Uninformed Search (Ch. 3-3.4)

Announcements

Moodle vote is up, so far:

Environment classification

Pick a game/hobby/sport/pastime/whatever and describe both the PEAS and whether the environment/agent is:

1. Fully vs. partially observable (how much see?)
2. Single vs. multi-agent (need to consider others?)
3. Deterministic vs. stochastic (know outcome of action?)
4. Episodic vs. sequential (independent vs. dependent)
5. Static vs. dynamic (have time to think?)
6. Discrete vs. continuous (how finely measure?)
7. Known vs. unknown (know the rules?)

Agent models

Can also classify agents into four categories:

1. Simple reflex
2. Model-based reflex
3. Goal based
4. Utility based

Top is typically simpler and harder to adapt to similar problems, while bottom is more general representations

Agent models

A <u>simple reflex</u> agents acts only on the most recent part of the percept and not the whole history

Our vacuum agent is of this type, as it only looks at the current state and not any previous

These can be generalized as:
"if state = ____ then do action ____"
(often can fail or loop infinitely)

Agent models

A <u>model-based reflex</u> agent needs to have a representation of the environment in memory (called <u>internal state</u>)

This internal state is updated with each observation and then dictates actions

The degree that the environment is modeled is up to the agent/designer (a single bit vs. a full representation)
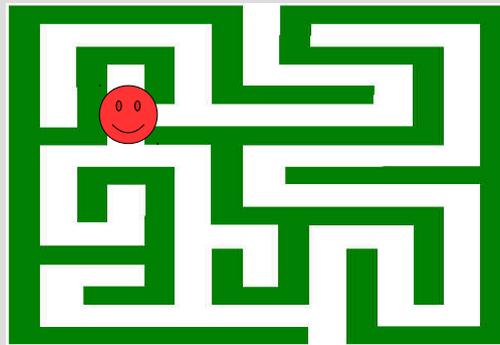
Agent models

This internal state should be from the agent's perspective, not a global perspective
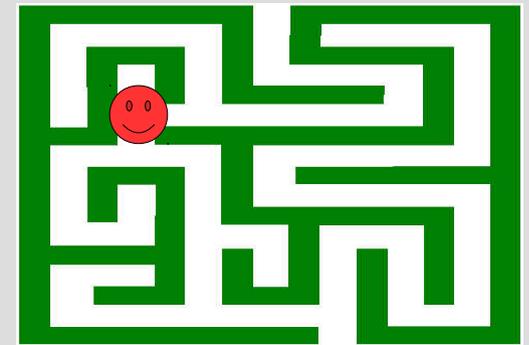(as same global state might have different actions)

Consider these pictures of a maze:
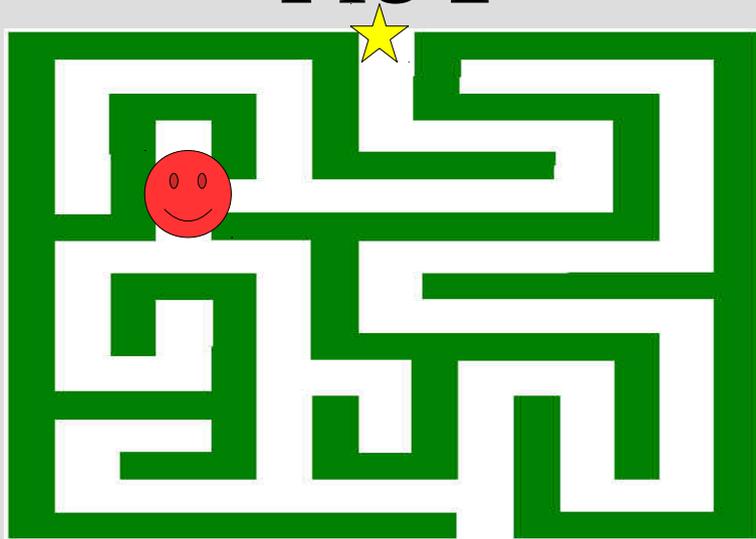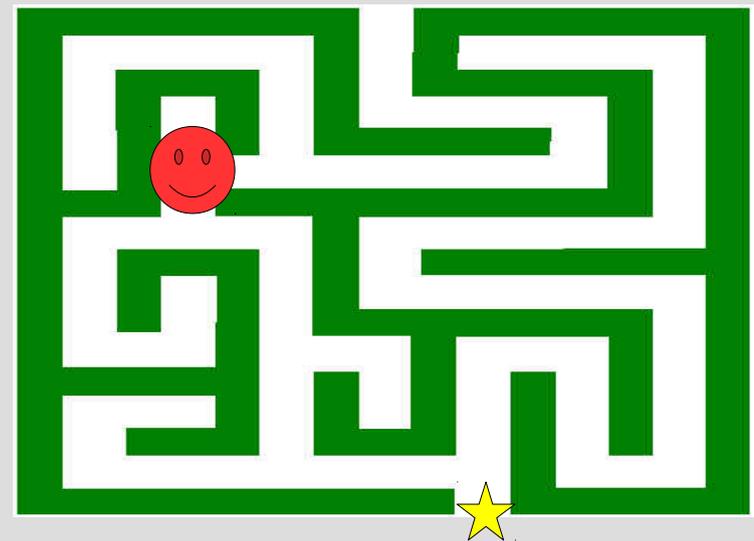Which way to go?

Pic 1

Pic 2

Agent models

The global perspective is the same, but the agents could have different goals (stars)

Pic 1   Pic 2

Goals are not global information

Agent models

For the vacuum agent if the dirt does not reappear, then we do not want to keep moving

The simple reflex agent program cannot do this, so we would have to have some memory (or model)

This could be as simple as a flag indicating whether or not we have checked the other state

Agent models

The <u>goal based</u> agent is more general than the model-based agent

In addition to the environment model, it has a <u>goal</u> indicating a desired configuration

