FEDERAL UNIVERSITY OF THE RIO DE JANEIRO STATE
CENTER OF EXACT SCIENCES AND TECHNOLOGY
SCHOOL OF APPLIED INFORMATICS

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
ESCOLA DE INFORMÁTICA APLICADA

Performance of the Julia Programming Language in Different Search Methods

Alice da Fonseca Monteiro

Supervisor
Pedro Nuno de Souza Moura

RIO DE JANEIRO, RJ – BRAZIL
JULY 2018

Catalogação informatizada pelo(a) autor(a)

M772

Monteiro, Alice da Fonseca
Performance of the Julia Programming Language in Different Search Methods / Alice da Fonseca Monteiro. -- Rio de Janeiro, 2018.
34

Orientador: Pedro Nuno de Souza Moura.
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal do Estado do Rio de Janeiro, Graduação em Sistemas de Informação, 2018.

1. Julia. 2. labyrinth. 3. search. I. Moura, Pedro Nuno de Souza, orient. II. Título.

Performance of the Julia Programming Language in Different Search Methods

Alice da Fonseca Monteiro

Undergraduate thesis presented to the Applied Informatics School of the Federal University of the Rio de Janeiro State to obtain the title of Bachelor in Information Systems.

Approved by:

____

Pedro Nuno de Souza Moura (Unirio)

____

Leonardo Luiz Alencastro Rocha (Unirio)

RIO DE JANEIRO, RJ – BRAZIL.

JULY 2018

Acknowledgements

I would like to thank those who helped me make this possible; who showed me a path; who gave me the initial push I needed; and who kept supporting me through the whole path.

RESUMO

Busca é uma das principais técnicas de solução de problemas em inteligência artificial. Este estudo aborda o problema de solução de labirintos utilizando busca e testa o desempenho de algoritmos de busca implementados em Julia, linguagem de alto nível desenvolvida para computação numérica. Foram feitos experimentos de tempo comparando o desempenho de Julia e Python. Os resultados sugerem que Julia tem melhor desempenho em instâncias maiores, o que corrobora a literatura sobre o tema.

Palavras-chave: Julia, busca, labirinto.

ABSTRACT

Search is one of the main problem-solving techniques in artificial intelligence. The present study addresses the maze solving problem using search and tests the performance of search algorithms implemented in Julia. Julia is a high-level language developed for numerical computing. Similar search algorithms were implemented in Python to show the difference between the two languages. The literature shows that Julia performs better with larger instances, which has been suggested by the results.

Keywords: Julia, search, maze.

Contents

1 Introduction 1
1.1 Motivation 1
1.2 Problem Statement 1
1.3 Objectives 2
1.4 Text Structure 2
2 Main Concepts 3
2.1 Julia 3
2.2 Search 4
2.3 Maze Solving 7
3 Methodology 8
3.1 Mazes 8
3.2 Search Methods 9
3.3 Time Measurements 11
4 Results 13
4.1 Time Measurement Results 13
4.2 Time Measurement Functions 18
5 Conclusion 20
5.1 Final Remarks 20
5.2 Study Limitations 21
5.3 Future Research 21
Appendices 22
A. Code for breadth-first search in Julia 22
B. Code for depth-first search in Julia 23
C. Code for heuristic and A* search in Julia 24

List of Tables

Table 1: Results for the 21 x 21 maze. 13
Table 2: Mean results for the 21 x 21 maze. 14
Table 3: Results for the 31 x 31 maze. 15
Table 4: Mean results for the 31 x 31 maze. 15
Table 5: Results for the 41 x 41 maze. 16
Table 6: Mean results for the 41 x 41 maze. 16
Table 7: Results for the 201 x 201 maze. 17
Table 8: Mean results for the 201 x 201 maze. 18

List of Figures

Figure 1. Benchmark times relative to C (smaller is better, C performance = 1.0). Source: julialang.org. 3
Figure 2. Tree-search algorithm. Source: Russel/Norvig. 5
Figure 3. Graph-search algorithm. Source: Russel/Norvig. 6
Figure 4: Mazes. A: Orthogonal maze (rectangular cells). B: Sigma maze (hexagonal cells). C: Delta maze (triangular cells). D: Theta maze (circular maze). Source: mazegenerator.net. 7
Figure 5: 21 x 21 maze. Wall are represented as "X" and paths as ".". The initial state is represented as "I" and the goal state as "F". 8
Figure 6: Breadth-first search algorithm pseudocode. 9
Figure 7: Depth-first search algorithm pseudocode. 10
Figure 8: A* search and heuristic algorithm pseudocode. 11
Figure 9: Mean results of time measurements for the 21 x 21 maze. 14
Figure 10: Mean results of time measurements for the 31 x 31 maze. 15
Figure 11: Mean results of time measurements for the 41 x 41 maze. 17
Figure 12: Mean results of time measurements for the 201 x 201 maze. 18
Figure 13: Sample result from the time measurement function in Julia for the 201 x 201 maze. 18
Figure 14: Sample result from the time measurement function in Python for the 201 x 201 maze. 19

1 Introduction

1.1 Motivation

Performance is a crucial factor for programming languages used for scientific computing, as large amounts of data need to be analyzed. Conversely, there has always been the need to develop languages with a shallow learning curve. Fortran, a very popular programming language among researchers, is the short for Formula Translating System, and was developed as an effort to make it possible for scientists to write programs themselves. Julia claims to bring the best of both worlds: To be a high-level, high-performance dynamic programming language for numerical computing that combines the performance of C with the productivity of high-level languages.

