MCGILL 2015

MAZE SOLVING ALGORITHM INSPIRED BY 'SMART' BIOLOGICAL AGENTS

ECSE457 DP10 FINAL REPORT

ELITSA ASENOVA | 260481980 | ECSE 457
EILEEN FU | 260482511| ECSE 457

PROJECT SUPERVISOR: DAN NICOLAU
McGill University April 2015

Abstract

Microscopic fungi were found to exhibit “smart” space searching behavior in microconfined spaces. Their strategies can be examined to design new and improved algorithms for solving NP-complete, graph-based decision problems, or emulated when implementing artificial intelligence navigation systems. In this project, we aim to study the pathfinding strategies of these microorganisms, and translate their behavior to a software language in order to create a reliable, bio-inspired algorithm.

Graph-based problems such as uninformed maze-solving, travel planning and production scheduling are NP-complete or NP-hard, and are extremely difficult to solve sequentially and without heuristics in polynomial time. However, the fungus we studied uses branching and directional memory to efficiently explore mazes. This behavior is modelled computationally in an algorithm that combines and improves upon traditional methods such as depth-first search and branch-and-bound, creating a maze-solving strategy that emulates natural growth patterns of filamental fungi.

The algorithm was developed using the Java programming language and displayed in animation with a Java application. It was tested against other maze searching algorithm in order to provide an estimate of its efficiency. In addition, a simple game is added which lets the user compete with our algorithm to solve a maze.

Acknowledgement

We want to thank Hsin-Yu Lin (Jamee) (Master Mechanical Engineering student at McGill University) for providing us with her MATLAB simulation of the fungi pathfinding strategy.

Table of Contents

Abbreviations, Definitions and Notations

B&B	Branch-and-Bound; a B&B algorithm searches state space and explores branches while checking them against upper and lower estimated bounds
BIA	Bio-inspired algorithm
BFS	Breadth-first search; a FIFO graph-searching algorithm that runs on O(E) time
DFS	Depth-first search; a recursive or LIFO graph-searching algorithm that runs on O(E) time, where E is the number of edges
EA	Evolutionary Algorithm
EP	Evolutionary Programming; a paradigm of EA
ES	Evolutionary Strategies; a paradigm of EA
FIFO	First in first out; FIFO algorithms are implemented with queues
GA	Genetic Algorithms; a paradigm of EA
GP	Genetic Programming; a paradigm of EA
GPU	Graphical processing unit
LIFO	Last in first out; LIFO algorithms are implemented with stacks
NP	Non-deterministic polynomial time; A problem in NP cannot be solved, but can be verified, in polynomial time
NP-complete	Complexity class within a NP; all NP-complete problems can be transformed to each other in polynomial time
P	Polynomial time; a problem in P can be solved in polynomial time
PEA	parallel evolutionary algorithm
Polynomial time	An algorithm runs in polynomial time if its time complexity is O(f(n)), where f(n) is a polynomial expression
Pseudo-polynomial time	An algorithm runs in pseudo-polynomial time if its running time is polynomial in the *numeric value* of the input, but is exponential in the *length* of the input, i.e. the number of bits required to represent it.
TSP	Travelling Salesman Problem; a classic NP-complete problem
Weakly NP-complete	NP-complete with known pseudo-polynomial time solutions

1 Introduction

The contents of this section draw heavily on Section 1.1 from first semester's report [24].

NP-complete problems are commonly encountered in science, mathematics, and day-to-day life. Examples of common NP-complete problems include scheduling, path-finding, and optimal game-playing. These problems involve deep decision trees with a high branching factor, and cannot be solved in polynomial time with known computational process. Most existing algorithms conduct high precision, sequential brute force searches through each possible path, attempting each possible combination until a correct solution is found. Consequently, time and space complexity increase exponentially with the depth and branching factor of the search tree.

Whereas traditional computational methods fail, the same class of problems have been observed to be easily solvable by biological agents, from simple microorganisms, to biologically complex species such as animals and humans. These agents process information in parallel and use heuristics to create an approximately optimal solution, trading off precision for much greater time efficiency. For instance, time-table scheduling and traffic routing are done routinely and effectively by humans. On a microscopic level, experiments have shown that *Physarum polycephalum*, a slime mold, is able to find the shortest path to nutrients in confined, maze-like environments, and to reproduce maps of train networks when food sources representing major cities are placed on a petri dish [1]. In addition, problems that cannot be solved by individual biological agents, due to the limited processing ability of each, can also be solved by large groups in explicit or tacit cooperation. A fairly recent example is the program Foldit, an online protein-folding game that used crowdsourcing to decipher in ten days the crystal structure of a virus that puzzled scientists for fifteen years [2].

In the search for an efficient solution to NP-complete problems, it would save us a lot of effort to avoid re-inventing the wheel. All of the abovementioned factors indicate that such a solution may already exist in nature. By simulating naturally occurring algorithms, we can develop very efficient and effective methods for solving previously difficult computational problems. These solutions may then be applied to build practical artificial intelligence systems.

2 Background Theory

2.1 Computational Complexity: P vs NP

The contents of this section draw heavily on Section 2.1 from first semester's report [24].

P and NP are two classes of problems frequently discussed in computational complexity theory. P stands for polynomial, and the P complexity class contains the set of all decision problems that can be solved in polynomial time for a deterministic Turing machine, which is a machine where a fixed set of rules prescribes at most one action to be performed at any given state. The NP class contains the set of all decision problems where a given correct solution can be verified by a deterministic Turing machine in polynomial time. An alternative but equivalent definition of NP is the class of problems solvable in polynomial time by a *non-deterministic* Turing machine, i.e. one where more than one action could be performed at any given state. Naturally, NP contains P, but it is currently an unsolved problem in computer science whether P is equal to NP. Another class of problems is known

as NP-hard, which contains the set of all decision problems to which every problem in NP is reducible in polynomial time.

Of particular interest to us is the complexity known as NP-complete, which contains the set of all problems belonging to both NP and NP-hard. In other words, NP-complete is the set of problems in NP to which every other problem in NP is reducible in polynomial time. Therefore, if a polynomial-time solution is found for any known NP-complete problem, the same solution could be applied to find polynomial-time solutions for all other problems in NP.

It is important to note at this point that the algorithms we seek in this project are not "true" polynomial-time solutions to NP problems, and certainly could not answer the P vs NP conundrum. Our algorithms will rely heavily on approximations and are only precise up to a certain confidence level. For the majority of practical applications, however, a quick but imprecise solution is sufficient.

2.2 Graph Theory and Mazes

The contents of this section draw heavily on Section 2.2 from first semester's report [24].

A graph is a data structure where a set of objects, called vertices, are connected by undirected or directed links, called edges. A complete graph is a graph that contains a path from any one of its component vertex to any other. In this project, we assume that we will only be dealing with connected graphs.

A tree is a connected graph with no cycles, meaning that one cannot start at a vertex, follow a consecutive series of adjacent vertices, and arrive back at the same vertex.

Graph theory is commonly used to solve problems in NP and NP-hard, as one can model a large number of these problems after a graph structure and apply conventional graph traversal algorithms to find a solution. A well-known NP-hard problem in combinatorial optimization is the Travelling Salesman Problem (TSP), which has several applications in planning, logistics, and microchip manufacturing. TSP can be formulated as such: given a list of cities and the cost of travel between them, find is the lowest-cost route that visits each city once and returns to the original city. Using graph representation, each city is represented as a vertex and each route between two cities, an edge.

The naïve approach to solving TSP is to use brute force search, traversing every possible combination of cities until a solution is found. While possible when the number of cities is small, this algorithm is O(n!). It suffers from combinatorial explosion, namely, the extremely large growth in processing time as the number of vertices (and consequently, edges) is increased.

By setting search constraints, however, complexity can be greatly reduced. The branch-and-bound algorithm, for example, heuristically generates an initial solution to the problem. It then "branches out" by making small modifications one at a time to the solution, storing the best solution so far, until a correct solution is found. The number of permutations is much smaller in this case, and the algorithm returns a satisfactory solution quickly.

Uninformed maze-solving is an NP-complete problem with a complexity exponential to the branching factor of maze cells. Mazes can be represented with a graph by modelling each cell as a vertex, and each path connecting two cells as an edge. In addition, the entrance and exit of the maze are also modeled as vertices. The problem then becomes one of finding a path—usually the shortest path—from the entrance vertex to the exit vertex. Further problems are introduced when different types of

mazes are considered. A standard, or perfect maze contains no loops, and its graph representation is a tree. Imperfect mazes, meanwhile, contain loops and hence several solutions from entrance to exit. Mazes can also be simply-connected, that is, have all of its walls connected to the outer boundary, or disjoint. For informed maze-solving, where the entire maze is visible, best-first heuristic algorithms such as A* can be used. For uninformed search, however, currently existing algorithms often use brute force.

