Problem Solving Agents and Uninformed Search

An intelligent agents act to increase their performance measure. Some do this by adopting a goal.

Four general steps in problem solving:

Goal formulation – deciding on what the goal states are
- based on current situation and agent's performance measure
- What are the successful world states

Problem formulation -
- how can we get to the goal, without getting bogged down in the detail of the world. Walking robot - not concerned about moving one inch ahead, but, in general concerned about moving ahead.
- What actions and states to consider given the goal
- state the problem in such a way that we can make efficient progress toward a goal state.

Search
- Determine the possible sequence of actions that lead to the states of known values and then choose the best sequence.
- Search algorithms – input is a problem, output is a solution (action sequence)

Execute
- Given the solution, perform the actions.

Problem Solving Agent – Special type of goal based agent.

Environment –

static – agent assumes that in the time it takes to formulate and solve the problem the environment doesn't change

observable – initial state and current state is known

discrete – more than one solution possible

deterministic – the solution, when executed will work!!

Deterministic, fully observable ⇒ *single state problem*
- Agent knows exactly which state it will be in;

A path sequence is an ordered listing of states and actions used in achieving a goal. solution is a sequence

SIMPLE PROBLEM SOLVING AGENT

Agent first formulates goal and problem, then determines an action sequence, after which it executes the sequence.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) return an action
    static: seq, an action sequence
            state, some description of the current world state
            goal, a goal
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then
            goal ← FORMULATE-GOAL(state)
            problem ← FORMULATE-PROBLEM(state,goal)
            seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Defining the Problem

initial state – starting state.

actions - all possible actions available to the agent. Given a state *s*, Actions(s) returns the set of actions that can be executed in *s*

transition model – description of what each action does. A successor is any state reachable from a given state by applying a single action.

state space – set of all states reachable from the initial state by a sequence of actions.

path – a sequence of actions causing you to move from one state to another.

goal test – how do we know we are in a goal state? Apply function to given state to see if it is goal state path cost – some paths are more costly than others. We have some function that assigns cost to a path.

problem - initial state, actions, transition model , goal test, path cost

Solution - path from initial state to a goal state.
Optimal solution - lowest path cost of all solutions

Process of finding a solution is called search

For multiple state problem – problems consist of initial state set, set of operators that specify for each action a set of states reached from any given state.

How do we find and test a problem solving method?
Can look at how a method performs on “toy problems”?
A toy problem is a small, sometimes complex, problem that characterizes a larger set of problems.

How does the method perform on real-world problems?

Ex. Vacuum cleaner world

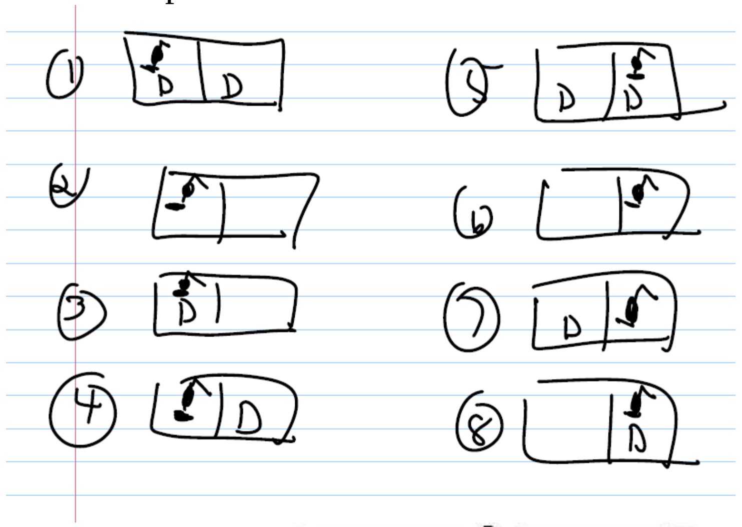
- a) two locations - which may or may not contain dirt
- b) Vacuum is in one of the locations
- c) Actions - move left
  move right
  suck

