Intelligent Systems - Coursework 1

Alexander Sadler - aes1g15@soton.ac.uk - 27621383

November 2016

1 Approach

I went through several iterations of writing and redesigning the objects in my code, attempting to find a sweet spot of efficiency and genericness and taking into account things that could be made more generic so that I could re-use code as much as possible between the different search algorithms. I will explain my final approach.

I wrote my code using Rust (`https://doc.rust-lang.org/book/getting-started.html`). Once Rust is installed, the code can be run by executing the command `cargo run --release`.

World

I firstly wrote a representation of a single Blocksworld puzzle board state (*World* struct). *Block*s and the *Agent* (Henceforth referred to as *Entities*, and implemented as such in code) each map to their *Location* in a *Bidirectional Map* (A relation I suppose). One might think that to store the Blocksworld puzzle, a 2D array would be ideal; I originally implemented it this way but found a Bidirectional map to be more space efficient and quicker to look up by Entity. I also tried a Hash Map, however it used up a surprisingly large amount of memory.

The *World* also knows its width and height.

World contains a reasonable number of methods, I shall describe one of the most interesting; *clone_and_move_agent()*. Within a search, each *Node* must store a pointer to the state of the Blocksworld board and many of these will be unique. Therefore every time a new *Node* is created, its state must be cloned. Each new *Node* will need a cloned World with the Agent moved by one square in one of the four directions. This method takes a World and a direction and returns a clone with the Agent Entity moved in that direction. If the new Agent location is out of the World bounds, an Error is raised, if another Entity resides there, the two Entities are swapped.

Searcher

Searcher is a trait (pretty much an abstract class). It contains all of the code (hopefully) that is common between the different search algorithms and specifies an interface that they all must implement.

The *search()* method in Searcher orchestrates most of the searching. Generally, it initialises the fringe with the root element and then loops, popping Nodes from the fringe and adding their children in a random order onto the fringe until it has found the goal. How an element is pushed and popped to and from the fringe is not specified so that each Search algorithm can use its own data structure. The nodes are added randomly because without this feature, depth first is less likely to find a solution quickly, as it will get into loops, going in the same direction over and over again.

The trait also has a method to check if a Node contains the goal state - it calls a method on World which compares two Worlds for equality, ignoring the agent.

Node

Node is also a trait - A* uses a different Node implementation to the other search algorithms. Searcher is generic over these.

All nodes must know their parent, the world state they contain and their depth within the search tree.

Breadth First Searcher

The effort of making searching generic pays off in the implementation of breadth first search. It implements the Searcher trait, stores the start and goal Worlds and uses a VecDeque to store the fringe. VecDeque is a double ended queue internally using a vector.

Breadth first can be simply modelled using a first in first out queue, so the *fringe_push()* and *fringe_pop()* methods push to the back and pop from the front of the fringe respectively.

Depth First Searcher

Depth first search is implemented in almost exactly the same way as breadth first except that it uses a last in first out queue, also using VecDeque but pushing to the back and popping from the back (a stack, essentially).

Iterative Deepening Searcher

Iterative deepening is also relatively easy to implement having implemented the others because it is essentially a repeated depth first search with an increasing depth limit until the solution is found. There is some slightly more complicated logic to ensure that the expanded nodes count is the sum of each iteration.

A* Searcher

Finally, A* Search. This algorithm required quite a lot more work to work with the Searcher trait. It uses a priority queue (specifically a Binary Heap) to store the fringe, so that the "best Node" is expanded each time.

Each Node in A* needs to store its distance away from the start and its heuristic. The distance away from the start is just the distance of the Node's parent from the start plus one, however the heuristic is a little more complicated. The *heuristic()* method calculates this value by measuring the distance between each block's location in the node's state and the same block's location in the goal state.

The priority queue sorts each node by its start to self cost + its heuristic, with the node with the lowest cost being the next to come off the list.

An extra which I was surprised to find causes A* to expand considerably less Nodes is checking if a Node's state is already on the fringe and if it is, checking if that Node has a lower start to node cost. If both conditions occur, the node is not added to the fringe as their is already a better way of reaching that state. The priority should already perform this sorting and I would be interested to see if the cost of searching the fringe for Nodes with the same state but lower costs is actually worth it.

2 Evidence

I shall now provide evidence of each search method working. For each of them, I will go through an extremely simple example in full detail, explaining what happens to the fringe at every step and I will show the solution found for a more complicated problem.

The first problem will have an optimal solution at depth 2. I will show the full search tree and the order in which nodes are expanded and checked. The goal will be highlighted in green. '...' indicates that there is a node, but it is not necessary to know what it is and it has been omitted as it will not fit in my diagrams.

The second problem will be the problem given in the coursework specification, which has an optimal solution at depth 14. I will show just the path that the search algorithm has found, as the search tree is to large to include.

'@' indicates the location of the agent and '*' is a wall (the agent cannot move here).

```
* * * * * *
*         *
*   A     *
*   @     *
*   B C   *
* * * * * *
```

Figure 1: The first problem, solution depth 2

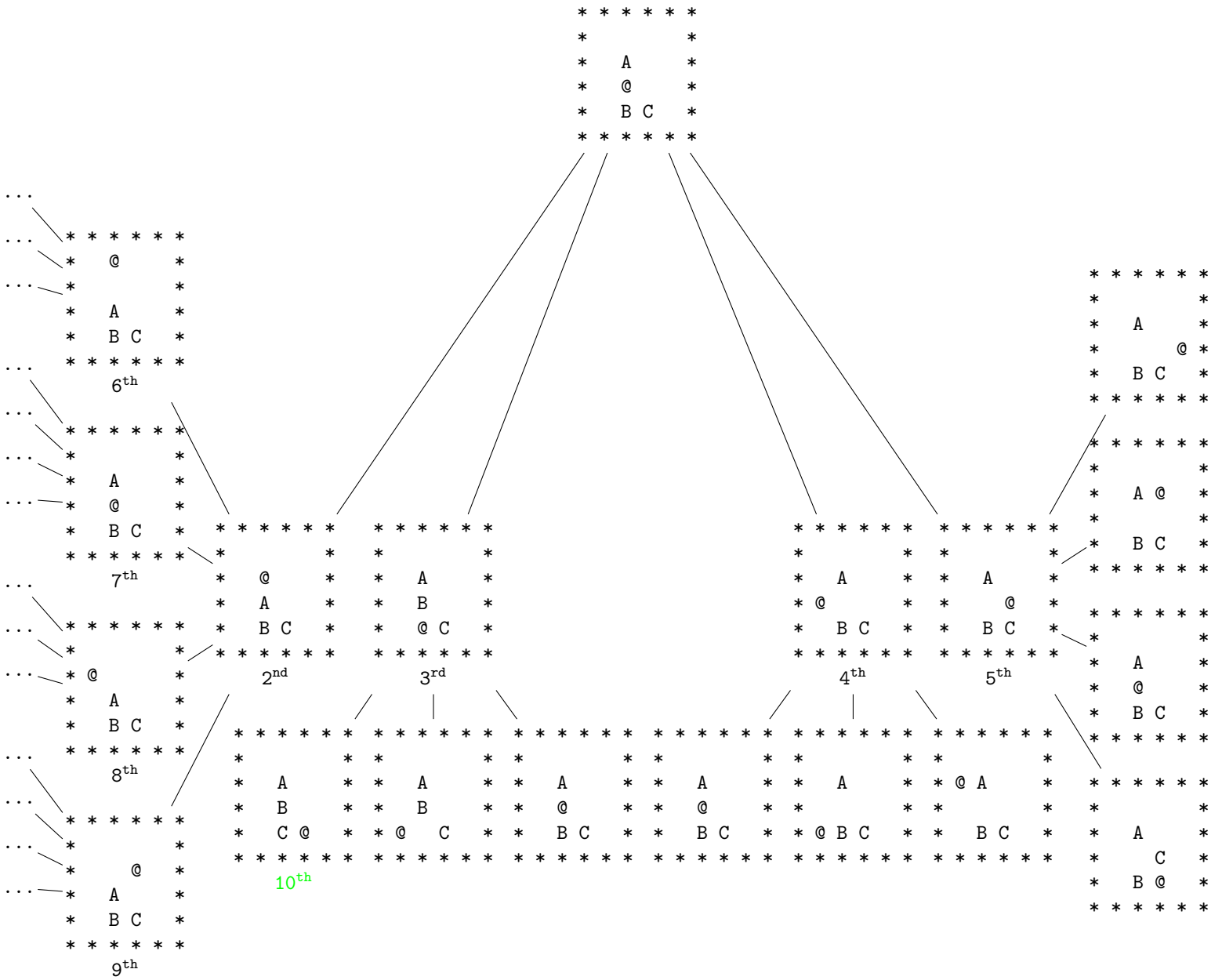
```
* * * * * *
*         *
*         *
*         *
* A B C @ *
* * * * * *
```