2.3 Behavior of Slime Molds and Fungi

The contents of this section draw heavily on Section 2.3 from first semester's report [24].

Despite not having a central nervous system, simple organisms such as slime molds and fungi have been observed to display "intelligent" behavior in both natural and laboratory environments. Before analyzing the fungus we are provided with, we briefly researched *Physarum polycephalum*, a slime mold whose maze-solving behavior was previously discovered, and has inspired several biological algorithms.

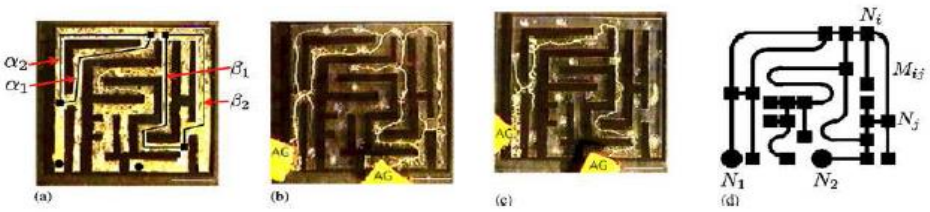


Figure 1. Physarum Polycephalum finds the shortest path in a maze [1]

P. polycephalum is an amoeboid protist that appear as a slimy, yellow network of protoplasmic strands, called a plasmodium. In 2000, scientists at Hokkaido University discovered that, by scattering pieces of *P. polycephalum* plasmodium in a maze, the slime mold slowly grew to fill every path [1]. Placing food at the entrance and exit of a maze would cause the slime mold to retract its branches from dead-end corridors and eventually form the shortest path possible between the food sources [1]. Furthermore, *P. polycephalum* has been found to possess a form of special memory; by crawling across a surface, it leaves a trail of slime, and avoids crossing this trail should the slime mold encounter it again [1]. Figure 1 below illustrates this behavior.

Other experiments have found similarly intelligent behavior in this species under different situations. When researchers placed oat flakes in the same position as major cities and urban centers on a map, and placed a piece of *P. polycephalum* plasmodium on the same map, it at first grew to fill the entire map [4]. But shortly after, the excessive strands retracted, and what was left was a network system connecting food sources that is remarkably similar to human-designed railway and road systems as illustrated by Figure 2. In other words, the plasmodium did not branch out to food in a randomized manner; instead, it sought to conserve energy by finding the most efficient route connecting food morsels, governed by some natural algorithm [1].

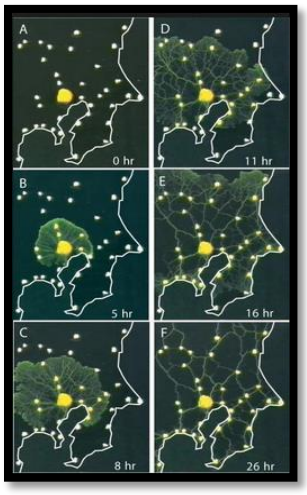


Figure 2. P. Polycephalum networks [4]

Higher fungus, in both phyla Basidiomycota and Ascomycota, demonstrate a similar form of intelligence [3]. The fungus used in this research are multicellular and filamental, and grows by extending a specialized type of cell, called hyphae, without cell division. In both natural and controlled environments, filamental fungi have been observed to conduct efficient space search when exploring microconfined spaces with their hyphae. On a microscopic scale, at mycelial level, hyphal branches are driven by a variety of sensing mechanisms at the extending apex [3]. This way, the fungus can react to its environment, and behave according to dynamic information such as available space, light distribution, and concentration of nutrients or toxins. Unlike slime molds, which grow rapidly to maximize area coverage and then prunes off unsuccessful branches, a fungus uses dynamically obtained information about its surroundings to determine, immediately, where its branches should grow [3]. When grown in a maze, the fungus shows a consistently high success rate and low exit time [3]. Several of its strategies have been identified, such as collision-induced branching and directional memory, discussed in detail in the Design and Results section of this report. Analyzing the strategies used by this fungus is crucial for us to formulate an efficient, bio-inspired graph searching algorithm.

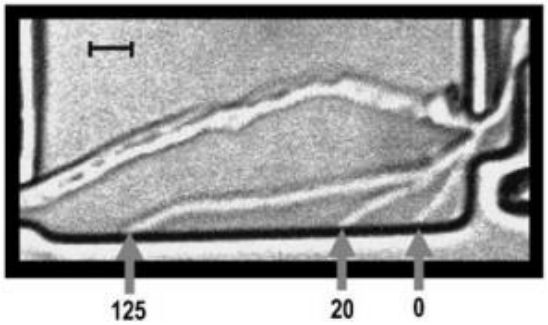


Figure 3: Composite image of several fungal hyphae [3]

2.3.1 Directional Memory

The contents of this section draw heavily on Section 4.2 from first semester's report [24].

The hyphae of the filamental fungi we observed exhibits directional memory. While complex wall geometry can and do guide the overall direction of growth of a hyphal filament, the hypha will change back to its original direction of growth when possible [3]. Each hypha enters the maze from a specific angle. When the apex of a hypha meets an obstacle, it noses the wall of the obstacle and is forced to grow in a direction parallel to that wall. However, when the wall ends, the hypha immediately reverts to its starting angle. [3] This behavior is exhibited for all convex, circumventable obstacles encountered in the maze, and at all hyphal lengths, meaning that no matter how far the hypha has grown, it will always follow the growth direction determined at branch emergence [3].

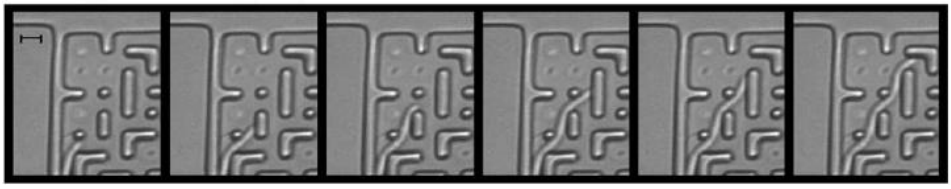


Figure 4: Fungal hypha exhibiting directional memory [3]

2.3.2 Collision-Induced Branching

The contents of this section draw heavily on Section 4.2 from first semester's report [24].

When a fungal hypha collides with a convex corner, it can exhibit one of two different behaviors. 21% of the time, the hyphal tip turns and begins following the wall; 79% of the time, however, a new behavior is observed: the tip simply stops changing directions, and instead continues extending into

the corner, causing the hypha to bend [3]. This in turn triggers new branches to emerge at sub-apical areas of the fungus, which grow into unexplored regions of the maze. These sub-apical hyphae have their own emergent angles different from the branch they emerged from, and follow directional memory. Whether the fungus follows a corner or induces branching depends on the initial branching direction: the apices only turn around a corner if the resulting growth direction is less than 93° to 94° [3]. Should this threshold be exceeded, sub-apical branching is induced.

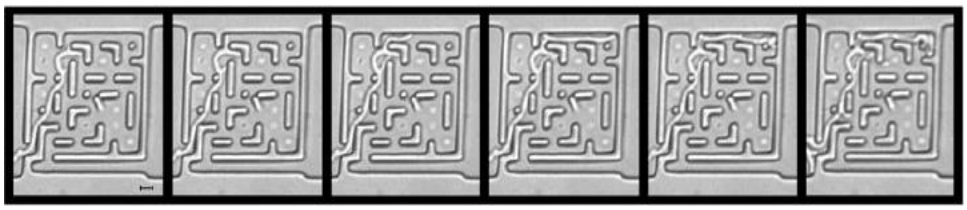


Figure 5: Fungal hypha exhibiting branching upon corner collision [3]

By growing sup-apical branches instead of turning sharp corners, the fungus avoids the continued growth of branches that are stuck in dead ends, and instead explores new regions. This behavior resembles a modified version of Trémaux's algorithm, also known as depth-first search (DFS), a recursive algorithm designed to solve mazes by exploring each possible path as far as possible until a dead-end is reached, at which point it searches another path [9].

3 Project Requirements and Constraints

Objective

The contents of this section draw heavily on Section 1.2 from first semester's report [24].

