EE 234a Project Report

Haochen Zhang

Friday 3/17/23 11:59pm

Link to Code: `https://github.com/Hzcaltech/EE234B`

1 Big Picture

In class, we have studied A* algorithm and several path-planning algorithms. As the A* algorithm can not handle the dynamic environment effectively, I want to explore an algorithm that can solve the dynamic path planning algorithm, and I choose the variant of A* algorithm which is the Lifelong Planning A* which is shorted for LPA*. I also compared the difference between A* and LPA* to see how effective LPA* is compared with A*. The project implemented LPA* algorithm in a 2D grid-based environment with changing obstacles.

2 Approach

I approached the problem by tackling the tasks into simple problems. I first implemented the basic A* algorithm with setups used in HW1 of EE133b. Then, I modified some of the setups and implemented the LPA* algorithm. I first verify the functionality of the LPA* algorithm in a static environment where the obstacle is not changing. Then I tested LPA* algorithm under the situation that some obstacles are generated during the iterations. Finally, I tested LPA* algorithm under the situation that some obstacles disappear during the iteration. Fig. 1 and Fig. 2 show an example of visualization of the disappearing of obstacles and path replanning.

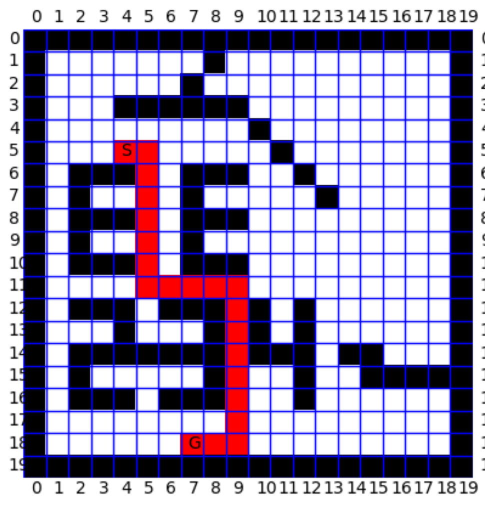

Figure 1: Path with an obstacle at (14,5).

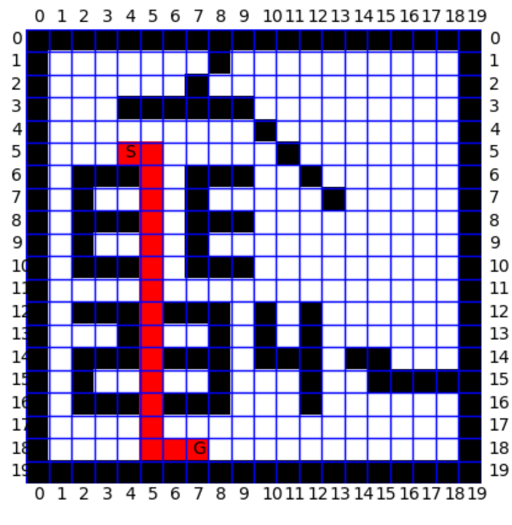

Figure 2: Path without an obstacle at (14,5).

More details on the specifics of the implementation will be discussed in the technical details section.

3 Technical Details

3.1 A* algorithm

A* algorithm is a classic path planning algorithm that performs well in a static environment where obstacles do not change frequently in other words, the requirement of replanning is insignificant. The algorithm is implemented in HW1 of EE133b in grid-based 2D space which will not be discussed in detail since the focus of the project is the LPA* algorithm. Here's the implemented code from 133b HW1:

Listing 1: A* algorithm implemented in 2D grid spave.

```
# Run the full A* algorithm.
def astar(start, goal, visual):
    # Prepare the still empty *sorted* on-deck queue.
    onDeck = []
    Processed = []
    # Setup the start state/cost to initialize the algorithm.
    start.status = State.ONDECK
    start.creach = 0.0
    start.cost   = costtogo(start, goal)
    start.parent = None
    bisect.insort(onDeck, start)

    # Continually expand/build the search tree.
    print("Starting the processing...")
    while True:
        # Show the grid.
        visual.update()

        #############
        print("onDeck")
        print(onDeck)
        print("processed")
        print(Processed)
        Nlist = start.neighbors
        pnode = onDeck.pop(0)
        for node in Nlist:
            if(node.status == State.UNKNOWN):
                node.creach = start.creach + 1
                node.status = State.ONDECK
                node.cost = costtogo(node, goal) + node.creach
                node.parent = start
                bisect.insort(onDeck, node)
            elif(node.status == State.ONDECK):
                node.creach = start.creach + 1
                node.status = State.ONDECK
                node.cost = costtogo(node, goal) + node.creach
                node.parent = start
                onDeck.sort()

        pnode.status = State.PROCESSED

        start = onDeck[0]
        bisect.insort(Processed,pnode)
        if (goal.status == State.PROCESSED):
            break
        #############

    # Show the final grid.
    print("Goal state has been processed.")
    visual.update()

    # Create the path to the goal (backwards) and show.
    print("Marking path...")
    #############
```

```
    path = []
    while goal.parent != None:
        goal.status = State.PATH
        path.append(goal)
        goal = goal.parent
    goal.status = State.PATH
    path.append(goal)
    path.reverse()
    print("path")
    print(path)
    #############
    visual.update()
    visual.update()
    return
```

3.2 Lifelong Planning A* algorithm

Lifelong Planning A* algorithm is an incremental version of the A* algorithm that can handle the dynamic environment. To be more specific, with the same starting point and end point, the LPA* algorithm does not need to recalculate the entire map to replan the path. However, one important thing to notice here is that the LPA* algorithm needs fixed starting and end points which the D* lite algorithm does not need.

Here's the pseudocode and a simple flow chart of the LPA* algorithm:

Algorithm 1 Lifelong Planning A* Algorithm

```
Initialization:
create an empty priority queue U
set the rhs value of all nodes be inf
set the rhs value of start nodes be 0
insert start node into U
while True do
    ComputeShortestPath():
    if map change happened then
        Update cost for the node that changed status
        Call UpdateVertex() for all nodes that are affected by the cost change
    end if
end while

ComputeShortestPath:
while calculateKey(U[0]) < calculateKey(goal) or goal.rhs != goal.g do
    node = U.pop(0)
    if node.g > node.rhs then
        node.g = node.rhs
    else
        node.g = inf
        UpdateVertex(node)
    end if
    UpdateVertex(node.neighbor)
end while

UpdateVertex(node):
if node != start then
    node.rhs = min(neighbor.g + heuristic(node,neighbor) for neighbor in node.neighbors)
end if
if node is in U then
    U.remove(node)
end if
if node.g != node.rhs then
    U.insert(node) based on calculateKey(node)
end if

calculateKey(node):
return [min(node.g,node.rhs)+heuristic(node,goal),min(node.g,node.rhs)]
```

In the algorithm, the rhs value is the one-step look-ahead value. And the g value is the estimated distance traveled. Combining rhs value and g value, the algorithm is able to better determine which node to expand. The value that directly decide the priority of nodes are the return value of calculateKey which has the form [k1,k2]. k1 will be used to decide priority first, as the k1 values of different nodes are equal, k2 values will be used to decide the priority.

Listing 2: Class Implementation. The less than is written for sorting priority list. The report function is used for debugging and verifying. The distance function uses manhattan distance as the heuristic function. For obstacles

```python
class State:
    # Possible status of each state.
    WALL      = -1      # Not a legal state - just to indicate the wall
    UNKNOWN   =  0      # "Air"
    ONDECK    =  1      # "Leaf"
    PROCESSED =  2      # "Trunk"
```

```
    PATH      =  3      # Processed and later marked as on path to goal

    STATUSSTRING = {WALL:      'WALL',
                    UNKNOWN:   'UNKNOWN',
                    ONDECK:    'ONDECK',
                    PROCESSED: 'PROCESSED',
                    PATH:      'PATH'}

    STATUSCOLOR = {WALL:      np.array([0.0, 0.0, 0.0]),   # Black
                   UNKNOWN:   np.array([1.0, 1.0, 1.0]),   # White
                   ONDECK:    np.array([0.0, 1.0, 0.0]),   # Green
                   PROCESSED: np.array([0.0, 0.0, 1.0]),   # Blue
                   PATH:      np.array([1.0, 0.0, 0.0])}   # Red

    # Initialization
    def __init__(self, row, col):
        # Save the location.
        self.row = row
        self.col = col

        # Clear the status and costs.
        self.status = State.UNKNOWN
        self.creach = 0.0       # Actual cost to reach
        self.cost   = 0.0       # Estimated total path cost (to sort)
        self.rhs = np.inf
        self.g = np.inf
        # Clear the references.
        self.parent    = None
        self.neighbors = []
        self.goal = self

    # Define less-than, so we can sort the states by cost.
    def __lt__(self, other):
        if min(self.g,self.rhs) +self.distance(self.goal) < min(other.g,other.rhs) ...
            +other.distance(other.goal):
            return True
        elif min(self.g,self.rhs) +self.distance(self.goal) == min(other.g,other.rhs) ...
            +other.distance(other.goal):
            if min(self.g,self.rhs) < min(other.g,other.rhs):
                return True
        else:
            return False
        #return self.cost < other.cost

    # Define the Manhattan distance.
    def distance(self, other):
        if self.status != State.WALL and other.status != State.WALL:
            return abs(self.row - other.row) + abs(self.col - other.col)
        else:
            return np.inf


    # Return the color matching the status.
    def color(self):
        return State.STATUSCOLOR[self.status]

    # Return the representation.
    def __repr__(self):
        return ("<State %d,%d = %s, g %f,rhs %f,k1 %f,k2 %f>\n" %
                (self.row, self.col,
                 State.STATUSSTRING[self.status],self.g,self.rhs, min(self.g,self.rhs) ...
                     +self.distance(self.goal),min(self.g,self.rhs)))
```

