## Scenario and Default operations

Our scenario is an agent in a maze of rooms. Rooms are connected by doors and agents can open, close, lock, unlock the doors using different keys which can be found in the world. Agents can move around the maze by entering different rooms until they reach the end state.

## Tuple Size

### Overview of the test

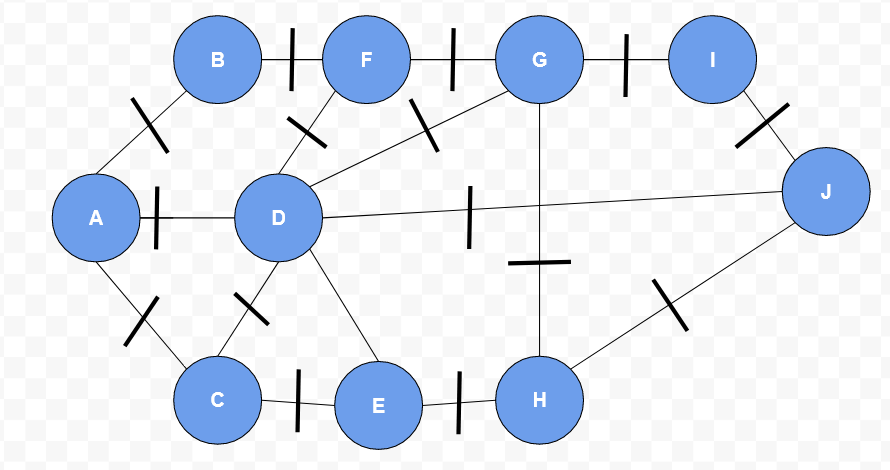
In this experiment we are evaluating the efficiency, performance and results of both the ops-search and planner in respect to changes in the tuple size, specifically, how many keys the agent is holding. We expect that the more keys that the agent is holding the longer the search will take.

### Plan of how the analysis will be performed

In the tests I will adjust the state so the agent will hold more keys. I will add more keys to the state and run the tests to see the ‘base’ result. Then, I will make the agent hold more keys to see if it affects the search result.

### Diagram of the State

The state has multiple ways of getting to the same location/room which is an additional analysis for this test.



### Graphs and Results

#### Overview of tests

I will run a series of tests for both ops-search and planner starting from basic operations and changes to the state to more complicated and advanced scenarios.

### Analysis and Explanation of results

### From my analysis and experiments, I have concluded that the total time taken for ops-search to complete the operation is not directly dependant upon the number of goal states specified but is in fact relational to the number of intermediate operations it must complete.

### In tests one to five, where the agent engaged with two adjacent rooms. We can see that the more goal states that are added the longer the process takes to complete. In test five, it becomes almost impossible to solve as we left the process running for one hour and it was still incomplete.

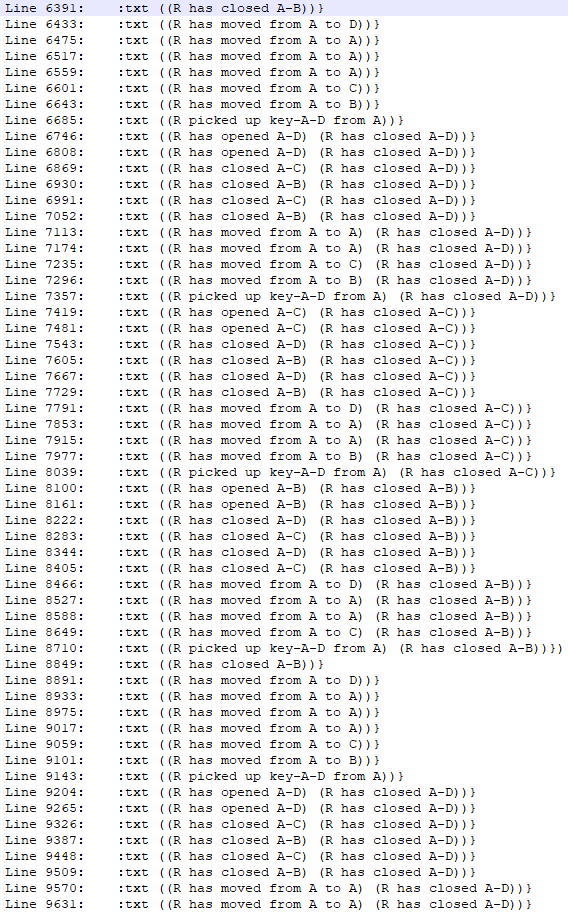
### However, in tests six and seven, we attempted to make the agent go to deeper depths and perform operations in rooms which where not directly next to each other. From this we could see that even though we specified less goal states, it took longer than before as more intermediate operations (operations that needed to happen in order to complete the overall objective) needed to be complete.

