A* Algorithm for Solving Sokobond Puzzles

Addin Munawwar Yusuf - 13521085
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): moonawardev@gmail.com

***Abstract*—The A* algorithm is a common algorithm that is used for pathfinding optimization in computer science problems and game development. The sokobond game is one of the grid-based puzzle games that can be solved optimally using the A* algorithm. The paper shows that with A* algorithm with manhattan distance heuristic, the optimal solution to Sokobond puzzle level can be achieved. (*Abstract*)**

Keywords—A* algorithm; sokobond; pathfinding; puzzle; game development; optimization;manhattan distance;

I. Introduction

Sokobond is a puzzle game that revolves around the concept of chemical bonding. The game is played in a grid-based area where atoms are scattered throughout the environment. The goal is simple: arrange the atoms in such a way that it satisfies the chemical bonding rules. The Sokobond game is made and developed by two indie developers: Alan Hazelden and Harry Lee. It was released in August 2013 in various platforms, including Windows, macOS, Linux, and just recently for Nintendo Switch in September 2021. From its first release, it quickly gained recognition from many puzzle game communities and even received many good critics, due to its unique gameplay and simple, yet clever level design. In metacritics, Sokobond got a rating of 82 with a total of 7 critics reviews.

The development of Sokobond was inspired by the idea of merging chemistry and puzzles. Alan Hazelden, known for his work on puzzle games such as "Hexcells" and "Ruckingenur II," collaborated with Harry Lee, a chemist and game developer, to create a game that would introduce players to the concept of chemical bonding through engaging puzzles. The name "Sokobond" was inspired by a classic puzzle game "sokoban", where players push crates to certain locations. Sokobond takes the similar approach of grid-based format from this game with the twist of chemical bonding mechanics.

Although the game uses the concept of chemical bonding, players don't really need to have prior knowledge regarding that, as the developers make it easy for players to intuitively understand the rules of the game. The gameplay itself is quite simple. As said before, the game is played in a small grid-based area with atoms distributed throughout the environment. The player is initially given control of one atom or element in that area. Every element has a set number of bonding electrons that is specific to particular elements. Carbon has four, nitrogen tree, oxygen two, hydrogen one, etc. The rule of chemical bonding comes into play when two compatible atoms in the arena are brought into contact. The two compatible atoms will then create a bond and form a molecule. For instance, if two hydrogen atoms and an oxygen atom are brought into contact, they bond to form a water molecule.

As the game progresses to later stages, the game will become more difficult. Some new different mechanics are introduced and may even cause a lot of players to be stuck at a level. Although anyone can play this game, a prior knowledge of basic chemistry will definitely help players to progress more easily.

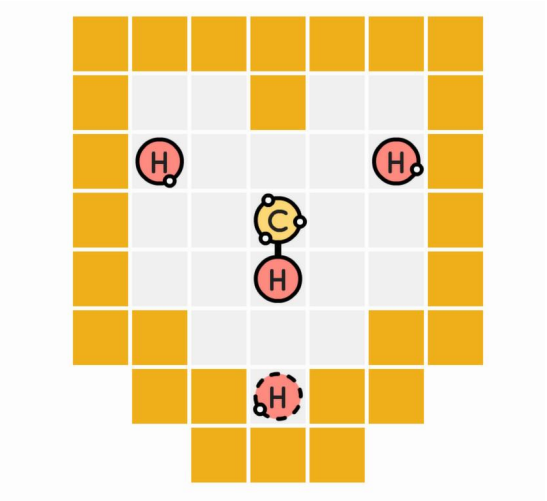


Fig. 1. One of the early level in Sokobond game
(Source: https://store.steampowered.com/app/290260/Sokobond/)

A* Algorithm is a popular route finding algorithm that efficiently could find the shortest path between two nodes and produce optimal solutions. It considers the cost to reach the node, and a heuristic/ estimated remaining cost from node to goal as guidance for the search route. A* explores the graph by iteratively selecting the node with the lowest total cost, calculated as the sum of the cost to reach the node and the estimated cost to the goal. The applications of A* algorithm are such as route planning, puzzle solving, game AI, etc.

In the context of using A* algorithm in Sokobond, A* algorithm is used to find the most optimal solution to a sokobond level. In this paper, the author will explore how A* algorithm can be implemented to solve some early levels in the first stage of Sokobond game (before some new mechanics added in the second stage) and discuss how sokobond level can be mapped into A* problem. In the later section, there will also be some tests of the algorithm.

II. Base Theory

A. A* Algorithm

A* Algorithm is one of the most popular algorithm used for pathfinding and graph traversal. It can be considered as the extension of the earlier algorithm, Dijkstra algorithm. The difference between the two is, while in Dijkstra algorithm, the goal node is every other nodes except the starting node, A* algorithm only has one specific goal node. Besides that, A* also uses heuristic to fasten up the search toward the goal node.

A* algorithm is a kind of informed search, meaning that it has the information about the goal node, and thus, can lead the algorithm to reach the goal node faster. The goal of this algorithm is to find the most optimal route to the given goal node with the cost as minimum as possible. Cost can be in the form of the amount of steps, time, distance traveled, and others, depending on the context of the problem. It does so by maintaining a tree of paths, starting from the beginning node, and extending each node in some priority order until it reaches the goal node.

