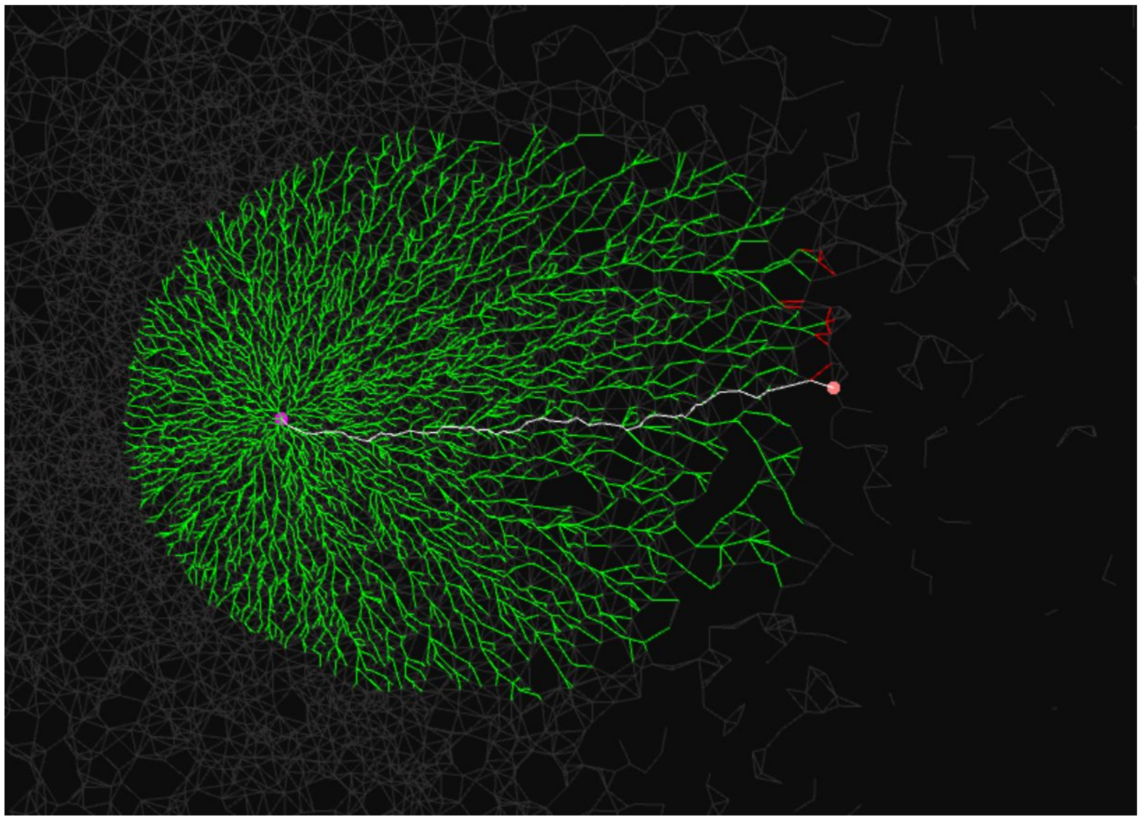
Breadth First Heuristic Search



PREPARED FOR

Assignment 3
CS6380: Artificial Intelligence Search Methods For Problem Solving

PREPARED BY

MM19B012

Pragalbh Vashishtha

REPORT SUMMARY

Breadth First Heuristic Search:

BFHS was introduced to the AI world in 2006 and was developed by **Rong Zhou** and **Eric A. Hansen**

BFHS was introduced as a way to combat the memory requirements of A*. The Breadth first search strategy is combined with an upper limit which will be referred to as "U" in this report. By imposing this limit, the algorithm escapes inspection of a large quantity of nodes, and this depends on the 'tightness' placed on by "U".

In this implementation of BFHS, Best first search is used to set "U". Beam search is generally used, but Best First places a tighter bound and improves efficiency. The nodes it inspects are not rechecked for any improvement in path, nor is any closed node or open node added back in search. This is done to prevent complication of the precursor algorithm. Also it can be noted that this version of Best First Search can be treated as a beam search with non bounded width.

Best first heuristic search limits the f-value of the nodes to "U". This f-value is a combination of the 'g-value' and the 'h-value', which are the distance from the start node to the present node and the estimated distance between the present node and the goal node respectively. This is inspired from A* as this is a combination of both a pushing force (from the start) and a pulling force (from the goal). It is also admissible and will find the best path to the goal.

