EECS 391: P1: The eight-puzzle

To run the REPL:

```
> cargo build --release
> REM (Or Unix equivalent)
> target\release\jas497-p1.exe
```

While `cargo run` builds faster, this will run faster.

On the first run, this will download the necessary crates (Rust packages) and compile everything. The executable also takes a filename as an argument.

Code design

The codebase is organized as a typical Rust project, split into `puzzle.rs` and `main.rs` (splitting into "library" and "interface", repecstively). The interface handles file I/O, the REPL, and converting opcodes into function calls. The library handles the algorithms and opcode argument processing. Simple, definition-level functions like `h1()` and `h2()` were top-level/static, while the State and Node structs and their traits handle the rest. The important functions are named `solve_beam()` and `solve_astar()` and are called by the public API function `solve()`. None of the solvers can mutate the puzzle state, so it is safe to call them one after another without a `setState` in between.

Development setup

- Windows 10 Anniversary Update 64-bit
- Rust and Cargo
- Emacs for Windows (`rust-mode`)

Code correctness

The code was primarily tested with the REPL, although a file-based approach is also available (and tested). To make this process less finger-intensive, abbreviations were defined for common opcodes; to find them, `cd jas497-p1/src` and `$ grep -n '" | "' *.rs`; for example, `printState` was aliased to `p`. The RNG is preseeded with `RNG_SEED` in the library file; for different results, change it to a different array of four integers and `$ cargo run`. Alternatively, re-work the code to use the default system-seeded RNG.

The static helper methods (h1 et al.) were tested with the Rust playground. Here is an example session:

```rust
/* copied from source */
const SIDE_LENGTH: usize = 3;
const GOAL_STATE: [char; SIDE_LENGTH * SIDE_LENGTH] = ['b','1','2', '3','4','5', '6','7','8'
/// always against the goal
/// might be useful: `let goal_state = GOAL_STATE.to_vec();`
fn h2(goal_state: &Vec<char>, other: &Vec<char>) -> u32 {
    let mut out: u32 = 0;
    for (other_pos, other_c) in other.iter().enumerate() {
        if *other_c == 'b' { continue; }
        let other_pos = other_pos as i32;
        let goal_pos = goal_state.iter().position(|&c| c == *other_c).unwrap() as i32;
        let goal_x = goal_pos % SIDE_LENGTH as i32;
        let goal_y = goal_pos / SIDE_LENGTH as i32;
        let other_x = other_pos % SIDE_LENGTH as i32;
        let other_y = other_pos / SIDE_LENGTH as i32;
        let l1_norm = ((other_x - goal_x).abs() + (other_y - goal_y).abs()) as u32;
        out += l1_norm;
    }
    out
}
/* end of copied from source */

fn main() {
    let goal_state = GOAL_STATE.to_vec();
    let cost2  = h2(&goal_state, &vec!['1','2','b', '3','4','5', '6','7','8']); // trivial
    let cost18 = h2(&goal_state, &vec!['7','2','4', '5','b','6', '8','3','1']); // from boo
    println!("Hello, world! {}, {}", cost2, cost18); // Hello, world! 2, 18
}
```

Experiments

Fraction solvable as a function of maxNodes

This used `experiment-a.txt`. FIXME still running

A* heuristic choice

Speed: h1 was as fast to compute as h2, with no clear winner either way (using `cargo build --release`).

Logically, h2 should be better than h1 since it counts in the same units as the weight (both in valid moves) (analogous to usual distance-based A* heuristic using L2 norm for arbitrary (or at least eight-connected) planar motion).

According to the textbook, h2 is indeed better than h1. However, as of this writing, my testing has not found a case where either has succeeded where the other has failed.

Solution length

Time: beam search usually terminates faster than the other two, which run in about the same amount of time.

Moves: So far, when solutions have been found, they have always been the same between all three methods. There may be fewer shortcuts in this search space than expected.

Fraction solvable

Theoretically, all puzzles are solvable, since `randomizeState` is fully reversible. In practice, no start state was found that one could solve that another could not.

Discussion

Which algorithm is better suited? Which shorter? Which superior?

If one always runs `beam` with large beam size values, then it is slowest, and thus least well suited to the task.

Other observations

I had never written anything Rust before doing this project. The learning curve for Rust is the reason why this is late. It is also not as idiomatic as it could be; for example, `h1()` could probably use `.fold()` and `set_state()` should be a one-liner (compiler should understand, though).

Regarding the algorithms, neither was especially difficult, although Rust's notion of ownership and lifetimes did slow the process somewhat.

Cache of runs

Trivial example

This one is here to show basic functionality and a nearly-instantaneous (18ms) solution. For comparison, h1 took 5ms to find the same path, while `beam 30` took 17ms and ‘beam 11 took 5ms.

```
Hello, 391 grader!
P1> r 23
P1> p
425
13
678

P1> solve a h2
Goal!  Moves: 7
Directions to move the blank tile (invert for physical tile):
up
left
left
down
right
up
left
Took: PT0.017588008S seconds.
```

What if the blank were in the other corner? (solution may or may not exist)

As it turns out, it does.

```
P1> setState 12345678b
P1> printState
123
456
78

P1> solve A-star h2
Directions to move the blank tile (invert for physical tile):
Goal!  Moves: 22
left
up
right
up
left
left
down
right
right
up
left
down
```

```
down
left
up
up
right
right
down
left
left
up
```

Horribly scrambled case

Even Rust took multiple minutes to run this one. Clearly, either this implementation is sub-optimal or the problem really is this hard.

```
P1> p
847
 25
613
P1> solve a h2
Goal!  Moves: 25
Directions to move the blank tile (invert for physical tile):
up
right
down
down
left
up
right
down
right
up
up
left
left
down
right
down
left
up
right
right
down
left
```

```
up
up
left
```