Queen's UNIVERSITY

GRP_71: Pathfinding

Ethan Callanan (18ecc6)

Adela Yang (19hy7)

Rami Ballalou (18rb54)

Max Kang (18mjbk)

Course Modelling Project

CISC/CMPE 204

Logic for Computing Science

December 6, 2020

Abstract

Path-finding is a very relevant and useful AI application, especially with the advent of self-driving cars. This project aims to assess if there is a possible path from a start point to an end point; as well as how many and what they all are. As in real life, there can be blockages and not all paths are valid.

A path-finding context was simulated through the use of a model that will correspond a square grid of adjustable size, and locate a path of accessible squares; if no such model exists, then there is not a valid pathway from the start to finish. The model will return a list of variables representing the squares in the grid.

The squares are all defined and set under a list, with the coordinates of each square being an element. All the inaccessible or blocked squares are set to False, while the squares that are accessible are set to True. The same is seen for path list, there's a new list for each square within the grid, True or False is assigned to each element to depict whether the square was used in the path or not.

To achieve this within propositional logic, we wrote a custom depth first search and backtracking algorithm in Python.

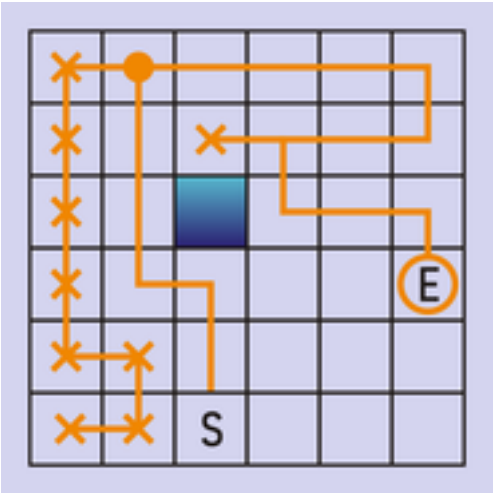

Diagram 1

A depth first search works by starting at the starting block and exploring as far as possible along a path, it'll calculate each possible next square and the corresponding visiting list to make the next move. If a path is not found after moving in a valid direction, then backtrack and try another direction. To further include propositional logic, each square that is accessible is added as a constraint otherwise it is negated. Once the end has been reached, it'll be added as a constraint. The algorithm also prevents the path from going back over a square it's already been on by checking that each possible next square move is not in the path list already.

The model is designed to recursively generate all of paths possible from starting square to the ending square. It'll find a path to the end from each of the current squares neighbors, and keep branching out to return all the paths possible. We ensured to note how many of those solutions (paths) are unique solutions.

The pathfinder is capable of working in a multitude of different contexts, as squares can be made accessible and inaccessible. Our grid size is also a variable and therefore changeable allowing a lot of flexibility and diversification.

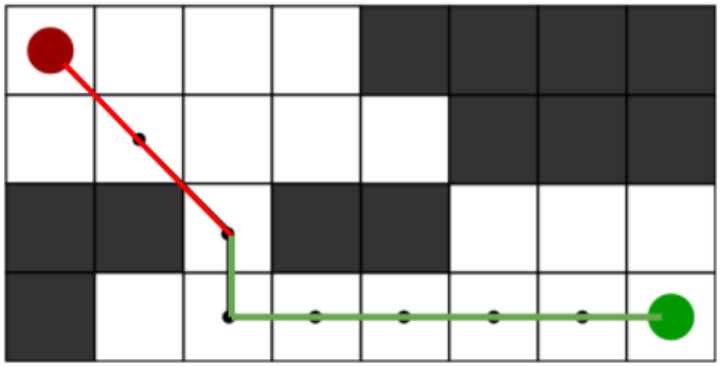

Diagram 2

It's also important to note that the movement of the path finder is restricted to horizontal and vertical, but NOT diagonal as shown in Diagram 2. The red path represents an illegal move while the green path is a legal move.

Propositions

- **xij**: Each xij represents whether the corresponding square in the grid is accessible or not. If the proposition is true the square is accessible, if it is false the square is inaccessible. One of these propositions will be set as the starting square (**s**) and one will be set as the ending square (**e**).
- **yij**: Each yij represents whether the corresponding square in the grid is in the path or not. If the proposition is true the square is in the path, if it is false the square is not in the path.

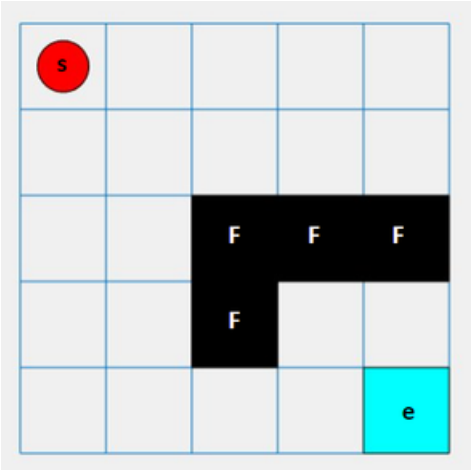

Diagram 3

Diagram 3 offers a visual representation of the propositions in a single grid example. The starting square is s = x00, and the ending square is e = x44. The inaccessible squares are x22, x23, x24, and x32 and they will be False in the model. The rest of the squares are accessible and will be True.

Constraints

The constraints for this model are generated by the path finding algorithm based on the grid configuration defined by the user.

