CSE 3401: Intro to Artificial Intelligence Uninformed Search

- Required Readings: R & N Chapter 3, Sec. 1-4.
- Lecture slides adapted from those of Fahiem Bacchus.

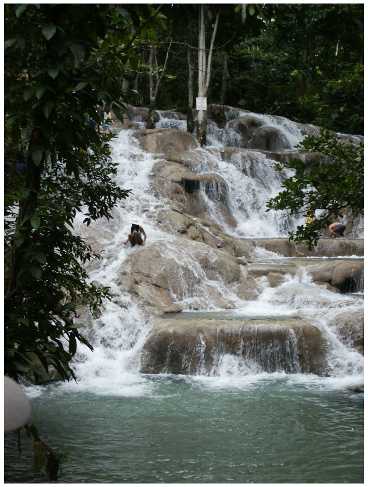
Why Search

- Successful
  - Success in game playing programs based on search.
  - Many other AI problems can be successfully solved by search.
- Practical
  - Many problems don't have a simple algorithmic solution. Casting these problems as search problems is often the easiest way of solving them. Search can also be useful in approximation (e.g., local search in optimization problems).
  - Often specialized algorithms cannot be easily modified to take advantage of extra knowledge. Heuristics in search provide a natural way of utilizing extra knowledge.
- Some critical aspects of intelligent behaviour, e.g., planning, can be naturally cast as search.

Example, a holiday in Jamaica

Things to consider

- Prefer to avoid hurricane season.
- Rules of the road, larger vehicle has right of way (especially trucks).
- Want to climb up to the top of Dunns river falls.



But you want to start your climb at 8:00 am before the crowds arrive!

- Want to swim in the Blue Lagoon

- Want to hike the Cockpit Country

- No roads, need local guide and supplies.

- Easier goal, climb to the top of Blue Mountain

- Near Kingston.
- Organized hikes available.
- Need to arrive on the peak at dawn, before the fog sets in.
- Can get some Blue Mountain coffee!

How do we plan our holiday?

- We must take into account various preferences and constraints to develop a schedule.
- An important technique in developing such a schedule is "hypothetical" reasoning.
  - e.g., if I fly into Kingston and drive a car to Port Antonio, I'll have to drive on the roads at night. How desirable is this?
  - If I'm in Port Antonio and leave at 6:30am, I can arrive a Dunns river falls by 8:00am.

How do we plan our holiday?

- This kind of hypothetical reasoning involves asking
  - "what state will I be in after the following sequence of events?"
- From this we can reason about what sequence of events one should try to bring about to achieve a desirable state.
- Search is a computational method for capturing a particular version of this kind of reasoning.

Search

- There are many difficult questions that are not resolved by search. In particular, the whole question of how does an intelligent system formulate its problem as a search problem is not addressed by search.
- Search only shows how to solve the problem once we have it correctly formulated.

The formalism.

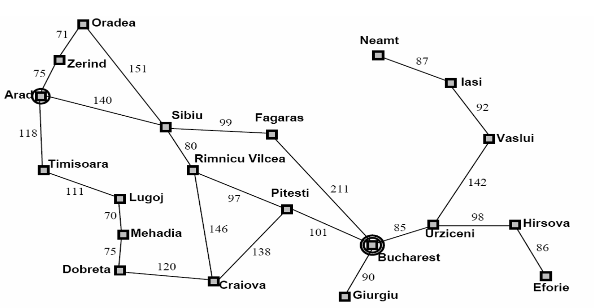
- To formulate a problem as a search problem we need the following components:
  - Formulate a state space over which to search. The state space necessarily involves abstracting the real problem.
  - Formulate actions that allow one to move between different states. The actions are abstractions of actions you could actually perform.
  - Identify the initial state that best represents your current state and the desired condition one wants to achieve.
  - Formulate various heuristics to help guide the search process.

The formalism.

- Once the problem has been formulated as a state space search, various algorithms can be utilized to solve the problem.
  - A solution to the problem will be a sequence of actions/moves that can transform your current state into state where your desired condition holds.

Example 1: Romania Travel.

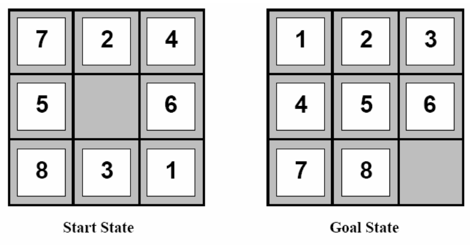
Currently in Arad, need to get to Bucharest by tomorrow to catch a flight.



Example 1.