Search is one of the main techniques for problem solving in artificial intelligence, as many artificial intelligence problems can be modeled as state spaces that need to be explored to find a solution. Mazes are a simple abstraction where search techniques can be successfully applied. Therefore, in the present study, mazes are used to evaluate the performance of Julia using different search methods to solve them.

1.2 Problem Statement

Scientific programming requires high performance, yet domain experts tend to use slower dynamic languages, such as Python, due to the steep learning curve of high-performing languages such as C. Julia is a high-level dynamic programming language, with good readability and shallow learning curve. One of its main advertised features is high performance, allowing it to approach and often match the performance of C, which is a reference in terms of speed. This would eliminate the performance for ease of use trade-off that usually occurs when using high-level dynamic languages.

1.3 Objectives

The present study aims to evaluate the performance of Julia and find out whether it would be a suitable language for maze solving. Different search methods are implemented along with time measurements. For comparison purposes, the same methods are implemented in Python.

1.4 Text Structure

The present study is structured as the following chapters beyond the introduction: Chapter 2 describes the main concepts related to the study: the Julia programming language, the maze problem, and search algorithms used for maze solving. Chapter 3 describes the methodology used to generate the results, which are presented and discussed on Chapter 4. Finally, Chapter 5 brings final considerations, limitations of the study and suggestions for future research on the subject.

2 Main Concepts

This chapter describes the main concepts related to the present study. The first section provides an overview of Julia. The second section addresses search and the search methods implemented in the study. The third section describes the maze solving problem.

2.1 Julia

Julia is a high-level dynamic programming language for numerical computing [1]. It has been in development since 2009 and was first released in 2012 as an open source project available under the MIT License for open source software. Benchmarks show that it approaches and often matches the performance of C [2]. Figure 1 shows benchmark times for Julia.

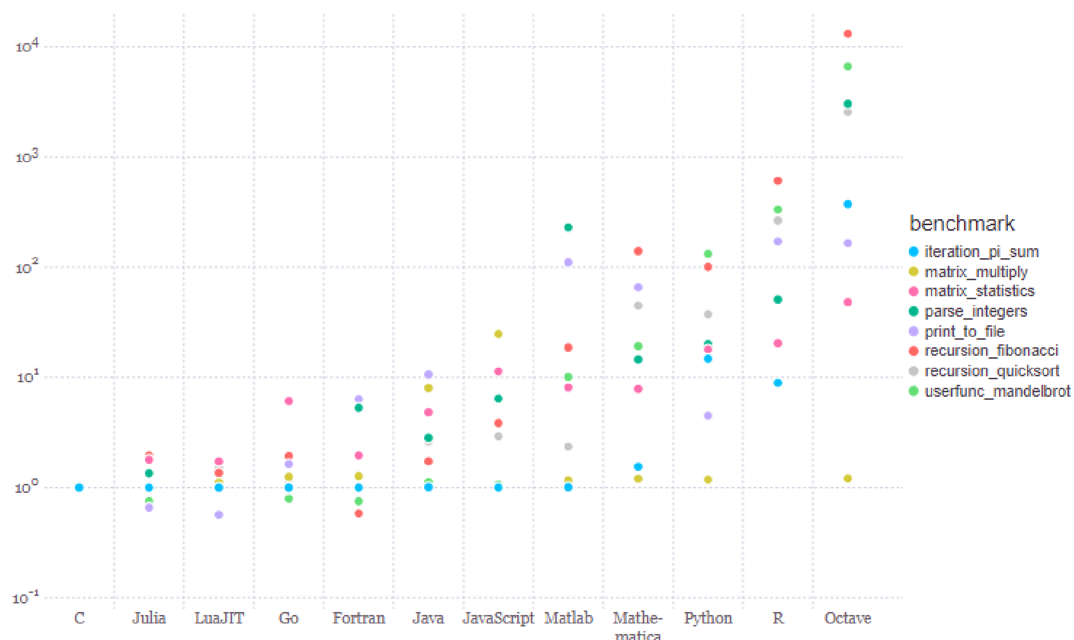

Figure 1. Benchmark times relative to C (smaller is better, C performance = 1.0). Source: julialang.org.

Julia's design combines a number of technologies to deliver a high-performance language, while maintaining ease of use and increased productivity due to the high abstraction level provided as a dynamic high-level language.

Julia works with an out-of-the-box LLVM-based just-in-time (JIT) compiler for just-in-time generation of machine code. Declaring the type of variables in the code is optional, as it has a dynamic type system. Types are inferred based on data contained in the variables using an algorithm based on forward dataflow analysis [3].

Julia's primary means of abstraction is dynamic multiple dispatch [4]. The majority of Julia functions are generic functions, which select native code implementations across multiple combinations of argument types.

Numerical computing requires performance-critical numerical libraries, which depend on the details of the internal implementation of the high-level language. A main indicator of the validity of Julia's design is that its standard library, which encompasses most of the core functionality of standard technical computing environments, is implemented in Julia itself.

Julia is still a fairly young language, especially when the current version number is considered (0.6.3), but its foundation is stable, with no backwards-incompatible changes since the first release [2].

2.2 Search

Search is one of the main problem-solving approaches in artificial intelligence.

Search is the process of an agent constructing sequences of actions that achieve goals. Before solving a problem by using search, a goal must be identified and a well-defined problem must be formulated.

A well-defined problem has an initial state, a description of all the possible actions available to the agent, a transition model, which defines what each action does, the goal test, which determines whether a given state is a goal state, and a path cost function that assigns a numeric cost to each path. In addition, some level of abstraction must be applied to remove as much detail as possible while preserving the validity of abstract actions and ensuring they are easy to carry out in the real world.