Figure 2: The second problem, solution depth 14

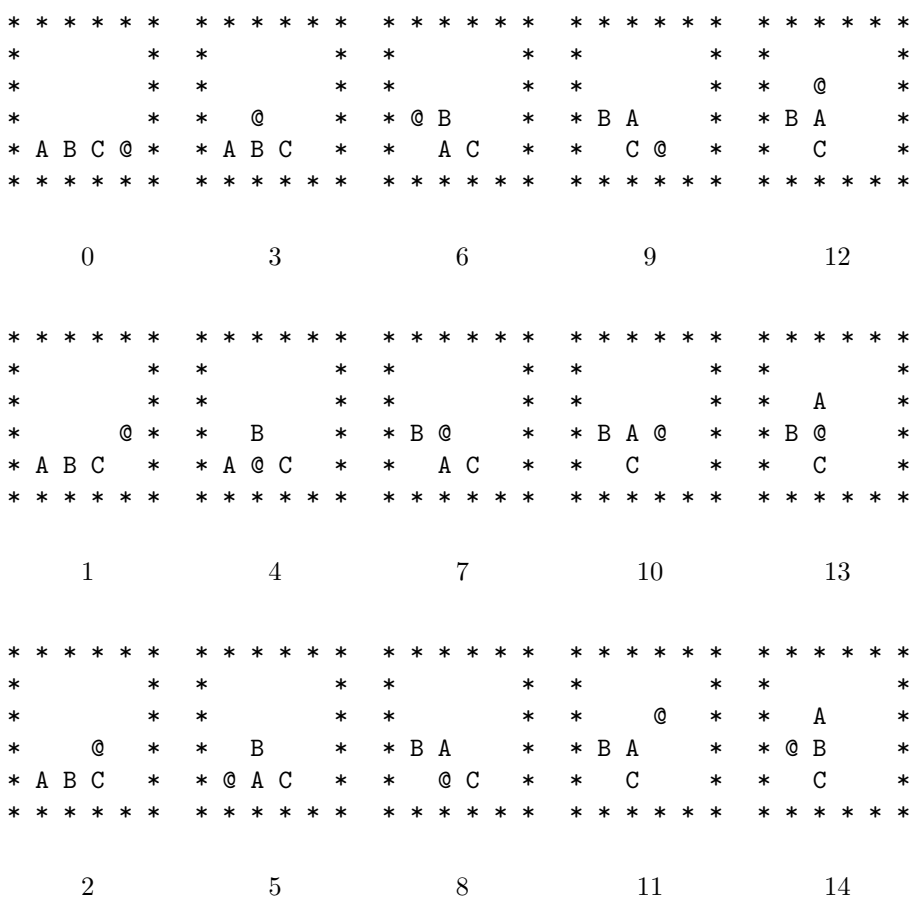
2.1 Breadth First Search

Breadth First Search will expand and check all of the nodes on each depth before moving onto the next depth. Nodes are expanded in the order in which they are added to the fringe.

2.1.1 Problem 1


6th
7th
2nd
3rd
4th
5th
8th
9th
10th

2.1.2 Problem 2



Expanded Nodes: 8919043

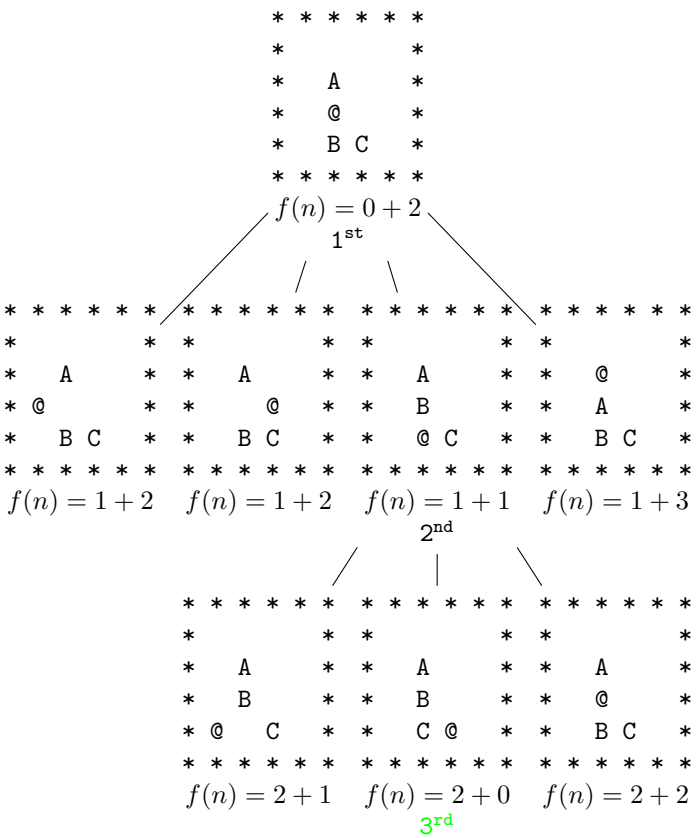
2.2 A* Search

A* search will expand the next lowest cost node from the fringe (stored in a priority queue). The cost is determined by the function $f(n) = g(n) + h(n)$, where $g(n)$ is the exact cost to reach the node n from the start node (coincidentally equivalent to the depth of the node with this problem), $h(n)$ is an estimated cost to reach the goal from this node (a heuristic function) and $f(n)$ is then the estimated cost to reach the goal going via this node.

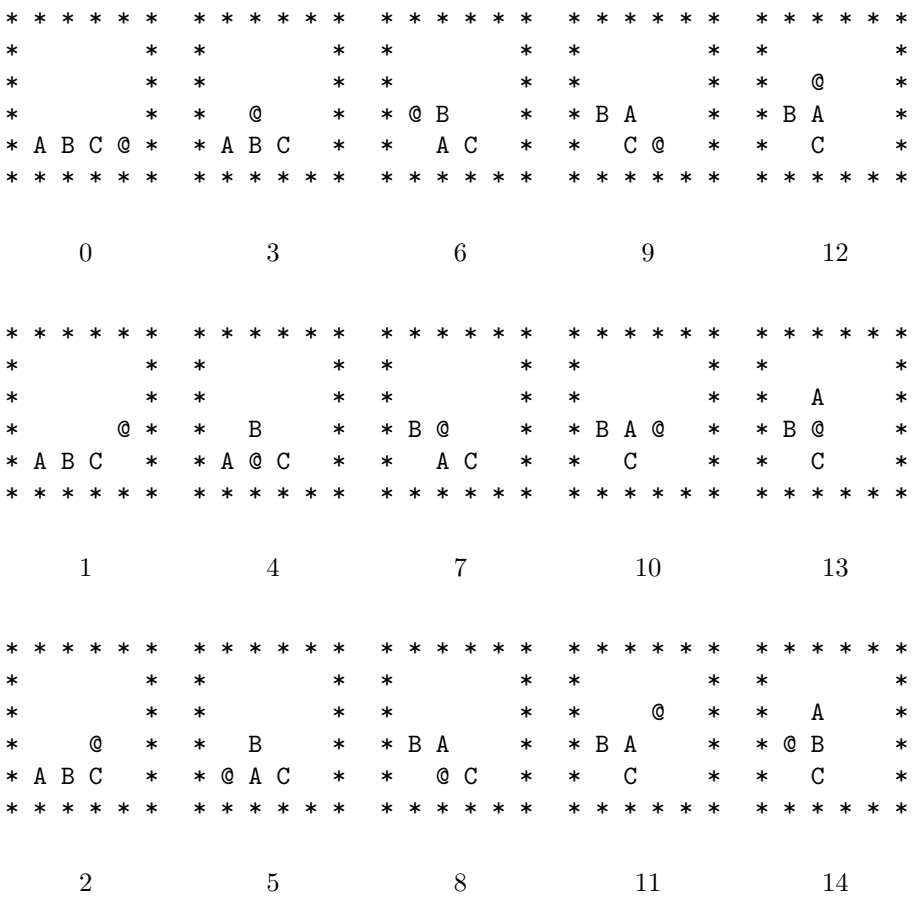
The heuristic used in this implementation is the Manhattan distance (the distance calculated by following the grid lines, the horizontal and vertical difference between the node and the goal state). Another possible heuristic would be to use the straight line distance, calculated using Pythagoras' theorem, however this is less accurate to the problem; we cannot move the agent in a straight line in all directions.

