CS440-a1 Problem Solving

Dhruv Patel Neel Prabhu Saipranav Kalapala

09 February 2022

Introduction

For this project we created software in Python to showcase A* and Theta* path-finding algorithms on user-inputted grids. The grids consist of cells which are either unblocked or blocked. These cells have edges along the border of the cells and diagonals which act as paths. With a specified starting point (x_1, y_1) and a goal/end point (x_2, y_2), the user has the option to visualize either A* or Theta* algorithm on the grid.

```
README

This program runs search algorithms on a grid.

In order to run the program, change directory into the folder containing main.py, then run the
    main.py file (shell command: "python3 main.py").

The program uses tkinter to display a GUI, so an appropriate compatible software is needed to
    run this program.

At any point the user may enter Q to quit.

The input is a text file of format specified in the project description. The program first
    takes an input for a path to the grid you want to search. There are sample files in the
    folder Grids. Grid0.txt is an example of a grid with no path between the start and goal,
    whereas Grids1.txt to Grids50.txt are randomly generated grids of size 100x50 with 10%
    blocked cells and contain a path between the start and goal (they were prechecked for
    blocked paths). The user is recommended to add testing grids to this folder to keep it
    organized, making the input "[Path to Folder]/Grids/[Grid Name].txt" for the file path

The program then takes a single capital character to run a search or change files.

F: Allows the user to change the file for the grid

A: Runs A* search on the grid, then displays the resulting path on a GUI

T: Runs Theta* search on the grid, then displays the resulting path on a GUI

GUI: In order to display the h, g, and f values of verticies that have been searched, the user
    may press on the vertex itself, and the values will display at the bottom of the screen. If
     the vertex has not been searched, the text will not be updated. In order to close the GUI,
     simply click the X, and the terminal will allow for the next input once it has been closed
    .
```

Above information is also provided in a README file which is included along with the source code and the report.

Interface

The program uses a combination of command line interface (CLI) and graphical user interface (GUI) to interact with the user. Program initially uses CLI for the user to input the grid file and select the algorithm of their choosing to run on the grid and then uses a GUI to display the resulting path. Pictures of of the CLI and GUI are displayed below:

On the displayed path on the GUI, the start vertex is highlighted in green, and the end vertex is highlighted in blue, while all the intermediary vertices that exist on the resulting path are highlighted in yellow, and the path itself is red. Each highlighted vertex is also clickable and on click shows the g, h, and f values of that vertex, as computed by the algorithm.

After each run, the program loops and the user can input a new file to run. At any point the user can quit out of the program using 'Q'.

```
nvp40@ice:~/AI/Project1/CS440-a1$ python3 main.py
Enter the full path to the file, or Q to quit: Grids/Grid0.txt
No path found between the start and goal.
Enter the full path to the file, or Q to quit: Grids/Grid1.txt
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: A
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: T
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: F
Enter the full path to the file, or Q to quit: Grids/Grid10.txt
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: A
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: T
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: F
Enter the full path to the file, or Q to quit: Grids/Grid35.txt
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: A
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: T
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: F
Enter the full path to the file, or Q to quit: FakeGrid
Invalid Path.
Enter the full path to the file, or Q to quit: Grids/Grid4.txt
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: A
Enter A for A* search, T for Theta* search, F to change file, or Q to quit: Q
nvp40@ice:~/AI/Project1/CS440-a1$
```

Figure 1: Command Line Interface (CLI)