A solution to a problem is an action sequence leading from the initial state to a goal state, i.e., a path from the initial state to a goal state, whose quality is measured by the path cost function. An optimal solution has the lowest path cost among all solutions.

Search algorithms work by considering different possible solutions, i.e., action sequences. First, the initial state is tested to verify whether it is a goal state. Then, the current state is expanded by generating a new set of states depending on the available actions. The states then can be expanded are called the frontier of the problem. The next step is to choose the next state and continue the search process. All search algorithms share this basic structure, named tree-search. They vary according to how they choose which state to expand next, i.e., the search strategy. Figure 2 shows a tree-search algorithm.

function TREE-SEARCH(problem) **returns** a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

Figure 2. Tree-search algorithm. Source: Russel/Norvig.

Some problems can be defined to eliminate redundant paths by limiting the available actions. When redundant paths are unavoidable, a data structure called the explored set is added to keep track of visited nodes. This new algorithm is called the graph-search algorithm. Therefore, tree-search algorithms consider all possible paths to find a solution, while graph-search algorithms avoid considering redundant paths. Figure 3 shows a graph-search algorithm.

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
        loop do
            if the frontier is empty then return failure
            choose a leaf node and remove it from the frontier
            if the node contains a goal state then return the
            corresponding solution
            add the node to the explored set
            expand the chosen node, adding the resulting nodes to
            the frontier
                only if not in the frontier or explored set
```

Figure 3. Graph-search algorithm. Source: Russel/Norvig.

There are informed and uninformed search strategies. In uninformed search strategies, also called blind search, methods have access only to the problem definition, with no additional information about states. All they can do is distinguish a goal state from a non-goal state, and they differ based on the order in which nodes are expanded. Uninformed search examples are breadth-first search and depth-first search.

Breadth-first search is a simple strategy in which the root node is visited and expanded, then all of its successors are expanded next, then their successors, and so on. This is achieved by using a FIFO queue for the set of all available nodes for expansion (i.e., the frontier). This way, new nodes go to the back of the queue and old nodes get expanded first.

Depth-first search always expands the deepest node in the frontier, proceeding to the deepest level of the search tree, where nodes have no successors. A LIFO queue is used so the most recently generated node is chosen for expansion.

In addition to the problem definition, informed search strategies use problem-specific knowledge to select nodes that are going to be expanded. The general approach is called best-first search, which uses an evaluation function to provide a cost estimate, and the node with the lowest evaluation is expanded first. Most best-first algorithms use a heuristic function, which provides the estimated cost of the cheapest path from the state to a goal state.

One thing to note is that the heuristic function must be admissible and consistent. An admissible heuristic never overestimates the cost to reach the goal. A heuristic is consistent if the estimated cost of getting to the goal from a node n is no greater than the step cost of getting from n to each of its successors n' plus the estimated cost of getting from n' to the goal.

A* (A-star) search is an informed search method that uses a heuristic function to estimate the cost to reach the goal. It combines the cost to reach the node and the cost to get from the node to the goal to estimate the cost of the cheapest solution.

2.3 Maze Solving

Mazes can be modeled as state spaces that can be solved using search. Paths are divided into units with the same size that represent the states that can be accessed. Some states cannot be accessed, i.e., walls. Search strategies find the path from the initial state to a goal state by performing the available actions according to the maze type and the algorithm chosen. Figure 4 shows a few examples of different types of mazes.

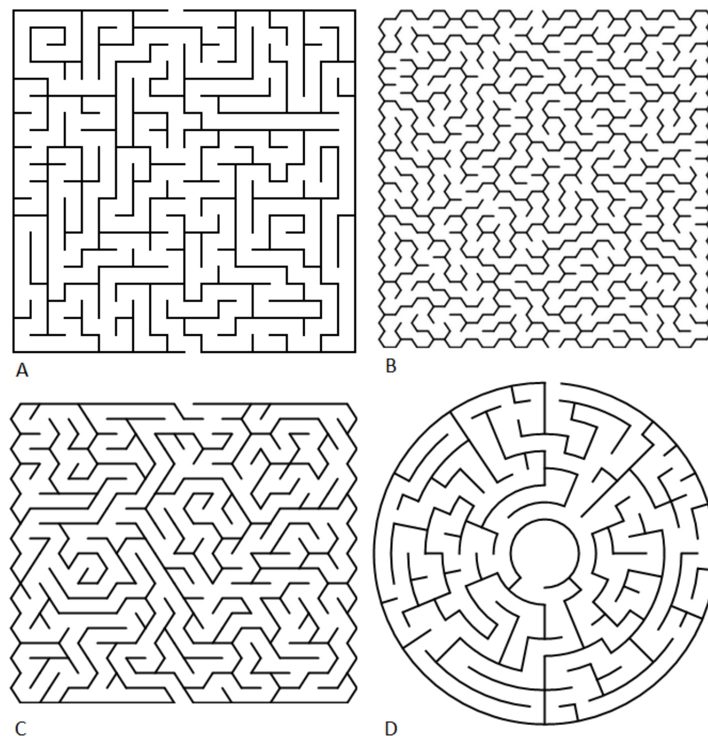

Figure 4: Mazes. A: Orthogonal maze (rectangular cells). B: Sigma maze (hexagonal cells). C: Delta maze (triangular cells). D: Theta maze (circular maze). Source: mazegenerator.net.

3 Methodology

This chapter describes the methodology used to develop the study. The first section presents the mazes used to run the search algorithms. The second section describes how the search methods were implemented. The third section describes how time measurements were made.

3.1 Mazes

The mazes explored were contained in a set of four text files with different sized mazes (21 x 21, 31 x 31, 41 x 41, and 201 x 201 cells). Mazes were orthogonal and had only one possible solution. Figure 5 shows one of the mazes used.