- $\mathbf{A_{ij}} \wedge \cdots \wedge \neg\mathbf{B_{ij}} \wedge \cdots \wedge \mathbf{S} \wedge \mathbf{E}$: This is the constraint that defines the accessible and inaccessible square in the grid. S is the grid proposition for the starting square, E is the grid proposition for the end square. There is an A_{ij} (grid proposition) for each accessible square, and a B_{ij} (grid proposition) for each inaccessible square. Note that the start and end squares will also exist as either an A_{ij} or B_{ij} depending on whether the user makes the squares accessible or not. This prevents the SAT solver from finding any solutions if the user blocks the start or end square, as there will be a contradiction ($S \wedge \neg S$ or $E \wedge \neg E$).

- $(\mathbf{G} \vee \mathbf{Y}) \wedge (\mathbf{Y} \vee \neg\mathbf{Y})$: This is the main constraint for the single pathfinder that determines if a square should be in the path or not. It is derived from $Y \vee (G \wedge \neg Y)$ where Y is the proposition for a square and G is the output from calling the path generator (recursively) on the previous square. In other words, the constraint states that a square is either in the path or there is a path from the previous square to the end that does not use this square. This results in the pathfinder finding the most logically complex path possible.

- $\mathbf{P_1} \vee \mathbf{P_2} \vee \cdots \vee \mathbf{P_n}$: This is the constraint used by the all paths pathfinder. Each P_k is a valid path for the given grid configuration. By constraining the disjunction of all the paths, the SAT solver is able to find a model for each of the paths (as well as models that satisfy more than just one path). This allows us to count the number of possible paths and the number of unique paths.

- $\mathbf{P_k} = \mathbf{S} \wedge \mathbf{Y_{ij}} \wedge \cdots \wedge \mathbf{E}$: This is the definition of each disjoined proposition in the above constraint. S is the path proposition that corresponds to the starting square, Y_{ij} is the path proposition for a square in the path, and E is the path proposition that corresponds to the ending square.

Model Exploration

The notion of how we defined our grid and squares has changed. At first, each square was represented by a variable, then an object was used to hold all the squares with properties for the coordinates, but finally we found the best solution was to use individual propositions for grid squares and path squares, and represent the coordinates in a third list of tuples. All the lists are generated at the same time so the indexing is consistent throughout.

Following the feedback given by Prof. Muise, we got rid of a constraint which ensured that each path proposition was true. We removed this constraint as it'd make our model way more optimized as this constraint would exist for each path possible, which could be millions theoretically. As advised by Prof. Muise, we also added more functionality to our model such as finding out how many paths exist, and what paths they are instead of just looking at whether there is a path. This would render our model more than just a one-dimensional project.

In the beginning of development, a constant static grid was considered, however, we decided against that. We added a depth of flexibility by introducing a settings.py, which would allow the user to change the majority of variables such as size of grid, start coords, end coords, and inaccessible squares. This allows us to see how the model reacts and handles different situations, which opens a window of infinite possible arrangements.

Our pathfinder was able to find every path possible from start to end, however, with all those paths found it was also important to find what paths were unique. Our program is also able to compute the likelihood of each square being in a path at the end of run time. We explored blocking the start and end to ensure no solution was returned, as a base test case.

Negating any constraints in the algorithm caused the SAT solver to either find no solutions, or find incorrect solutions. This reinforces the importance of each constraint in the final model.

During development we tried many different solutions. Our initial approach was to run through the grid sequentially, adding constraints in the form of implications (changed to CNF) for each square that determine whether or not it can be in a path. This approach was deeply flawed as if the starting square wasn't at the coordinates (0, 0) and the end square wasn't at (N, N) (where N is the size of the grid), then it would be impossible to properly calculate the constraint for a square. We also tried a recursive approach without backtracking which also didn't work. Without backtracking, the SAT solver would reach a contradiction (resulting in no solutions) as soon as it hit an inaccessible square.

First-Order Extension

The first order extension was done through extending our model to a more predicate logic setting, updating both proposition and constraints. This is done through looking for areas in our constraints which could be replaced by keywords such as "all", "every", "only", "at least" etc. Using these keywords, we're able of converting our propositional clauses into predicate logic clauses.

- Universe of Discourse: A = Grid Size: i x j, the universe of discourse will be defined to include every square within the grid depending on the grid size.
- s = xij = True for starting location. $\Rightarrow \exists$(s= xij). There exists at least one starting point in any scenario of our propositional model.
- e = xij = True for ending location. $\Rightarrow \exists$(e= xij). There exists at least one ending point in any scenario of our propositional model.

- $P_k = S \wedge Y_{ij} \wedge \cdots \wedge E \Rightarrow P_k = \forall(Y_{ij})(S \wedge Y_{ij} \wedge E)$. Ensuring that every path proposition no matter the coordinate for the square is in the path, the grid would be the universe of discourse ensuring no coordinate that's not apart of the grid is inputted.

References

- Bettilyon, T., 2020. Breadth First Search And Depth First Search. [online] Medium. Available at: ¡https://medium.com/tebs-lab/breadth-first-search-and-depth-first-search-4310f3bf8416¿ [Accessed 30 November 2020].
- GeeksforGeeks. 2020. A* Search Algorithm - Geeksforgeeks. [online] Available at: ¡https://www.geeksforgeeks.org/a-search-algorithm/¿ [Accessed 30 November 2020].