BFS: Hide and Seek

Tecnológico de Monterrey

Programming Language Final Project
Benjamín Valdés
Javier Iñaqui Aicinena Vargas
A01700585
11/22/2019

Contents

Hide and Seek

Introduction

Hide and seek is a popular children's game in which any number of players hide themselves in a set environment, to be found by one or more seekers. The game is played by one player chosen closing their eyes and counting to a predetermined number while the other players hide. After reaching this number, the player who is counting attempts to locate the rest of the players.

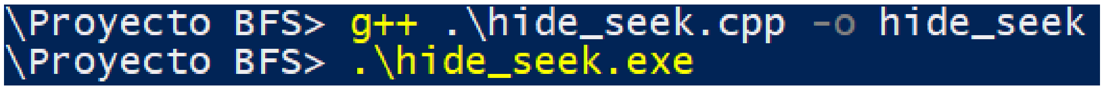
The purpose of this project is to simulate a simple hide and seek game, where only 2 players are involved, the one who hides(H) and the one who searches (S). Added to this, a Breadth First Search algorithm was implemented to help the Searcher find the other player with the shortest possible path.

Instructions of use

The set is represented as a matrix which contains 5 characters:

Character	Description
'S'	Player who searches.
'H'	Player who hides.
'_'	Empty space.
'#'	Barrier.
'|'	Solution path

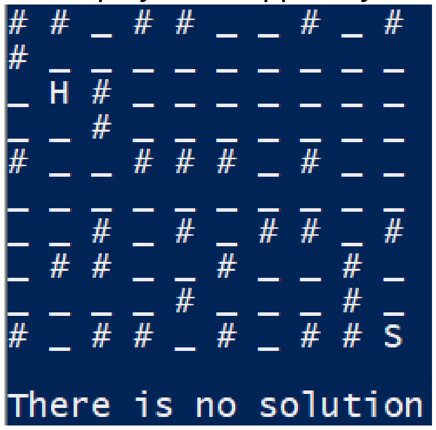
Since this is not an interactive game the only thing you need to do is compile the hide_seek.cpp file.



```
\Proyecto BFS> g++ .\hide_seek.cpp -o hide_seek
\Proyecto BFS> .\hide_seek.exe
```

There can be 2 possible outputs:

- **No solution:**
  Either player is trapped by barriers.

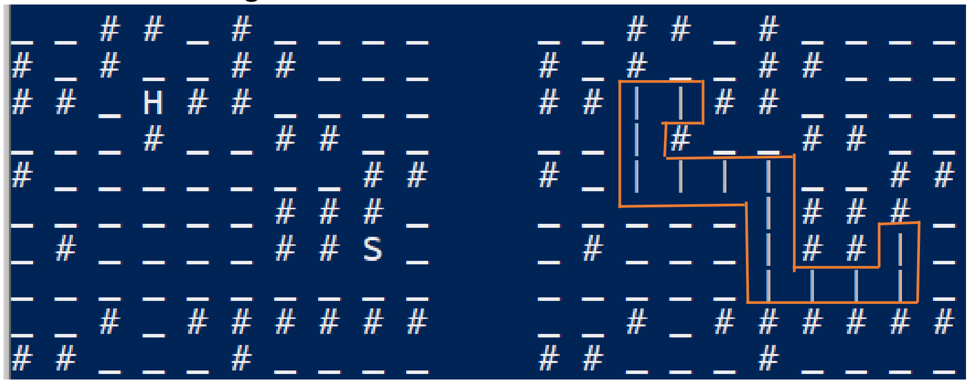


```
# # _ # # _ _ # _ #
# _ _ _ _ _ _ _ _ _
_ H # _ _ _ _ _ _ _
_ _ # _ _ _ _ _ _ _
# _ _ # # # _ # _ _
_ _ _ _ _ _ _ _ _ _
_ _ # _ # _ # # _ #
_ # # _ _ # _ _ # _
_ _ _ _ # _ _ _ # _
# _ # # _ # _ # # S

There is no solution
```

- **Shortest Path:**
  A new Matrix is printed with the shortest path.

Original **Solved**



```
_ _ # # _ # _ _ _ _
# _ # _ _ # # _ _ _
# # _ H # # _ _ _ _
_ _ _ # _ _ # # _ _
# _ _ _ _ _ _ _ # #
_ _ _ _ _ _ # # # _
_ # _ _ _ _ # # S _
_ _ _ _ _ _ _ _ _ _
_ _ # _ # # # # # #
# # _ _ _ # _ _ _ _
```

```
_ _ # # _ # _ _ _ _
# _ # _ _ # # _ _ _
# # | | # # _ _ _ _
_ _ | # _ _ # # _ _
# _ | | | | _ _ # #
_ _ _ _ _ | # # # _
_ # _ _ _ | # # | _
_ _ _ _ _ | | | | _
_ _ # _ # # # # # #
# # _ _ _ # _ _ _ _
```

The orange line is illustrative.