```
X X X X X X X X X X X X X X X X X X X X X
X . . . . . . . . . . . . . . . . . X F X
X . X X X X X X X X X X X . X X X X X . X
X . X . . . . . . . . . X . . . . . X . X
X . X X X X X X X . X . X X X X X . X . X
X . X . . . . . X . X . . . . . X . . . X
X . X . X X X . X X X X X . X . X X X X X
X . . . X . X . . . . . X . X . . . . . X
X X X X X . X X X X X . X . X . X X X . X
X . . . . . X . . . X . X . X . X . X . X
X X X . X . X . X . X . X X X . X . X . X
X . . . X . X . X . X . . . . . X . X . X
X . X X X . X . X . X X X X X X X . X . X
X . X . . . X . X . . . X . . . . . . . X
X . X X X X X . X X X . X . X X X X X X X
X . . . . . X . . . X . . . X . . . X . X
X . X X X . X X X . X X X X X . X . X . X
X . X . X . . . X . . . X . . . X . X . X
X . X . X X X . X X X . X . X X X . X . X
X I . . . . X . . . . . . . X . . . . . X
X X X X X X X X X X X X X X X X X X X X X
```

Figure 5: 21 x 21 maze. Wall are represented as "X" and paths as ".". The initial state is represented as "I" and the goal state as "F".

Each maze was stored in a two-dimensional array with 1 representing walls and 0 representing paths, and the start and the goal nodes were stored as tuples of coordinates. A dictionary of adjacencies was built from the array, having nodes (tuples of coordinates) as keys and arrays with neighboring nodes along with their direction (char N for North, S for South, E for East, and W for West) as values. The dictionary of adjacencies and the start and goal nodes were passed as arguments to all functions.

3.2 Search Methods

Standard data structures were used to implement the search methods, and iterative implementations of breadth-first search, depth-first search, and A* search were used. Graph-search, which avoids repeated states and redundant paths, was used in all implementations. Figures 4 and 5 show the pseudocode for the breadth-first search and depth-first search algorithms.

function BREADTH-FIRST-SEARCH(dictionary, start, goal) **returns** path and visited nodes, or failure

    SET *queue* to empty list
    SET *visited* to empty set
    SET *path* to empty string
    PUSH (*start*, *path*) to *queue*
    WHILE *queue* is not empty
        REMOVE *firstItem* from *queue*
        SET (*current*, *path*) to *firstIitem*
        IF *current* is equal to *goal*
            ADD *current* to *visited*
            RETURN *path*, *visited*
        IF *current* is not in *visited*
            ADD *current* to *visited*
            FOR (*direction*, *neighbor*) in *dictionary* key *current*
                PUSH (*neighbor*, *path*direction*) to *queue*
    RETURN failure

Figure 6: Breadth-first search algorithm pseudocode.

function DEPTH-FIRST-SEARCH(dictionary, start, goal) **returns** path and visited nodes, or failure

> **SET** *stack* to empty list
> **SET** *visited* to empty set
> **SET** *path* to empty string
> **PUSH** (*start*, *path*) to *stack*
> **WHILE** *stack* is not empty
>   **REMOVE** *lastItem* from *stack*
>   **SET** (*current*, *path*) to *lastIitem*
>   **IF** *current* is equal to *goal*
>   **ADD** *current* to *visited*
>   **RETURN** *path*, *visited*
>   **IF** *current* is not in *visited*
>   **ADD** *current* to *visited*
>   **FOR** (*direction*, *neighbor*) in *dictionary* key *current*
>   **PUSH** (*neighbor*, *path**direction*) to s*tack*
> **RETURN** failure

Figure 7: Depth-first search algorithm pseudocode.

For A* search, the Manhattan distance was chosen as heuristic function. For points p1 at (x1, x2) and p2 at (y1,y2), the Manhattan distance is calculated as below:

$$|x1 - x2| + |y1 - y2|$$

It is an admissible heuristic for the orthogonal mazes used in the present study, as it never overestimates the cost to reach the goal, which cannot be less than the distance between the goal and the start measured along axes at right angles.

The absolute values were obtained using the `abs` function. Figure 6 shows the pseudocode for the A* algorithm.

```
function A-STAR-SEARCH (dictionary, start, goal) returns path and visited nodes, or failure
    SET priorityQueue to empty list
    SET visited to empty set
    SET path to empty string
    SET order to 0 + heuristic(start, goal)
    SET cost to 0
    SET current to start
    PUSH (order, cost, path, current) to priorityQueue
    WHILE priorityQueue is not empty
        REMOVE lastItem from priorityQueue
        SET (order, cost, path, current) to lastItem
        IF current is equal to goal
            ADD current to visited
            RETURN path, visited
        IF current is not in visited
            ADD current to visited
            FOR (direction, neighbor) in dictionary key current
                PUSH (cost + heuristic(neighbor, goal), cost + 1,
            path*direction, neighbor) to priorityQueue
    RETURN failure

