Code Structure

- **node.py**: The basic unit of our tree. Each node contains their own state (position of pieces), and can decide whether the game end or not. Most of the node can be expanded, unless they reach the final state.
- **search.py**: The main entry of the project
- **travel.py:** Decide how we are going to expand the tree
- **utils.py**: Contains some constants variable and general method could be used for many files in order to save the memory and repeatedly creating the same thing again and again.

Game As a Search Problem

We regard this game as a search problem from the following perspective

- **State**: Each node contains a state, different node contains different state. How board looks like described a state, every times the board changed, we regard it as a new state. (Unless there is already a better duplicate node in dict already)
- **Actions**: There are three possible actions, MOVE, JUMP, EXIT.
- **Goal tests**: If there are no player pieces on board, we say that we reached the final state.
- **Action cost**: Each action cost 1 (MOVE, JUMP, EXIT)
- In this case, our problem is a discrete, deterministic, sequential, static and fully observable.

Algorithms

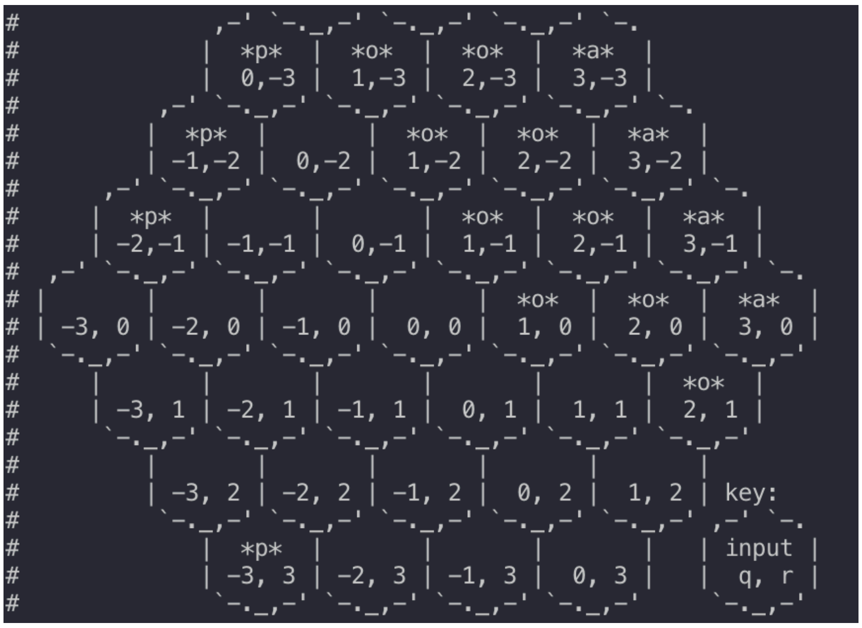
- Why A*
    - For this project, we explored two different algorithms which are **A*** and **RBFS**, A* was chosen at the end. The reason for that is RBFS is a kind of algorithms that have an excellent space complexity but keep the similar time complexity theoretically, since only one branch is expanded at a time. From a practical perspective, it is much slower than A* . The worst case for the space complexity is that there are $C_4^{37} = 66045$ nodes (we have our own way to remove duplications, refer to the following section) in memory, but for all the case we tried, we even did not reach half of it. Even if each node takes up to 10kb memory, there are only 660 MB memory required for the worst case which is affordable for most of the modern computers. Therefore, we chose A* for better performance.
- Pre-processing
    - Before the start of searching, we construct an important dictionary for the current board. For each reachable destination, we obtain the cost from any reachable position on the board to that goal by using BFS and each action (including MOVE and JUMP) cost 1. We do it at most four times, which can be done in constant time (same as explore $4 * 37 = 148$ coordinates) for all test cases. After that, we merge those dict to obtain a single dict and only keep the lowest cost for each coordinate. In this case, the values for that dictionary represent the lowest cost for moving a single piece from that place to one of the reachable destination including the effect of the blockes.
- Heuristic
    - Our heuristic function is equivalent to $math.ceil(cost/2) + 1$, we assume that we can JUMP at every state when we moving our pieces. For odd number of cost, at least you need to MOVE once, therefore, we are trying to find the ceil of the float. The +1 at the end means that one cost required for the piece to exit the board. As we assume that the piece only JUMP (the fastest way of moving any piece under the rule of this game) at each turn, we can conclude that h^* is dominating h. Therefore, our heuristic is admissible.
    - The reason that we chose this function
        - Easy to use and calculate
        - Decent performance when there are many pieces, since around $\frac{1}{3}$ actions are JUMP.
        - Worse case are vary easy to solve in practice. To make sure all actions are MOVE, either a large proportion grids on board are blocked or there are only 1 piece. Both problems have very small search space.
- Properties of A*
    - Completeness: Yes, unless infinite nodes with $f \leq f(goal)$, in this project, we can always assume this is true.
    - Optimality: admissible are proved, therefore, the optimality can be guaranteed.

