Application Project Report

Group Members Info: Mertcan İLHAN, Oğuzhan YAZICI, Tunahan İÇÖZ

Project Title: *A vs Dijkstra Shortest Path Simulation for Multi-Goal NPC Navigation**

Submission Date: 29.12.2025

1. Introduction

In many everyday systems — such as games, robotics, and delivery technologies — navigating intelligently through a complex environment is a common challenge. This project focuses on how a non-player character (NPC) can reach multiple target points on a grid while avoiding obstacles and using the shortest possible path.

The application was developed as a visual and interactive simulation for an undergraduate Design and Analysis of Algorithms course. Its primary aim is to compare two widely used pathfinding algorithms, Dijkstra and A*, not only from a theoretical standpoint but also by observing their behavior in practice, especially in scenarios involving multiple goals or dense obstacle placement.

The tool allows users to place a red start point, multiple green goal points, and obstacles on a grid. Once the environment is defined, it visually animates the pathfinding process of both algorithms, allowing users to clearly observe differences in how each algorithm explores the search space. Although both algorithms eventually produce the same optimal path in terms of length, the time taken and number of visited nodes vary significantly.

Overall, the project served both as a technical challenge and a valuable learning opportunity. It helped the team translate abstract algorithmic concepts into a concrete, visual, and interactive experience that supports deeper understanding and engagement.

2. Problem Definition and Objective

2.1 Problem Definition and Motivation

In many real-time systems, an autonomous agent (such as a robot or a non-player character in a game) must move from a starting position to one or several target locations while avoiding obstacles. This is a shortest path problem on a grid or graph. The difficulty increases when:

- the environment contains many obstacles,

- there are multiple goals instead of a single destination,
- and the system is expected to respond quickly (near real-time).

In traditional course settings, algorithms like Dijkstra and A* are taught theoretically on static examples. However, students and practitioners often lack a visual and interactive way to see how these algorithms behave when:

- the number of goals changes (e.g., 1, 5, 10),
- the density of obstacles increases,
- or the search area becomes larger.

The main problem addressed by this project is:

How can we intuitively and experimentally compare Dijkstra and A in complex, multi-goal grid environments, in terms of both correctness and efficiency, rather than only learning them as abstract formulas?*

This is important because understanding the practical behavior of these algorithms directly affects the design of smarter game AI, more efficient robots, and better navigation systems.

2.2 Literature Background and Existing Tools

Dijkstra's algorithm is a classic solution for finding the shortest path from a source node to all other nodes in a weighted graph with non-negative edge costs. A* extends this idea by adding a heuristic function to guide the search, reducing the number of explored nodes while still guaranteeing optimality when the heuristic is admissible.

In the literature and practice:

- **Game engines** such as Unity and Unreal Engine typically use grid-based A* for NPC pathfinding, sometimes with navigation meshes and heuristic optimizations.
- **Graph libraries** like NetworkX or Boost Graph Library provide implementations of Dijkstra and A*, but they usually return only the final path and numeric metrics, without interactive visualization of the search process.
- **Educational visualizers** exist on the web, but most of them:
    - focus on a *single* goal,
    - do not handle multi-goal tours,
    - or do not directly compare A* and Dijkstra on the *same* map under controlled conditions.

What is missing in many of these tools is:

- a multi-goal scenario where the agent must visit all selected targets with a near-optimal or optimal tour,
- a clear side-by-side comparison of A* and Dijkstra in terms of visited nodes, execution time, and path length,
- and a step-by-step visualization of the search process (open set, closed set, current node) that helps students see *how* the algorithms think, not just what result they return.

Our project aims to fill this gap by combining multi-goal navigation, interactive grid editing, and algorithm comparison in one educational application.

2.3 Project Objectives

The specific goals of this project are:

- To implement an interactive grid-based simulation where the user can:
  - place a single start node (NPC),
  - place multiple goal nodes,
  - add or remove obstacles,
  - and adjust grid size.
- To support two pathfinding algorithms:
  - Dijkstra (uninformed search),
  - A* with Manhattan heuristic (informed search),

and run them on the same scenarios for a fair comparison.

- To extend the problem from a single-goal shortest path to a multi-goal tour, where the NPC visits all goals in an order that minimizes the total travel distance (using a brute-force TSP approach for a limited number of goals).
- To record and display, for each algorithm:
  - total path length (number of steps),
  - total number of visited nodes,
  - total execution time in milliseconds,

and use these metrics to evaluate how they behave under different conditions (different numbers of goals, different obstacle densities, different grid sizes).

- To visualize the search process step by step by:
    - showing open set, closed set, and current node,
    - animating the final shortest path for each segment,
    - and finally replaying the full tour from the start position along the optimal route.

By achieving these objectives, the application provides a practical and visual solution to the defined problem: it allows users to experimentally understand how Dijkstra and A* perform in realistic multi-goal navigation scenarios, rather than just seeing them as theoretical algorithms on paper.

