

# HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

**SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY**

------------

PROJECT REPORT

*AI Snake game problem*

Class: Introduction to Artificial Intelligence – 131117

**GROUP**

1. Trương Anh Huy – 20200287
2. Tống Trần Minh Đức –
3. Bùi Hữu Thành Công –

# INTRODUCTION

In the course Introduction to AI, we have studied about several kinds of Intelligent Agents such as simple reflex agents, model-based reflex agents, goal-based agents, utility-based agents, knowledge-based agents. Among them, utility-based agents, consider search cost and the path cost in the progress of trying to reach the goal. This agent is the topic we choose for our programming project.

In this report, we present the AI gardening problem and propose some methods for solving this problem. We will also concentrate on explain in details how each algorithm works with some helpful analyses.

**Contents**

1. [Presentation of the subject 1](#_bookmark0)
2. [Description of the problem 2](#_bookmark1)
   1. [Detailed description of the problem 2](#_bookmark2)
   2. [Problem formulation 2](#_bookmark3)
3. [Algorithm selection 3](#_bookmark4)
   1. [Algorithm for searching 4](#_bookmark5)
      1. [A\* Search Algorithm 4](#_bookmark6)
      2. [Breadth First Search(BFS) 5](#_bookmark7)
   2. [Algorithm for traversing :Depth First Search Algorithm (DFS) 6](#_bookmark8)
4. [Algorithms implementation in the propose problem 6](#_bookmark9)
   1. [Main difficulty 6](#_bookmark10)
   2. [Propose a solution to deal with the difficulties 7](#_bookmark11)
   3. [Algorithm improvement by the solution 8](#_bookmark12)
5. [Results comparison of the algorithm used for solving the problem 8](#_bookmark13)
   1. [Providing quantitative performance indicators 8](#_bookmark14)
   2. [Explaining these results 9](#_bookmark15)
6. [Conclusion and further discussions 10](#_bookmark16)
7. [List of tasks 10](#_bookmark17)
   1. [Programming tasks 10](#_bookmark18)
   2. [Analytic tasks 10](#_bookmark19)
8. [List of bibliographic reference 11](#_bookmark20)

## Presentation of the subject

There is a m x m square grid. We denote (x,y) is the location of one square. The grid contains the following:

* + A snake which spans a few adjacent squares (Denote 1 for the head and 2 for body)
  + A food in a single square (only one food can exist at a time) (Denote 3)
  + Empty squares (Denote 0)

**0**

**3**

**0**

**0**

**0**

**0**

**0**

**1**

**0**

**0**

**0**

**0**

**2**

**0**

**0**

**0**

**0**

**2**

**0**

**0**

**0**

**0**

**0**

**0**

**0**

The snake starts at the middle of screen with an initial length and can perform the following actions:

* + The head of the snake can move in 4 directions: up, down, left, right; dragging the whole body along.

The snake can not go the direction of its adjacent body part

For example: the snake can not go down in the instance above.

* + When the head is in the same position as the food, the snake eats the food and increase its length by 1. After this, a new food is spawn at a random square where there is not a part of the snake (a 0 square)
  + The snake dies when the head meets any other body part or go outside of the grid.

Functionality of the AI:

* + The AI can control the snake in any direction (according to the rule).
  + The goal is to stay alive while eating food whenever possible

(prioritize survivability above all else).

## Description of the problem

### Detailed description of the problem

* + - The environment is known, fully observable, stochastic (deterministic in between eating section), static and discrete. Thus, the solution is not a fixed sequence of actions.
    - There are two main parts in this problem to deal with:
      * Searching: to find a path to the food
      * Traversing: guarantee snake's survival when no path to food is available and after eating food.
* How we solve these problems:
* To solve searching problem we use search algorithms
* To solve the traversing problem we make sure the snake always has a path from its head to its tail

