AI LAB 1 REPORT

Implementation of Uninformed search, Informed search, Genetic and Constraint Satisfaction Algorithms

Abhinav Khanna(2017CSB1061)

CS512: Artificial Intelligence

Section 1: Lion Tale

INTRODUCTION

In this section, we have implemented various uninformed and informed search strategies: Depth first search, Breadth first search, Varying Cost Search and A* search. The goal is to find a path through the maze from the start point to the end point, avoiding obstacles on the way.

COMMON IMPLEMENTATION DETAILS

These are implementation details that are common in all the algorithms in the lion maze problem. Firstly, the position of a cell is indicated as follows: the topmost row is row 0, the bottom row is row 8. Similarly, the leftmost column is column 0, whilst rightmost column is column 8. Each position is represented as a [row, column] pair.

Q1: Depth first search

To implement DFS in the maze, run the file “lion_in_maze_DFS.py”. Each position of the blue box tells you which node is being searched. If the search is successful, the output on the terminal will be the “Meat found”, followed by the directions from the start position([0,0]) to the end position([9,9]). It also outputs the total cost, the time taken(in sec) and the maximum memory used by the frontier(in bytes). If however it is not possible to reach the goal state, then the output is simply, “Meat is unreachable”.

The DFS algorithm is **complete**, but **not optimal**, since it chooses the first solution it finds.

DFS Time Complexity

The **time complexity for DFS is O(N^2)**, where the dimension of the entire maze is N x N. That becomes evident in the case shown in figure 1. In this case, the path is about (N^2/2) cells long.

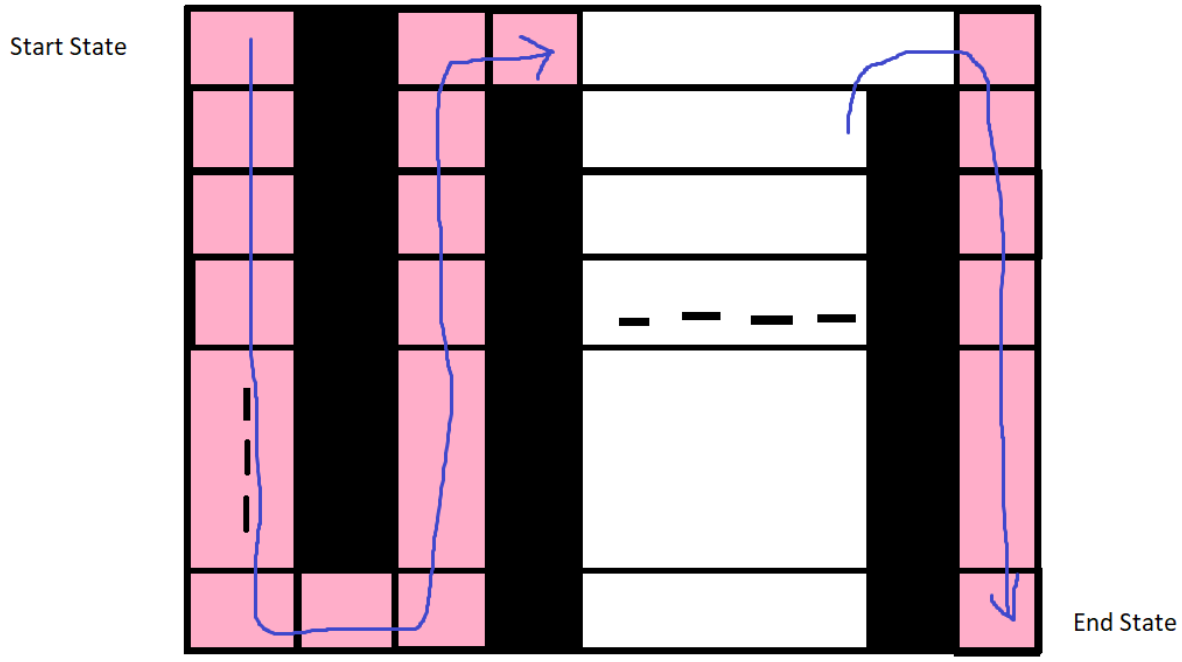


Figure 1

Q2: Breadth first search

To implement BFS in the maze, run the file “lion_in_maze_BFS.py”. All the outputs stay the same as discussed in the case of DFS.

The BFS algorithm is both **complete** and **optimal**.

