Report LINFO1361: Assignment 3

Group N°13

Student1: Delsart Mathis

Student2: / (*Separate Duo*)

April 23, 2024

1 Search Algorithms and their relations (3 pts)

Consider the maze problems given on Figure 1. The goal is to find a path from 🚹 to € moving up, down, left or right. The black cells represent walls. This question must be answered by hand and doesn't require any programming.

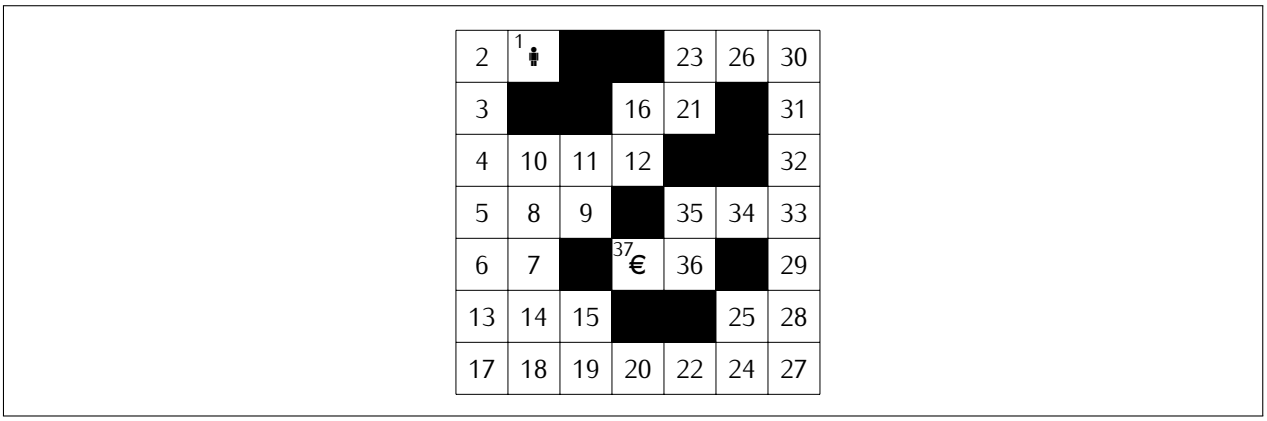
1. Give a consistent heuristic for this problem. Prove that it is consistent. Also prove that it is admissible. **(1 pt)**

One possible *consistent heuristic* for this problem is the **Manhattan distance** between the **current state** and the **goal state**, which measures the distance between two points using the **L1 norm**. In other words: $d(A,B) = |X_B - X_A| + |Y_B - Y_A|$. To prove its **consistency**, let's show that **for any node n**, $h(n) \leq c(n,n') + h(n')$. Intuitively, $h(n) = c(n,n') + h(n')$ if and only if n' is on the path from h(n). In all other cases, it will be a detour and we will have $h(n) < c(n,n') + h(n')$. Thus the triangle inequality is respected and therefore this **property ensures that the heuristic is consistent**. Therefore, it is also **admissible** because ***consistent*** $\Rightarrow$ ***admissible***. However, we can still demonstrate it. The **Manhattan distance is only equal to zero when** $A = B$**, which corresponds to the goal state**. In **all other cases**, The Manhattan distance is **positive** because physically, it's a distance and mathematically, it's a sum of absolute values. Thus, the **heuristic never overestimates the cost to reach the goal state and is therefore admissible**.

2. Show on the left maze the states (board positions) that are visited when performing a uniform-cost graph search, by writing the order numbers in the relevant cells. We assume that when different states in the fringe have the smallest value, the algorithm chooses the state with the smallest coordinate (i,j) ($(0,0)$ being the bottom left position, i being the horizontal index and j the vertical one) using a lexicographical order. **(1 pt)**

2	1 🚹			21	23	25
3			16	19		27
4	6	9	13			30
5	8	12		35	34	32
7	11		37 €	36		33
10	15	18			29	31
14	17	20	22	24	26	28

3. Show on the right maze the board positions visited by A^* graph search with a manhattan distance heuristic (ignoring walls), by writing the order numbers in the relevant cells. A state is visited when it is selected in the fringe and expanded. When several states have the smallest path cost, they are visited in the same lexicographical order as the one used for uniform-cost graph search. **(1 pt)**



2 N-Amazons problem (8 pts)

1. Model the N problem as a search problem; describe: **(2 pts)**

   - States
   - Initial state
   - Actions / Transition model
   - Goal test
   - Path cost function