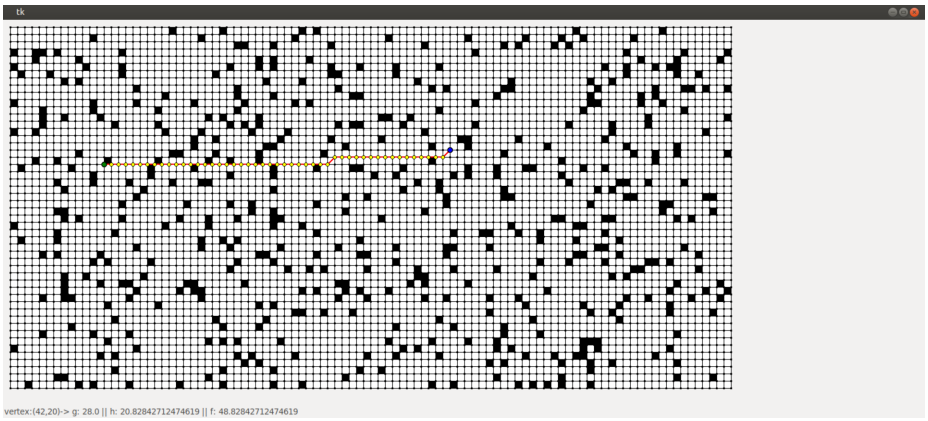

Figure 2: Graphical User Interface (GUI)

A* Manual Computation

$h(s) = \sqrt{2} \cdot min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) + max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) - min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|)$

Start with A4:

- A4
  - g = 0
  - h = $2\sqrt{2} + 1 \approx 3.83$
  - f = $2\sqrt{2} + 1 \approx 3.83$

Expand A4. Remove A4 from the fringe and add it to the closed list. Add A3, B3, B4 to the fringe:

- A3
  - g = 1
  - h = $2\sqrt{2} \approx 2.83$
  - f = $2\sqrt{2} + 1 = 3.83$
- B3
  - g = $\sqrt{2} \approx 2.83$
  - h = $\sqrt{2} + 1 \approx 2.41$

- f = $2\sqrt{2} + 1 = 3.83$

- B4
  - g = 1
  - h = $\sqrt{2} + 2 \approx 3.41$
  - f = $\sqrt{2} + 3 \approx 4.41$

Expand A3. Remove A3 from the fringe and add it to the closed list. Add A2, B2 to the fringe.

- A2
  - g = 2
  - h = $\sqrt{2} + 1 \approx 2.41$
  - f = $\sqrt{2} + 3 = 4.41$

- B2
  - g = $\sqrt{2} + 1 \approx 2.41$
  - h = $\sqrt{2} \approx 1.41$
  - f = $2\sqrt{2} + 1 = 3.83$

Expand B3. Remove B3 from the fringe and add it to the closed list. Add C2, C3 to the fringe:

- C2
  - g = $2\sqrt{2} \approx 2.83$
  - h = 1
  - f = $2\sqrt{2} + 1 = 3.83$

- C3
  - g = $\sqrt{2} + 1 \approx 2.41$
  - h = 2
  - f = $\sqrt{2} + 3 = 4.41$

Expand B2. Remove B2 from the fringe and add it to the closed list. Add A1, B1, C1 to the fringe:

- A1
  - g = $2\sqrt{2} + 1 \approx 3.83$
  - h = 2
  - f = $2\sqrt{2} + 3 = 5.83$

- B1
  - g = $\sqrt{2} + 2 \approx 3.41$
  - h = 1
  - f = $\sqrt{2} + 3 = 4.41$

- C1 = Goal; Path Found
  - g = $2\sqrt{2} + 1 \approx 3.83$
  - h = 0
  - f = $2\sqrt{2} + 1 \approx 3.83$

C1 is the goal, so A* is completed. One shortest path is A4, A3, B2, C1 of cost $2\sqrt{2}+1 \approx 3.83$.

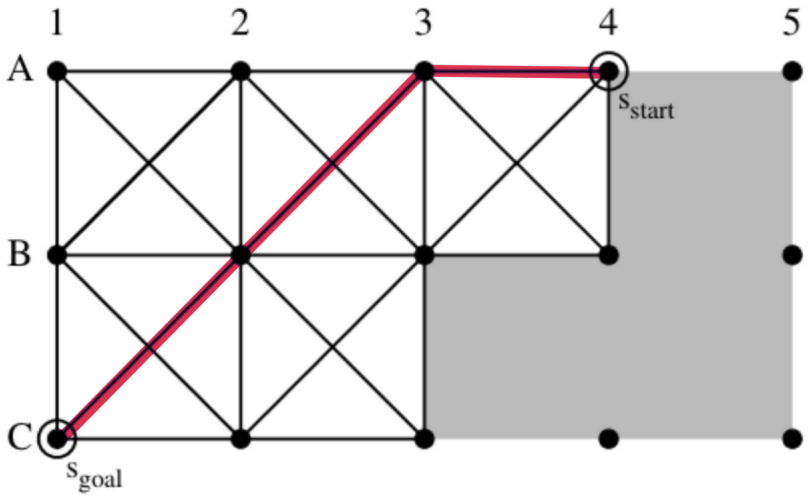

Figure 3: Path found by A*

Theta* Manual Computation

$$h(s) = c(s, s') \qquad c(s, s') \textit{ is the straight line distance between } s \textit{ and } s'$$