I have written which nodes were expanded and in what order (indicated by 1^{st} etc.) and the estimated cost to reach the goal via each node next to each node.

2.2.1 Problem 1



2.2.2 Problem 2



Expanded Nodes: 496

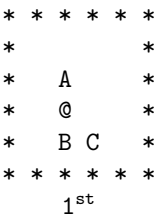
2.3 Iterative Deepening

Iterative Deepening Search sets a maximum depth that it will create nodes up to and then, starting at a maximum depth of zero, increases that maximum until it finds the solution. The nodes are created by expanding the deepest node (specifically the last node added to the fringe).

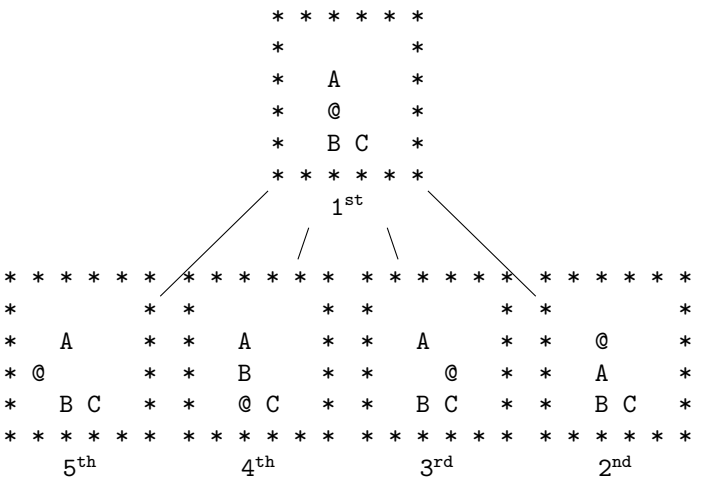
I have shown each iteration of the algorithm (It. 0 etc.) along with a number indicating the order in which the nodes were checked and (not always) expanded.

