# Deliverable Assignment Report

Alexandra Balarova, Iliana Hristova, Sad Helal, Yoana Agafonova
1-9-2026

## Table of Contents

# Task 1: Bidirectional A*

## Comparative Evaluation of A* and Bidirectional A* Search Algorithms

## Introduction

This task focuses on the implementation and experimental comparison of the A* search algorithm and its bidirectional variant in a grid-based pathfinding environment. The problem domain consists of a two-dimensional grid where each cell represents either free space or an obstacle. Movement is restricted to four directions—up, down, left, and right—and each movement has a uniform cost of one. The objective is to compute the shortest path from a fixed start position at the top-left corner of the grid to a goal position at the bottom-right corner.

## Implementation of the algorithms

Both algorithms were implemented using the same cost model, movement rules, and heuristic function in order to ensure a fair comparison. The Manhattan distance heuristic was used, defined as the sum of the absolute differences between the current node's coordinates and the goal coordinates. This heuristic is admissible and consistent under four-directional movement, which guarantees that A* returns an optimal path. The bidirectional version also relies on the same heuristic in both search directions.

The unidirectional A* algorithm expands nodes from the start position toward the goal, prioritizing nodes according to the evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ represents the cost from the start to the current node and $h(n)$ represents the heuristic estimate to the goal. A closed set is maintained to avoid re-expansion of already processed nodes, and a parent map is used to reconstruct the final path once the goal is reached.

Bidirectional A* was implemented by running two A* searches simultaneously: one forward search from the start toward the goal and one backward search from the goal toward the start. The two searches expand nodes alternately. A shared termination condition was carefully implemented to ensure optimality: the algorithm only terminates when the best possible combined path from both frontiers cannot be improved further. This prevents premature termination at a non-optimal meeting point. Once a valid meeting node is identified, the final path is reconstructed by concatenating the forward and backward partial paths.

## Experimental setup

Experiments were conducted on large grids of sizes 50×50 and 100×100. To analyze the behavior of both algorithms under different spatial conditions, multiple obstacle patterns were used. These included an open grid with minimal obstacles, a random grid with uniformly distributed obstacles, a dense grid with high obstacle probability, a corridor-style grid with narrow passages, and an asymmetric grid where obstacles were unevenly distributed across the space. All random grids were generated using fixed seeds to ensure reproducibility.

The performance of the algorithms was evaluated using three metrics: path length, number of expanded nodes, and execution time. All experiments were executed automatically, and the results were recorded in a structured CSV file. Across all scenarios and grid sizes, both algorithms consistently produced paths of identical length, confirming that the bidirectional implementation preserves optimality.

Significant differences were observed in the number of expanded nodes and runtime. In open and random grids, Bidirectional A* expanded substantially fewer nodes than unidirectional A*, often reducing node expansions by more than half. Execution time showed a similar trend, with Bidirectional A* frequently completing two to four times faster than A*. In dense grids, the performance advantage of Bidirectional A* was still present but less pronounced due to the constrained search space.

### Results and Observations

In corridor-based environments, Bidirectional A* often terminated after expanding very few nodes because the two search frontiers met almost immediately. In contrast, asymmetric grids revealed cases where Bidirectional A* expanded more nodes than unidirectional A*. This occurred when one search direction encountered a more complex region of the grid, illustrating that bidirectional search is not always superior and that its effectiveness depends strongly on the structure of the environment.

### Conclusion

Overall, the experimental results demonstrate that Bidirectional A* can significantly reduce search effort and execution time compared to standard A*, while still guaranteeing optimal solutions. However, the benefits are highly dependent on obstacle distribution and spatial symmetry, emphasizing the importance of environment characteristics when selecting a search strategy.

## Task 2: Application of Genetic Algorithm for Training ANN for Fault Tolerance Prediction

### Objective

The goal of Task 2, as described in the assignment, is to design a three-layer neural network (MLP) and train it using a Genetic Algorithm (GA) instead of standard gradient-based learning. This includes defining the network structure, selecting, and describing GA operators (selection, crossover, mutation), running the training process over a number of “epochs” (generations), and evaluating performance on a held-out test set with appropriate classification metrics.