Start with A4. Remove A4 from the fringe and add it to the closed list. :

- A4
  - g = 0
  - h = $\sqrt{13} \approx 3.61$
  - f = $\sqrt{13} \approx 3.61$

Expand A4. Remove A4 from the fringe and add it to the closed list. Add A3, B3, B4 to the fringe:

- A3
  - g = 1
  - h = $2\sqrt{2} \approx 2.83$
  - f = $2\sqrt{2} + 1 = 3.83$
- B3
  - g = $\sqrt{2} \approx 1.41$
  - h = $\sqrt{5} \approx 2.24$
  - f = $\sqrt{2} + \sqrt{5} = 3.65$
- B4
  - g = 1
  - h = $\sqrt{10} \approx 3.16$
  - f = $1 + \sqrt{10} \approx 4.16$

Expand B3. Remove B3 from the fringe and add it to the closed list. Add A2, B2, C2, C3 to the fringe:

- A2
  - g = 2
  - h = $\sqrt{5} \approx 2.24$
  - f = $2 + \sqrt{5} = 4.24$
- B2
  - g = $\sqrt{5} \approx 2.24$
  - h = $\sqrt{2} \approx 1.41$
  - f = $\sqrt{5} + \sqrt{2} = 3.65$
- C2
  - g = $2\sqrt{2} \approx 2.83$
  - h = 1
  - f = $2\sqrt{2} + 1 \approx 3.83$
- C3
  - g = $\sqrt{2} + 1 \approx 2.41$
  - h = 2
  - f = $\sqrt{2} + 3 \approx 4.41$

Expand B2. Remove B2 from the fringe and add it to the closed list. Add A1, B1, C1 to the fringe:

- A1
  - g = 3
  - h = 2
  - f = 5
- B1
  - g = $\sqrt{10} \approx 3.16$
  - h = 1
  - f = $\sqrt{10} + 1 \approx 4.16$
- C1 = Goal; Path Found
  - g = $\sqrt{13} \approx 3.61$
  - h = 0
  - f = $2\sqrt{13} \approx 3.61$

C1 is the goal, so Theta* is completed. One shortest path is A4, C1 of cost $\sqrt{13} \approx 3.61$.

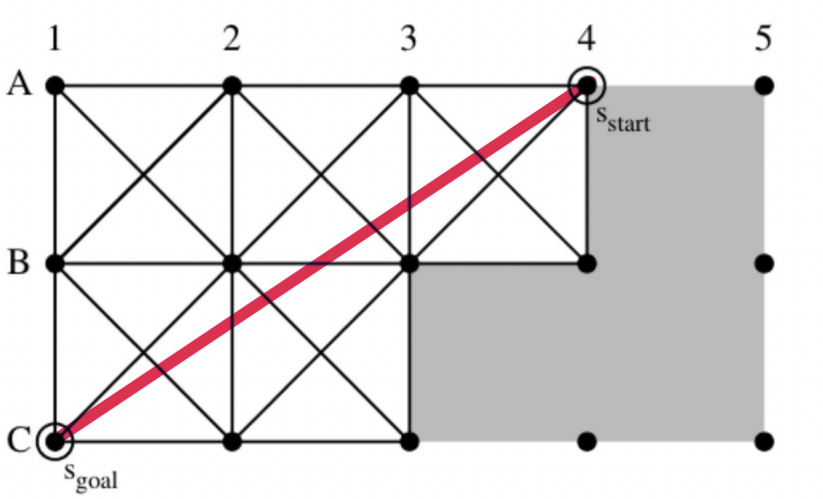

Figure 4: Path found by Theta*

A* and Theta* Implementation

In our program, A* and Theta* uses the same basic framework. To implement A* and Theta* we used some common data structures to store and represent data.

