `parent` dictionary or array.
- Reconstruct and print the actual shortest path.
- Include a case where no route exists.
- Include a case where `start == target`.
- Explain why BFS gives the fewest number of edges.

Suggested approaches to compare

- Method A: BFS shortest path with parent reconstruction.
- Method B: DFS that stops when it first finds the target. It may find a path, but not necessarily the shortest one.
- Optional Method C: compare adjacency lists and adjacency matrices on a sparse synthetic metro network.

Demo focus

- Show the queue for a tiny graph by hand or with debug output.
- Show what `parent[v]` means and how the final path is rebuilt.
- Show one example where DFS returns a longer path than BFS.

Stretch goals (optional)

- Add line names and count line changes.
- Add a transfer penalty. This turns the task into a weighted shortest-path problem, so Dijkstra becomes useful.

2. Maze escape: BFS on a grid

Goal. You are given a rectangular maze made of characters. `S` is the start, `E` is the exit, `#` is a wall, and `.` is an open cell. Find the shortest route from `S` to `E` using up, down, left, and right moves.

This is a graph problem even if you do not explicitly build a graph. Every open cell is a vertex. Two cells are connected if they are neighboring open cells.

Required content

- Parse a small maze from a list of strings or from a text file.
- Implement BFS directly on grid cells `(row, column)`.
- Store parents and reconstruct the path.
- Print the maze again with the path marked, for example with `*`.
- Test at least three cases: no path, start next to exit, and a maze with several possible routes.

Suggested approaches to compare

- Method A: BFS shortest path.
- Method B: DFS path search. It can reach the exit, but its route may be longer.
- Optional Method C: BFS from the exit to compute distance-to-exit for all cells.

Demo focus

- Explain the hidden graph: cells are vertices, legal moves are edges.
- Show the path drawn on the maze, not only printed as coordinates.
- Compare path length and number of visited cells for BFS and DFS.

Stretch goals (optional)

- Add multiple exits and stop BFS when the nearest exit is found.
- Add doors and keys as extra state. Then a vertex is not only a cell, but also the set of keys currently held.

3. Delivery costs: cheapest route with Dijkstra

Goal. A delivery company has a road network. Intersections are vertices, roads are edges, and each road has a non-negative cost: distance, time, or fuel. Given a start and a destination, find the cheapest route.

This project is meant to show why BFS is not enough when edges have different weights.

Required content

- Model the graph as a weighted adjacency list: for each vertex store pairs `(neighbor, cost)`.
- Implement Dijkstra with `heapq`.
- Store parents and reconstruct the cheapest path.
- Show a small graph where BFS chooses a path with fewer edges but higher total cost.
- State clearly that Dijkstra requires non-negative edge weights.

Suggested approaches to compare

- Method A: BFS after pretending every road has cost 1. This minimizes the number of edges, not the total cost.
- Method B: Dijkstra with a priority queue.
- Optional Method C: Dijkstra with a simple linear scan instead of a heap.

Demo focus

- Show the heap entries, for example `(current_cost, vertex)`.
- Show one example where the cheapest route uses more edges but lower total cost.
- Count heap pops for different graph sizes.

Stretch goals (optional)

- Add a check that refuses negative edge weights.
- Compare fastest route and shortest-distance route by using different edge weights.

4. Rescue coverage: components and multi-source BFS

Goal. A region contains villages connected by roads. Some villages have rescue stations. You want to answer two questions: which connected regions exist, and for every village, which rescue station is nearest in number of roads?

This project combines connected components with multi-source BFS.

Required content

- Model villages as vertices and roads as undirected unweighted edges.
- Use BFS or DFS to compute connected components.
- Use multi-source BFS: put all rescue stations into the queue at distance `0`.
- For every reachable village, output the nearest station and distance.
- Include a component with no rescue station.
- Define what happens when two stations are tied.

Suggested approaches to compare

- Method A: run BFS separately from every rescue station, then take the best distance for each village.
- Method B: one multi-source BFS from all stations at once.
- Optional Method C: first find components, then report which components are not covered by any station.

Demo focus

- Show that all rescue stations enter the queue at the same time.
- Explain why repeated BFS does the same work many times.
- Show a small table: village, nearest station, distance.

Stretch goals (optional)

- If roads have travel times, replace multi-source BFS with multi-source Dijkstra.
- Add a map-like grid and visualize coverage zones.

5. Robot route with a hint: Dijkstra versus A-star

Goal. A robot moves on a grid from `S` to `G`. Some cells are blocked. All moves have cost `1`, or optionally some cells have terrain costs. Compare Dijkstra with A-star. A-star should usually explore fewer cells because it uses a hint: the Manhattan distance to the goal.

The point is not to create a perfect game AI. The point is to show that A-star is Dijkstra plus a goal-directed priority.

Required content

- Implement Dijkstra on grid cells using a priority queue.
- Implement A-star using priority `f = g + h`, where `g` is the known cost so far and `h` is the Manhattan-distance heuristic.
- Reconstruct and print the path.
- Show that both algorithms return the same optimal path cost on your tests.
- Count how many cells each algorithm pops from the priority queue.
- Explain that the heuristic must not overestimate if you want guaranteed optimal paths.

Suggested approaches to compare

- Method A: Dijkstra, equivalent to A-star with `h = 0`.
- Method B: A-star with Manhattan distance.
- Optional Method C: deliberately use a bad heuristic and show that it can become faster but unsafe.

Demo focus

- Print both path cost and number of popped cells.
- Show a case where A-star explores less of the map than Dijkstra.
- Explain the formula `f = g + h` in one sentence: known cost so far plus optimistic guess to the goal.

Stretch goals (optional)

- Add terrain costs. Use Manhattan distance multiplied by the minimum possible step cost.
- Add diagonal moves and discuss how the heuristic should change.