BFS Space Complexity

The worst case(space-wise) will be if there are no obstacles, in which case all the nodes will either be in the frontier or the explored set. Thus, **space complexity for BFS is O(N^2)**, the dimensions of the maze being N x N.

Q3: Varying Cost Search

To implement VCS in the maze, run the file “lion_in_maze_VCS.py”. All the outputs stay the same as discussed in the case of DFS.

VCS is **complete and optimal**, since it is basically a heuristic search where cost of heuristic function is 0 at all nodes, hence the heuristic is admissible and consistent.

VCS **time complexity is O(N^2)**, where the same example as shown in Figure 1 acts as the worst case.

Q4: A* algorithm

To implement A* in the maze, run the file “lion_in_maze_A_star.py”. All the outputs stay the same as discussed in the case of DFS.

A* is both **complete and optimal**, since the Manhattan heuristic(used in this implementation) is both admissible and consistent.

A* Time complexity

A* **time complexity is O(N^2)**, where the same example as shown in Figure 1 acts as the worst case.

Run Time Statistics

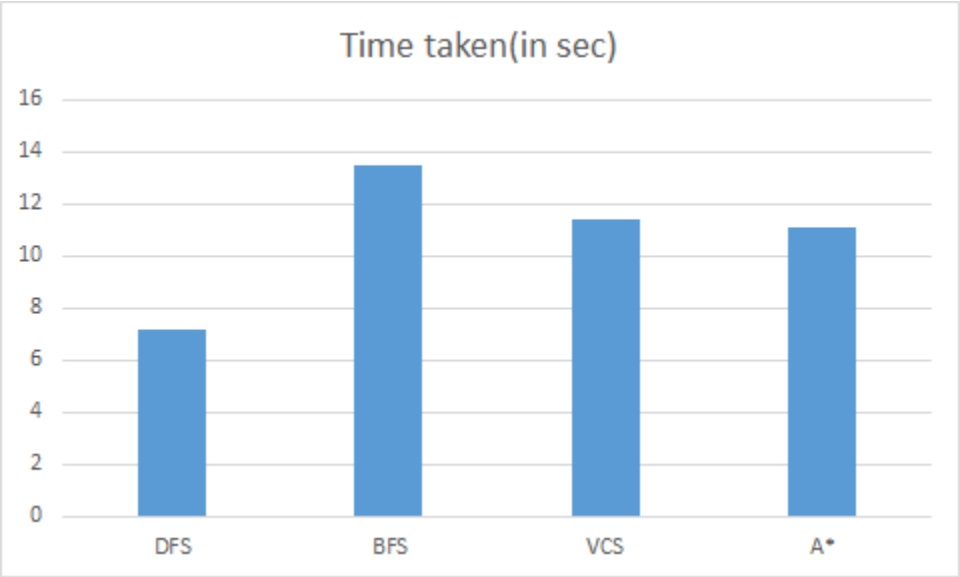


Figure 3

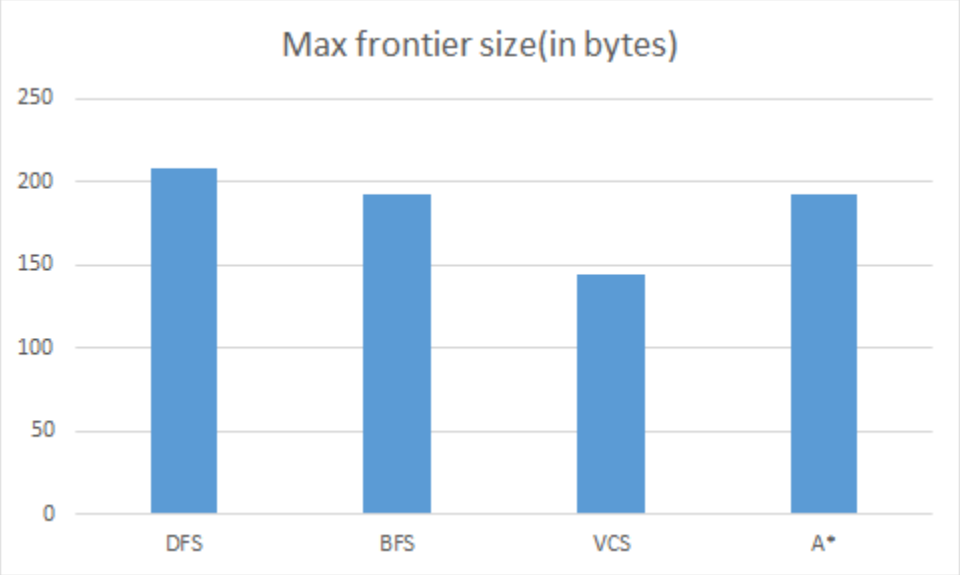


Figure 4

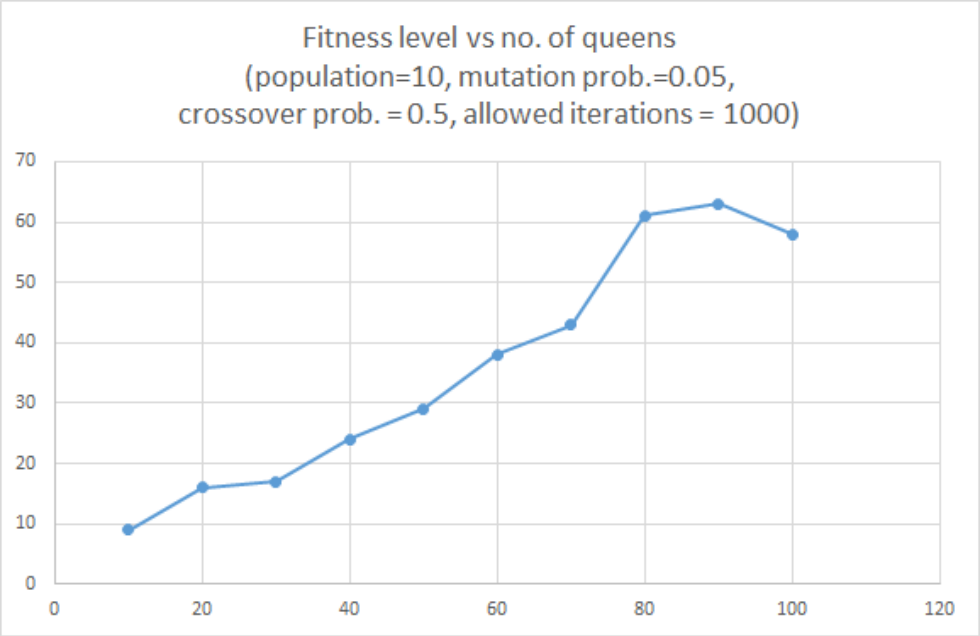
Section 2: 8-Queen puzzle

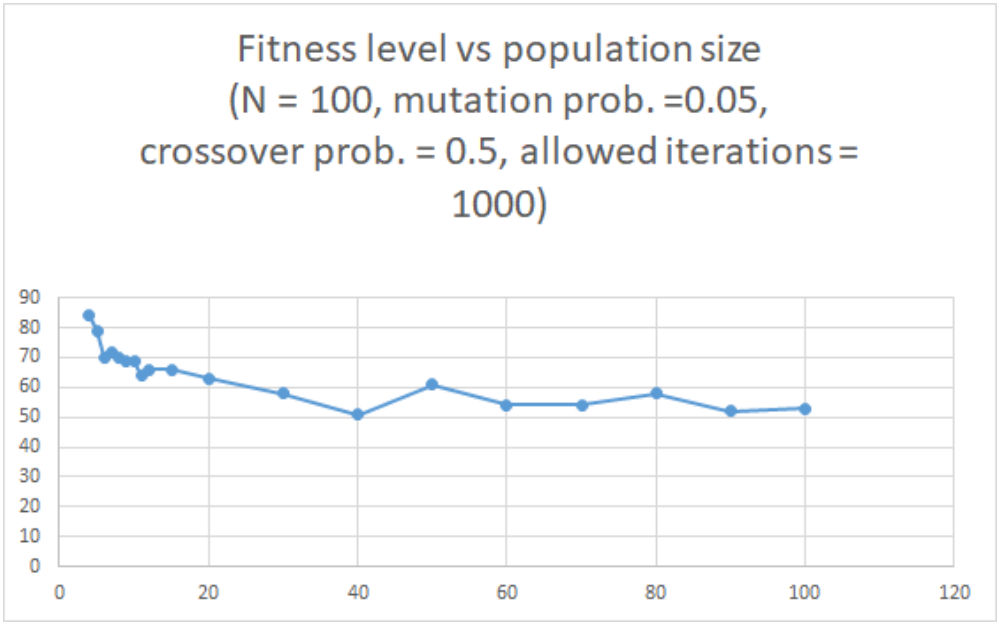
INTRODUCTION AND IMPLEMENTATION

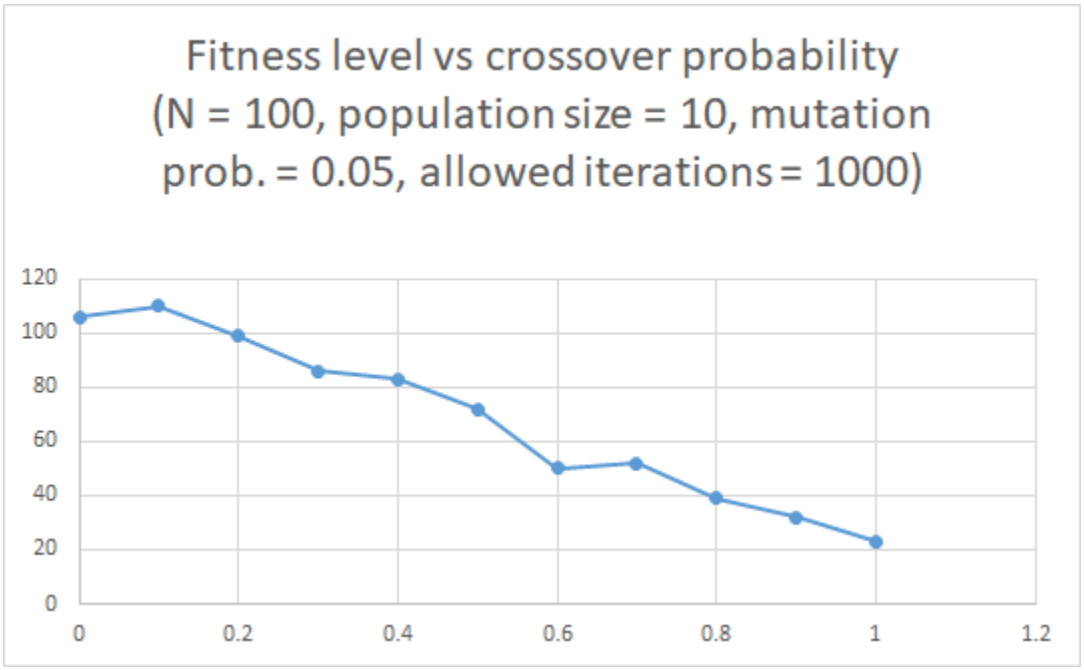
In this section, we have implemented the Genetic Algorithm to solve the n-queens problem. Being a local search algorithm, it does not necessarily find the exact solution. Rather, it tries to generate the individual with the best fitness level. In this particular implementation, fitness is defined as the number of attacking queen pairs. Hence, our aim is to reduce the fitness level, contrary to usual Genetic Algorithms, where the aim is to maximize fitness level. In this implementation, the two individuals with the best fitness levels are chosen, and crossover happens with some probability. The generated progeny may also undergo mutation, but with a very small probability. If any of the new children have better fitness than the parent, then that child replaces the parent in the population. To run the algorithm, run the file “GA.py”.