One major class of NP-complete problems is maze-solving, a subclass of routing problems. In this project, we aim to create a bio-inspired algorithm for efficient maze-solving by observing the natural strategies used by microscopic, multicellular fungi. Similar to the previous experiments done using *Physarum polycephalum*, in this project we observe and simulate the behavior of several species filamentous fungi, all of which have demonstrated "intelligent" characteristics in exploring and colonizing microconfined, mazelike networks [3].

In this design project we aim to produce a better uninformed algorithm than depth- and breadth-first search inspired by the behavior of the fungus that produce a satisfactory solution to 2D mazes. The performance of the developed algorithm is evaluated in terms of running time and level of precision based on the optimality the solution (where the optimal solution is the one with shortest exit time).

Product Format

The final product is a cross platform Java desktop application. The result of this project will be limited to software, as the potential applications of the algorithm to other theoretical and real world problems are outside of the scope of this project. The final product includes a module for maze generation and another for maze solving. The user is allowed to customize certain variables in either module, such as maze size and generation algorithm, and observe the simulated fungus solve the

maze in real time. Additionally, a game component is added where the user can simultaneously compete against the program to solve a maze. Here, the algorithm is slowed down in order to provide a fair user experience.

Software Requirements

- programming language: Java
- cross-platform
- Java based user interface
- Eclipse IDE

Team Background

This project is part of 2-semester design project developed for a required undergraduate course in the faculty of Electrical, Computer and Software Engineering in McGill University. Both team members have multiple experiences designing applications and algorithms in Java, and analyzing their computational performance. Additionally, Eileen F. has taken several courses in artificial intelligence, while Elitsa A. has taken courses in parallel computing. The combined technical knowledge, along with experience working in engineering design teams were essential for this project.

Target Audience

The final product of this project is aimed at researchers and developers in the fields of computer science and engineering. Developing efficient pathfinding algorithms is a major topic of research in the computer science. These algorithms can be used to solve theoretical optimization problems such as TSP, the open-shop scheduling problem, and the circuit satisfiability problem. With a little modification, they can in turn be adapted to real world applications such as DNA sequencing, microchip design, vehicle routing, and motion-planning in robotics.

Budget

The first semester for this project was allocated to research and planning. During this semester we have accomplished the following: problem identification, analysis of criteria and constraints, evaluation of implementation strategies, and development of a preliminary algorithm.

The second semester was allocated to the design, implementation and testing of the software product (see section 4). Maze generation and user interface were allocated for weeks of the project since they don't involve as much work.

The time allocated to this project per week is similar to that of a typical 3 credit undergraduate course. For each member of our team, an average of 3 hours per week were allocated for the project, including team and supervisor meetings (a total of 84 hours/student, assuming 14 weeks semester). Our estimates were very close to the actual time spent as shown by Table 1. Figure 6 shows a timeline of our work.

Table 1 Task duration semester 2

	Time spent (in hours)						
	Algorithm design	Algorithm implementation	User interface	Game	Testing	Documentation & Meetings	Total
Elitsa A.	5	10	15	10	18	27	85
Eileen F.	15	35	20	0	7	10	87

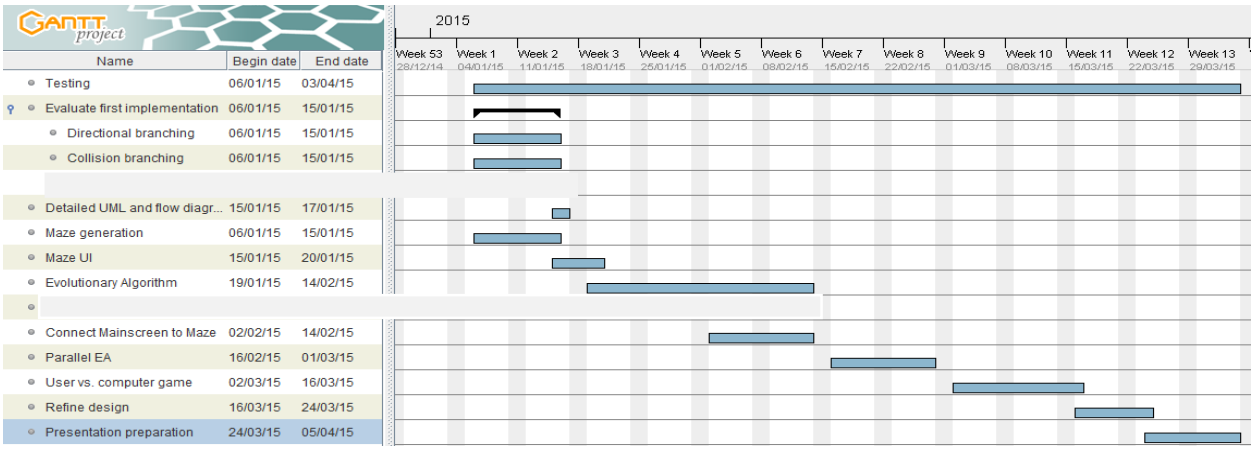


Figure 6 Gantt chart of project timeline

4 Design and Implementation

Note: Class Diagram of the project can be found in Appendix A.

4.1 Work from first semester

In the first part of this design project we identified a computationally hard pathfinding problem. Specifically, we concentrated on maze solving since it has vast applications in graph theory and artificial intelligence navigation. The next step was learning and researching the strategies various microorganisms, such as fungi, bacteria, and amoeba, used to navigate in microconfined spaces. Our final algorithm design is inspired by, and based on, our supervisor's study on the maze exploration behavior of filamental fungi [3], described in the section 2. After exploring several possible implementation strategies, such as heuristic branch and bound and the application of evolutionary algorithm, we ultimately decided to focus on realistically simulating the observed behavior of fungi, and concentrate on the two main fungal strategies described in 'Fungi Use Efficient Algorithms for the Exploration of Microfluidic Networks' [3] : directional memory and collision-induced branching.

4.2 Software tools

Programming Language

The programming language for the algorithm design and the user interface of the program is Java. Its support for concurrency, object-oriented programming, and the availability of intuitive graphical design libraries are features that greatly benefit many of our design decisions.

By supporting concurrency through its easy-to-use Thread library, Java provides the ability to simulate parallelism through multithreading, allowing our algorithms to performs many tasks simultaneously. While with a single processor, Java threads are not truly parallel, on multi-core computers the operating system may assign each thread to a different processor and implements true parallelism.

By being object-oriented, Java supports the abstraction of data structures into objects, thereby allowing the intuitive implementation of abstract data types used in this project. Additionally, a large number of external libraries are available for Java, which may be used to supplement available class libraries and simplify our design.

Development Environment

The Eclipse IDE (integrated development environment) was chosen since it is one of the most advanced and supported development environments for Java and both members are familiar with its functionalities. Below is a list of some of its components used for this project:

- Debugger: used to debug the app
- Junit : used for testing separate classes and their functions
- Javadocs: used to generate API documentation in HTML format from doc comments in source code
- Executable JAR: used to package all of the classes and create an executable .jar file with which to run the applet
- Git: version control for easier collaboration between team members and version retrieval
- Swing library : used to create main screen and the maze animations

4.3 Maze Generation

Since maze-solving can be generalized to graph searching, it is natural that our maze is generated by a graph-based algorithm, as shown by Figure 7. Specifically, the graph was implemented in the form of a w-by-h grid of cells (w and h are integers given by the user and represent the width and height of the maze). Each cell represents a square on the grid, which has 4 walls, and are surrounded by 4 other cells. If each cell is the vertex of a graph, then a cell may be connected to another cell if there are no walls between them. Each cell can have from 0 to 4 walls; 0 walls means the cell is a 4-way intersection, and 4-walls means the cell is completely closed off.

The maze is initialized to be a two-dimensional array of disconnected vertices, visually represented to be rectangular grid. Two randomized options are provided for maze generation, each following a separate algorithm for removing walls in the grid, allowing for two “styles” of mazes to be created for testing and simulation. The first algorithm implements a queue-based depth-first-search. Starting

from the entrance cell, provided by the user, a random wall leading to an adjacent cell is removed, and that cell is marked visited. The algorithm then proceeds to remove a random wall in the newly-visited adjacent cell, and the next, until the user-supplied exit point is found and all cells have been visited. This results in a maze containing many long corridors, and few branches.

The second option implements Kruskal's algorithm. In this case, the initial maze of closed cells can be seen as a grid of disjoint sets, each containing only one vertex. A separate array containing all the walls is also created. Then, at random, walls are chosen out of this array; if the wall divides two cells that belong to separate sets, the wall is removed and the two cells are joined into one set. This is repeated until all cells belong in a single set. In contrast to DFS, Kruskal's algorithm results in a maze that branches frequently in all directions, and contains few long paths.