3. Tools, Methods and Technologies Used

3.1 Programming Language

Python 3

The application was implemented entirely in Python 3, primarily due to its suitability for rapid prototyping within the limited timeframe of a course project. Python's clean syntax allowed us to implement complex logic like Dijkstra and A algorithms in a readable manner, making the code self-explanatory. Furthermore, the standard library provided essential modules such as heapq for priority queues and itertools for handling permutations, eliminating the need for external dependencies.

3.2 GUI Framework and Visualization

Tkinter (Python's standard GUI library)

Tkinter was used to build the graphical user interface and visualize the grid, NPC, goals, and obstacles. The reasons for choosing Tkinter are:

- It is part of the Python standard library (no extra installation required).
- It is sufficient for simple, 2D, grid-based visualizations and basic animations.
- It supports mouse and keyboard events, allowing the user to:

- left-click to place/remove start, goals, and walls,
- press keys (R, G, W, A, D, Space, C) to control modes and algorithms.

The grid is drawn using Tkinter's Canvas widget, where each cell is a rectangle, the NPC is a red circle, and goals are green circles. Colors are also used to show algorithm states (open set, closed set, current node, final path).

3.3 Python Standard Libraries

The following standard libraries were used:

- **heapq**
  Used to implement the priority queue required by both Dijkstra and A*. The open set of nodes is managed using a min-heap, which provides efficient extraction of the next node with minimum cost.
- **itertools**
  Used for generating permutations of goal nodes in the multi-goal scenario. This supports the brute-force TSP-like search for the best visiting order when the number of goals is small (e.g., 1, 5, 10).
- **time**
  Used to measure execution time (in milliseconds) of each algorithm for each segment. These values are later used to compare A* and Dijkstra in terms of performance.

Optionally, small helper modules such as math or random were used during testing, but the final application mainly relies on the three libraries above.

3.4 Development Tools

- **Code Editor:** Visual Studio Code / PyCharm / any Python-friendly IDE (one was used for coding, debugging, and running the project).
- **Operating System:** Windows (the application was developed and tested on a Windows environment).

No external pathfinding or GUI framework (such as Pygame, NetworkX, or external AI libraries) was used. All algorithm logic was implemented manually to keep the project educational and transparent.

3.5 Development Method

Instead of strictly following formal software engineering methodologies like Scrum, we adopted a practical "bottom-up" approach suitable for our team size and timeline. We knew that trying to build the GUI and the algorithms simultaneously would lead to messy code, so we broke the development into three distinct phases:

- **Phase 1: The Engine First.** We started by implementing the core logic—grid representation and the single-goal A* algorithm—entirely in the console. We needed to be sure the math was correct before worrying about pixels.
- **Phase 2: Visual Integration.** Once the algorithms were finding correct paths in the backend, we introduced Tkinter. This was the most critical step: connecting the abstract algorithm steps (open/closed sets) to visual updates (changing cell colors) without freezing the application interface.
- **Phase 3: Complexity and Polish.** Finally, we added the "Multi-Goal" logic using permutations and refined the user experience. We spent the last few days purely on "breaking" our own code—creating impossible maps and resizing grids to find edge cases, which helped us fix several critical bugs before the final submission.

This workflow allowed us to isolate problems quickly: if the path was wrong, we checked the algorithm; if the animation lagged, we checked the GUI.

4. Application Development Process

This section describes how the application was developed step by step, from initial planning to coding, integration and testing.

4.1 Planning and Requirement Analysis

The development process started with defining what the application should do in practice:

- Represent the environment as a grid with:
    - one start node (NPC),
    - multiple goal nodes,
    - and obstacles (walls).
- Support two algorithms for shortest path finding:
    - Dijkstra,
    - A* with Manhattan heuristic.
- Allow the user to:
    - place start, goals, and walls interactively with the mouse,
    - change the grid size,
    - choose the algorithm (A* or Dijkstra),
    - and run the simulation with a single key.
- Extend the problem from a single-goal path to a multi-goal tour where the NPC visits all selected goals.
- Collect and display performance metrics:
    - total path length,

- number of visited nodes,
- execution time in milliseconds.

These requirements were written down and used as a checklist for the development stages.

4.2 Design of the System

Before coding, the structure of the program was designed around a few key components:

- **Grid Representation:**
  For the system design, we represented the grid as a 2D list where cells are binary-coded: 0 for free space and 1 for obstacles. The start point and goal coordinates are stored separately to allow dynamic updates. The GUI layout relies on Tkinter's Canvas widget, drawing cells as rectangles and agents as colored circles (red for NPC, green for goals) to visualize the state changes in real-time
- **GUI Layout with Tkinter:**
  - A Canvas is used to draw the grid.
  - Each cell is a rectangle.
  - The NPC is drawn as a red circle.
  - Each goal is drawn as a green circle.
  - Colors are used to highlight algorithm states:
    - walls = black,
    - open set = yellow,
    - closed set = gray,
    - current node = blue,
    - final path = turquoise.