Abstracting to a goals generalizes your method to different (similar) problems
(for example, a model-based agent could solve one maze, but a goal can solve any maze)
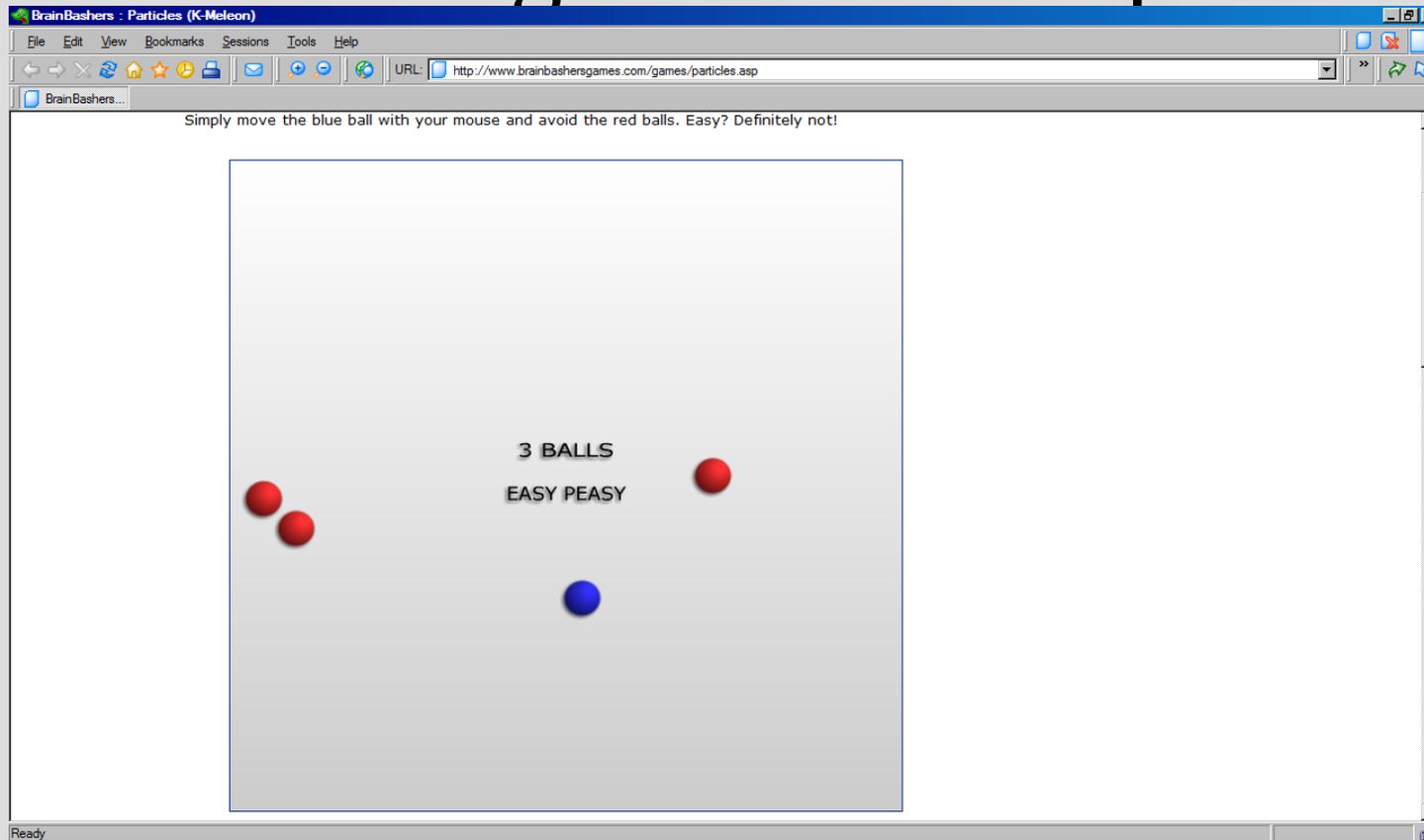
Agent models

A <u>utility</u> based agent maps the sequence of states (or actions) to a real value

Goals can describe general terms as "success" or "failure", but there is no degree of success

In the maze example, a goal based agent can find the exit. But a utility based agent can find the shortest path to the exit
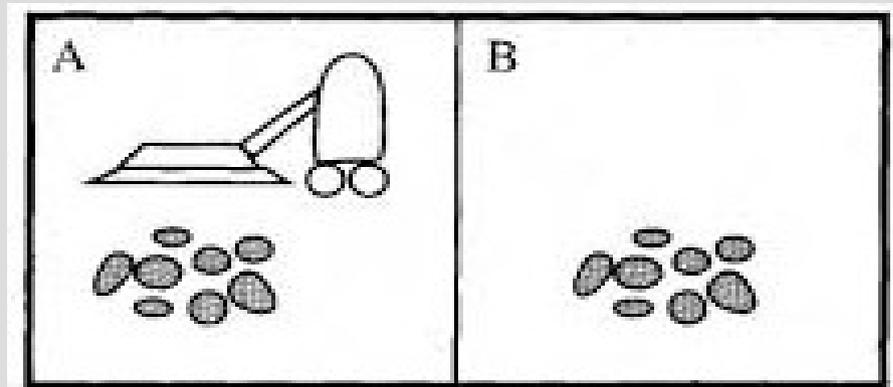
Agent models

What is the agent model of particles?

Think of a way to improve the agent and describe what model it is now

Agent models

What is the agent model of our vacuum?
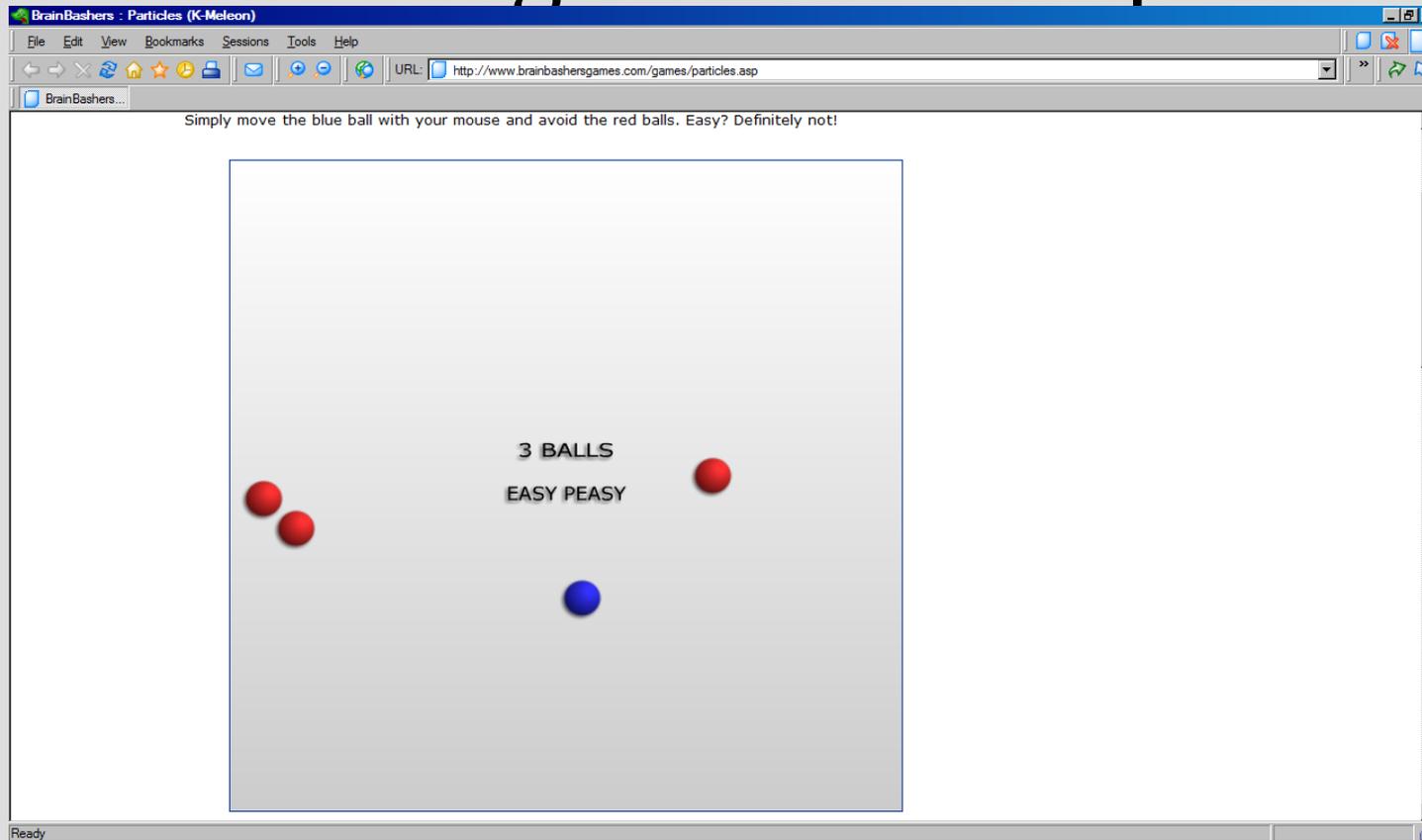

if [Dirty], return [Suck]
if at [state A], return [move right]
if at [state B], return [move left]

Agent models

What is the agent model of particles?