Finally, for each of the two maze generating options, the user may choose to create either a perfect or imperfect maze. A perfect maze is created by default, and allows only one solution. An imperfect maze can be created by randomly removing more walls from a perfect maze.

A graph-based implementation has the advantage of being easy to modify. For example, by changing the maximum number of walls from 4 to 6, we can create a hexagonal maze. By randomizing the maximum number of walls for each, we can create an entirely random maze. Due to animation constraints, these options were not implemented, although current data structures in the code provides a reconfigurable framework should work on this project be continued in the future.

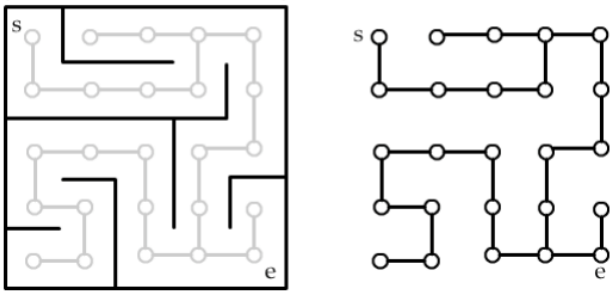


Figure 7 Maze as a graph data structure

Figure 8a shows a snapshot of the main screen of the application. Figure 8b shows the fungus simulation screen. Several options exist to allow users to customize maze generation.

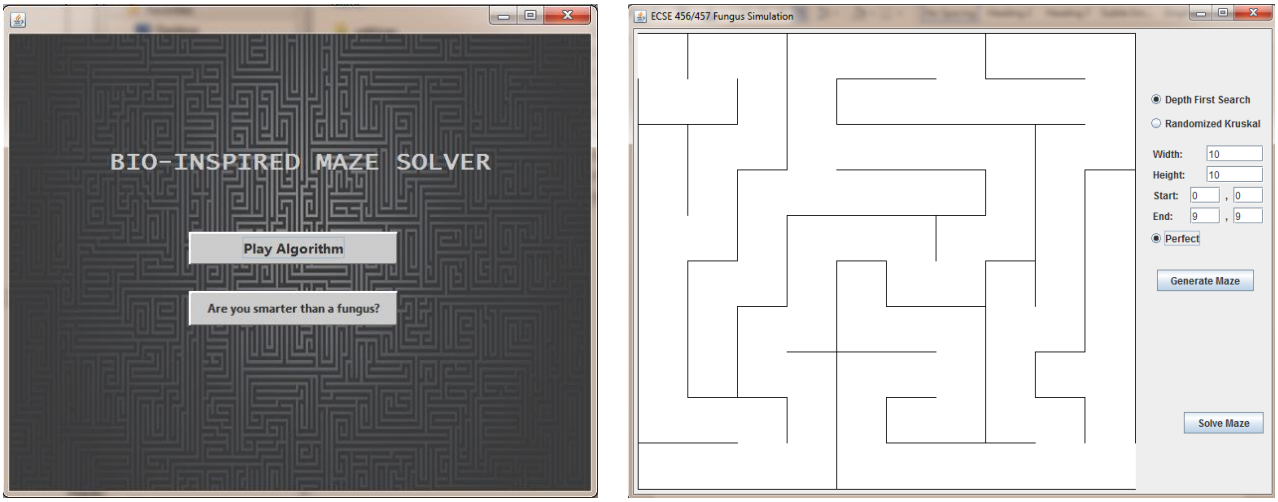


Figure 8 Main screen and maze generation screen

4.4 Directional memory implementation

The algorithm for directional memory is based on the Pledge algorithm, which was devised to solve any maze whose entrance and exit are on the outside walls [9]. The Pledge algorithm begins with an arbitrary direction that the solver will try and go towards. When an obstacle is met, the solver will follow the wall of the obstacle at a direction closes to the original, and begins counting the angle turned [9]. For example, in a rectangular maze, turning right at an intersection is 90°, while turning left is -90°. When angle count reverts to 0°, the obstacle no longer blocks the initial direction. The solver must now stop following the obstacle wall and continue in the original direction [9].

Pledge algorithm is guaranteed to solve a maze, but if used exclusively by a fungus, it is inefficient, requiring backtracking at dead ends. To avoid 180° turns and U-shaped growth, which wastes valuable resources, the fungus have been observed to use another strategy, namely collision-induced branching.

Directional memory is easily implemented by including the starting angle of each branch object as a class constant, and the current angle of the branch as a class variable. A simulated fungal branch will travel in its starting angle as far as possible, whenever possible, and only changes its current angle when no paths are available in its starting direction.

Figure 9 displays two different instances of the algorithm behavior based on directional memory.

Figure 9 Directional memory

4.5 Collision-induced branching implementation

As previously described in section 3.2.2, the fungal algorithm is slightly different from standard depth-first search (DFS). While standard DFS considers a cell to be a dead end only if it leads to no unvisited nodes, the fungus instead sees any node that requires a great degree of turning to be a dead

end [3]. However, the difference here is trivial; in cases where the fungus collides with a corner and refuses to turn, even when unvisited cells can be reached from that corner, it instead grows new hyphal branches into the unvisited cell [3]. Programmatically, the above two cases can be seen as equivalent. Another difference is that, when branching is induced, the new hyphal branch does not necessarily grow from the last visited cell that has paths leading to unvisited cells. From an algorithm perspective, instead of using recursion or a stack, cells were stored in a data structure that allows direct access, in this case, an array. This way, when a dead end is reached, the next cell to visit is determined algorithmically rather than using a LIFO method.

Figure 10 displays various stages of the collision-induced algorithm behavior.

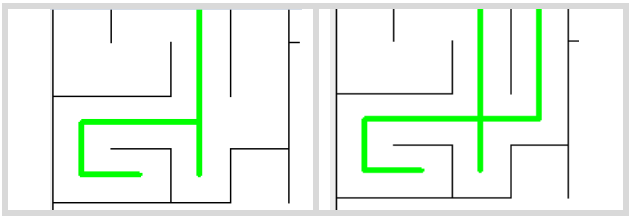


Figure 10 Algorithm stages showing collision-induced implementation

4.6 User vs. computer game

The last component of our design is a user vs. computer feature where the user can compete against the designed algorithm on a predefined maze. The maze explorer is an open source project [25]. No open source projects were found of 3D maze explorers, however this applet [25] shows a very small section of the maze (Fig. 11), thus mimicking the first person maze experience. To provide a fair user experience, the algorithm was slowed down

Figure 11 Game view

5 Results and Tests

Note: All tables corresponding to the figures below can be found in Appendix B.

Test A: Reachable state space

Mazes can serve as a background to state-space searching because it is composed of an environment (the maze) that is divided into equally sized units (states). For this test, we defined a start state (beginning of the maze) and final state (end of maze) and counted the number of covered nodes. Since this an uninformed search (i.e. the search is the same no matter the context), we estimate that the bioinspired algorithm (BIA) will be less efficient than informed search but better than a regular depth- or breadth- first search since the fungi simulation branches whenever it encounters a dead end. We also use A* as an informed search algorithm, in order to make a full comparison. In Fig. A, the first plot shows the progression of state exploration on a non-randomized maze generated by DFS (limited dead ends) while the second plot is the same exploration on randomized maze created by Kruskal's algorithm (multiple dead ends).