- A small control panel on top allows the user to change:
    - Rows, Cols and apply a new grid size.

- **Algorithm Abstraction:**
  Dijkstra and A* were designed to share the same core structure:
    - both use a priority queue (heapq),
    - both maintain g-costs, a came_from dictionary, and visited sets,
    - the only difference is the heuristic:
        - in Dijkstra: heuristic = 0,
        - in A*: heuristic = Manhattan distance.

- **Multi-Goal Strategy:**
  For visiting all goals, a brute-force TSP-style approach was chosen:
    - generate all permutations of goal orders (when number of goals is small),
    - compute shortest path distance between each pair using Dijkstra,
    - choose the order with minimum total cost,
    - then run the selected algorithm (A* or Dijkstra) segment by segment for animation and metrics.

4.3 Coding the Core Logic

The implementation started with the core algorithm and data structures:

1. **Grid and Utility Functions**
    - Implemented helper functions such as:
        - neighbors(pos) to return valid adjacent cells (up, down, left, right),

        - reconstruct_path(came_from, start, goal) to rebuild the path from the parent links.

2. *Single-Goal A and Dijkstra**
    - Implemented a general search function that can behave as:
        - **Dijkstra:** when the heuristic term is set to zero,
        - *A:** when the heuristic term is the Manhattan distance.
    - The function returns:
        - the final path,
        - the number of visited nodes,
        - the list of intermediate "steps" (current, open, closed) for animation,
        - and the execution time in milliseconds.
3. **Multi-Goal Tour Computation**
    - Implemented compute_best_order(start, goals):
        - builds a distance matrix between start and all goals using Dijkstra,
        - tries all permutations of visiting orders using itertools.permutations,
        - selects the order with minimum total path cost.
4. **Metrics Collection**
    - For each segment (from one point to the next), the program accumulates:
        - path length (number of steps),
        - number of visited nodes,

        - execution time.
    - These are aggregated to compare Dijkstra vs A* over the entire tour.

4.4 GUI Integration and Animation

After the core logic was working in the console, the next step was to integrate it with the Tkinter GUI:

1. **Interactive Editing**
    - Mouse clicks were bound to a handler that:
        - sets the start position in “start mode”,
        - toggles goals in “goal mode” (multiple goals allowed),
        - toggles walls in “wall mode”.
    - Keyboard keys were mapped as follows:
        - S → start mode,
        - G → goal mode,
        - W → wall mode,
        - A → select A*,
        - D → select Dijkstra,
        - Space → run simulation,
        - R → reset the grid,
        - C → random scenario.
2. **Dynamic Grid Resizing**

- User inputs for Rows and Cols are read from entry boxes.
- The cell size (CELL_SIZE) is recalculated to ensure the grid fits within a maximum canvas width/height.
- The grid and canvas are redrawn accordingly.

3. **Search Visualization**
   - At each algorithm step, the GUI:
     - colors closed cells gray,
     - colors open cells yellow,
     - highlights the current node in blue,
     - moves the NPC (red circle) to the current node.
   - After the search finishes for a segment, the final path is animated in turquoise.
4. **Full Tour Replay**
   - Once all segments are processed:
     - the grid colors are reset (walls remain unchanged),
     - the NPC returns to the start position,
     - the application replays the entire shortest tour from start to all goals in one continuous animation.
   - At the end, a summary text is displayed:
     - total path length,
     - total visited nodes,
     - total time for the chosen algorithm vs the other one.

4.5 Testing, Debugging and Refinement

The final stage of development focused on testing and refining the behaviour:

- **Correctness Tests**
    - Checked that both Dijkstra and A* always produced the same shortest path length on the same map.
    - Verified that the multi-goal order produced a consistent, minimal-total-distance tour for small goal counts.
- **Scenario-Based Tests**
    - Ran the application with:
        - 1 goal, no obstacles,
        - 1 goal, dense obstacles,
        - 5 goals,
        - 10 goals,
        - different grid sizes.
    - Observed algorithm behaviour and confirmed that A* typically explored fewer nodes and ran faster than Dijkstra.
- **Usability Improvements**
    - Added clear on-screen instructions (current mode, selected algorithm).
    - Ensured that invalid actions (placing walls on start/goal, unreachable goal situations) are handled gracefully through messages.
    - Tuned animation speeds for clarity (not too fast, not too slow).

Through these steps, the project evolved from a simple single-goal console pathfinder into a fully interactive *A vs Dijkstra multi-goal comparison and visualization tool*.

5. Testing Process and Results

This section explains how the application was tested, what kinds of test cases were used, and how the algorithms were analyzed in terms of correctness, efficiency, and behavior in edge cases.