Think of a way to improve the agent and describe what model it is now

Agent learning

For many complicated problems (facial recognition, high degree of freedom robot movement), it would be too hard to explicitly tell the agent what to do

Instead, we build a framework to learn the problem and let the agent decide what to do

This is less work and allows the agent to adapt if the environment changes

Agent learning

There are four main components to learning:
1. Critic = evaluates how well the agent is doing and whether it needs to change actions (similar to performance measure)
2. Learning element = incorporate new information to improve agent
3. Performance element = selects action agent will do (exploit known best solution)
4. Problem generator = find new solutions (explore problem space for better solution)

State structure

States can be generalized into three categories:

1. Atomic (Ch. 3-5, 15, 17)
2. Factored (Ch. 6-7, 10-11, 13-16, 18, 20-21)
3. Structured (Ch. 8-9, 12, 14, 19, 22-23)
(Top are simpler, bottom are more general)

Occam's razor = if two results are identical, use the simpler approach

State structure

An <u>atomic</u> state has no sub-parts and acts as a simple unique identifier

An example is an elevator:
Elevator = agent (actions = up/down)
Floor = state

In this example, when someone requests the elevator on floor 7, the only information the agent has is what floor it currently is on

State structure

Another example of an atomic representation is simple path finding:
If we start (here) in Vincent 16, how would you get to Keller's CS office?

V. 16 -> Hallway1 -> V. Stairs -> Outside -> Walk to KHKH -> K. Stairs -> CS office

The words above hold no special meaning other than differentiating from each other

State structure

A <u>factored</u> state has a fixed number of variables/attributes associated with it

Our simple vacuum example is factored, as each state has an id (A or B) along with a "dirty" property

In particles, each state has a set of red balls with locations along with the blue ball location

State structure

<u>Structured</u> states simply describe objects and their relationship to others

Suppose we have 3 blocks: A, B and C
We could describe: A on top of B, C next to B

A factored representation would have to enumerate all possible configurations of A, B and C to be as representative

State structure

We will start using <u>structured</u> approaches when we deal with logic:

Summer implies Warm
Warm implies T-Shirt

The current state might be:
!Summer (¬Summer)
but the states have intrinsic relations between each other (not just actions)

Search

Goal based agents need to search to find a path from their start to the goal (a path is a sequence of actions, not states)

For now we consider <u>problem solving</u> agents who search on atomically structured spaces

Today we will focus on <u>uninformed</u> searches, which only know cost between states but no other extra information

Search

In the vacuum example, the states and actions are obvious and simple

In more complex environments, we have a choice of how to abstract the problem into simple (yet expressive) states and actions

The solution to the abstracted problem should be able to serve as the basis of a more detailed problem (i.e. fit the detailed solution inside)

Search

Example: Google maps gives direction by telling you a sequence of roads and does not dictate speed, stop signs/lights, road lane

Search

In deterministic environments the search solution is a single sequence (list of actions)

Stochastic environments need multiple sequences to account for all possible outcomes of actions

It can be costly to keep track of all of these and might be better to keep the most likely and search again if you are off the sequences

Search

There are 5 parts to search:
1. Initial state
2. Actions possible at each state
3. Transition model (result of each action)
4. Goal test (are we there yet?)
5. Path costs/weights (not stored in states)
   (related to performance measure)

In search we normally fully see the problem and the initial state and compute all actions

Small examples

Here is our vacuum world again:

1. initial

4. goals
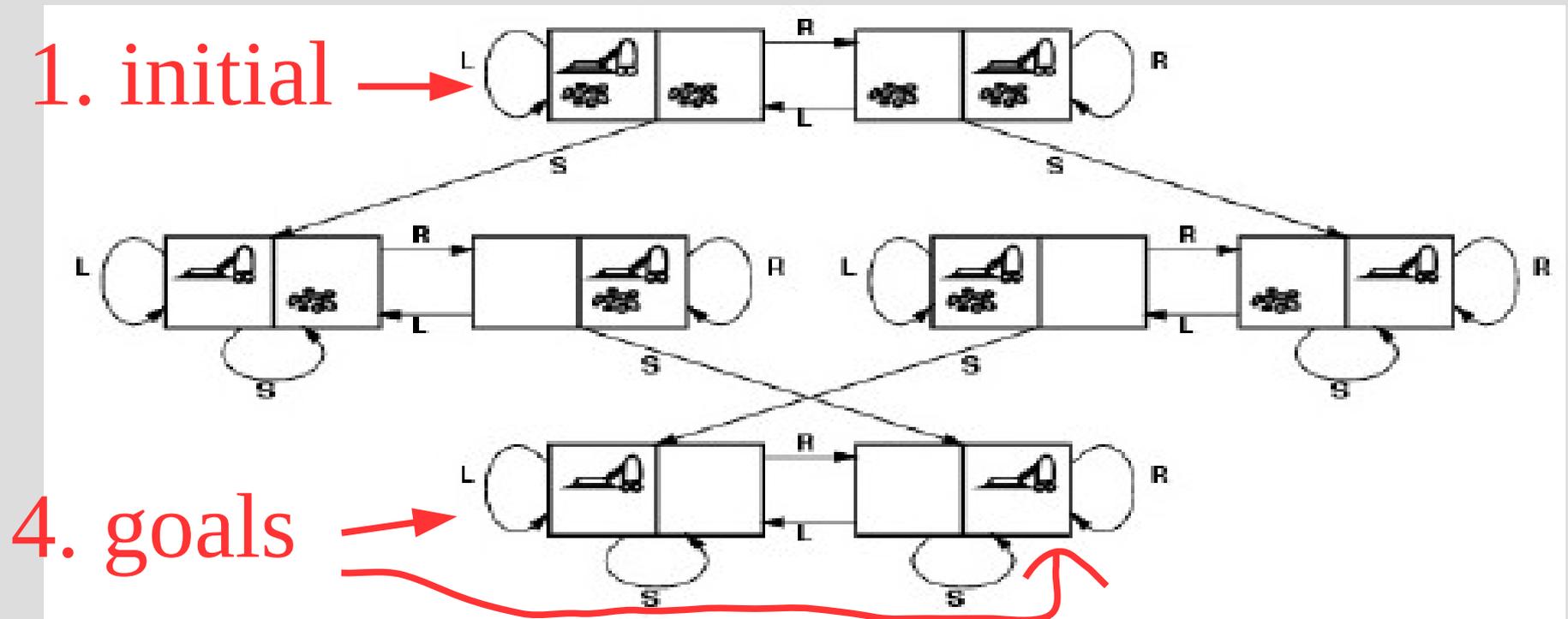
2. For all states, we have actions: L, R or S
3. Transition model = black arrows
5. Path cost = ??? (from performance measure)

Small examples

8-Puzzle
1. (semi) Random
2. All states: U,D,L,R
4. As shown here —→

5. Path cost = 1 (move count)
3. Transition model (example):

Result(,D) =

(see: https://www.youtube.com/watch?v=DfVjTkzk2Ig)

Small examples

8-Puzzle is NP complete so to find the best solution, we must brute force

3x3 board = = 181,440 states

4x4 board = 1.3 trillion states
    Solution time: milliseconds

5x5 board = 10^{25} states
    Solution time: hours

Small examples

8-Queens: how to fit 8 queens on a 8x8 board so no 2 queens can capture each other