Why was this project chosen?

The reason I chose this project is because it seems to me that we can all go back to our childhood when we played hide and seek where the role of searching was not desired by the players since it was difficult to find them without losing.

This solution proposes a fun way to find players, and at the same time, helps people understand a searching algorithm such as breadth first search, which is often difficult to understand.

Solution

The way in which breadth first search was implemented in this problem was using a queue of the following nodes to visit (Q), a matrix that keep track of the visited nodes (visited) to avoid going through the same place twice, and finally the matrix of previously visited node (previous) to obtain the path with the help of a stack.

To have a better understanding on how the algorithm works let's go through a small example and their iterations.

- **Iteration 0:** We must mark the starting position in the visited matrix with a '1' and push that same position into the queue. Since there is no previous position the Previous matrix stays the same.

Starting Set

	0	**1**	**2**	**3**
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	**1**	**2**	**3**
0	0	0	0	0
1	0	0	0	0
2	1	0	0	0
3	0	0	0	0

Previous

	0	**1**	**2**	**3**
0	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
1	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
2	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
3	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}

Q

0
{0, 2}

- **Iteration 1:** We are going to pop the first element of the queue (Blue cell) to see which of their adjacent nodes are possible path solutions, in this case since it can only move up or down, those will be added to the queue and marked in the Visited matrix and their previous node will be added to the corresponding matrix.

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	1	0	0	0
3	1	0	0	0

Previous

	0	1	2	3
0	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
1	{0, 2}	{-1, -1}	{-1, -1}	{-1, -1}
2	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
3	{0, 2}	{-1, -1}	{-1, -1}	{-1, -1}

Q

0	1
{0, 1}	{0,3}

- **Iteration 2:** Repeat until the queue element that you are checking is the same as the Hidden player position (H).
-

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	0	0	0
3	1	0	0	0

Previous

	0	1	2	3
0	{0, 1}	{-1, -1}	{-1, -1}	{-1, -1}
1	{0, 2}	{0, 1}	{-1, -1}	{-1, -1}
2	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
3	{0, 2}	{-1, -1}	{-1, -1}	{-1, -1}

Q

0	1	2
{0, 3}	{1, 1}	{0, 0}

- **Iteration 3:** At this point you should already understand how the algorithm works, you must repeat the previous steps, if you still have doubts just keep reading, otherwise you can go to iteration 10.

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	🟦 _	_	_	_

Visited

	0	1	2	3
0	1	0	0	0
1	1	1	0	0
2	1	0	0	0
3	1	1	0	0

Previous

	0	1	2	3
0	{0, 1}	{-1, -1}	{-1, -1}	{-1, -1}
1	{0, 2}	{0, 1}	{-1, -1}	{-1, -1}
2	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
3	{0, 2}	{0, 3}	{-1, -1}	{-1, -1}

Q

0	1	2
{1, 1}	{0, 0}	{1, 3}

- **Iteration 4:**

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	🟦 _	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	0
1	1	1	1	0
2	1	0	0	0
3	1	1	0	0

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{-1, -1}
1	{0, 2}	{0, 1}	{1, 1}	{-1, -1}
2	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
3	{0, 2}	{0, 3}	{-1, -1}	{-1, -1}

Q

0	1	2	3
{0, 0}	{1, 3}	{2, 1}	{1, 0}

- **Iteration 5:** In this case since the adjacent nodes have already been visited, we just pop the element from the queue and go to the next one.

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	0
1	1	1	1	0
2	1	0	0	0
3	1	1	0	0

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{-1, -1}
1	{0, 2}	{0, 1}	{1, 1}	{-1, -1}
2	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
3	{0, 2}	{0, 3}	{-1, -1}	{-1, -1}

Q

0	1	2
{1, 3}	{2, 1}	{1, 0}

- **Iteration 6:**

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	0
1	1	1	1	0
2	1	0	0	0
3	1	1	1	0

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{-1, -1}
1	{0, 2}	{0, 1}	{1, 1}	{-1, -1}
2	{-1, -1}	{-1, -1}	{-1, -1}	{-1, -1}
3	{0, 2}	{0, 3}	{1, 3}	{-1, -1}

