Problem, Search and beyond

Chuanyuna Liu 884140, Zhuoqun Huang 908525

1 Problem

1.1 General Problem

We formulate our problem with the following five concepts:

- **State**($\mathbb{S}$) including a **initial state** (s_0) the agent begins with
- **Actions**($\mathbb{A}$) available to agent at each State
- **Transition function** ($F : \mathbb{S} \times \mathbb{A} \to \mathbb{S}$) that takes a (state, action) pair and return a new state
- **Goal Test**($GT : S \to \{True, False\}$) that takes a state s and return true if $s \in S_{goal}$, the **Goal State** of the problem.
- **Path Cost** ($C : \mathbb{S} \times \mathbb{S} \to \mathbb{R}$) That takes two states and return cost moving from one to another

1.2 Single-player Chexers

In this section, we formally describe how the above framework fits the Chexers game.

- Denote set of pieces with $\mathbb{P} = \{(r, q, t)\}$.
  where
  $$r, q, -(r + q) \in [-3, 3] \text{ denotes the location on board.}$$
  $$t \in \{red, blue, green, block\} \text{ stands for type of piece.}$$
- $\mathbb{S} := \{p_i \in \mathbb{P}, t_b | i \leq n\}$ where n is number of pieces on board, where t_b is the searching type.*
- $\mathbb{A} := \{(Move, p_i), (Jump, p_i), (Exit, p_i)\}, \forall\ p_i$ where $t(p_i) = t_b.|\mathbb{A}_s| \leq 6n_p\ \forall s.$
- $f(s, a) := s'$. where s' differs exactly by one piece $r(p_{i,s}), q(p_{i,s}) \neq r(p_{i,s'}), q(p_{i,s'})$ or $p_i \notin s'$
- $c(s, s') := 1, \forall s, s'$ if $\exists a$ such that$f(s, a) = s'$
- $gt(s) = True$ if $\nexists p_i$ such that $t(p_i) = t_b$

*Initial State s_0 given by problem specification.

2 Search

2.1 preliminary

We use the following Algorithm 1, to search for our goal. Based on our problem specification, we have all required components except for **H**($node$), so we will propose one type of **H** we found to be most effective of all and compare it against **Null** $H = 0$ and **bad H** $H_{bad}(node)$[1].

2.2 Heuristic

As a Problem We propose wing problem definition for mapping a good heuristic value for each pieces and we define $H(s) = \sum_{p_i} h(p_i), \forall p_i$ that $t(p_i) = t_b$ after relaxing the problem to be: a piece can choose freely between **move**, **jump** regardless normal constrains and **cannot** move on to blocks.

- $\mathbb{S} := \{(cost_i, position_i) | location_i \in board\}$*,
- $\mathbb{A}$: for pos_i with $cost_i = min(cost_n)$, $\forall\ pos_{j \neq i}$ reachable from pos_i and $cost_j > cost_i + 1$. Update $(pos_j, cost_i + 1)$. If $\nexists pos_j$, remove $(pos_i, cost_i)$ from s and put $h(pos_i) = cost_i$
- $c(s, s') = 0$ and $f(s, a)$ follows definition in $\mathbb{A}$
- $gt(s') = True$ if $s' = \emptyset$

*$s_0 = \{(1, pos_i) | can\_exit(pos_i) = True\} | \{(\infty, pos_i) | can\_exit(pos_i) = False\}$

Heuristic problem solution Finding heuristic is straight forward with the given above problem definition. Supporting all the given operations to 1, and you should end up with a complete heuristic map.

Admissible? The algorithm is guaranteed to provide us with a admissible cost estimation for each game state:
- The relaxed rule let pieces always able to jump.
- This will in all cases reduce the number of action taken, by always increasing the distance a piece can move.

2.3 Property of search

Efficiency The efficiency of A* algorithm heavily depends on how accurate the heuristic is. As we can see from the example below. Our heuristic (shown in Referencesfig:heurstics) is very close to the real cost.