- values: hashmap to store the h and g values in a tuple with the vertex as key
- parents: hashmap to store the parent of vertices being explored
- visited: list to store the visited vertices
- fringe: custom implementation of a binary heap for the fringe

We first built a 2D matrix that contained the information of the grid cells, with each index in the grid representing a cell, and its value is binary (0 or 1) with 1 representing a blocked cell and 0 representing an unblocked cell.

Then based on the cell matrix, we build an adjacency list of all the vertices and their neighbors. We used a hashmap with the vertex (x,y) as key and a list of vertices as key.

0.0.1 A* Algorithm

The neighbors hashmap along with the start and end vertex is passed onto the a* algorithm which then performs the search and returns True or False based on if path is found or not, and along with that it returns the values hashmap and the parents hashmap, which is then passed on to the GUI handler which prepares the grid and the path and the vertices and displays it. The actual algorithm for A* was adapted from the pseudocode provided in the writeup.

Some example outputs of A* are given below:

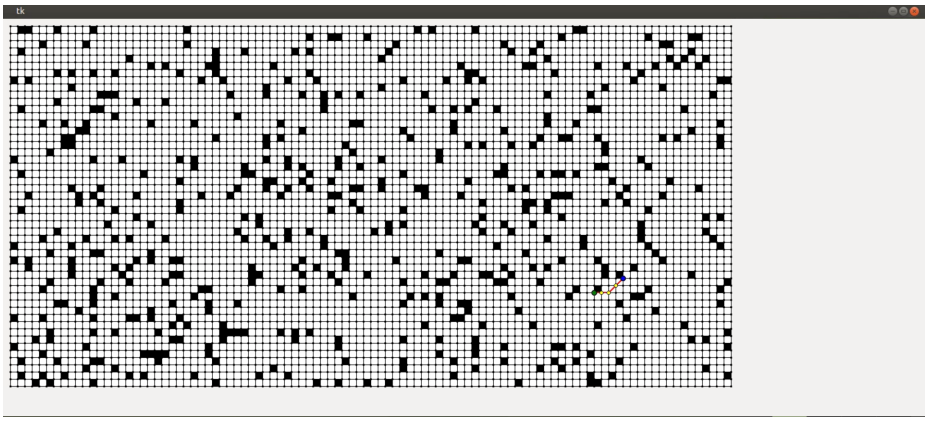

Figure 5: A* Example 1

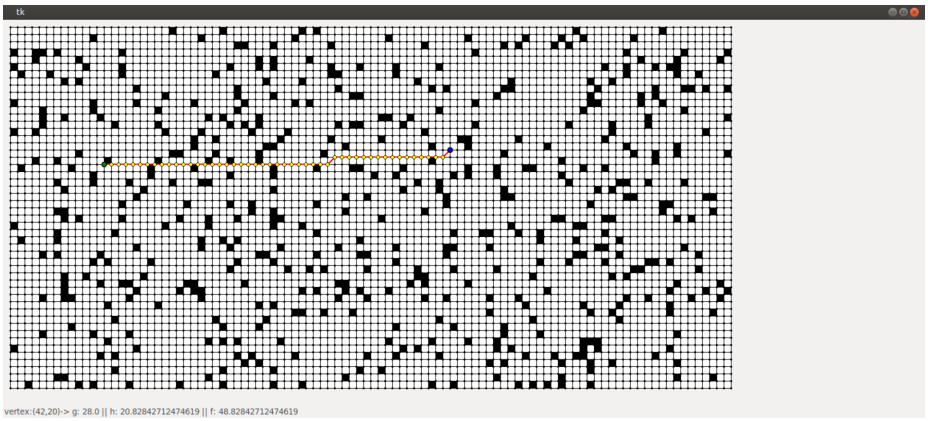

Figure 6: A* Example 2

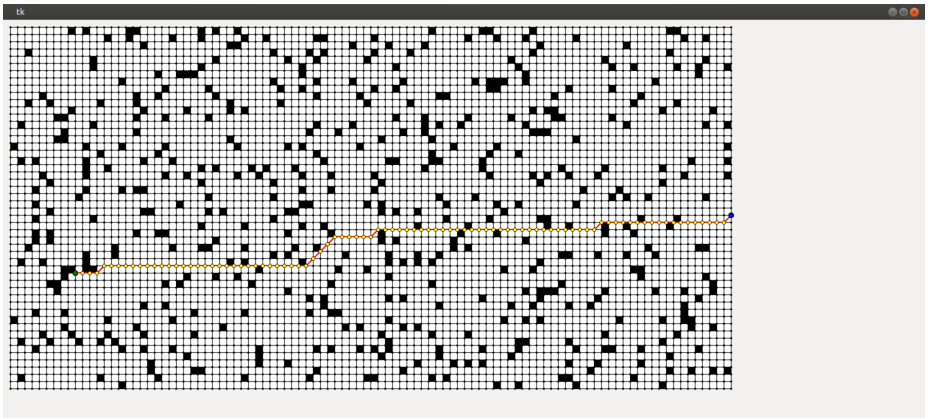

Figure 7: A* Example 3

0.0.2 Theta* Algorithm

Theta* differs from A* in the implementation, you also need the cell matrix to determine line of sight between two vertices. It also uses a different heuristic than A*. The neighbors hashmap, start and end vertices, and

the cell matrix is passed on to the algorithm. For line of sight, the code was adopted from the pseudocode and the cell matrix was designed so that it would work as is with the line of sight algorithm. Similar to A*, Theta* return True or False base on whether a path was found, and also return the parents hashmap and the values hashmap which are then passed on to the GUI handler to be displayed.

Some example outputs of T* are given below:

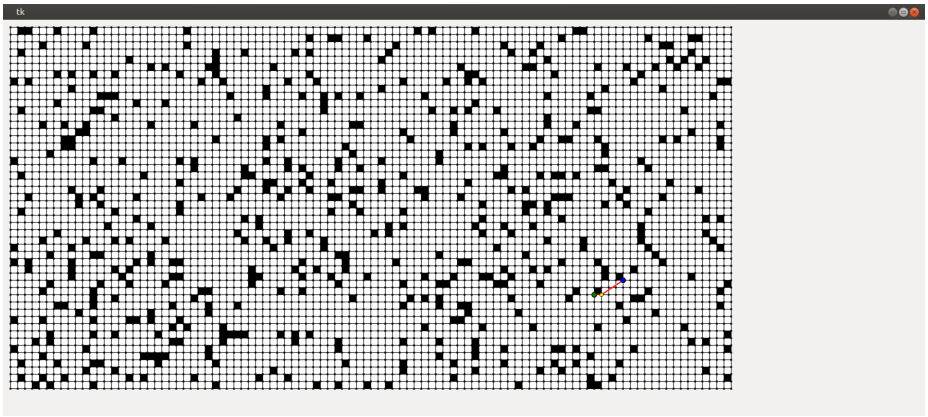

Figure 8: T* Example 1

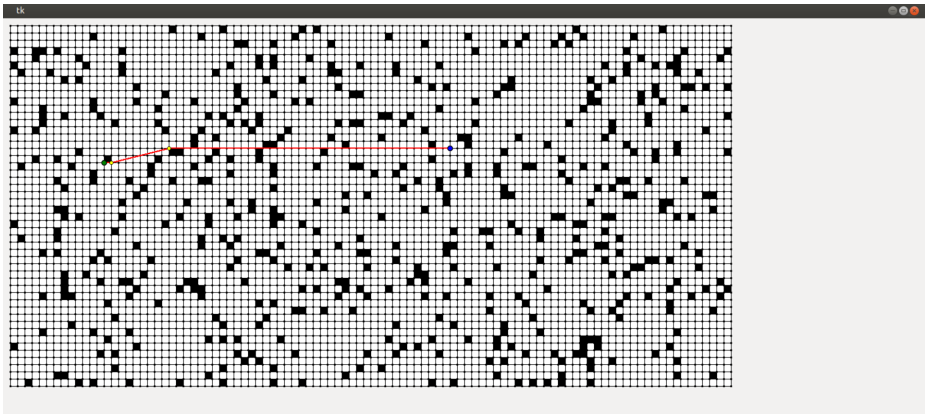

Figure 9: T* Example 2

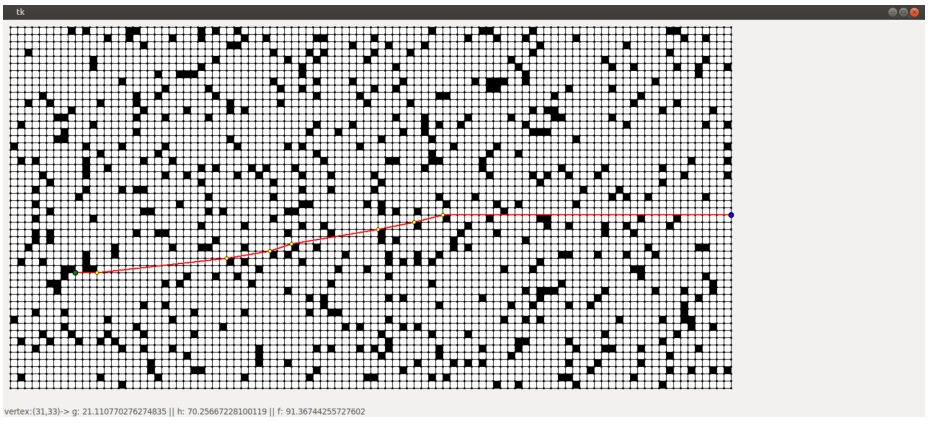

Figure 10: T* Example 3

A* Proof

A heuristic function is said to be consistent if for all vertices s, s' and actions a from s to s':

$$h(s) \leq c(s, a, s') + h(s')$$

where c(s, a, s') is the step cost of going from s to s' using action a.

The h(s) we use for A* is:

$$h(s) = \sqrt{2} \cdot min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) + max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) - min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|)$$

In other words, the heuristic is the ideal distance (using only directly adjacent or diagonal vertices) between the current vertex and the goal, assuming no blocked cells.

Because the algorithm may only move along a path that connects vertices directly adjacent (of cost 1) or diagonal (of cost $\sqrt{2}$) to the vertex s:

$$c(s, a, s') = \sqrt{2} * d + 1 * j$$

where d is the number of diagonal movements in a and j is the number of adjacent movements in a.

Assume h(s) > c(s, a, s') + h(s') for some s, a, and s'. Then, the ideal distance using only adjacent and diagonal movements between s and the goal is greater than the sum of (the cost of traveling between s and s') and (the ideal distance using only adjacent and diagonal movements between s' and the goal). Take the path from s to s' (call it P). This has cost c(s, a, s'). Then add to P the ideal path between s' and the goal. This has cost h(s'). This new path is an ideal path using only adjacent and diagonal movements between s and the goal of cost c(s, a, s') + h(s'). Then, h(s) is not the ideal distance between s and the goal, as there is an ideal path with smaller cost. Contradiction.