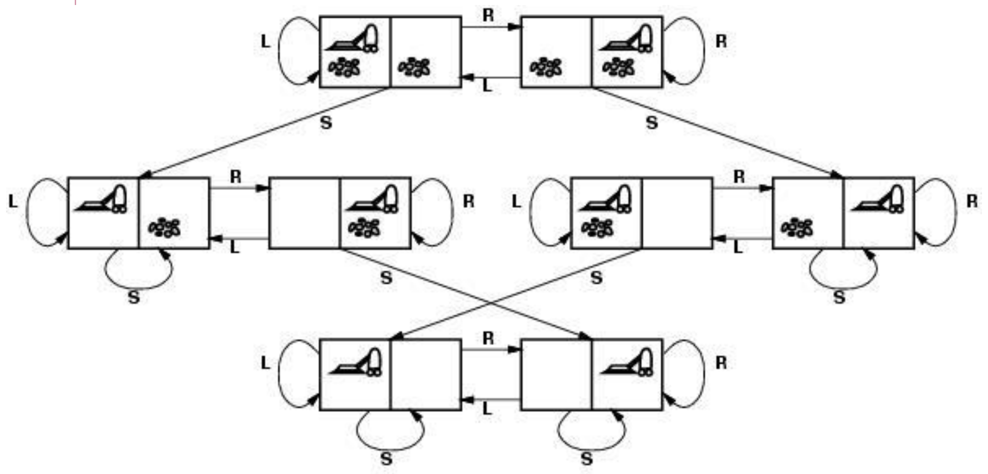
Important part of problem solving is deciding on what goes into state and transition model descriptions

Vacuum cleaner world – motor on/off, power of suck, color of vacuum:

Abstraction – the process of removing detail from a problem

$2 \text{ X } 2^2 = 8$ possible states



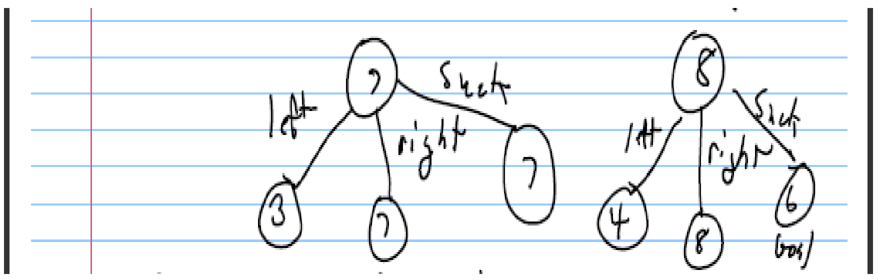


States: two locations with or without dirt: $2 \text{ x } 2^2=8$ states.
Initial state: Any state can be initial
Actions: {*Left*, *Right*, *Suck*}
Transition model: Left, Right, and Suck, moves left, moves right, and suctions respectively. If in rightmost square, Right has no effect. In leftmost square, Left has no effect. Suck in a clean square has no effect.
Goal test: Check whether squares are clean.
Path cost: Each step costs 1, so path cost is the number of steps.

Type of problem depends on the amount of info from the agent's sensors

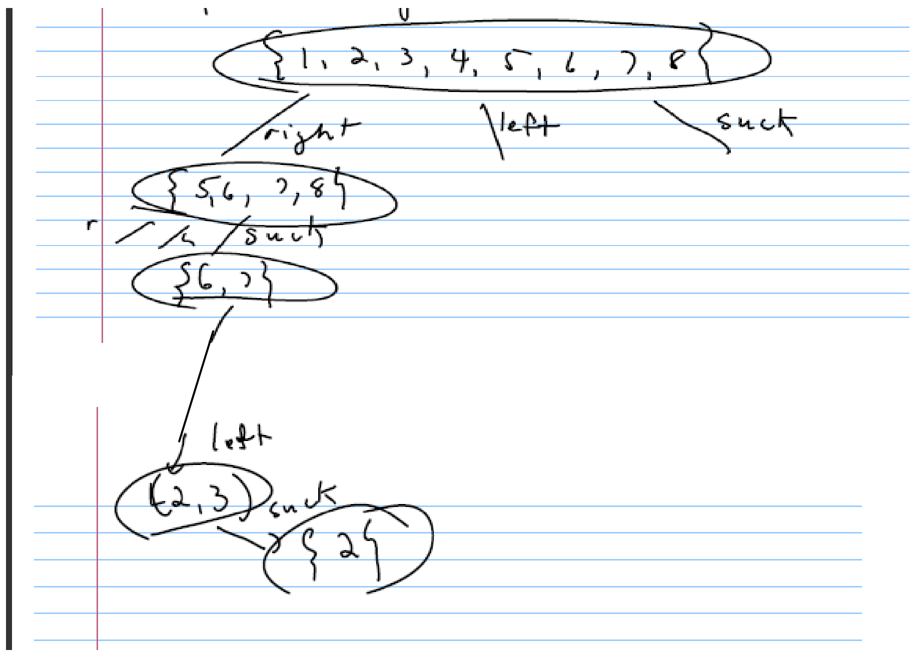
Single state problem, for the agent:

