FINAL MCP PROJECT

ALGORITHMS ANALYSIS AND DESIGN
Academic Year 2023-2024

JAIME HERNÁNDEZ DELGADO
48776654W

University of Alicante

Contents

1 Introduction

In this work, a branch and bound algorithm has been implemented to solve the minimum path problem in a map represented as a cost matrix. The objective is to find the least costly path from the initial position (0, 0) to the final position (n-1, m-1) of the map, where n and m are the dimensions of the map.

The algorithm uses a search strategy based on the expansion of promising nodes and the pruning of non-promising nodes. Various techniques have been employed to improve the efficiency of the algorithm, such as the use of pessimistic and optimistic bounds, and the implementation of pruning mechanisms.

2 Data Structures

2.1 Node

In branch and bound algorithms, a node represents a possible state within the problem's search space. Each node contains information that reflects a possible partial solution to the problem and is a potential point from which the algorithm can continue exploring more complete solutions.

In the implementation of this algorithm, a node is represented by a tuple containing the following elements:

- `row`: Row of the node on the map.
- `col`: Column of the node on the map.
- `cost`: Accumulated cost to reach the node.
- `heuristic`: Optimistic bound of the node.
- `pessimistic_bound`: Pessimistic bound of the node.
- `path`: Vector that stores the path of movements to reach the node.

2.2 Live Nodes List

A priority queue (`priority_queue`) is used to store the live nodes. Nodes in the priority queue are ordered according to their optimistic bound, so that the node with the lowest optimistic bound is at the top of the queue. This is crucial for the efficient operation of the branch and bound algorithm, as it allows us to explore the most promising nodes first.

The priority queue is a suitable data structure for this problem because it allows efficient extraction of the node with the lowest optimistic bound, which is essential for the branch and bound algorithm to function effectively.

2.2.1 Implementation

Below is the definition of the priority queue and how the nodes are used in the algorithm:

```
// Node type definition using a tuple
using Node = tuple<int, int, int, int, int, vector<int>>;

// Comparator class for the priority queue
class CompareNode {
public:
    bool operator()(const Node& a, const Node& b) {
        return get<3>(a) > get<3>(b);
    }
};
```

In this code, each node is represented as a tuple containing the row, column, cost, optimistic bound, pessimistic bound, and the vector of movements. The `CompareNode` class is used to compare nodes in the priority queue according to their optimistic bound.

2.2.2 Using the Priority Queue

The priority queue is used in the `mcp_bb` function to maintain and process live nodes. Below is a snippet of the code where the priority queue is initialized and used:

```
void mcp_bb(...) {
    priority_queue<Node, vector<Node>, CompareNode> pq;
    (...)

    while (!pq.empty()) {

        (Variables are initialized)

        if (row < 0 || row >= n || col < 0 || col >= m) {
            ++n_unfeasible;
            continue;
        }

        if (heuristic >= bestCost) {
            ++n_promising_but_discarded;
            continue;
        }

        if (row == n - 1 && col == m - 1) {
            ++n_leaf;
            ++n_explored;
            if (cost < bestCost) {
                bestCost = cost;
                bestMoves = get<5>(node);
                ++n_best_solution_updated_from_leafs;
            }
            continue;
        }

        if (cost >= best_costs[row][col]) {
            ++n_not_promising;
            continue;
        }

        visited[row][col] = true;
        ++n_explored;

        best_costs[row][col] = cost;
        generateChildren(node, mapa, n, m, pq, visited, cota_optimista,
        (...)
    }
}
```

In this code snippet, the priority queue (`pq`) is initialized and used to store and process nodes. Nodes are extracted from the priority queue according to their optimistic bound and expanded to generate child nodes, which are reinserted into the queue if they are promising.

2.2.3 Complexity

The complexity of operations with the priority queue (`priority_queue`) is $O(\log n)$ for insertions and extractions, where n is the number of nodes in the queue. This makes the priority queue an efficient data structure for this problem, as it effectively handles the exploration of the most promising nodes first.

2.2.4 Advantages of Using a Priority Queue

- **Efficiency in extracting the most promising node**: The priority queue allows extracting the node with the lowest optimistic bound in $O(\log n)$, which is crucial for maintaining the efficiency of the algorithm.
- **Handling of live nodes**: Facilitates the handling of live nodes and the exploration of nodes in the appropriate order according to their optimistic bound, improving the algorithm's effectiveness.

These characteristics make the priority queue a powerful tool for implementing the branch and bound algorithm in the minimum path problem.