- **Results and Discussion:**

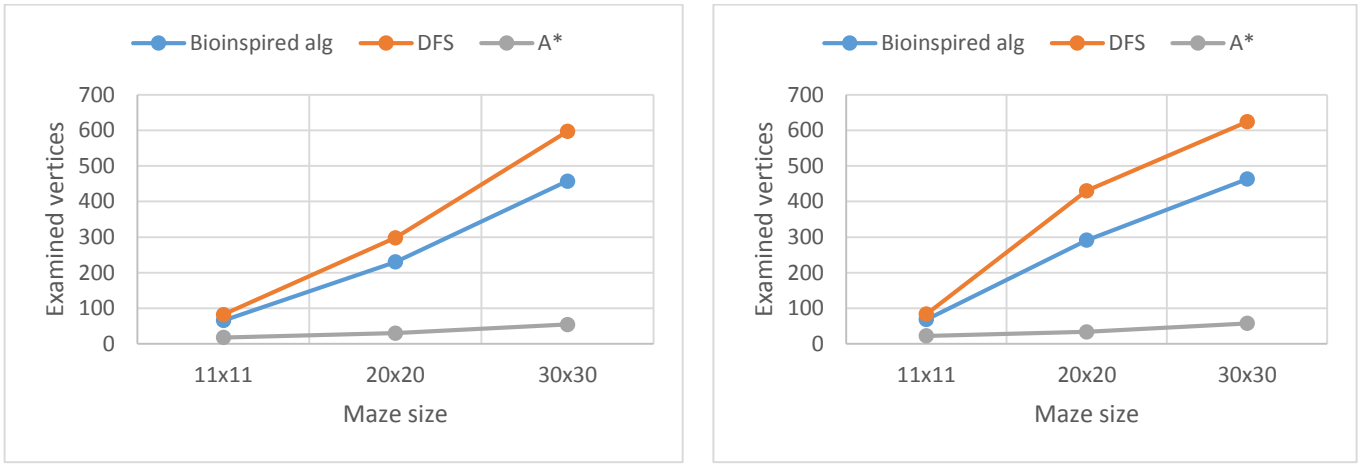


Figure J Reachable state space a) non randomized space and b) randomized space

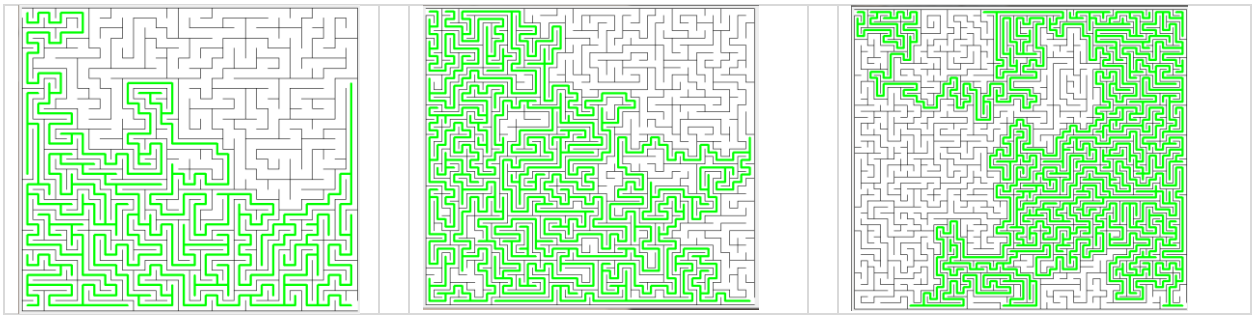


Figure K Simulations of BIA alg. for 20x20, 30x30 and 50x50 maze sizes

As shown by Fig. A (Table 1 and 2 in Appendix B), DFS takes the most space when performing maze search, while BIA is ~20% more efficient in larger mazes (30x30 and up). A*, being an informed search, as expected is much more compact.
Also, it is important to note the different between randomized and nonrandomized maze. In the nonrandomized maze generated by DFS (left side of Fig. A), BIA and DFS are more similar in performance due to the limited number of dead ends. With randomized mazes (generated by Kruskal's algorithm) where they are multiple dead ends, the difference between BIA and DFS is visible even with smaller mazes.

Test B: Completeness

Starting from a root node (beginning of maze), this test aims to find to what extent the bioinspired algorithm finds the leaf node (end of maze). The tests were run on different maze sizes and by placing the starting and ending vertices at various positions.

- **Results and Discussion:**

As shown by Table 3 in Appendix B, the end of the maze is found 100% of the time. Upon encountering a dead end, the algorithm goes back to a previously branching point and resumes from there. While it might not be the best solution, the algorithm will always find the exit on finite mazes.

Test C: Optimality

The objective of this test is to determine the effectiveness of the algorithm in finding a least-cost solution. Since the exit of the maze is unknown, we expect an average of 50% chance of finding the shortest path.

- **Results and Discussion:**

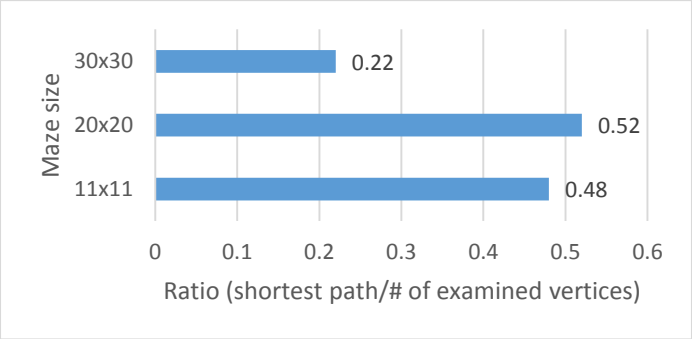


Figure L Shortest path effectiveness in different maze sizes

As shown by Fig. C (Table 4 in Appendix B), the reliability of finding the shortest path starts from ~50% in small maze sizes (11x11) and decreases with larger maze sizes. It is important to note that real fungi, tested under laboratory conditions, can pick up various indications in the solution as to where the target is and find it faster. For a more effective search, it can be useful to model those condition, if provided the necessary information.

Test D: Comparison with other maze solving algorithms

1) BIA vs. DFS

This test aim to provide a comparison of BIA against another uninformed maze search. The test was performed multiple times (where the averages is displayed in Fig. D) for different maze sizes. The mazes generated were non-randomized. Experiment run on Dell Inspiron i3.

- **Results and Discussion**

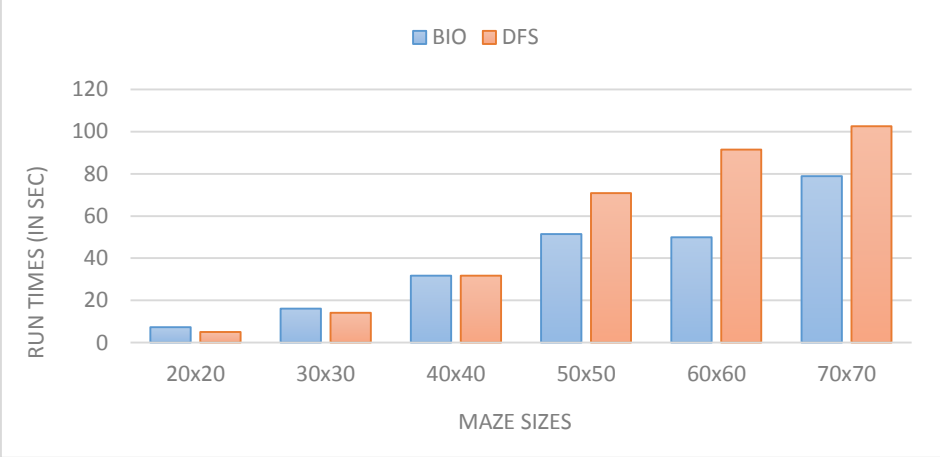
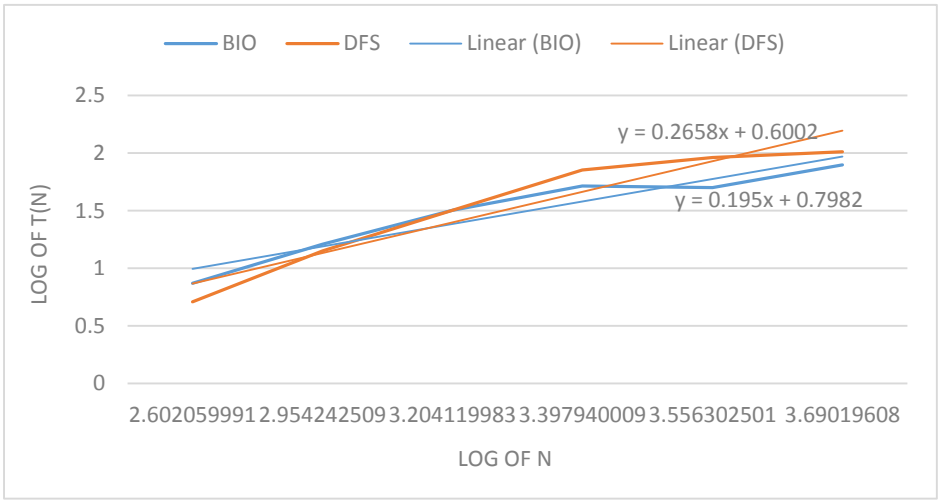


Figure M BIA vs DFS execution times on different maze sizes

As shown by Fig. D (table 5 in Appendix B), in smaller maze sizes (up to 30x30), DFS performs slightly better. However, at 40x40 there is an equilibrium after which the bio-inspired algorithm performs 20-40% better (tested until 70x70).
If we assume the dominant term of the running time is of the form

$$T(n) = cn^k$$

where n is the maze size then from Fig. N, we can derive the following asymptotic complexity of BIA and DFS.



$$DFS = 0.27n^{0.6}$$
$$BIA = 0.2n^{0.8}$$

Figure N Log T(n) vs. log n

2) BIA vs. informed maze search algorithm

Table 2 shows the experimental run times of the designed algorithm (i.e. bio-inspired algorithm) and other popular algorithms for maze solving for different maze sizes. Algorithm for this test is run on a Dell Inspiron i3. The mazes generated were non-randomized.