$$Therefore, h(s) \leq c(s, a, s') + h(s')$$

By the theorem proven in class, if the heuristic function is consistent, then A* is optimal. Therefore, A* is optimal.

The branching factor of this graph is of degree ≤ 8, as the number of vertices directly adjacent or diagonal to a vertex is at most 8. All edge costs of this graph are either 1 or $\sqrt{2}$ as described before, so they exceed some finite cost. Therefore, A* is complete.

Because A* is optimal and complete, we can conclude that A* will always find the shortest path on these grids.

Custom Binary Heap Implementation

For the binary heap implementation, we used a standard python list (array) and built methods to implement heap operations. The following methods were implemented for heap operations:

- ***insert***: key is inserted at the end of the list, and then it is compared with its parent and swapped if the value (g + h) is less than its parents'. This is looped until the item is placed in the heap accordingly.
- ***remove***: key is swapped with the the last item in the list, and then the key is deleted and then the list is heapified from the 0th index to maintain heap property.
- ***pop***: pop removes the first item in the heap, i.e. the key with the lowest value, and then calls heapify to maintain heap property.
- ***min_heapify***: this method maintains the heap property. Assumes the sub-trees are heapified. Compares the noew with its children and swaps if their values are lower that the parent node. Then recursively calls min_heapify on the smallest index.