

Robot Navigation Implementation

Kyle Gibbs (102107602)

Swinburne University of Technology

COS30019 Introduction to AI

Anika Kanwal

Bao Vo

21st April 2023

Word Count: 2509

Table of Contents

Table of Contents.....	2
Instructions.....	3
Introduction.....	3
Glossary.....	5
Search Algorithms.....	6
Implementation.....	8
Features / Bugs / Missing.....	11
Research.....	11
Conclusion.....	11
Acknowledgements / Resources.....	12
Referencing.....	13

Instructions

In order to use this Robot Navigation program, the first needs a file setup in a supported specification. This order goes;

- The first row is the total rows and columns of the environment, surrounded by square brackets. (E.g. [5, 11])
- The second row is for the initial x and y state of the agent cell, surrounded by rounded brackets. (E.g. (0, 1))
- The third row is for the goal states, which are the locations the agent is attempting to reach, each coordinate surrounded by rounded brackets and those choices separated by a | line. (E.g. (7, 0) | (10, 3))
- All other rows from the fourth row onwards describe walls, with the first two numbers describing the top leftmost corner, followed by its width and then height. (E.g. (2, 0, 2, 2) would describe a wall that starts in the cell (2, 0), and it is 2 cells wide and 2 cells high)

This file will be used in a batch file to allow for more flexible searches. The program will then ask which type of search the user wants to perform. The program will return the results and then close.

Introduction

For the Robot Navigation problem, I have an environment the size of $N \times M$, where both N and $M > 1$. There is a robot that is also referred to as the Agent Cell. In this environment, there are a number of walls that the robot can not stand in. Located in the environment are a number of goals that the robot is attempting to path to. This graph is set up to briefly show what this problem may look like, with a smaller environment of 4 cells by 6 cells. In this example, the cells at the coordinates of (1,1), (1, 2), (3, 4) and (3, 5) are walls.