5.1 Testing Strategy

The testing process combined several approaches:

- **Unit-style testing of core functions**

  The main algorithmic functions such as
  - neighbors(pos),
  - reconstruct_path(came_from, start, goal),
  - search_with_steps(start, goal, algorithm),
  - compute_best_order(start, goals)

    were tested on small, hand-crafted grids to ensure they returned expected paths and distances.
- **Integration testing of GUI + algorithms**

  The interaction between the Tkinter interface and the pathfinding logic was tested by:
  - placing start, goals, and walls with the mouse,
  - switching modes and algorithms with the keyboard,
  - running the simulation and observing whether the animation, metrics, and visual states (open/closed/current) were consistent.
- **Scenario-based manual testing**

  Different grid configurations were created to simulate realistic use cases:
  - varying numbers of goals (1, 5, 10),
  - different obstacle densities (no walls, sparse walls, dense walls),

- different grid sizes.

  For each scenario, both A* and Dijkstra were executed and their behavior compared.

- **Informal user testing**

  Group members used the application as if they were students trying to understand the algorithms. Based on their feedback, the on-screen instructions, key bindings, and animation speeds were adjusted to make the tool more intuitive.

5.2 Test Cases and Observed Results

5.2.1 Single Goal, No Obstacles

Scenario:

- Medium grid (e.g., 15×20),
- one start and one goal,
- no walls.

Expected behavior:

Both algorithms should find the straight-line Manhattan shortest path, with identical path length.

Observed results:

In the second scenario involving obstacles, we observed that while both algorithms successfully found the optimal path, their search patterns differed significantly. Dijkstra expanded nodes in a uniform "circular wave" regardless of the goal's direction, unnecessarily exploring safe but irrelevant areas. In contrast, A demonstrated a focused search behavior, navigating around obstacles more strictly towards the target. This resulted in A* processing significantly fewer nodes compared to Dijkstra, even though the final path length remained identical.

5.2.2 Single Goal, With Obstacles

Scenario:

- Same grid size,
- one goal,
- several walls placed to create detours and narrow corridors.

Expected behavior:

The NPC must avoid obstacles and still reach the goal with minimum steps. Both algorithms should return the same shortest path length, but search patterns should differ.

Observed results:

- Both algorithms always found a valid shortest path whenever one existed.
- Dijkstra expanded nodes in a more "circular wave" around the start, often exploring many cells that do not belong to the final path.
- A* focused the search towards the goal, exploring fewer side branches.
- In more complex obstacle layouts, A* typically visited much fewer nodes and ran significantly faster, while the final path length matched Dijkstra exactly.

5.2.3 Multiple Goals (5 and 10 Goals)

Scenario:

- One start and 5 or 10 goals distributed across the grid,
- sparse obstacles,
- multi-goal tour computed using a brute-force permutation approach.

Expected behavior:

- The compute_best_order function should find the visiting order with minimal total distance (for the limited number of goals).
- For this fixed order, both Dijkstra and A* should:
    - produce identical segment path lengths,
    - differ only in visited nodes and execution time.

Observed results:

- The multi-goal order was consistent: the agent did not make visually "strange" detours and the total path length matched the sum of shortest paths between successive goals.
- On each segment:
    - A* and Dijkstra again produced the same shortest path length,
    - A* visited fewer nodes and completed the segment faster.
- Over the entire tour (5 or 10 goals), A* accumulated:
    - significantly fewer visited nodes in total,

- noticeably smaller total running time.

The advantage of A* became more evident as the number of goals increased, because the tour contained more segments and therefore more repeated searches.

5.2.4 Dense Obstacles and Unreachable Goals

Scenario:

- Many walls blocking certain regions,
- some goals placed behind walls so they are unreachable,
- or narrow passages that make the search harder.

Expected behavior:

- If a goal is completely unreachable, the search should terminate and report that no path exists.
- The application should inform the user and not crash.

Observed results:

- When a segment between two points had no valid path, the distance in the internal matrix became effectively infinite.
- compute_best_order detected that a full tour visiting all goals was not possible and the application displayed an informative message instead of trying to animate a broken path.
- This prevented crashes and made the system more robust in edge cases.

5.2.5 Bugs Found and Fixes

During testing, several issues were discovered and fixed:

- **Bug 1: Placing goals or start on walls**
  - Initially, it was possible to place a goal or the start node on a cell that was previously marked as a wall, causing inconsistent states.
  - **Fix:** The click handler was updated to clear walls when placing start/goals, and to block placing walls on existing start/goal cells.
- **Bug 2: Grid resize breaking references**
  - After changing the grid size, some internal references to rectangles and cells were outdated, leading to wrong drawing or no animation.

- **Fix:** On each resize, the grid, canvas, and all cell rectangles are fully reinitialized (init_grid_draw() and redraw_static()), and the NPC position is reset.

- **Bug 3: Multi-goal tour with unreachable segment**
  - Early versions did not correctly handle a case where a specific segment between two goals had no path; this could result in invalid permutations being treated as valid.
  - **Fix:** If any distance is infinite for a segment, that permutation is marked infeasible. If all permutations are infeasible, the program informs the user that a complete tour is impossible.