function heuristic (cell, goal) returns minimum absolute distance between cell and goal
    RETURN minimum absolute distance(cell[1] - goal[1]) + minimum absolute
distance(cell[2] - goal[2])
```

Figure 8: A* search and heuristic algorithm pseudocode.

3.3 Time Measurements

All time measurements were made using an Intel Core i3-4340 CPU @ 3.60 GHz with 8 GB RAM running Windows 10.

Julia 0.6.2.2 was used and scripts were executed from the Julia REPL. Times were measured using the **`@time`** macro, which is recommended by Julia developers for time measurement purposes.

Python was used to provide a time reference. Python 3.6.5 was used and scripts were executed from the command line. Times were measured using the **`timeit()`** function.

The next chapter presents the results obtained from the time measurements.

4 Results

This chapter presents the results obtained from the time measurements of each search method implemented.

4.1 Time Measurement Results

Tables 1, 3, 5, and 7 show a sample of the time measurement results obtained for breadth-first search (BFS), depth-first search (DFS), and A* search (A*) for the 21 x 21, 31 x 31, 41 x 41, and 201 x 201 mazes. Tables 2, 4, 6, and 8 show the mean results of the time measurements performed on each maze and Figures 8–11 illustrate these results. Python results are included as a reference, and all results are expressed in seconds.

The 21 x 21 maze was fairly small and, as expected, the search methods were able to solve it very fast. A* implemented in Julia was the slowest method, taking up to 0.001 seconds (Table 1), but still very fast.

Table 1: Results for the 21 x 21 maze.

21 x 21					
Julia			**Python**		
BFS	**DFS**	**A***	**BFS**	**DFS**	**A***
0.0002	0.0001	0.0009	0.0002	0.0002	0.0005
0.0002	0.0001	**0.0010**	0.0002	0.0002	0.0005
0.0003	0.0001	0.0009	0.0002	0.0002	0.0006
0.0002	0.0001	0.0009	0.0003	0.0002	0.0005
0.0003	0.0001	**0.0010**	0.0003	0.0002	0.0005
0.0004	0.0009	**0.0010**	0.0003	0.0002	0.0005
0.0001	0.0003	**0.0010**	0.0004	0.0003	0.0008
0.0002	0.0002	**0.0010**	0.0004	0.0002	0.0005
0.0002	0.0001	**0.0010**	0.0004	0.0003	0.0008
0.0003	0.0001	0.0009	0.0002	0.0003	0.0005

Table 2 and Figure 9 show that the mean time for each search method in both languages was quite similar.

Table 2: Mean results for the 21 x 21 maze.

21 x 21 - Means					
Python			**Julia**		
BFS	**DFS**	**A***	**BFS**	**DFS**	**A***
0.0003	0.0002	0.0006	0.0002	0.0002	0.0010

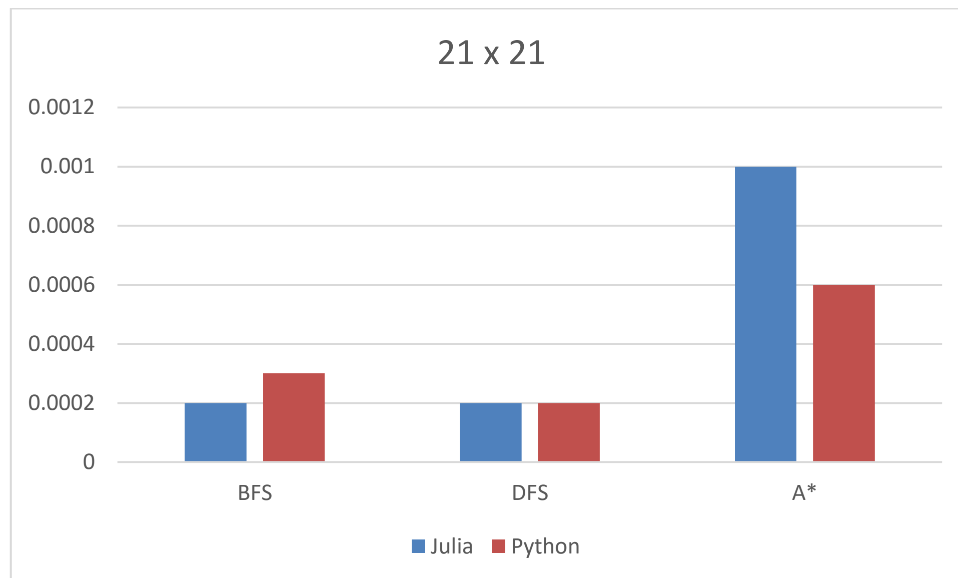

Figure 9: Mean results of time measurements for the 21 x 21 maze.

The 31 x 31 maze was slightly bigger, but time measurements were similar to those obtained in the 21 x 21 maze. Table 3 shows that A* was considerably slower than BFS and DFS, taking up to 0.0025 seconds in Julia. However, some of the results for A* in Julia were achieved in half of this time, in a minimum of 0.0012 seconds.

Table 3: Results for the 31 x 31 maze.

31 x 31					
Julia			**Python**		
BFS	**DFS**	**A***	**BFS**	**DFS**	**A***
0.0002	0.0001	**0.0011**	0.0006	0.0002	0.0006
0.0003	0.0001	**0.0024**	0.0006	0.0002	0.0008
0.0002	0.0001	**0.0012**	0.0004	0.0001	0.0006
0.0003	0.0002	**0.0012**	0.0004	0.0001	0.0010
0.0002	0.0001	**0.0023**	0.0005	0.0002	0.0010
0.0003	0.0001	**0.0013**	0.0004	0.0002	0.0007
0.0002	0.0002	**0.0024**	0.0003	0.0001	0.0006