Two ways to model this:
Incremental = each action is to
   add a queen to the board
   (1.8×10^{14} states)
Complete state formulation = all 8 queens start
   on board, action = move a queen
   (2057 states)
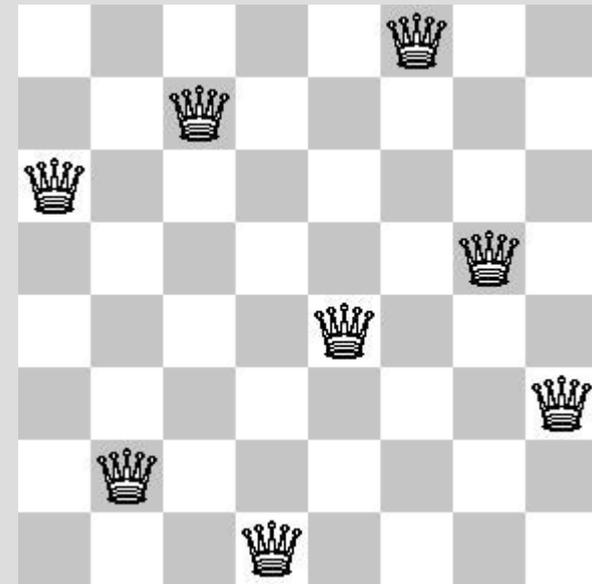
Real world examples

Directions/traveling (land or air)

Model choices: only have interstates?
Add smaller roads, with increased cost?
(pointless if they are never taken)

Real world examples

Traveling salesperson problem (TSP): Visit each location exactly once and return to start
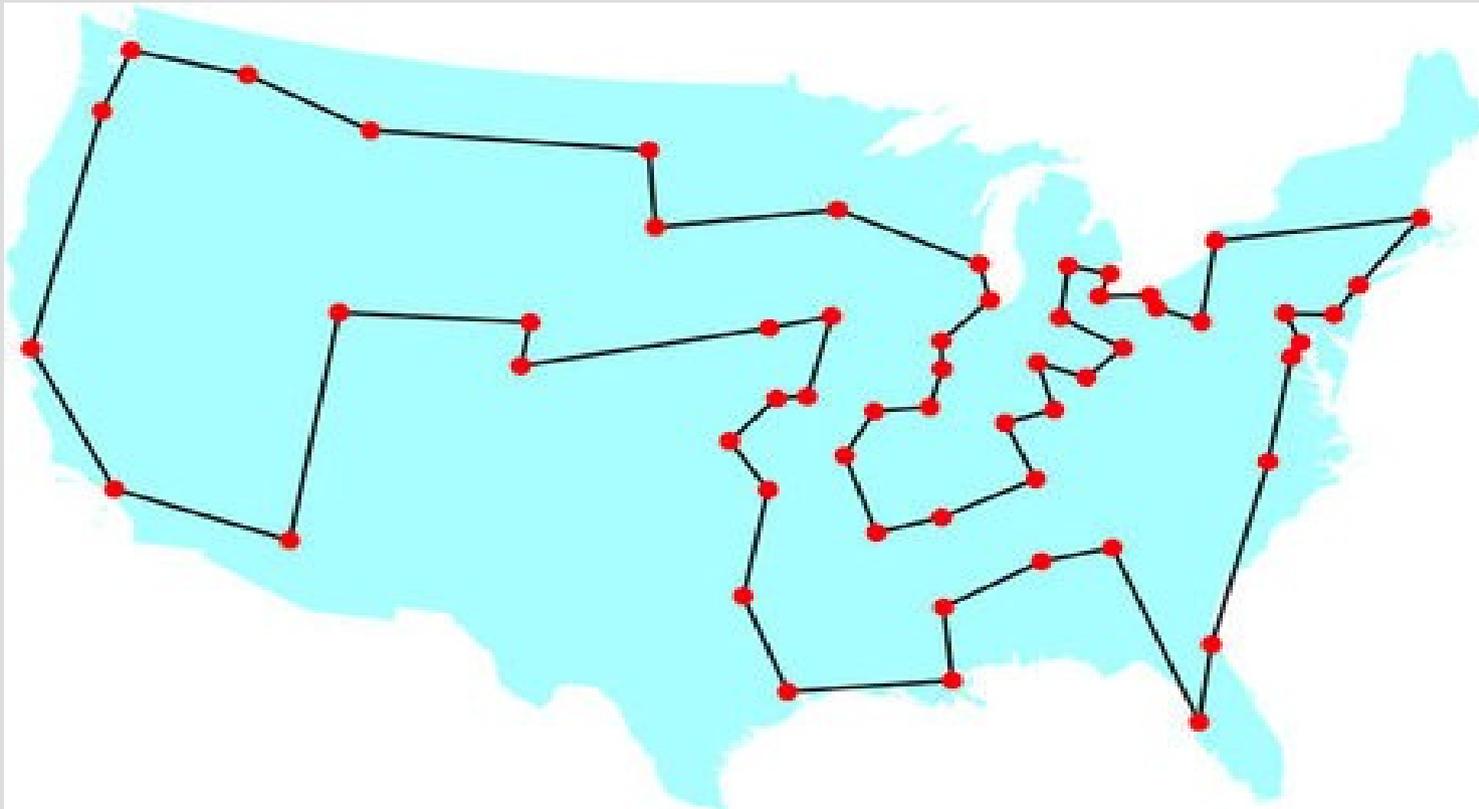
Goal: Minimize distance traveled

Search algorithm

To search, we will build a tree with the root as the initial state

```
function tree-search(root-node)
    fringe ← successors(root-node)
    while ( notempty(fringe) )
        {node ← remove-first(fringe)
        state ← state(node)
        if goal-test(state) return solution(node)
        fringe ← insert-all(successors(node),fringe) }
    return failure
end tree-search
```

Any problems with this?