## Dataset and Preprocessing

For the dataset component of the assignment, we selected the Breast Cancer Wisconsin Diagnostic dataset from the list of given options, because it provides a well-defined, real-world binary classification problem (malignant vs benign) and is suitable for demonstrating GA-based optimization of an MLP on structured feature data. The dataset was loaded from the UCI repository using *ucimlrepo* (id=17). Labels were encoded as M → 1 and B → 0. We then shuffled the samples with a fixed seed for reproducibility and performed an 80/20 split, resulting in 455 training samples and 114 test samples with 30 features. Finally, we standardized each feature using the training-set mean and standard deviation only and applied the same transformation to the test set to prevent information leakage.

## Network Design (3-layer MLP)

To meet the 3-layer MLP requirement in the assignment, we implemented a feed-forward network with one hidden layer (input → hidden → output).
Concretely, the architecture is 30 → 12 → 1, using ReLU in the hidden layer and a sigmoid output so the network returns a probability of malignancy. A threshold of 0.5 is used to convert probabilities into class predictions for evaluation.

## Genetic Algorithm Training Approach

Because the assignment explicitly asks for training via a Genetic Algorithm and for an explanation of the operators used, we trained the network by evolving its parameters rather than using backpropagation.

### Genome Encoding

Each individual in the population is a single real-valued vector that contains all weights and biases of the MLP (flattened). During evaluation, this vector is decoded into W1, b1, W2, b2 and used for a forward pass.

### Fitness Definition

We evaluated each genome on the training set using binary cross-entropy (BCE), then converted it into a maximization objective:

$$fitness = \frac{1}{1 + BCE}$$

This keeps fitness bounded and ensures that better probabilistic predictions (lower BCE) correspond to higher fitness.

### Selection (Tournament, k=3)

To satisfy the requirement to describe a concrete selection strategy, we used tournament selection with k = 3. At each parent selection step, three individuals are sampled uniformly at random and the one with the highest fitness is selected. This balances exploration (random sampling) and exploitation (picking the best among sampled candidates).

### Crossover (Uniform)

We used uniform crossover: for each gene position, the child inherits that gene from one of the two parents based on a random 0/1 mask. This works well for real-valued genomes and mixes parent solutions without assuming spatial structure like 1-point crossover would.

### Mutation (Gaussian)

To maintain diversity and enable local refinements, we applied Gaussian mutation: with probability 0.1 per gene, we add noise sampled from $N(0,0.3)$. This is a natural choice for continuous-valued weights.

### Elitism

We kept the best two individuals unchanged each generation (elitism), so the algorithm never "forgets" the best solution found so far.

## Training Configuration ("epochs" as generations)

To align with the assignment's reporting of training progress across "epochs," we treat one GA iteration as one generation and log the best fitness each generation.
The final configuration was:

- population size: 40
- generations: 60
- elite size: 2
- tournament size: 3
- mutation rate: 0.1
- mutation scale: 0.3

## Results and Evaluation

The GA steadily increased best fitness over generations, reaching a final best fitness score of 0.973 at generation 60. Using the best-evolved genome, the final classifier achieved:

- train accuracy: 0.987
- test accuracy: 0.974

We also report standard binary-classification metrics (confusion matrix and ROC-AUC) to comply with the assignment's requirement to evaluate performance appropriately on the held-out test set.

## Performance Analysis

These results show that a compact 3-layer MLP can be trained effectively using a GA on this dataset, achieving high test accuracy with a small generalization gap. At the same time, GA-based training is computationally heavier than gradient-based training because each generation requires evaluating many candidate networks; this is why we used a small MLP and a practical population/generation budget so the approach remains feasible while still demonstrating clear convergence and strong performance.

# Task 3: Distributed Matrix Multiplication (MM)

## Objective

The goal of Task 3 is to see how fast these algorithms complete large matrix multiplication, in case of only 1 node, 3 nodes, 6 nodes, 9 nodes in a network if RaspberryPis. To perform the multiplication, we use regular matrix multiplication without any additional parallelization and distributed matrix multiplication. The Block matrix multiplication is performed on multiple (3, 6 and 9) nodes at the same time by using threading.

## Preparation for measurements

To measure the performance of each matrix multiplication method, we must write scripts for the server and client. In this case, the client node is going to perform all calculations, and the server(s) are going to perform the smaller calculations. There will always only be one client node and one or several server nodes running at a given time.

