PA1 CS180

Gabriel Henry J. Lopez (2016-46039)

Algorithm Details

When the program starts, it collects all the args and uses opencv2 to parse the image from the args. Then the program parses the image to find the start and the end. Then the search function runs and the output is sent to the output file arg. More details on each below:

Priority Queue

I used a python list-based priority queue with added functions. Each element in the list is another list which represents a node.

The structure of a node

[y coordinate, x coordinate, f value, [parent y coordinate, parent x coordinate, parent f value], g value]

Added functions

- find_parent(data)
    - Find the parent of node data and returns it
- lowest_similar(data)
    - Finds a similar object in the queue with the lowest f and returns it

The priority queue is prioritized according to the f values of each element in the list

Start & End locators

Functions find_start and find_end are both used to locate the start and end of the map. Once the map is parsed, the start is simply found by locating an element where the blue channel is 255. Similarly, the end is found by looking for the element with a red channel of 255

Parsing Map

When the map is parsed, each 50x50 square needs to be treated as one element in the list. To parse it, loops in the code simply increment its values by 50 instead of 1.

Search

The search function takes in the map, the start coordinates, the end coordinates, the algorithm, and the heuristic.

Pseudocode:

Initialize the open and closed priority queues
Insert start into the priority queue
While open_list is not empty
    Get q from the open queue
    If q is the end value, then stop the search
    Generate successors of q
    Parse the color data of q
    For each successor do
        Get color data of successor
        Calculate for f
            If the algo is UCS, f is just the difference in intensities
            If the algo is GBFS, f is the heuristic
            If the algo is A*, f is the difference in intensities + the heuristic
        Add f value to the element
        Add the parent of the element to the element
        Append the g value to the element
            For A* and UCS the g value is used for future f computations
            For GBFS, the g value is used solely for calculating the total distance at the end of the program
        If there is a similar element in the open list with a lower f, then don't add the successor
        I there is a similar elenet in the closed list with a lower f, then don't add the successor
        Otherwise, add the successor to the open list
    Insert q to the closed list
Once it is empty, copy the map into another variable
If show attempts arg is true
    Copy the closed list into an attempts list
    Draw all the attempt lines with a gray line
Set curr to q

While curr is not the start
    Draw a line between curr and its parent
Print out the values of nodes expanded and line distance
Write the image to the outputs arg

Heuristic Calculations

The calculate heuristic function can calculate different heuristics based on the value of the heuristics arg. The default heuristic is Fixed Singular Dimensional Distance (See Heuristics below)

Heuristics

There are 3 heuristics that are programmed into the system currently

Manhattan Distance

Regular Manhattan Distance.

Calculation:

|x1 - x2| + |y1 - y2|

Pros:

- One of the best ways to calculate for the heuristic

Cons:

- It fails once you have walls and obstacles

Diagonal Distance

Regular Diagonal Distance

Calculation:

max(|x1 - x2|, |y1 - y2|)

Pros:

- There are no pros

Cons:

- It's meant to work for games where you can move in 8 directions instead of just 4
- Also fails when you have walls and obstacles

Fixed Singular Dimensional Distance

The heuristic function I created. Picks a single dimension at the start of the search (x,y) then uses that singular dimensional distance as a heuristic for the whole search. The dimension picked is always the dimension with the highest singular dimensional distance between the start and the end nodes at the start of the search. E.g. if the SDD(start_x, end_x) is larger than SDD(start_y, end_y) at the

This was inspired by Diagonal Distance which is always going to

Singular Dimensional Distance (SDD)

The distance between two nodes if they were one dimensional.
Example:
(1,3) and (2,5)
SDD(x1, x2) = 1
SDD(y1, y2) = 2

Calculation:

At the beginning of the search, it sets the single dimension to use for the entire search. The dimension can be either x or y.
It is then calculated as:

SDD(dimension1, dimension2) = |dimension1 - dimension2|

Example:
SDD(x1, x2) = |x1 - x2|
SDD(y1, y2) = |y1 - y2|

Proof of Admissability:

The value of SDD can never be greater than the actual distance between the elements. They can only be equal if the start and the end are both in a straight line

Pros:

- Moves towards the end with less expanded nodes

Cons:

- Path moving towards the end might not always be the best path
- Once at the same single dimension as the end, (when SDD equals 0) the heuristic breaks
    - Which makes it really bad if the start and end are on opposite corners of the map

Suggested Heuristics

These heuristics were ideas that I wanted to expand on but I wasn't able to implement them properly

Dynamic Singular Dimensional Distance

Similar to the Fixed Singular Dimensional Distance, but once SDD(dimension) = 0, then it will switch the dimension to the other dimension

Pros:

- Fixes the issue of being stuck at SDD(dimension) = 0 and being left with no heuristic

Cons:

- Once you switch dimensions, you will either have to
    - Remove all items from the open list
    - Refresh all the heuristics in the open list with the new dimension

  Both of these are somewhat costly solutions

Predictive Singular Dimensional Distance

Where the SDD really shines, is when it is used in conjunction with detecting walls and obstacles.

Implementation

When the node has a neighbor (another node in one of 4 dimensions) that is an obstacle (walls) and said neighbor has a lower SDD, then we can add 1 for the heuristic. A wall with a lower SDD means that there is a wall in between the location of your current node and the node moving towards the end. By adding 1, we're making the heuristic more accurate because we're taking into account the fact that you have to make at least another step to go around the wall.

This implementation only works with SDD, and not with MH or DD because a wall with a lower MH or DD might not actually be blocking your way towards the goal e.g. when the end is to the NW diagonal of your current node and the wall with lower MH is to the north of your current node but there’s no wall to your west.

SDD is best utilized when in conjunction with predictions