

ASSIGNMENT-1(Group 3)

Group Details

Name	ID No.
Keshav Mishra	12341140
Shivam Singh	12342020
Rishi Kharya	12341790
Sarthak Gopal Sharma	12341920

1 Question 1

1.1 Problem Description

We are given:

- A graph with N nodes (locations) and M edges (roads).
- Each edge has a distance in kilometers.
- P passengers, each requiring transport from a pickup node to a drop-off node.
- Congestion rules:
 1. At most two taxis can occupy an edge at the same time.
 2. If a third taxi attempts to enter, it must wait for $W = 30$ minutes before retrying.
- Speed of taxis: $S = 40$ km/h.

The objective is to compute the route for each taxi, apply waiting rules, and calculate:

1. Total Completion Time (sum of all taxi times).
2. Makespan (maximum time taken by a single taxi).

1.2 Methodology

Pathfinding with A* Search

To compute the shortest path for each passenger's trip, we use the A* algorithm:

- $g(n)$ = actual travel time from the start node to n .
- $h(n)$ = heuristic estimate of travel time from n to the goal.

Heuristic

We relax the congestion constraint and compute the shortest travel times from every node to the goal using Dijkstra's algorithm. This value is used as the heuristic:

$$h(n) = \text{Shortest uncongested travel time from } n \text{ to goal}$$

Admissibility: Since congestion can only increase travel time, the uncongested shortest-path time never overestimates the true cost. Hence $h(n) \leq h'(n)$, where $h'(n)$ is the actual minimal cost with congestion.

Consistency: For any edge (u, v) with travel time $c(u, v)$, we have:

$$h(u) \leq c(u, v) + h(v)$$

because Dijkstra's shortest-path distances satisfy the triangle inequality. Thus, the heuristic is consistent.

Therefore, A* with this heuristic is guaranteed to return optimal paths (ignoring congestion), which we then adjust during simulation.

Congestion Handling

After paths are computed, we simulate taxi movement using an **event-driven approach**. Events are:

- **ATTEMPT_TO_ENTER:** Taxi tries to enter an edge. If capacity is available, it succeeds; otherwise it waits $W = 30$ minutes before retrying.
- **EXIT:** Taxi leaves an edge, freeing capacity for others.

This ensures congestion rules are respected while taxis follow their precomputed shortest paths.

Implementation

The program is implemented in Python. Dijkstra is used to precompute heuristics, A* is used for routing, and an event-driven simulator enforces congestion rules.

Code Snippet

```

1 def dijkstra_all(goal, graph, nodes):
2     dist = {n: math.inf for n in nodes}
3     dist[goal] = 0.0
4     pq = [(0.0, goal)]
5     while pq:
6         d, u = heapq.heappop(pq)
7         if d > dist[u]:
8             continue
9         for v, w in graph[u]:
10            nd = d + w
11            if nd < dist[v]:
12                dist[v] = nd
13                heapq.heappush(pq, (nd, v))
14    return dist

```

1.3 Results

For the given sample input, the program produced the following output:

```

Taxi 1:
Passenger 2->7
Route: 2 -> 3 -> 5 -> 7

```

Total time = 165.0 minutes

Taxi 2:

Passenger 1->8

Route: 1 -> 2 -> 3 -> 5 -> 7 -> 8

WAIT on edge (2->3): 30 minutes

Total time = 315.0 minutes

Taxi 3:

Passenger 3->4

Route: 3 -> 2 -> 4

Total time = 165.0 minutes

Total Completion Time = 645.0 minutes

(span = 315.0 minutes)

1.4 Conclusion

The approach of using Dijkstra-based travel times as a heuristic in A* ensures efficient and optimal route computation (ignoring congestion). Since the heuristic is both admissible and consistent, A* guarantees correct shortest paths.

The event-driven simulation enforces congestion constraints realistically, introducing delays where necessary. Results show that only Taxi 2 experienced waiting, while others completed without delay. The method thus balances computational efficiency with accurate congestion-aware scheduling.

2 Question 2

2.1 Problem Description

You are given three reservoirs connected in a mesh topology by bidirectional valves. Each reservoir has an initial amount of water and a target amount it must reach.