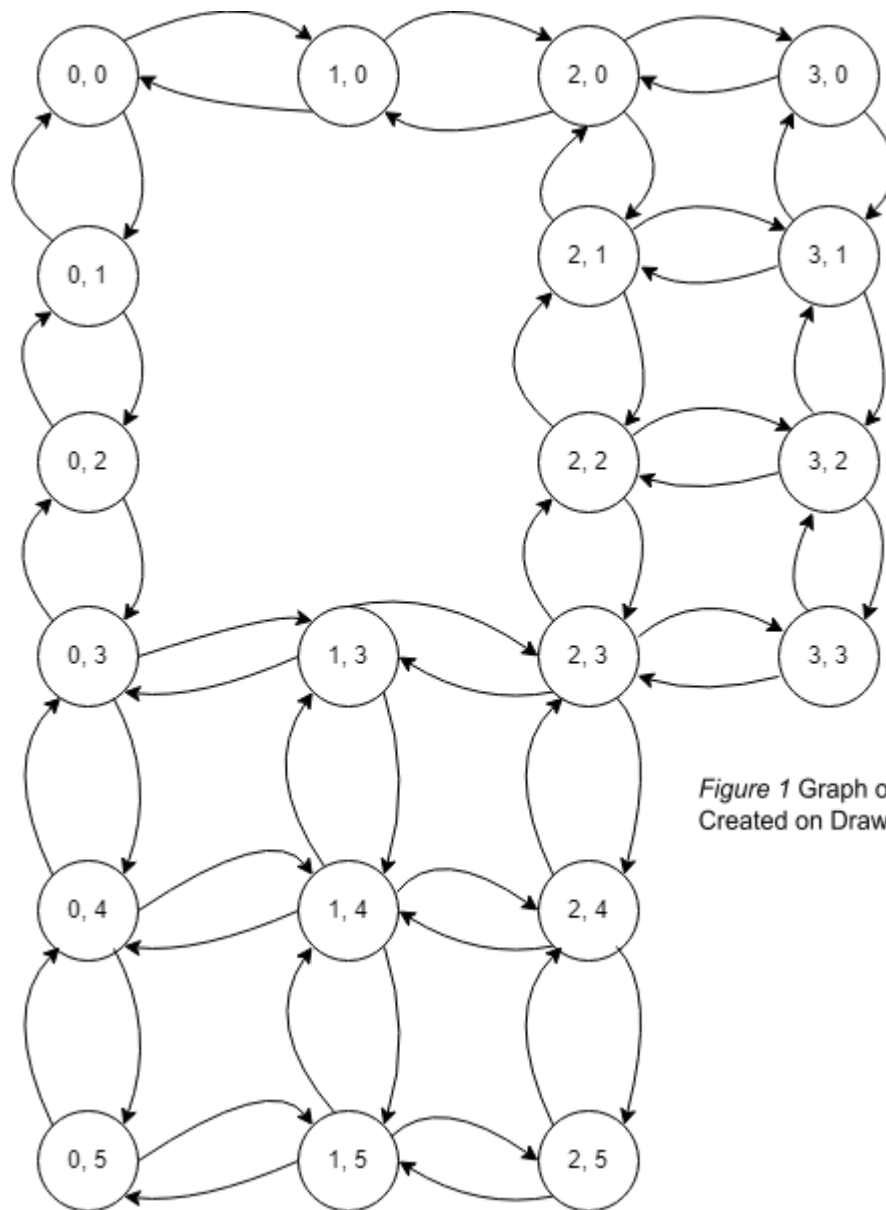


Figure 1 Graph of RobotNavProblem Layout,
Created on Draw.io, by Kyle Gibbs 19/04/2023

This graph shows that it's a continuous movement where the robot can feasibly move to and from any space that isn't a wall. Now using a tree graph, I will show how a Depth First Search algorithm may work in this same environment. For this, I will assume that the robot has a starting coordinate of (0,1) and it's attempting to reach a goal state of (1, 4). I will also assume there are looping checks to stop the search from going infinite. The movement of

the robot will be attempted in the order of going down before right, before left, and before up.

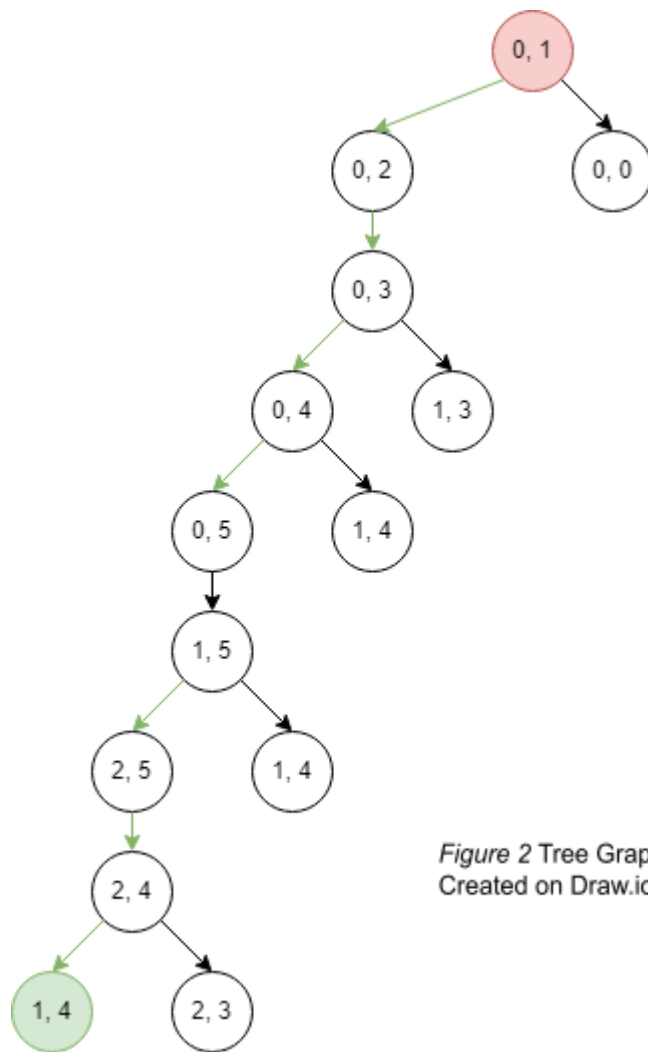


Figure 2 Tree Graph of RobotNavProblem,
Created on Draw.io, by Kyle Gibbs 19/04/2023