- its world accessible
- knows current state
- knows what each action does
- knows result of action

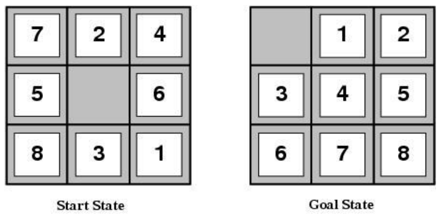


Let's say the agent has no sensors (sensorless problem)
Given a state, the agent can easily calculate which action to take.

Given our vacuum cleaner agent we have a multistate problem to solve.

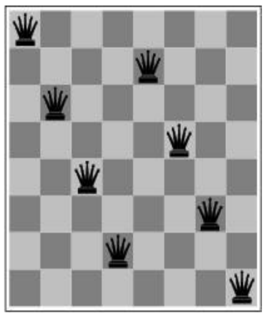


Eight Puzzle



States: Integer location of each tile
Initial state: Any state can be initial
Actions: Move blank {*Left*, *Right*, *Up*, *Down*}
Transition model: Given a state and an action, return the resulting state. i.e. Left on start state above results in the blank and 5 interchanged.
Goal test: Check whether goal configuration is reached
Path cost: Number of steps to reach goal

EIGHT QUEENS PROBLEM



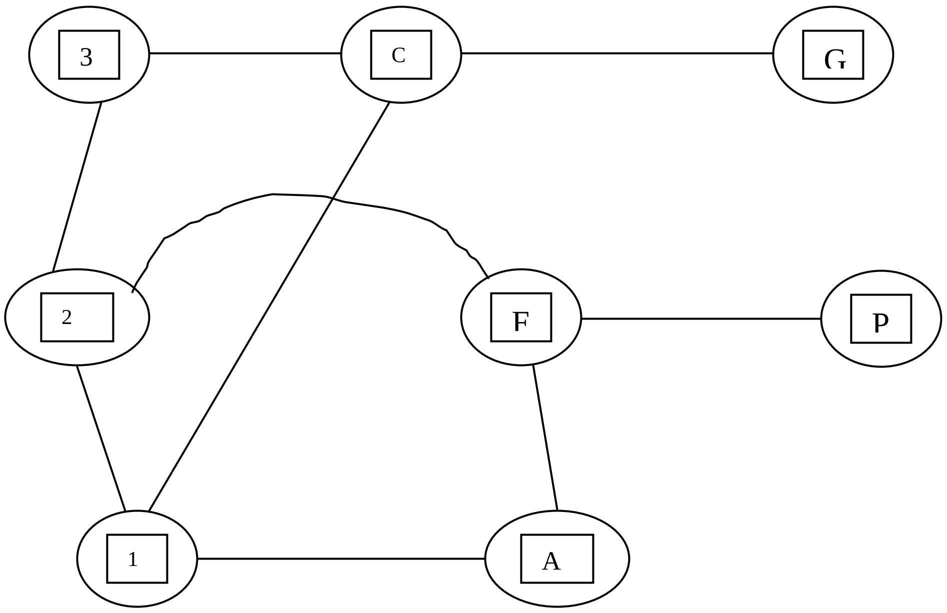
States: Any arrangement of 0 to 8 queens on the board
Initial state: No queens
Actions: Add queen in empty square
Transition model: Returns the board with the queen added to the specified square
Goal test: 8 queens on board and none attacked
Path cost: None
    1.8×10^{14} possible sequences to investigate

Another way to state the problem:
States: *n* ($0 \leq n \leq 8$) queens on the board, one per column in the *n* leftmost columns with no queen attacking another.
Actions: Add queen in leftmost empty column such that is not attacking other queens
    2057 possible sequences to investigate
How you formulate a problem affects how you find the solution.