• A ***state*** can be represented as an N x N grid, where each square can either be occupied by an Amazon (symbolized by 'A') or be empty (symbolized by '#'). While this method is intuitive, it requires storing N^2 elements in memory for each state. A more efficient approach is to represent a state as a list of N elements, where each index corresponds to a column and the value associated with that index represents the row where an Amazon is placed in that column. A value of -1 indicates that no Amazon is placed in that column.
• The ***initial state*** is a list of N elements initialised to -1 because there are initially no Amazons on the board.
• ***Actions*** involve placing an Amazon in the first empty column of the grid, ensuring that the Amazon is positioned in a square where no attacks occur. In other words, an action entails placing an integer between 0 and N - 1 in the first index of the list where the current value is -1. Only integers that avoid conflicts between Amazons are considered as valid actions. An attack conflict arises when one Amazon threatens another, indicating they share the same row, column (impossible with the Problem modeling), diagonal, or occupy specific squares on the board. These squares correspond to the extended knight's moves: either 3 tiles in one direction followed by 2 tiles in another, or 4 tiles in one direction followed by 1 tile in another.
• The ***transition model*** entails placing the row index (specified by the action) representing a new Amazon into the first element of the list where the value is currently equal to -1.
• The ***goal test*** involves checking if the last element of the list contains an integer between 0 and N - 1 (different of -1). If so, it means that all Amazons have been placed on the board.
• The ***path cost function*** can be defined as the total number of Amazons placed on the board. Each placement of an Amazon incurs a unit cost, and the total path cost is the sum of these individual placement costs. The **optimal cost** is thus N.

2. Give an upper bound on the number of different states for an N-Amazons problem with N=n. Justify your answer precisely. **(0.5 pt)**

To calculate the total number of possible configurations for the N-Amazons problem on a board of **size n x n**, we consider that each Amazon **can be placed in any of the n rows for each column, without any restrictions**. In other words, we do not take into account the constraint that there can be no attack conflicts between the Amazons placed on the board. Hence, the **total number of possible configurations** is simply n^n, as **for each column**, there are **n available squares to place an Amazon**. The more mathematical explanation is as follows. The formula n^n represents a form of permutation, where each element **can be chosen from n possible choices, and this is repeated n times**. This formula is used in the context of **arrangements with repetition**. **Another upper bound** that is more precise is **n!**, as we know that the **Amazons cannot be placed in the same row**. Once an Amazon is placed in a row, that row cannot be used again. Therefore, for the first action, there are n possibilities, then n-1 possibilities, n-2, and so on. Hence, a more precise upper bound is indeed n!. To find these upper bounds, I **relax the problem** (remove a constraint). ***The more we relax the problem, the less precise the upper bound becomes***.

3. Give an admissible heuristic for a N=n. Prove that it is admissible. What is its complexity ? **(1 pts)**

I have opted for a ***heuristic*** that minimizes conflicts between each already placed Amazon on the chessboard and the empty tiles in the remaining columns (= empty columns). For each attack by an Amazon, the *heuristic* score is incremented by 1. Hence, this *heuristic* effectively represents the number of tiles that are under attack by the already placed Amazons, making it impossible to place a new Amazon on those tiles. It's important to note that a single tile can be attacked by multiple Amazons, resulting in multiple conflicts for that tile. Moreover, this heuristic doesn't take into account the conflicts between each Amazon already placed because, by construction of the problem, each action consist to place a new Amazon without creating a conflict (see question *2.1*).
This *heuristic* is ***admissible*** because its **minimum score is 0 and is only reached when all the columns are filled**. **As soon as a column is not filled, the *heuristic* score is greater than 0** because there is at least $n - 1$ conflicts since two Amazons cannot be on the same row, demonstrating that the ***heuristic*** **never overestimates the cost to reach a goal state**. Thus, this *heuristic* is indeed ***admissible***.
The time complexity of this *heuristic* is $\mathcal{O}(x * (n - x) * n)$, where **x** is the **number of Amazons already placed on the chessboard** and **n** is **the size of the chessboard**. An **upper bound** on this complexity is therefore $\mathcal{O}(n^3)$.

5. **Implement** your solver. Extend the *Problem* class and implement the necessary methods and other class(es) if necessary. **(0.5 pt)**

6. **Experiment**, compare and analyze informed (*astar_graph_search*), uninformed (*breadth_first_graph_search* and *depth_first_graph_search*) graph search of aima-python3 on N = [10, 11, 12, 13, 20, 25, 30]. **(3 pts for the whole question)**

   Report in a table the time and the number of explored nodes and the number of steps to reach the solution.

   Are the number of explored nodes always smaller with *astar_graph_search*? What about the computation time? Why?

   When no solution can be found by a strategy in a reasonable time (say **3 min**), indicate the reason (time-out and/or exceeded the memory).

Firstly, it's noticeable that the **number of steps** is equivalent to the **size of the chessboard for all searches**, which is expected due to the construction of the problem discussed in question *2.1*.
Next, it can be observed that the ***execution time*** of A* is significantly **smaller** compared to algorithms like BFS_graph or DFS_graph. The reason for this is that **A* is guided by a heuristic and is therefore an informed search**. It doesn't blindly search the entire search tree like uninformed searches. Since the **maximum depth of the tree is N** and the **optimal solution is guaranteed to be at the last depth**, **DFS_graph is much faster than BFS_graph** because it **searches in depth** and the **optimal solution is at the last depth of the search tree**. **BFS_graph**, on the other hand, **searches in breadth and explores almost the entire search tree** (except the last layer, where it's not always necessary to search all of this layer). Thus, a breadth-first search is not optimal for this problem. We can also notice that, for small instances (N <= 13), DFS_graph is slightly faster than A*.
Regarding the number of ***explored nodes***, BFS_graph obviously **explores the most nodes since it explores the entire tree** (except the last layer where it's not mandatory). Then, **for small instances** (N=11/12/13), the number of **nodes explored by A* and DFS_graph are comparable and small** (slightly better for A*). However, for **larger chessboards** (N >= 20), **A* remains very efficient and explores a reasonable number of nodes** while **DFS_graph explores a very large number of nodes**.
Overall, A* stands out as the most efficient algorithm in terms of both time and the number of explored nodes, especially as the problem size increases. Its informed search strategy allows it to focus on the **most promising paths, resulting in faster convergence to the optimal solution**.