Water transfers happen by opening a valve between two reservoirs. When a valve is open, water flows until:

- The destination reservoir is full, **or**
- The source reservoir reaches the safety threshold ($\geq 20\%$ of its total capacity).

At most one valve can be opened at a time. The total amount of water is conserved.

Task: Determine the sequence of valve operations that transforms the initial distribution into the target distribution, while minimizing the number of valve openings.

2.2 Algorithm

This algorithm models the problem of transferring water between three reservoirs as a state-space search. Each state is represented by the current water levels in the three reservoirs, and valid successor states are generated by simulating the opening of a valve between two reservoirs. Transfers are constrained so that the source reservoir never drops below 20% of its capacity and the destination never exceeds its maximum capacity. The algorithm checks whether the current state matches the target state within a small tolerance, ensuring numerical stability. Three search strategies are implemented to find a solution. Breadth-First Search (BFS) explores states level by level and guarantees finding the solution with the minimum number of valve operations, though it can consume a lot of memory. Depth-First Search (DFS), on the other hand, explores states deeply before backtracking and may reach a solution quickly but does not guarantee the minimum sequence of steps and risks going too deep. A* search combines the actual cost of reaching a state with an admissible heuristic that estimates the number of remaining transfers based on volume differences. This makes it more efficient than BFS while still ensuring an optimal solution. Finally, the program reconstructs and displays the sequence of valve operations for each algorithm, showing the intermediate water levels after every step along with the total number of operations.

2.3 Results

Input

```
Capacities: 8.0 5.0 3.0
Initial:    8.0 0.0 0.0
Target:     2.4 5.0 0.6
```

BFS Solution

```
Start state: [8.0, 0.0, 0.0]
Open valve (1->2) -> (3.0, 5.0, 0.0)
Open valve (1->3) -> (1.6, 5.0, 1.4)
Open valve (3->1) -> (2.4, 5.0, 0.6)
Number of valve operations = 3
```

DFS Solution

```
Start state: [8.0, 0.0, 0.0]
Open valve (1->3) -> (5.0, 0.0, 3.0)
Open valve (3->2) -> (5.0, 2.4, 0.6)
Open valve (1->2) -> (2.4, 5.0, 0.6)
Number of valve operations = 3
```

A* Solution

```
Start state: [8.0, 0.0, 0.0]
Open valve (1->2) -> (3.0, 5.0, 0.0)
Open valve (1->3) -> (1.6, 5.0, 1.4)
Open valve (3->1) -> (2.4, 5.0, 0.6)
Number of valve operations = 3
```

3 Question 3

3.1 Problem Statement

Problem Statement

Consider the game of Tic-Tac-Toe where **X is the MAX player**. Given the game board below where it is X's turn to play next, we need to:

1. Show the complete game tree
2. Apply the Minimax algorithm to find the optimal move
3. Apply Alpha-Beta pruning for optimization
4. Compare the efficiency of both approaches

Initial Game State:

—	X	—
⁰ O	1 —	2 —
3 X	4 —	5 O

6 7 8

Available moves for X: Positions {0, 2, 4, 5, 7}

Evaluation Function

The evaluation function for any board state s is defined as:

$$\text{Evaluation}(s) = 8X_3(s) + 3X_2(s) + X_1(s) \quad (1)$$

$$- (8O_3(s) + 3O_2(s) + O_1(s)) \quad (2)$$

Where:

- $X_n(s)$: Number of lines with exactly n X's and no O's
- $O_n(s)$: Number of lines with exactly n O's and no X's
- Lines include: 3 rows + 3 columns + 2 diagonals = 8 total lines

3.2 Solution Approach

Solution Approach

We implement two algorithms to solve this problem:

1. **Minimax Algorithm:** Complete search of game tree
2. **Alpha-Beta Pruning:** Optimized search with branch elimination

Both algorithms follow the same principle:

- **MAX player (X)** tries to maximize the evaluation score
- **MIN player (O)** tries to minimize the evaluation score
- Search continues until terminal states are reached