- State space.
  - States: the various cities you could be located in.
    - Note we are ignoring the low level details of driving, states where you are on the road between cities, etc.
  - Actions: drive between neighboring cities.
  - Initial state: in Arad
  - Desired condition (Goal): be in a state where you are in Bucharest. (How many states satisfy this condition?)
- Solution will be the route, the sequence of cities to travel through to get to Bucharest.

Example 2. The 8-Puzzle



- Can slide a tile into the blank spot. (Equivalently, can think of it as moving the blank around).

Example 2. The 8-Puzzle

- State space.
  - States: The different configurations of the tiles. How many different states?
  - Actions: Moving the blank up, down, left, right. Can every action be performed in every state?
  - Initial state: as shown on previous slide.
  - Desired condition (Goal): be in a state where the tiles are all in the positions shown on the previous slide.
- Solution will be a sequence of moves of the blank that transform the initial state to a goal state.

Example 2. The 8-Puzzle

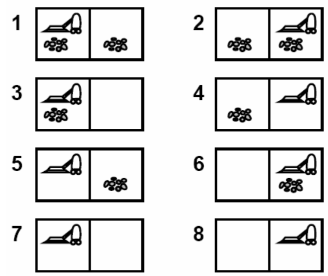
- Although there are 9! different configurations of the tiles (362,880), in fact the state space is divided into two disjoint parts.
- Only when the blank is in the middle are all four actions possible.
- Our goal condition is satisfied by only a single state. But one could easily have a goal condition like
  - The 8 is in the upper left hand corner.
    - How many different states satisfy this goal?

Example 3. Vacuum World.

- In the previous two examples, a state in the search space corresponded to a unique state of the world (modulo details we have abstracted away).
- However, states need not map directly to world configurations. Instead, a state could map to the agent's mental conception of how the world is configured: the agent's knowledge state.

Example 3. Vacuum World.

- We have a vacuum cleaner and two rooms.
- Each room may or may not be dirty.
- The vacuum cleaner can move left or right (the action has no effect if there is no room to the right/left).
- The vacuum cleaner can suck; this cleans the room (even if the room was already clean).

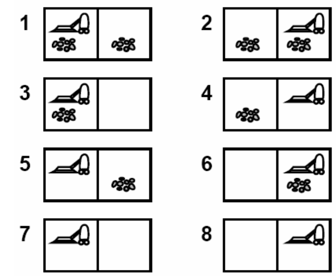


Physical states

Example 3. Vacuum World.

Knowledge level State Space

- The state space can consist of a set of states. The agent knows that it is in one of these states, but doesn't know which.

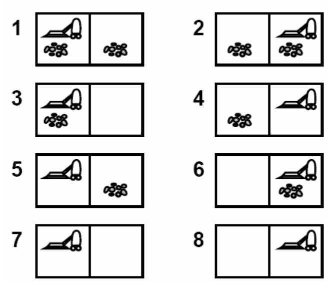


Goal is to have all rooms clean.

Example 3. Vacuum World.

Knowledge level State Space

- Complete knowledge of the world: agent knows exactly which state it is in. State space states consist of single physical states:
- Start in {5}:
  <right, suck>

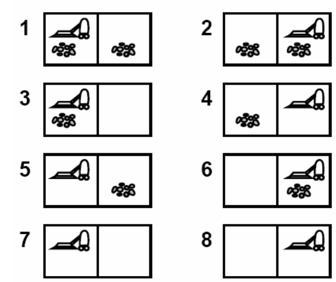


Goal is to have all rooms clean.

Example 3. Vacuum World.

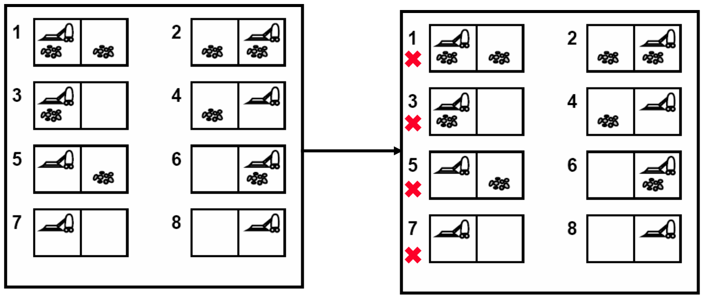
Knowledge level State Space

- No knowledge of the world. States consist of sets of physical states.
- Start in {1,2,3,4,5,6,7,8}, agent doesn't have any knowledge of where it is.
- Nevertheless, the actions <right, suck, left, suck> achieves the goal.



Goal is to have all rooms clean.