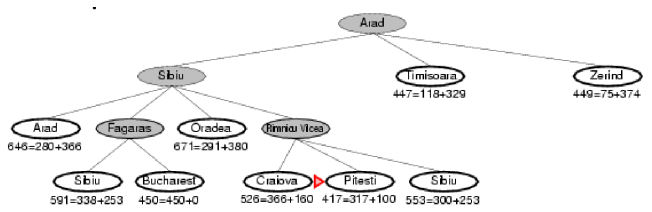


Fig. 2. A* algorithm tree of paths example
(Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf)

When expanding the tree of paths, A* algorithm needs to choose which path should be prioritized to be extended next. It does so by choosing the path that generally minimizes (can also be maximized in maximization cases) the total cost, calculated as the sum of the cost to reach the node and the estimated cost to the goal. It generally follows the formula,

$$f(n) = g(n) + h(n)$$

For. 1. Formula for total cost of a path node
(Source: author documentation)

where $g(n)$ is the cost to reach node n, and $h(n)$ is the heuristically estimated remaining cost from node n to goal node. $f(n)$ is the total cost that should be minimized or maximized when choosing a path to expand. Optionally, when generating a new path, we can also prune some paths that are not leading towards the goal node.

A* is often used and liked among programmers due to its nature of:

- Completeness : Yes, unless there are infinitely many nodes with $f \leq f(G)$.
- Time Complexity : Exponential: $O(b^m)$, with b as branching factor and m as depth of optimal solution.
- Optimal Solution : Yes, as long as $h(n)$ is admissible (will be discussed in later section).
- Space Complexity : $O(b^m)$. Not as efficient because it has to store every node in the memory.

The big picture step by step of A* algorithm is as follows.

1) Initialize the priority queue.
2) Initialize starting node. Set the starting node as the current node.
3) If the current node is a goal node, stop search.
4) Generate children of the current node and add it to the priority queue, with priority value of total cost ($f(n) = g(n) + h(n)$). Optionally, prune every child that doesn't lead to the goal node.
5) Pop element with lowest cost from priority queue and set it as current node.
6) Continue step 3-6 until the path is found.

The pseudocode of A* algorithm is as follows.

A* Algorithm Pseudocode

```
Local Dictionary
   pQ : priorityQueue {sorted by total cost
   ascending}
   goal : Node {goal Node}

   struct Node {
        value : Element
        path : char[]
        totalCost : int
   }
```

```
function aStar(node : Node) -> char[]:
{Returns the solution path from start to goal
Node}
    if (node is Goal):
        -> node.path
    else:
        for Node n in generateChild(node):
                pQ.add(n, n.totalCost)
            if (pQ.empty):
                -> no solution
            else:
                -> aStar(pQ.pop())

function generateChild(node : Node) -> Node[]:
{Generate promising path from node}
        nodes : Node[]
        for direction in possibleDirection:
          if (node.promising(direction)):
              {pruning}
              {fn = gn + hn}
              nodes.add(new Node(value, path +
direction, totalCost + fn))
        -> nodes
```

Snapshot. 1.Pseudocode of AStar algorithm with recursive approach
(Source: author documentation)

B. Admissible A* Algorithm Heuristic

As mentioned previously, in order for A* algorithm to produce the optimal solution, the heuristic function — $h(n)$ — must be admissible. $h(n)$ is admissible if the value produced by the function never overestimates the actual cost to reach the

goal node. Having a heuristic function that overestimates the actual cost can lead to the search going into the wrong direction, since the heuristic is more powerful than the cost evaluation by $g(n)$. The illustration of how non-admissible heuristic can be seen here.

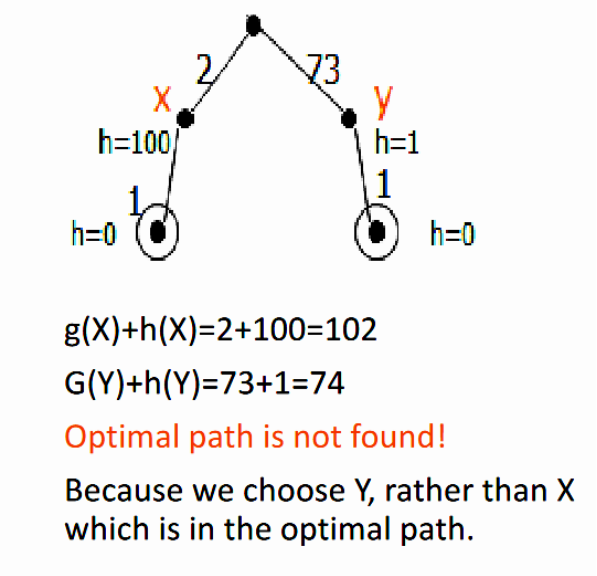


Fig. 3. Non-admissible heuristic leading search to non-optimal solution
(Source: https://store.steampowered.com/app/290260/Sokobond/)

Formally, an admissible heuristic function is defined as $h(n)$, where for every node n , $h(n)$ satisfies $h(n) \leq h^*(n)$, where $h^*(n)$ is the actual cheapest cost from node n to goal node. Consequently, an admissible heuristic function is a heuristic function that never overestimates the cost to reach the goal node, i.e. it is optimistic.