Search algorithm
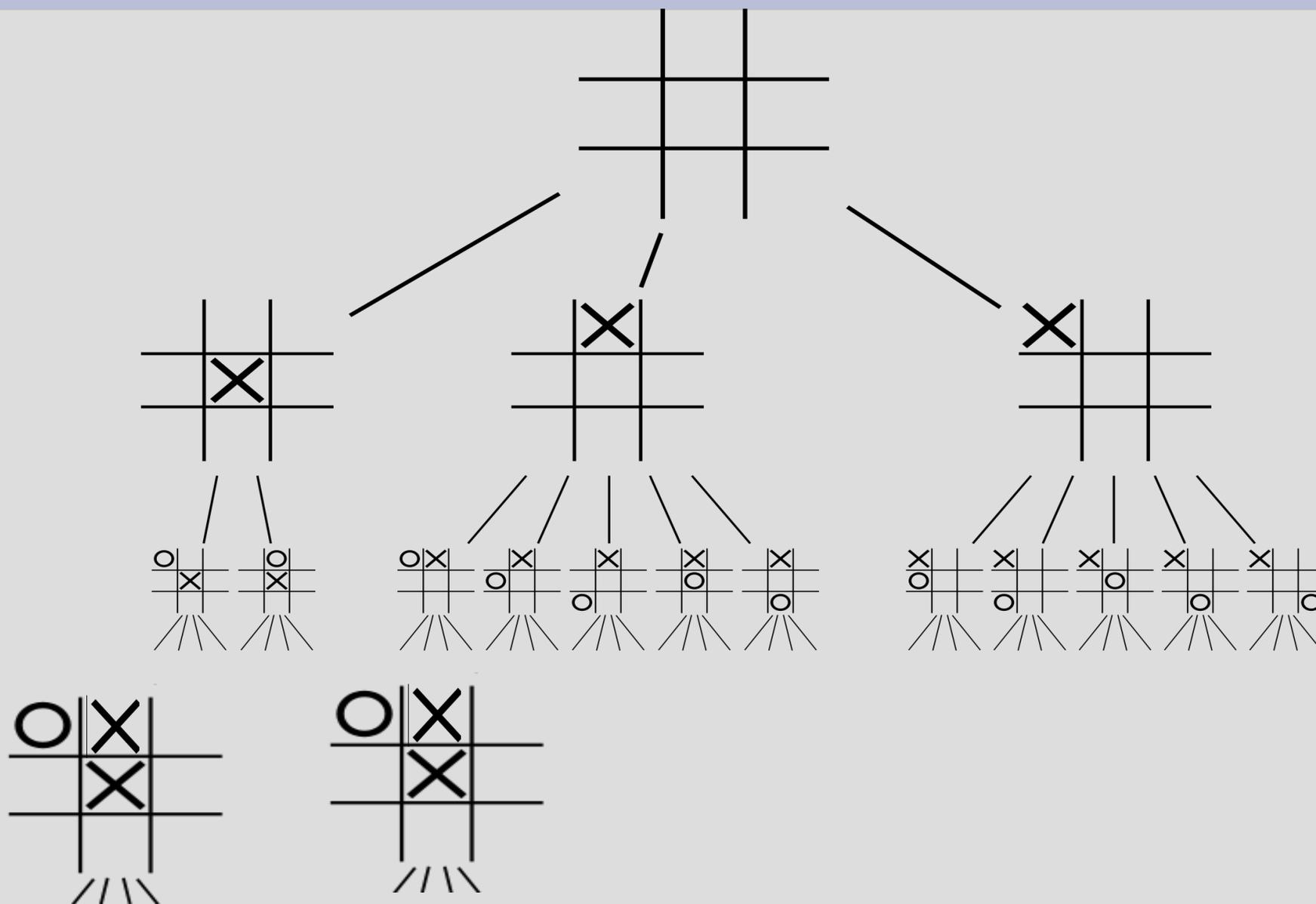
Search algorithm

8-queens can actually be generalized to the question:
Can you fit n queens on a z by z board?

Except for a couple of small size boards, you can fit z queens on a z by z board

This can be done fairly easily with recursion

(See: nqueens.py)

Search algorithm

We can remove visiting states multiple times by doing this:

```
function tree-search(root-node)
    fringe ← successors(root-node)
    explored ← empty
    while ( notempty(fringe) )
        {node ← remove-first(fringe)
            state ← state(node)
            if goal-test(state) return solution(node)
            explored ← insert(node,explored)
            fringe ← insert-all(successors(node),fringe, if node not in explored)
        }
    return failure
end tree-search
```

But this is still not necessarily all that great...

Search algorithm

Next we will introduce and compare some tree search algorithms

These all assume nodes have 4 properties:
1. The current state
2. Their parent state (and action for transition)
3. Children from this node (result of actions)
4. Cost to reach this node (from root)

Search algorithm

When we find a goal state, we can back track via the parent to get the sequence

To keep track of the unexplored nodes, we will use a queue (of various types)

The explored set is probably best as a hash table for quick lookup (have to ensure similar states reached via alternative paths are the same in the has, can be done by sorting)

Search algorithm

The search algorithms metrics/criteria:
1. Completeness (does it terminate with a valid solution)
2. Optimality (is the answer the best solution)
3. Time (in big-O notation)
4. Space (big-O)

b = maximum branching factor
d = minimum depth of a goal
m = maximum length of any path
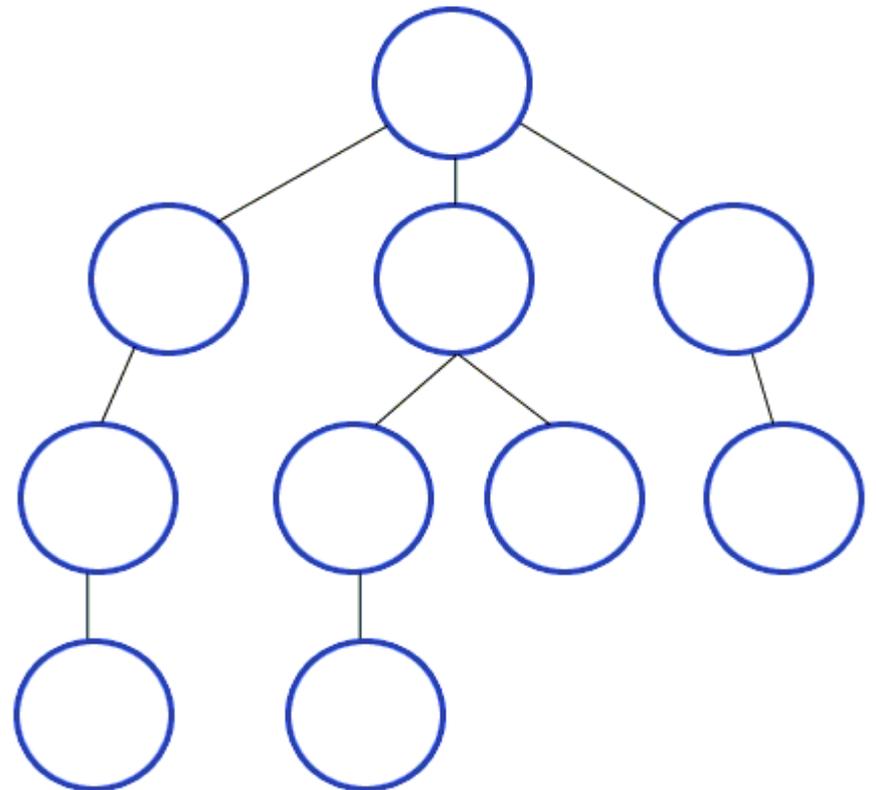
Uninformed search

Today, we will focus on <u>uninformed search</u>, which only have the node information (4 parts) (the costs are given and cannot be computed)

Next time we will continue with <u>informed searches</u> that assume they have access to additional structures of the problem (i.e. if costs were distances between cities, you could also compute the distance "as the bird flies")
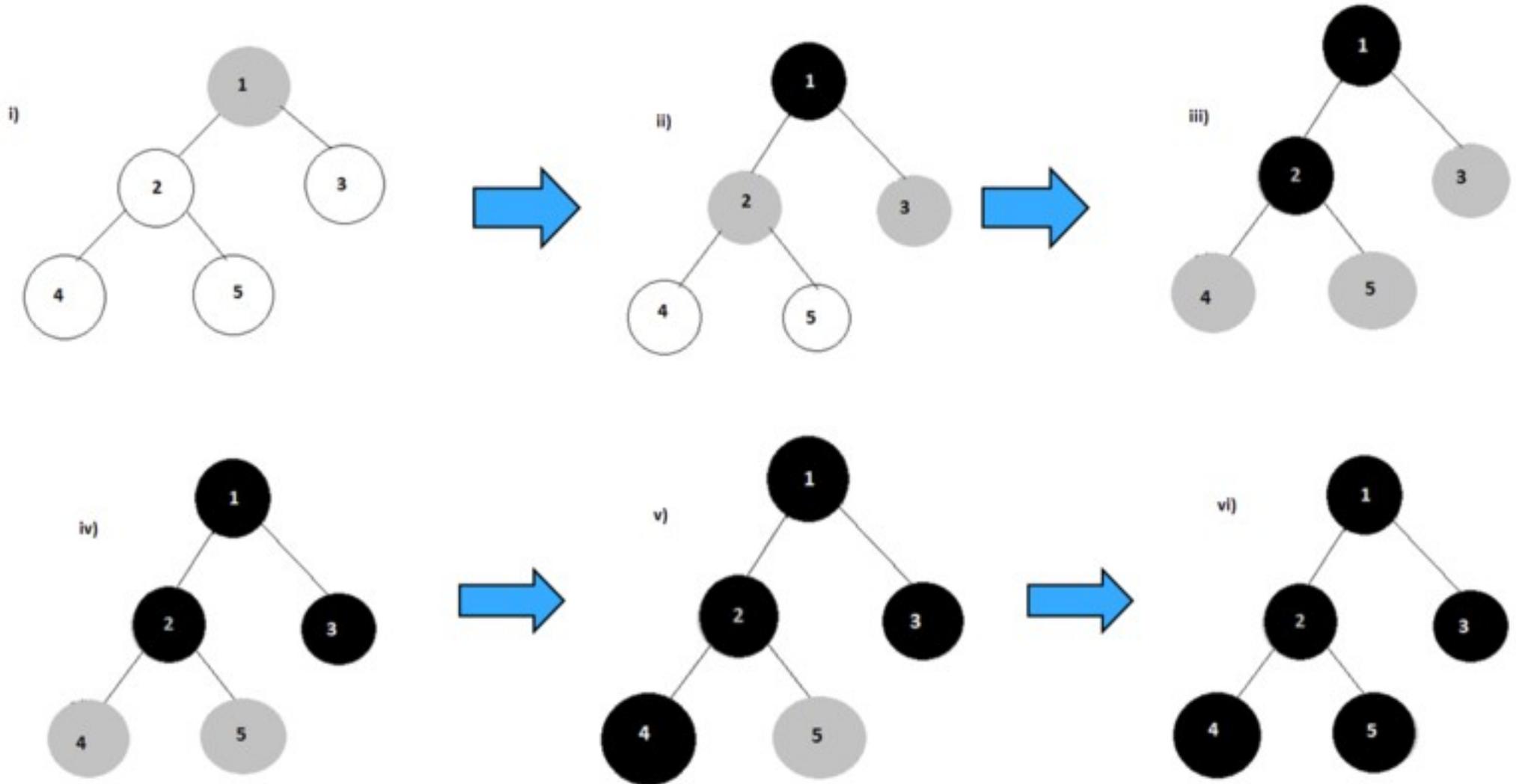
Breadth first search