3.3 Results

The following table summarizes the performance of both algorithms:

Table 1: Algorithm Performance Comparison

lightblue Move	Position	Minimax Value	MM Nodes	AB Nodes
(0,0)	0	0	47	20
lightgreen (0,2)	2	5	35	17
lightgreen (1,1)	4	5	39	31
(1,2)	5	0	61	42
(2,1)	7	-8	43	34

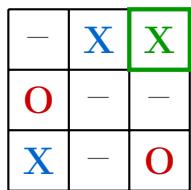
Detailed Move Analysis

Optimal Moves Analysis

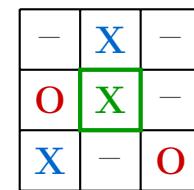
The algorithm identified **two optimal moves** with equal value:

- **Position (0,2) - Top Right Corner: Value = 5**
- **Position (1,1) - Center: Value = 5**

Let's analyze why these moves are optimal:



(a) Move (0,2) - Value: 5



(b) Move (1,1) - Value: 5

Figure 1: Optimal Move Positions

Strategic Analysis

Why Position (0,2) is Optimal:

- Creates a potential winning line in the top row
- Blocks O from using the top-right corner
- Maintains multiple threat lines for future moves

Why Position (1,1) is Optimal:

- Controls the center - most strategic position
- Creates threats in multiple directions (diagonal, row, column)
- Forces O into a defensive position

Efficiency Comparison

Table 2: Overall Algorithm Efficiency

lightblue Algorithm	Total Nodes	Pruning Events	Efficiency Gain
Minimax	225	0	—
Alpha-Beta	144	38	36% reduction

Key Performance Insights:

1. **Node Reduction:** Alpha-Beta pruning reduced node exploration by 36%
2. **Pruning Effectiveness:** 38 pruning events eliminated unnecessary branches
3. **Optimal Solution:** Both algorithms found the same optimal moves
4. **Computational Efficiency:** Alpha-Beta is significantly more efficient for larger game trees

Game Tree Visualization

The following diagram shows a partial game tree for the optimal move analysis:

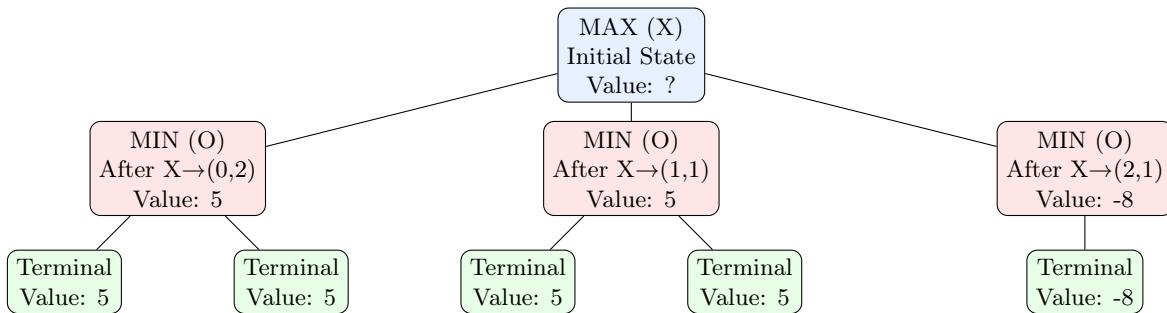


Figure 2: Simplified Game Tree Structure

3.4 Conclusion

The Minimax algorithm with Alpha-Beta pruning optimization successfully solved the Tic-Tac-Toe problem, demonstrating the effectiveness of game theory approaches in artificial intelligence.

4 Question 4

4.1 Problem Overview

The chip placement problem involves optimizing the positions of 8 rectangular chips on a 10×10 grid to minimize a conflict score. Each chip is constrained to a specific row (fixed y-coordinate) and can only move horizontally. The conflict score combines wiring costs between connected chips and overlap penalties.