There are two types of heuristic function:

1) Exact heuristic.
2) Approximation heuristic.

Exact heuristic does compute the exact value of h with some algorithm, such as Dijkstra's algorithm, but generally is time consuming. The technique includes pre-compute the distance between each node before running the a* algorithm. If there are no obstacles/blocked cells in the area, exact value of h can just be found without doing pre-computation, by using distance formulas, such as euclidean distance.

Approximation heuristic is a little bit more clever than the exact heuristic. It does not do any pre-computation to calculate the distance of every pair of nodes, but rather uses the information about the goal node to calculate the remaining distance between a node n to goal node. There are many algorithms that are commonly used as approximation heuristic functions. Those are including manhattan distance, diagonal distance, and euclidean distance. These algorithms are admissible in most cases. These are the explanations for each algorithm.

– **Name**:
Manhattan Distance

Description:
Distance is calculated as a sum of absolute values of differences between goal's node x and y, to the current node's x and y respectively.

Pseudocode

```
h = abs(current.x – goal.x) +
      abs(current.y – goal.y)
```

Snapshot. 2.Pseudocode of Manhattan distance
(Source:https://www.geeksforgeeks.org/a-search-algorithm/)

Use case
When we are allowed to move in four directions only (up, down, right, left).

– **Name**:
Diagonal Distance

Description:
Distance is calculated as a sum of absolute values of differences between goal's node x and y, to the current node's x and y respectively.

Pseudocode

```
dx = abs(current.x – goal.x)
dy = abs(current.y – goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx,
dy)

// the calculation more complex due to
possibility of diagonal movement

D is the length of each node(usually = 1)
and D2 is the diagonal distance between
each node (usually = sqrt(2)).
```

Snapshot. 3. Pseudocode of diagonal distance
(Source: https://www.geeksforgeeks.org/a-search-algorithm/)

Use case
When we are allowed to move in eight directions (similar to king movement in chess).

– **Name**:
Euclidean Distance

Description:
Distance is calculated as a square root of the sum of differences between goal's node x and y, to the current node's x and y, squared. Using the pythagorean theorem.

Pseudocode

```
h = sqrt((current.x – goal.x)² +
    (current_cell.y – goal.y)²)
```

Snapshot. 4. Pseudocode of euclidean distance
(Source: https://www.geeksforgeeks.org/a-search-algorithm/)

Use case
When we are allowed to move in any direction.

C. *Sokobond Game*

Sokobond is a puzzle game that revolves around the concept of chemical bonding. The game is played in a grid-based area where atoms are scattered throughout the environment. The goal is simple: arrange the atoms in such a way that it satisfies the chemical bonding rules. The player is initially given control of one atom or element in that area. Every element has a set number of bonding electrons that is specific to particular elements. Carbon has four, nitrogen tree, oxygen two, hydrogen one, etc.

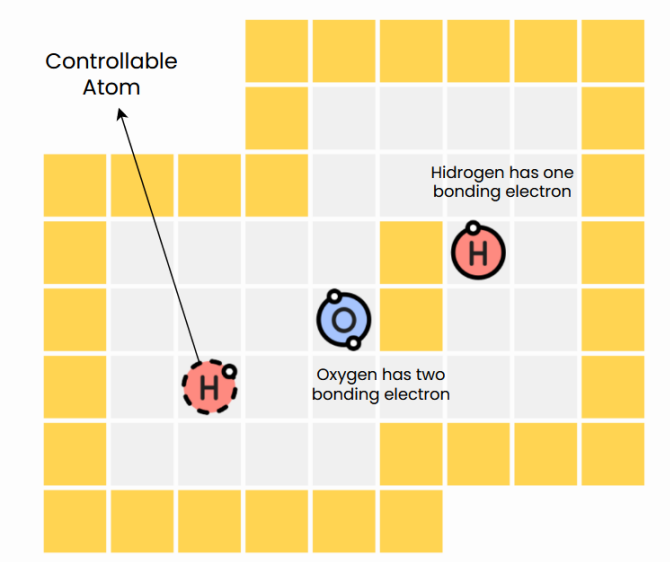
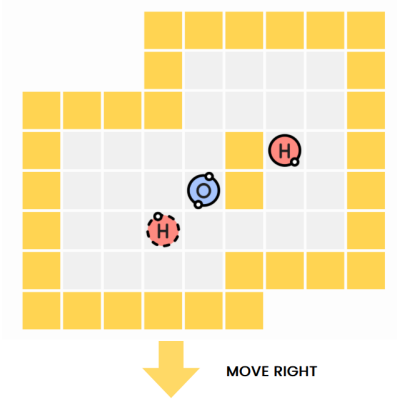


Fig. 4. Example level of sokobond. Hydrogen has one
(Source: author documentation)

The rule of chemical bonding comes into play when two compatible atoms in the arena are brought into contact. The two compatible atoms will then create a bond and form a molecule. For instance, if two hydrogen atoms and an oxygen atom are brought into contact, they bond to form a water molecule. After the atoms are bonded, they can only be moved together. If not careful, those bonds can cause the player to get stuck and make the level unsolvable, at the point where they have to undo until the previous viable point.