These fixes improved both correctness and user experience.

5.3 Algorithm Analysis

5.3.1 Correctness

The correctness of the implementation was evaluated in two layers:

1. **Single-Goal Correctness**
   - For any start–goal pair on the grid, Dijkstra's algorithm is known to return the optimal shortest path when all edge costs are non-negative (in our case, each move has cost 1).
   - A* with Manhattan heuristic is guaranteed to be optimal on a 4-directional grid, because the heuristic never overestimates the true distance.
   - In all tested scenarios, the path length reported by A* and Dijkstra was always identical, which is strong empirical evidence that both algorithms are correctly implemented.
2. **Multi-Goal Tour Correctness**
   - The distance between each pair of points (start and goals) is computed using Dijkstra, which is optimal.
   - For a small number of goals, all permutations of goal orders are checked. This brute-force method guarantees that the selected order has the minimum total distance among all possible orders.

- Therefore, for the tested ranges (e.g., 1, 5, 10 goals), the total tour is also optimal with respect to the shortest path distances between points.

5.3.2 Efficiency and Complexity

From a theoretical perspective, both A* and Dijkstra run in
O(E log V) time on a graph, where V is the number of vertices and E is the number of edges.

In our grid representation, each cell corresponds to a vertex and each cell has up to four neighbours (up, down, left, right), so we have:

- $V = N = R \cdot C$(number of cells),
- $E \approx 4N = O(N)$.

Both algorithms use a binary heap priority queue and therefore have the same asymptotic worst-case complexity. However, their practical behaviour is very different:

- **Dijkstra** explores nodes in an expanding "wavefront" from the start and does not use any information about the goal location.
- **A*** uses a heuristic (in our case, Manhattan distance) to guide the search towards the goal, which significantly reduces the number of nodes that are inserted into and extracted from the priority queue.

In our experiments:

- For simple maps, the difference in execution time is small, but A* still visits fewer nodes than Dijkstra.
- As the grid becomes larger or contains more obstacles, Dijkstra tends to explore a large portion of the map, while A* remains more focused around the relevant region.
- Over a multi-goal tour, these per-segment savings accumulate. A* usually achieves:
  - fewer total visited nodes,
  - smaller total execution time,

  while producing the same shortest path length as Dijkstra.

Time Complexity Analysis

Let the grid have $R \times C$cells and let $N = R \cdot C$be the total number of positions. In graph terms, we have $V = N$vertices and at most $E \approx 4N = O(N)$edges.

For a single start–goal query, both Dijkstra and A*:

- insert each vertex into the priority queue at most once,
- relax each edge at most once.

This gives the standard time complexity:

$$T_{\text{single}}(N) = O((V + E)\log\ V) = O(N\log\ N),$$

and space complexity O(N) for storing the g-values, heuristic values, parent pointers, and visited sets.

The Manhattan heuristic used by A* reduces the number of explored nodes in practice, but it does not change this worst-case Big-O bound compared to Dijkstra.

For the multi-goal tour computation, let k be the number of goals:

1. **Pairwise distance matrix.**
   We first compute all pairwise shortest paths between the start and each goal by running Dijkstra for each ordered pair.
   There are $O(k^2)$such pairs, and each shortest-path computation costs $O(N\log\ N)$.
   This step therefore costs:
   $$O(k^2 N\log\ N).$$

2. **Brute-force TSP over goals.**
   Next, we enumerate all permutations of the $k$goals to find the visiting order with minimum total distance.
   There are $k!$permutations, and computing the tour length for one permutation takes $O(k)$time.
   This adds an extra $O(k \cdot k!)$factor.

Combining these two parts, the complexity of the multi-goal ordering step is:

$$T_{\text{order}}(N, k) = O(k^2 N\log\ N + k \cdot k!).$$

After the visiting order is fixed, the simulation runs the selected algorithm (A* or Dijkstra) for each of the k segments of the tour, and also runs the other algorithm for comparison. Each segment takes $O(N\log\ N)$, so this phase contributes an additional:

$$O(kN\log\ N).$$

Animating the path on the GUI takes $O(L)$operations, where L is the total tour length, which is typically dominated by the pathfinding cost.

In summary:

- For a **single shortest-path query**, both A* and Dijkstra have the same asymptotic complexity $O(N\log\ N)$on our grid.
- In practice, A* is significantly more efficient because it explores far fewer nodes thanks to the heuristic.
- The **multi-goal extension** is theoretically optimal but has factorial complexity $O(k \cdot k!)$in the number of goals. This explains why our implementation is practical only for small $k$values (e.g., 1, 5, 10 goals) and becomes slow or unstable when the number of goals is much larger.

