8 Puzzle Problem

Table of Contents

Introduction

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right).

```
   1  3       1     3       1  2  3       1  2  3       1  2  3
4  2  5   =>  4  2  5   =>  4     5   =>  4  5      =>  4  5  6
7  8  6       7  8  6       7  8  6       7  8  6       7  8
initial                                                  goal
```

How to install and run

First install `rustup` from here. `rustup` is the toolchain for a language called `rust`, which this project is written it. Installing it will allow you to run my program.

> If you don't want to install it, I've also included a Windows executable that can be run without any dependencies. Look for `project_1.exe` in the root folder.

Once `rustup` is installed, execute the following commands to run the code -

```
git clone https://github.com/DhruvDh/project_1_itcs_6150.git
cd project_1_itcs_6156 && cargo run
```

If you wish to run the program on custom values of initial and goal state, find `fn main()` inside `src/main.rs` and change this line to reflect the values of initial and goal state you desire -

```
let mut problem_1 = Problem::new(
        vec![1, 2, 3, 7, 4, 5, 6, 8, 0], // initial state array
        vec![1, 2, 3, 8, 6, 4, 7, 5, 0], // goal state array
    );
```

Change the cost function by changing the argument for the `solve` function call.

```
problem_1.solve("Manhattan"); // any value other than manhattan will use hamming distance as cost function.
```

Problem Formulation

- **Goal**: Pieces end up in locations as described by the goal state.
- **States**: All possible permutations of the puzzle.
- **Actions**: Move the blank Up, Down, Left, or Right.
- **Performance Measure**: Number of total moves in the solution, if the solution exists.

Problem Structure

The program is composed of 3 major things. 2 structs `Problem` and `State`, and a `main` method. The problem state exposes a `solve` method, which is all that is needed to find a solution. More detailed information can be found by clicking on the respective struct's name further down this page.

Global Variables

There are no global variables in this implementation.

Structs

`State`

A struct to encapsulate State information.

Properties

- `is` : A vector of integers, used to store the positions of numbers in the puzzle.
- `cost` : Cost of this state (g() + h())
- `g` : Cost to get here from the root.
- `h` : Estimated cost to the goal state.
- `kind` : The kind of action to be enacted to reach this state from the parent.

Further details

I am using Binary heap as a priority queue to choose the next state to move to. The priority here is the "cost" in reverse order. To implement this I need to implement ordering trait for State. This is done via implementing 4 traits - `Ord`, `PartialOrd`, `Eq` (or equality), `PartialEq`. The ordering trait is implemented such that a State with a lower cost is considered greater than a state with a higher cost.

Methods

`State` doesn't have any methods implemented.

`Problem`

A struct to encapsulate information about the current problem.

Properties

- `state` : A smart pointer to an instance of struct State.
- `goal_state` : A vector of integers defining the goal state.
- `visited` : A HashSet that stores all states that have already been visited.
- `under_consideration` : A BinaryHeap that keeps track of states we'll be choosing the next state from.
- `no_generated` : A counter to keep track of number of nodes generated.
- `no_expanded` : A counter to keep track of number of nodes expanded.
- `heuristic` : A string describing the current heuristic being used.

Methods

The following are the methods implemented by the struct Problem.

```
pub fn new(init_state: Vec<isize>, goal_state: Vec<isize>) -> Problem
```

A constructor that produces instances of struct Problem. Takes initial state and goal state as arguements.

```
pub fn expand(&mut self) -> Vec<Rc<State>>
```

Expands the current state of the Problem (self.state). Returns a vector of smart pointers to newly generated states.

```
pub fn solve(&mut self, heuristic: &str)
```

Chooses the best possible state and moves to it in a loop until the goal state is reached.

```
pub fn trace_soln(&self)
```

Traces the path from goal state to root and reverses it.

Examples

Case 1:

```
Current State:             Goal State:
-------------             -------------
| 1 | 2 | 3 |             | 1 | 2 | 3 |
| 7 | 4 | 5 |             | 8 | 6 | 4 |
| 6 | 8 | 0 |             | 7 | 5 | 0 |
-------------             -------------
Solving using Manhattan distance...
Expanded 19 nodes.
Generated 33 nodes.
Solution is ["Up", "Left", "Down", "Left", "Up", "Right", "Down", "Right"]

Current State:             Goal State:
-------------             -------------
| 1 | 2 | 3 |             | 1 | 2 | 3 |
| 7 | 4 | 5 |             | 8 | 6 | 4 |
| 6 | 8 | 0 |             | 7 | 5 | 0 |
-------------             -------------
Solving using Hamming distance...
Expanded 38 nodes.
Generated 63 nodes.
Solution is ["Up", "Left", "Down", "Left", "Up", "Right", "Down", "Right"]
```