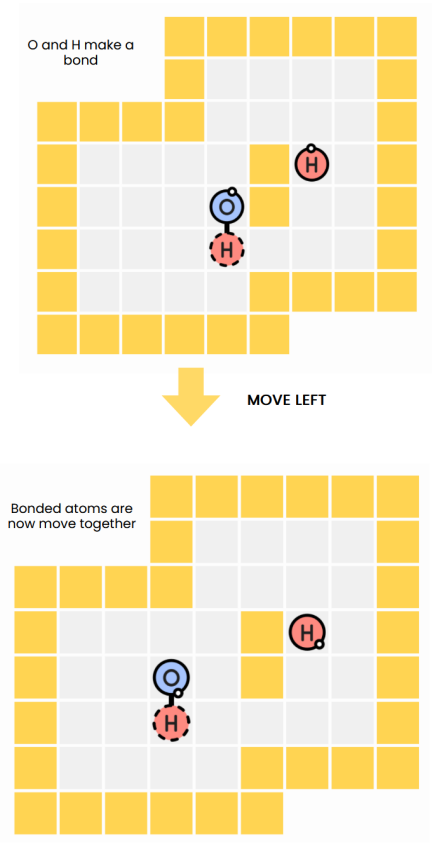


Fig. 5. Demonstration of how atom bonds work
(Source: author documentation)

The level is ended if there are no more free atoms that are scattered around the grid arena, and the atoms have formed a valid chemical molecule or compound.

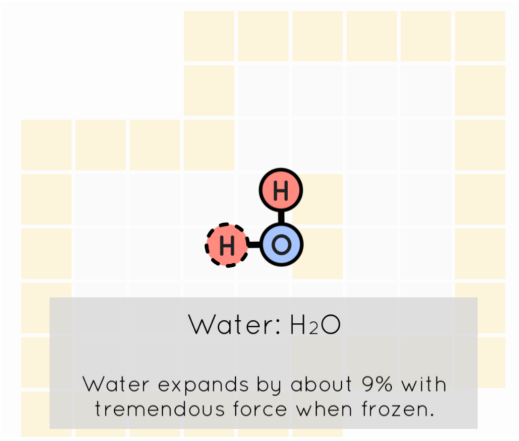


Fig. 6. Level finished in sokobond level
(Source: author documentation)

In each step of the level, there are only four valid input directions that can be performed: MOVE UP, MOVE DOWN, MOVE RIGHT, MOVE LEFT. The solution to the level is just a set of direction inputs that leads to atoms configured in the correct way.

III. IMPLEMENTATION

A. *Mapping Sokobond to A* Algorithm Domain*

1) Solution Space

Solution space of sokobond puzzle can be represented in a vector with n-tuple sized:

$$X = (x_1, x_2, \ldots, x_n)$$

with $x_1, x_2, \ldots, x_n \in$ {UP, DOWN, RIGHT, LEFT}

or for abbreviation

$$x_1, x_2, \ldots, x_n \in \{U, D, R, L\}$$

For. 2. Solution Space for Sokobond in A* Algorithm Domain (Source: author documentation)

with n is the depth of optimal solution in A* search. In the next parts, tuple X might be referred to as a path or solution.

2) Bounding Function

The bounding function in sokobond puzzle is used to prune some of the nodes that are not promising — does not lead to the goal direction. In the Sokobond game, the node is pruned if the movement of the molecule/ compound is illegal, such as stuck into the wall or getting out of the arena bounds.

Another variable that causes a node to be pruned is visited tile. If the tile has been visited and there is not any interaction yet with any free elements in the area, then the path to that visited tile won't be considered. After any interaction is done with a free element, all tiles will be set to unvisited, so the new bonded compound can revisit those tiles.

The pseudocode of the bounding function is as follows.

```
Sokobond Solver Bounding Function

Local Dictionary
   compound: Compound {set of bonded atoms}
   tiles: array of (array of Tile)

   class Compound {
        mainEl : Element
        bondedEl : Element[]
   }

   class Tile {
        type: TileType {Wall, Element, Empty}
   }

   class Vector2 {
       x : int
       y : int
   }

function canMoveCompoundTo(Vector2 direction) ->
boolean
{Return true if compound can be moved in the
direction}
    for Element el in compound:
        targetTile = getTile
             (el.location + direction)
        if (targetTile.type = Empty) then
            -> false
        else if (targetTile.type = Element) then
            if (targetTile can't be pushed to
                direction) then
                 -> false
     -> true
```

Snapshot. 5. Bounding function of sokobond solver (Source: author documentation)

3) Node or State Representation

In sokobond solver, nodes are represented with class as follows.

```
class Node {
    arena : Grid
    path : Character[] {char ∈ U, D, R, L}
    totalCost : int
}
```

Snapshot. 6. Node or state representation of sokobond solver (Source: author documentation)

Grid is a class that represents the arena, containing information about a configuration of atoms in the grid, including their position and bonding status. It captures states of the level in that point after a set of instructions stored in the path.

4) Generating Function

Generating function of the sokobond puzzle is a function that generates new tree path nodes with the BFS approach, then adding the new node to the priority queue. The pseudocode of this function is:

```
Sokobond Solver Generating Function

Local Dictionary
   pQ : priorityQueue {sorted by total cost
   ascending, with comparator}

procedure generateChild(node : Node):
{Generate promising path from node}
       {possibleDirection : U, D, R, L}
       for direction in possibleDirection:
         {pruning}
         if (canMoveCompoundTo(direction)):
             {fn = gn + hn}
             pQ.add(new Node(value, path +
direction, totalCost + fn))
       -> nodes
```

Snapshot. 7. Generating function of sokobond solver (Source: author documentation)

5) Total cost ($f(n)$)

In A* algorithm, the total cost ($f(n)$) is calculated as sum of $g(n)$ and $h(n)$ (as mentioned in the previous section).