5.3.3 Edge Case Behavior

Several edge cases were specifically tested to understand algorithm behavior:

- **Start equals goal**
  - When the start cell is the same as the goal, both algorithms immediately return a path of length zero. The animation reflects this by not moving the NPC.
- **No goals selected**
  - If the user tries to run the simulation without any goal, the application shows an informative message and does not attempt to run an algorithm.
- **No possible path (completely blocked goal)**
  - When walls fully isolate a goal, both algorithms eventually exhaust the open set and report that no path exists. This is propagated to the multi-goal module, which then reports that a full tour cannot be constructed.
- **Very dense obstacles**

- For heavily blocked maps where only narrow tunnels exist, both algorithms slow down because many nodes must be considered.
- A* still has an advantage as it explores fewer blind branches, but in extreme cases, execution time naturally increases for both methods due to the problem complexity.

Overall, the tests indicate that the implemented algorithms are correct, efficient for the considered scenarios, and robust against common edge cases. The visual and metric-based comparison clearly demonstrates the practical difference between Dijkstra and A* in realistic grid-based navigation tasks.

5.4 Scenario-Based Quantitative Comparison of A* and Dijkstra

To quantitatively compare A* and Dijkstra under different conditions, we designed three controlled scenarios. In all cases, both algorithms were run on the same grid, with the same start and goal positions and identical obstacles. For each scenario, we measured:

- **Path length** (number of steps in the final shortest tour),
- **Visited nodes** (total number of expanded nodes),
- **Execution time** in milliseconds.

The results are summarized below:

Scenario	Grid Size	Goals	Algorithm	Path Length	Visited Nodes	Time (ms)
1 Start, 1 Goal	15 × 15	1	A*	28	29	0.09
			Dijkstra	28	225	0.91
1 Start, 5 Goals	30 × 30	5	A*	16	369	1.47
			Dijkstra	16	2258	22.42
1 Start, 10 Goals	30 × 30	10	A*	109	200	0.68
			Dijkstra	109	1688	7.26

Scenario 1: 1 Start – 1 Goal (15×15)

In the simplest case with a single start and a single goal on a 15×15 grid:

- Both A* and Dijkstra produced the *same optimal path length* of 28.

- However, Dijkstra visited about 7.8× more nodes (225 vs 29).
- The execution time of Dijkstra was also roughly 10× slower (0.91 ms vs 0.09 ms).

This confirms that, even in a relatively small and simple map, A* already gains efficiency by focusing the search towards the goal, while preserving optimality.

Scenario 2: 1 Start – 5 Goals (30×30)

In the second scenario, the grid size was increased to 30×30 and the NPC had to visit 5 different goals in an optimal tour:

- Both algorithms again produced the same total path length of 16.
- A* visited 369 nodes, while Dijkstra visited 2258 nodes, which is about 6.1× more.
- In terms of execution time, A* finished in 1.47 ms, whereas Dijkstra needed 22.42 ms, approximately 15× slower.

As the problem becomes more complex (larger grid + multiple goals), the performance gap between A* and Dijkstra becomes much more visible. The heuristic of A* reduces unnecessary exploration and accumulates significant time savings over multiple segments of the tour.

Scenario 3: 1 Start – 10 Goals (30×30)

The third scenario further increases the complexity by using 10 goals on a 30×30 grid:

- The total path length for the full tour was 109 for both A* and Dijkstra, again demonstrating that both algorithms are finding the same optimal solution.
- A* visited 200 nodes in total, while Dijkstra visited 1688 nodes, which is about 8.4× more.
- Execution times were 0.68 ms for A* and 7.26 ms for Dijkstra, so Dijkstra was roughly 10.7× slower.

Interestingly, even though the tour is much longer (109 steps), A* still maintains low visited-node counts and execution time, thanks to its heuristic guidance. Dijkstra, on the other hand, repeatedly explores large regions of the grid when computing each segment of the multi-goal tour.

Discussion of the Scenario Results

Across all three scenarios, some consistent patterns emerge:

- **Correctness:**

  For every test case, A* and Dijkstra returned the same path length. This empirically confirms that, with the Manhattan heuristic, A* remains optimal on our grid.
- **Search Effort (Visited Nodes):**

  Dijkstra always visits significantly more nodes than A*:
  - ~7.8× more nodes in the 1-goal scenario,
  - ~6.1× more nodes in the 5-goal scenario,
  - ~8.4× more nodes in the 10-goal scenario.
- **Execution Time:**

  The execution time grows more quickly for Dijkstra as the grid and number of goals increase. A* consistently completes the tour about 10–15× faster than Dijkstra in these experiments.

Overall, these quantitative results strongly support the theoretical expectation:
A* and Dijkstra are equally correct in terms of optimal path length, but A* is much more efficient in practice, especially when the environment becomes larger or includes multiple goals and obstacles.

6. Evaluation and Discussion

This section evaluates the performance, usability, and overall success of the developed application. It also reflects on what worked well and what could be improved in future versions.