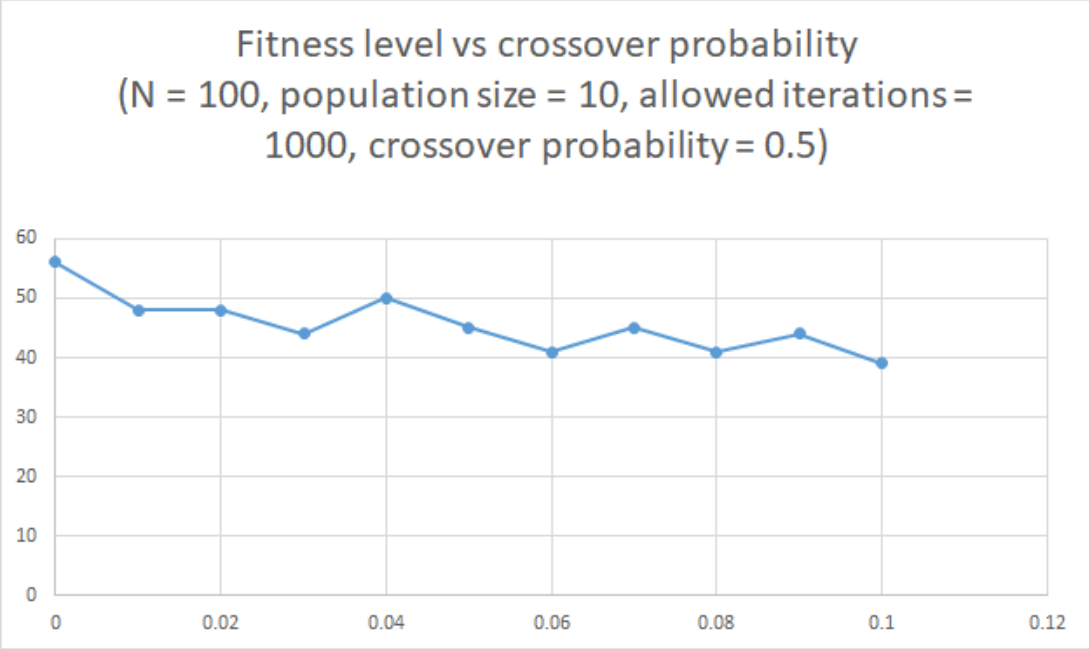
RUN TIME STATISTICS

Variation in the fitness level with changes in number of queens, population size, crossover probability and mutation probability is shown below.




Fitness level vs population size
(N = 100, mutation prob. =0.05,
crossover prob. = 0.5, allowed iterations =
1000)
90
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100
120


Fitness level vs crossover probability
(N = 100, population size = 10, mutation
prob. = 0.05, allowed iterations = 1000)
120
100
80
60
40
20
0
0
0.2
0.4
0.6
0.8
1
1.2



Section 3: Solving Wordoku

INTRODUCTION

In this section, the wordoku puzzle has been solved by two methods, Backtracking and Min-conflict. The most important difference between the two methods is that Backtracking always finds a valid solution, whereas Min-conflict might not, since it is a Local Search algorithm, which can get stuck at some local optima.

To run the backtracking algorithm, run the file “Wordoku_Backtracking.py”. The wordoku puzzle is read from the file “input.txt”, and if the woroku can be solved, then the output is written in the file “solution.txt”. If however, it cannot be solved completely, then the program simply outputs “No solution exists” in the terminal. It also outputs all

the meaningful words found in the rows, columns and diagonals(not that substrings of the rows/columns/diagonals are also considered). The number of nodes generated, the total clock time and the total search time is also given in the output.

To run the Min-conflict algorithm, run the file “Wordoku_minconflict.py”. Input and output remain the same, except that there is no notion of nodes in this case, hence the number of nodes is also not printed. It should be noted that it might not solve the wordoku entirely. Thus, total conflicts are counted for each iteration(analogous to number of attacking queen pairs in the previous section). Our aim is to minimize the total conflicts.

Important note: To check the meaningful words, the Pyenchant library has been used, which has to be installed separately, using the command “pip install pyenchant”. Note that whether a word is meaningful or not is determined solely by the enchant library.

COMPLEXITY ANALYSIS

Backtracking

Time Complexity: Let the number of values in the domain be D, and N x N be the dimension of the wordoku. In the worst case scenario, we would have to try all D values for each of the N^2 cells. Thus, time complexity is O(D^(N^2)).

Space Complexity: We consider one wordoku puzzle at a time. Thus space complexity is O(N^2).

Min-conflict

Time complexity: Let the max number of allowed iterations be MAX_STEPS. In each step, one variable, in this case one cell, is chosen, and all the D values in the domain are checked to determine which one reduces conflict the most. Thus, the overall time complexity is O(MAX_STEPS * D).

Space Complexity: We consider one wordoku puzzle at a time. Thus space complexity is O(N^2).

OBSERVATIONS

Backtracking

Parameters	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5
# of Nodes	45597	6776	6391	6442	83
Total Clock Time(in sec)	2.06	1.74	1.77	1.77	1.71
Total Search Time(in sec)	0.34	0.045	0.05	0.05	0.01

Min-conflict

Max # of iterations = 1000

Parameters	Test Case 1	Test Case 2	Test Case 3	Test Case 4	Test Case 5
Total Clock Time(in sec)	4.04	3.33	3.66	3.44	3.64
Total Search Time(in sec)	2.28	1.63	1.94	1.7	1.95