Listing 3: CalculateKey. Calculating Key base on g value and rhs value and heuristic function.

```
def calculateKey(node,goal):
```

```
    return [min(node.g,node.rhs) +node.distance(goal) ,min(node.g,node.rhs)]
```

Listing 4: UpdateVertex. This function updates the rhs value of node and update the priority queue

```
def updateVertex(start,node,U):
    print("updating Vertex")
    print(node)
    if node != start:
        for neighbor in node.neighbors:
            tmp = node.rhs
            #print(neighbor)
            #print(neighbor.distance(node))
            node.rhs = min(node.rhs, neighbor.g + neighbor.distance(node))
            #print(node.g)
            #print(node.rhs)
            #print(tmp)
            if tmp != node.rhs:
                if(neighbor.parent != node):
                    print("change connection from")
                    print(node.parent)
                    print("to")
                    print(neighbor)
                    node.parent = neighbor
                    #print(neighbor)

    if node in U:
        U.remove(node)
        print("remove")
        print(node)
    if node.g != node.rhs:
        bisect.insort(U, node)
        print("insert")
        print(node)
```

Listing 5: Number of particles Calculation. We calculated number of particles needed based on KL distance.

```
    def colorboxes(self):
        # Determine the colors.
        color = np.ones((self.rows,self.cols,3))
        for row in range(self.rows):
            for col in range(self.cols):
                color[row,col,:] = self.states[row][col].color()

        # Set the boxes.
        return self.ax.imshow(color, interpolation='none', aspect='equal',
                              extent=[0, self.cols, 0, self.rows], zorder=0)

    # Add some text - this won't show until the next pause().
    def write(self, row, col, text):
        plt.text(0.33 + col, self.rows - 0.67 - row, text)

    # Update, changing the box colors according to the states.
    def update(self):
        # Remove the previous boxes and replace with new colors.
        self.boxes.remove()
        self.boxes = self.colorboxes()

        # Force the figure to update.  And wait to hit enter.
        plt.pause(0.001)
        input('Hit return to continue')


#
#   State Object
```

```
#
#   The state object (one per box/element in the grid), includes the
#   status (UNKNOWN, ONDECK, PROCESSED), cost, as well as a list of
#   neighbors.
```

Listing 6: Adding obstacle to the grid space. The obstacle statues change will induce change in the g value and rhs value of node

```
def updateObstacle(updateList,row,col,states):
    states[row][col].status = State.WALL
    states[row][col].rhs = np.inf
    states[row][col].g = np.inf
    for node in findDirectChild(states, states[row][col]):
        updateList.append(node)
    return updateList
```

Listing 7: Removing obstacle from the grid space. The obstacle statues change will induce change in the g value and rhs value of node

```
def removeObstacle(updateList,row,col,states):
    states[row][col].status = State.UNKNOWN
    states[row][col].rhs = 100
    states[row][col].g = np.inf
    states[row][col].neighbors = ...
        [states[row-1][col],states[row+1][col],states[row][col-1],states[row][col+1]]
    for node in findDirectChild(states, states[row][col]):
        updateList.append(node)
    return updateList
```