6.1 Performance Evaluation

From a performance perspective, the application achieved its main goal: it provided a clear and measurable comparison between Dijkstra and A* on the same grid configurations.
In almost all test scenarios:

- Both algorithms produced identical shortest path lengths, confirming that the heuristic used in A* (Manhattan distance) does not sacrifice optimality.
- A* consistently:
  - visited fewer nodes,

    - and ran faster in terms of execution time (measured in milliseconds), especially as the grid size, number of obstacles, or number of goals increased.

The performance difference became more noticeable in:

- **dense obstacle** scenarios, where Dijkstra explored a large portion of the grid in a blind wavefront,
- **multi-goal tours**, where the pathfinding step is repeated multiple times and the savings of A* accumulate.

Although the implementation uses a brute-force approach for the multi-goal visiting order (TSP-style), it remains practical for the target scale of the project (1–10 goals). For larger numbers of goals, this approach would not be efficient, but such cases were outside the scope of this course project.

Overall, the performance results are consistent with theoretical expectations and demonstrate the practical advantage of heuristic search in grid-based navigation.

6.2 Usability and Educational Value

In terms of usability, the application proved to be intuitive and easy to interact with:

- The user can:
    - place the start, multiple goals, and walls directly on the grid with the mouse,
    - select the algorithm (A* or Dijkstra) with a single key press,
    - and start the simulation with the space bar.
- On-screen labels clearly indicate:
    - the current editing mode (start / goal / wall),
    - the currently selected algorithm,
    - and the summary of performance metrics after each run.

The visualization is especially helpful for learning:

- Open set, closed set, and current node are shown in different colors.
- The NPC’s movement is animated step by step, making the search process easier to follow.
- After the segment-wise animation, the application replays the full optimal tour from the starting point, giving a clear picture of the final result.

This makes the tool not only a working pathfinding system, but also an educational visualization that supports the understanding of the Design and Analysis of Algorithms course topics. Informal user testing (by group members) indicated that the interface is understandable after a short explanation and that the color-coded animations are effective for illustrating how the algorithms explore the grid.

6.3 Achievement of Project Goals

The original objectives of the project were:

1. Implement Dijkstra and A* on a grid.
2. Allow interactive placement of start, multiple goals, and obstacles.
3. Extend the problem to a multi-goal tour and compute an order that minimizes total distance.
4. Visually animate the search process and the final paths.
5. Collect and compare performance metrics (path length, visited nodes, time) for both algorithms.

Based on the implementation and testing:

- All these goals were successfully met:
    - The algorithms work correctly on a variety of scenarios.
    - The multi-goal tour is computed using a brute-force TSP approach for a small number of goals.
    - The difference between Dijkstra and A* is visible both numerically and visually.
- The application fulfills its main educational purpose: it allows users to experiment with different maps and immediately see how the choice of algorithm affects efficiency.

Therefore, the project can be considered successful in both functional and educational terms.

6.4 What Worked Well

Several aspects of the project worked particularly well:

- **Shared Algorithm Structure:**
  Implementing Dijkstra and A* using almost the same code (with only the heuristic differing) made the comparison fair and the implementation easier to maintain.
- **Step-by-Step Visualization:**
  Showing open, closed, and current nodes during the search provided a strong visual intuition about how each algorithm “thinks”.
- **Multi-Goal Extension:**
  Turning the classic single-goal shortest path into a multi-goal tour (with an order chosen by brute-force TSP) added a realistic and interesting dimension to the project.
- **Interactive Grid Editing:**
  The ability to freely draw obstacles and place goals anywhere allowed flexible scenarios and made testing more engaging.

6.5 Limitations and Possible Improvements

Despite its strengths, the application also has some limitations and areas for improvement:

- **Scalability of the Multi-Goal Approach:**
  The brute-force permutation method for choosing the best visiting order does not scale to a large number of goals. In future work, heuristic or approximation methods (e.g., greedy TSP heuristics, genetic algorithms) could be used to handle more goals efficiently.
- **Movement Model:**
  The current implementation supports only 4-directional movement (up, down, left, right). Adding diagonal movement and adjusting the heuristic accordingly could make the paths more natural in some scenarios.
- **More Advanced Visualization and UI:**
  While Tkinter is sufficient for the current project, a more modern UI framework (such as Pygame, PyQt, or a web-based interface) could provide smoother animations, better scaling on different screen sizes, and richer user interactions.

- **Automatic Test Suite and Logging:**
  At the moment, most tests are manual and scenario-based. A future version could include:
  - automated unit tests,
  - a logging system to export results (e.g., to CSV) for more systematic analysis.

In summary, the application meets its primary goals of implementing and comparing Dijkstra and A* in a multi-goal, obstacle-filled grid environment. It performs well for the intended problem size, is usable and informative for its target audience, and provides a solid foundation for further extensions in both performance and user experience.