Q

0	1	2
{2, 1}	{1, 0}	{2, 3}

Iteration 7:

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	0
1	1	1	1	1
2	1	0	1	0
3	1	1	1	0

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{-1, -1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{-1, -1}
3	{0, 2}	{0, 3}	{1, 3}	{-1, -1}

Q

0	1	2	3
{1, 0}	{2, 3}	{3, 1}	{2, 2}

- **Iteration 8:**

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	0
1	1	1	1	1
2	1	0	1	0
3	1	1	1	0

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{-1, -1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{-1, -1}
3	{0, 2}	{0, 3}	{1, 3}	{-1, -1}

Q

0	1	2
{2, 3}	{3, 1}	{2, 2}

- **Iteration 9:**

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	🟦 _	_

Visited

	0	1	2	3
0	1	1	0	0
1	1	1	1	1
2	1	0	1	0
3	1	1	1	1

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{-1, -1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{-1, -1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Q

0	1	2
{3, 1}	{2, 2}	{3, 3}

- **Iteration 10:** Even though we have already found the hidden player, we must keep going until the first element of the queue is that position.

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	🟦 _
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	1	0	1	1
3	1	1	1	1

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Q

0	1	2	3
{2, 2}	{3, 3}	{3, 0}	{3, 2}

- **Iteration 11:**

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	1	0	1	1
3	1	1	1	1

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Q

0	1	2
{3, 3}	{3, 0}	{3, 2}

- **Iteration 12:**

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	1	0	1	1
3	1	1	1	1

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Q

0	1
{3, 0}	{3, 2}

- **Iteration 13:** We have finally reached our goal! Now we must build the path

Starting Set

	0	1	2	3
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

Visited

	0	1	2	3
0	1	1	0	1
1	1	1	1	1
2	1	0	1	1
3	1	1	1	1

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

0
{3, 2}

The way we are going to get the path is by following the Previous matrix and adding them to the stack.

- **Iteration 1:** We will start at the hidden player, push its position and go through their previous nodes to push them as well into the stack.

Previous

	0	1	2	3
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Path

{3, 1}
{3, 0}

- **Iteration 2:**

Previous

	0	**1**	**2**	**3**
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Path
{2, 1}
{3, 1}
{3, 0}

- **Iteration 3:**

Previous

	0	**1**	**2**	**3**
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Path
{1, 1}
{2, 1}
{3, 1}
{3, 0}

- **Iteration 4:**

Previous

	0	**1**	**2**	**3**
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Path
{0, 1}
{1, 1}
{2, 1}
{3, 1}
{3, 0}

- **Iteration 5:**

Previous

	0	**1**	**2**	**3**
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Path
{0, 2}
{0, 1}
{1, 1}
{2, 1}
{3, 1}
{3, 0}

- **Iteration 6:** We have reached the start node, and we know it because its previous node is set to {-1, -1}.

Previous

	0	**1**	**2**	**3**
0	{0, 1}	{1, 1}	{-1, -1}	{3, 1}
1	{0, 2}	{0, 1}	{1, 1}	{2, 1}
2	{-1, -1}	{-1, -1}	{2, 1}	{3, 1}
3	{0, 2}	{0, 3}	{1, 3}	{2, 3}

Path
{0, 2}
{0, 1}
{1, 1}
{2, 1}
{3, 1}
{3, 0}

Therefore, our path is [{0, 2}, {0,1}, {1,1}, {2,1}, {3,1}, {3,0}].

Starting Set

	0	**1**	**2**	**3**
0	_	_	#	H
1	_	_	_	_
2	S	#	_	_
3	_	_	_	_

GitHub repo

https://github.com/JIAV94/BFS/blob/master/hide_seek.cpp

Recorded Video

https://drive.google.com/file/d/13OBR8wZRHOCA7hB9qNVQ0lB2F3cZbkuc/view?usp=sharing

How could this project be extended?

This project could be extended by adding more players to be found and return their respective paths. Also, a graphic interface could be really helpful because as it is right now is hard to look at the elements of the “playground” (map).

Also there could be a comparison between several searching algorithms and evaluate the solving time.

References

Hide-and-seek. (n.d.). Retrieved November 23, 2019, from https://en.wikipedia.org/wiki/Hide-and-seek.

Fiset, W. (2019, May 29). Breadth First Search Adjacency List Iterative. Retrieved November 24, 2019, from https://github.com/williamfiset/Algorithms/blob/master/com/williamfiset/algorithms/graphtheory/BreadthFirstSearchAdjacencyListIterative.java.