Bi-Directional A*

Group 21
Haoyang Zhang
Dhruvin Patel
Deep Pandya

***Abstract*—**This project looks at the implementation of bidirectional A*. The project will visualize Bidirectional A* and the advantages of it over the unidirectional A*. The algorithm will be visualized in a clear manner which will be easy to understand. The Bi-Directional A* algorithm is relatively lesser known as compared to other search algorithms such as unidirectional A*, Dijkstra's, Iterative Deepening A* algorithms.

I. Project Description

This project aims at the visualization of the Bi-Directional A* Algorithm. This is to be achieved through an executable file in which the user will draw a graph arbitrarily using nodes and edges. The graph will contain a start node "S" and a goal node "G". For each node, the user will input the heuristic value for that particular node to "S" and "G". Similarly, for each edge the user will input the weight for the respective edge. Implementation of this algorithm lets you know the shortest path from start node to the goal node. This idea can be extended for finding shortest paths on maps, the closeness of the relationship between two persons on social networks, etc. The project is also useful to show why in some cases the Bi-Directional A* Algorithm will fail to return shortest path between "S" and "G". The successful completion of project is feasible in the given time frame as proposed along with a possibility of the addition of another bi-directional search in the form of Bi-Directional Dijkstra's algorithm as well as unidirectional A* and Dijkstra's algorithm. The Bi-Directional A* algorithm is relatively lesser known as compared to other search algorithms such as unidirectional A*, Dijkstra's, Iterative Deepening A* algorithms. The roadblocks for this project are:

- Modification of heuristics for both paths from "S" to "G" and "G" to "S" so that we can meet our "goal" which is not the goal node "G" but the intersection of both paths.
- A* is a unidirectional search, whereas the Bi-Directional A* will simultaneously perform two searches, i.e. from "S" to "G" and "G" to "S". in some cases , it is possible that these two searches miss each other completely and fail to meet somewhere in between "S" and "G" resulting in two paths leading to twice as consumption of time and space.
- Checking if the path returned if optimal or not. If it is not optimal, then can there be an algorithm to modify the path in order to make it optimal.
- Design of an appropriate UI to interact with the user.

Improving the Bi-Directional A* Algorithm to return an optimal path, dealing with the roadblocks for the algorithm and learning to program the GUI are some of the major challenges for this project. The time allocated to learn GUI programming would help gain experience for the team on GUI programming to make a better interface for the user. Utilizing the time for brainstorming another algorithm to implement may also be useful to come up with an optimized version of Bi-Directional A* that would return an optimal path.

A. Stage1 - The Requirement Gathering Stage.

In this project, there will be a blank screen in the application on which the user will draw a graph using nodes and edges and input the edge weights as well as heuristic for each node. The application will then run the algorithm for the user showing step-by-step traversal from start node to goal node.The deliverables for this stage include the following items:

- Academics The teacher will have a sample graph saved in a .pkl and will upload this file in the application. The application will run the Bi-Directional A* on that graph. The teacher may also want to use a sample graph drawn on the application screen for educational purposes in future. For doing so, the teacher can extract a .pkl file for the respective graph from the application on the click of a button. The student will input a graph and will run the application in order to know the mechanism of the algorithm in a clear way.
  - Scenario: A typical example of a real world scenario can be taken in the form of a classroom. A teacher will show a sample graph and render the algorithm on that graph to show the students the logic behind Bi-Directional A* algorithm. The application developed for this project will demonstrate how the algorithm works. A student may ask to change the graph a little and observe the impact of change on the algorithm. This too can be accommodated in the application file.
  - Input: The graph in a pkl format or can draw a graph.
  - Output: Visualization of each step of BDA* along with the path.
- Researcher: A person who wants to compare different graph search algorithms and wants to find the optimal algorithm their work can make use of this application.
  - Scenario: The researcher inputs a graph for which they have performed other search algorithms and will compare them to the result of our application. The researcher will know each step and thus can make improvements for the research purposes.

  - Input: The graph in pkl format or can draw a graph.
  - Output: Visualization of each step of BDA* along with the path.