7. Conclusion, Future Work, and Team Contributions

7.1 Conclusion

In this project, we designed and implemented an interactive pathfinding simulation that compares Dijkstra and A* algorithms in a grid-based environment with multiple goals and obstacles. The application allows users to place a start point, several goals, and walls, and then visually observe how each algorithm searches for the shortest paths and how it behaves as the environment becomes more complex.

The results clearly show that:

- Both algorithms consistently produce the same optimal path length,
- but A* is more efficient in terms of:
  - number of visited nodes,
  - and execution time,
    especially when the grid is larger, contains more obstacles, or includes multiple goals.

Through this project, we strengthened our understanding of:

- graph-based shortest path algorithms,
- heuristic search and the role of admissible heuristics,
- how algorithm complexity appears in practice (not only in Big-O notation),
- and how to turn theoretical concepts from the *Design and Analysis of Algorithms* course into a working, visual application.

We also gained practical experience in using Python, Tkinter, and structured software development steps (planning, design, coding, testing, and evaluation).

7.2 Future Work

Although the project successfully met its main goals, there are several directions for future improvement:

- **Scalable Multi-Goal Optimization**
  Currently, the visiting order of goals is computed using a brute-force permutation approach, which is optimal but not scalable for many goals. Future work could integrate:
  - greedy TSP heuristics,
  - metaheuristics (e.g., genetic algorithms, simulated annealing),
    to handle larger numbers of targets efficiently.
- **Extended Movement and Cost Models**
  The current model only allows 4-directional movement with uniform cost. Possible extensions include:
  - diagonal movement with appropriate cost and heuristic,
  - different terrain types with different movement costs.
- **Richer User Interface and Platform**
  The Tkinter-based interface is simple and effective for our needs, but the application could be improved by:

    - using a game-oriented framework (e.g., Pygame) or a modern GUI framework (e.g., PyQt) for smoother animations,
    - or even porting the visualization to a web-based environment to make it more accessible.
- **Automatic Testing and Data Export**
  Adding:
    - an automated test suite,
    - and an option to export metrics (path length, visited nodes, time) to files (e.g., CSV),
      would make it easier to conduct larger experimental studies and compare algorithm performance more systematically.

These improvements would turn the current educational prototype into a more robust and extensible platform for research and teaching in pathfinding and heuristic search.

7.3 Team Contributions

The project was developed collaboratively by three students: Mertcan İLHAN, Tunahan İÇÖZ, and Oğuzhan YAZICI. Each member contributed to different aspects of the work:

- **Mertcan İLHAN**
    - Implemented the core pathfinding logic for Dijkstra and A* (including the Manhattan heuristic and path reconstruction).
    - Integrated the algorithms with the multi-goal tour computation and performance measurement (path length, visited nodes, execution time).
    - Helped design the test scenarios and interpreted the experimental results for the report.
- **Oğuzhan YAZICI**
    - Designed and implemented the Tkinter-based graphical user interface, including the grid drawing, NPC and goal visualization, and color-coded search animation (open, closed, current, final path).

    - Developed the interactive controls for placing start, goals, and walls, changing grid size, and running/resetting simulations.
    - Worked on improving usability (on-screen instructions, key bindings, layout).
- **Tunahan İÇÖZ**
    - Focused on testing, debugging, and refinement of the application, including edge cases such as unreachable goals, dense obstacles, and grid resizing.
    - Collected and organized the experimental data used to compare Dijkstra and A*.
    - Contributed heavily to the documentation and report writing, especially in the testing, evaluation, and discussion sections.

Overall, the collaboration allowed us to combine algorithmic thinking, user interface design, and systematic testing into a complete and coherent project that meets the objectives of the course.

8. References

[1] E. W. Dijkstra, “A Note on Two Problems in Connexion with Graphs,” *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Transactions on Systems Science and Cybernetics*, vol. SSC-4, no. 2, pp. 100–107, 1968.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Hoboken, NJ, USA: Pearson, 2020.

[5] Python Software Foundation, “Python 3 Documentation,” Accessed: Nov. 28, 2025. [Online]. Available: https://docs.python.org/3/

[6] Python Software Foundation, “Tkinter — Python Interface to Tcl/Tk,” Accessed: Nov. 28, 2025. [Online]. Available: https://docs.python.org/3/library/tkinter.html

[7] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*, 3rd ed. Berlin, Germany: Springer, 2008.

[8] “IEEE Citation Guidelines,” IEEE, Accessed: Nov. 28, 2025. [Online]. Available: https://www.ieee.org/documents/ieee_citation_guidelines.pdf

[9] “A* Pathfinding for Beginners,” *Red Blob Games*, Accessed: Nov. 28, 2025. [Online]. Available: https://www.redblobgames.com/pathfinding/a-star/introduction.html

[10] NetworkX Developers, “Shortest Path Algorithms — NetworkX Documentation,” Accessed: Nov. 28, 2025. [Online]. Available: https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html