**Computational Complexity Analysis**

**Related Code:**

class SearchNode

{

public:

  State state;

  int gn = 0;

  int fn;

  SearchNode \*parent = nullptr;

  SearchNode(State s) : state(s){};

};

struct compare

{

  bool operator()(const SearchNode \*const &a, const SearchNode \*const &b) const

  {

    return (a->fn > b->fn);

  }

};

void Solver::availableActionsWithHeuristic(State s, priority\_queue<Action> &actions, vector<Goal> g)

{

  while (!actions.empty())

  {

    actions.pop();

  };

  for (int i = 0; i < s.returnGridSize(); i++)

  {

    for (int j = 0; j < s.returnGridSize(); j++)

    {

      State a(s);

      if (i != j)

      {

        if (a.moveBlock(i, j))

        {

          Action act(i, j);

          act.calculateHeuristic(a, g.at(0));

          actions.push(act);

        }

      }

    }

  }

};

void Solver::aStarAlgo(State s, vector<Goal> g)

{

  priority\_queue<SearchNode \*, vector<SearchNode \*>, compare> searchNodes;

  stack<State> path;

  int steps;

  Action act(1, 1); // arbitrary

  act.calculateHeuristic(s, g[0]);

  SearchNode \*se = new SearchNode(s);

  se->fn = act.heuristic;

  searchNodes.push(se);

  while (!searchNodes.empty())

  {

    SearchNode \*current = searchNodes.top();

    searchNodes.pop();

    if (checkGoals(current->state, g))

    {

      steps = current->fn;

      while (current != nullptr)

      {

        path.push(current->state);

        current = current->parent;

      }

      break;

    }

    priority\_queue<Action> actions;

    availableActionsWithHeuristic(current->state, actions, g);

    while (!actions.empty())

    {

      State a(current->state);

      if (a.moveBlock(actions.top().source, actions.top().destination))

      {

        if (!visitedStates.count(a))

        {

          visitedStates.insert(a);

          act.calculateHeuristic(a, g[0]);

          SearchNode \*temp = new SearchNode(a);

          temp->parent = current;

          temp->gn = current->gn + 1;

          temp->fn = act.heuristic + temp->gn;

          searchNodes.push(temp);

        }

      }

      actions.pop();

    }

  }

  while (!path.empty())

  {

    path.top().printBoard();

    path.pop();

  }

  cout << "Found in " << steps << " steps!\n";

}

**Algorithm summarised by word:**

The function Solver::aStarAlgo(State s, vector<Goal> g) is an implementation of the A\* algorithm. This function takes in a State object, which represents the initial state, and a vector of Goal objects (actually only 1 goal!), which represents the goal state. The algorithm works by creating a priority queue of **search nodes**, where each search **node contains a state and information about the path taken to reach that state through using pointer “parent”.** The priority queue **is ordered by the sum of the cost of the path to the current state and the estimated cost from the current state to the goal state.** The algorithm then **iteratively expands the search nodes** in the queue, **selecting the one with the lowest cost and checking if it is a goal state**. **If it is not a goal state, the algorithm generates a set of possible actions and corresponding states that can be taken from the current state,** evaluates each possible state, and adds the resulting search nodes to the **priority queue**. This process **continues until a goal state is found or the queue is empty.**

**Algorithm complexity:**

**STL data structures used: Priority queue, stack, unordered\_set**

* priority\_queue<SearchNode \*, vector<SearchNode \*>, compare> searchNodes;
* priority\_queue<Action> actions;
* stack<State> path;
* unordered\_set<State, StateHasher> visitedStates = {};

**First run within:**

  while (!searchNodes.empty())

Graphical user interface, text

Description automatically generated

If we neglect all O(1) then one run would be:

O ( log[(n-1)^2]\*(n-1)^2+ 7n + log[(n-1)^2] )

* O (log[(n-1)^2]\*(n-1)^2)
* O (log(n)^2\*n^2)

The number of expansion of a lowest f-score state after the each run would determine the final complexity. My assumption is the height of an expanded tree would never exceed “n^2-n” (number of blocks with **no solution, f-score never decrease and new states keeps added to visitedState**).

But since the number of expandable states is decreased as visitedState grows (as we go deeper in tree)

(From n^2-n to -> 0)

If no solution then the big O complexity is

**O(log(n)^2\*n^2]^[(n^2)/2])**

If we assume there is a solution, then final complexity will be the first run to the power of log(g-score)

(g-score of the result always smaller than n^2)

Then the practical big O complexity is

**O(log(n)^2\*n^2]^log(n^2))**