➢ **Results and Discussion**

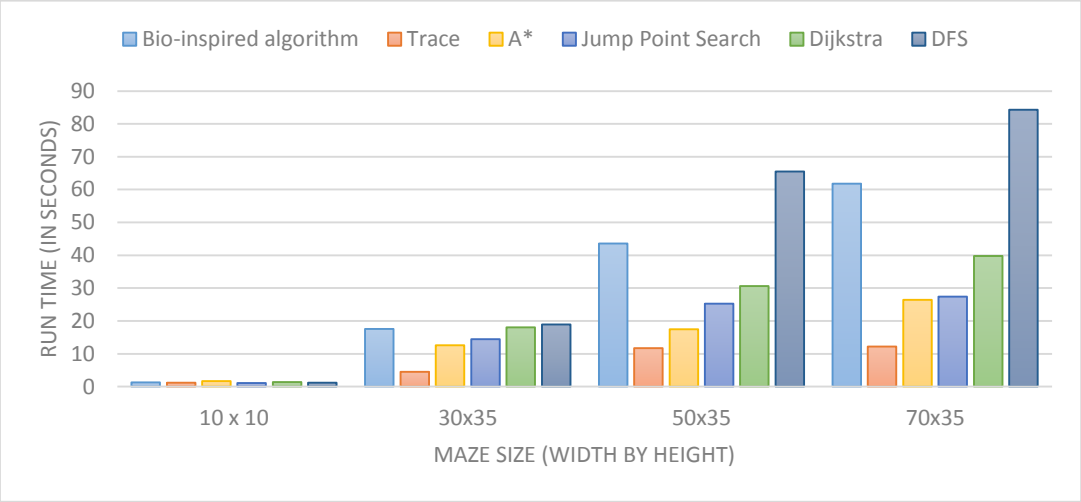


Figure O Comparison of pathfinding algorithms

As expected, Figure E (table 6 in Appendix B) shows that for large maze sizes, the execution time increases significantly for uninformed algorithms (BIA and DFS in this example).

Results Recap

Compared to another uninformed maze search such as DFS, the BIA performs 20-40% better both in space search and in running time. We also noticed that as maze sizes increase, our algorithm performance becomes much more prominent against DFS. This is due to the directional branching and the collision-induced branching strategies. Moreover, if new branches could be made to explore in parallel, the algorithm could rival even informed pathfinding algorithms.

6 Impact on Society and Environment

In this project, we have designed a computer application. As with most software products, the environmental impacts and potential risks to users are negligible. All data and information concerning the various biological agents were previously collected and directly provided to us, and thus neither team member risked potential contamination with biological material.

In terms of manufacturing costs, we have to account for human resources needed for researching, designing, implementing, testing, documenting and maintaining the application. Consumers cost depends largely on the commercial use of the designed product/algorithm. So far, this bio-inspired algorithm is only implemented on an academic level and could be further improved by increasing levels of parallelism and by developing a more advanced heuristic technique.

Additionally, we tested the algorithm on different multicore machines (dual, quad, and hexa), but in order to fully explore its capabilities we would require the use of expensive and difficult to access distributed systems and supercomputers.

In terms of benefits, this algorithm can provide solutions (up to a certain graph size) to computationally intense graph based uninformed maze-solving. While the data structures implemented in the current version of software is tailored to maze solving, it could be reconfigured and adapted to other problems such as travel planning and production scheduling. An even more interesting application is in artificial intelligence systems that require autonomous navigation. Most autonomous robots are inspired by some degree by natural behaviour. With the emergence of micro robots, where detail in design is crucial, navigation algorithms inspired by natural behavior is an attractive alternative to traditional programs. By modelling filamental fungi strategies in our algorithm, we provide a new way for autonomous self-navigation in microconfined spaces for small and distributed navigating components.

The social impact of this project lies in its interdisciplinary aspect. It unifies concepts and experimental methods in biology with engineering design, while taking into account the costs, risk and environmental impact of the both disciplines. The fields of biomimicry and bioengineering frequently observe and employ natural behaviors and algorithmic strategies of living organisms to engineering applications—in our case, NP-complete pathfinding problems. As students of software and computer engineering, with background and interest in the life sciences, the project was a valuable experience in interdisciplinary research and design.

7 Report on Teamwork

During the first semester, both members worked closely together to identify a problem, conduct research and discuss findings, ideas and possible implementation strategies. Towards its end, there was a clear idea on what the product will look like and the strategies implemented in the next half of the project. We assigned tasks to both team members, and designed a tight schedule that nevertheless allowed for flexibility. The second part of the project required more individual work and below is a summary of each member's contribution.

With background in artificial intelligence (COMP-424 and ECSE-526), Eileen Fu was able to evaluate the extent to which an evolutionary approach was possible in this project. Upon discussion, both members decided to concentrate on the implementation of two fungal strategies described in the sections above, as it was decided that an evolutionary algorithm would be exceedingly difficult to implement based on available background information on the species itself. While Eileen Fu worked primarily on maze generation and solutions, Elitsa Asenova designed the UI and provided the crucial framework for threaded animation. Both members reviewed their code regularly using GitHub, and stayed in touch with regular meetings, both in person and through social media. While Eileen F. handled most of the internal logic of the program, Elitsa A. designed the main screen of the app and connected with it the game component and the generated maze. She also performed and documented the testing of the algorithm.

Overall, the team worked very well together, communication was fast and efficient and the work was fairly distributed. The only note in approaching future design projects would be to not underestimate the final integration of all of the components and the testing of the entire system. For instance, due to being developed separately, some issues were had when Eileen F. first attempted to animate a multi-branched algorithm. A threading error caused old branches to disappear whenever a new branch is generated. This problem was eventually solved through better communication. As a result, we realized that, while single components might work correctly alone, unexpected errors and performance issues often arise once they are combined. Continual integration and testing is therefore of primary concern.

Conclusion

In this project, we developed a bio-inspired maze-solving algorithm polynomial up to an arbitrary maze size. The design was based on the naturally intelligent behavior of microscopic fungi, where a space search was made more efficient with the use of directional memory and the collision-induced branching. Testing the algorithm with different maze sizes and on commercial computers showed that there is still space for improvement in terms of speed. For example, the directional memory could be implemented to follow exactly the fungus behavior by remembering the exact original angle (and not only in increments of 90°). This would be interesting when dealing with autonomous robots and when solving irregular mazes. Also, higher parallelization scheme and the use of powerful computers would allow for a more optimized computation. This project takes a multidisciplinary approach to software engineering by unifying biological studies and engineering design methods, and the product has the potential to find a wide range of applications in theoretical computer science and artificial intelligence.

References

Note: the majority of the articles (unless specified) were found on the Engineering Village database via a McGill student access or via a McGill library request.

[1] Nakagaki, Toshiyuki; Yamada, Hiroyasu; Tóth, Ágota (2000, September 28). "Intelligence: Maze-solving by an amoeboid organism". *Nature* 407

[2] Solve Puzzles for Science | Foldit. (n.d.). Retrieved November 23, 2014, from http://fold.it/portal/

[3] Hanson, K., Nicolau Jr, D., Filipponi, L., Wang, L., Lee, A., Nicolau, D. Fungi use efficient algorithms for the exploration of microfluidic networks. (2006).

[4] Tero.A.,Takagi,S.,Saigusa,T. and al. Rules for Biologically Inspired Adaptive Network Design. *Science*. Vol.327 no.5964 pp.439-442 DOI:10.1126/science.1177894

[5] Michalewics, Z., Hinterding, R., Michalewicz, M. Evolutionary Algorithms.

[6] Sudholt, D. "Parallel Evolutionary Algorithms". Chapter in the Handbook of Computational Intelligence. University of Sheffield.

[7] Alba, E. "Parallelism and Evolutionary Algorithms". IEEE Transactions on evolutionary computation. Vol. 6, No.5. October 2002.

