Overview

To find all the Hamiltonian paths within a 8x8 grid, our solution is derived from the Depth-First-Traversal of a Graph. The solution encodes the grid using a single 64-bit integer (bitmask) to record visited cells and introduces parallel processing with early pruning to improve efficiency.

There are five main Java methods implemented in this project, each method has its own purposes, functionalities and connection to other methods. Each method's purpose and highlights will be elaborated in the table below:

Fields:

- `TOTAL_STEPS`: Constant (63) indicating the number of moves after the initial cell.
- `START` and `END`: Constants indicating the start (0) and end (56) cells.
- Direction arrays (`UDIR`, `DDIR`, `LDIR`, `RDIR`, `ALLDIR`) holding movement offsets.
- `count`: A global, volatile long variable storing the total count of valid paths.
- `visitedMask`: A long representing visited cells as a bitmask.
- `allowedDirsPerStep`: A 2D array storing the allowed directions for each of the 63 steps.
- `PARALLEL_DEPTH`: Defines how many initial steps are parallelized.

Method	Purpose	Functionality	Connection
`preComputeDirsPerStep(String input)`	Precomputes the allowed movement directions for each step based	Reads the input string (63 characters, one for each step) Assigns a set of	Prepares the movement rules for use in both `backtrack` and `parallelBacktrack`

	on the input string	allowed directions (`UDIR`, `DDIR`, `LDIR`, `RDIR`, `ALLDIR`) to each step in the `allowedDirsPerStep` array	
`isValidMove(int current, int dir)`	Checks whether moving in a given direction is valid from the current cell	Ensures that moves say within the bounds of the grid	Used in both `backtrack` and `parallelBacktrack` to filter out invalid moves
`backtrack(int current, int step, long visitedMask)`	Performs sequential backtracking to explore all valid paths	If all steps are completed (`step == TOTAL_STEPS`) and the endpoint is reached, it increments the path count. For each allowed direction at the current step: - Computes the next cell and checks its validity. - Recursively calls itself to	Handles path exploration after the initial parallel steps in `parallelBacktrack`

		continue the path.	
`parallelBacktrack(int current, int step, long visitedMask)`	Combines multi-threading with backtracking to speed up path exploration.	If `step < PARALLEL_DEPTH`, spawns a new thread for each valid direction. Join all threads to ensure all paths are explored before proceeding. After `PARALLEL_DEPTH`, switches to sequential backtracking using	Drives the overall algorithm and ensures efficient parallelization at the root level.
`incrementCount()`	Safely increments the shared count variable	Used synchronized to ensure thread safety when updating the count.	Called whenever a valid path is completed in both `backtrack` and `parallelBacktrack`

Table 1: Methods and Roles in the MultiThreadBacktrack Class

Data Structures and Algorithms

Data Structures

- Underlying data structure

At the core of the system is an implementation of a Graph. The 8x8 grid could be represented as a graph by implementing a 2D array of integers as such:

```
int[][] arr = new int[8][8]
```

This is demonstrated in the Test class - which is our original implementation. By adopting the Decrease-and-Conquer, we reduced the grid size in order to verify the correctness of the system. However, in order to optimize the performance of the system, the actual representation of the Graph data structure is simplified down to a single integer from 0 to 63, which is our final version of the system - the `MultiThreadBacktrack` class.

- Bitmask for visited cells:

The program uses a `long` integer as a bitmask to track visited cells in the 8x8 grid. The current state of the grid during any particular step is represented by this `long`. This is a very efficient data structure, as the 64-bit capacity of a `long` aligns perfectly with the 64 cells of the grid. The indices of the cells are demonstrated in Table 2. Each bit in the bitmask corresponds to a cell, where a 1 indicates the cell has been visited, and a 0 indicates it is unvisited. This representation helps the team avoid the original approach of using a two-dimensional boolean array, saving memory and improving access speed. For example, marking a cell as visited involves setting the corresponding bit using the OR operation in `visitedMask | (1L << cellIndex)`, while checking if a cell has been visited is done using the AND operation in `(visitedMask & (1L << cellIndex)) != 0`. These operations are constant-time `O(1)` instead of linear time `O(n)` for the boolean array approach, making the bitmask the more optimal solution for frequent updates during backtracking.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

Table 2: 8x8 grid represented by a 64-bit bitmask with bit indices

- Precomputed direction array:

The `allowedDirsPerStep` is a two-dimensional integer array that stores the possible movement directions at each step in the traversal, as specified by the input string. This array is precomputed before backtracking begins to reduce overhead during recursion. Each entry in the array is a subset of the four possible directions (up, down, left, right) or all directions if the input character is a wildcard ('*'). For example, if the input character is 'D', the corresponding entry in `allowedDirsPerStep` contains only the "down" direction. By precomputing these values, the program avoids repeated string interpretation during recursion, thereby optimizing performance. However, this function also add a little bit more overhead by storing an additional array of size N (N is the length of the input string, which is 64 in the case of a 8x8 grid) and a pre-computing step with the `precomputeDirsPerStep()` method that traverses the input string with the time complexity of `O(N)`.