Breadth first search checks all states which are reached with the fewest actions first

(i.e. will check all states that can be reached by a single action from the start, next all states that can be reached by two actions, then three...)

Breadth first search

(see: https://www.youtube.com/watch?v=5UfMU9TsoEM)
(see: https://www.youtube.com/watch?v=nI0dT288VLs)

Breadth first search

BFS can be implemented by using a simple FIFO (first in, first out) queue to track the fringe/frontier/unexplored nodes

Metrics for BFS:

Complete (i.e. guaranteed to find solution if exists)

Non-optimal (unless uniform path cost)

Time complexity = $O(b^d)$

Space complexity = $O(b^d)$

Breadth first search

Exponential problems are not very fun, as seen in this picture:

| Depth	Nodes	Time		Memory	
2	110	.11	milliseconds	107	kilobytes
4	11,110	11	milliseconds	10.6	megabytes
6	10^6	1.1	seconds	1	gigabyte
8	10^8	2	minutes	103	gigabytes
10	10^{10}	3	hours	10	terabytes
12	10^{12}	13	days	1	petabyte
14	10^{14}	3.5	years	99	petabytes
16	10^{16}	350	years	10	exabytes

Figure 3.13   Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform-cost search

Uniform-cost search also does a queue, but uses a priority queue based on the cost (the lowest cost node is chosen to be explored)
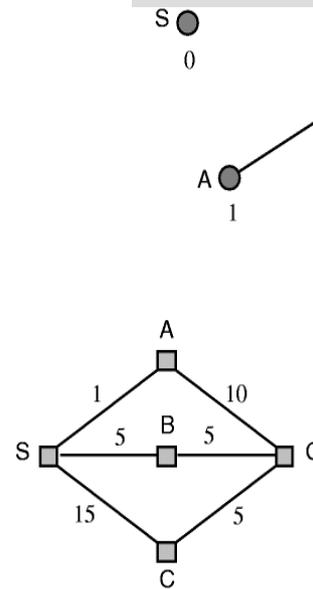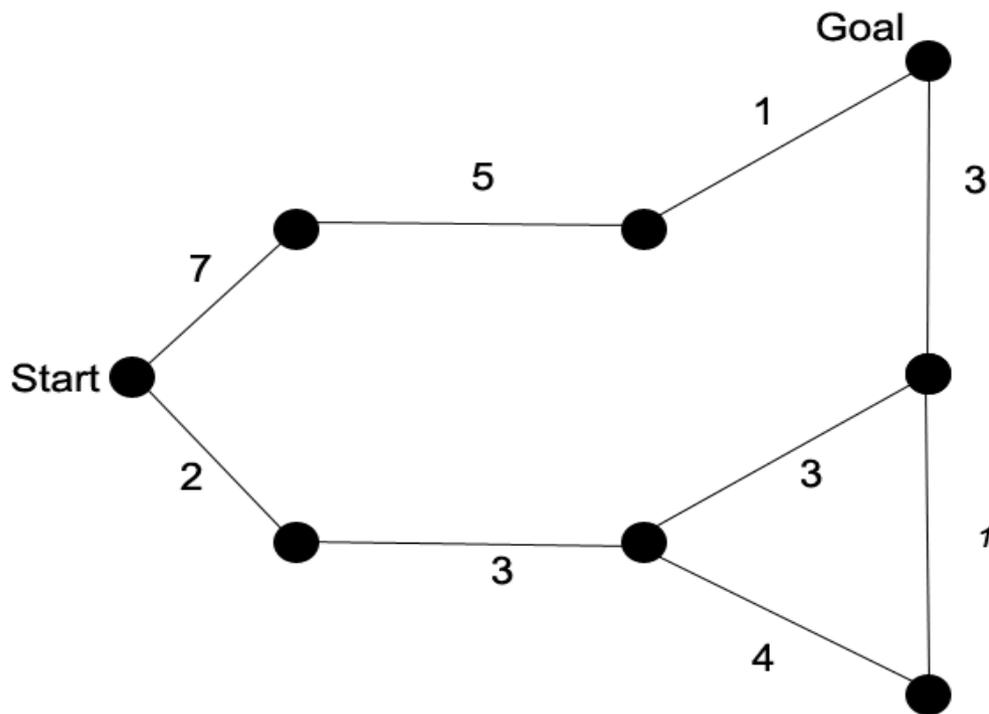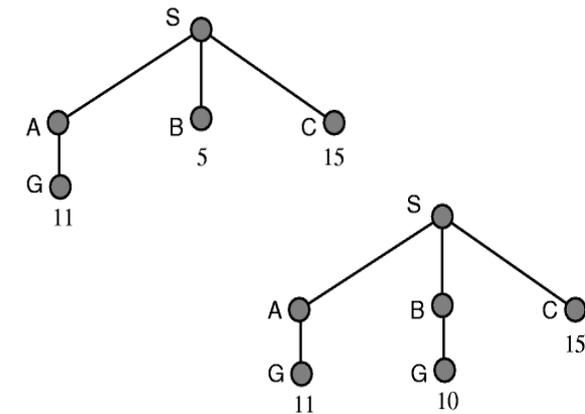
Uniform-cost search

The only modification is when exploring a node we cannot disregard it if it has already been explored by another node