- Time complexity: $O(b^{\epsilon d})$ for constant step cost which I use 1 for all movements, where $\epsilon = (h^* - h)/h^*$, for the best case, which the true optimal path is that piece JUMP for each turn, the answer will be find in linear time $O(d)$. For the worst case, which piece have to move at each turn, the time complexity is $O(b^{\frac{1}{2}d})$, which is still a significant improve in real problem solving.
- Space Complexity: $O(b^d)$, since it keeps all the node in memory. As I side in previous sections, this is acceptable even for the worst case.

* Data structure for arranging the frontier
  - We use priority queue to maintain the frontier. The cost for that is $O(nlogn)$ for the whole search, whereas, using a custom search method take up to $O(n^2 logn)$ which is horrible.
* Method for removing duplicate nodes
  - We use a dictionary to keep all visited state, once a same state has been reached by expanding another different node on the tree, we starting to compare the g (cost to reach this state so far). If it has a lower g, we update our dict, expand the node and put it successor into the frontier, otherwise, we simply ignore that node since we already have a better way to reach this state. This method dramatically reduce the time complexity and space complexity of our search in practice, since less node are explored and millions of duplicate nodes are not been added to our frontier. More practical data will be listed in the next section.

Test cases

We made up of some test cases to test how good is our algorithm from different perspective.

* How good is our heuristic and frontier update strategy
  - {"colour" : "red", "pieces" : [[0,-3],[-3,0],[-3,3],[0,3]], "blocks": []}
  - Optimal solution: 18 steps, explored nodes: 7741, node in frontier: 20760, takes 1.6s in dimefox
  - During 4 EXIT and 5 JUMP actions (9 in total), our heuristic function are same as h^*
  - This problem could represent one kind of question which has multiple pieces that can freely move around without blocks. For less node, although MOVE action may occur more frequently, but reducing node can result in a dramatically decrease in search space.
  - In conclusion, for this kind of questions, the time complexity are very close to optimal for relatively large number of nodes, and space complexity are much better than the worst case. For small number of nodes, since the dramatically decease in search space, it also performs excellently.
* Break simple min-distance heuristic
  - 

```
#              ,-' `-._,-' `-._,-' `-._,-' `-.
#             |  *p*  |  *o*  |  *o*  |  *a*  |
#             |  0,-3 |  1,-3 |  2,-3 |  3,-3 |
#          ,-' `-._,-' `-._,-' `-._,-' `-._,-' `-.
#         |  *p*  |       |  *o*  |  *o*  |  *a*  |
#         | -1,-2 |  0,-2 |  1,-2 |  2,-2 |  3,-2 |
#      ,-' `-._,-' `-._,-' `-._,-' `-._,-' `-._,-' `-.
#     |  *p*  |       |       |  *o*  |  *o*  |  *a*  |
#     | -2,-1 | -1,-1 |  0,-1 |  1,-1 |  2,-1 |  3,-1 |
#  ,-' `-._,-' `-._,-' `-._,-' `-._,-' `-._,-' `-._,-' `-.
# |       |       |       |       |  *o*  |  *o*  |  *a*  |
# | -3, 0 | -2, 0 | -1, 0 |  0, 0 |  1, 0 |  2, 0 |  3, 0 |
#  `-._,-' `-._,-' `-._,-' `-._,-' `-._,-' `-._,-' `-._,-'
#     |       |       |       |       |       |  *o*  |
#     | -3, 1 | -2, 1 | -1, 1 |  0, 1 |  1, 1 |  2, 1 |
#      `-._,-' `-._,-' `-._,-' `-._,-' `-._,-' `-._,-'
#         |       |       |       |       |       |
#         | -3, 2 | -2, 2 | -1, 2 |  0, 2 |  1, 2 | key:
#          `-._,-' `-._,-' `-._,-' `-._,-' `-._,-' ,-' `-.
#             |  *p*  |       |       |       |   | input |
#             | -3, 3 | -2, 3 | -1, 3 |  0, 3 |   |  q, r |
#              `-._,-' `-._,-' `-._,-' `-._,-'     `-._,-'
```

  - This is where the pre-defined dict really useful. If the heuristic only consider the distance to the closest non-blocked destination, it will quickly starting to do BFS which is extremely slow when comparing to A^* with good heuristic.