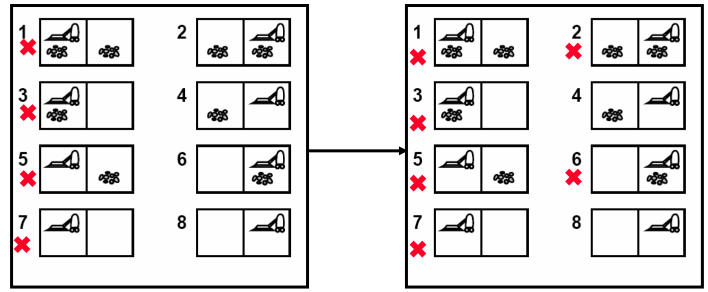
Example 3. Vacuum World.



Initial state.
{1,2,3,4,5,6,7,8}

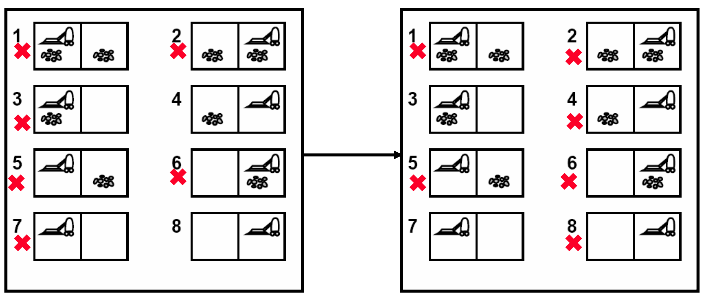
Right

Example 3. Vacuum World.



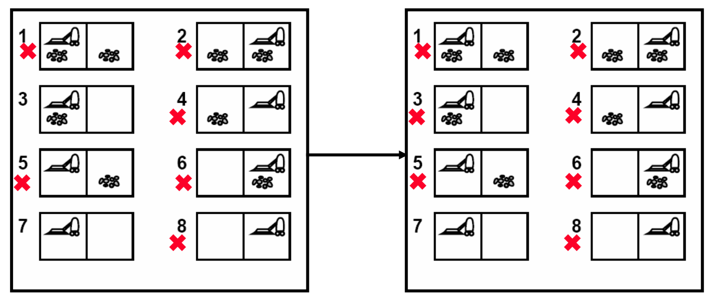
Suck

Example 3. Vacuum World.



Left

Example 3. Vacuum World.



Suck

More complex situations.

- The agent might be able to perform some sensing actions. These actions change the agent's mental state, not the world configuration.
- With sensing can search for a contingent solution: a solution that is contingent on the outcome of the sensing actions
  - <right, **if** dirt **then** suck>
- Now the issue of interleaving execution and search comes into play.

More complex situations.

- Instead of complete lack of knowledge, the agent might think that some states of the world are more likely than others.
- This leads to probabilistic models of the search space and different algorithms for solving the problem.
- Later we will see some techniques for reasoning and making decisions under uncertainty.

Algorithms for Search.

- Inputs:
  - a specified initial state (a specific world state or a set of world states representing the agent's knowledge, etc.)
  - a successor function S(x) = {set of states that can be reached from state x via a single action}.
  - a goal test a function that can be applied to a state and returns true if the state is satisfies the goal condition.
  - A step cost function C(x,a,y) which determines the cost of moving from state x to state y using action a. (C(x,a,y) = ∞ if a does not yield y from x)

Algorithms for Search.

- Output:
  - a sequence of states leading from the initial state to a state satisfying the goal test.
  - The sequence might be
  - annotated by the name of the action used.
  - optimal in cost for some algorithms.

Algorithms for Search

- Obtaining the action sequence.
  - The set of successors of a state x might arise from different actions, e.g.,
    - x → a → y
    - x → b → z
    - Successor function S(x) yields a set of states that can be reached from x via a (any) single action.
      - Rather than just return a set of states, we might annotate these states by the action used to obtain them:
        - S(x) = {<y,a>, <z,b>}
          y via action a, z via action b.
        - S(x) = {<y,a>, <y,b>}
          y via action a, also y via alternative action b.