[8] Alba, E. "Parallel evolutionary algorithms can achieve super-linear performance". Information Processing Letters 82 (2002) 7-13. University of Malaga.

[9] Abelson; diSessa (1980), *Turtle Geometry: the computer as a medium for exploring mathematics*

[10] Yuan, C., Wu, X., Hansen, E. Solving Multistage Influence Diagrams using Branch-and-Bound Search

[11] Pappu, S. Evolutionary Algorithms (Project Report). Sardar Patel Institute of Technology. October 11.

[12] Cervone, G., Michalski, R., Kaufman, K., Panait, L. Combining Machine Learning with Evolutionary Computation : Recent Results on LEM. George Mason University.

[13] Wong, M-L., Wong, T-T., For, K-L. Parallel Evolutionary Algorithms on Graphics Processing Units. Available on: http://www.gpucomputing.net/sites/default/files/papers/3004/cec2005.pdf

[14] Chiu, D., Pezzoli, E., Wu, H., Stroock, A., Whitesides, G. (2001, January 8). Using three-dimensional microfluidic networks for solving computationally hard problems.

[15] Zhang, X., Zhang, Y., Wei, D., & Deng, Y. (2012). Solving shortest path problems with interval arcs based on an amoeboid organism algorithm. *Journal of Information and Computational Science*, *9*, 2081-2088.

[16] Xiaoge Zhang; Yajuan Zhang; Zili Zhang; Yong Deng, "A method to solve shortest path finding in directed graph based on an amoeboid organism," *Control and Decision Conference (CCDC), 2012 24th Chinese* , vol., no., pp.3699,3702, 23-25 May 2012. Available on: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6243094&isnumber=6242916

[17] Chen, J-C., Conrad, M. A multilevel neuromolecular architecture that uses the extradimensional bypass principle to facilitate evolutionary learning. *Department of Computer Science, Wayne State University Detroit*.

[18] Park, D. Application of Horton's first and second laws of branching to fungi. *Botany Department, The Queen's University, Belfast.*

[19] Choubey, N. A-Mazer with Genetic Algorithm. International Journal of Computer Applications (0975-8887) Volume 58-No.17. November 12.

[20] Nicolau, D.Jr, Hanson, K., Nicolau, D. Modeling of the Growth of Filamentous Fungi in Artificial Microstructures. University of Oxford.

[21] Pipe, A., Fogarty, T., Winfield, A. Balancing Exploration with Exploitation – Solving Mazes with Real Numbered Search Spaces. University of the West of England.

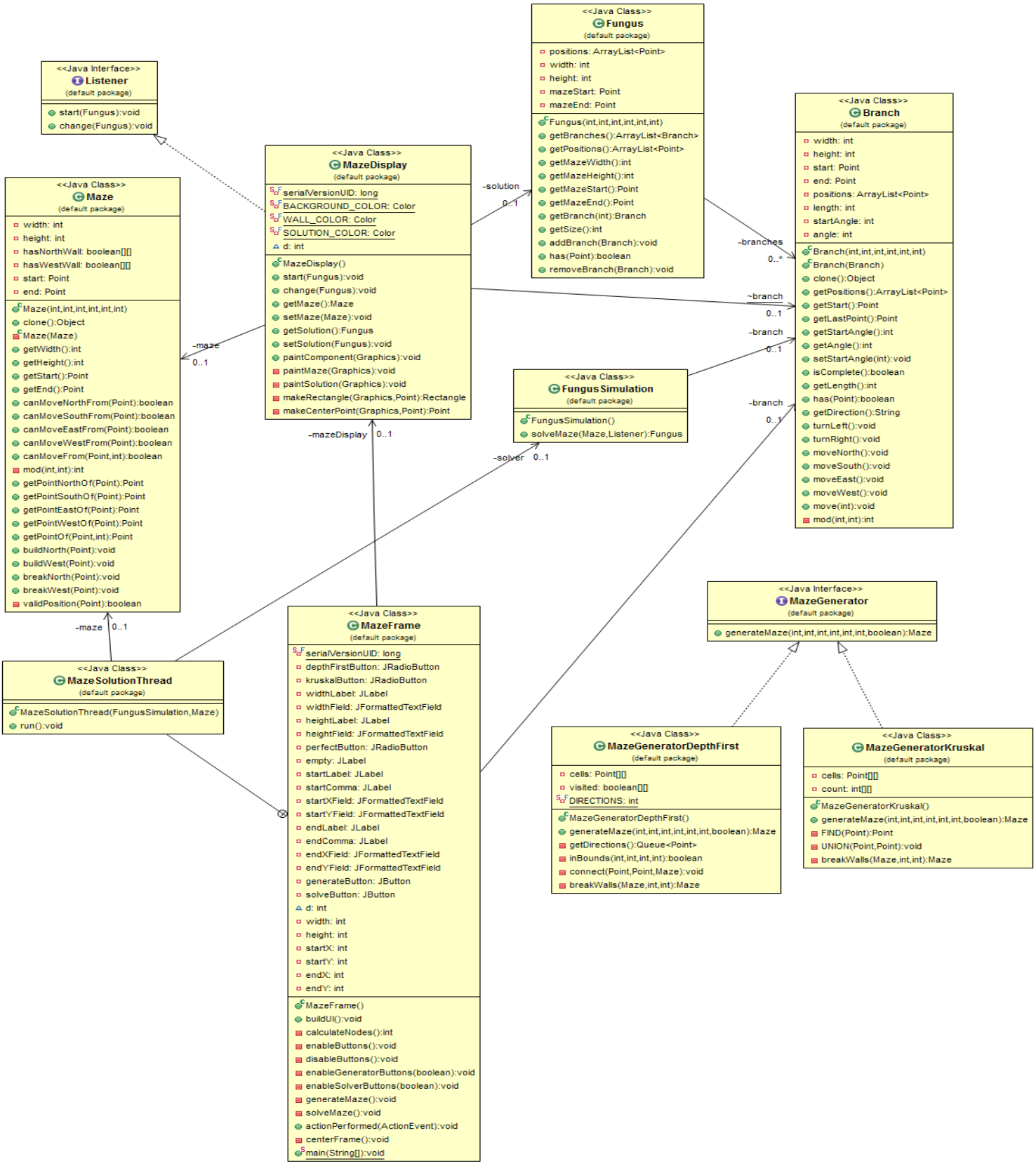
[22] Held, M., Edwards2, C., Nicolau, D. Fungal Intelligence; Or on the behaviour of microorganisms in confined micro-environments. University of Liverpool.

[23] Nicolau, D.Jr, Hanson, K., Nicolau, D. Modeling of the Growth of Filamentous Fungi in Artificial Microstructures. University of Oxford.

[24] Fu, E., Asenova, E. Biocomputation with 'Smart' Biological Agents. Final Report ECSE 456. McGill University Fall 2014.