Nodes in OPEN or CLOSED are not added to OPEN again. However, if a better path is found to a previous node in CLOSED, the 'g-value' of the children are also updated; while for nodes in open, when a better path is found, updating the value is sufficient

Detailed explanation of code

QUEUE is OPEN. VISITED is CLOSED. This is inspired by the bfs implementation in the Demo

1. Path Reconstruction in BFHS

BFHS may sometimes visit nodes in a path that is not optimal and without reevaluating these nodes, it will not be able to return an optimal path

```python
def setnodeval(node,g,h):# SETS VALUES FOR NODE
        self.set_node_attribute(node,'h',h)
        self.set_node_attribute(node,'g',g)
        self.set_node_attribute(node,'f',g+h)

def propogate(node):#PATH RECONSTRUCTION

        gp=self.get_node_attribute(node,'g')
        for neighbor in self.neighbors(node):
                g=self.get_node_attribute(neighbor,'g')
                gnew=gp+self.get_edge_weight(node,neighbor)
                try:
                        if gnew< g:
                        setnodeval(neighbor,gnew,h)
                        self.set_parent(neighbor,node)
                        if neighbor in visited:
                                propogate(neighbor)
                except:
                        setnodeval(neighbor,gnew,h)
                        self.set_parent(neighbor,node)
                        if neighbor in visited:
                        propogate(neighbor)
```

These function is used for path reconstruction by Propagation of the new g value

Each child of passed node, the value is compared with the old g value. If the old value exists and is lesser, the new value of g is assigned. The reason h value is also assigned will be explained soon after.

If the children of the child node is also in VISITED, propagation function is called to affect its children nodes.

Now let us get to the meat of the issue. Sometimes, when propagating new g-values, nodes that had children beyond the bound “U”, can come within the border and now have value: `gnew + h < U.` However the nodes don’t have a parent set, nor do they have any g value, and hence comparing g and gnew is not possible. This issue is not seen in the standard version of the algorithm as the values are initially set to infinity, but that isn’t the case in this implementation. So the algorithm tries to compare values, if that is not possible, an exception is raised and is handled by assigning values to the new nodes.

Now as h value is not assigned, setnodeval needs to assign the h value to these new nodes. Hence this function also sets value of heuristic to goal

The call to these functions are made in the piece of code below. Smaller font is used

```
if neighbor in queue:# neighbor in open, then check if new parent is better path
   oldg= self.get_node_attribute(neighbor,'g')
   if g< oldg:
                setnodeval(neighbor,g,h)
                self.set_parent(neighbor,node)
   elif neighbor in visited:#    has children who will be affected
                oldg= self.get_node_attribute(neighbor,'g')
                if g< oldg:
                        setnodeval(neighbor,g,h)
                        self.set_parent(neighbor,node)
                        propogate(neighbor)
                # reflect changes in g from new parent
   else:
                        # simple add node
        self.set_node_attribute(neighbor,'h',h)
        self.set_node_attribute(neighbor,'g',g)
        self.set_node_attribute(neighbor,'f',g+h)
        self.set_parent(neighbor,node)
        queue.append(neighbor)
```

2. Arranging nodes in queue in BFS

In BFS which is used as a precursor to the BFHS, we need to arrange nodes according to their heuristic to the goal aka the h value. However the queue uses the API method to retrieve open and manipulating that to include the heuristic is not wise, so queue1 is used. Now for each child, the list [node, heuristic] is stored in the nested list, queue1. Then for each element in queue1, the nodes are arranged according to the second nested element and then added to the original queue. Depending on the width, the first w elements are added, but in this implementation all are added to result in a tighter bound

```
queue1=[]
queue.append(start)
queue1.append([start,self.heuristic(start,goal)])
.
. SOME CODE HERE
.

for neighbor in self.neighbors(node):
        if neighbor not in visited and neighbor not in queue:
              queue1.append([neighbor,self.heuristic(neighbor,goal)])
              self.set_parent(neighbor,node)

queue1.sort(key = lambda x: x[1]) # sorting wrt heuristic
               queue.clear()
               #clear the queue to add according to heuristic from queue1
for i in range(len(queue1)):
   queue.append(queue1[i][0])
```

3. Setting up “U”

“U” is initially set up to be 2* heuristic of start nod to goal. If BFS finds the goal, then the cost of that path is set to be the borde (cost) aka “U”

```
cost=border=2*self.heuristic(start,goal)
.
.SOME CODE HERE
.
 if self.found_goal:
              path=self.gen_path()
               # self.show_path(path)
              cost=0
              for i in range(len(path)-1):
                     cost+=self.get_edge_weight(path[i],path[i+1])
```

The cost is calculated by adding the elements in the path. This method is also used to return path for BFHS