## Problem formulation

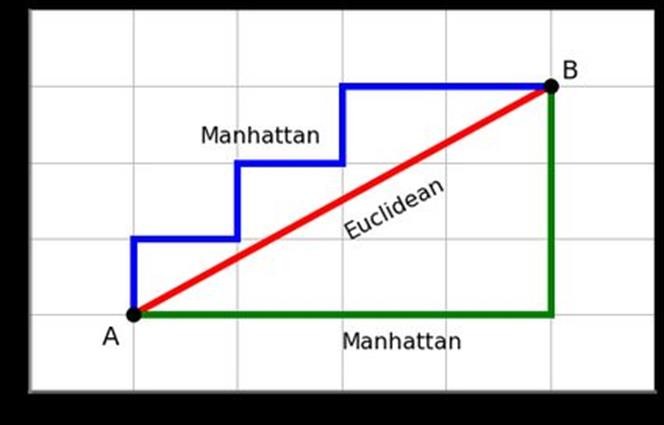
* + - States:
* A list of unfixed size listing all the square which is body part of the snake from head to tail. Each square consists of:
  + The position of the square
  + The direction that square is heading
* The position of the food
  + - Initial state: A list of size INITIAL\_LENGTH (the length the snake starts with) listing all body part of the snake and a randomly generated food position.
    - Actions: go up, go down, turn left, turn right.
    - Transition model: Expected effects
    - Goal test: This checks whether the snake is at its maximum length
    - Path cost: each action cost 1, so the path cost is number of actions taken.

## Algorithm selection

For searching the shortest path, we apply two algorithms: A\* searching algorithm and Breadth first search algorithm (BFS).

For traversing we apply several methods to guarantee the snake's survival

* BFS is one of the best algorithms to find the shortest path in matrix problem. To find the shortest path, all you have to do is start from the source and perform a BFS and stop when you found your destination. The only additional thing you need to do is have an array previous[] which will store the previous node for every node visited.
* A\* searching algorithm is also one of the best algorithms to find the shortest path in matrix problem. This algorithm is complete and optimal. Besides, the heuristic function in this problem is quite clear, which is the Manhattan distance (the sum of absolute differences between two point coordinates).
* Combining A\* search(or BFS) algorithm with DFS algorithm, we can find the shortest path and traverse through every single cell in the matrix.

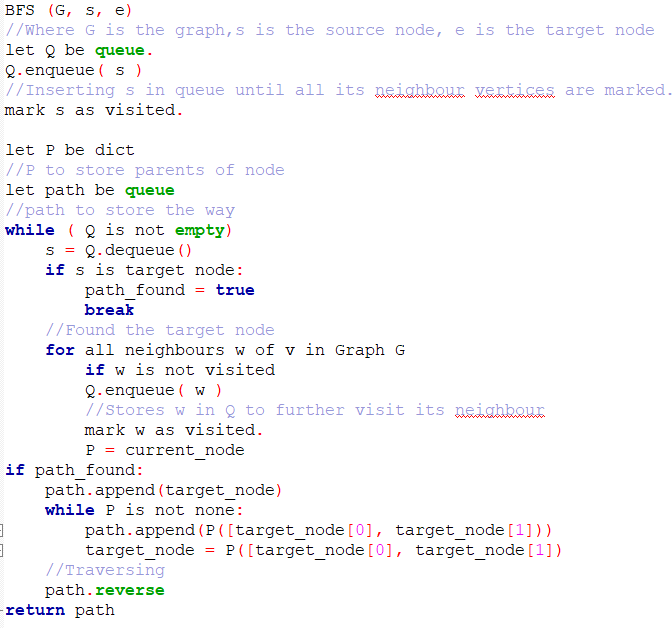


### Algorithm for searching: 3.1.1 A\* Search Algorithm

We apply the A\* search into this problem:

* + - Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. So here we use A\* Search Algorithm.
    - The heuristic function in this algorithm is the Manhattan distance (the sum of the absolute differences between coordinates of two points) of the point considered. (For example, the Manhattan distance between (x1,y1) and (x2,y2) is |x1 - x2| + |y1 - y2|)
    - The reason we use Manhattan distance:
      * It is admissible.
      * Since we can only move the blocks 1 at a time and in only one of 4 directions (right, left, top, bottom), the optimal scenario for each block is that it has a clear, unobstructed path to its goal state.

### 3.1.2 Breadth First Search(BFS)

We apply the BFS into this problem:

* + - * Consider a square grid having many obstacles and we are given a starting cell and a target cell. There are many ways to traverse graphs. BFS is the most commonly used approach, so we apply this algorithm.
      * Let's first start out with search - and search for a target node. Besides the target node, we'll need a start node as well. The expected output is a path that leads us from the start node to the target node.
      * When we're reconstructing the path (if it is found), we're

going backwards from the target node, through it's parents, retracing all the way to the start node. Additionally, we might want to reverse the path for our own intuition of going from the start\_node towards the target\_node.

