Design and Analysis of Algorithms

Project Report — Detailed Technical Analysis

TracksGame

Serpentine Path Solver

Bidirectional A* · Divide & Conquer · Recursive Backtracking · Merge Sort

"A complete technical walkthrough of all algorithms, data structures, and design decisions — written to enable confident explanation to an examiner."

Team Members

Name	Roll Number
Niranjan Reddy	cb.sc.u4cse24735
Naveen Velan	cb.sc.u4cse24734
GS Mithesh	cb.sc.u4cse24715
Sidambarisvar Balamurugan	cb.sc.u4cse24751

Department of Computer Science and Engineering

2026

0. Contents

1. The Big Picture — What Is This Game?

TracksGame is a **constraint-satisfaction puzzle game** built in Java Swing. The player must draw a continuous railway track from point **A** to point **B** on a grid, subject to strict rules: every row and column on the grid has a number that tells the player exactly how many track cells must pass through it — not more, not less. Think of it like Sudoku, but instead of filling numbers, you are drawing a path.

> The game has two completely separate concerns that must never be confused:
>
> 1. **The Solver** — finds the solution path internally using two simultaneous searches running in a background thread.
> 2. **The Visual** — only ever draws ONE green line on screen (the player's built path). The animation is a *replay* of the solver's already-computed answer, not the solver itself running.

This distinction is the most important conceptual point in the entire project. The bidirectional nature of the solver exists entirely within its internal computation. After the solver finishes, a single pre-computed list of cells is returned, and the animation simply walks through that list one step at a time.

1.1 Code Structure Overview

The entire program is one Java file with the following class hierarchy:

```
TracksGame (extends JFrame)          // the application window
|
|-- enum GameMode                    // PATTERN1..5, RANDOM
|
|-- static class Cell                // one square on the grid
|
|-- class GamePanel (JPanel)         // grid, game logic, solver
|    |-- class FState                // one node in the FORWARD
     search
|    `-- class BState                // one node in the BACKWARD
     search
|
`-- class SidePanel (JPanel)         // buttons and status labels
```

Listing 1: High-level class structure of TracksGame

2. The Cell — The Atom of the Game

Every square on the grid is represented by a `Cell` object:

```
static class Cell {
    int r, c;               // row and column     its address on
        the grid
    boolean hasTrack;       // is this cell part of the drawn
       path?
    boolean isCrossed;      // has the player marked it with an
       X?
    boolean isHint;         // is it currently highlighted
       yellow?
    Cell trackParent;       // which cell came BEFORE this in
       the path?
}
```

Listing 2: The Cell class — the fundamental data unit

2.1 Why `trackParent` Is the Most Important Field

`trackParent` turns the entire grid into a **singly linked list**. Every time a move is made:

```
private void makeMove(int r, int c) {
    Cell nextCell = grid[r][c];
    nextCell.hasTrack = true;
    nextCell.trackParent = currentHead;  // point back to
       previous cell
    currentHead = nextCell;              // advance the head
}
```

Listing 3: makeMove — how the path linked list is built

So if the path is A → (1,2) → (1,3) → (1,4), the linked list is:

$$\texttt{(1,4)} \leftarrow \texttt{(1,3)} \leftarrow \texttt{(1,2)} \leftarrow \texttt{A} \leftarrow \texttt{null}$$

To reconstruct the entire path, simply follow `trackParent` pointers from `currentHead` all the way back to `null`. This mechanism is used by both the renderer (to draw the green line) and the solver (for path reconstruction after finding the meeting point).

3. The Clues System — Constraint Definition

When a puzzle is generated, two integer arrays are computed:

```
int[] rowClues;    // rowClues[i] = exact number of track cells
    in row i
int[] colClues;    // colClues[i] = exact number of track cells
    in col i

private void calculateCluesFromPath(List<Cell> path) {
    rowClues = new int[gridSize];
```

```
6       colClues = new int[gridSize];
7       for (Cell c : path) {
8           rowClues[c.r]++;  // count how many path cells are in
                each row
9           colClues[c.c]++;  // count how many path cells are in
                each col
10      }
11  }
```

Listing 4: Clue arrays computed from the solution path

These clues are displayed on the grid edges. The player must satisfy ALL of them simultaneously — this is what makes TracksGame a **constraint satisfaction problem**. The clue for each row/column is not just a hint; it is a hard requirement that must be exactly met.