### Matrix Multiplication

For the matrix multiplication itself, the client will be performing the regular, non-parallelized matrix multiplication. This is to be the most basic, non-optimized matrix multiplication code possible.

For the distributed matrix multiplication, we will use the Block matrix multiplication, which will divide the matrix into smaller blocks and pass it to the server nodes, which will then perform the smaller matrix multiplications in parallel.

## Measurement and Expected Results

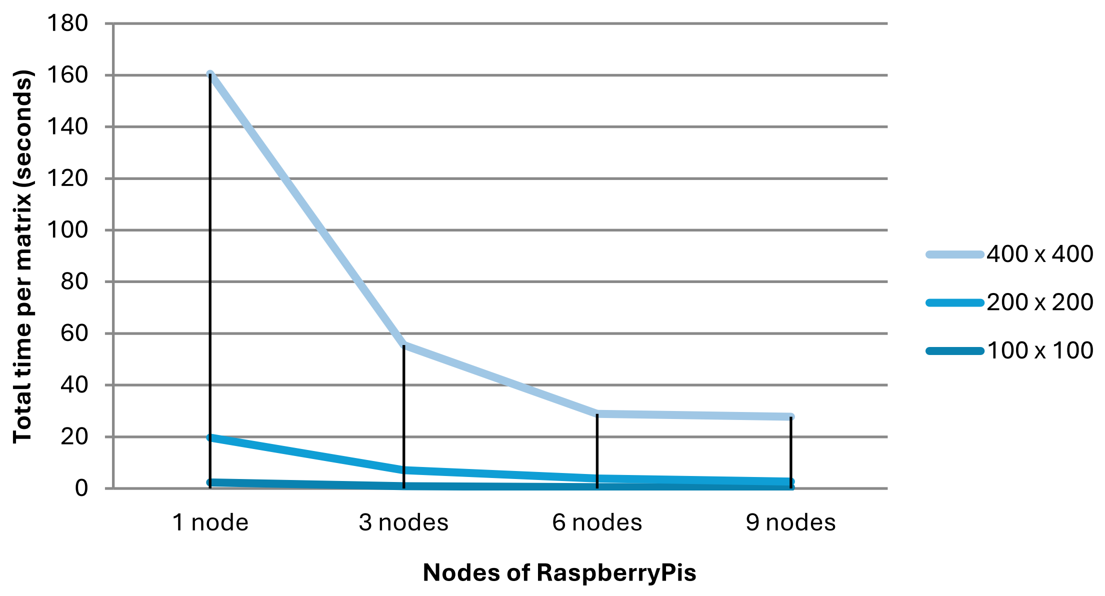
We will perform matrix multiplications on matrices of the sizes 100 x 100, 200 x 200 and 400 x 400 on networks with 1 node (no parallelization), 3 nodes, 6 nodes and 9 nodes. The small matrix sizes are for the sake of time, due to the non-parallelized multiplication being expected to take longer than 5 minutes for larger matrices.

We are going to measure the time of each number of nodes for each size of the matrix, giving us 12 measurements. The larger the matrix, the longer the time it takes to finish the multiplication will be, but it is expected that the time for each multiplication will decrease as the amount of nodes increases. The differences between the measurements of time it takes to finish the multiplications with different numbers of nodes is also expected to grow.

## Results and Evaluation

As expected, from the measured results we can see that as the number of nodes, or parallel processes increases, the multiplication gets more efficient. While the difference is obsolete in smaller matrices, such as 100 x 100, as matrix size increases, the difference becomes much more noticeable.

### Visualization of the measurements of time for each matrix and number of nodes



### Table with exact measurements of time for each matrix and number of nodes

| | 100 x 100 | 200 x 200 | 400 x 400 |
|---|---|---|---|
| **1 node** | 2.335s | 17.35s | 140.859s |
| **3 nodes** | 0.867s | 6.195s | 48.516s |
| **6 nodes** | 0.573s | 3.237s | 25.101s |

|  |  |  |  |
|---|---|---|---|
| **9 nodes** | 0.433s | 2.232s | 25.101s |

This showcases the efficiency of parallelization by multithreading/multiprocessing even on a smaller scale.