In this case, $g(n)$ or total cost to reach node n is the amount of steps taken from the beginning. This can also just be calculated by the length of path in the node.

On the other hand, $h(n)$ heuristic function that author chose for this sokobond solver is manhattan distance, since it fits so well for search that can move in four directions as mentioned before in the previous section. This $h(n)$ is admissible, since manhattan distance calculates the least step taken possible from node n to reach goal node, thus, it never overestimates the cost. Manhattan distance calculated is the distance between the compound and the closest free element.

Here is the pseudocode.

```
class Node {
    arena : Grid
    path : Character[] {char ∈ U, D, R, L}
    totalCost : int

    totalCost {f(n)} <-
         {h(n)} arena.getClosestElmtDist() +
         {g(n)} path.length
}

function getClosestElmtDist() -> int
    int closestDist = int.MAX_VALUE;
    for (Element elmt in freeElements) {
       int dist <- compoundLocation.
          manhattanDistance(elmt.Location);
       if (dist < closestDist) then
           closestDist <- dist;
   -> closestDist;
```

Snapshot. 8. f(n) total cost of sokobond solver
(Source: author documentation)

6) Goal Node

Goal node is reached whenever there is no more free element that hasn't been bonded yet in the arena. The implementation is quite simple. Just check whether the amount of free elements is zero.

```
function isCompoundOnGoal(node : Node) ->
boolean
       -> node.freeElements.size() == 0;
```

Snapshot. 9. goal node checking
(Source: author documentation)

B. *A* Main Algorithm*

Now that all of sokobond element has been mapped to A* problem space, we can now create the main A* algorithm that utilizes all elements that have been mapped previously. Just like the pseudocode that has been discussed about in section II, the A* algorithm for sokobond solver would look like this.

A* Algorithm Pseudocode

```
Local Dictionary
   pQ : priorityQueue {sorted by total cost
   ascending}

   class Node {
       arena : Grid
       path : Character[] {char ∈ U, D, R, L}
       totalCost : int

       totalCost {f(n)} <-
          {h(n)} arena.getClosestElmtDist() +
          {g(n)} path.length
   }
```

```
function solveSokobond(start : Node) -> char[]:
{Returns the solution path from start to goal
Node, uses to initialize AStar}
     generateChild(start)
     -> aStar(start)

function aStar(node : Node) -> char[]:
{Returns the solution path from start to goal
Node}
    if (node.isCompoundOnGoal(node)):
       -> node.path
    else:
        if (pq.isEmpty()) then
           {try generate current}
           generateChild(node)
           if (pq.isEmpty()) then {still empty}
               -> no solution
        else
           generateChild(node)
        -> aStar(pQ.pop())
```

Snapshot. 10. A* algorithm of sokobond solver
(Source: author documentation)

With this algorithm, we can solve the sokobond level now.

C. *Further Reference*

For further reference about the algorithm, you can checkout the author repository of Sokobond Solver program written in Java language here:

https://github.com/moonawar/SokobondSolver.git

IV. Testings

For testing, the author will use the three levels of Sokobond in the first stage. You can also try it yourself with different levels by using the program provided earlier in section III.C. This program is only able to solve the sokobond levels in the first stage. The reason for this is because in the second stage, there is a new different mechanic introduced that changes how the puzzle is played.

For the source config level file of this program, use this example below. You can also see some predefined level configuration in the program repository provided earlier.

```
<numOfRows> <numOfColumns> <N (num of elements)>
<Element1Char> <bondLimit>
<Element2Char> <bondLimit>
...
<ElementNChar> <bondLimit>
<Grid config; '#' for wall, ' ' for empty space,
'<elChar>' for element>
<row [0, numOfRows)> <col [0, numOfColumns)>
--- starting position of element controlled

Example:
8 9 2
H 1
O 2
   ######
   #    #
####    #
#    #H #
#   O#  #
# H     #
#    ####
######
5 2
```

Snapshot. 11. Example config file
(Source: author documentation)

1) `Let's Go` Level

Level layout :

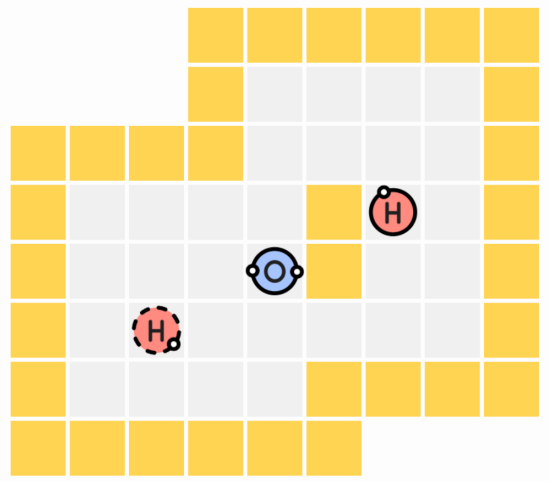


Fig. 7. Level `Let's Go` of the sokobond
(Source: author documentation)