2.3.1 Problem 1

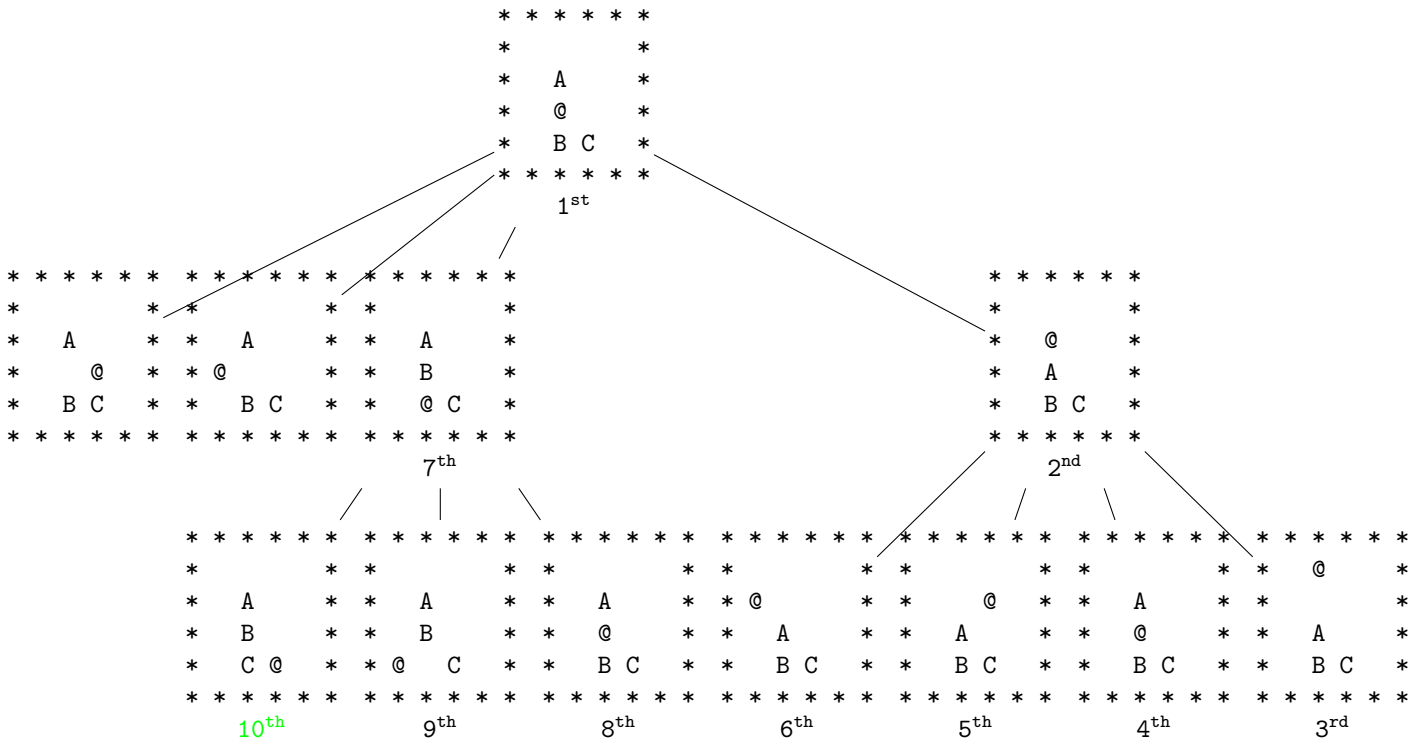
It. 0



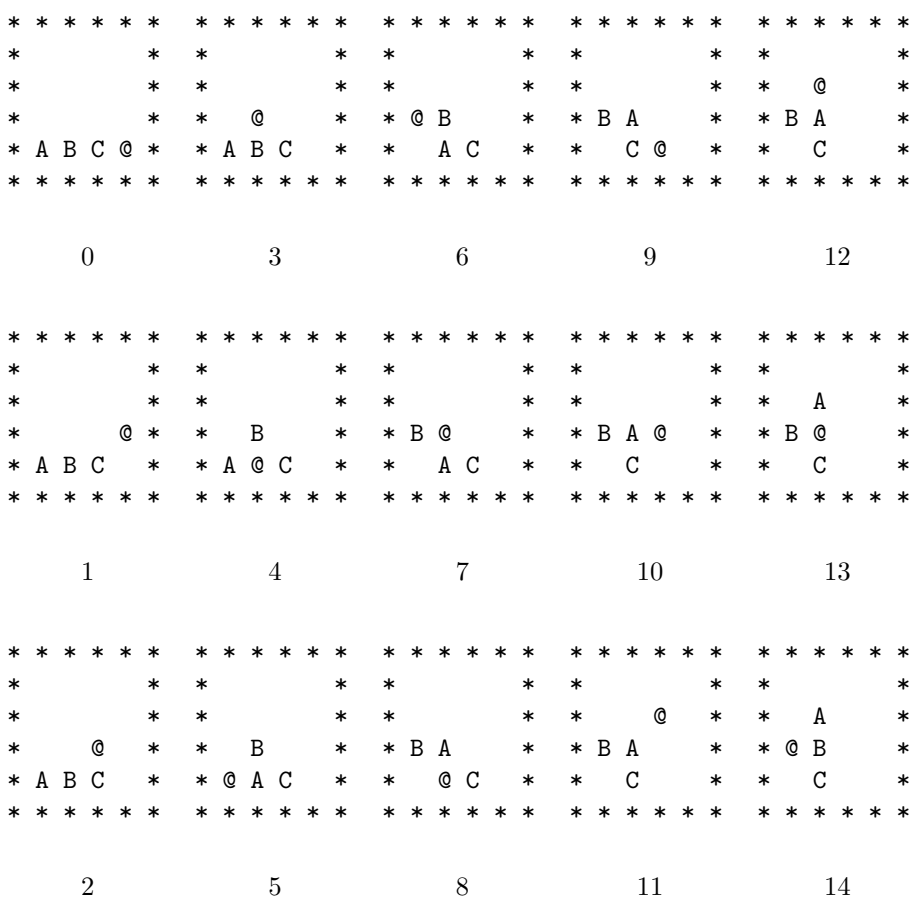
It. 1



It. 2



2.3.2 Problem 2

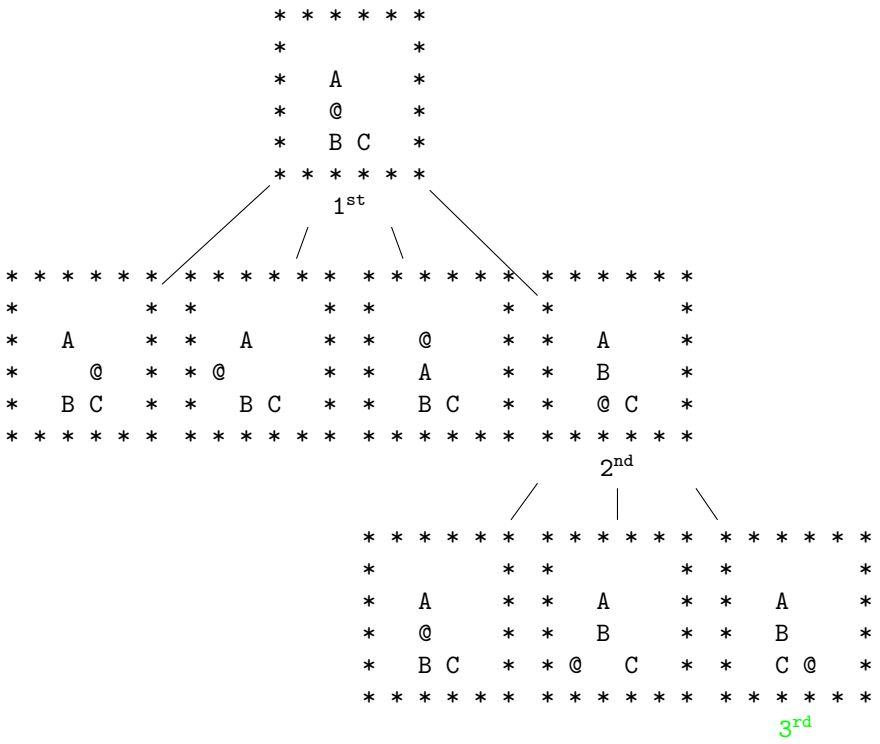


Expanded Nodes: 103534

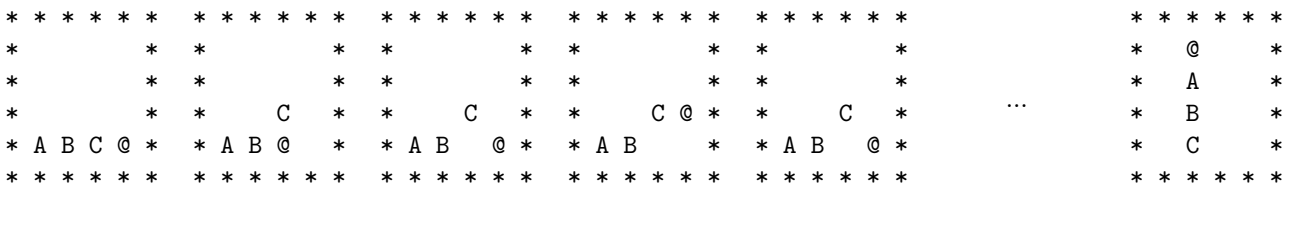
2.4 Depth First

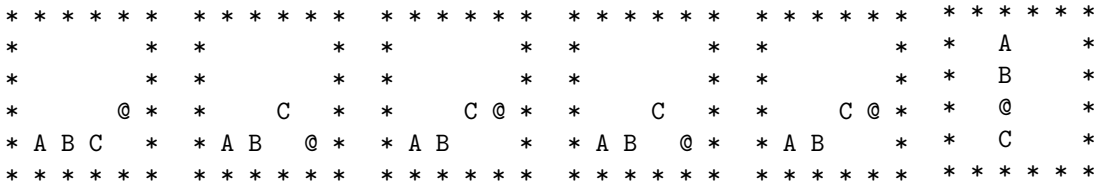
Depth first search keeps expanding and checking the deepest node (last added) in the fringe until it finds the solution.

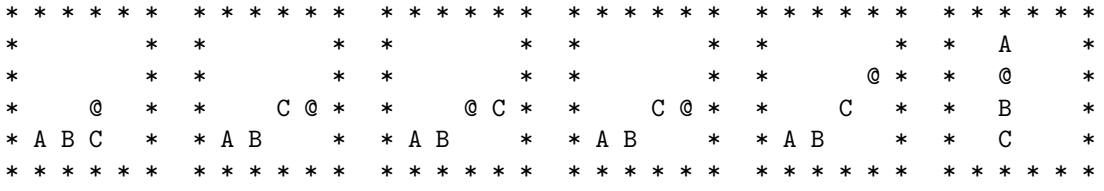
2.4.1 Problem 1



2.4.2 Problem 2







Expanded Nodes: 41785

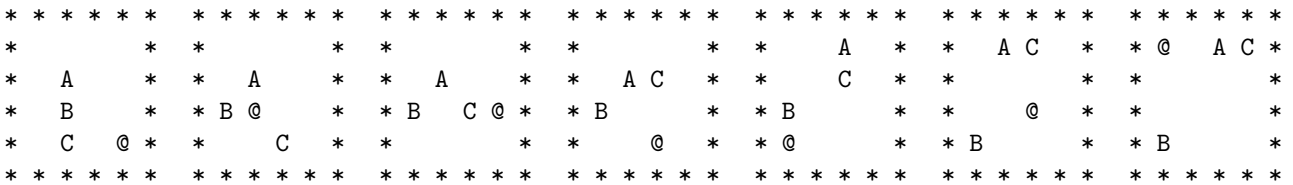
3 Scalability Study

To examine the scalability of the different search algorithms I decided to run them on problems where the optimal solution was at different depths.

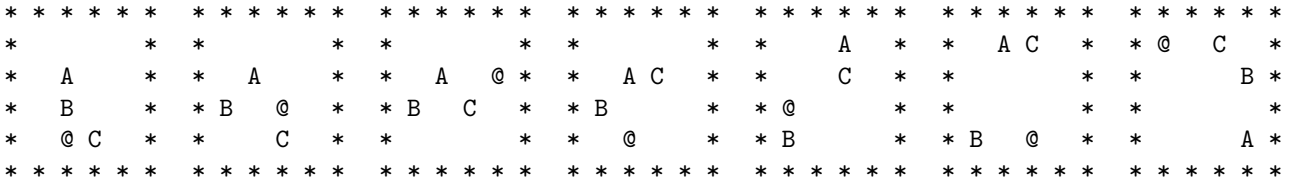
Instead of generating these manually, I used the searcher structs I had created in the first part to search for starting states that fit my criteria. It works by using a depth first search starting at the goal state (all problems finish at the same goal state, the given one), which will expand nodes and search until it has found one for each solution depth, up to depth 26, which I found to be the hardest I could automatically generate using this method.

To determine how deep the goal was, on the optimal path, from each node as a starting position, I ran an A* searcher which would find the optimal solution in a relatively fast time.

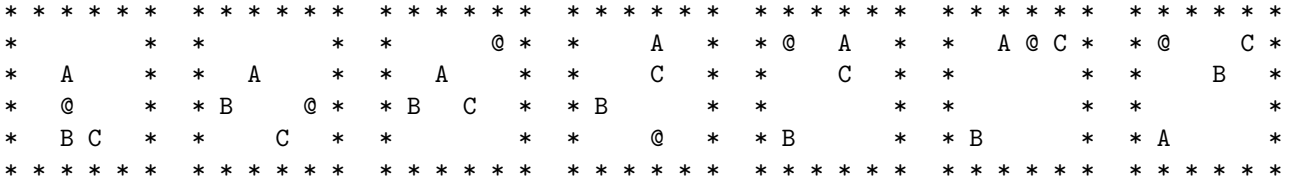
Below I give the problems generated by this algorithm and three graphs showing the time taken (nodes expanded) for all four search algorithms on each problem.