4. Puzzle Generation — Recursive Backtracker

4.1 Why Is This Needed?

A random path from A to B will not work because it might branch or revisit cells. A specially designed recursive algorithm is needed to guarantee:

- The path is a single, clean, non-branching line.
- The puzzle is guaranteed to have a valid solution.
- Row and column clues can be uniquely computed from the path.

4.2 Algorithm — Step by Step

Step 1 — Choose a random inner starting cell:

```
1 int startR = rand.nextInt(gridSize - 2) + 1;  // not on the
      edge
2 int startC = rand.nextInt(gridSize - 2) + 1;
```

Listing 5: Choosing a starting cell away from edges

Starting away from the grid edges gives more room to grow in all four directions, producing more complex winding paths.

Step 2 — Recursively extend the path with the no-branching rule:

```
1 private void generatePathRecursive(Cell current, boolean[][]
      visited,
2           List<Cell> currentPath, List<Cell> bestPath,
3           int target, Random rand) {
4
5       // Always track the longest path found so far
```

```
6       if (currentPath.size() > bestPath.size()) {
7           bestPath.clear();
8           bestPath.addAll(currentPath);
9       }
10      if (bestPath.size() >= target) return;  // stop if long
            enough
11
12      List<Cell> neighbors = getNeighbors(current);
13      Collections.shuffle(neighbors, rand);   // randomise
            direction order
14
15      for (Cell n : neighbors) {
16          // THE NO-BRANCHING RULE: neighbour must have <= 1
                visited neighbour
17          if (!visited[n.r][n.c] && countNeighbors(n, visited) <=
                1) {
18              visited[n.r][n.c] = true;
19              currentPath.add(n);
20              generatePathRecursive(n, visited, currentPath,
21                                    bestPath, target, rand);
22              if (bestPath.size() >= target) return;
23
24              // BACKTRACK: undo the step and try another
                    direction
25              currentPath.remove(currentPath.size() - 1);
26              visited[n.r][n.c] = false;
27          }
28      }
29  }
```

Listing 6: The core recursive backtracker with no-branching guarantee

4.3 The No-Branching Rule Explained

The condition `countNeighbors(n, visited) <= 1` is the heart of correctness. Before stepping into cell `n`, we count how many of `n`'s neighbours are already visited. If 2 or more are visited, stepping into `n` would create a scenario where the path could branch or loop, violating the single-path requirement. So we skip it.

Step 3 — Backtrack when stuck: If no valid neighbours exist, the recursion returns, and the code removes the last cell and restores its visited flag, effectively trying a different direction.

Why 50% grid coverage? The target path length is `(gridSize * gridSize) * 0.50`. A path that is too short is trivially easy. Too long and the solver becomes extremely slow due to exponential complexity. 50% gives a good balance of puzzle difficulty versus solver feasibility.

5. Game Modes and Deterministic Seeds

```
switch(mode) {
    case PATTERN2: seed = 12399L;  break;
    case PATTERN3: seed = 44455L;  break;
    case PATTERN4: seed = 99887L;  break;
    case PATTERN5: seed = 101010L; break;
    default:       seed = System.currentTimeMillis(); break; // RANDOM
}
```

Listing 7: Fixed seeds ensure the same puzzle every run

Fixed patterns pass a constant `long` seed to `Random`, so the shuffle order inside `generatePathRecursive` is always identical, producing the exact same puzzle. The `RANDOM` mode uses the current system time, giving a different puzzle every run.

6. Player Interaction and Move Validation

6.1 Mouse Events

- **Left click** on an adjacent cell: attempt to extend the path.
- **Right click** on any empty cell: toggle an X marker (`isCrossed`), a planning aid like pencil marks in Sudoku.

6.2 Move Validation — Four Independent Checks

```
public boolean isValidMove(int r, int c) {
    Cell target = grid[r][c];

    // Check 1: Cell must not already be visited or crossed out
    if (target.hasTrack || target.isCrossed) return false;

    // Check 2: Must be exactly 1 step away (orthogonal adjacency only)
    if (Math.abs(r - currentHead.r)
      + Math.abs(c - currentHead.c) != 1) return false;

    // Check 3: Row budget must not be exceeded
    if (countTrackInRow(r) >= rowClues[r]) return false;

    // Check 4: Column budget must not be exceeded
    if (countTrackInCol(c) >= colClues[c]) return false;

    return true;
}
```

Listing 8: isValidMove — four constraints that must ALL pass

Important limitation: `isValidMove` only catches immediate violations. A move can pass all four checks and still lead to a dead end further down the line. This is precisely why the solver exists — to look ahead and determine whether a complete valid solution still remains reachable.