2.3 Memoization

To improve the algorithm's performance, a memoization matrix is used to store the minimum costs found for each position on the map. This avoids recalculating costs for nodes that have already been processed, thus reducing execution time. The memoization technique is essential for problems involving repetitive calculations in overlapping subproblems, such as the minimum path in a map.

2.3.1 Implementation

Below is how memoization is implemented in the algorithm. First, the memoization matrix is initialized and then used during node exploration to store and query minimum costs:

```
vector<vector<int>> precalculate_pessimistic_bound(...) {
    vector<vector<int>> memo(n, vector<int>(m, MAX_COST));
    memo[n-1][m-1] = mapa[n-1][m-1];

    for (int i = n-1; i >= 0; --i) {
        for (int j = m-1; j >= 0; --j) {
            if (i == n-1 && j == m-1) continue;
            int min_cost = MAX_COST;
            for (int k = 0; k < 3; ++k) { // Only 3 moves: north, northeast, and east
                int ni = i + dx[k], nj = j + dy[k];
                if (ni >= 0 && ni < n && nj >= 0 && nj < m) {
                    min_cost = min(min_cost, memo[ni][nj]);
                }
            }
            memo[i][j] = min_cost + mapa[i][j];
        }
    }
    return memo;
}
```

In this code, the `memo` matrix is initialized with the maximum cost (`MAX_COST`) for all positions, except for the final position (n-1, m-1), which is initialized with the map cost at that position. Then, all positions on the map are iterated in reverse order, and for each position, the minimum cost to reach the final position is calculated considering only three possible moves: north, northeast, and east. The minimum cost is stored in the `memo` matrix.

2.3.2 Using Memoization

The memoization matrix is used in the main algorithm to check if a node has already been processed with a lower cost. If a lower cost is found in the memoization matrix, the node is not processed again, saving execution time.

```
if (node->cost >= best_costs[node->row][node->col]) {
    n_not_promising++;
    continue;
}
best_costs[node->row][node->col] = node->cost;
```

In this code snippet, it checks if the accumulated cost of the current node is greater than or equal to the best-known cost for that position in the memoization matrix (`best_costs`). If so, the node is discarded as non-promising. Otherwise, the best-known cost for that position is updated.

2.3.3 Complexity

The complexity of precalculating the pessimistic bound is $O(n \cdot m)$, where n is the number of rows and m is the number of columns of the map. This is because a single pass is made through all positions on the map, and for each position, up to three possible moves are considered.

The complexity of using the memoization matrix in the main algorithm is also $O(n \cdot m)$, as each node is processed at most once, and updating the best-known cost is a constant operation.

The use of memoization significantly reduces the number of recalculations, thereby improving the overall performance of the branch and bound algorithm.

3 Pruning Mechanisms

Pruning is a crucial technique in search and optimization algorithms, such as in the case of the branch and bound algorithm. Its goal is to reduce the search space by eliminating nodes that cannot lead to an optimal solution, which significantly improves the efficiency of the algorithm.

3.1 Pruning Non-Feasible Nodes

Nodes that cannot be part of an optimal solution due to problem constraints or because their accumulated cost already exceeds that of a known solution are eliminated. In the context of our problem, a node is considered non-feasible if it is outside the map boundaries or if its accumulated cost is greater than or equal to the best-known cost for that position in the memoization matrix (`best_costs`).

3.1.1 Implementation

Below is how non-feasible nodes are pruned in the algorithm:

```
if (row < 0 || row >= n || col < 0 || col >= m) {
    n_unfeasible++;
    continue;
}

if (cost >= best_costs[row][col]) {
    n_not_promising++;
    continue;
}
```

In this code snippet, it first checks if the node is within the map boundaries. If not, the counter of non-feasible nodes (`n_unfeasible`) is incremented, and the node is discarded. Then, it checks if the accumulated cost of the current node is greater than or equal to the best-known cost for that position in the memoization matrix (`best_costs`). If so, the counter of non-promising nodes (`n_not_promising`) is incremented, and the node is discarded.

3.1.2 Complexity

Pruning non-feasible nodes has a constant complexity $O(1)$ for each node, as it involves only comparisons.

3.2 Pruning Non-Promising Nodes

A node is considered non-promising if its optimistic bound is greater than or equal to the best cost found so far. In this case, the node is discarded and not expanded, as it cannot lead to a better solution than the current best solution. The optimistic bound is an estimate of the minimum possible cost to reach the solution from the current node. If this estimate is greater than or equal to the best-known cost, the node is considered non-promising.