[1] In the following analysis, b for **branching factor**, d for **depth to optimal solution**

Algorithm 1 General A* algorithm

```
PRIORITY-QUEUE                                        ▷ Min Priority Queue (min key)
    ADD(q, key, value),                               ▷ Add (key, value) to q. If value exists, update the key
    POP(q)                                            ▷ Pop the value with least key
    GET(q, value)                                     ▷ Get the key associated with a value
                                                      ▷ All above Queue operations can operate in Θ(1)

NODE                                                  ▷ stores associated state and its parent
Require: EXPAND(node)                                 ▷ expand a node to get it's children
Require: G(node)                                      ▷ get total cost arriving this node
Require: H(node)                                      ▷ Computes an admissible estimation of cost to goal state
Require: C(node1, node2)                              ▷ Give Path cost arriving node 2 from node 1
procedure A*(problem, initial)
    openSet ← PRIORITY-QUEUE(H(initial), initial)
    closedSet ← {}
    while openSet is not empty do                     ▷ O(b^d) repetitions
        node ← POP(openSet)
        ADD(closedSet, node)
        if GOAL-TEST(node) is True then
            return node
        end if
        for child in EXPAND(node) do                  ▷ O(b) repetitions
            cost = G(node) + C(node, child) + H(child)
            if child in closedSet then continue
            else if child not in openSet or cost < GET(openSet, child) then
                ADD(openSet, cost, child)
            end if
        end for
    end while
    return no solution
end procedure
All operations O(1) unless explicitly stated, we denote the complexity of search for problem also here.
```

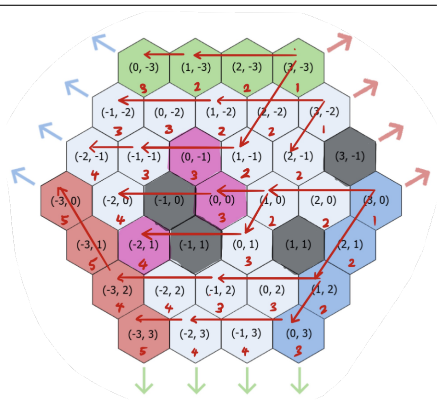


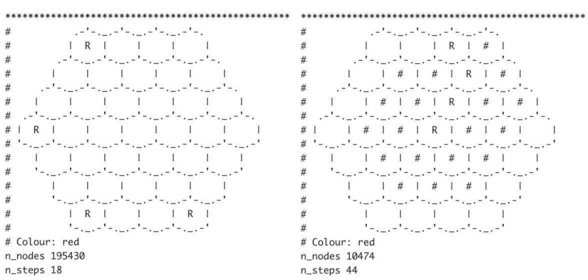
Figure 1: Map formed by heuristic

Optimality A* algorithm can only find the optimal solution if the heuristic is admissible. We relaxed the rule to allow pieces jumping freely without the need to leapfrog another piece. Because jump move allows pieces to move twice the normal move, our heuristic at most underestimates the real cost by a factor of 2, and cannot be faster the real cost. This satisfied admissibility. Monotonicity is also met because each move adds 1 unit of cost. Hence our program is optimal.

Completeness A* algorithm is complete if the graph contains finite nodes. For our problem, both the board and number of pieces are finite leading to a bounded size for state space, concludes the completeness of our algorithm.

3 Beyond

The complexity of problem is **exponentially** related to **number** of **moving pieces**. This is due to the effect of more moving options leading to a higher branching factor. The **further** pieces are from the **goal**, deeper the program has to search.
A* algorithm stores each layer of nodes in a priority queue and only expend the *node* n with the $f(n)$. In the **worst case**, it expands all the nodes fully, store and sort them in order of cost, giving $O(b^d)$ complexity for both **space and time**. In the best case, our heuristic matches the real cost, and A only expand nodes with the correct path. This gives us $O(bd)$. The **high memory complexity** of A* ($O(b^d)$) can be avoided by using **iterative deepening A search**. This would reduce A* algorithm's **space complexity** to $O(bd)$ (by trading off some time complexity).
The amount of free space on the board also affects the complexity of algorithm. From 2 we can see that as the board becomes more empty, we have more actions to consider. Despite the low cost solution of 2a, its free space is drastically larger, leading to a significantly higher number of expanded nodes than that of 2b.



(a) More options (b) Less Options

Figure 2: Search space illustration