Listing 8: Find the successors of node.

```
def findDirectChild(states,node):
    Dlist = []
    row = node.row
    col = node.col
    if row - 1 ≥ 0:
        if states[row-1][col].parent == states[row][col]:
            Dlist.append(states[row-1][col])
    if row + 1 < len(states):
        if states[row+1][col].parent == states[row][col]:
            Dlist.append(states[row+1][col])
    if col - 1 ≥0:
        if states[row][col-1].parent == states[row][col]:
            Dlist.append(states[row][col-1])
    if col + 1 < len(states[0]):
        if states[row][col+1].parent == states[row][col]:
            Dlist.append(states[row][col+1])
    return Dlist
```

Listing 9: Propagation of the cost change due to the adding or removing obstacles.

```
def updateCost(states,goal,row,col,U,start,change):
    print(row,col)
    #updateVertex(start,states[row][col],U)
    if change == 1:
        states[row][col].rhs=np.inf
        states[row][col].g = np.inf
    elif change == -1:
        states[row][col].rhs = 100
        states[row][col].g = np.inf

    if row - 1 ≥ 0:
```

```
        if states[row-1][col].parent == states[row][col]:
            #states[row-1][col].rhs = np.inf
            #states[row-1][col].parent = None
            updateCost(states,goal,row-1,col,U,start,change)
    if row + 1 < len(states):
        if states[row+1][col].parent == states[row][col]:
            #states[row+1][col].rhs = np.inf
            #states[row +1][col].parent = None
            updateCost(states,goal,row+1,col,U,start,change)
    if col - 1 ≥0:
        if states[row][col-1].parent == states[row][col]:
            #states[row][col-1].rhs = np.inf
            #states[row][col-1].parent = None
            updateCost(states,goal,row,col-1,U,start,change)
    if col + 1 < len(states[0]):
        if states[row][col+1].parent == states[row][col]:
            #states[row][col+1].rhs = np.inf
            #states[row][col+1].parent = None
            updateCost(states,goal,row,col+1,U,start,change)
```

4 Results

4.1 LPA* versus A* Data

The runtime of the average replanning time after the first iteration of LPA* is 0.0009649 in the case that a node in the original path is changed to an obstacle.

The runtime of the average replanning time after the first iteration of LPA* is 0.0006094 in the case an obstacle is removed from the potential path.

The runtime of the average replanning time after the first iteration of LPA* is 4.053e-06 in the case that a node not in the original path is changed to an obstacle.

The runtime of the average replanning time after the first iteration of LPA* is 7.629e-06 in the case an obstacle is removed but is not from the potential path.

The runtime of the average planning time of the A* algorithm using the same obstacles as above is 0.0002125.

4.2 Explanation

This result is actually surprising for me at the first glance since I would expect LPA* to outperform A*. However, the A* algorithm outperformed LPA* algorithm when the path is changed due to the change of obstacles. Only when the change of obstacle does not affect the path, does LPA* outperform A*. However, the runtime calculated for LPA* is all on the second iteration and the first iteration takes an average time of 0.0018689632415771484. The significant difference between the runtime of the first iteration of LPA* and A* may be caused by the implementation detail. After doing more research on LPA*, I found that the LPA* will outperform A* when the number of edges updated cost due to the change of obstacles need to be less than 1 % of the total number of edges according to [KLF04]. Due to the map setting, the number of cost updates due to one block change status from obstacle to air or from air to obstacle will affect much more than 1 % edges in my experiment. And this can also support that when the change of obstacles does not affect the edge, LPA* outperforms A* by a lot.

5 Conclusion

In this final project, I explored the A* algorithm and LPA* algorithm. I compared the runtime of the LPA* algorithm under different cases to A* algorithm. However, I didn't find a case that the path is changed but the LPA* algorithm still outperforms the A* algorithm. This probability needs a huge map and reasonable iterations to achieve.

In the future, I think if there's a chance, I will try to elaborate more on this project and try to include more cases of the LPA* algorithm to analyze it in more detail. Also, I want to include more path-planning algorithms to compare and see their advantages and disadvantages. In addition, I think it is possible to improve the implementation of the LPA* algorithm to make it perform better.

References

[KLF04] Sven Koenig, Maxim Likhachev, and David Furcy. "Lifelong Planning A". In: *Artificial Intelligence* 155.1 (2004), pp. 93–146. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2003.12.001. URL: https://www.sciencedirect.com/science/article/pii/S000437020300225X.