Config File:

1.txt

```
8 9 2
H 1
O 2
   ######
   #    #
####    #
#    #H #
#   O#  #
# H     #
#    ####
######
5 2
```

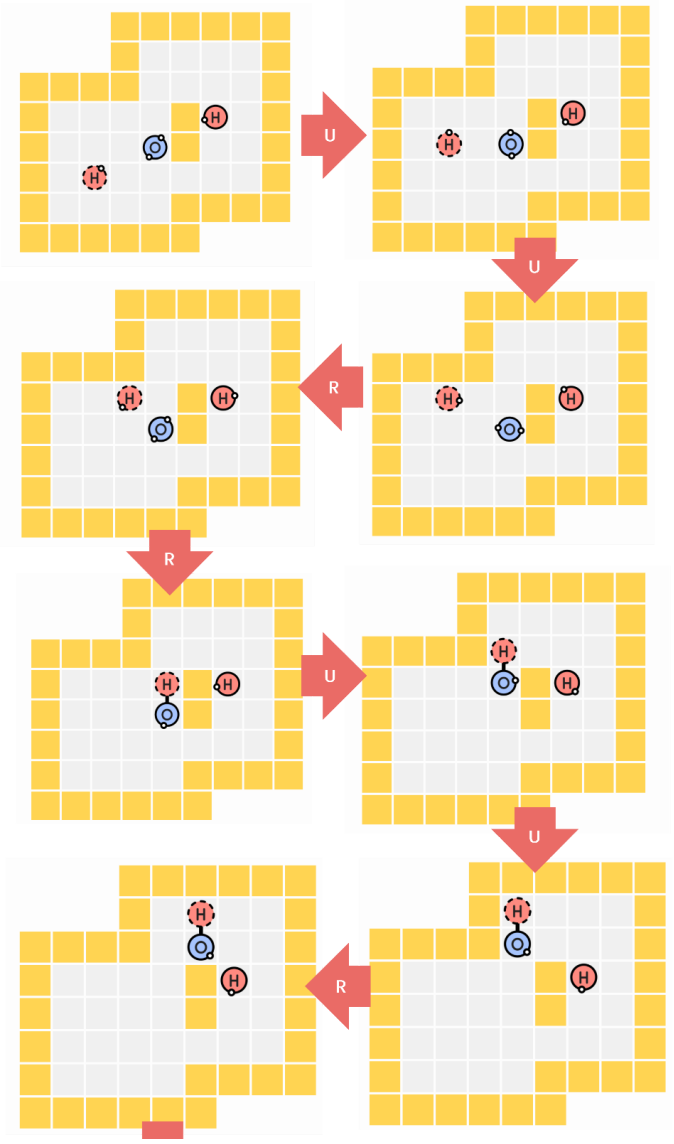
Snapshot. 12. Level 1 config file
(Source: author documentation)

Result :

```
Select the level file:
C:\Addin Munawwar\2023\5. May\sokobond-solver\test\1.txt
Searching for solutions...
[U, U, R, R, U, U, R, R]
Time elapsed: 1192 ms
```

Fig. 8. Level 1 of the sokobond solver output
(Source: author documentation)

Result In Game:



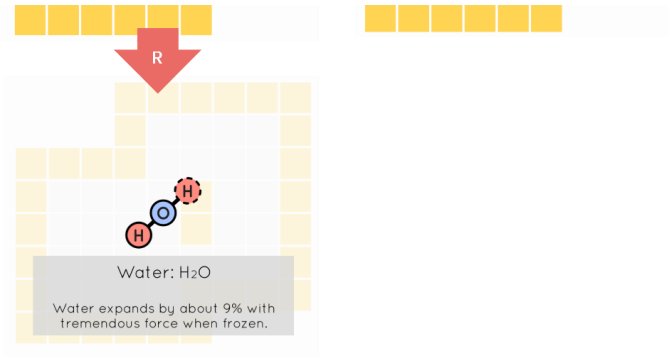


Fig. 9. Level `Let's Go` of the sokobond solver gameplay
(Source: author documentation)

2) `Push` Level

Level layout :

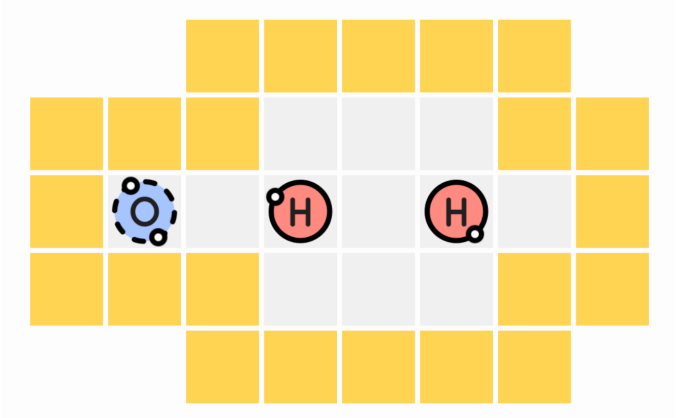


Fig. 10. Level `Push` of the sokobond
(Source: author documentation)

Config File:

2.txt

```
5 8 2
H 1
O 2
  #####
###   ##
#O H H #
###   ##
  #####
2 1
```

Snapshot. 13. Level 1 config file
(Source: author documentation)

Result :

```
Select the level file:
C:\Addin Munawwar\2023\5. May\sokobond-solver\test\2.txt
Searching for solutions...
[R, R, U, R, D, R]
Time elapsed: 102 ms
```

Fig. 11. Level 1 of the sokobond solver output
(Source: author documentation)

Result In Game:

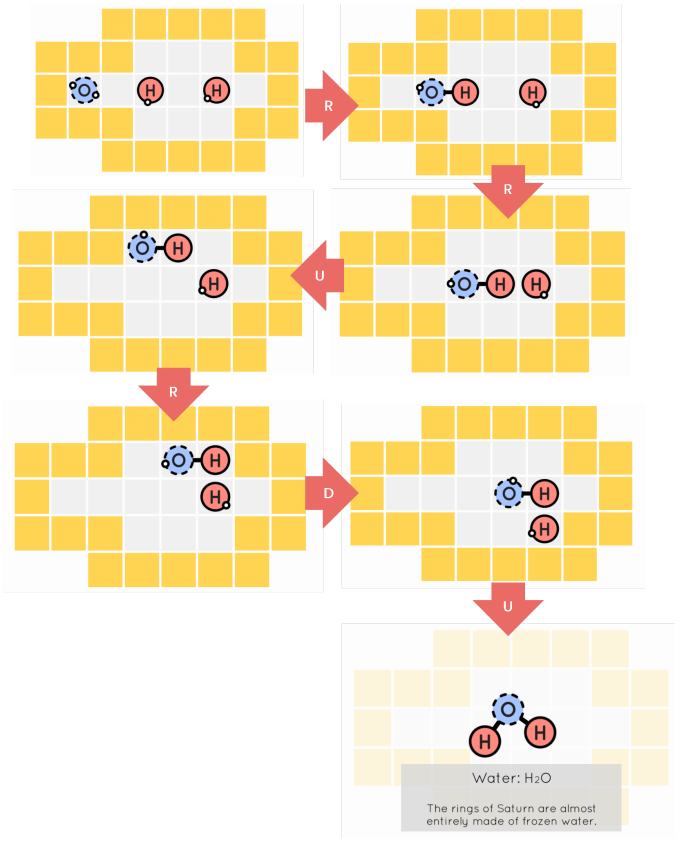


Fig. 12. Level `Push` of the sokobond solver gameplay
(Source: author documentation)

3) `Lotus` Level

Level layout :

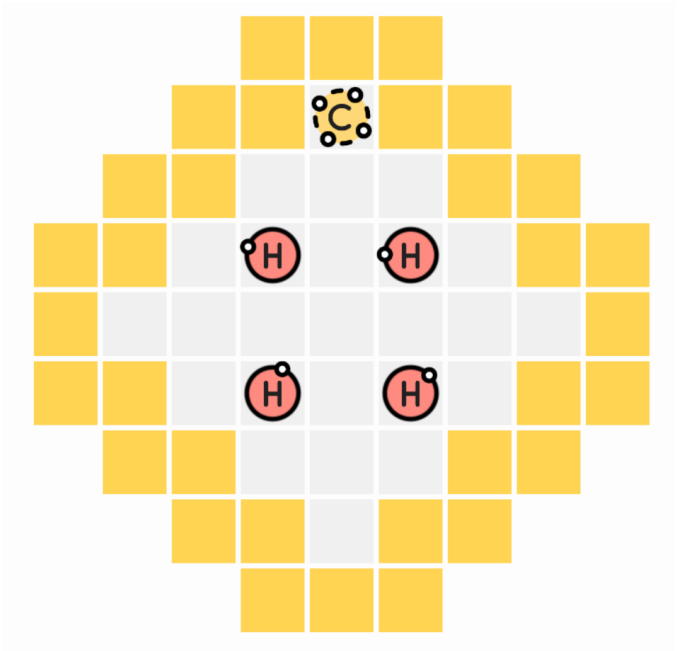


Fig. 13. Level `Lotus` of the sokobond
(Source: author documentation)

Config File:

3.txt

```
9 9 2
H 1
C 4
   ###
  ##C##
 ##   ##
## H H ##
#       #
## H H ##
 ##   ##
  ## ##
   ###
1 4
```

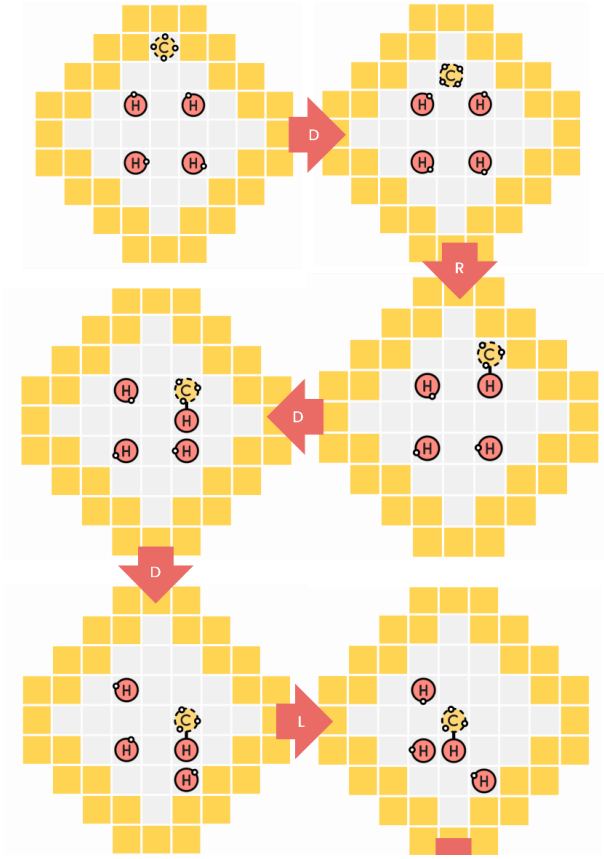
Snapshot. 14. Level 1 config file
(Source: author documentation)