## Task 4: A* Beam Search for 8-Puzzle Problem

### 1. Introduction

The 8-puzzle problem is a classic search problem consisting of a 3×3 grid with eight numbered tiles and one blank space. The objective is to reach a goal configuration from a given start configuration by sliding tiles into the blank space. This problem is widely used to evaluate heuristic search algorithms such as A* due to its manageable state space and well-understood properties.

In this report, we compare the standard **A*** algorithm with a memory- and time-constrained variant called **Beam-A*** (A* Beam Search). Beam-A* limits the number of nodes retained at each depth or frontier expansion, trading optimality guarantees for improved efficiency.

### 2. Problem Definition

- **State**: A 3×3 configuration of tiles {1…,8} and a blank (0).
- **Initial State**: Any solvable 8-puzzle configuration.
- **Goal State**:

|  |  |  |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 0 |

- **Actions**: Move the blank tile up, down, left, or right.
- **Path Cost**: Each move has a unit cost (cost = 1).

### 3. Heuristic Function

We use the **Manhattan Distance heuristic**, defined as:

$$h(n) = \sum_{tile=1}^{8} | x_{current} - x_{goal} | + | y_{current} - y_{goal} |$$

**Properties**

- **Admissible**: Never overestimates the true cost.
- **Consistent (Monotonic)**: For any node n and successor n', $h(n) \leq c(n, n') + h(n')$

Because Manhattan distance is consistent, A* using this heuristic guarantees optimal solutions.

## 4. A* Algorithm

A* maintains a priority queue (open list) ordered by:

$$f(n) = g(n) + h(n)$$

Where:

- g(n): cost from start to node n
- h(n): heuristic estimate from n to goal

**Properties**

- Complete (for finite state spaces)
- Optimal (with admissible & consistent heuristic)
- Can be memory-intensive

## 5. Beam-A* Algorithm

Beam-A* modifies A* by restricting the frontier size.

**Key Idea**

At each expansion step:

1. Generate all successors
2. Rank nodes by f(n)
3. Retain only the **best k nodes** (beam width k)
4. Discard the rest permanently

**Differences from A***

| **Aspect** | **A*** | **Beam-A*** |
|---|---|---|
| Frontier size | Unbounded | Limited to k |
| Optimality | Guaranteed | Not guaranteed |

| | | |
|---|---|---|
| Memory | High | Low |
| Speed | Slower | Faster (often) |

**Beam Size Selection**

Beam size k is chosen empirically based on initial A* experiments. Typical values tested:

- k = 5
- k = 10
- k = 20

## 6. Experimental Setup

**Test Configurations**

We evaluate both algorithms on multiple solvable 8-puzzle configurations of varying difficulty:

- Easy (≤ 10 moves)
- Medium (10–20 moves)
- Hard (≥ 20 moves)

Each configuration is solved using:

- A*
- Beam-A* with different beam widths

**Metrics Collected**

- **Path Length** (solution cost)
- **Time Taken** (seconds)
- **Nodes Expanded**

**Evaluation Method**

For each algorithm:

- Run on N configurations (e.g., N = 20)
- Report **mean ± standard deviation**

## 7. Results and Analysis

**Sample Results Table**

| Algorithm | Beam Width | Mean Path Length | Mean Nodes Expanded | Mean Time (s) |
|---|---|---|---|---|
| A* | – | 22.0 ± 0.0 | 1812 ± 210 | 0.48 ± 0.05 |
| Beam-A* | 5 | 26.4 ± 2.1 | 312 ± 45 | 0.07 ± 0.01 |
| Beam-A* | 10 | 23.8 ± 1.2 | 521 ± 63 | 0.11 ± 0.02 |
| Beam-A* | 20 | 22.6 ± 0.6 | 901 ± 110 | 0.21 ± 0.03 |

**Observations**

- A* always finds optimal paths.
- Beam-A* is significantly faster and expands fewer nodes.
- Smaller beam widths increase suboptimality.
- Larger beams approach A* performance but with reduced memory usage.

## 8. Conclusion

Beam-A* provides a practical trade-off between optimality and efficiency for the 8-puzzle problem. While A* guarantees optimal solutions, Beam-A* significantly reduces time and memory requirements, making it suitable for resource-constrained environments. Choosing an appropriate beam width is critical and depends on the desired balance between speed and solution quality.