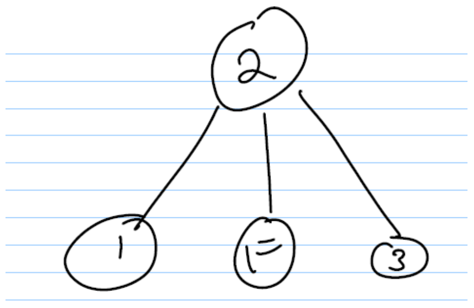
Once a problem is formulated, we need to solve it. We need to "search" for the solution. There are many methods for searching for the solution to a problem. How can we distinguish between one search method and another?

Do we get a solution from the search method?
Is the solution optimal?
How much time and memory does the method take? (search cost)
Do we search the entire search space? complete? i.e. are we guaranteed to find a solution?

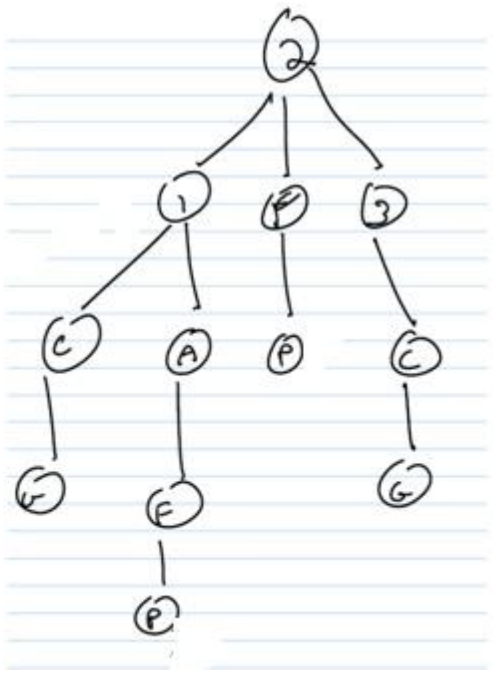
<u>Problem</u>:
It is a nice spring day, birds are singing, air smells great.
You decide to walk from 2N to 1P.
We are going to create a map of the campus.
Not every detail needs to be in our map (abstraction)



Start at the initial state, apply operators and generate a new set of states.



Search strategy - choose one option, see where it gets you, then choose another option.

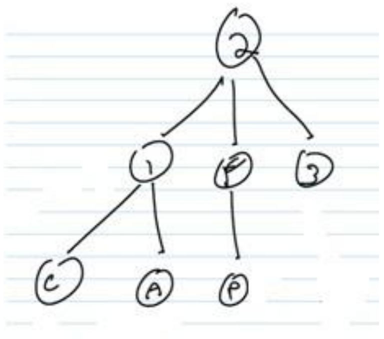


Search tree - differs from state space. There can be an infinite # of paths through search trees.

Tree lingo - parent node
ancestor node
child node
depth of a node - # of nodes on the path to this node.

Uninformed search (blind search) - strategies where no information about the number of steps or the path cost from the current state to the goal.

breadth first search - nodes at the same depth are expanded in sequence. All the nodes at depth *d* are expanded before depth *d + 1*.



If solution exists - will be found. Finds the shallowest solution first.
Complete, optimal when path cost is a linear function of depth.
Time and memory complexity is a function of the branching factor.

The branching factor *b* is the # of states each state can be expanded to.
The bigger *b* the worse breadth first is.

$1 + b + b^2 + b^3 + \dots b^d$

Complexity is $O(b^d)$

Assume we can expand 1000 nodes / sec and node storage is 100 bytes.