7. The Bidirectional A* Solver — Complete Deep Dive

7.1 Why Not a Simple DFS or BFS?

A regular DFS from A to B on a constrained grid has exponential complexity. For a 10×10 grid with a path of length 50, the naive search might explore $O(b^{50})$ states where $b \approx 3$ (average branching factor after constraint pruning). This is completely infeasible in real time.

7.2 The Divide & Conquer Insight

Split the problem into two halves:

- **Forward search:** Explore from the current path head toward B.
- **Backward search:** Explore from B toward the current path head.
- Let both searches run simultaneously and **meet in the middle**.

If the full path has length L, each search finds a path of length $L/2$. Since complexity is exponential:

$$\text{Standard search: } O(b^L) \quad \longrightarrow \quad \text{Bidirectional: } O(b^{L/2}) = O\left(\sqrt{b^L}\right)$$

This reduces the search space to its **square root** — an enormous practical improvement.

7.3 The 50/50 Alternation

```
boolean expandForward = false;  // flips every iteration

while (!fOpen.isEmpty() && !bOpen.isEmpty()) {
    if (System.currentTimeMillis() - startTime > timeLimitMs)
        return null;
    if (++it > maxIterations) break;

    expandForward = !expandForward;  // F, B, F, B, F, B...

```

```
    if (expandForward) {
        // expand one forward state
    } else {
        // expand one backward state
    }
}
```

Listing 9: Strict 50/50 alternation between forward and backward

The old version expanded whichever queue was smaller. This version alternates strictly every iteration — ensuring perfectly balanced exploration and guaranteeing that both searches advance at the same rate, maximising the meet-in-the-middle benefit.

7.4 State Representation — Why Position Alone Is Not Enough

Each node in the search is NOT just a grid position. It is a **complete snapshot** of the search's progress:

```
class FState {
    Cell pos;          // WHERE am I on the grid?
    int g;             // HOW MANY steps have I taken from the
       start?
    int f;             // g + heuristic (priority in the queue)
    int[] rowUsed;     // HOW MANY cells placed in EACH ROW so
       far?
    int[] colUsed;     // HOW MANY cells placed in EACH COLUMN so
        far?
    BitSet visited;    // WHICH CELLS have I already visited?
    FState parent;     // pointer back for path reconstruction
}
```

Listing 10: FState — a full forward search state

Consider two paths that both arrive at cell (3,4):

- Path 1 arrived via rows 1,2,3 — consuming those rows' budgets.
- Path 2 arrived via different rows — consuming different budgets.

These are fundamentally different states even though they are at the same cell. If we only tracked position, we would incorrectly prune one of them, potentially missing the only valid solution. The constraint arrays make every state unique.

`BState` is identical in structure but used by the backward search, with `parent` pointing toward B instead of toward the start.

7.5 The Closed Set Key — The Critical Fix

```
private String keyState(int cellId, int[] rowUsed, int[]
    colUsed) {
     StringBuilder sb = new StringBuilder(16 + gridSize * 6);
     sb.append(cellId).append('|');
     for (int i = 0; i < gridSize; i++) sb.append(rowUsed[i]).
        append(',');
     sb.append('|');
     for (int i = 0; i < gridSize; i++) sb.append(colUsed[i]).
        append(',');
     return sb.toString();
     // Example: "20|1,1,1,0,0,0,0,0,|2,0,1,0,0,0,0,0,"
}
```

Listing 11: State key encodes cell AND full constraint usage

A state is only considered "already explored" if the SAME cell was reached with the EXACT same row and column usage. Two paths arriving at the same cell via different routes produce different keys and are both explored. Using only the cell index as the key would silently prune valid paths and produce incorrect or missing solutions.

7.6 The Manhattan Distance Heuristic

```
private int manhattan(Cell a, Cell b) {
     return Math.abs(a.r - b.r) + Math.abs(a.c - b.c);
}
// f-value = g + manhattan(current, goal)
// States with lower f-value are expanded first (PriorityQueue)
```

Listing 12: Admissible heuristic for A* priority

The heuristic is **admissible** — it never overestimates the actual remaining distance (since the shortest path between two points on a grid IS the Manhattan distance). This property guarantees A* finds the optimal solution. States geometrically closer to the goal are prioritised, dramatically reducing unnecessary exploration.