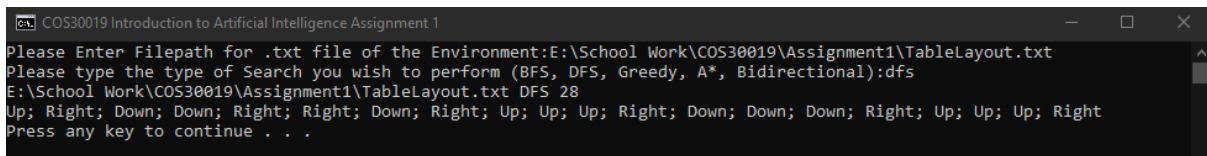
Glossary

- DFS: Depth First Search
- BFS: Breadth-First Search
- Agent (Cell): Same as the Robot (Cell), just another name for it
- GBFS: Greedy Best First Search
- AS: A* Search
- BDS: Bidirectional Search
- LIFO: Last in First Out
- FIFO: First in First Out

Search Algorithms

For this robot navigation problem, I have implemented a total of 6 different search algorithms, each with 2 different forms they can take. For each search algorithm, I have the normal version that tries to walk its way to the goal, and I have a jump implementation, more on that later.

The first implemented algorithm was the DFS search algorithm. With this one, it will always take the primary direction first, until it can no longer follow that path. This primary direction was laid out to us in the accompanying documentation. To recap, the agent will always attempt to go up before left, before going down, and before attempting to go right.

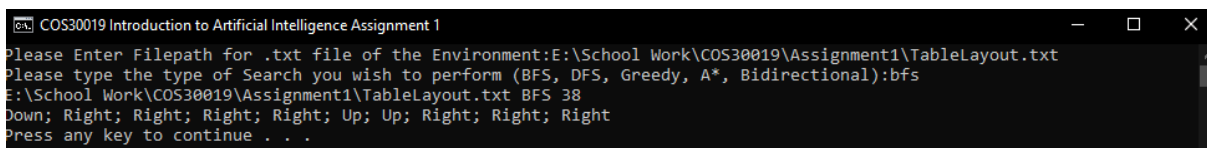


```
COS30019 Introduction to Artificial Intelligence Assignment 1
Please Enter Filepath for .txt file of the Environment:E:\School Work\COS30019\Assignment1\TableLayout.txt
Please type the type of Search you wish to perform (BFS, DFS, Greedy, A*, Bidirectional):dfs
E:\School Work\COS30019\Assignment1\TableLayout.txt DFS 28
Up; Right; Down; Down; Right; Right; Down; Right; Up; Up; Up; Right; Down; Down; Down; Right; Up; Up; Up; Right
Press any key to continue . . .
```

Figure 3 Result from DFS,
Screenshot from running program, by Kyle Gibbs 19/04/2023

Looking at the result that this search returns, I can see that while it doesn't visit too many nodes, it does take many steps to actually get through the environment.

Next, I have the implementation of the BFS search algorithm. This search algorithm opens all available paths at a time and traverses them simultaneously in order to find what it perceives as the best path. It then returns whatever path first reaches a goal.



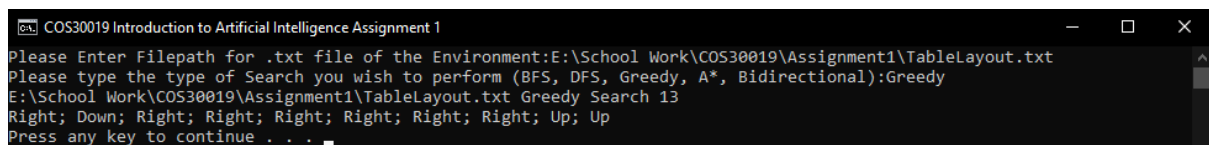
```
COS30019 Introduction to Artificial Intelligence Assignment 1
Please Enter Filepath for .txt file of the Environment:E:\School Work\COS30019\Assignment1\TableLayout.txt
Please type the type of Search you wish to perform (BFS, DFS, Greedy, A*, Bidirectional):bfs
E:\School Work\COS30019\Assignment1\TableLayout.txt BFS 38
Down; Right; Right; Right; Right; Right; Up; Up; Right; Right; Right
Press any key to continue . . .
```

Figure 4 Result from BFS,
Screenshot from running program, by Kyle Gibbs 19/04/2023

The result from this one is more promising than that of the DFS results. While the BFS travels to more nodes overall, the final route it finds is much shorter. While the increase

in nodes travelled may prove to become problematic with larger environments, at this size it is appropriate to use.