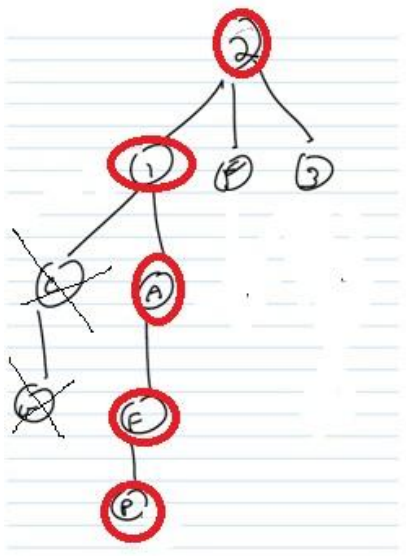
d = 6
b = 10
time = 18 mins
memory 111Mb

d = 8
time 31 hours
memory 1 terabyte

d = 14
time = 3500
memory = 11,111 terabytes (1 exabyte)

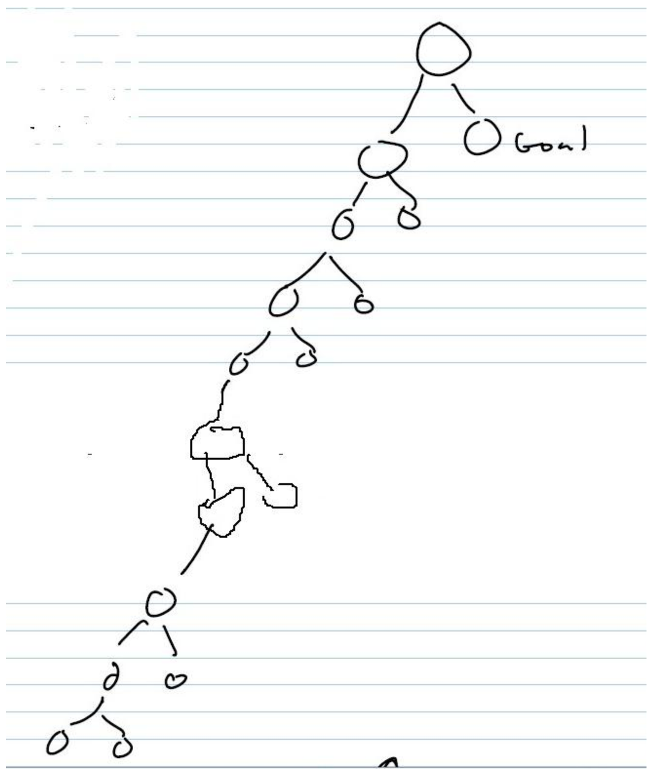
Time is horrible but space considerations are worse!!

Depth first search - always expand one node to the deepest level of the tree.



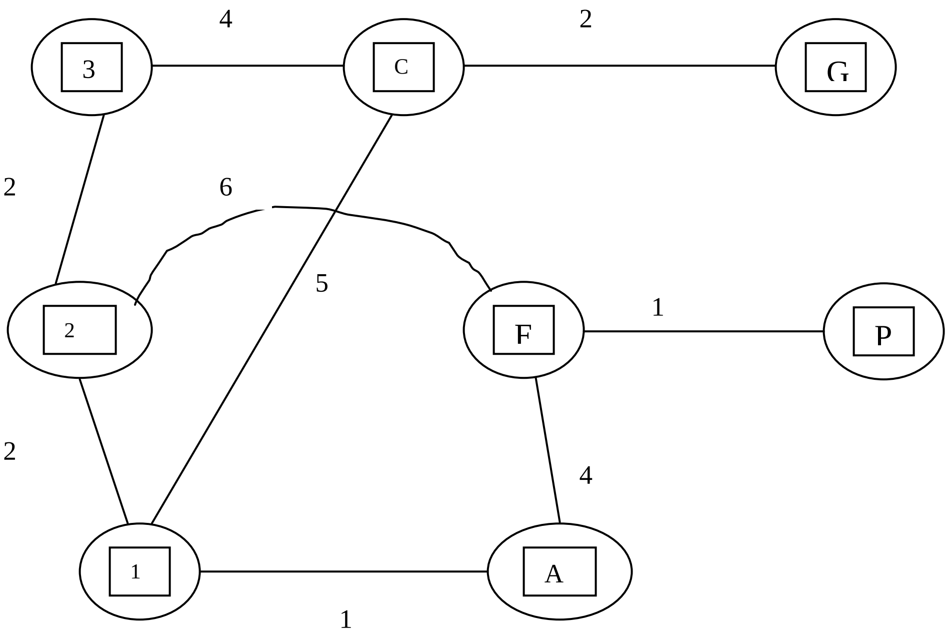
Memory requirements are *bm* where *b* is the branching actor and m is the maximum depth
Need only remember one path at a time. Breadth first you need to remember all paths up to the current level.
Time complexity O(b^m)