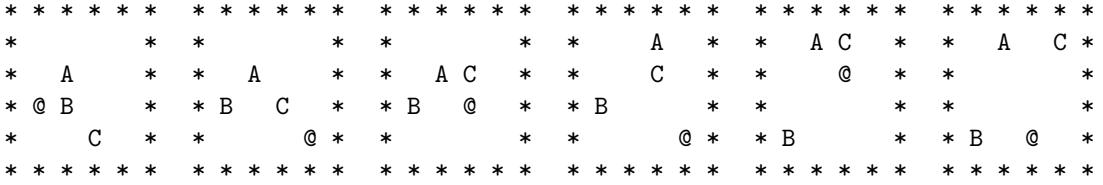
Difficulty: 0 Difficulty: 4 Difficulty: 8 Difficulty: 12 Difficulty: 16 Difficulty: 20 Difficulty: 24



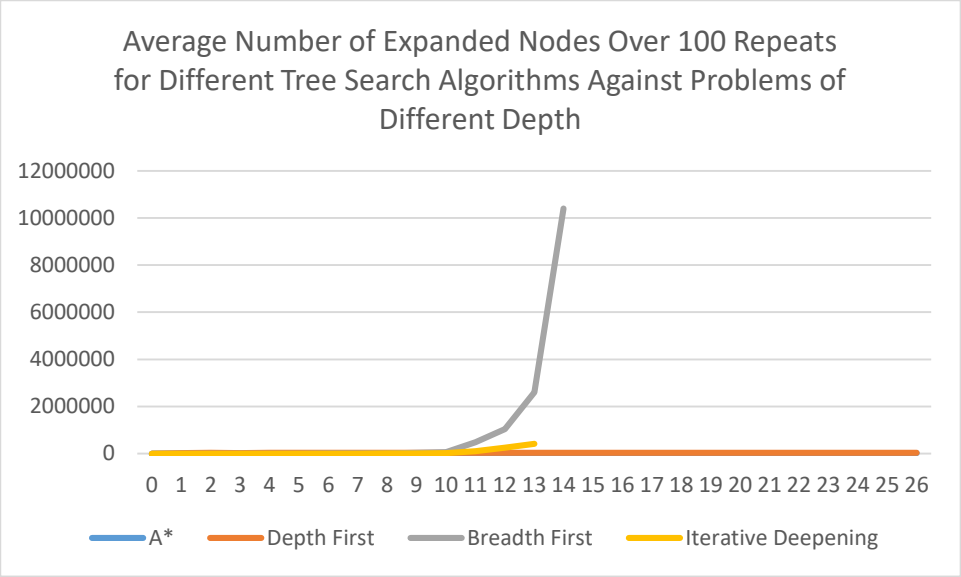
Difficulty: 1 Difficulty: 5 Difficulty: 9 Difficulty: 13 Difficulty: 17 Difficulty: 21 Difficulty: 25

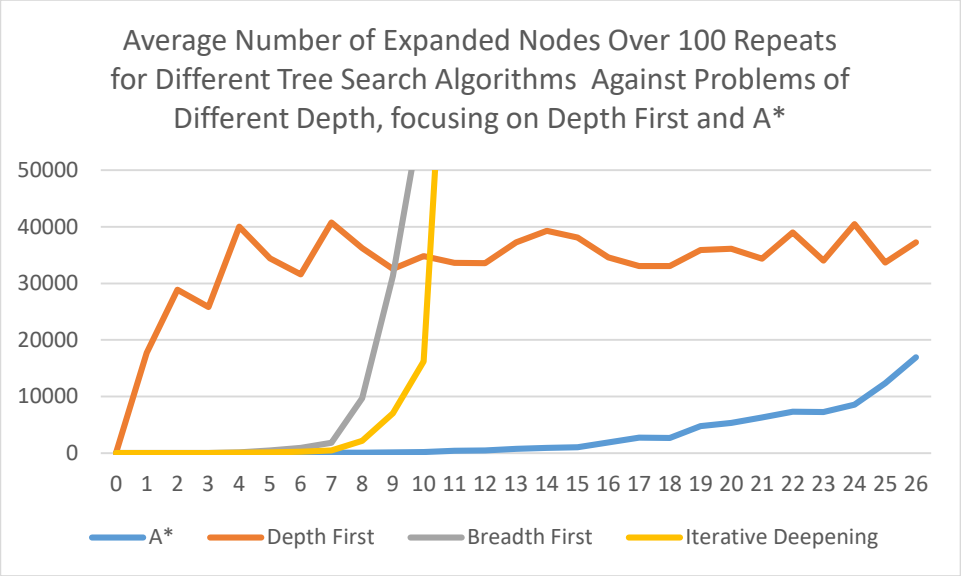


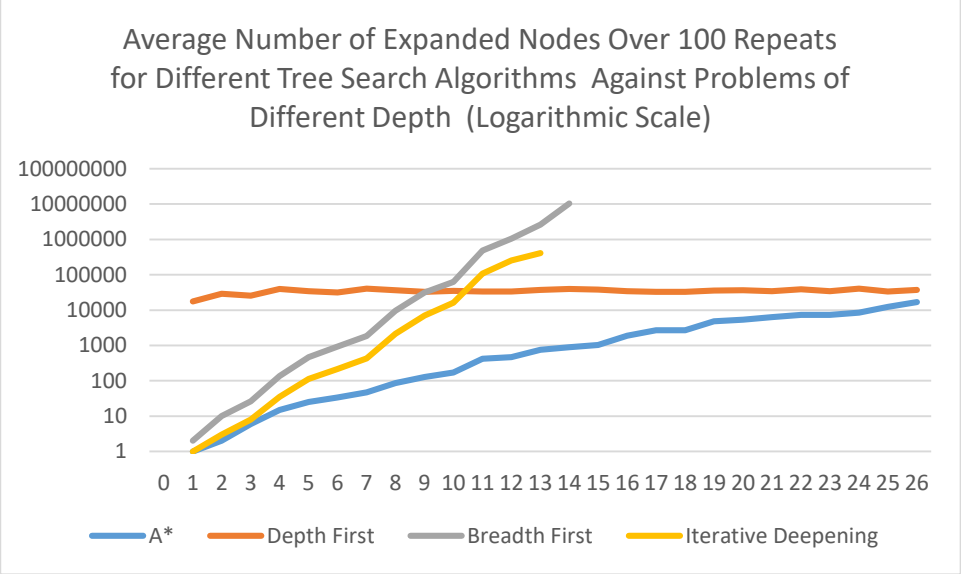
Difficulty: 2 Difficulty: 6 Difficulty: 10 Difficulty: 14 Difficulty: 18 Difficulty: 22 Difficulty: 26



Difficulty: 3 Difficulty: 7 Difficulty: 11 Difficulty: 15 Difficulty: 19 Difficulty: 23


Average Number of Expanded Nodes Over 100 Repeats for Different Tree Search Algorithms Against Problems of Different Depth
12000000
10000000
8000000
6000000
4000000
2000000
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
A*
Depth First
Breadth First
Iterative Deepening


Average Number of Expanded Nodes Over 100 Repeats for Different Tree Search Algorithms Against Problems of Different Depth, focusing on Depth First and A*
50000
40000
30000
20000
10000
0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
A*
Depth First
Breadth First
Iterative Deepening


Average Number of Expanded Nodes Over 100 Repeats for Different Tree Search Algorithms Against Problems of Different Depth (Logarithmic Scale)
100000000
10000000
1000000
100000
10000
1000
100
10
1
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
A*
Depth First
Breadth First
Iterative Deepening

3.1

We can see from the first graph that at difficulties of around 10, breadth first search and iterative deepening start expanding a lot more nodes than the other two algorithms. Because of this, I decided to create another graph with a shorter y axis to show how A* and Depth first perform. The final graph has a logarithmic scale, which is extremely useful as it allows us to compare all of the algorithms and it also shows the exponential nature of some of them. I will now compare and analyse each algorithm, comparing the worst with the next worst and so on until I am left with the best algorithm (at least for this problem definition).

Breadth First and Iterative Deepening

On the final graph, both of these algorithms follow each other quite closely indicating that iterative deepening will only be able to solve slightly more difficult problems than breadth first within the same time. One thing to note that these graphs are not showing is how many nodes are being stored at any one time in the fringe. Iterative deepening does not store the whole tree as it iterates down, only the current branch (roughly).