### ~~3.1.3 Depth First Search(DFS)~~

Depth first search yields such bad results that we no longer consider it.

### Algorithm for traversing

For traversing we have the following algorithm:

* Step 1: The snake use a searching algorithm to find a path from its head to the food (from now referred as path 1). Move to step 4 if path 1 is not available.
* Step 2: Create a virtual snake identical to the original snake and let it follow path 1. Then, check if there a path is available from the virtual snake's head to its tail (from now refereed as path 2). If path 2 not available go to step 4.

Note: If path 2 is available we refereed to path 1 as a safe path.

* Step 3: If both path 1 and 2 are available, let the snake follow path1
* Step 4: If path 1 or path 2 is not available, let the snake follow its tail (from now we refer to this action as recover).

note:

+) all these steps are repeated for each move of the snake

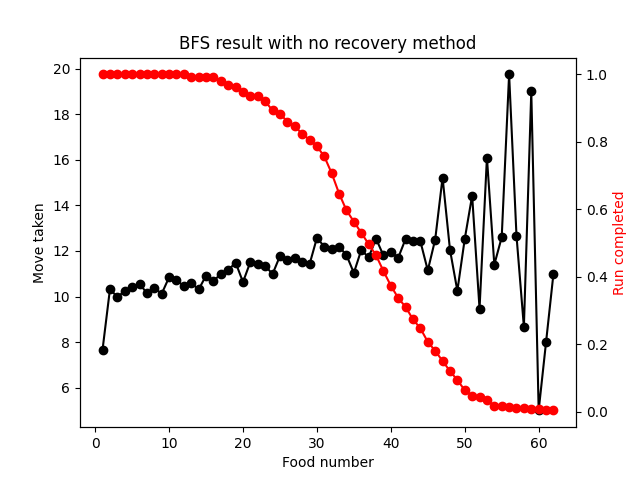
+) from now on, a safe path is referring to the path that if the snake follows it through there will be a path from the snake's head to its tail

I.e. if path 2 is available then path 1 is safe

## Algorithms implementation in the propose problem

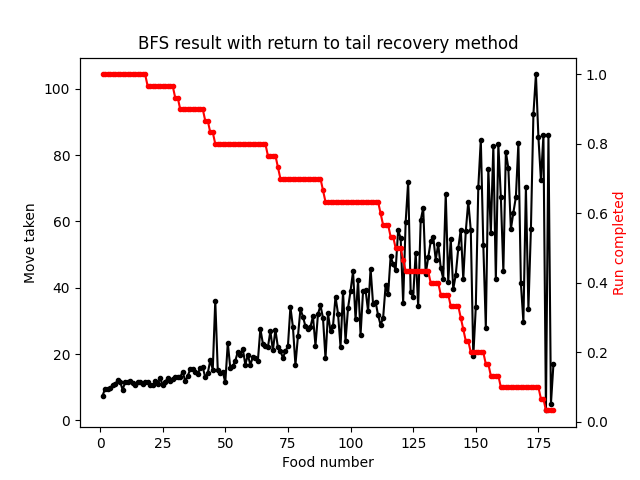
During the work, we have faced some difficulties and come up with corresponding solutions:

* + - At first, we let the snake follow the safe path to food once it is found. But as the snake moves a shorter path may emerge since the obstacles change as the snake dragging its body along
* We decide to search for new path every time the snake moves. This will increase calculation time but decrease number of steps taken which is the priority of our evaluation.
  + - Recovery method (as mentioned in Step 4 of traversing algorithm), how the snake follow its tail, greatly affect the result of the AI. This will be our main focal point as the searching problem is quite simple:
      * If we only use search algorithm with no recovery method the snake will most likely die shortly after start.