	*A** Graph			BFS Graph			DFS Graph		
Size Probl. (N)	NS	T	EN	NS	T	EN	NS	T	EN
10	10	674.956 μs	25	10	0.173 s	2298	10	3.007 ms	325
11	11	973.473 μs	18	11	0.886 s	7149	11	232.65 μs	24
12	12	11.314 ms	339	12	8.474 s	23923	12	2.123 ms	173
13	13	3.168 ms	95	13	105.935 s	87922	13	2.295 ms	107
20	20	18.189 ms	236	TO	TO	TO	20	1.069 s	51861
25	25	0.159 s	1488	TO	TO	TO	25	10.356 s	319795
30	30	0.47 s	3129	TO	TO	TO	30	236.313 s	5149276

NS: Number of steps — **T**: Time — **EN**: Explored nodes - **TO**: Time Out - **EM**: Exceeded memory

6. **Submit** your program on INGInious, using the *A** algorithm with your best heuristic(s). Your file must be named *namazon.py*. Your program should be able to, given an integer as argument, return the correct output. Your program must print to the standard output a solution to the N's given in argument for the N-Amazons problem, satisfying the described output format. **(2 pts)**

I followed the instructions and implemented the problem **using the modeling presented in question *2.1***. Additionally, I implemented a **class named *State***, which takes as arguments a **list of N elements** representing the chessboard with Amazons on it, as well as the **index of the first empty column from the left**. This class facilitates the implementation of the ***str()*** method for **formatting the output**, as well as the ***eq(), hash()***, and ***lt()*** methods, which are essential for search methods. The heuristic function for ***A**** is explained in question *2.3*.

3 Local Search: Sudoku Problem (8 pts)

1. Formulate the Sudoku problem as a Local Search problem (problem, cost function, feasible solutions, optimal solutions). **(2 pts)**

• ***Problem:*** Given a partially filled 9x9 grid (Sudoku puzzle), represented as a list of lists representing a 2D grid where each cell is either filled with a number or with a 0 indicating an empty cell, the objective is to fill in the empty cells such that each row, column, and 3x3 subgrid contains all the digits from 1 to 9 without repetition.
• ***Cost Function:*** The cost function in this problem is the evaluation of how close a particular state of the Sudoku grid is to the goal state. In other words, it measures the number of conflicts or violations of Sudoku rules present in the grid. This cost function would assign a higher cost to states with more conflicts. An optimal solution is found when the cost evaluated by this function is equal to 0 (when all the rules are adhered to without any violations).
• ***Feasible Solutions:*** Feasible solutions are grid configurations where all the digits 1 to 9 appear exactly once in each row, column, and 3x3 subgrid, and where initially fixed cells are respected. So it's a grid where all the constraints are adhered to.
• ***Optimal Solutions:*** The optimal solution to the Sudoku problem is the best feasible solution where the cost evaluated bu the cost function is equal to zero. This corresponds to a completed Sudoku grid where all cells are filled with digits 1 to 9, satisfying all row, column, and subgrid constraints. In this problem, all feasible solutions are also optimal, as we do not consider the number of states to reach the goal or any other selection criteria. In other words, once a solution satisfies all the constraints and rules of Sudoku, it is deemed optimal, regardless of the path taken to achieve it.

1. You are given a template on Moodle: `sudoku.py`. Implement your own simulated annealing algorithm and your own `objective_score` function. Your program will be evaluated in on 15 instances (during 3 minutes each) of which 5 are hidden. We expect you to solve (get the optimal solution) at least 12 out of the 15. **(6 pt)**

First, I set the ***cooling rate*** to **0.9999** to **allow for the maximum number of iterations** to converge towards the optimal solution. Additionally, I created a function that generates a **set containing initially fixed positions** in the Sudoku grid. I also create a function **to fill randomly the board** initially to win some time. For the ***neighbor generation function***, I made optimizations to make it more efficient than generating completely random neighbors. Here, **I randomly choose a cell among those not initially fixed. Then, I select a random number that does not create a conflict. If all numbers result in a conflict, I repeat the process. After 5 iterations, I return a random number in the selected cell**. This strategy helps to **escape local minima and reach the global minimum, which is 0** for this objective function. With this improvement, my solver performs significantly faster. Regarding the ***objective function***, I count the **number of empty cells and the number of conflicts** present in the Sudoku, taking into account the initially fixed cells as well.