All in all this shows that iterative deepening, although generally only slightly better than breadth first with respect to time complexity, performs much better in terms of space complexity.

Winner: Iterative Deepening.

Iterative Deepening and Depth First

Disregarding the problem of 0 difficulty, looking at the last two graphs it is clear to see that within the first 8 or so problems, iterative deepening is far superior to depth first in its number of expanded nodes, however past that point the amount of expanded nodes for iterative deepening balloons exponentially.

What is not shown on these graphs though, is that depth first is never finding the optimal solution (it is possible, but increasingly unlikely as solution depth increases). Again though, it is not so clear cut; iterative deepening is only able to run on the computer I used for testing up to about difficulty 14 in a reasonable amount of time and memory. Therefore if we can overlook the quite substantial problem of not finding the optimal solution, depth first is better.

Winner: Depth First.

Depth First and A*

No matter how we look at A* it's obvious that it is superior to all of the other algorithms in all ways. Its time complexity is excellent, its space complexity is similarly great. The only problem I can see is that the time complexity is starting to approach that of depth first in the final difficulties. It is likely that for hard problems (and I found this when choosing a search algorithm to look for problems of specific difficulties) Depth first will be quicker than A*, this may be useful when an optimal solution is not required, however for this problem and many others A* is the ideal solution.

Winner: A*.

4 Extras and Limitations

If I had more time, I would like to make the searches generic over the problem definition so that Nodes could contain anything (i.e. not just limited to Blocksworld Worlds) and different types of Nodes would be used for different search problems (Blocksworld, 8-Puzzles, etc.). Node implementations would define methods to generate their own children and how to check if they are the goal Node. Each search type could define it's own trait that a Node must implement to be used with that search algorithm, e.g A* would specify that the Node must implement a heuristic function and a cost function.

I previously mentioned that I used my searching code to solve a different problem; how to create new and difficult search problems so that I could compare the different algorithms in the scalability survey. I tried a few ways of generating them, but settled on a depth first which only stores the last node added in the fringe, which checks every node using A* to find how deep its optimal solution is. This was surprisingly able to find much hard problems than replacing the depth first with a greedy best first search. It demonstrates that on small problems and where it is hard to find a good prediction for future costs, randomness works very well.

If allowed to use graph search I think it would be possible to create a speed up in certain areas at the expense of memory. A* search is not in danger of using up too much memory on such a small problem and when I tested it, I found that it expanded 300 nodes when using graph search versus 500 nodes with tree search. It would be interesting the effect this has on the other algorithms.

5 Code

5.1 World

```
use bidir_map::BidirMap;

use std::iter;

#[derive(Debug, Clone, PartialEq)]
pub struct World {
    pub entities: BidirMap<Entity, Location>,
    width: isize,
    height: isize,
}

impl World {
    pub fn new(width: usize,
               height: usize,
               entity_starts: &BidirMap<Entity, Location>)
               -> Result<World, WorldError> {
        let width = width as isize;
        let height = height as isize;
        World::check_start_invariants(width, height, entity_starts)?;

        Ok(World {
            entities: entity_starts.clone(),
            width: width,
            height: height,
        })
    }
    pub fn pretty_print(&self) {
        let wall_char = '*';
        let agent_char = '@';
        let none_char = ' ';
        let padding_char = ' ';

        let horizontal_wall = iter::repeat(format!("{}{}", wall_char, padding_char))
            .take(self.width as usize + 2)
            .collect::<String>();

        println!("{}", horizontal_wall);
        for y in 0..self.height {
            print!("{}{}", wall_char, padding_char);
            for x in 0..self.width {
                match self.get_grid_location(&Location::new(x, y)).unwrap() {
                    Entity::Agent => print!("{}", agent_char),
                    Entity::Block(block_char) => print!("{}", block_char),
                    Entity::None => print!("{}", none_char),
                }
                print!("{}", padding_char);
            }
            print!("{}\n", wall_char);
        }
        println!("{}", horizontal_wall);
    }
    pub fn latex_print(&self) {
        let wall_char = '*';
        let agent_char = '@';
        let none_char = '~';
        let padding_char = '~';
```

```
    let horizontal_wall = iter::repeat(format!("{}{}", wall_char, padding_char))
        .take(self.width as usize + 2)
        .collect::<String>();

    print!("{}\\\\{}", horizontal_wall, wall_char);
    for y in 0..self.height {
        print!("{}{}", wall_char, padding_char);
        for x in 0..self.width {
            match self.get_grid_location(&Location::new(x, y)).unwrap() {
                Entity::Agent => print!("{}", agent_char),
                Entity::Block(block_char) => print!("{}", block_char),
                Entity::None => print!("{}", none_char),
            }
            print!("{}", padding_char);
        }
        print!("{}\\\\{}", wall_char, wall_char);
    }
    println!("{}\\\\", horizontal_wall);
}

pub fn clone_and_move_agent(&self, direction: &Direction) -> Result<World, WorldError> {
    let old_agent_location = self.entities.get_by_first(&Entity::Agent).unwrap();
    let new_agent_location = Location::new(old_agent_location.x +
                                           match *direction {
                                               Direction::Left => -1,
                                               Direction::Right => 1,
                                               _ => 0,
                                           },
                                           old_agent_location.y +
                                           match *direction {
                                               Direction::Up => -1,
                                               Direction::Down => 1,
                                               _ => 0,
                                           });

    Self::check_location_invariants(self.width, self.height, &new_agent_location)?;

    let mut clone_world = self.clone();

    let new_agent_location_entity = clone_world.get_grid_location(&new_agent_location).unwrap();
    match new_agent_location_entity {
        Entity::None => (),
        _ => clone_world.set_entity_location(new_agent_location_entity, old_agent_location.clone())
    }
    clone_world.set_entity_location(Entity::Agent, new_agent_location);

    Ok(clone_world)
}
pub fn get_grid_location(&self, location: &Location) -> Result<Entity, WorldError> {
    Self::check_location_invariants(self.width, self.height, location)?;
    Ok(self.entities.get_by_second(location).map(|ent| ent.clone()).unwrap_or(Entity::None))
}
pub fn get_entity_location(&self, entity: &Entity) -> Result<&Location, WorldError> {
    self.entities.get_by_first(entity).ok_or(WorldError::NonExistentEntityError)
}
pub fn set_entity_location(&mut self, entity: Entity, location: Location) {
    self.entities.insert(entity, location).unwrap();
}
pub fn eq_ignore_agent(&self, other: &World) -> bool {
    if self.width != other.width || self.height != other.height {
        return false;
```

```
        }
        if self.entities.len() != other.entities.len() {
            return false;
        }
        self.entities
            .iter()
            .filter(|&&(ref ent, _)| *ent != Entity::Agent)
            .all(|&(ref ent, ref loc)| other.entities.get_by_first(ent) == Some(loc))
    }

    fn check_location_invariants(width: isize,
                                 height: isize,
                                 location: &Location)
                                 -> Result<(), WorldError> {
        if location.x >= width || location.x < 0 || location.y >= height as isize ||
           location.y < 0 {
            return Err(WorldError::EntityOutOfBoundsError);
        }

        Ok(())
    }
    fn check_start_invariants(grid_width: isize,
                              grid_height: isize,
                              entity_starts: &BidirMap<Entity, Location>)
                              -> Result<(), WorldError> {
        let mut agent_count: u8 = 0;
        for &(ref entity, ref location) in entity_starts.iter() {
            match *entity {
                Entity::Agent => agent_count += 1,
                _ => (),
            }
            Self::check_location_invariants(grid_width, grid_height, &location)?;
            if agent_count > 1 {
                return Err(WorldError::InvalidNumberOfAgentsError);
            }
        }
        if agent_count == 0 {
            return Err(WorldError::InvalidNumberOfAgentsError);
        }
        // Todo: Do not allow multiple Entity s to exist in same location.
        Ok(())
    }
}


#[derive(Clone, PartialEq, Hash, Eq, Debug)]
pub enum Entity {
    Agent,
    Block(char),
    None,
}

#[derive(Clone, Debug, Hash, PartialEq)]
pub struct Location {
    x: isize,
    y: isize,
}
impl Location {
    pub fn new(x: isize, y: isize) -> Location {
        Location { x: x, y: y }
    }
```

```
    pub fn distance_to(&self, other: &Location) -> usize {
        ((self.x - other.x).abs() + (self.y - other.y).abs()) as usize
    }
}

#[derive(Copy, Clone)]
pub enum Direction {
    Up,
    Down,
    Left,
    Right,
}
impl Direction {
    pub fn directions_array() -> [Direction; 4] {
        static DIRECTIONS: [Direction; 4] =
            [Direction::Up, Direction::Down, Direction::Left, Direction::Right];
        DIRECTIONS.clone()
    }
}

#[derive(Debug)]
pub enum WorldError {
    EntityOutOfBoundsError,
    InvalidNumberOfAgentsError,
    NonExistentEntityError,
}
```

