Software project

**Notes:**

* idea: make parent class *Conflict* with sub classes *position\_conflict* and *edge\_conflict* ? So maybe conflict just has time and agents ? And others have position aka coordinate and edge has tuple of coordinates. How would this affect the methods for finding conflicts? I feel like as it is now it wouldn’t affect the methods but I haven’t looked much into it.
* I don’t like how messy it is to identify conflicts; there are too many methods, or it just looks messy.

**Code:**

*data.py*

* class *Agent* 
  + *origin* which is an integer tuple *(x,y)*
  + *destination* also integer tuple
  + *index* is a non-negative integer which serves as an identifier
  + *path* is a list of nodes which is how agent gets from *origin* to *destination.* Not sure if necessary- yes I think it is because this is what is updated after optimal solver is used to resolve conflicts. Also, would be needed later for graphics.
* class *Position\_Conflict*
  + *time* is a non-negative integer which represents the discrete time slice at which the conflict occurs
  + *agents* is a list of *Agent* objects which are involved in the conflict. At initialization the only agent involved is the one that was being “investigated” at the time of instantiation. This makes more sense after reading about how conflicts are found. Basically, we create a position conflict for every position that an agent is at and if we find another agent that is at that position at the same time then we just add to this list instead of making a new *Position\_Conflict* and at the end we delete *Position\_Conflict* instances that only contain one agent because that isn’t actually a conflict.
  + *position* is an integer tuple *(x,y)* which is the coordinate that along with *time* identifies the conflict.
  + note: maybe update or delete the string representation because the current version only has agents and position and should include time as well. But I don’t think I need this anymore
* class *Edge\_Conflict*
  + *time* same as for position conflict
  + *agents* also same
  + *edge* is a tuple of coordinate so *((x1,y1),(x2,y2))*
  + same note about string representation as above
* *class Sub\_Nodes*

Wanted this class to be something I can use to make sub-environments that I can feed to optimal solver, but I’m not confident that it will work / is built well enough. Consider thinking of other ways to do it this and represent the sub environment. Maybe just a graph class with some type of special constructor or maybe just a graph and subgraph class. Would subgraph be a child of graph?

What I need for the picat solver:

List of agents involved,

Restrictions – where these agents can’t go, for each restricted path I need to know the time, location and agent that is there

Nodes – all the nodes that are available to these agents

Neighbors – a dictionary of type node:neighbors

Filename that the instance will be written in

* Method *read\_map* 
  + process lines describing map type
  + TODO: what is the first line that says “type XXX” what are the different map types??
  + read map portion line by line and add to list of nodes
* Method *get\_agents*
  + in the main method we can get information that specifies number of agents to read
  + TODO: check if number specified is more than total agents in the file??
  + keep reading agents from file while we have less than specified amount or before we reach the end of the file. Whichever comes first. If no amount is specified, we have a default value of -1 which tells us to read until the end of the file.
* Method *find\_edge\_conflict* 
  + gets a time and edge and iterates through list of edge conflicts until it finds a conflict with the same time and edge value or reaches the end of the list. Returns index of found conflict or -1 if no conflicts match.
* Method *find\_position\_conflict* 
  + Analogous to previous method but with position of conflicts instead of edge.
* Method *update\_conflicts*
  + iterates through agent’s path and for each edge and position checks if a corresponding conflict already exists. If it does, then the agent is appended to list of agents present in that conflict, otherwise a new conflict is created and appended to the corresponding list.
* Method *get\_neighbor\_dict ????*
* *main* method:
  + get arguments from input. User can specify a *.scen* file and/or a *.map* file and/or the number of agents to process from the *.scen* file. If no file name/ number of agents is specified, then there are default values “*maze-32-32-2-even-1.scen”* and -1 (signals that all agents are to be processed).
  + load map into 2D array using *read\_map* function
  + read and save agent information into an array of *Agent* objects using *get\_agents* method
  + use *informed\_search* method in *informed\_search.py* to find a path for each agent. After path is found for agent update conflict lists using method *update\_conflicts.*
  + TODO for each (or somehow combine) conflict(s) turn into sub-environment
  + call method that turns sub-environment into an instance for optimal solver
  + call optimal solver
  + update old paths using output from optimal solver