The third search algorithm I implemented is the GBFS. This method looks for what path currently will take as close to the goal as possible. It will then attempt to follow this path, while still following the order described during the BFS implementation. GBFS uses a heuristic function in order to find its optimal movement. In this case, that function is $f(n) = h(n)$, where $h(n)$ is the overall cheapest path. This is calculated by adding a Weighted distance to the closest node and adding the total distance to all the nodes. This should ensure that whenever a tie of the closest distance occurs, it will always follow the path that is closest to all the goals. Not only that, in a case where there are many goals away from the agent, but one right next to it, it should always go for the closest one to it first.



```
COS30019 Introduction to Artificial Intelligence Assignment 1
Please Enter Filepath for .txt file of the Environment:E:\School Work\COS30019\Assignment1\TableLayout.txt
Please type the type of Search you wish to perform (BFS, DFS, Greedy, A*, Bidirectional):Greedy
E:\School Work\COS30019\Assignment1\TableLayout.txt Greedy Search 13
Right; Down; Right; Right; Right; Right; Right; Right; Right; Up; Up
Press any key to continue . . .
```

Figure 5 Result from GBFS,
Screenshot from running program, by Kyle Gibbs 19/04/2023

As is evident by both the nodes visited and the final path, GBFS has been the most successful search yet. Due to it calculating the total distance from the goals (not just the distance to one goal), it starts by going right as that gets it closer to two, with the current environment listed in the assignment documentation. This direct path, coupled with the incredibly low node opening of 13, means that this is currently the best search algorithm.

The final search algorithm that was set by the assignment documentation is the A* search algorithm. Like the GBFS, it looks at the path that will get it as close to the goals as possible. Unlike the GBFS, it also takes into consideration the cost to move from one cell to another. While this doesn't have much of an impact in this current stage, it will matter when it comes to implementing a jump function for different movements.

```
COS30019 Introduction to Artificial Intelligence Assignment 1
Please Enter Filepath for .txt file of the Environment:E:\School Work\COS30019\Assignment1\TableLayout.txt
Please type the type of Search you wish to perform (BFS, DFS, Greedy, A*, Bidirectional):A*
E:\School Work\COS30019\Assignment1\TableLayout.txt A* 13
Right; Down; Right; Right; Right; Right; Right; Right; Up; Up
Press any key to continue . . .
```

Figure 6 Result from AS,
Screenshot from running program, by Kyle Gibbs 19/04/2023

Looking at the results, it's the same as the GBFS, which is expected as in this current implementation the cost to move to any cell is one.

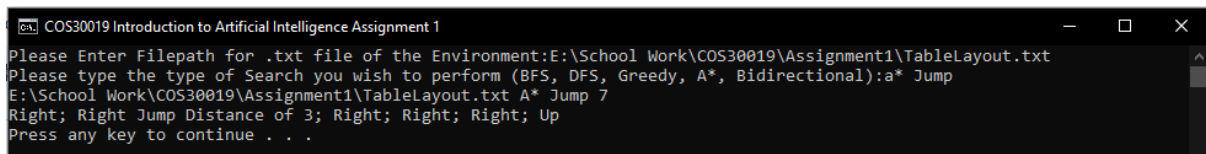
The first of the custom searches implemented is a Bidirectional Search. This traverses the tree from both the agent cell and the goal cells, going into the agent cell and entering a cell that one of the goal cells has also visited. It traverses the tree in the same method as BFS.

```
COS30019 Introduction to Artificial Intelligence Assignment 1
Please Enter Filepath for .txt file of the Environment:E:\School Work\COS30019\Assignment1\TableLayout.txt
Please type the type of Search you wish to perform (BFS, DFS, Greedy, A*, Bidirectional):Bidirectional
E:\School Work\COS30019\Assignment1\TableLayout.txt Bidirectional 48
Down; Right; Right; Right; Right; Down; Right; Right; Right; Up; Up
Press any key to continue . . .
```

Figure 7 Result from Bidirectional Search,
Screenshot from running program, by Kyle Gibbs 19/04/2023

This has a unique path to getting to the goal cell since the way the goal populates the cells means that different cells interact with it first. While not being a bad solution lengthwise, it does end up opening the most nodes out of all the other search algorithms. Because of this, this algorithm may not be applicable to this specific environment.