Case 2:

```
Current State:             Goal State:
-------------             -------------
| 2 | 8 | 1 |             | 3 | 2 | 1 |
| 3 | 4 | 6 |             | 8 | 0 | 4 |
| 7 | 5 | 0 |             | 7 | 5 | 6 |
-------------             -------------
Solving using Manhattan distance...
Expanded 14 nodes.
Generated 26 nodes.
Solution is ["Up", "Left", "Up", "Left", "Down", "Right"]

Current State:             Goal State:
-------------             -------------
| 2 | 8 | 1 |             | 3 | 2 | 1 |
| 3 | 4 | 6 |             | 8 | 0 | 4 |
| 7 | 5 | 0 |             | 7 | 5 | 6 |
-------------             -------------
Solving using Hamming distance...
Expanded 16 nodes.
Generated 28 nodes.
Solution is ["Up", "Left", "Up", "Left", "Down", "Right"]
```

Case 3:

```
Current State:             Goal State:
-------------             -------------
| 0 | 1 | 3 |             | 1 | 2 | 3 |
| 4 | 2 | 5 |             | 4 | 5 | 6 |
| 7 | 8 | 6 |             | 7 | 8 | 0 |
-------------             -------------
Solving using Manhattan distance...
Expanded 9 nodes.
Generated 19 nodes.
Solution is ["Right", "Down", "Right", "Down"]

Current State:             Goal State:
-------------             -------------
| 0 | 1 | 3 |             | 1 | 2 | 3 |
| 4 | 2 | 5 |             | 4 | 5 | 6 |
| 7 | 8 | 6 |             | 7 | 8 | 0 |
-------------             -------------
Solving using Hamming distance...
Expanded 11 nodes.
Generated 20 nodes.
Solution is ["Right", "Down", "Right", "Down"]
```

Case 4: (No solution)

```
Current State:             Goal State:
-------------             -------------
| 0 | 3 | 1 |             | 1 | 2 | 3 |
| 4 | 2 | 5 |             | 4 | 5 | 6 |
| 7 | 8 | 6 |             | 7 | 8 | 0 |
-------------             -------------
Solving using Manhattan distance...
thread 'main' panicked at 'Reached a dead end.', src\libcore\option.rs:1038:5
note: Run with `RUST_BACKTRACE=1` environment variable to display a backtrace.
error: process didn't exit successfully: `target\debug\project_1.exe` (exit code: 101)
```