0.0002	0.0001	**0.0012**	0.0006	0.0002	0.0010
0.0004	0.0001	**0.0013**	0.0005	0.0002	0.0010
0.0002	0.0002	**0.0025**	0.0003	0.0001	0.0006

These intermittent results greatly affected the mean time for A* in Julia, as observed in Table 4 and in Figure 10.

Table 4: Mean results for the 31 x 31 maze.

31 x 31 - Means					
Python			**Julia**		
BFS	**DFS**	**A***	**BFS**	**DFS**	**A***
0.0005	0.0002	0.0008	0.0003	0.0001	0.0017

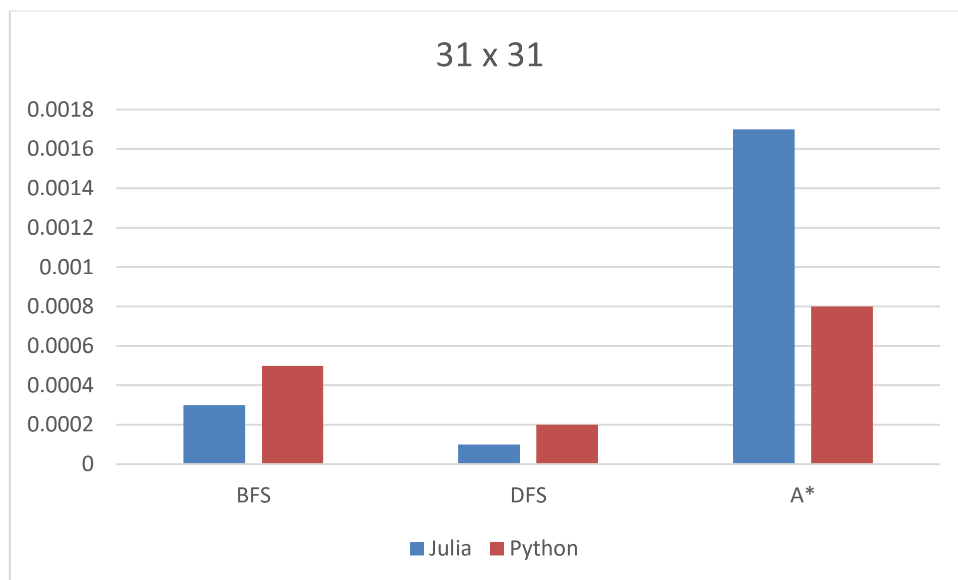

Figure 10: Mean results of time measurements for the 31 x 31 maze.

DFS implemented in Julia proved to be the most efficient method-language combination in the 41 x 41 maze, running in 0.0003–0.0006 seconds, almost always less than the minimum it took for other methods to run (Table 5).

Table 5: Results for the 41 x 41 maze.

41 x 41					
Julia			**Python**		
BFS	**DFS**	**A***	**BFS**	**DFS**	**A***
0.0006	0.0004	**0.0128**	0.0008	0.0007	0.0023
0.0005	0.0002	0.0037	0.0014	0.0013	0.0032
0.0006	0.0003	0.0035	0.0014	0.0013	0.0034
0.0007	0.0003	0.0088	0.0008	0.0008	0.0028
0.0007	0.0003	0.0036	0.0012	0.0007	0.0018
0.0007	0.0002	0.0034	0.0009	0.0008	0.0019
0.0009	0.0003	0.0036	0.0009	0.0008	0.0025
0.0011	0.0003	0.0034	0.0013	0.0012	0.0021
0.0007	0.0006	0.0034	0.0014	0.0013	0.0019
0.0291	0.0005	0.0035	0.0008	0.0008	0.0019

In Julia, an extreme result of 0.0291 seconds was obtained for BFS. Similarly, an extreme result of 0.0128 seconds was obtained for A*. Again, these results affected the mean time for BFS and A* in Julia (Table 6 and Figure 11), which otherwise would have been similar to those obtained with Python.

Table 6: Mean results for the 41 x 41 maze.

41 x 41 - Means					
Python			**Julia**		
BFS	**DFS**	**A***	**BFS**	**DFS**	**A***
0.0011	0.0010	0.0024	0.0036	0.0003	0.0049

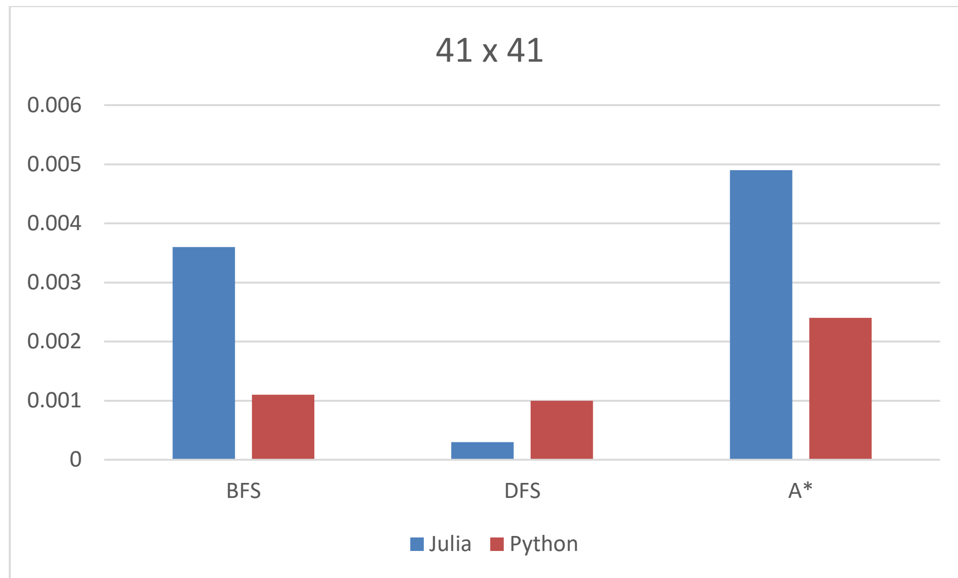

Figure 11: Mean results of time measurements for the 41 x 41 maze.

In the 201 x 201 maze, the results for BFS and DFS implemented in Julia exhibited inconsistencies, with the maximum time sometimes taking almost three times longer than the minimum time (Table 7).

Table 7: Results for the 201 x 201 maze.

201 x 201					
Julia			**Python**		
BFS	**DFS**	**A***	**BFS**	**DFS**	**A***
0.0678	0.0723	0.1572	0.0442	0.0398	**0.0983**
0.0661	0.0675	0.1693	0.0490	0.0459	0.0836
0.1843	0.0679	0.1623	0.0458	0.0461	0.0809
0.0687	**0.0620**	0.1620	0.0462	0.0485	0.0845
0.0676	0.0690	**0.1781**	0.0447	0.0473	0.0861
0.0692	0.0636	0.1712	0.0534	0.0456	0.0897
0.0638	**0.1894**	0.1637	0.0464	0.0446	0.0867
0.0685	0.1706	0.1745	0.0438	0.0520	0.0825
0.0662	0.0819	0.1900	0.0430	0.0484	0.0839
0.0685	0.1706	0.1745	0.0466	0.0475	0.0800

A* search implemented in Julia was markedly slower than that implemented in Python, reaching almost double of the maximum time spent by A* implemented in Python (Table 8 and Figure 12).

Table 8: Mean results for the 201 x 201 maze.

201 x 201 - Means					
Python			Julia		
BFS	DFS	A*	BFS	DFS	A*
0.0463	0.0466	0.0856	0.0791	0.1015	0.1703

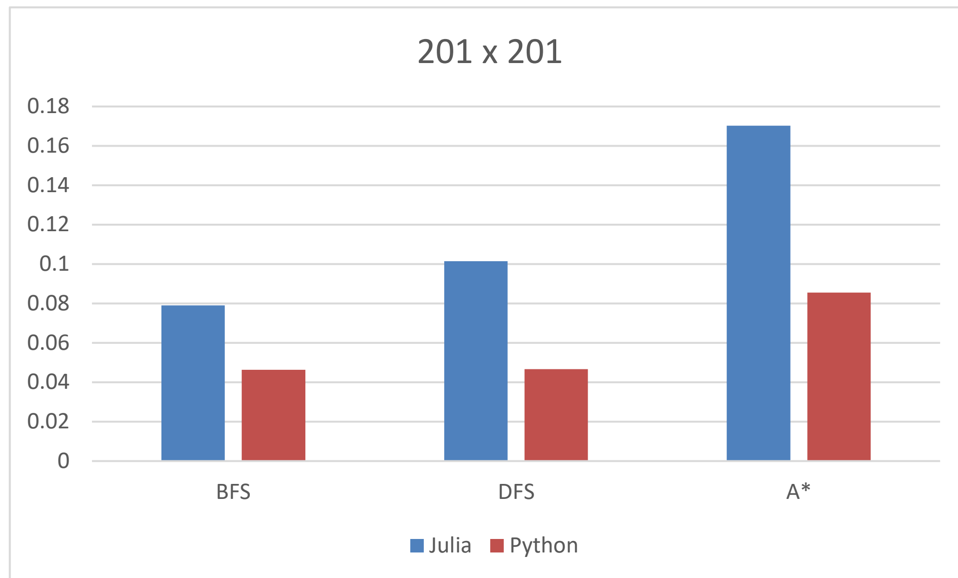

Figure 12: Mean results of time measurements for the 201 x 201 maze.

4.2 Time Measurement Functions

The **`@time`** macro in Julia measures not only the time it takes for a function to run, but also returns the memory allocated and the garbage collection time. Figure 13 shows a sample output from the **`@time`** macro, taken from the 201 x 201 maze.