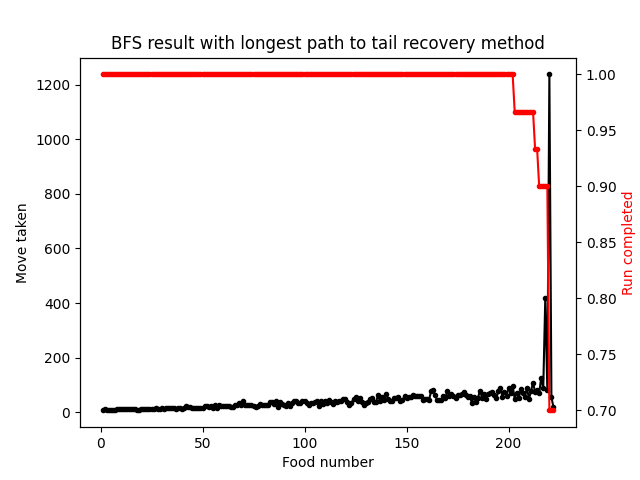


* + - * After that we decide to add a recovery method: if path 1 is not safe or when path 1 is not available do not follow it, instead make the snake follow the shortest path to its tail (search algorithm is used after each move of the snake to check for path to food)

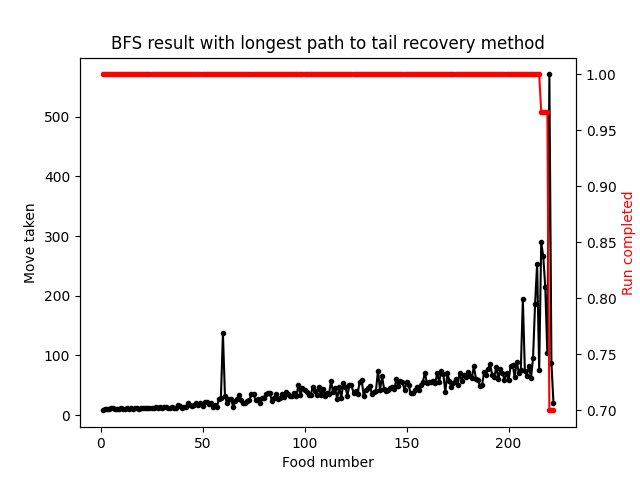
Using this method, most run are stuck in a loop so we add a maximum number of moves without eating (when this number is reached the run is immediately terminated) and yield the following result



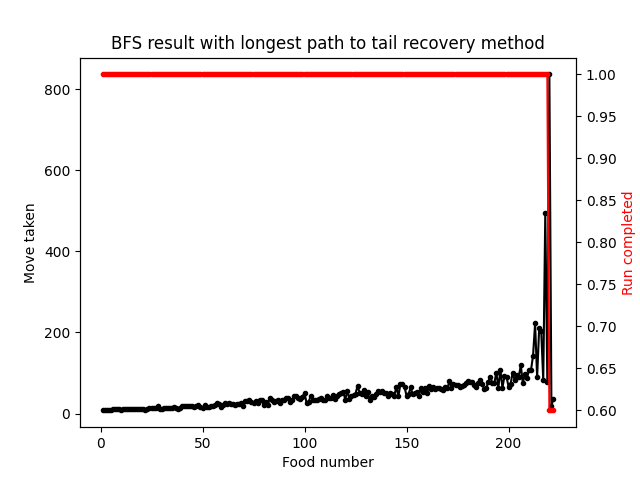
* + - * Since using the above method most run is terminated because of loop. We decide to change the recovery method to move the snake away from its tail if a safe path 1 is not available instead of the shortest path to tail to allow the snake to live longer without entering a loop



* + - * Using the above algorithm, there are still some cases that the snake got stuck in a loop. So we add a new method that makes the snake move to a random safe square when a safe path 1 is not available and move without eating has passed half of its maximum value



* + - * We try to mix random and longest path recover method (depend on odd or even food number) to see if the result get better but it seems to get worse so we revert back to previous recover method



* + - * We could not find a recover method that make the whole algorithm complete (or maybe it's already is with limitless moves) this will be studied further in future project

## Algorithm improvement by the solution

* A\* Search Algorithm
  + It costs much less time.
* DFS Algorithm
  + With two simple functions we use above, we can now see all the actions the machine needs to take to finish this problem.