7.7 Initialising Both Searches

Forward search starts at the current head, pre-loaded with the counts of already-placed track cells:

```
// Walk the existing placed path to get current row/col usage
Cell t = forwardStart;
while (t != null) {
     int id = idx(t.r, t.c);
     if (!fixed.get(id)) {
          fixed.set(id);      // mark as fixed (backward cannot
             enter these)
          fRow0[t.r]++;
          fCol0[t.c]++;
     }
```

```
    t = t.trackParent;      // follow linked list back to start
}

FState fStart = new FState(
    forwardStart,                          // position = current
        head
    0,                                     // g = 0 steps from here
    manhattan(forwardStart, endNode),      // heuristic to B
    fRow0, fCol0,                          // existing constraint
        usage
    (BitSet) fixed.clone(),                // already visited cells
    null                                   // no parent
);
```

Listing 13: Forward search initialisation from current game state

Backward search starts at B with only B itself counted:

```
BitSet bVis0 = new BitSet(gridSize * gridSize);
bVis0.set(idx(endNode.r, endNode.c));  // only B is visited
int[] bRow0 = new int[gridSize];
int[] bCol0 = new int[gridSize];
bRow0[endNode.r] = 1;    // B counts as 1 cell in its row
bCol0[endNode.c] = 1;    // B counts as 1 cell in its column

BState bStart = new BState(
    endNode,                               // position = B
    0,                                     // g = 0 steps from B
    manhattan(endNode, forwardStart),      // heuristic to current
        head
    bRow0, bCol0, bVis0, null
);
```

Listing 14: Backward search initialisation from node B

7.8 The Frontier Registries — How Searches Find Each Other

```
// fAt.get(id) = all forward states that have reached cell id
// bAt.get(id) = all backward states that have reached cell id
ArrayList<ArrayList<FState>> fAt = new ArrayList<>();
ArrayList<ArrayList<BState>> bAt = new ArrayList<>();
for (int i = 0; i < gridSize * gridSize; i++) {
    fAt.add(new ArrayList<>());
    bAt.add(new ArrayList<>());
}
```

Listing 15: Per-cell state registries for efficient meeting detection

Every time a new state is created, it is added to the appropriate registry: `fAt.get(nid).add(ns)` for forward, `bAt.get(nid).add(ns)` for backward. When one search expands a cell, it checks the other search's registry for that cell. If the other search has been there, a potential meeting point is found.

7.9 The Meeting Point Check — Conquer Phase

When the forward search expands a cell, it immediately checks whether the backward search has also visited that cell:

```
int meetId = idx(cur.pos.r, cur.pos.c);
for (BState other : bAt.get(meetId)) {
    // Budget check: combined path must not exceed remaining budget
    if (cur.g + other.g > remainingBudget) continue;

    // Condition 1: paths must not overlap (except at meeting cell)
    if (!disjointExceptMeet(cur.visited, other.visited, cur.pos))
        continue;

    // Condition 2: combined row/col usage must EXACTLY match clues
    if (!combineFeasible(cur.rowUsed, cur.colUsed,
                         other.rowUsed, other.colUsed, cur.pos))
        continue;

    // Both conditions pass -> SOLUTION FOUND
    return reconstructBidirectional(cur, other, forwardStart);
}
```

Listing 16: Meeting point detection and validation

Condition 1 — Disjoint paths:

```
private boolean disjointExceptMeet(BitSet fVis, BitSet bVis,
    Cell meet) {
    BitSet tmp = (BitSet) fVis.clone();
    tmp.and(bVis);  // intersection: cells visited by BOTH searches
    int meetIndex = idx(meet.r, meet.c);
    if (tmp.isEmpty()) return true;
    // ONLY the meeting cell itself is allowed in both
    return tmp.cardinality() == 1 && tmp.get(meetIndex);
}
```

Listing 17: Ensuring the two path halves do not overlap

The `and` operation on two `BitSets` finds their intersection — cells visited by both searches. The only cell allowed to appear in both is the meeting point itself. If any other cell appears in both, the combined path would visit that cell twice, which violates the no-revisit rule.