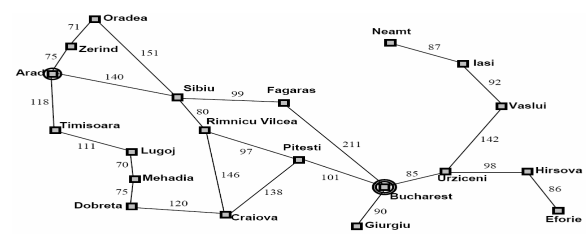
Tree search

- Assuming search space is a tree, not a graph.
- We use the successor state function to simulate an exploration of the state space.
- Initial call has Frontier = initial state.
  - Frontier/fringe is the set of states we haven't yet explored/expanded.

```
TreeSearch(Frontier, Successors, Goal? )
  If Frontier is empty return failure
  Curr = select state from Frontier
  If(Goal?(Curr)) return Curr.
  Frontier' = (Frontier – {Curr}) U Successors(Curr)
  return TreeSearch(Frontier', Successors, Goal?)
```


Tree search in Prolog

```
treeS([[State|Path]|_],Soln) :-
  Goal?(State), reverse([State|Path], Soln).
treeS([[State|Path]|Frontier],Soln) :-
  GenSuccessors(State,Path,NewPaths),
  merge(NewPaths,Frontier,NewFrontier),
  treeS(NewFrontier,Soln).
```

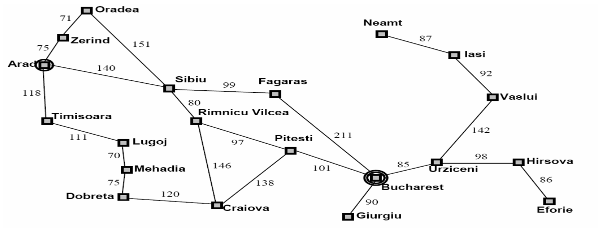
{Arad},

{Zerind, Timisoara, Sibiu},

{Zerind, Timisoara, Arad, Oradea, Fagaras, RimnicuVilcea },

{Zerind, Timisoara, Arad, Oradea, Sibiu, Bucharest, RimnicuVilcea },

Solution: Arad -> Sibiu -> Fagaras -> Bucharest
Cost: 140+99+211 = 450



{Arad},

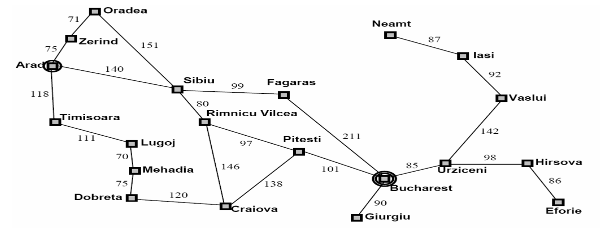
{Zerind, Timisoara, Sibiu},

{Zerind, Timisoara, Arad, Oradea, Fagaras, RimnicuVilcea},

{Zerind, Timisoara, Arad, Oradea, Sibiu, Pitesi, Craiova<via RimnicuVilcea>},

{Zerind, Timisoara, Arad, Oradea, Sibiu, Craiova<via Pitesi>, Bucharest, Craiova<via RimnicuVilcea>},

- Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
  Cost: 140+80+97+101 = 418



{Arad<>},

{Zerind<Arad>, Timisoara<Arad>, Sibiu<Arad>},

{Zerind <Arad>, Timisoara <Arad>, Oradea <Sibiu;Arad>, Fagaras<Sibiu;Arad>, Arad<Sibiu;Arad>, RimnicuVilcea<Sibiu;Arad>},

{Zerind <Arad>, Timisoara <Arad>, Oradea <Sibiu;Arad>, Fagaras<Sibiu;Arad>, Zerind<Arad;Sibiu;Arad>, Timisoara<Arad;Sibiu;Arad>, Sibiu<Arad;Sibiu;Arad>, RimnicuVilcea<Sibiu;Arad>},

No solution found, search does not terminate because of cycles!

Selection Rule.

- The example shows that order states are selected from the frontier has a critical effect on the operation of the search.
  - Whether or not a solution is found
  - The cost of the solution found.
  - The time and space required by the search.

Critical Properties of Search.

- Completeness: will the search always find a solution of a solution exists?
- Optimality: will the search always find the least cost solution? (when actions have costs)
- Time complexity: what is the maximum number of nodes than can be expanded or generated?
- Space complexity: what is the maximum number of nodes that have to be stored in memory?

Uninformed Search Strategies

- These are strategies that adopt a fixed rule for selecting the next state to be expanded.
- The rule is always the same whatever the search problem being solved.
- These strategies do not take into account any domain specific information about the particular search problem.
- Popular uninformed search techniques:
  - Breadth-First, Uniform-Cost, Depth-First, Depth-Limited, and Iterative-Deepening search.

Selecting vs. Sorting

- A simple equivalence we will exploit:
  - Order the elements on the frontier.
  - Always select the first element.
- Any selection rule can be achieved by employing an appropriate ordering of the frontier set.

Breadth First.

