Assignment 1 Written Portion

Dylan M Ang

Modified January 24, 2022

1 Question 1 Exponential Cost Heuristic

```python
def h(self, map_, start):
    # get goal
    goal = map_.goal
    # get distances
    y_dist = abs(goal.y - start.y)
    x_dist = abs(goal.x - start.x)
    # get height difference
    goal_height = map_.getTile(goal.x, goal.y)
    start_height = map_.getTile(start.x, start.y)
    delta_h = goal_height - start_height
    # min steps
    s = min(x_dist, y_dist) + abs(y_dist - x_dist)
    if goal_height > start_height: # going up
        # h = l * s
        # 2^l * h/l = total cost
        m_diff = 1/math.log(2)
        m_cost = 2 ** m_diff
        return (delta_h / m_diff) * m_cost
    elif goal_height < start_height: # going down
        return s * 2**((delta_h)/s) if s != 0 else 0
    else: # same level
        return s
```

There are three cases,

1.1 The goal state is lower than the start.

- $h = \begin{cases} s * 2^{\Delta_h / s} & s \neq 0 \\ 0 & s = 0 \end{cases}$

- Where s is the minimum number of steps to reach the goal state and Δ_h is the difference in height of the goal state and the start state.
- The ideal path to any lower goal state is one that is consistenly decreasing since each step costs between 0 and 1, but going flat or up at any time adds a cost greater than or equal to 1.
- To get a path that is always decreasing, we take the total difference in height and divide by the minimum number of steps to get the average decrease in height, Δ_h for an optimal path.
- Then, we take the exponential cost of a Δ_h change in height, and multiply by s for the total cost.
- The general idea is to return the cost of an ideal path, which will be the lowest possible path cost. It isn't possible to get a lower path cost, therefore this heuristic is admissible for this case.

1.2 The goal state is on the same level than the start.

- $h = s$ where s is the minimum number of steps required to reach the goal.
- The ideal path from a start state to a goal state at the same tile height is a flat path with no valleys or hills.
- Even if we find a path downwards (which costs less than going flat), it isn't worth it to take that path because you will eventually need to go back up and that will cost more than it saves. For example, descending one unit will cost 0.5, saving 0.5, but going back up one unit will cost an extra 1 unit.

1.3 The goal state is higher than the start.

- $h = \frac{\Delta_h}{\delta_h} * 2^{\delta_h}$ where $\delta_h = \frac{1}{ln(2)}$
- δ_h is the average amount to go up by each step and Δ_h is the total difference in height.
- This heuristic estimate uses a similar method to 1.1. The optimal path for going upwards is to go up by a consistent amount each step. However, unlike 1.1, the minimum number of steps doesn't necessarily produce the optimal path.

- $\Delta_h = \delta_h * s$.
- Total Cost $= 2^{\delta_h} * \frac{\Delta_h}{\delta_h}$ There must exist a value in which it becomes worth it to take extra steps to avoid a higher upward path.
- The number of steps in a perfect path, s, depends on the amount by which you go up each step, δ_h. But, δ_h depends on s.
- However, we want the minimum value of δ_h, which we can get by taking the derivative of the Total cost function and finding the roots.
- Doing this allows us to find that $\delta_h = \frac{1}{ln(2)}$.
- In other words, the most efficient cost path is the one which traverses up by $\frac{1}{ln(2)}$ units of height every step.
- This is admissible since it estimates the cost for an perfectly optimal path. No path which ends above the start height can follow a more efficient path, therefore the heuristic is admissible.

2 Question 1 Division Cost Heuristic

```
def h(self, map_, start):
    # get goal
    goal = map_.goal
    # get distances
    y_dist = abs(goal.y - start.y)
    x_dist = abs(goal.x - start.x)
    # min steps
    s = min(x_dist, y_dist) + abs(y_dist - x_dist)
    # get height
    start_height = map_.getTile(start.x, start.y)
    v = math.log(2, start_height)
    return max( (s - v)/2, 0 )
```

- $Cost = max(\frac{s-v}{2}, 0)$
- Where s is the Chebyshev distance and v is the number of tiles which can't have half cost.
- Imagine the two cases, we have to go up or flat to get to the goal state, or we have to go down to the goal state.

- If we have to go up or flat, the cost is no lower than 1, therefore we can count 0.5 for each of those steps and it will be an underestimate.
- If we have to go down, the cost could be less than 0.5, therefore we subtract those steps from the estimate, effectively counting 0 for each of those steps. We count 0 since the cost of downards steps simply divides by larger and larger numbers, approaching 0. However, it only goes as low as being divided by 256, so 0 is still an underestimate.
- Therefore this heuristic is admissible.