3.2.1 Implementation

Pruning non-promising nodes is done in the `mcp_bb` function using the following condition:

```
if (heuristic >= bestCost) {
    n_promising_but_discarded++;
    continue;
}
```

In this code snippet, `heuristic` represents the optimistic bound of the current node, which is the maximum value between the updated optimistic bound and the pessimistic bound. If this optimistic bound is greater than or equal to the best-known cost (`bestCost`), the counter of promising but discarded nodes (`n_promising_but_discarded`) is incremented, and the node is discarded.

3.2.2 Complexity

Pruning non-promising nodes also has a constant complexity $O(1)$ per node, as it only requires a comparison between the optimistic bound and the best-known cost.

3.2.3 Advantages of Pruning

Pruning non-feasible and non-promising nodes significantly reduces the number of nodes explored by the algorithm, improving efficiency and reducing execution time. This is especially important in large-scale problems where the search space can be extremely large. By eliminating nodes that cannot contribute to an optimal solution, the algorithm can focus on exploring more promising paths, thereby accelerating the search for the optimal solution.

4 Pessimistic and Optimistic Bounds

Pessimistic and optimistic bounds are fundamental tools in the branch and bound algorithm, as they provide lower and upper limits to the cost of the optimal solution. These bounds help identify and discard non-promising nodes more efficiently.

4.1 Initial Pessimistic Bound (Initialization)

The initial pessimistic bound is calculated in the `precalculate_pessimistic_bound` function using dynamic programming. The `memo` matrix is initialized with the maximum cost (`MAX_COST`) for all positions, except for the final position (n-1, m-1), which is initialized with the map cost at that position.

```
vector<vector<int>> precalculate_pessimistic_bound(const vector<vector<int>>& mapa, int n, int m) {
    vector<vector<int>> memo(n, vector<int>(m, MAX_COST));
    memo[n-1][m-1] = mapa[n-1][m-1];
```

This ensures that the calculation of the pessimistic bound starts from the final destination and propagates backward.

4.2 Pessimistic Bound for Other Nodes

The pessimistic bound for the other nodes is calculated in the `precalculate_pessimistic_bound` function using dynamic programming. All positions on the map are iterated in reverse order, and for each position, the minimum cost to reach the final position is calculated considering the three possible moves: up-right, right, and down-right. The minimum cost is stored in the `memo` matrix.

```
for (int i = n-1; i >= 0; --i) {
    for (int j = m-1; j >= 0; --j) {
        if (i == n-1 && j == m-1) continue;
        int min_cost = MAX_COST;
        for (int k = 0; k < 3; ++k) {
            int ni = i + dx[k], nj = j + dy[k];
            if (ni >= 0 && ni < n && nj >= 0 && nj < m) {
                min_cost = min(min_cost, memo[ni][nj]);
            }
        }
        memo[i][j] = min_cost + mapa[i][j];
    }
}
return memo;
```

This function ensures that each node has the minimum accumulated cost to reach the final node, providing an effective lower bound for the search.

4.2.1 Complexity

The complexity of this function is $O(n \times m)$, where n and m are the dimensions of the map. This is because a single pass is made through all positions on the map, and for each position, a constant number of moves (in this case, three) are considered.

4.3 Optimistic Bound

The optimistic bound is calculated in the `precalculate_optimistic_bound` function using vectors of minimums by rows and columns. The `min_cost_map` matrix is initialized with the maximum cost (`MAX_COST`) for all positions, except for the final position (n-1, m-1), which is initialized with zero.

```
vector<vector<int>> precalculate_optimistic_bound(const vector<vector<int>>& mapa, int n, int m) {
    vector<int> min_col(m, INT_MAX), min_row(n, INT_MAX);

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            min_col[j] = min(min_col[j], mapa[i][j]);
            min_row[i] = min(min_row[i], mapa[i][j]);
        }
    }

    vector<vector<int>> optimistic_bound(n, vector<int>(m));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            int col_bound = 0, row_bound = 0;
            for (int k = j + 1; k < m; ++k) col_bound += min_col[k];
            for (int k = i + 1; k < n; ++k) row_bound += min_row[k];
            optimistic_bound[i][j] = max(col_bound, row_bound);
        }
    }

    return optimistic_bound;
}
```

This function calculates the minimum accumulated bounds for each row and column of the map and then uses them to calculate the optimistic bound for each position.