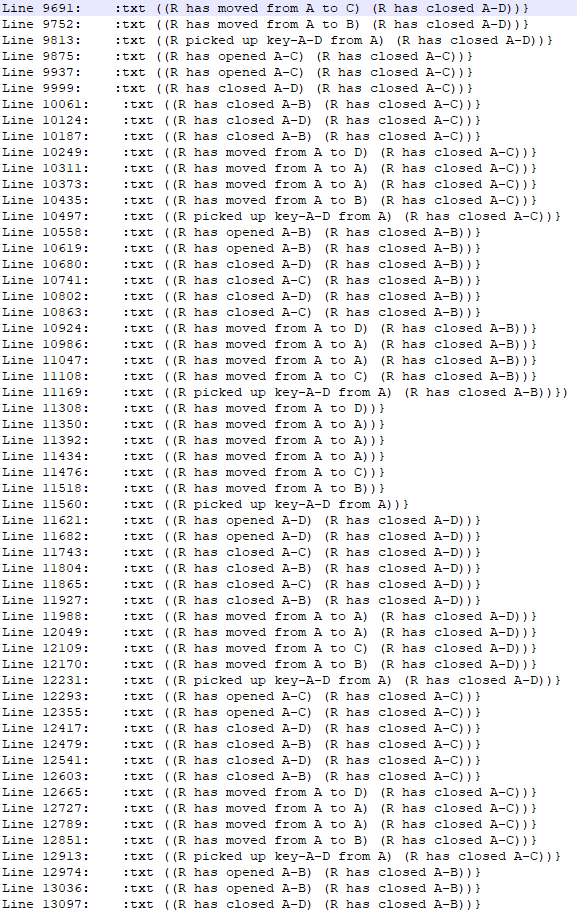
### ***The time complexity of a breadth first search for ops-search is 0(V+E) which is the number of vertices plus edges it must visit.***

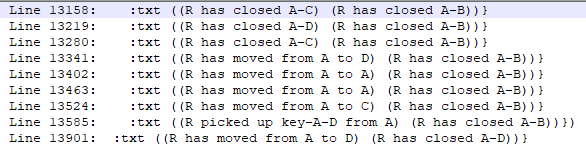
### Ops-search is a type of breadth first search, so it will start at a specified node and added all adjacent nodes to a queue and visit each of them. Ops-search will then look at the state it’s been given and compare the *:pre* of each operation for an action with the state. If the *:pre* operation is valid (the state and the *:pre* match) then the *:add* and *:del* operations for that action will be applied and the *:txt* operator printed out. If the goal state then contains the new state, the program will terminate and give an answer. Otherwise ops-search will continue applying and checking other operators for different nodes until it finds a solution or exhausts all options. By definition, the more operators it must perform the more nodes it will likely visit and the more matches it will have to apply inorder to get the final goal state.

As we can see from a selection of *:txt* outputs below, the ops-search will check every operator and apply the ones where the *:pre* matches the state regardless of if it is ‘logical’ to perform that operation. For example, in test-two, there is no need to try and pick up a key as the specified goal state is to move to a room where all the doors are already open. By performing these actions the time complexity of the request is greatly increased. Additionally, the ops-search engine ‘repeats’ operations that have already been tested which leads to unnecessary checks being performed and applied.

### 



****

****