Chet Gnegy –chetgnegy@gmail.com

NavPal Documentation for Map Representation and Path Planning

Map Representation:

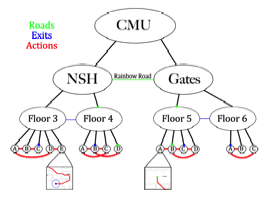
The map representation aspect of the NavPal project works as follows: The map as a whole is set up as a hierarchical structure. The lowest level being the grid level. The grid level is defines the granularity at which the user can movement is represented. On the maps that were used, the size of a grid cell is about 1.4 meters. Only one map can be viewed at a time, or more specifically, one floor. Each map consists of an interconnected network of rooms. So, working up from the lowest level, the nodes, we have rooms, floors, and buildings. Theoretically, this could continue to the city level or higher and the path planner should still work. These objects are all of a similar type, called Vertex. This is used in the same sense as in any standard documentation of the A\* algorithm. They are connected by Edges. In this program, all graph level hierarchies (everything above nodes) implement Vertex and Edge. This allows for a recursive planning algorithm. At the room level, we have Actions (think doorways) connecting the Rooms. These connections are determined at runtime. Note that most of the map’s data are stored in text files and are read in rather than calculated. Profiling data indicates that determining actions takes about 30ms per map and is therefore not a huge computational concern. For comparison, the sector and room data (explained later) may take over a minute to load from scratch and it is much more convenient to store it in a text file. The to and from field of Actions provide data as to what the action connects. On the higher level we have Exits connecting floors. Stairwells and elevators are treated as edges in this case, linking the floors. When these connections are established, they are compared by their string fields. An exit called “stairs” is only connected to another floor’s “stairs” if their longMes fields are both the same. Exits titled “Exit” are not considered because they are assumed to lead outside. This is a safe assumption because only administrator privileges can create Exits. Since only one building was used, no higher level Edges were implemented. This could continue to roads linking buildings, highways linking cities, etc. The vertices and edges allow for an abstract structure for the maps, but each level works in different ways. The building level currently only functions as a structure to hold the floors, and the room level only exists to aid the path planner and provide data to the path predictor. Mostly all interactions will take place on the floor level, represented by the Map class.

The map class stores many important structures. Annotations, for example, are stored here. The rooms also store their own annotations, but this was established only to aid the path prediction. Annotations consist of obstacles, landmarks, and exits. Exits were discussed earlier and are mainly used as Edges. The user also can specify to leave via the nearest exit. This will work in the single floor representation, but needs some modification to work on the multi-floor representation. It uses the closest exit (Euclidean distance) to the user and plans accordingly. On multiple floors, this will not be so straightforward and no efforts have been put towards resolving this so far. Obstacles are markers that indicate that grid cells are not traversable even though they may be free space on the map. They are circular and their radius is specified by the size field. Landmarks are essentially destinations that the user can choose from. They are like obstacles, but they are merely a point (size = 1) and are traversable. These will also need to be adapted for multi-floor.

For convenience, there are structures like RoomsMap[][]. This takes the longest of all the matrices to calculate from scratch. Hence it is stored in (MapName)\_sector.txt on the SD card. This matrix stores the id’s of the room that each grid cell is in. Therefore, RoomsMap is the same size as the map (node[][]) The id’s of the room are the same as their index in Map.rooms. These fields only exist when there is an svg file provided to the application. These are obtained by converting the pdf on the CMU website to an svg document. Inkscape is a program that can be used to analyze these files. The application reads these files and stores the coordinates of the rooms in Room.vertex and will proceed to calculate RoomsMap upon loading if the files are not provided. The reading in of files is documented in the code, specifically, MapIO, ImageConvert, and Room classes.

Once the map is loaded, the DisplayView class controls all movement of the map and scaling. The image transform matrix must be used as a medium between the users clicks, the Map.node[][] matrix and the actual display of the image. All calculations are done using the coordinates in the node[][] matrix. The ‘scale’ variable is not really used any more. It can be used for specifying an initial zoom on the map. This would be done in initialize display in the post scale line. The variable res is constant once the map has been read in. It is a scaling factor between the size of nodes[][] and the size of the floor plan in pixels. Or put differently, it is the number of square pixels per grid cell. 6 is a good number. Any larger and the resolution of the map gets really bad. Any higher and the path planner takes considerably longer.