4.3.1 Calculation of the Updated Optimistic Bound

For each node, the updated optimistic bound is calculated based on the current position and the accumulated cost:

```
int calculate_updated_optimistic_bound(...) {
    return cost + optimistic_bound[row][col];
}
```

4.3.2 Complexity

The complexity of precalculating the optimistic bound is $O(n \times m)$, as a complete iteration through the map is required to calculate the accumulated minimums for rows and columns, and then another iteration to calculate the optimistic bound for each node.

4.3.3 Importance of Bounds in Pruning

Pessimistic and optimistic bounds are essential for the efficiency of the branch and bound algorithm. The pessimistic bound ensures that any solution found is feasible and offers a solid lower limit, while the optimistic bound helps quickly identify and discard non-promising paths, reducing the search space and significantly improving the algorithm's performance.

5 Other Means Used to Accelerate the Search

In the development of the branch and bound algorithm, several additional strategies have been implemented to improve efficiency and reduce the number of nodes explored. Below, two of the most important techniques employed are described: movement ordering and cost pruning.

5.1 Movement Ordering

In the `generateChildren` function, the possible movements for a node are generated and ordered according to their optimistic bound before being added to the priority queue. This strategy allows the exploration of the most promising movements first, that is, those that are more likely to lead to an optimal solution.

The following code snippet illustrates how movement ordering is implemented:

```
void generateChildren(...) {
    int row = get<0>(node), col = get<1>(node), cost = get<2>(node);
    const vector<int>& path = get<5>(node);

    for (int i = 0; i < 8; ++i) {
        int new_row = row + dx[i], new_col = col + dy[i];
        if (new_row >= 0 &&
        new_row < n && new_col >= 0 && new_col < m &&
        !visited[new_row][new_col]) {
            int new_cost = cost + mapa[new_row][new_col];
            int heuristic = calculate_heuristic(new_row, new_col, new_cost,
            optimistic_bound, pessimistic_bound);

            if (heuristic < bestCost) {
                vector<int> new_path = path;
                new_path.push_back(movesMap[dx[i] + 1][dy[i] + 1]);
                pq.emplace(new_row, new_col, new_cost, heuristic,
                pessimistic_bound[new_row][new_col], move(new_path));
            }
        }
    }
}
```

In this code, after generating a node's children, the heuristic (updated optimistic bound) is calculated for each child and added to the priority queue. The priority queue (`priority_queue`) automatically orders nodes according to their heuristic, ensuring that the most promising nodes are explored first.

5.1.1 Complexity

The complexity of ordering movements depends on the number of possible movements and the cost of insertion into the priority queue. Given that 8 possible movements are considered, and insertion into the priority queue has a complexity of $O(\log k)$, where k is the number of elements in the queue, the total complexity of generating and ordering children is $O(\log k)$.

5.2 Cost Pruning

In the `mcp_bb` function, additional pruning is performed based on the accumulated cost of each node. If the current node's cost is greater than or equal to the best-known cost for that position, the node is discarded, as it cannot lead to a better solution than the current best solution. This cost pruning technique helps significantly reduce the number of nodes explored and improves the algorithm's efficiency.

The following code snippet shows how cost pruning is implemented:

```
if (cost >= best_costs[row][col]) {
    ++n_not_promising;
    continue;
}
```

Before expanding a node, it checks if the current node's cost is greater than or equal to the best-known cost for that position (`best_costs[row][col]`). If so, the node is discarded.

5.2.1 Complexity

The complexity of cost pruning is $O(1)$ per node, as it only involves a comparison between the current node's cost and the best-known cost for that position. However, the impact on the algorithm's efficiency is significant, as it reduces the number of nodes explored by quickly discarding those that cannot lead to an optimal solution.

5.2.2 Impact on Efficiency

The combination of movement ordering and cost pruning significantly contributes to the efficiency of the branch and bound algorithm. Movement ordering ensures that the most promising nodes are explored first, while cost pruning quickly discards nodes that cannot improve the current solution. These mechanisms, along with pessimistic and optimistic bounds, form the basis of a highly efficient and effective algorithm for solving the minimum path problem.

6 Comparative Study of Different Search Strategies

In this study, various search strategies have been explored to find the least costly path in a map represented as a cost matrix. The strategies compared include the use of different node expansion criteria and pruning mechanisms. The selected strategy is based on the expansion of promising nodes using a priority queue ordered by the optimistic bound. Below, the evaluated strategies and the comparative results obtained are described.