We might have found a shorter path and thus need to update the cost on that node

We also do not terminate when we find a goal, but instead when the goal has the lowest cost in the queue.
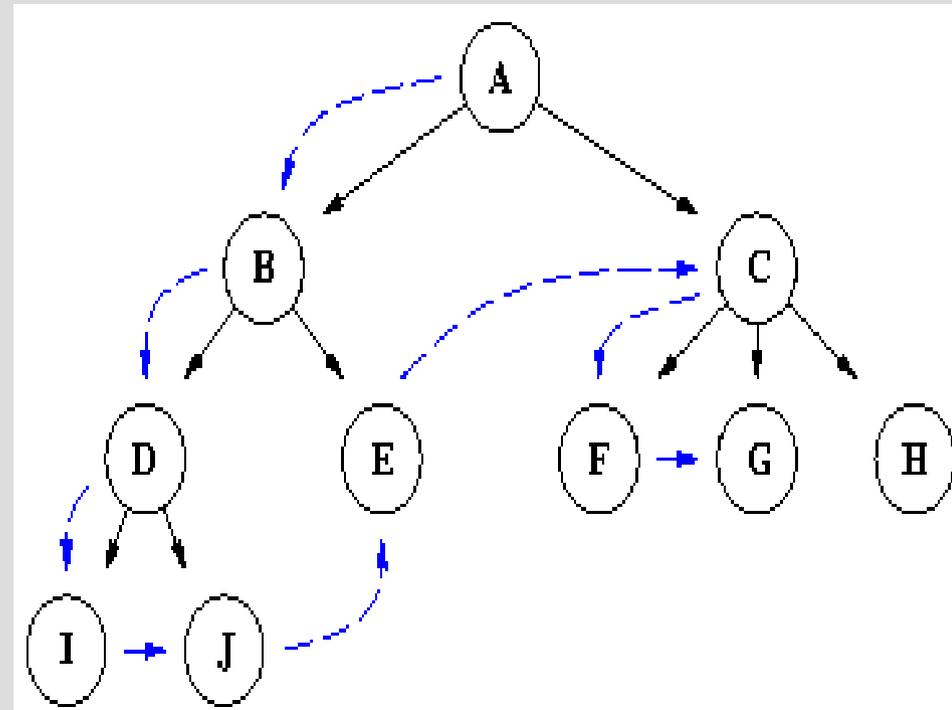
Uniform-cost search

UCS is..

1. Complete (if costs strictly greater than 0)
2. Optimal

However....
3&4. Time complexity = space complexity
= O(b$^{1+C*/\text{min(path cost)}}$), where C* cost of
optimal solution (much worse than BFS)

Depth first search

DFS is same as BFS except with a FILO (or LIFO) instead of a FIFO queue
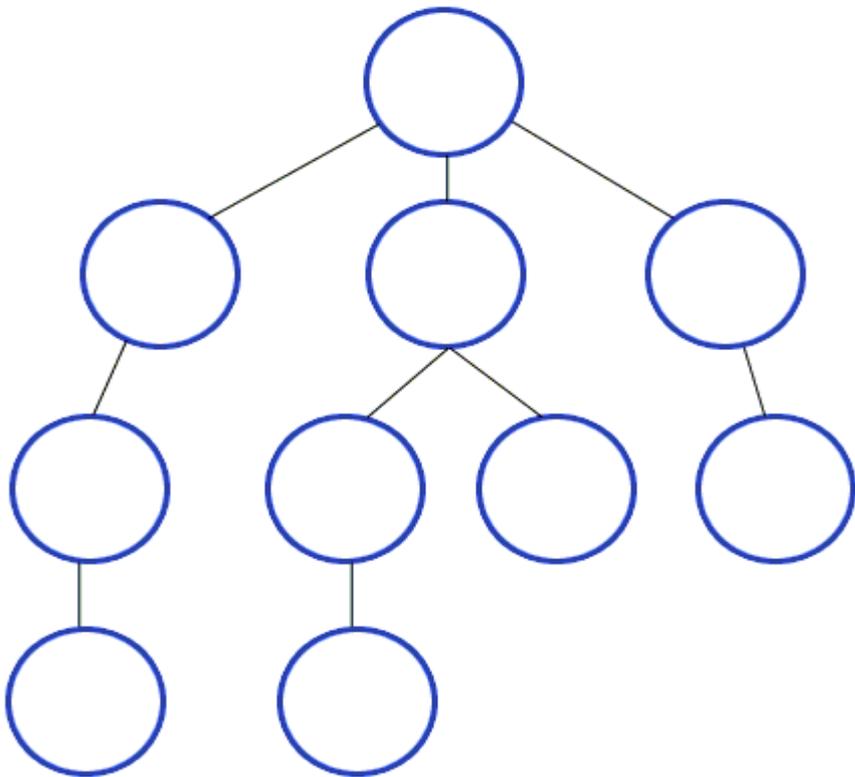
Depth first search

Metrics:
1. Might not terminate (not correct) (e.g. in vacuum world, if first expand is action L)
2. Non-optimal (just... no)
3. Time complexity = $O(b^d)$
4. Space complexity = $O(b*d)$

Only way this is better than BFS is the space complexity...

Depth limited search

DFS by itself is not great, but it has two (very) useful modifications

Depth limited search runs normal DFS, but if it is at a specified depth limit, you cannot have children (i.e. take another action)

Typically with a little more knowledge, you can create a reasonable limit and makes the algorithm correct

Depth limited search

However, if you pick the depth limit before d, you will not find a solution (not correct, but will terminate)

Iterative deepening DFS

Probably the most useful uninformed search is <u>iterative deepening DFS</u>

This search performs depth limited search with maximum depth 1, then maximum depth 2, then 3... until it finds a solution