The notion of a complete map simply means that an svg file is supplied. The program will only be able to function at a minimal level (if at all) without one. That being said, once the map data has been stored in a text format, the svg is not worth holding on to. Without svg data, the map will have no rooms, only a map with grid cells. The localization will still work (even though none of the incomplete maps have been tested with localization) but the path predictor will have no data and will be essentially useless.



The Path Planner:

The image above shows a visual representation of the map structure. The path planner works in such a way that when supplied with a destination and a user location, if will generate a path between them. This is set up to work recursively on the graph level and should be equipped to handle any number of hierarchies. Consider the user in room E of NSH floor 3 and the destination on floor 5 of the Gates building in room A. The first thing that the path planner does, is it traces up the tree to a common ancestor. In this case, the common ancestor is the CMU node. The discovery of a common ancestors indicates the existence of a path. The planning starts at one level below the common node. So, the planner looks for a connection between NSH and Gates. The one it finds, though there may be many, is Rainbow Road (the skybridge). It stands that if the user can find a path to Rainbow Road, they can successfully find a path to the destination. The planner then prepares to run at the floor level. The planner needs a path between the floor that the user is on and the floor that Rainbow Road is on. These are connected, for example, by elevator. It stands that if the user can find a path to the elevator, that they can find a path to Rainbow Road. The planner now prepares to run on the room level. The elevators on floor three are located in room C, but the user is in room E. The planner runs again finding a path from room E to D to A to B to C. Since this is the lowest hierarchy represented by the graph planner, the elevator and the user are passed to the grid planner. The grid planner is restricted to running only on the grid cells located in rooms A,B,C,D, and E. If there was some room F on NSH floor 3 that was connected to room C, the path planner would not be allowed to enter. This makes the path planner much faster. The blue and green dots on the chart (the ones in the network, not in the bottom-most boxes) represent that the nodes of the exits (elevators in this example) and of the road (Rainbow road) must be passed down to the grid level. This is so that the planner knows to guide the user to the elevator.

The grid planner does not run all the way to the 5th floor of the Gates building. It only leads the user to the elevator. Once the user reaches the 4th floor of NSH, the planner recalculates and the user is guided to Rainbow Road. Once the user reaches Gates 5 from Rainbow Road, the user is in room B. The path planner runs again, finding common node Floor 5 (Gates) and the planner only runs on the rooms and finally the grid (restricted to A and B). This has been tested only on the two floors, but should work with more complex map hierarchies with little to no modification. Note that no replanning takes place on the graph level and that it is recalculated each time. This needs revised (consider that the graph level need not be recalculated unless the user leaves the path designated by which rooms are enRoute() or leaves the floor). Also, in its current state, the first exit connecting floor 3 and floor 4 is used, even if it is not the optimal exit. This is another modification that will need made.

Grid replanning is handled in a way differently to how it is done in most D\* lite papers. The algorithm works backwards starting with the destination, this allows the user to move with very little recalculation. Each node is labeled with a time that it was added to OPEN and a time that it was added to CLOSED. When the user moves, the youngest node within an immediate proximity is chosen and the OPEN and CLOSED sets are reset to their state at the time in which that node was added to the open set. The planner is then run from that state until the user is located.

A note on restricting rooms with the enRoute parameter: nodes inside restricted rooms can be added to the OPEN set, but cannot be expanded upon or added to CLOSED unless that room becomes enRoute (requires replanning). This keeps the path planner from being blocked by a closed node in a doorway should the path change.

The planner right now only looks for the connection between the fewest number of floors or buildings or whatever. It disregards the possibility that by taking a route through 4 floors may actually be less walking than another route through 2 floors. This can be corrected, or at least approximated by using a cost function. Ideally, edges will have a cost as well as vertices. A basic cost function could be:

Room cost: proportional to area of room

Floor cost: proportional to number of rooms

Or alternatively, and perhaps more accurately:

Room cost: average distance between any two doorways.

Floor cost: average distance between any pair of exits.