5.2 Searcher

```
extern crate rand;

use ::blocksworld::world;
use std::rc::Rc;
use self::rand::{thread_rng, Rng};

mod breadth_first_searcher;
mod depth_first_searcher;
mod iterative_deepening_searcher;
mod a_star_searcher;
pub use self::breadth_first_searcher::BreadthFirstSearcher;
pub use self::depth_first_searcher::DepthFirstSearcher;
pub use self::iterative_deepening_searcher::IterativeDeepeningSearcher;
pub use self::a_star_searcher::AStarSearcher;

pub trait Searcher {
    type NodeType: Node;

    // search() performs the main loop of a search operation
    // - pushing children to the fringe and popping the next node for checking.
    // Returns either Ok(goal_node, expanded_nodes) or Err(error, expanded_nodes)
    fn search(&mut self,
              max_depth: Option<u64>)
              -> Result<(Self::NodeType, u64), (SearcherError, u64)> {
        let start_world_clone = self.get_start_world().clone();
        let root_node = self.new_node(0, Box::new(start_world_clone), None);
        self.fringe_push(root_node);

        let mut expanded_nodes = 0;
        let mut directions = world::Direction::directions_array();
        loop {
            let parent_rc = Rc::new(self.fringe_pop()
```

```
                .ok_or((SearcherError::GoalNotFoundError, expanded_nodes))?);
            if self.goal_reached(&*parent_rc) {
                return match Rc::try_unwrap(parent_rc) {
                    Ok(node) => Ok((node, expanded_nodes)),
                    Err(_) => unreachable!(),
                };
            }
            let child_depth = parent_rc.get_depth() + 1;
            match max_depth {
                Some(max_depth) => {
                    if child_depth > max_depth {
                        continue;
                    }
                }
                None => (),
            }

            thread_rng().shuffle(&mut directions); // For depth first especially, add children in a ran
            for direction in directions.iter() {
                parent_rc.get_world()
                    .clone_and_move_agent(direction)
                    .and_then(|new_world| {
                        let new_node =
                            self.new_node(child_depth,
                                          Box::new(new_world),
                                          Some(parent_rc.clone()));
                        self.fringe_push(new_node);
                        Ok(())
                    })
                    .ok();
            }

            expanded_nodes += 1
        }
    }
    fn goal_reached(&self, node: &Self::NodeType) -> bool {
        node.get_world().eq_ignore_agent(self.get_goal_world())// The agent location doesn't matter.
    }

    fn new_node(&self,
                depth: u64,
                world: Box<world::World>,
                parent: Option<Rc<Self::NodeType>>)
                -> Self::NodeType;
    fn get_start_world(&self) -> &world::World;
    fn get_goal_world(&self) -> &world::World;
    fn fringe_push(&mut self, node: Self::NodeType);
    fn fringe_pop(&mut self) -> Option<Self::NodeType>;
}

pub trait Node {
    fn get_world(&self) -> &world::World;
    fn get_depth(&self) -> u64;
    fn get_parent(&self) -> Option<Rc<Self>>;

    // Prints a node's parents and their parents etc. until it reaches the root
    fn print_tree(&self) {
        self.get_world().pretty_print();
        println!("{}", self.get_depth());
        let mut parent = self.get_parent();
        loop {
```

```
            match parent {
                Some(node_rc) => {
                    node_rc.get_world().pretty_print();
                    println!("{}", node_rc.get_depth());
                    parent = node_rc.get_parent();
                }
                None => break,
            }
        }
    }
}

// A node implementing the most basic level of features.
pub struct BasicNode {
    depth: u64,
    world: Box<world::World>,
    parent: Option<Rc<BasicNode>>,
}
impl BasicNode {
    pub fn new(depth: u64, world: Box<world::World>, parent: Option<Rc<Self>>) -> Self {
        BasicNode {
            depth: depth,
            world: world,
            parent: parent,
        }
    }
}
impl Node for BasicNode {
    fn get_world(&self) -> &world::World {
        &*self.world
    }
    fn get_depth(&self) -> u64 {
        self.depth
    }
    fn get_parent(&self) -> Option<Rc<BasicNode>> {
        match &self.parent {
            &Some(ref parent_rc) => Some(parent_rc.clone()),
            &None => None,
        }
    }
}

#[derive(Debug)]
pub enum SearcherError {
    GoalNotFoundError,
}
```

5.3 BreadthFirstSearcher

```
use std::collections::VecDeque;
use std::rc::Rc;

use super::BasicNode;
use super::Searcher;
use super::SearcherError;
use ::blocksworld::world;

pub struct BreadthFirstSearcher {
    start_world: world::World,
    goal_world: world::World,
    fringe: VecDeque<BasicNode>,
```

```
}
impl BreadthFirstSearcher {
    pub fn new(start_world: world::World, goal_world: world::World) -> BreadthFirstSearcher {
        BreadthFirstSearcher {
            start_world: start_world,
            goal_world: goal_world,
            fringe: VecDeque::new(),
        }
    }
    pub fn search(&mut self) -> Result<(BasicNode, u64), (SearcherError, u64)> {
        Searcher::search(self, None)
    }
}
impl Searcher for BreadthFirstSearcher {
    type NodeType = BasicNode;
    fn get_start_world(&self) -> &world::World {
        &self.start_world
    }
    fn get_goal_world(&self) -> &world::World {
        &self.goal_world
    }
    fn fringe_push(&mut self, node: Self::NodeType) {
        self.fringe.push_back(node);
    }
    fn fringe_pop(&mut self) -> Option<Self::NodeType> {
        self.fringe.pop_front()
    }
    fn new_node(&self,
                depth: u64,
                world: Box<world::World>,
                parent: Option<Rc<Self::NodeType>>)
                -> Self::NodeType {
        Self::NodeType::new(depth, world, parent)
    }
}
```

5.4 DepthFirstSearcher

```
use std::collections::VecDeque;
use std::rc::Rc;

use super::BasicNode;
use super::Searcher;
use super::SearcherError;
use ::blocksworld::world;

pub struct DepthFirstSearcher {
    start_world: world::World,
    goal_world: world::World,
    fringe: VecDeque<BasicNode>,
}
impl DepthFirstSearcher {
    pub fn new(start_world: world::World, goal_world: world::World) -> DepthFirstSearcher {
        DepthFirstSearcher {
            start_world: start_world,
            goal_world: goal_world,
            fringe: VecDeque::new(),
        }
    }
    pub fn search(&mut self) -> Result<(BasicNode, u64), (SearcherError, u64)> {
        Searcher::search(self, None)
```

```
    }
}
impl Searcher for DepthFirstSearcher {
    type NodeType = BasicNode;
    fn get_start_world(&self) -> &world::World {
        &self.start_world
    }
    fn get_goal_world(&self) -> &world::World {
        &self.goal_world
    }
    fn fringe_push(&mut self, node: Self::NodeType) {
        self.fringe.push_back(node);
    }
    fn fringe_pop(&mut self) -> Option<Self::NodeType> {
        self.fringe.pop_back()
    }
    fn new_node(&self,
                depth: u64,
                world: Box<world::World>,
                parent: Option<Rc<Self::NodeType>>)
                -> Self::NodeType {
        Self::NodeType::new(depth, world, parent)
    }
}
```