Condition 2 — Exact constraint satisfaction:

```
private boolean combineFeasible(int[] fRow, int[] fCol,
                                int[] bRow, int[] bCol, Cell
                                    meet) {
```

```
    for (int i = 0; i < gridSize; i++) {
        // meeting cell is counted in BOTH fRow and bRow ->
           subtract once
        int rTotal = fRow[i] + bRow[i] - ((meet.r == i) ? 1 :
           0);
        if (rTotal != rowClues[i]) return false;  // must be
           EXACT

        int cTotal = fCol[i] + bCol[i] - ((meet.c == i) ? 1 :
           0);
        if (cTotal != colClues[i]) return false;  // must be
           EXACT
    }
    return true;
}
```

Listing 18: Verifying combined row and column clues are exactly met

When combining, the meeting cell is counted in both `fRow` and `bRow` (since both searches include it), so 1 is subtracted for that row and column. The combined totals must exactly equal every clue — not approximately, but precisely. This is what makes the puzzle a hard constraint problem rather than a simple path-finding problem.

7.10 Expanding Neighbours — Forward Search

```
for (Cell nxt : getNeighbors(cur.pos)) {
    int nid = idx(nxt.r, nxt.c);

    // Skip: player blocked, already in this path
    if (grid[nxt.r][nxt.c].isCrossed) continue;
    if (cur.visited.get(nid)) continue;

    // Update constraint counts for this new cell
    int[] nr = Arrays.copyOf(cur.rowUsed, gridSize); // MUST
       copy
    int[] nc = Arrays.copyOf(cur.colUsed, gridSize); // MUST
       copy
    nr[nxt.r]++;
    nc[nxt.c]++;

    // Prune immediately if any clue would be exceeded
    if (nr[nxt.r] > rowClues[nxt.r]) continue;
    if (nc[nxt.c] > colClues[nxt.c]) continue;

    // Clone visited set for this branch (MUST clone     not
       shared)
    BitSet nVis = (BitSet) cur.visited.clone();
    nVis.set(nid);

    int ng = cur.g + 1;
    int nf = ng + manhattan(nxt, endNode);  // A* priority
```

```
24
25      FState ns = new FState(nxt, ng, nf, nr, nc, nVis, cur);
26      fAt.get(nid).add(ns);
27      fOpen.add(ns);
28  }
```

Listing 19: Generating new forward states with constraint propagation

Why `Arrays.copyOf` and `BitSet.clone()` are critical: Without copying, all states spawned from the same parent would share the same `rowUsed`, `colUsed`, and `visited` arrays. Modifying one branch would corrupt all others. Each state must have its own independent copy of these arrays.

The backward search expansion is identical except for one extra check: `if (fixed.get(nid)) continue` — the backward search must not enter cells that are already part of the existing drawn path.

7.11 Path Reconstruction

Once the meeting point is found, the complete solution is assembled from both parent chains:

```
1  private List<Cell> reconstructBidirectional(FState fMeet,
2          BState bMeet, Cell forwardStart) {
3
4      // Trace forward chain: fMeet -> ... -> forwardStart (then
           reverse)
5      ArrayList<Cell> left = new ArrayList<>();
6      FState ft = fMeet;
7      while (ft != null) { left.add(ft.pos); ft = ft.parent; }
8      Collections.reverse(left);
9      // left now: forwardStart -> ... -> meetingPoint
10
11     // Trace backward chain: skip meeting cell (already in left
           )
12     ArrayList<Cell> right = new ArrayList<>();
13     BState bt = bMeet.parent;  // .parent skips the meeting
           cell
14     while (bt != null) { right.add(bt.pos); bt = bt.parent; }
15     // right now: cell_after_meet -> ... -> B
16
17     left.addAll(right);
18     // combined: forwardStart -> ... -> meetPoint -> ... -> B
19
20     // Remove the starting cell (it is already on the board)
21     if (!left.isEmpty() && left.get(0) == forwardStart)
22         left.remove(0);
23
24     return left;  // list of cells to place, in order
25 }
```

Listing 20: Reconstructing the full path from both parent chains

8. Visual Highlights — Seeing the Bidirectional Search

This version adds two `BitSets` that record which cells were visited by each search direction:

```
private BitSet forwardExplored  = null;  // painted LIGHT BLUE
private BitSet backwardExplored = null;  // painted LIGHT PINK

private void markExploredForward(Cell c) {
    forwardExplored.set(idx(c.r, c.c));
}
private void markExploredBackward(Cell c) {
    backwardExplored.set(idx(c.r, c.c));
}
```