Objective Function:

$$\text{Conflict}(P) = \sum_{(c_i, c_j) \in \text{Connections}} \text{WiringCost}(c_i, c_j) + \sum_{(c_i, c_j) \in \text{Overlaps}} \text{OverlapBlocks}(c_i, c_j) \quad (3)$$

Initial Configuration: 8 chips with dimensions and positions as shown in Table 1, connected via 10 netlist connections: $\{(1,2), (2,6), (2,3), (3,5), (4,5), (5,6), (1,6), (7,4), (7,2), (8,5)\}$.

4.2 Integer Descent Implementation

The algorithm iteratively evaluates moving each chip one position left or right, selects the move that minimally reduces the conflict score, and repeats until no improvement is possible.

Algorithm Steps:

1. Calculate current conflict score
2. For each chip, evaluate moves: $x_i \leftarrow x_i - 1$ and $x_i \leftarrow x_i + 1$
3. Select the move with minimum conflict score
4. Update chip position and repeat until convergence

4.3 Results

Optimization Progress

The algorithm converged in **6 iterations**, reducing the conflict score from 40 to 29 (27.5% improvement).

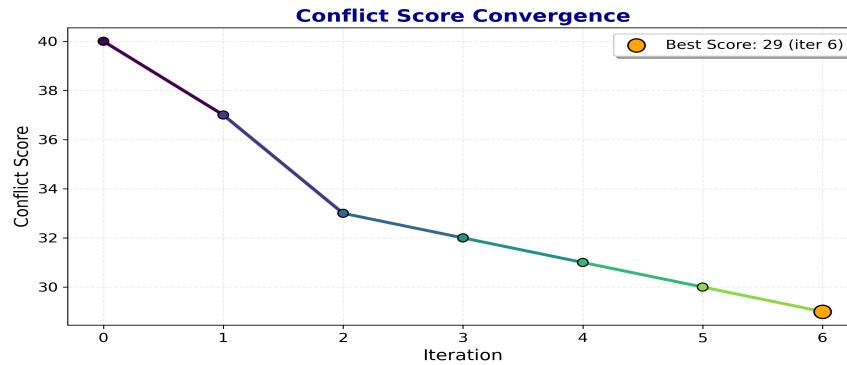


Figure 3: Conflict score convergence over iterations

Initial vs Final Configurations

10x10 Grid (Row 9 → 0)



Figure 4: Initial configuration (Conflict = 40)

10x10 Grid (Row 9 → 0)



Figure 5: Final optimized configuration (Conflict = 29)

Final Chip Positions and Results

Table 3: Final chip positions after optimization

Chip	Initial (x,y)	Final (x,y)	Dimensions	Movement
c_1	(0,0)	(1,0)	2×4	+1
c_2	(1,1)	(0,1)	2×5	-1
c_3	(1,0)	(3,0)	1×3	+2
c_4	(2,4)	(2,4)	2×5	0
c_5	(3,3)	(4,3)	2×4	+1
c_6	(2,2)	(3,2)	1×4	+1
c_7	(0,5)	(0,5)	2×5	0
c_8	(4,6)	(4,6)	1×3	0

Table 4: Optimization summary

Metric	Value
Initial Conflict Score	40
Final Conflict Score	29
Improvement	27.5%
Iterations to Convergence	6
Final Wiring Cost	20
Final Overlap Cost	9
Chips Moved	5 out of 8
Maximum Movement	2 units

4.4 Analysis and Conclusion

Key Achievements:

- **Rapid Convergence:** Local optimum reached in only 6 iterations, demonstrating algorithm efficiency
- **Significant Improvement:** 27.5% reduction in conflict score (40→29)

- **Balanced Optimization:** Effective trade-off between wiring costs (20) and overlap penalties (9)
- **Minimal Disruption:** Only 5 of 8 chips moved, with maximum displacement of 2 grid units
- **Feasible Solution:** All final positions respect grid boundaries and row constraints

Algorithm Performance: The integer descent method proved highly effective for this constrained optimization problem. The rapid convergence suggests the optimization landscape has favorable properties, with the initial configuration relatively close to a good local optimum.

Final Discrete Conflict Score: 29

The results validate that simple local search methods can achieve substantial improvements in chip placement problems, making them suitable for real-time optimization scenarios in electronic design automation where computational efficiency is critical.