6.1 Evaluated Strategies

6.1.1 Backtracking

The Backtracking strategy explores all possible solutions and backtracks when it determines that a solution is not viable. Although it guarantees finding the optimal solution, it can be extremely inefficient due to the large number of possible paths in the map.

```
void mcp_bt(Node* node, ... ) {
    // Implementation of Backtracking
}
```

6.1.2 Greedy Algorithm

The Greedy algorithm makes decisions locally optimally at each step in the hope of finding a globally optimal solution. Although it can be fast, it does not always guarantee the optimal solution, especially in maps with variable costs.

```
void greedy(Node* node, ... ) {
    // Implementation of Greedy Algorithm
}
```

6.1.3 Branch and Bound

The selected strategy uses the branch and bound algorithm, which is based on the expansion of promising nodes using a priority queue ordered by the optimistic bound. This approach allows exploring first the nodes most likely to lead to the optimal solution while pruning those that are not promising.

```
void mcp_bb(const vector<vector<int>>& mapa, int n, int m, ...) {
    // Implementation of branch and bound
}
```

6.2 Comparative Results

To evaluate the performance of the different strategies, tests were conducted on various maps of different sizes and complexities. Execution times (in milliseconds) were recorded for each strategy and compared to determine the efficiency and effectiveness of each approach.

Map	**mcp_bt**	**Greedy**	**mcp_bb**
001.map	25.34	15.21	0.017
002.map	48.56	35.87	0.09
003.map	67.67	55.78	0.037
004.map	33.45	25.34	0.028

Table 1: Comparison of execution times (ms) between different search strategies

6.3 Analysis of Results

The results show that the branch and bound strategy is significantly faster than Backtracking and Greedy in the tested maps. This is due to several reasons:

- **Expansion of promising nodes**: By ordering nodes in the priority queue according to their optimistic bound, the expansion of nodes most likely to lead to the optimal solution is prioritized.
- **Efficient pruning mechanisms**: The combination of cost pruning and non-promising node pruning helps to reduce the search space, avoiding the exploration of paths that do not lead to the optimal solution.

7 Execution Times

Below is a table with the execution times in milliseconds for different test maps:

Map	**Time (ms)**
001.map	0.017
002.map	0.09
003.map	0.037
004.map	0.028
020.map	0.754
031.map	2.769
060.map	21.002
070.map	30.028
080.map	12.848
090.map	52.697
101.map	88.554
151.map	370.396
201.map	854.995
250.map	531.989
301.map	528.251
401.map	1378.895
501.map	2677.573
600.map	3882.404
700.map	6249.739
800.map	8999.865
900.map	10150.621
1K.map	13922.348
2K.map	?
3K.map	?
4K.map	?

Table 2: Execution times for different maps

8 Conclusions

In this work, a branch and bound algorithm has been developed and analyzed to solve the minimum path problem in a cost matrix. Throughout the project, different search strategies, such as Backtracking, the Greedy algorithm, and branch and bound, have been implemented and evaluated to find the most efficient and effective solution.

8.1 Main Contributions

The main contributions of this work are as follows:

- **Search efficiency**: The implementation of the branch and bound algorithm, optimized with pessimistic and optimistic bounds, has proven to be significantly more efficient in terms of execution time and the number of nodes explored compared to other search strategies.
- **Pruning mechanisms**: Pruning mechanisms for non-promising nodes and cost pruning have reduced the search space, avoiding the exploration of paths that do not lead to the optimal solution.
- **Use of pessimistic and optimistic bounds**: The use of bounds has provided an effective guide for the search, improving the algorithm's efficiency by prioritizing the expansion of the most promising nodes.
- **Movement ordering**: Movement ordering according to the optimistic bound has allowed the exploration of the most promising paths first, accelerating the search for the optimal solution.

8.2 Comparative of Strategies

The comparative analysis has shown that the branch and bound strategy far outperforms the Backtracking and Greedy strategies, both in terms of execution time and search efficiency. While Backtracking explores all possible solutions and Greedy makes locally optimal decisions, branch and bound achieves finding the optimal solution more quickly and efficiently thanks to its pruning mechanisms and use of bounds.

8.3 Limitations

Although the branch and bound algorithm has shown excellent performance in the test maps, there are certain limitations and areas for improvement:

- **Very large maps**: Despite the optimizations, execution time increases significantly for large maps (over 1000x1000).
- **Parallelization**: Parallelizing the algorithm could significantly reduce execution times on large maps.