Listing 21: Visual exploration tracking fields

These are called inside the main loop every time a state is expanded:

```
// Inside forward expansion:
markExploredForward(cur.pos);
if (it % 1200 == 0) repaint();  // throttled repaint to avoid
    lag

// Inside backward expansion:
markExploredBackward(cur.pos);
if (it % 1200 == 0) repaint();
```

Listing 22: Marking exploration inside the main solver loop

The `it % 1200` throttle prevents the screen from being repainted thousands of times per second, which would cause severe lag. Every 1200 iterations, one repaint is triggered, giving a smooth visual update.

The colours are rendered in `paintComponent`:

```
int id = idx(r, c);

// Light blue: explored by forward search from A
if (forwardExplored != null && forwardExplored.get(id)) {
    g2.setColor(new Color(150, 200, 255, 120)); // semi-
       transparent
    g2.fillRect(x + 1, y + 1, CELL_SIZE - 2, CELL_SIZE - 2);
}

// Light pink: explored by backward search from B
if (backwardExplored != null && backwardExplored.get(id)) {
```

```
11      g2.setColor(new Color(255, 170, 190, 120)); // semi-
            transparent
12      g2.fillRect(x + 1, y + 1, CELL_SIZE - 2, CELL_SIZE - 2);
13  }
```

Listing 23: Painting forward (blue) and backward (pink) explored cells

- **Blue cells:** Explored by the forward search spreading from A.
- **Pink cells:** Explored by the backward search spreading from B.
- **Overlap region:** Where both colours meet — this is approximately where the meeting point was found. The two searches truly divided the problem in half.

After auto-solve completes, these highlights remain on screen as permanent visual evidence that two independent searches ran simultaneously.

8.1 Smart Time Limits

```
1  private long getSmartTimeLimitMs() {
2      if (gridSize <= 8) return 7000;    // 7 seconds for 8x8
3      if (gridSize == 9) return 9000;    // 9 seconds for 9x9
4      return 18000;                       // 18 seconds for 10x10
5  }
```

Listing 24: Grid-size-aware time limits for the solver

Larger grids are exponentially harder. If the solver exceeds the time limit, it returns `null` and `gracefulSolverExit()` is called, showing a user-friendly message instead of hanging indefinitely.

9. Merge Sort — Move Ordering Optimisation

Before testing candidate moves, they are sorted by Manhattan distance to B using a custom implementation of Merge Sort:

```
1  private void mergeSort(List<Cell> list, Comparator<Cell>
       comparator) {
2      if (list.size() <= 1) return;              // base case
3      int mid = list.size() / 2;
4      List<Cell> left  = new ArrayList<>(list.subList(0, mid));
5      List<Cell> right = new ArrayList<>(list.subList(mid, list.
           size()));
6      mergeSort(left, comparator);               // sort left half
7      mergeSort(right, comparator);              // sort right half
8      merge(list, left, right, comparator);      // merge sorted
           halves
```

```
}

private void merge(List<Cell> result, List<Cell> left,
                   List<Cell> right, Comparator<Cell> comp) {
    int i = 0, j = 0, k = 0;
    while (i < left.size() && j < right.size()) {
        if (comp.compare(left.get(i), right.get(j)) <= 0)
            result.set(k++, left.get(i++));
        else
            result.set(k++, right.get(j++));
    }
    while (i < left.size())  result.set(k++, left.get(i++));
    while (j < right.size()) result.set(k++, right.get(j++));
}
```

Listing 25: Custom Merge Sort implementation for move ordering

The comparator used in both the computer turn and hint system:

```
mergeSort(validMoves, (a, b) -> {
    int distA = Math.abs(a.r - endNode.r) + Math.abs(a.c -
       endNode.c);
    int distB = Math.abs(b.r - endNode.r) + Math.abs(b.c -
       endNode.c);
    return Integer.compare(distA, distB);
});
// After sorting: validMoves[0] is the cell closest to B
```

Listing 26: Sorting moves by distance to endpoint B

Why this matters: By trying the move closest to B first, the solver encounters the meeting point earlier in its forward search, reducing unnecessary exploration. This is a pruning optimisation — it does not affect correctness but significantly improves average-case speed.

Time complexity: $O(n \log n)$ where n is the number of valid moves (at most 4 for orthogonal movement).