```
BFS
  0.073456 seconds (218.02 k allocations: 147.081 MiB, 30.15% gc time)
DFS
  0.171273 seconds (200.42 k allocations: 133.879 MiB, 64.01% gc time)
A Star
  0.293412 seconds (1.45 M allocations: 170.634 MiB, 50.16% gc time)
```

Figure 13: Sample result from the time measurement function in Julia for the 201 x 201 maze.

We can observe that garbage collection accounts for a large share of the time it takes for functions to run in Julia.

In Python, the **`timeit()`** function was used, which by default, automatically disables garbage collection at the time of the measurements. Figure 14 shows a sample output from the **`timeit()`** function, taken from the 201 x 201 maze.

```
BFS
0.0475431
DFS
0.0476411
A Star
0.0959256
```

Figure 14: Sample result from the time measurement function in Python for the 201 x 201 maze.

5 Conclusion

This chapter presents the conclusions of the present study. The first section contains the final remarks regarding the results obtained. The second section states the limitations of the study and the third section suggests topics for future research on the subject.

5.1 Final Remarks

The results of the present study show that both Julia and Python are efficient languages for maze solving. Julia did not show any significant advantages over Python. Most of the times, Python even outperformed Julia. However, a few considerations should be made about this.

First, the code used for the search algorithms was a simple implementation, with no type declarations. This makes it difficult to follow one of the main recommendations made by Julia's developers, which is to maintain type-stability. Although types can be inferred based on the data contained, this makes it more difficult for the compiler to optimize code and may impact performance.

In addition, memory preallocation, which is another recommendation to improve performance, was not used. This may have significantly impacted the performance of search methods implemented in Julia, as garbage collection time accounted for a big share of the time it took for each function to run. Sometimes, garbage collection time reached up to 75% of the time search methods took to run in the largest instances of mazes.

Finally, the timing functions used were slightly different, as the **`@time`** macro in Julia considers garbage collection time and the **`timeit()`** function in Python does not. For example, the total time for DFS in the 201 x 201 maze was 0.1713 with 64.01% of

garbage collection time. Without garbage collection, the function would take approximately 0.06164 seconds to run (Figure 13).

Having these considerations in mind, it should also be noted that Julia is a new language, with not much content available for it when compared to popular languages such as Python. For the version used in the present study, not even a debugger was available. Nonetheless, coding in Julia was a very pleasant experience, and developers should definitely experiment with it.

5.2 Study Limitations

In the present study, the recommended methods to improve the performance of Julia were not used, although the Python implementation was also not optimized. In addition, a very small set of mazes was used to run the search algorithms, and all of them were small. This may have impacted the results, as literature suggests that Julia starts to show its better performance with larger instances [9]. In addition, the timing functions were not equally set-up, as Julia's **`@time`** macro considers garbage collection time and **`timeit()`**, in Python, does not.

5.3 Future Research

Future research should consider using larger instances of mazes to obtain more data and provide more precise results. In addition, code in Julia should be optimized to take advantage of the language's type declaration. Methods that consider CPU time instead of elapsed time should also be implemented to obtain more precise results and show the potential better performance of Julia.

Appendices

This section contains the appendices for the present study.

A. Code for breadth-first search in Julia