Timeline for the project:

- October 21st to October 27th: Complete the description of the project
- October 27th to November 2nd: Learn GUI Programming
- November 2nd to November 9th: Draft a pseudo code and prepare a flow diagram for the project
- November 9th to November 11th: Implement Bi-Directional A* Algorithm
- November 11th to November 24th: Implement the GUI
- November 24th to November 26th: Brainstorm for another algorithm to implement
- November 26th to November 29th: Implement the new algorithm
- November 29th to December 7th: Prepare the final report and presentation ppt.

Division of labor:

- Haoyang: Implementation and optimization of BDA*
- Deep: GUI and Report
- Dhruvin: GUI and report

B. Stage2 - The Design stage.

- System Flow description: The visualization of BDA* Algorithm is achieved through a GUI. The GUI opens a window in which there is a blank space provided to draw the desired graph. The user can draw vertices, edges, source and goal nodes. The user then assigns weights and heuristic values. Once the graph has been drafted, the user may click on the confirm button to indicate that the graph has been finalized. The program then checks for validity of graph. If the graph is valid, the BDA* algorithm is implemented and the user will be shown step by step advancement of the path from the source to goal. The constraints for the validity of graph are discussed later in the document. The system and algorithmic flow diagrams are shown below.
- Flow Diagram.

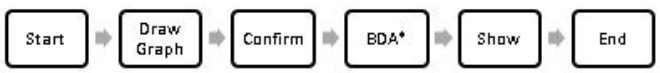

Fig. 1. System Flow

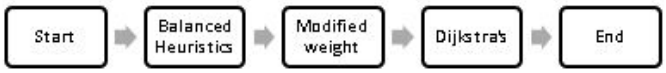

Fig. 2. Algorithm Flow

- High Level Pseudo Code System Description:

```
def system(): #the whole system loop
    while True:
        get a new message msg from
            message queue
        if msg is closeMsg:
            return
        else if msg is editPaintMsg:
            disable BDA* button
            continuousStepFlag = False
            use paintBoard function to
                process msg
            send a new message
                updateMsg
        else if msg is confirmMsg:
            use confirm function to
                check the graph
            if graph is valid:
                enable BDA* button
        else if msg is BDASMsg:
            use preprocess function on
                G to prepare for BDA*
                searching loop
            send a new message stepMsg
        else if msg is stepMsg:
            use step function to
                process 1 step of BDA*
            send a new message
                updateMsg
            if 2 searching parts meet:
                continuousStepFlag =
                    False
                send a new message
                    postprocessMsg
            else if at least 1
                searching part cannot
                continue:
                continuousStepFlag =
                    False
            else:
                continuousStepFlag =
                    True
        else if msg is updateMsg:
            use updatePaint function to
                show the graph
            if continuousStepFlag is
                True
                send a new message step
        else if msg is postprocessMsg:
            use postprocess to trace
                the path
            send a new message
                updateMsg
    return
```

```
def paintBoard(msg): #paint graph
    if msg is newV:
        check if there is already a
            vertex at the click
            position
        if there is:
            report error to users
        else:
            creat a new vertex and edit
                it
    if msg is newE:
        check if there are already 2
            vertices at 2 ends of drag
        if there are not:
            report error to users
        else:
            creat a new edge and edit
                it
    if msg is editV:
        check if there is already a
            vertex at the click
            position
        if there is not:
            report error to users
        else:
            find that vertex and edit
                it
    if msg is editE:
        check if there are already 2
            vertices at 2 ends of drag
        if there are not:
            report error to users
        else:
            find that edge and edit it
    if msg is deleteV:
        check if there is already a
            vertex at the click
            position
        if there is not:
            report error to users
        else:
            find that vertex, delete it
                and delete all edges
                of it
    if msg is deleteE:
        check if there are already 2
            vertices at 2 ends of drag
        if there are not:
            report error to users
        else:
            find that edge and delete
                it
    return
```

```
def confirm(G): #check if G is valid or
    not
    if there is not a start vertex:
        report error to users
        return False
    if there is not a goal vertex:
        report error to users
        return False
    use dijkstra's to compute the real
        distance of each vertex from
        start vertex

    if goal vertex is not reachable:
        report warn to users
        return True
    use dijkstra's on the reversed
        graph to compute the real
        distance of each vertex from
        goal vertex

    for each vertex of the graph:
        if the real distance is less
            than potential:
            report warn to users
    return True

def preprocess(G): #preprocess of BDA*
    for each vertex v of the graph:
        compute balanced potential
    for each edge of the graph:
        compute modified weight using
            balanced potential
    initialize searching
    return

def step(G): #BDA* loop
    if any priority queue is empty:
        return False
    update distance and previous vertex
        simultaneously from start
        vertex and goal vertex for one
        step

    if they meet at some vertex:
        return meeting vertex
    else:
        return None

def postprocess(G): #generate path and
    maybe preprocess for optimal path
    from meeting vertex, use previous
        vertex to trace the path to
        goal vertex and start vertex,
        and combine them
    return
```

```
def updatePaint(G):
    paint all vertives and edges of the
        graph
    highlighting the visited vertices
        and returned path
    return
```