10. The Computer's Turn

```
private void computerTurn() {
    // Step 1: Gather all valid moves from current head
    List<Cell> validMoves = new ArrayList<>();
    for (Cell neighbor : getNeighbors(currentHead))
        if (isValidMove(neighbor.r, neighbor.c))
            validMoves.add(neighbor);

    // Step 2: Sort by Manhattan distance to B (Merge Sort)
    mergeSort(validMoves, (a, b) -> Integer.compare(
        manhattan(a, endNode), manhattan(b, endNode)));
```

```
11
12      // Step 3: Test each candidate      pick first that keeps
            puzzle solvable
13      Cell bestMove = null;
14      for (Cell candidate : validMoves) {
15          Cell orig = candidate.trackParent;
16          candidate.trackParent = currentHead;    // SIMULATE the
                move
17
18          if (runBidirectionalAStar(candidate) != null) {
19              bestMove = candidate;               // solution
                    exists -> pick it
20              candidate.trackParent = orig;
21              break;
22          }
23          candidate.trackParent = orig;           // undo
                simulation
24      }
25
26      // Step 4: Make the move (or graceful exit if none found)
27      if (bestMove != null) makeMove(bestMove.r, bestMove.c);
28      else gracefulSolverExit("Solver Stuck", "No safe move found
            .");
29  }
```

Listing 27: Computer turn — greedy look-ahead using the D&C solver

Component	Algorithm	Role
Move ordering	Merge Sort	Sorts candidates by distance
Move selection	Greedy look-ahead	First feasible move
Feasibility check	Bidirectional A* (D&C)	Validator (not decision maker)

The D&C solver is called *as a subroutine* to validate moves, not as the move decision mechanism itself. A true D&C computer move would be: `solution = runBidirectionalAStar(currentHead); makeMove(solution.get(0));` — one solver call, take the first step of its result.

11. The Hint System

The hint system follows the same logic as the computer turn, but instead of making the move, it highlights the recommended cell in yellow:

```
1  public void showHint() {
2      // ... (same sorting and solver-testing loop as
           computerTurn) ...
3
```

```
    // Instead of makeMove(), just highlight the cell
    if (finalHint != null) {
        grid[finalHint.r][finalHint.c].isHint = true;  // yellow highlight
        sidePanel.updateStatus("Hint: " + finalHint);
    }
    repaint();
}
```

Listing 28: Hint system — same logic as computer turn but visual only

The `isHint` flag is rendered as a yellow fill:

```
if (grid[r][c].isHint) {
    g2.setColor(new Color(255, 255, 100, 200));  // yellow, semi-transparent
    g2.fillRect(x + 1, y + 1, CELL_SIZE - 2, CELL_SIZE - 2);
}
```

Listing 29: Rendering the hint highlight in paintComponent

12. Auto-Solve — The Critical Question Answered

12.1 The Timeline

```
public void autoSolve() {
    autoSolving = true;
    forwardExplored.clear();    // reset visual highlights
    backwardExplored.clear();

    new Thread(() -> {
        // PHASE 1: Solver runs completely in background
        // Blue/pink highlights appear during this phase
        List<Cell> solution = findSolution();  // <- ALL computation here

        SwingUtilities.invokeLater(() -> {
            // PHASE 2: Animate the pre-computed answer
            // Green line builds here     one step per 100ms
            solutionPath = solution;
            animateSolution(0);
        });
    }).start();
}

private void animateSolution(int index) {
    Cell nextCell = solutionPath.get(index);
    makeMove(nextCell.r, nextCell.c);   // place track (updates linked list)
```

```
23      repaint();                                  // draw green line
24      new Timer().schedule(new TimerTask() {
25          @Override public void run() {
26              SwingUtilities.invokeLater(() -> animateSolution(
                    index + 1));
27          }
28      }, 100);  // 100ms delay between steps
29  }
```

Listing 30: Auto-solve launches solver in a background thread

12.2 Why You Only See One Path Building from A to B

Phase 1: Solver Running (Background Thread)	Phase 2: Animation (Swing Thread)
Both searches run simultaneously. Blue cells spread from A.	Solver has already finished. Only the pre-computed answer is replayed.
Pink cells spread from B. They meet in the middle.	`makeMove()` called once per 100ms. Green line builds A → B.