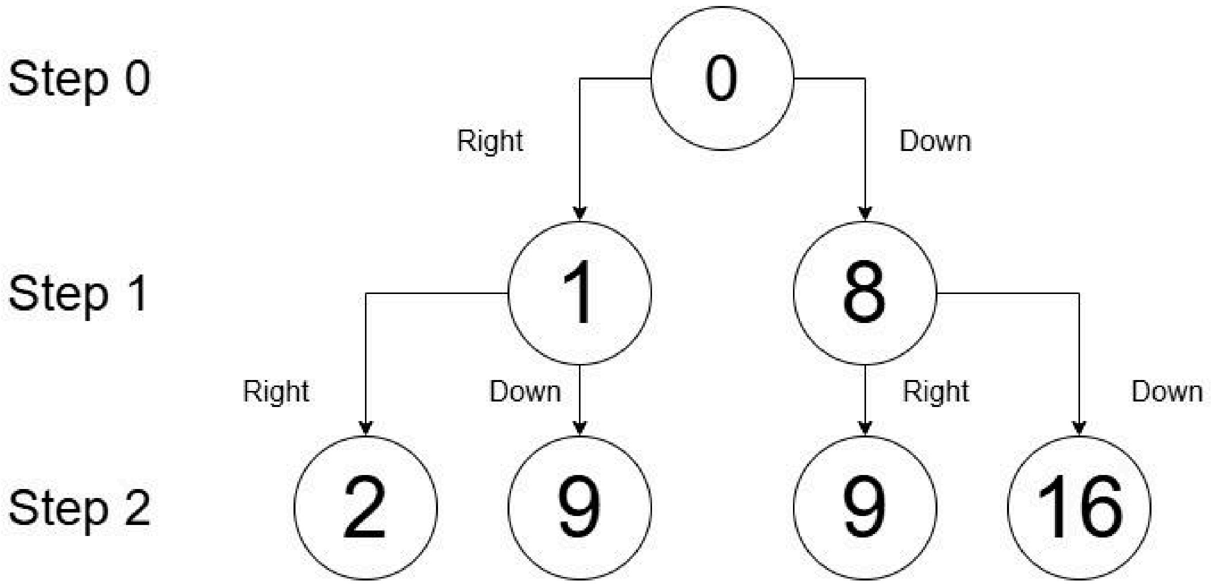
- Threads array:

To enable parallel execution, the program dynamically creates an array of `Thread` objects for initial recursive steps. Each thread independently explores a branch of the backtracking tree, allowing multiple paths to be evaluated concurrently. This approach is particularly effective for reducing the depth of sequential recursion by distributing the computational load across CPU cores. The threads array ensures that all started threads are joined before the main thread proceeds using the synchronized method `incrementCount()`, storing the total path count in a centralized variable `count`, maintaining the correctness of the computation.

Algorithms

- Underlying Algorithm

The basis for our backtracking algorithm is the Depth-First-Traversal of a Tree. Each step is represented as a node with each subtree being a possible path. The algorithm visits the deepest node of the left-most subtree first before backtracking to the parent nodes and moving to the next subtree.



- Recursive backtracking:

At the core of the program is a recursive backtracking algorithm that explores all possible paths through the grid. The decision to use a backtracking algorithm comes from the requirement to count all possible valid paths with constraints (paths must explore all cells of the grid exactly once). The team decided that this was the only intuitive way to solve this problem as other approaches, such as dynamic programming or other advanced algorithms that make use of heuristics for optimization, are usually used to find the most optimized path instead of counting all paths.

Recursive backtracking is a variation of brute force search by finding all possibilities within a given set of constraints until a solution is reached [1], and by doing it recursively with the base case being the state which is considered the end of the grid, the algorithm runs until it finds all possible solutions within the constraints,.

Starting from the left corner, the algorithm considers all valid moves at each step, marking the current cell as visited and recursively exploring the next step. The base case is reached when the total number of steps equals the input length (63). At this point, the algorithm checks whether the current cell is the bottom-left corner, incrementing the count of valid paths if true. This recursive exploration traverses through the entire recursive tree in a similar fashion to a depth-first search algorithm, ensuring that all potential paths are evaluated. Below is the pseudo-code to demonstrate this algorithm.

```
/* input: the input string */
/* step: the index of the current step (from 0 to 63) */
/* current: the index of the current cell within the 64-bit bitmask
(from 0 - 63) */
/* visited: the bitmask that marks which cells have been visited*/
/* directions: an array of [-8, +8, -1, +1] for Down, Up, Left and Right
respectively */
backtrack(string input, int step, int current, long visited)
    /* Base case: when the pointer reach the end of the grid */
    if (step == input.length)
        if (current == endCell) /* the end cell for this particular
    problem is the bottom-left cell, which is index 56 */
            solutionCount++
        endif
        return
    endif

    for (direction : directions) /* Down, up, left, right */
        int next = current + direction

        /* check valid move and recurse*/
        long bit = 1 << next; /* visit the next cell by incrementing
1 on the bit index of the next move */
        if ((visitedMask & bit) != 0)
            continue;
        endif
        backtrack(next, step + 1, visitedMask | bit);
    endfor
```

- Parallelized backtracking:

The algorithm parallelizes the initial steps of backtracking using Java's Thread objects. Up to a predefined depth (`PARALLEL_DEPTH`, default is 3), each recursive call spawns a new thread for every possible move. Each thread executes independently, exploring a unique branch of the backtracking tree. After the parallel phase, the algorithm switches to sequential backtracking for deeper exploration within each thread. This approach of creating a limited amount of threads balances efficiency while ensuring that the program runs reliably without crashing due to large workload.