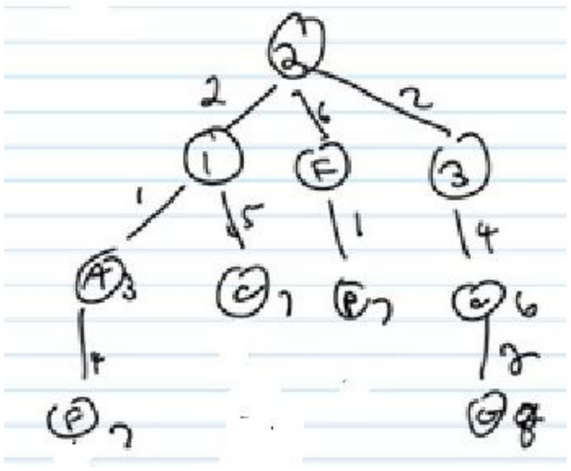
downside - Bad when tree is skewed and paths go deep or even infinite (never get a solution)



<u>Variations of breadth first and depth first search:</u>

Uniform cost search - variation of breadth first search. expand the lowest cost function on the fringe.





First solution found is cheapest solution

Complete, optimal, time complexity b^d Space complexity b^d

Breadth first = uniform cost when g(n) = Depth(n)
g(n) is the cost function

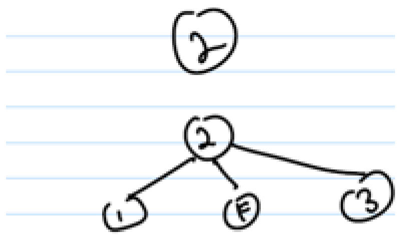
Depth limited search - modification of depth first
Set a limit on the depth you are going to search.
Memory = *bl* where *l* is the is the limit
time complexity = O(b^l)

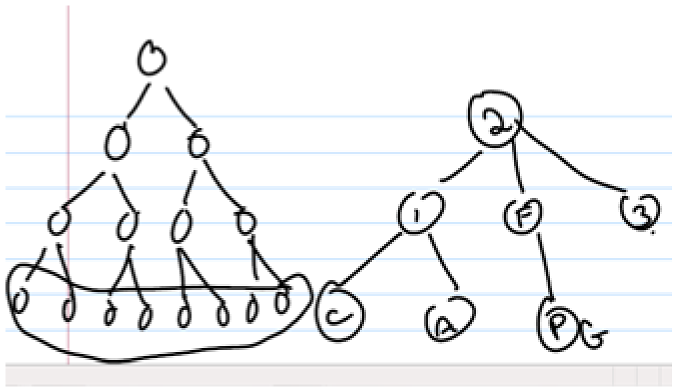
Incomplete if we choose $l < d$
Not optimal if we choose $l > d$

Can be viewed as a special case of depth-limited search with $l = \infty$

Iterative deepening - try each successive depth in turn. Combines breadth first and depth first . First search using BFS to depth 0, then depth 1, then depth 2, etc.

States may be expanded multiple times. this isnot so bad since an exponential problem has most nodes in bottom level.





Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$

Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

For $b = 10$, $d = 5$,

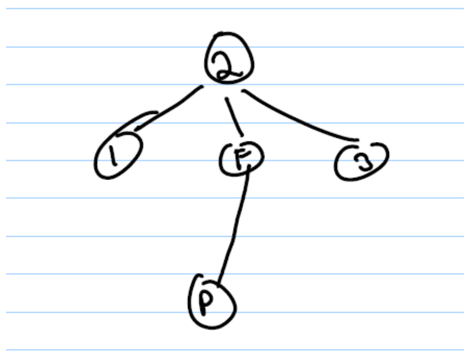
$N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$

$N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

Overhead = (123,456 - 111,111)/111,111 = 11%

Bi directional search - search forward from initial state and backward from goal state. solution is hwre they meet.