The single most important point for the examiner: The bidirectional search happens entirely in Phase 1, invisibly, before the animation starts. What you see as auto-solve is NOT the solver — it is the solver's pre-computed answer being played back. There is only ever one path drawn on screen because only one path exists — the assembled solution from both halves. Think of it like a GPS: it considers multiple routes internally to find the best, then you drive only that one route.

13. Win Condition Verification

```
1  private void checkGameStatus() {
2      if (currentHead == endNode) {  // path has reached B
3          gameOver = true;
4          boolean valid = true;
5          for (int i = 0; i < gridSize; i++) {
6              // EVERY row AND column clue must be EXACTLY
                   satisfied
7              if (countTrackInRow(i) != rowClues[i] ||
8                  countTrackInCol(i) != colClues[i]) {
9                  valid = false;
10                 break;
11             }
12         }
13         if (valid) sidePanel.updateStatus("PUZZLE SOLVED!");
14         else       sidePanel.updateStatus("Constraints Failed")
               ;
```

```
        }
    }
```

Listing 31: Win condition — reaching B is not enough alone

Reaching B is necessary but not sufficient. A path that arrives at B while leaving some rows or columns under-filled is rejected. This forces the path to be as long and winding as the clues demand — you cannot "cut corners".

14. Complete System Flow

Game Start
→ `generatePathRecursive()` — Recursive Backtracker builds winding path
→ `calculateCluesFromPath()` — derives `rowClues[]` and `colClues[]`

Player Clicks (left)
→ `isValidMove()` — 4 constraint checks (visited, adjacent, row, col)
→ `makeMove()` — sets `hasTrack`, `trackParent`, advances head
→ `checkGameStatus()` — reached B? all clues exact?
→ *if not over:* trigger `computerTurn()` after 50ms

Computer Turn (background thread)
→ `mergeSort()` — order valid moves by Manhattan distance
→ For each candidate: `runBidirectionalAStar()` to test feasibility
→ Forward A*: expand from head, `markExploredForward()`
→ Backward A*: expand from B, `markExploredBackward()`
→ Meeting? `disjointExceptMeet()` + `combineFeasible()`
→ `reconstructBidirectional()` → complete path
→ `makeMove(bestMove)` — computer places its track

Auto-Solve
→ `runBidirectionalAStar()` runs fully in background thread
→ Blue/pink highlights update every 1200 iterations
→ Solution returned as list of cells
→ `animateSolution()` replays answer at 100ms per step
→ Green line builds from current head to B

Win Check
→ `currentHead == endNode` AND all row/col clues exactly satisfied

15. Complexity Summary

Component	Algorithm	Complexity	Why
Puzzle Generation	Recursive Backtracker	$O(N^2)$ avg	DFS on $N \times N$ grid
Move Ordering	Merge Sort	$O(n \log n)$	$n \leq 4$ moves
Solver	Bidirectional A*	$O(b^{L/2})$	Meet-in-the-middle
Solver (naive)	Single-direction DFS	$O(b^L)$	Full search space
Win Verification	Linear scan	$O(N)$	Check all clues

Table 1: Complexity of each major component

Where b = branching factor (≈ 3 after constraint pruning), L = solution path length, N = grid size.

The key insight is that $O(b^{L/2}) = O\left(\sqrt{b^L}\right)$ — the bidirectional approach reduces the exponent by half, giving a square-root improvement over the naive approach. For $L = 32$ and $b = 3$: $3^{32} \approx 1.85 \times 10^{15}$ versus $3^{16} \approx 43{,}046{,}721$ — a reduction by a factor of over 43 million.

16. Conclusion

TracksGame demonstrates the integration of four distinct algorithmic ideas:

1. **Recursive Backtracker** generates valid, solvable, non-branching puzzles with derived row/column constraints.
2. **Bidirectional A* (Meet-in-the-Middle)** solves the constraint satisfaction problem by dividing search into two halves meeting in the middle, reducing exponential complexity to its square root.
3. **Merge Sort** orders candidate moves by heuristic distance, improving pruning efficiency in both the hint and computer-turn systems.
4. **Constraint propagation** at every expansion step prunes states that would violate row or column clues before they are ever fully explored.

The visual highlights (blue from A, pink from B) provide direct, observable evidence of the divide-and-conquer strategy at work. The green animated line is not the solver — it is the solver's pre-computed answer being replayed. This separation between *computation* and *visualisation* is the key architectural decision that enables the smooth interactive experience.