3 Question 3 Times

I have also recorded the Dijkstra implementation results to make it easier to cross reference the explored node count.

```
Dijkstra Exponential
====================
$ python Main.py -cost exp -AI Djikstra -seed 0
Time (s):  5.649763107299805
Path cost: 215.5
Nodes explored:  155417

$ python Main.py -cost exp -AI Djikstra -seed 1
Time (s):  5.400264024734497
Path cost: 219.0
Nodes explored:  146477

$ python Main.py -cost exp -AI Djikstra -seed 2
Time (s):  5.77966570854187
Path cost: 226.5
Nodes explored:  155809

$ python Main.py -cost exp -AI Djikstra -seed 3
Time (s):  8.303066968917847
Path cost: 349.5
Nodes explored:  222299

$ python Main.py -cost exp -AI Djikstra -seed 4
Time (s):  5.982449293136597
```

```
Path cost: 249.5
Nodes explored:  160998

$ python Main.py -cost exp -AI Djikstra -seed 5
Time (s):  7.308098077774048
Path cost: 260.5
Nodes explored:  195514
```

A* Exponential

```
$ python Main.py -cost exp -AI AStarExp -seed 0
Time (s):  0.7449138164520264
Path cost: 215.5
Nodes explored:  10577

$ python Main.py -cost exp -AI AStarExp -seed 1
Time (s):  2.1006457805633545
Path cost: 219.0
Nodes explored:  10997

$ python Main.py -cost exp -AI AStarExp -seed 2
Time (s):  1.3788530826568604
Path cost: 226.5
Nodes explored:  12508

$ python Main.py -cost exp -AI AStarExp -seed 3
Time (s):  6.387869358062744
Path cost: 349.5
Nodes explored:  127963

$ python Main.py -cost exp -AI AStarExp -seed 4
Time (s):  1.9555137157440186
Path cost: 249.5
Nodes explored:  20790

$ python Main.py -cost exp -AI AStarExp -seed 5
Time (s):  2.2884678840637207
Path cost: 260.5
Nodes explored:  28340
```

Dijkstra Division

```
======================
$ python Main.py -cost div -AI Djikstra -seed 0
Time (s):  5.707179069519043
Path cost: 197.0832924952719
Nodes explored:  160032

$ python Main.py -cost div -AI Djikstra -seed 1
Time (s):  5.694881916046143
Path cost: 197.16506331383655
Nodes explored:  160156

$ python Main.py -cost div -AI Djikstra -seed 2
Time (s):  5.697736978530884
Path cost: 197.00834323943513
Nodes explored:  159806

$ python Main.py -cost div -AI Djikstra -seed 3
Time (s):  5.67550802230835
Path cost: 198.13126781818744
Nodes explored:  161046

$ python Main.py -cost div -AI Djikstra -seed 4
Time (s):  5.834923028945923
Path cost: 197.24833470274933
Nodes explored:  160025

$ python Main.py -cost div -AI Djikstra -seed 5
Time (s):  5.90990686416626
Path cost: 197.44394694763412
Nodes explored:  160795

A* Division
======================
$ python Main.py -cost div -AI AStarDiv -seed 0
Time (s):  2.1589503288269043
Path cost: 197.0832924952719
Nodes explored:  35837

$ python Main.py -cost div -AI AStarDiv -seed 1
Time (s):  2.3007118701934814
Path cost: 197.16506331383655
```

```
Nodes explored:  35881

$ python Main.py -cost div -AI AStarDiv -seed 2
Time (s):  2.3018319606781006
Path cost: 197.00834323943513
Nodes explored:  35888

$ python Main.py -cost div -AI AStarDiv -seed 3
Time (s):  2.1023271083831787
Path cost: 198.13126781818744
Nodes explored:  36333

$ python Main.py -cost div -AI AStarDiv -seed 4
Time (s):  2.4772160053253174
Path cost: 197.24833470274933
Nodes explored:  35900

$ python Main.py -cost div -AI AStarDiv -seed 5
Time (s):  2.647191047668457
Path cost: 197.44394694763412
Nodes explored:  36075
```

4 Question 4 Changes

To speed up times for the Mount Saint Helens search, I implement a bidirectional A* search. I basically just keep 2 copies of the cost, prev, and explored dicts. I use the python zip() function to allow iterating over both lists at the same time. Instead of breaking when v is the end point, I break when we reach the middle point. In order to check for when we reach the middle point, I check if the current node v appears in v2's explored dict and vice versa. In other words, if the node we are examining for the forward search shows up in the explored list of the backwards search, then we've hit the middle point.

The heuristic had to be modified in order to work with the bidirectional search. Originally the heuristic didn't need an end parameter for the goal since it could get it from the map_ object. However, since the bidirectional search needs to use the heuristic for distance to the start when doing the secondary search, I needed a way to input which end I wanted it to check.

This shouldn't be an issue, but if for some reason there was a discrepancy

of the times on the testing hardware, I thought I'd include my output.

```
$ python Main.py -filename msh.npy -AI AStarExp
Time (s):  15.79529094696045
Path cost: 505.0
Nodes explored:  116150

$ python Main.py -filename msh.npy -AI AStarMSH
Time (s):  8.209494829177856
Path cost: 505.0
Nodes explored:  84229
```