Result :

```
Select the level file:
C:\Addin Munawwar\2023\5. May\sokobond-solver\test\3.txt
Searching for solutions...
[D, R, D, D, L, L, D, R, D]
Time elapsed: 1823 ms
```

Fig. 14. Level 1 of the sokobond solver output
(Source: author documentation)

Result In Game:



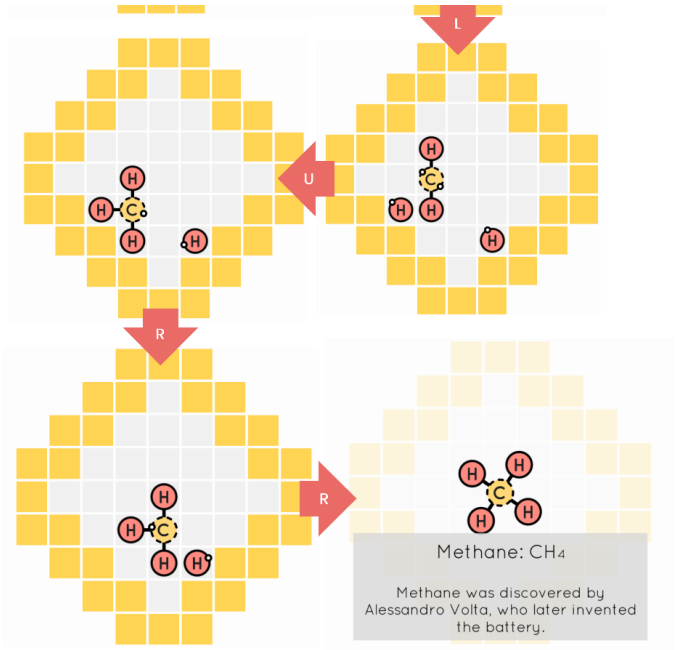


Fig. 15. Level `Lotus` of the sokobond solver gameplay
(Source: author documentation)

V. CLOSING

A. *Conclusion*

A* Algorithm is a popular route finding algorithm that efficiently could find the shortest path between two nodes and produce optimal solutions. It is applied in many fields such as route planning, puzzle solving, game AI, etc.

In the context of puzzle solving, A* algorithm can, for instance, be used to solve the Sokobond Puzzle Game with the heuristic of manhattan distance. The result shows, that A* algorithm does solve the sokobond puzzle both effectively and efficiently. It succeeds in producing the optimal path and solving the puzzle in reasonable time (0.1-1 seconds depending on complexity).

B. *Suggestion*

The current state of the solver is not yet perfect. Currently, it is only able to be solved. For further development, the current Sokobond Solver might be extended to solve more complex problems with new mechanics that are available in the later stages of Sokobond Game.

There might also be a better heuristic that can be used on this A* algorithm for sokobond solver. Experiments with different heuristics to see which works the best for the game can be considered for future development.

ACKNOWLEDGMENT

First and foremost, I would like to give a big appreciation to my teacher on algorithm design subject, Mr. Rinaldi Munir for giving this task on writing a paper about algorithm design.

I feel very grateful because this task has helped me a lot in understanding a* algorithm and even implement it in such a fun way, which is solving a puzzle game. I also want to thank my family who supported me academically and also my colleagues who helped me through a lot of hard times this semester, so I can finish this paper. Overall, I'm so grateful to be able to study in ITB that provides the necessary resources and facilities that facilitated the smooth progress of this paper.

REFERENCES

[1] Artificial intelligence: a modern approach. Norvig, Peter (4th ed.). Russell, Stuart J. (2018). Boston: Pearson.

[2] "Engineering Route Planning Algorithms". *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Lecture Notes in Computer Science. Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. (2009).

[3] Penentuan Rute (Route/Path Planning) Bagian 2 : Algoritma A* [Online].https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf. Diakses pada 21 Mei 2023.

[4] Sokobond [Online]. https://www.sokobond.com/. Diakses pada 21 Mei 2023.

[5] Sokobond Official Press Kit [Online]. https://www.sokobond.com/presskit/. Diakses pada 21 Mei 2023.

[6] A* Search Algorithm [Online]. https://www.geeksforgeeks.org/a-search-algorithm/. Diakses pada 21 Mei 2023.

[7] Sokobond: Review[Online]. https://www.destructoid.com/reviews/review-sokobond/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Mei 2023

Addin Munawwar Yusuf - 13521085