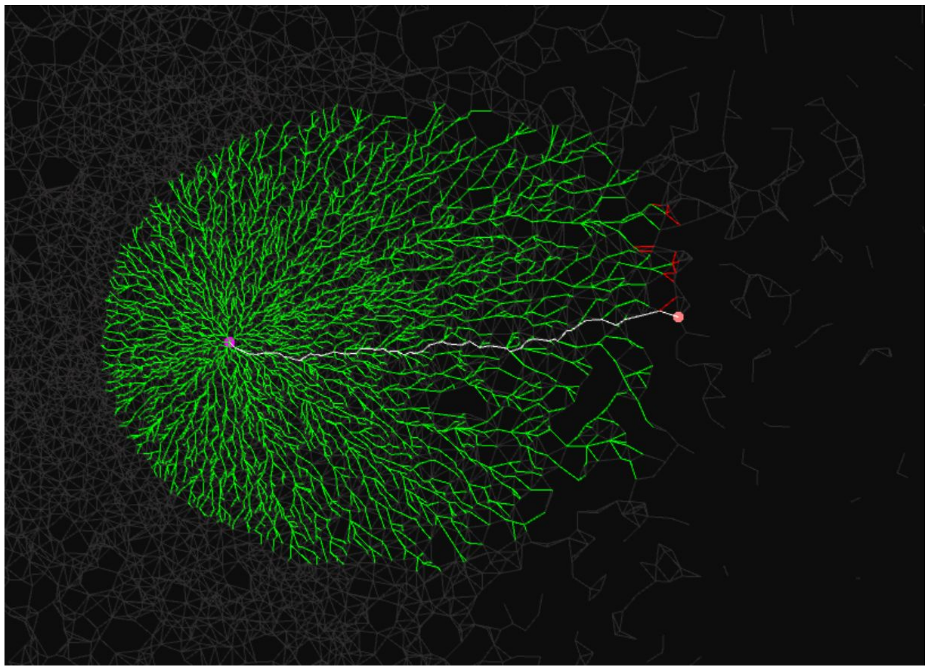
4. Outcomes:

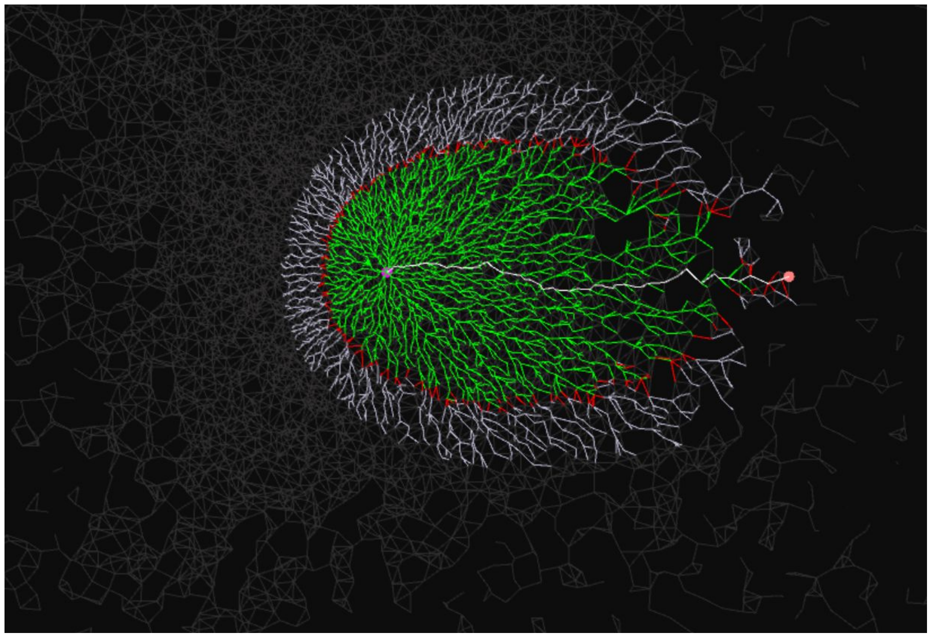
The code returns the optimal path to goal. It takes less space than A* algorithm and fewer nodes are in OPEN. Thus BFHS can be used employing less memory and still being admissible

Two images are attached below. In the second one A* is overlaid over BFHS. Although A* inspects fewer nodes, it has a much larger OPEN. The OPEN layer grows as the number of nodes increases. However in BFHS, the OPEN layer is small, and lesser memory is required to inspect the nodes.

The first image is BFHS.

The second image is BFHS and A* overlay.