The next set of search algorithms is the same as the one mentioned above, except they have a function to jump over walls. While this doesn't have an impact on how the program runs, I think it's important to show the types of results this can create. For this example, I will run the AS jump implementation, and see the results



```
COS30019 Introduction to Artificial Intelligence Assignment 1
Please Enter Filepath for .txt file of the Environment:E:\School Work\COS30019\Assignment1\TableLayout.txt
Please type the type of Search you wish to perform (BFS, DFS, Greedy, A*, Bidirectional):a* Jump
E:\School Work\COS30019\Assignment1\TableLayout.txt A* Jump 7
Right; Right Jump Distance of 3; Right; Right; Right; Up
Press any key to continue . . .
```

Figure 8 Result from AS Jump,
Screenshot from running program, by Kyle Gibbs 19/04/2023

As you can see, with the jump function active, it drastically decreases the moves used, as well as the number of nodes traversed.

Implementation

All the search implementations that I performed took in a TableMap class. This class had a layout consisting of a list of lists, that contained Cells. It also contained a definition for the goal cells, saved as another list of cells, and the agent cell.

For the DFS implementation, it keeps the Cells to visit in a stack, which is a LIFO system. This means that the last object, or in this case the most up-to-date cell, is the first object pulled when I perform a pop action. This keeps the cells in order from last to be visited to first to be visited. This, paired with the fact that I place the orders into the stack in the reverse order mentioned early (e.g. right is placed before down, which is placed before left etc...) means that it will follow and travel down one path before trying to explore other routes. All of the implementations perform checks to see if the cell it's trying to visit has been visited before. It does this by keeping the visited cells in a HashSet. I store it in this since I don't care about the order of the cells, just that there aren't duplicates. The pseudocode for placing the cells in our stack goes as follows:

If the selectedMoves List is not null

Then foreach Cell stored in selectedMoves

If the cell is not contained within the visitedCells HashSet

Then make that cells parent the current agentCell

Then push that cell onto the stack

```
        End If
    End Foreach
End If
```

Moving onto the BFS implementation, it works very similarly to the DFS solution. The biggest difference is I use a Queue since I want the BFS to be in a FIFO system, allowing us to check each leaf of the tree at once. This also means I have to check for each move in the correct order. The pseudocode for checking which move to perform goes as follows:

```
Create a List of Cells called possibleMoves, and start it as a blank list.

If the current agentCell's y position is not 0, and the cell above it is not a wall

    Then add the result from map.moveUp to possibleMoves

End If
```

I then repeat this for each possible move, making sure to check if it's not on an edge that affects the move, and that if it performs the move, it won't be placed inside a wall.

For the GBFS, I aim to order the list in ascending order, and then reverse the list order. Because of this, it doesn't matter what order I add moves to the possibleMoves list, but I've still put them in the correct order for readability. The main section of this is that it orders the selectedMoves list by the totalCost function that each Cell contains. This specific code is found within the Cell class, and its structure is as follows:

```
Create a minDistance int that is equal to the max value of int

Create a total in equal to 0

Go through each item in goalDistance, increasing the value of int i once per increase
until you are equal with or greater than the count of goalDistance

    Add the goalDistance at index i to the total

    If the goalDistance at index i is smaller than the minDistance

        Then make minDistance equal to the goalDistance at index i

    End If
```

End For loop

Return the minDistance plus 0.5 times the total

The reason why I add 0.5 to the minDistance before multiplying the whole thing by the total is to ensure that the result will always be least when the cell is closest to one of the goals. This should mean that the agent will always move to the cell closest to the majority of the goals while prioritising the closest goal.

The final main implemented search method is the AS. This performs almost identically to the GBFS except while calculating the order, it adds up the totalCost of the move with the moveCost that each Cell has. Unfortunately, each move having the same cost, means that it doesn't change the result much.

When I implemented the jump searches, this added moveCost became more important. The way the jump function works is that it changes the moveCost based on how far it's moved. It calculates this by moving in a direction until it no longer has a wall in the same space as it, or until it's about to jump out of bounds. Then I put 2 to the power of a number obtained by checking how many times it had to move - 1. This means the further the agent has to jump, the more it costs exponentially.