- Algorithms and Data Structures.
  - Algorithm:
    BDA* Algorithm: The BDA* algorithm implements A* algorithm from both ends of a graph simultaneously. One A* algorithm is implemented from the start node towards goal node and the other A* algorithm is implemented from the goal node towards the start node. Each vertex is given two values of heuristic, one from start node and one from the goal node. The algorithm calculates normalized weights using the heuristic values of corresponding vertices for each edge using the formula:

$$h(u) = \frac{h_s(u) - h_g(u)}{2}$$

$$w_{new} = w_{old} - (h(u) - h(v))$$

    The algorithm calculates two normalized weights for each edge, one for forward A* and another for backward A*. Using these normalized weights, A* is applied in both directions and the path from the start node to goal node is found when these two converge to the same point in the graph.
  - Data Structures:
    1) Python Dictionary: The python dictionary stores data in a similar way as the regular dictionary. It has keys which are associated with corresponding values and these values can be referenced using the keys.
    2) Priority Queue: A priority queue is stored in form of array/list and implemented using heaps. New values are inserted at the back and removal of values is done from the front. However, the priority of the elements is always maintained at the front and the element with the lowest priority is moved to the back.
- Constraints: The visualization project does not use a database for storing values. Therefore, there are no integrity constraints applicable to the project. However, there are a few algorithmic constraints to drawing a graph as follows:
  - If there are no weights specified by the user, then the default value of the edge weight is 1.
  - If the start node or the goal node is not specified by the user, then an error is displayed prompting user to enter either or both of the start node and goal node.
  - If the path from the start node to goal node doesn't exist, then a warning is displayed prompting the user that no path exists.
  - If the heuristic is not admissive, then prompt the user warning that heuristic might not yield the best path. The user then decides whether he wants to enter heuristic values again or continue anyway.
  - Each edge must have two existing vertices as its ends. And there cannot be an edge from u to u.
  - Each vertex must be at distinct positions.
  - Each edge must have a distinct vertex pair as its identification. There cannot be two edges from u to v.

C. Stage3- The Implementation Stage

The project was made using the python 3 language and it also involved programming a GUI so that the user can develop their graphs and see how the algorithm works, step by step. To see the functioning project, one external library PyQt5 is needed. After that is installed, the project can be run by running the file GUI.py. We have included few sample graph snapshots below:

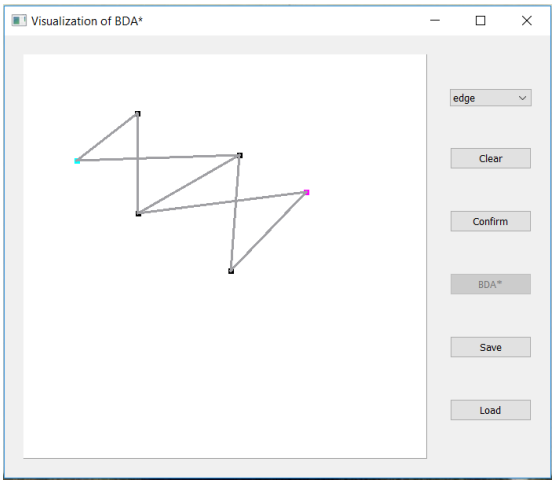

Fig. 3. Sample Graph 1

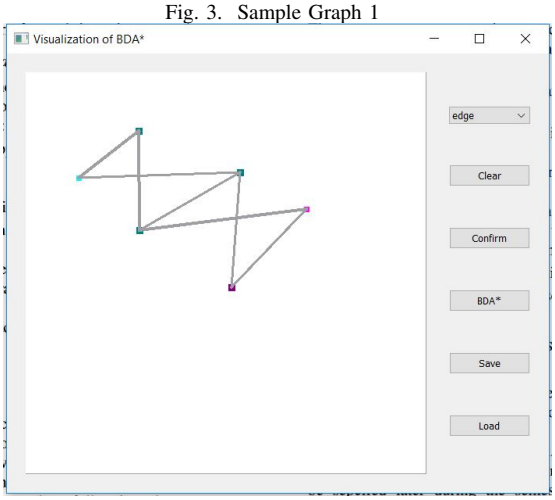

Fig. 4. Sample Graph output

- By analyzing different types of graphs we see that the algorithm does not always give you the shortest path. There may be cases in which the two path: one from source and other from the goal node don't meet till at nay node except the terminal nodes.
- The code for the project can be found in the zip file with the report.

D. Stage 4- User Interface

The user has to open the file GUI.py. It can be done in command prompt using the command GUI.py. A new window opens which has a blank canvas and a few options on the right. The user has to draw a directed graph. The user can specify what he wants to draw. If there is a path, the button BDA* button is now clickable. These are the properties of the user interface:

- The user has an option to draw of the following: Vertex, Edge, Source Vertex, Goal Vertex. An edge can be drawn by clicking a vertex and dragging th pointer to other vertex. A vertex can be drawn by just clicking on the canvas.
- Additionally, the user can edit an edge, update it's weights, delete a vertex or an edge. The graph is now made.
- If there is no path from Source to Goal, an error window pops up that there is no path. The user can also clear the graph drawn and generate a randomized graph.
- When the graph is connected from S to G, it checks for the heuristics. If it is not admissive, it prompts the user about it. However, the user can still choose to run BDA* as it is a logical error and the algorithm can still find a path.
- At each iteration, the program will prompt one step and highlight the vertexes that are connected. The nodes reached from source are magenta in color and those from the goal are green in color.
- Whenever the path is found, it prompts Done! and highlights the path.
- The following are errors shown while running the program:
  - An edge has to be from one vertex to another. It can't be random.
  - A source and goal vertex has to be specified.

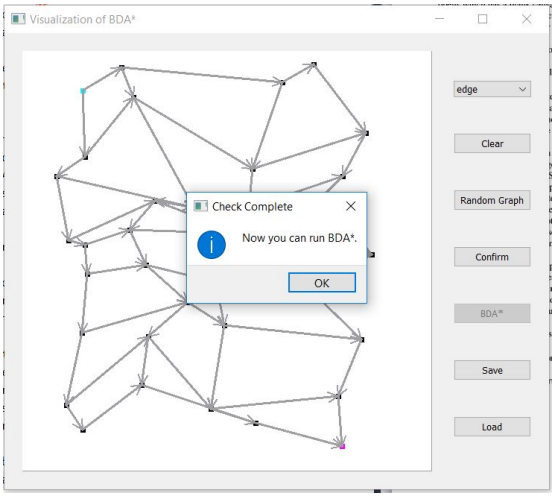

Fig. 5. BDA* UI

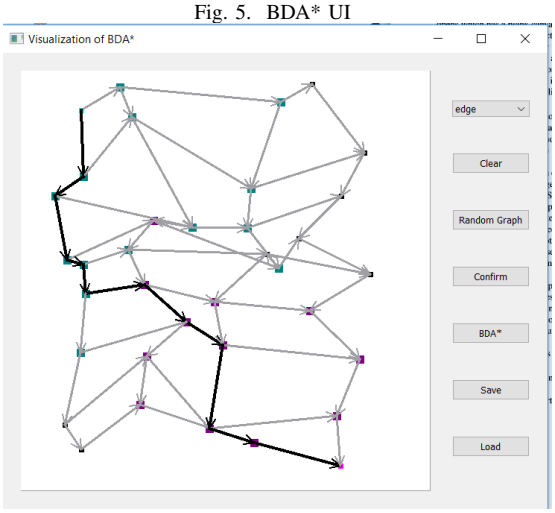

Fig. 6. Showing Path