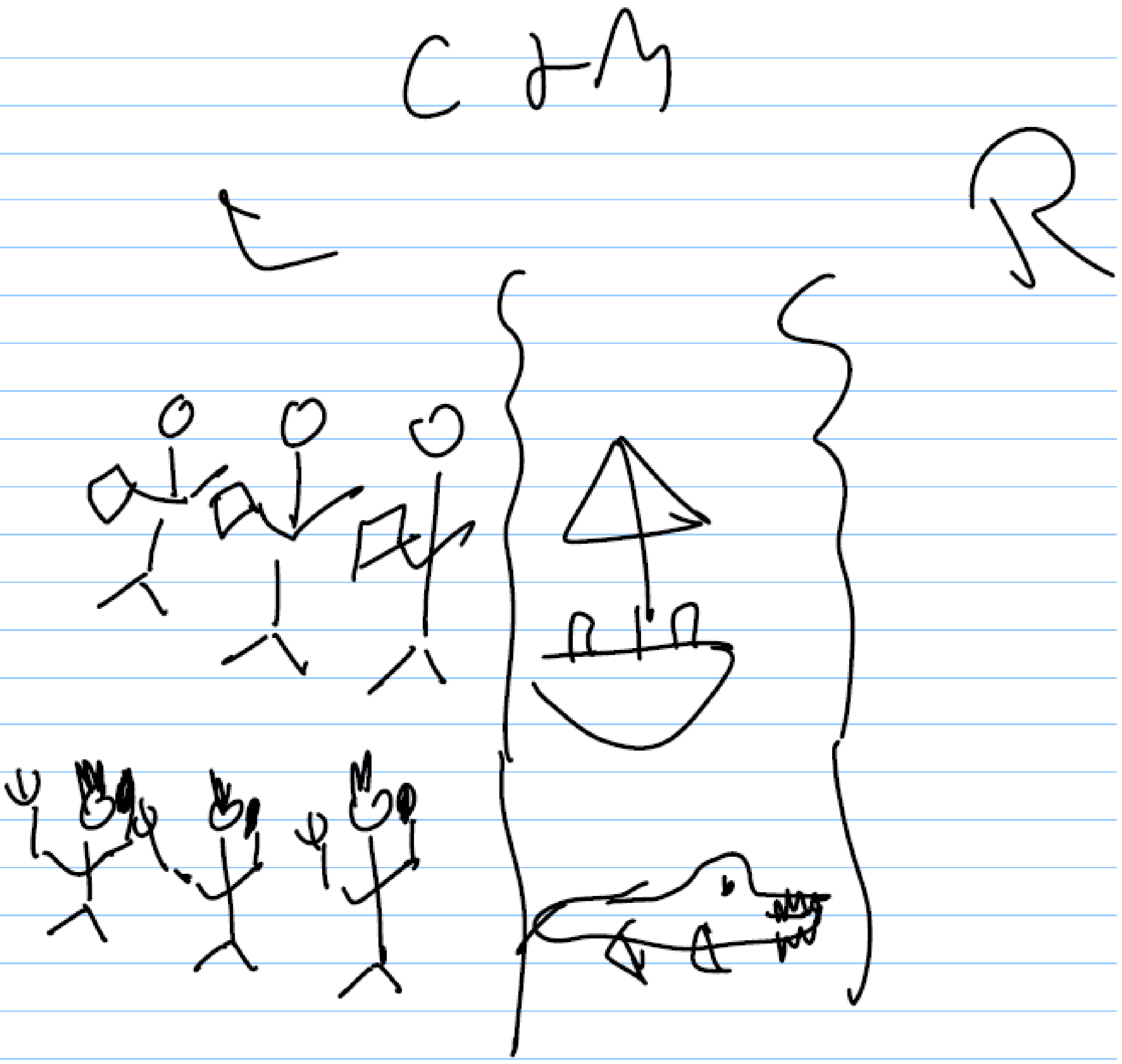
$O(b^{d/2})$ since each search only has to go halfway.



Downside - search backwards generates all predecessors of a node. Sometimes calculating predecessors is difficult

Need to have efficient way to see if node generated is in other half of search.
Check if expanded node is in fringe of other half of search tree.

What type of search is best to do in each half? BFS, DFS??
Cannibals and missionaries - 3 missionaries, 3 cannibals, one boat that can hold 1 or 2 people.
move everyone to other side such that cannibals never outnumber missionaries.
The cannibals are hungry.



Hw Problem 3.9