The first of the custom searches, the BDS, uses an algorithm that traverses both towards the goal from the agent's starting location and away from the goal from the goal's location. It does this in a lengthy process by creating duplicate versions of each hashSet and Queue for the other 2 items moving and then performing the normal actions. The agentCell checks to see if the cell it's trying to move to has a parent, and if it does it knows that it's found a connection with one of the goal cells.

Features / Bugs / Missing

Currently the Bidirectional Search can only support environments with 2 goals, meaning that for different solutions it will only look for 2 out of the total goals, or for problems with fewer goals, just completely break.

Currently, the Bidirectional Search doesn't print the last step needed to complete the full moves. It knows that it has reached the end, just not to print that last move.

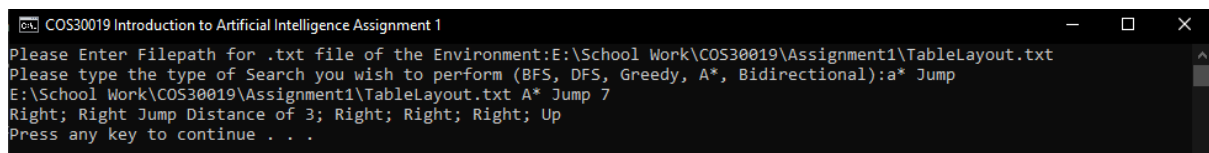
The program is missing the second Custom Search.

Research

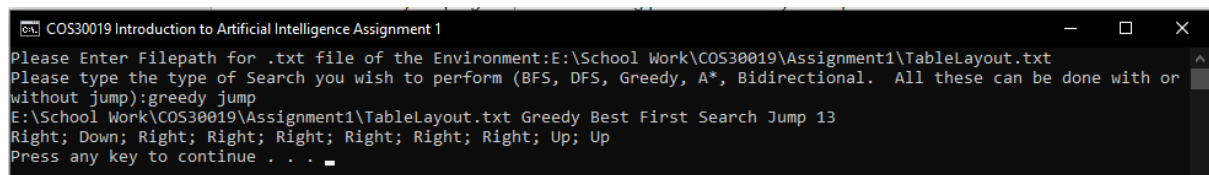
The main research initiative I performed in this project was to add a jump feature to try and produce faster paths that opened fewer nodes. This worked for the most part, and I mentioned most of its key features in the above sections. This proved to be a worthwhile endeavour as it produced some vastly different results to the normal movement, and even found what I believe to be the most optimal search option.

Conclusion

Out of all the tested search algorithms, by far the 2 best-performing ones was GFBS and AS.



*Figure 9 Result from AS Jump,
Screenshot from running program, by Kyle Gibbs 19/04/2023*



*Figure 10 Result from GBFS Jump,
Screenshot from running program, by Kyle Gibbs 19/04/2023*

While GFBS makes the most sense when all movement options cost the same, the added jump command makes AS the best search option. Not only does it get to the goal using very few moves, but it also does open the least amount of nodes. There are definitely some areas of the code that can be improved with all the different implementations, most notably the ones that are repeated a lot.

Acknowledgements / Resources

During my work I used a webpage from javaTpoint called Informed Search Algorithms. I primarily used this website to better understand the heuristics when it came to implementing them for both GBFS and AS. It was also a good resource to quickly remember the difference between each one.

In order to fully understand stacks and their roles in coding, I used a website called TutorialsTeacher, and their very helpful page called C# - Stack<T>. This assisted me in being able to use stacks correctly in my code.

Due to my lack of knowledge on how to use batch files, I had to research how to properly use them. For this, I used a very helpful website called Make Use Of and used the page titled How to Create a Batch (BAT) File in Windows: 5 Simple Steps. With this, I could get the basics for the batch file down, and through a lot of trial and error, I got the BAT file working exactly as I wanted.

Referencing

javaTpoint 2021, *Informed Search Algorithms*, javaTpoint, viewed on 8 April 2023, <<https://www.javatpoint.com/ai-informed-search-algorithms>>

TutorialsTeacher ND, *C# - Stack<T>*, TutorialsTeacher, viewed on 28 March 2023, <<https://www.tutorialsteacher.com/csharp/csharp-stack>>

Buckley, Ian 2022, *How to Create a Batch (BAT) File in Windows: 5 Simple Steps*, viewed on 14 April 2023, <<https://www.makeuseof.com/tag/write-simple-batch-bat-file>>