[25] King, C. The Maze. Java Application. April 2013. Found on Github.com

APPENDIX A


<<Java Interface>>
Listener
(default package)
start(Fungus):void
change(Fungus):void
<<Java Class>>
Fungus
(default package)
positions: ArrayList<Point>
width: int
height: int
mazeStart: Point
mazeEnd: Point
Fungus(int,int,int,int,int,int)
getBranches():ArrayList<Branch>
getPositions():ArrayList<Point>
getMazeWidth():int
getMazeHeight():int
getMazeStart():Point
getMazeEnd():Point
getBranch(int):Branch
getSize():int
addBranch(Branch):void
has(Point):boolean
removeBranch(Branch):void
<<Java Class>>
Branch
(default package)
width: int
height: int
start: Point
end: Point
positions: ArrayList<Point>
length: int
startAngle: int
angle: int
Branch(int,int,int,int,int,int)
Branch(Branch)
clone():Object
getPositions():ArrayList<Point>
getStart():Point
getLastPoint():Point
getStartAngle():int
getAngle():int
setStartAngle(int):void
isComplete():boolean
getLength():int
has(Point):boolean
getDirection():String
turnLeft():void
turnRight():void
moveNorth():void
moveSouth():void
moveEast():void
moveWest():void
move(int):void
mod(int,int):int
<<Java Class>>
MazeDisplay
(default package)
serialVersionUID: long
BACKGROUND_COLOR: Color
WALL_COLOR: Color
SOLUTION_COLOR: Color
d: int
MazeDisplay()
start(Fungus):void
change(Fungus):void
getMaze():Maze
setMaze(Maze):void
getSolution():Fungus
setSolution(Fungus):void
paintComponent(Graphics):void
paintMaze(Graphics):void
paintSolution(Graphics):void
makeRectangle(Graphics,Point):Rectangle
makeCenterPoint(Graphics,Point):Point
-solution
0..1
-branches
0..*
~branch
0..1
-branch
0..1
-branch
0..1
<<Java Class>>
Maze
(default package)
width: int
height: int
hasNorthWall: boolean[][]
hasWestWall: boolean[][]
start: Point
end: Point
Maze(int,int,int,int,int,int)
clone():Object
Maze(Maze)
getWidth():int
getHeight():int
getStart():Point
getEnd():Point
canMoveNorthFrom(Point):boolean
canMoveSouthFrom(Point):boolean
canMoveEastFrom(Point):boolean
canMoveWestFrom(Point):boolean
canMoveFrom(Point,int):boolean
mod(int,int):int
getPointNorthOf(Point):Point
getPointSouthOf(Point):Point
getPointEastOf(Point):Point
getPointWestOf(Point):Point
getPointOf(Point,int):Point
buildNorth(Point):void
buildWest(Point):void
breakNorth(Point):void
breakWest(Point):void
validPosition(Point):boolean
-maze
0..1
<<Java Class>>
FungusSimulation
(default package)
FungusSimulation()
solveMaze(Maze,Listener):Fungus
-mazeDisplay
0..1
-solver
0..1
-maze
0..1
<<Java Class>>
MazeSolutionThread
(default package)
MazeSolutionThread(FungusSimulation,Maze)
run():void
<<Java Class>>
MazeFrame
(default package)
serialVersionUID: long
depthFirstButton: JRadioButton
kruskalButton: JRadioButton
widthLabel: JLabel
widthField: JFormattedTextField
heightLabel: JLabel
heightField: JFormattedTextField
perfectButton: JRadioButton
empty: JLabel
startLabel: JLabel
startComma: JLabel
startXField: JFormattedTextField
startYField: JFormattedTextField
endLabel: JLabel
endComma: JLabel
endXField: JFormattedTextField
endYField: JFormattedTextField
generateButton: JButton
solveButton: JButton
d: int
width: int
height: int
startX: int
startY: int
endX: int
endY: int
MazeFrame()
buildUI():void
calculateNodes():int
enableButtons():void
disableButtons():void
enableGeneratorButtons(boolean):void
enableSolverButtons(boolean):void
generateMaze():void
solveMaze():void
actionPerformed(ActionEvent):void
centerFrame():void
main(String[]):void
<<Java Interface>>
MazeGenerator
(default package)
generateMaze(int,int,int,int,int,int,boolean):Maze
<<Java Class>>
MazeGeneratorDepthFirst
(default package)
cells: Point[][]
visited: boolean[][]
DIRECTIONS: int
MazeGeneratorDepthFirst()
generateMaze(int,int,int,int,int,int,boolean):Maze
getDirections():Queue<Point>
inBounds(int,int,int,int):boolean
connect(Point,Point,Maze):void
breakWalls(Maze,int,int):Maze
<<Java Class>>
MazeGeneratorKruskal
(default package)
cells: Point[][]
count: int[][]
MazeGeneratorKruskal()
generateMaze(int,int,int,int,int,int,boolean):Maze
FIND(Point):Point
UNION(Point,Point):void
breakWalls(Maze,int,int):Maze

APPENDIX B

Table 2 Examined vertices in a DFS generated maze

		Examined vertices		
Maze size		**Bioinspired alg.**	**DFS**	**A***
11x11	1	105	94	18
	2	41	76	17
	3	98	48	17
	4	47	112	19
	5	51	81	18
	average	230	298.2	17.8
20x20	1	187	287	31
	2	368	301	29
	3	244	259	28
	4	171	296	30
	5	180	348	31
	average			29.8
30x30	1	867	724	55
	2	386	763	54
	3	210	579	53
	4	447	502	56
	5	376	418	53
	average	457.2	597.2	54.2

Table 3 Examined vertices in Kruskal generated maze

		Examined vertices		
Maze size		**Bioinspired alg.**	**DFS**	**A***
11x11	1	62	92	21
	2	87	58	23
	3	36	104	22
	4	98	86	25
	5	57	77	19
	average	68	83.4	22
20x20	1	174	318	35
	2	373	412	40
	3	301	381	34
	4	364	306	28
	5	245	284	29
	average	291.4	340.2	33.2
30x30	1	427	489	54
	2	749	710	58
	3	512	529	59
	4	498	673	58
	5	130	721	57
	average	463.2	624.4	57.2

Table 4 % of completeness of the algorithm (start node -> end node)

		Start -> End				
Maze size	**Try**	**top right -> bottom left**	**top right -> top left**	**inner right -> inner left (straight line)**	**inner right -> inner left (diagonal**	**Passing**
10x10	1	100%	100%	100%	100%	100%
	2	100%	100%	100%	100%	100%
	3	100%	100%	100%	100%	100%
	4	100%	100%	100%	100%	100%
	5	100%	100%	100%	100%	100%
20x20	1	100%	100%	100%	100%	100%
	2	100%	100%	100%	100%	100%
	3	100%	100%	100%	100%	100%
	4	100%	100%	100%	100%	100%
	5	100%	100%	100%	100%	100%
30x30	1	100%	100%	100%	100%	100%
	2	100%	100%	100%	100%	100%
	3	100%	100%	100%	100%	100%
	4	100%	100%	100%	100%	100%
	5	100%	100%	100%	100%	100%
40x40	1	100%	100%	100%	100%	100%
	2	100%	100%	100%	100%	100%
	3	100%	100%	100%	100%	100%
	4	100%	100%	100%	100%	100%
	5	100%	100%	100%	100%	100%
50x50	1	100%	100%	100%	100%	100%
	2	100%	100%	100%	100%	100%
	3	100%	100%	100%	100%	100%
	4	100%	100%	100%	100%	100%
	5	100%	100%	100%	100%	100%

Table 5 Ratio of examined vertices vs the shortest path

	Try	Examined vertices	Ratio
11x11 (shortest path = 33)	1	62	0.532258065
	2	87	0.379310345
	3	36	0.916666667
	4	98	0.336734694
	5	57	0.578947368
	average	68	0.485294118
20x20 (shortest path = 154)	1	174	0.885057471
	2	373	0.412868633
	3	301	0.511627907
	4	364	0.423076923
	5	245	0.628571429
	average	291.4	0.528483185
30x30 (shortest path = 106)	1	427	0.24824356
	2	749	0.141522029
	3	512	0.20703125
	4	498	0.212851406
	5	130	0.815384615
	average	463.2	0.228842832

Table 6 Execution times of bioinspired alg. vs DFS

	Run Times (in sec)											
Maze size ->	**20x20**		**30x30**		**40x40**		**50x50**		**60x60**		**70x70**	
Try	**BIA**	**DFS**	**BIA**	**DFS**	**BIA**	**DFS**	**BIA**	**DFS**	**BIA**	**DFS**	**BIA**	**DFS**
1	11.5	6.53	17.51	5.323	42.12	41.53	48.24	72.432	84.85	132.85	87.29	110.83
2	6.51	2.324	12.09	17.53	45.43	37.32	63.54	65.43	14.07	92.83	105.53	72.64
3	7.49	5.632	24.31	18.234	10.21	35.62	62.43	82.432	33.62	38.52	104.16	68.37
4	4.83	6.129	14.58	12.523	43.23	18.32	34.32	54.342	30.83	61.48	61.67	93.58
5	6.88	4.89	12.37	17.231	17.59	26.43	48.49	79.934	86.36	131.42	36.13	184.18
avg	7.442	5.101	16.172	14.1682	31.716	31.844	51.404	70.914	49.946	91.42	78.956	105.92

Table 7 Execution times of different pathfinding algorithms

Maze size (width by height)	**Run Time (in seconds)**						
	Bio-inspired algorithm	**Trace**	**Best-First-Search**	**A***	**Jump Point Search**	**Dijkstra**	**DFS**
10 x 10	1.3	1.16	1.18	1.7	1.1	1.36	1.2
15 x 15	3.87	1.73	1.739	2.42	1.573	3.97	4.21
20x20	7.4	2.172	2.22	6.51	4.404	8.96	5.101
30x35	17.53	4.5	4.8	12.55	14.41	18.05	18.92
50x35	43.54	11.74	10.3	17.494	25.26	30.63	65.53
70x35	61.84	12.16	14.65	26.43	27.39	39.78	84.3

Note: Manhattan heuristic for A*, IDA*, Best-First-Search, Jump Point Search, Trace