```
1   function bfs(graph, start, goal)
2       bfsqueue = [(start, "")]
3       visited = Set()
4       while length(bfsqueue) != 0
5           current, path = splice!(bfsqueue,1)
6           if current == goal
7               return path, push!(visited, current)
8           end
9           if current in visited
10              continue
11          end
12          push!(visited, current)
13          for (direction, neighbour) in graph[current]
14              push!(bfsqueue, (neighbour, path*direction))
15          end
16      end
17      return "NO WAY!"
18  end
```

B. Code for depth-first search in Julia

```
1   function dfs(graph, start, goal)
2       dfsstack = [(start, "")]
3       visited = Set()
4       while length(dfsstack) != 0
5           current, path = pop!(dfsstack)
6           if current == goal
7               return path, push!(visited, current)
8           end
9           if current in visited
10              continue
11          end
12          push!(visited, current)
13          for (direction, neighbour) in graph[current]
14              push!(dfsstack, (neighbour, path*direction))
15          end
16      end
17  end
```

C. Code for heuristic and A* search in Julia

```
1   function heuristic(cell, goal)
2       return abs(cell[1] - goal[1]) + abs(cell[2] - goal[2])
3   end
4
5   function astar(graph, start, goal)
6       pr_queue = []
7       heappush!(pr_queue, (0 + heuristic(start, goal), 0, "", start))
8       visited = Set()
9       while length(pr_queue) != 0
10          _, cost, path, current = heappop!(pr_queue)
11          if current == goal
12              return path, push!(visited, current)
13          end
14          if current in visited
15              continue
16          end
17          push!(visited,current)
18          for (direction, neighbour) in graph[current]
19              heappush!(pr_queue, (cost + heuristic(neighbour, goal),
•               cost + 1, path*direction, neighbour))
20          end
21      end
22      return "NO WAY!"
23  end
```

References

1. Julia. www.julialang.org (2018).
2. Balbaert, I. (2015) "Getting Started with Julia Programming". Birmingham, Packt Publishing.
3. Bezanson, J., Karpinski, S., Shah, V.B., Edelman, A. (2012) "Julia: A Fast Dynamic Language for Technical Computing". September. Available on:

   https://arxiv.org/pdf/1209.5145.pdf
4. Bezanson, J., Karpinski, S., Edelman, A., Shah, V.B. (2012) "Julia: A fresh approach to numerical computing". July. Available on:

   https://arxiv.org/pdf/1411.1607.pdf
5. Chandel, A., Sood, M. (2014) "Searching and Optimization Techniques in Artificial Intelligence: A Comparative Study & Complexity Analysis", International Journal of Advanced Research in Computer Engineering & Technology (IJARCET) Volume 3 Issue 3, March. Available on:

   http://ijarcet.org/wp-content/uploads/IJARCET-VOL-3-ISSUE-3-866-871.pdf
6. Kouatchou, J. (2016) "Basic Comparison of Python, Julia, R, Matlab and IDL", NASA Modeling Guru. National Aeronautics And Space Administration, December. Available on:

   https://modelingguru.nasa.gov/docs/DOC-2625
7. Puget, J. F. (2016) "A Speed Comparison Of C, Julia, Python, Numba, and Cython on LU Factorization", IBM Community, January. Available on:

   https://www.ibm.com/developerworks/community/blogs/jfp/entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization?lang=en
8. Russell, S. J., Norvig, P., & Canny, J. (2003). Artificial intelligence: A modern approach.
9. Sayan, S. (2017) "Performance analysis: Julia, Python & C". Available on: https://hackernoon.com/performance-analysis-julia-python-c-dd09f03282a3