- Precomputation of allowed directions based on input:

Before backtracking begins, the program precomputes the valid directions for each step based on the input string. For example, if the input specifies that step 5 allows only downward movement, the `allowedDirsPerStep` array contains the corresponding direction (+8) for that step. This preprocessing step eliminates the need to repeatedly interpret the input during recursion, reducing runtime overhead and streamlining the recursive calls. However, this optimization only works for inputs with specific moves and is less effective if the input string is mostly wildcards ("*").

- Bitwise operations:

A bitmask is used to track which cells in the grid have been visited during the backtracking process. Each cell in the 8x8 grid is represented by a single bit in a 64-bit `long`. The bit corresponding to a specific cell is set to `1` if the cell has been visited, and `0` otherwise. For example, the cell at index `i` is represented by the bit `1L << i`. The operation `(visitedMask & bit)` checks if a cell is already visited by performing a bitwise AND between the current mask and the bit for the target cell. If the result is non-zero, the cell has already been visited. To mark a cell as visited, the bit is added to the mask using a bitwise OR (`visitedMask | bit`). As mentioned above in the data structures section, this approach is efficient both in terms of memory space as well as time complexity for checking and updating operations.

Complexity Analysis

Time Complexity

- Precompute Directions: Utilizing a for loop to iterate through each of the elements in the input, thus a time complexity of **O(N)**.
- Backtracking: In the worst case, each step could branch up to 4 new branches, making the time complexity **O(N!)**. Due to the constraints of the problems, as the path grows, the number of available neighbors decreases. With early pruning of invalid paths, the realistic time complexity is much less than the approximation.

Space Complexity

- Backtracking: the recursion stack could go up to N in depth, leading to **O(N)** space complexity.
- The `visitedMask` is only 64 bit, taking O(1) space.
- `allowedDirsPerStep` is a static array of size `TOTAL_STEPS`, with each entry holding up to 4 ints, thus O(N).

Evaluation

Correctness:

- The algorithm's correctness was confirmed by reducing the input size. On smaller grid sizes (3x3 and 4x4), results were manually reviewed and verified. On larger grid sizes (5x5, 6x6 and 7x7), the output was compared against a brute force algorithm.
- The brute force algorithm (class Test) prints out every path along with the list of directions in order to verify accuracy.

Efficiency:

- By running the code with no parallel processing and without implementation of multithreading, the algorithm runtime was significantly higher and the CPU was not fully utilized.
- Through empirical analysis, the optimal number of initial steps to run in parallel was determined to be three. The tests were performed on four different devices, on both Windows and MacOS.

PARALLEL_DEPTH	Average Runtime (ms)
0 (Only sequential backtracking)	818392
1	483018
2	221908
3	191003
4	210544
5	198885

Conclusions

Overall, the solution provides a robust brute-force approach to fully traverse a grid. With the utilization of parallelization in the initial recursive steps, the program effectively computes through multi-core processors and can scale from 3x3 up to 8x8 grids. The use of a long bitmask instead of a two-dimensional array ensures constant-time cell checks (O(1)) and keeps memory usage low—exactly 64 bits for an 8x8 board. Within this program, overhead is put under control and visited cells can be efficiently detected. Additionally, dynamic direction constraints and edge-and-direction checks prevent the search from moving outside the grid, and initial pruning steps are incorporated to discard invalid moves early respectively.

However, the brute-force nature of the approach leads to inherent complexity and scalability limitations, especially in the case of a bigger grid while using full wildcard inputs, meaning the designed solution works efficiently only for the given scenario. While parallelization provides some speed-up, thread creation and synchronization costs can hinder the program's completion. Not to mention the fact that multithreading techniques implemented in this solution is currently at a basic level, and has yet to reach a high-level computational optimization through hardware utilization. Furthermore, the lack of sophisticated pruning or load-balancing strategies means that runtime can still become prohibitively large. Precomputation of directions is helpful, but its benefit is limited when many wildcards ("*") are present, causing an explosion of possible paths. Further refining pruning techniques, dynamically balancing parallel workloads, and introducing more intelligent heuristics could substantially improve efficiency and address the solution's current shortcomings.

In sum, the current design is effective within a controlled problem scope, demonstrating how bitmasks, limited parallelization, and direction precomputation can work together to enhance a Depth-First-Traversal solution. However, to achieve broader scalability and faster performance, further refinements in pruning, parallelization strategies, and heuristic-guided searches would be necessary.

References

[1] Job, D., & Paul, V. (2016). Recursive backtracking for solving 9* 9 Sudoku puzzle. *Bonfring International Journal of Data Mining*, *6*(1), 7-9.