Goal state
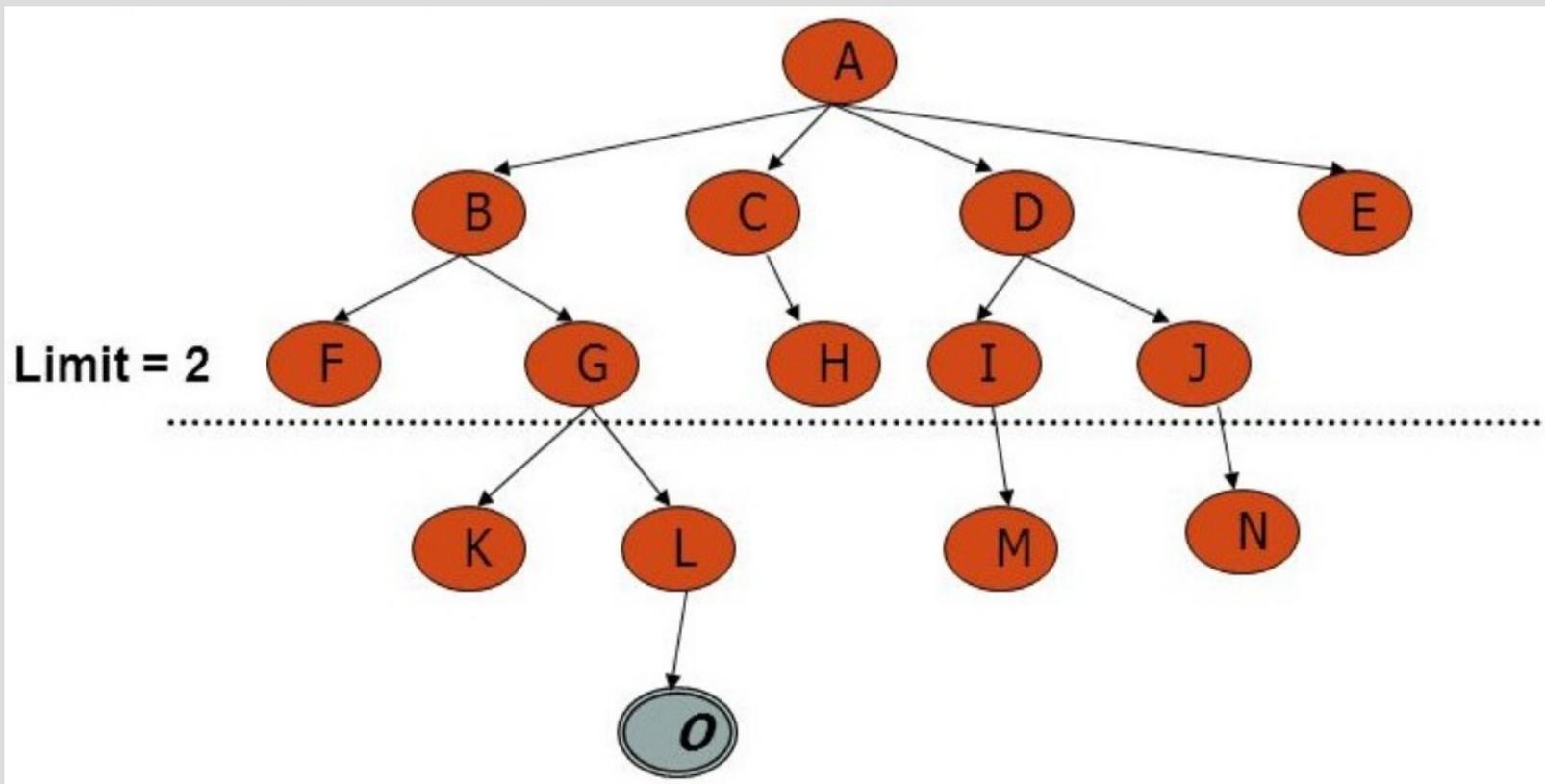
Iterative deepening DFS
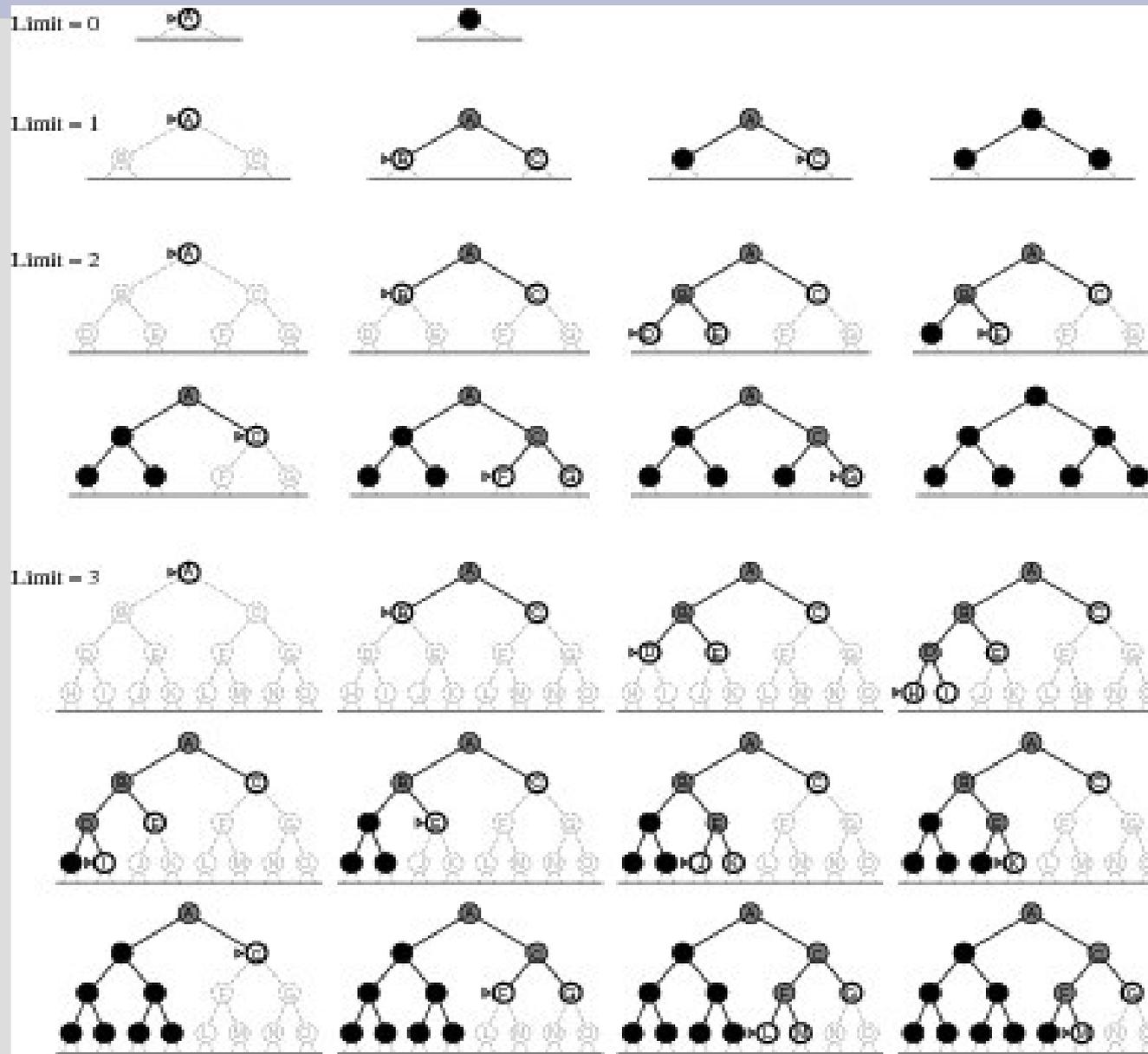
Iterative deepening DFS

The first few states do get re-checked multiple times in IDS, however it is not too many

When you find the solution at depth d, depth 1 is expanded d times (at most b of them)

The second depth are expanded d-1 times (at most b^2 of them)

Thus $d \cdot b + (d-1) \cdot b^2 + ... + 1 \cdot b^d = O(b^d)$

Iterative deepening DFS

Metrics:
1. Complete
2. Non-optimal (unless uniform cost)
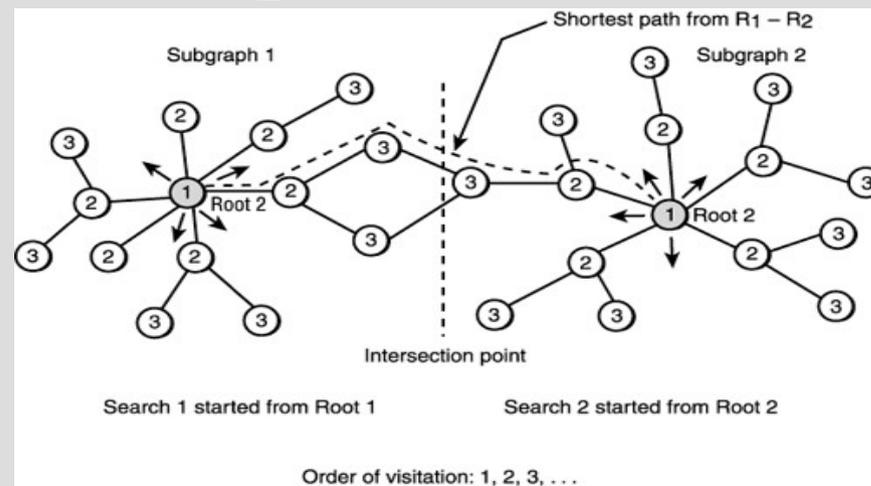3. $O(b^d)$
4. $O(b*d)$

Thus IDS is better in every way than BFS (asymptotically)

Best uninformed we will talk about

Bidirectional search

<u>Bidirectional search</u> starts from both the goal and start (using BFS) until the trees meet

This is better as $2*(b^{d/2}) < b^d$
(the space is much worse than IDS, so only applicable to small problems)

Shortest path from R1 – R2

Subgraph 1   Subgraph 2

Intersection point

Search 1 started from Root 1   Search 2 started from Root 2

Order of visitation: 1, 2, 3, . . .

Summary of algorithms
Fig. 3.21, p. 91

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

There are a number of footnotes, caveats, and assumptions.
See Fig. 3.21, p. 91.
[a] complete if b is finite
[b] complete if step costs $\geq \varepsilon > 0$
[c] optimal if step costs are all identical
     (also if path cost non-decreasing function of depth only)
[d] if both directions use breadth-first search
     (also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Generally the preferred uninformed search strategy