## Results comparison of the algorithm used for solving the problem

### Providing quantitative performance indicators

* + - Total number of test cases: 100.
    - Number of test cases for which the algorithms can successfully find a path without breaking: 57.
    - Percentage of problem completion for each algorithm (including both problems which do not exist any solution)
      * A\* search + DFS: 100%.
      * DFS + BFS: 100%.
    - Time and space complexities of each test cases observed in practice given in this table:

Data experiment.xlsx

## Explaining these results

* + - If any of our algorithms completes searching (including cases where it successfully touches every cell and cases where it is stuck by obstacles), it will return the time taken and the number of steps taken in cases it successfully touches every cell or “Can not find the path” in cases where it is stuck by obstacles.
    - For 2 ways of solving this problem, we both use DFS to traverse from one node to other, the only difference came from choosing the searching algorithm: A\* search or BFS. In general, for the small size test cases, the execution time between these two do not show any noticeable differences. But for some larger size problems, the BFS’s time for searching might be over 3 times slower than A\* search. This is reasonable since the A\* search is more well developed, so it should be more effective. However, the total steps of 2 algorithms are quite similar for most cases.
    - A\* search and BFS have the same space complexity in theory, which is O(b^d) as they store all generated nodes in the memory.

|  |  |  |  |
| --- | --- | --- | --- |
| **Criterion** | **A\* search** | **BFS** | **DFS** |
| **Complete?** | Yes | Yes | No |
| **Time Complexity** | (Depends on the heuristic) | O(b^d) | O(b^m) |
| **Space Complexity** | O(b^d) | O(b^d) | O(bm) |
| **Optimal?** | No | Yes | No |

## Conclusion and further discussions

* + In conclusion, we believe that we manage to solve this problem quite well, with not so much time given. We have created a simple AI gardening machine which can have a very good application to life. However, there is still much work to do left to make this more efficient. Because as you could see, both of the algorithms are really time consuming.
  + During the project, we have learnt a lot of things: A\* search algorithms, BFS algorithms, DFS algorithms. We have learnt how these algorithms work, how to apply, combine them, how to choose the right algorithms to deal with the problem, how to analyse data, time and space complexity. This is the first time our team have had a team-work project, which we believe is very essential for our future. Thanks to the team, we have finished this project.
  + Further discussions: In the future, we hope that we can continue to develop this project to make it more efficient (For example, put multiple warehouses in the matrix and the AI machine can visit the nearest warehouse when in need). We hope that one day we can make it into a real machine that can help people.

## List of tasks

### Programming tasks:

* Implementing BFS Algorithms: Hoàng Huy Chiến (75%), Nguyễn Kim Tuyến (25%).
* Implementing A\* Search Algorithms: Đỗ Tuấn Minh.
* Implementing DFS Algorithms: Đỗ Tuấn Minh.
* Generating random cases for the problem : Tô Thái Dương (50%), Đinh Ngọc Hạnh Trang (50%).

### Analytic tasks:

* Nguyễn Kim Tuyến : proposing our subject, write about BFS in the report, standardization of reports.
* Hoàng Huy Chiến : proposing our subject, making PDF to presentation.
* Đỗ Tuấn Minh: proposing our subject, choosing algorithms, idea to solve this problem, write 1,2,3.1.2,3.2,4,6,7,8 (except every thing about BFS) in the report
* Tô Thái Dương: proposing our subject, generating data sets, analyzing experiments results, making demo video.
* Đinh Ngọc Hạnh Trang: proposing our subject, generating data sets, analyzing experiments result, write 5 in the report.

## List of bibliographic reference

* *A\* Search: Concept, Algorithm, Implementation, Advantages, Disadvantages,* Source: [https://www.brainkart.com/article/A--Search--](https://www.brainkart.com/article/A--Search--Concept%2C-Algorithm%2C-Implementation%2C-Advantages%2C-Disadvantages_8883/) [Concept,-Algorithm,-Implementation,-Advantages,-Disadvantages\_8883/](https://www.brainkart.com/article/A--Search--Concept%2C-Algorithm%2C-Implementation%2C-Advantages%2C-Disadvantages_8883/)
* *Depth First Search,* Source: [https://en.wikipedia.org/wiki/Depth-](https://en.wikipedia.org/wiki/Depth-first_search) [first\_search](https://en.wikipedia.org/wiki/Depth-first_search)