Source Code

```rust
use std::cmp::Ordering;
use std::collections::BinaryHeap;
use std::collections::HashSet;
use std::fmt;
use std::rc::Rc;

pub struct State {
    is: Vec<isize>,
    cost: isize,
    g: isize,
    h: isize,
    parent: Option<Rc<State>>,
    kind: String,
}

impl Ord for State {
    fn cmp(&self, other: &State) -> Ordering {
        other.cost.cmp(&self.cost)
            .then_with(|| self.cost.cmp(&other.cost))
    }
}

impl PartialOrd for State {
    fn partial_cmp(&self, other: &State) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}
impl PartialEq for State {
    fn eq(&self, other: &State) -> bool {
        self.cost == other.cost
    }
}
impl Eq for State {}

pub struct Problem {
    state: Rc<State>,
    goal_state: Vec<isize>,
    visited: HashSet<Vec<isize>>,
    under_consideration: BinaryHeap<Rc<State>>,
    no_generated: isize,
    no_expanded: isize,
    heuristic: String
}

impl Problem {
    pub fn new(init_state: Vec<isize>, goal_state: Vec<isize>) -> Problem {
        Problem {
            state: Rc::new(State {
                is: init_state,
                cost: 0,
                g: 0,
                h: 999,
                parent: None,
                kind: String::from("None"),
            }),
            goal_state,
            visited: HashSet::new(),
            under_consideration: BinaryHeap::new(),
            no_generated: 0,
            no_expanded: 0,
            heuristic: String::from("Manhattan")
        }
    }
    pub fn expand(&mut self) -> Vec<Rc<State>> {
        let loc = self.state.is.iter()
            .position(|&x| x == 0)
            .expect("Cannot find the blank in current state.");

        let mut possible_states = Vec::with_capacity(4);
        self.no_expanded += 1; // incrementing
        macro_rules! do_the_needful {
            ($new_loc:expr, $kind:expr) => {
                let mut new_state = self.state.is.clone();
                new_state.swap(loc, $new_loc);

                if !self.visited.contains(&new_state) {
                    let mut cost = 0;
                    let new = vec![&self.state.is[0..3], &self.state.is[3..6], &self.state.is[6..9]].clone();
                    let goal = vec![&self.goal_state[0..3], &self.goal_state[3..6], &self.goal_state[6..9]];

                    for g in self.goal_state.iter() {
                            let (goal_x, goal_y) = find(&goal, g);
                            let (new_x, new_y) = find(&new, g);
                            cost += if self.heuristic == "Manhattan" {
                                isize::abs(goal_x - new_x) + isize::abs(goal_y - new_y)
                            } else {
                                if goal_x == new_x && goal_y == new_y {0} else {1}
                            };
                    }

                    let state = State {
                        is: new_state,
                        cost: self.state.g + 1 + cost,
                        g: self.state.g + 1,
                        h: cost,
                        parent: Some(self.state.clone()),
                        kind: String::from($kind),
                    };
                    possible_states.push(Rc::new(state));
                    self.no_generated += 1; // incrementing
                }
            };
        }

        match loc { // going up
            0...2 => (),
            i => {do_the_needful!(i - 3, "Up");}
        };

        match loc { // going down
            6...8 => (),
            i => {do_the_needful!(i + 3, "Down");}
        };

        match loc {
            0 | 3 | 6 => (), // going left
            i => {do_the_needful!(i - 1, "Left");}
        };

        match loc { // going right
            2 | 5 | 8 => (),
            i => {do_the_needful!(i + 1, "Right");}
        };

        possible_states // returning the vector
    }

    pub fn solve(&mut self, heuristic: &str) {
        println!("{:?}", self);
        self.heuristic = String::from(heuristic);
        println!("Solving using {} distance...", self.heuristic);

        loop {
            let possible_states = self.expand();

            self.under_consideration
                .append(&mut BinaryHeap::from(possible_states));
            let next_state = self.under_consideration.pop().expect("Reached a dead end.");
            self.visited.insert(next_state.is.clone());
            self.state = next_state;
            if self.state.h == 0 {
                println!(
                    "Expanded {} nodes.\nGenerated {} nodes.",
                    self.no_expanded, self.no_generated
                );
                break;
            }
        }
        self.trace_soln();
    }

    pub fn trace_soln(&self) {
        let mut soln: Vec<String> = vec![];
        let mut parent = self.state.parent.clone();

        while let Some(p) = parent {
            soln.push((*p).kind.clone());
            parent = (*p).parent.clone();
        }

        soln.reverse();
        println!("Solution is {:?}", &soln[1..]);
    }
}

fn main() {
    let mut problem_1 = Problem::new(
        vec![1, 2, 3, 7, 4, 5, 6, 8, 0],
        vec![1, 2, 3, 8, 6, 4, 7, 5, 0],
    );

    println!("\n\nProblem 1");
    problem_1.solve("Manhattan");

    let mut problem_1 = Problem::new(
        vec![1, 2, 3, 7, 4, 5, 6, 8, 0],
        vec![1, 2, 3, 8, 6, 4, 7, 5, 0],
    );

    problem_1.solve("Hamming");

    let mut problem_2 = Problem::new(
        vec![2, 8, 1, 3, 4, 6, 7, 5, 0],
        vec![3, 2, 1, 8, 0, 4, 7, 5, 6],
    );

    println!("\n\nProblem 2");
    problem_2.solve("Manhattan");

    let mut problem_2 = Problem::new(
        vec![2, 8, 1, 3, 4, 6, 7, 5, 0],
        vec![3, 2, 1, 8, 0, 4, 7, 5, 6],
    );

    problem_2.solve("Hamming");

    let mut problem_3 = Problem::new(
        vec![0, 1, 3, 4, 2, 5, 7, 8, 6],
        vec![1, 2, 3, 4, 5, 6, 7, 8, 0],
    );

    println!("\n\nProblem 3");
    problem_3.solve("Manhattan");

    let mut problem_3 = Problem::new(
        vec![0, 1, 3, 4, 2, 5, 7, 8, 6],
        vec![1, 2, 3, 4, 5, 6, 7, 8, 0],
    );

    problem_3.solve("Hamming");

    let mut problem_4 = Problem::new(
        vec![0, 3, 1, 4, 2, 5, 7, 8, 6],
        vec![1, 2, 3, 4, 5, 6, 7, 8, 0],
    );

    println!("\n\nProblem 4");
    problem_4.solve("Manhattan");

    let mut problem_4 = Problem::new(
        vec![0, 3, 1, 4, 2, 5, 7, 8, 6],
        vec![1, 2, 3, 4, 5, 6, 7, 8, 0],
    );

    problem_4.solve("Hamming");
}

pub fn find(_vec: &Vec<&[isize]>, r: &isize) -> (isize, isize) {
    let mut x = 0;
    let mut y = 0;
    for (i, row) in _vec.iter().enumerate() {
        if row.contains(&r) {
            x = i;
        }
        for (j, val) in row.iter().enumerate() {
            if *val == *r {
                y = j;
            }
        }
    }
    (x as isize, y as isize)
}
```