5.5 IterativeDeepeningSearcher

```
use std::collections::VecDeque;
use std::rc::Rc;

use super::BasicNode;
use super::Searcher;
use super::SearcherError;
use ::blocksworld::world;

pub struct IterativeDeepeningSearcher {
    start_world: world::World,
    goal_world: world::World,
    fringe: VecDeque<BasicNode>,
}
impl IterativeDeepeningSearcher {
    pub fn new(start_world: world::World, goal_world: world::World) -> IterativeDeepeningSearcher {
        IterativeDeepeningSearcher {
            start_world: start_world,
            goal_world: goal_world,
            fringe: VecDeque::new(),
        }
    }
    pub fn search(&mut self) -> Result<(BasicNode, u64), (SearcherError, u64)> {
        let mut expanded_nodes = 0;
        // Increase the max depth from zero until the goal is found
        // Sums up the expanded nodes for every iteration
        (0..)
            .map(|max_depth| {
                let search = Searcher::search(self, Some(max_depth));
                expanded_nodes += match search {
                    Ok((_, exp_nod)) => exp_nod,
                    Err((_, exp_nod)) => exp_nod,
                };
                search
            })
```

```
            .find(|result| result.is_ok())
            .unwrap()
    }
}
impl Searcher for IterativeDeepeningSearcher {
    type NodeType = BasicNode;
    fn get_start_world(&self) -> &world::World {
        &self.start_world
    }
    fn get_goal_world(&self) -> &world::World {
        &self.goal_world
    }
    fn fringe_push(&mut self, node: Self::NodeType) {
        self.fringe.push_back(node);
    }
    fn fringe_pop(&mut self) -> Option<Self::NodeType> {
        self.fringe.pop_back()
    }
    fn new_node(&self,
                depth: u64,
                world: Box<world::World>,
                parent: Option<Rc<Self::NodeType>>)
                -> Self::NodeType {
        Self::NodeType::new(depth, world, parent)
    }
}
```

5.6 AStarSearcher

```
use std::collections::BinaryHeap;
use std::cmp::Ordering;
use std::rc::Rc;

use super::Node;
use super::Searcher;
use super::SearcherError;
use ::blocksworld::world;

pub struct AStarSearcher {
    start_world: world::World,
    goal_world: world::World,
    fringe: BinaryHeap<AStarNode>,
}
impl AStarSearcher {
    pub fn new(start_world: world::World, goal_world: world::World) -> AStarSearcher {
        AStarSearcher {
            start_world: start_world,
            goal_world: goal_world,
            fringe: BinaryHeap::new(),
        }
    }
    pub fn search(&mut self) -> Result<(AStarNode, u64), (SearcherError, u64)> {
        Searcher::search(self, None)
    }

    // Calculates manhattan distance between given world and goal world for each block.
    fn heuristic(&self, world: &world::World) -> usize {
        world.entities
            .iter()
            .filter(|&&(ref ent, _)| *ent != world::Entity::Agent)
            .map(|&(ref ent, ref loc)| {
```

```
                loc.distance_to(self.get_goal_world().get_entity_location(&ent).unwrap())
            })
            .sum::<usize>()
    }

    // If a node's world state is already in the fringe,
    // and the node containing it has a higher priority, don't add this new node.
    fn is_node_unoptimal(&self, node: &AStarNode) -> bool {
        self.fringe
            .iter()
            .filter(|n| *n.get_world() == *node.get_world())
            .any(|n| node.start_to_self_cost >= n.start_to_self_cost)
    }
}
impl Searcher for AStarSearcher {
    type NodeType = AStarNode;
    fn get_start_world(&self) -> &world::World {
        &self.start_world
    }
    fn get_goal_world(&self) -> &world::World {
        &self.goal_world
    }
    fn fringe_push(&mut self, node: Self::NodeType) {
        if self.is_node_unoptimal(&node) {
            return;
        }
        self.fringe.push(node);
    }
    fn fringe_pop(&mut self) -> Option<Self::NodeType> {
        self.fringe.pop()
    }
    fn new_node(&self,
                depth: u64,
                world: Box<world::World>,
                parent: Option<Rc<Self::NodeType>>)
                -> Self::NodeType {
        let heuristic = self.heuristic(&*world);
        let start_to_self_cost = match &parent { // Each node is only 1 move away from its parent.
            &Some(ref parent_rc) => parent_rc.start_to_self_cost + 1,
            &None => 0,
        };
        AStarNode::new(depth, world, parent, start_to_self_cost, heuristic)
    }
}

#[derive(Clone)]
pub struct AStarNode {
    depth: u64,
    world: Box<world::World>,
    parent: Option<Rc<AStarNode>>,
    start_to_self_cost: usize,
    heuristic: usize,
}
impl AStarNode {
    fn new(depth: u64,
           world: Box<world::World>,
           parent: Option<Rc<Self>>,
           start_to_self_cost: usize,
           heuristic: usize)
           -> Self {
        AStarNode {
```

```
            depth: depth,
            world: world,
            parent: parent,
            start_to_self_cost: start_to_self_cost,
            heuristic: heuristic,
        }
    }
}
impl Node for AStarNode {
    fn get_world(&self) -> &world::World {
        &*self.world
    }
    fn get_depth(&self) -> u64 {
        self.depth
    }
    fn get_parent(&self) -> Option<Rc<Self>> {
        match &self.parent {
            &Some(ref parent_rc) => Some(parent_rc.clone()),
            &None => None,
        }
    }
}

// The below code sets up ordering so that the priority queue will order nodes by their f(n), minimum a
impl PartialEq for AStarNode {
    fn eq(&self, other: &AStarNode) -> bool {
        (self.start_to_self_cost + self.heuristic) == (other.start_to_self_cost + other.heuristic)
    }
}
impl Eq for AStarNode {}
impl PartialOrd for AStarNode {
    fn partial_cmp(&self, other: &AStarNode) -> Option<Ordering> {
        (other.start_to_self_cost + other.heuristic)
            .partial_cmp(&(self.start_to_self_cost + self.heuristic))
    }
}
impl Ord for AStarNode {
    fn cmp(&self, other: &AStarNode) -> Ordering {
        other.partial_cmp(self).unwrap()
    }
}
```

5.7 ProblemFinder

```
use blocksworld::world::World;
use blocksworld::search::BasicNode;
use blocksworld::search::Node;
use blocksworld::search::Searcher;

use std::collections::BTreeMap;
use std::rc::Rc;

pub fn solution_depth_difficulty(goal_world: World) -> Vec<(u8, World)> {
    let searcher = SolutionDepthSearcher::new(goal_world, 26);
    searcher.search()
}

// Uses a depth first style search to find problems of difficulties up to a certain difficulty.
// Each node generated has an A* search run on it to find its optimal path  size to the goal (difficult
pub struct SolutionDepthSearcher {
    start_world: World,
```

```
    fringe: Option<BasicNode>,
    max_difficulty: u8,
    solutions: BTreeMap<u8, World>,
}
impl SolutionDepthSearcher {
    pub fn new(start_world: World, max_difficulty: u8) -> SolutionDepthSearcher {
        SolutionDepthSearcher {
            start_world: start_world,
            fringe: None,
            max_difficulty: max_difficulty,
            solutions: BTreeMap::new(),
        }
    }
    pub fn search(mut self) -> Vec<(u8, World)> {
        let _ = Searcher::search(&mut self, None);
        self.solutions.into_iter().collect::<Vec<(u8, World)>>()
    }
}
impl Searcher for SolutionDepthSearcher {
    type NodeType = BasicNode;
    fn get_start_world(&self) -> &World {
        &self.start_world
    }
    fn get_goal_world(&self) -> &World {
        &self.start_world
    }
    fn goal_reached(&self, _: &Self::NodeType) -> bool {
        self.solutions.len() > self.max_difficulty as usize
    }
    fn fringe_push(&mut self, node: Self::NodeType) {
        self.fringe = Some(node);
    }
    fn fringe_pop(&mut self) -> Option<Self::NodeType> {
        let node = match self.fringe.take() {
            Some(node) => node,
            None => return None,
        };
        let mut a_star_searcher =
            ::blocksworld::search::AStarSearcher::new(node.get_world().clone(),
                                                      self.get_goal_world().clone());
        let result = a_star_searcher.search().unwrap();
        // If we haven't already found a problem world at this depth, add it, to the Map
        if !self.solutions.contains_key(&(result.0.get_depth() as u8)) {
            self.solutions.insert(result.0.get_depth() as u8, node.get_world().clone());
            println!("New solution, depth {}", result.0.get_depth());
        }

        Some(node)
    }
    fn new_node(&self,
                depth: u64,
                world: Box<World>,
                parent: Option<Rc<Self::NodeType>>)
                -> Self::NodeType {
        Self::NodeType::new(depth, world, parent)
    }
}
```