Code Attached:

```
class bfhs(AlgorithmBase):
    # INITIAL INITIALISATION
    def execute(self):
        start=self.start_nodes[0]
        goal=self.goal_nodes[0]
        queue,visited=self.get_list('open'),self.get_list('closed')
#############################################################

        #DEFINING FUNCTIONS

        def setnodeval(node,g,h):# SETS VALUES FOR NODE
                self.set_node_attribute(node,'h',h)
                self.set_node_attribute(node,'g',g)
                self.set_node_attribute(node,'f',g+h)


        def propogate(node):#PATH RECONSTRUCTION

                gp=self.get_node_attribute(node,'g')
                for neighbor in self.neighbors(node):
                        g=self.get_node_attribute(neighbor,'g')
                        gnew=gp+self.get_edge_weight(node,neighbor)
                        try:
                                if gnew< g:
                                        setnodeval(neighbor,gnew,h)
                                        self.set_parent(neighbor,node)
                                        if neighbor in visited:
                                                propogate(neighbor)
                        except:
                                setnodeval(neighbor,gnew,h)
                                self.set_parent(neighbor,node)
                                if neighbor in visited:
                                        propogate(neighbor)


        # The function is used for path reconstruction by Proogation
        #For each child of node, g is compared with its previous g value.
        #If previous g value does not exist then it directly assigns value.
        #recursion is used to reach all those affected
```

```
##################################### BEST FIRST
SEARCH################################################################################


        cost=border=2*self.heuristic(start,goal)#Assign initial value for U and
cost of best first search path
        queue1=[]
        queue.append(start)
        queue1.append([start,self.heuristic(start,goal)])
#       queue1 is used that is sorted wrt the heuristic and then added to queue
#       sort wrt heuristic only

        while queue:
                self.alg_iteration_start()
                node= queue.pop(0)
                rrrr=queue1.pop(0)
                visited.append(node)

                # GOAL CHECK
                if node==goal:
                        visited.append(node)
                        self.found_goal=True
                        break

                for neighbor in self.neighbors(node):
#               children check
                        if neighbor not in visited and neighbor not in queue:
#                       Assign parent and add [neighbor, heuristic] to queu1

queue1.append([neighbor,self.heuristic(neighbor,goal)])
                                self.set_parent(neighbor,node)

                queue1.sort(key = lambda x: x[1]) # sorting wrt heuristic
                queue.clear()
                #clear the queue to add according to heuristic from queue1
                for i in range(len(queue1)):
                        queue.append(queue1[i][0])

        if self.found_goal:
                #if goal is found assign path cost to border [aka U, upper Bound]
                path=self.gen_path()
                # self.show_path(path)
                cost=0
                for i in range(len(path)-1):
                        cost+=self.get_edge_weight(path[i],path[i+1])
                # print('cost by best first search',cost) for debugging

        else:
                self.show_info('No path available')


#################################################################################
############################################################
        #Clearing and prep for BFHS
        #Again setting heuristic values just in case.
        queue.clear()
        visited.clear()
        queue.append(start)
        border=cost
        self.set_node_attribute(start,'h',self.heuristic(start,goal))
        self.set_node_attribute(start,'g',0)
        self.set_node_attribute(start,'f',self.heuristic(start,goal))
###################################### BREADTH FIRST HEURISTIC
SEARCH#############################################################

        while queue:
                self.alg_iteration_start()
                node= queue.pop(0)
                visited.append(node)

#               CHECK FOR GOAL
                if node==goal:
                        visited.append(node)
                        self.found_goal=True
                        break
                for neighbor in self.neighbors(node):
#               ADDING CHILDREN FOR EACH NODE
                        h=g=oldg=0
                        h=self.heuristic(neighbor,goal)
                        g=self.get_node_attribute(node,'g')
                        g=g+self.get_edge_weight(node,neighbor)
                        if g+h <= border:
                                # ONLY IF h+g LESS THAN U WILL BE ADDED
                                if neighbor in queue:# neighbor in open, then check
if new parent is better path
                                        oldg= self.get_node_attribute(neighbor,'g')
                                        if g< oldg:
                                                setnodeval(neighbor,g,h)
                                                self.set_parent(neighbor,node)
                                elif neighbor in visited:#  has children who will be
affected
                                        oldg= self.get_node_attribute(neighbor,'g')
                                        if g< oldg:
                                                setnodeval(neighbor,g,h)
                                                self.set_parent(neighbor,node)
                                                propogate(neighbor)
                                                # reflect changes in g from new parent
                                else:
                                        # simple add node
                                        self.set_node_attribute(neighbor,'h',h)
                                        self.set_node_attribute(neighbor,'g',g)
                                        self.set_node_attribute(neighbor,'f',g+h)
                                        self.set_parent(neighbor,node)
                                        queue.append(neighbor)




                self.alg_iteration_end()





        if self.found_goal:
                # if goal found highlight path
                finalpath=self.gen_path()
                self.show_path(finalpath)
                finalcost=0
                for i in range(len(finalpath)-1):
                        finalcost+=self.get_edge_weight(finalpath[i],finalpath[i+1])
                self.show_info('cost by BFHS:: '+str(finalcost))
                # print('heuristic', self.get_node_attribute(finalpath[0],'g'))
same as above  used for debugging

        else:
                self.show_info('No path available')
```

Thank you