*optimal\_solver.py*

* This is a class called *Optimal\_Solver* which gets relevant data from the *data* class and works with the *.picat* solver to find conflict-free paths and relay this data back to the *data* class
* Initialization method:
  + All values (*agents, restrictions, nodes, neighbors, filename, makespan, sum\_of\_costs)* which are needed by the .picat solver were passed as input so we can immediately call the *create\_translation* method, which uses these values to create the corresponding input file for the optimal solver.
  + *Agents* is a list of agents that we want to solve for
  + *Restrictions* is a list of paths to avoid (???)
  + *Nodes* is a list of nodes that we can use
  + *Neighbors* is a dictionary with keys being nodes and value is a list of the node’s neighbors
  + *Makespan ???*
  + *Sum\_of\_costs ???*
  + *File* is where we have the input for the solver
* Method *call\_solver*
  + This uses *subprocess* library to call the optimal solver on the file, that we made during initialization
  + Correct usage:

*A black background with white text

Description automatically generated*

* Method *create\_translation* 
  + Makes a file with the correct information in the format:

*A screenshot of a computer

Description automatically generated*

* Method *get\_neighbors\_lines* 
  + Iterates through the list *nodes* and uses the dictionary *neighbors* to create a list of strings, where each string is one line of type “$neibs(Node, [Neighbors])”
* Method *get\_avoid\_lines* 
  + TODO: not sure what this should be doing/ what to do with this
* Method *process\_output* 
  + Right now, this just calls decode on the output from the *subprocess* that was called in the *call\_solver* method and then returns the result from that

*informed\_search.py*

* The *Vertex* class provides additional data for vertices visited by A\*
  + *coord* is the actual coordinate, which is a tuple *(x,y)*.
  + *distance* is the distance from *origin*; therefore, it is non-negative integer value.
  + *heuristic* is the estimated distance from the goal; also, a non-negative integer value.
  + *predecessor* is *None* or an instance of *Vertex* class, which allows access to the node that precedes this node in the path from origin to goal.
  + *explored* is a Boolean value which represents whether A\* has already processed this node. Upon initialization is set to *False*.
  + Operator ‘<’ compares distance from origin (*distance*)
* The method *get\_path* gets two *Vertex* objects, *origin* and *destination* and constructs a reversed path by making use of the *predecessor* property of the *Vertex* class. When the *origin* vertex is reached, then we’re done and can reverse the built path and return the path from *origin* to *destination* in correct order.
* Method *valid\_node* gets the array of possible nodes (*nodes*) and the node for checking (*node*) and returns true if *node* is a valid coordinate on the map that doesn’t contain an obstacle (represented by “@”).
* Method *get\_neighbors* returns a list of valid neighboring nodes to the specified node (*node*). The map (*nodes*) is also passed because it is used to check validity (validity is checked for each prospective neighbor individually using the *valid\_node* method).
* Method *heuristic* returns the Manhattan distance between the *origin* node and *destination* node, which are both integer tuples.
* Method *informed\_search* 
  + Uses A\* to find path from specified start and goal coordinates.
  + passes *nodes*, which is an array of rows; that means that to access the element in the 2nd row and 3rd column one would write *nodes[3,2],* aka *nodes[y,x]*.
  + also passes *origin\_coord* and *destination\_coord* which are integer tuples *(x,y).*
  + add more to the explanation of this method?

**Timeline**

* 30.07 Changed how conflicts are found

Used to be that after all paths are found go back and iterate over a list of the paths and then iterate through all possible times and find edge and position conflicts for that time. Now conflicts are updated after a path is found. So, there are no extra for-loops, and we update the conflicts immediately after the path is found using A\*. Code is a lot cleaner and more legible. At the end all you must do is reiterate through all conflicts and delete the ones that only contain one agent.

* 10.08 work with optimal solver

The subprocess call works now to run the optimal solver using picat.

* TODO:
  + make newer version of method in data class to create sub problem
  + check that the file made in optimal solver is working correctly
  + error handling in optimal solver??