- Place the successors of the current state at the end of the frontier, which then behaves as a FIFO queue.
- Example:
  - let the states be the positive integers {0,1,2,…}
  - let each state n have as successors n+1 and n+2
    - E.g. S(1) = {2, 3}; S(10) = {11, 12}
  - Start state 0
  - Goal state 5
  - [Draw search space graph]

Breadth First Example.

{0}
{1,2}
{2,2,3}
{2,3,3,4}
{3,3,4,3,4}
{3,4,3,4,4,5}
…

- [Draw search tree]

Breadth First Properties

- Measuring time and space complexity.
  - let b be the maximum number of successors of any state.
  - let d be the number of actions in the shortest solution.

Breadth First Properties

- Completeness?
  - The length of the path from the initial state to the expanded state must increase monotonically.
    - we replace each expanded state with states on longer paths.
    - All shorter paths are expanded prior before any longer path.
  - Hence, eventually we must examine all paths of length d, and thus find the shortest solution.

Breadth First Properties

- Time Complexity?
  - # nodes generated at…
  - Level 0 (root): 1
  - Level 1: 1* b [each node has at most b successors]
  - Level 2: $b * b = b^2$
  - Level 3: $b * b^2 = b^3$ ....
  - Level d: b^d
  - Level d + 1: $b^{d+1} - b = b(b^d - 1)$ [when last node is successful]
  - Total: $1 + b + b^2 + b^3 + \ldots + b^{d-1} + b^d + b(b^d - 1) = O(b^{d+1})$
  - Exponential, so can only solve small instances

Breadth First Properties

- Space Complexity?
  - $O(b^{d+1})$: If goal node is last node at level d, all of the successors of the other nodes will be on the frontier when the goal node is expanded, i.e. $b(b^d - 1)$

Breadth First Properties

- Optimality?
  - Will find shortest path length solution
  - Least cost solution?
    - In general no!
    - Only if all step costs are equal

Breadth First Properties

- Space complexity is a real problem.
  - E.g., let b = 10, and say 1000 nodes can be expanded per second and each node requires 100 bytes of storage:

Depth	Nodes	Time	Memory
1	1	1 millisec.	100 bytes
6	10^6	18 mins.	111 MB
8	10^8	31 hrs.	11 GB

- Run out of space long before we run out of time in most applications.

Uniform Cost Search.

- Keep the frontier sorted in increasing cost of the path to a node; behaves like priority queue.
- Always expand the least cost node.
- Identical to Breadth First if each transition has the same cost.

- Example:
  - let the states be the positive integers {0,1,2,…}
  - let each state n have as successors n+1 and n+2
  - Say that the n+1 action has cost 2, while the n+2 action has cost 3.
  - [Draw search space graph]

Uniform Cost Search.

{0[0]}
{1[2],2[3]}
{2[3],2[4],3[5]}
{2[4],3[5],3[5],4[6]}
{3[5],3[5],4[6],3[6],4[7]}
…

Uniform-Cost Search

- Completeness?
  - Assume each transition has costs ≥ ϵ > 0 (otherwise can have in finite path with finite cost)
  - The previous argument used for breadth first search holds: the cost of the expanded state must increase monotonically.
  - The algorithm expands nodes in order of increasing path cost.

Uniform-Cost Search

- Time and Space Complexity?
  - $O(b^{C^*/\epsilon})$ where C* is the cost of the optimal solution.
    - Difficulty is that there may be many long paths with cost ≤ C*; Uniform-cost search must explore them all.

Uniform-Cost Search

- Optimality?
  - Finds optimal solution if each transition has cost ≥ ϵ > 0.
    - Explores paths in the search space in increasing order of cost. So must find minimum cost path to a goal before finding any higher costs paths.

Uniform-Cost Search. Proof of Optimality.

1. Let c(n) be the cost of the path to node n. If n2 is expanded after n1 then c(n1) ≤ c(n2).

Proof:

- If n2 was on the frontier when n1 was expanded, in which case c(n2) ≥ c(n1) else n1 would not have been selected for expansion.
- If n2 was added to the frontier when n1 was expanded, in which case c(n2) ≥ c(n1) since the path to n2 extends the path to n1.
- If n2 is a successor of a node n3 that was on the frontier or added when n1 was expanded, then c(n2) > c(n3) and c(n3) ≥ c(n1) by the above arguments.

Uniform-Cost Search. Proof of Optimality.

2. When n is expanded every path with cost strictly less than c(n) has already been expanded (i.e., every node on it has been expanded).

Proof:

- Let <Start, n0, n1, …, nk> be a path with cost less than c(n). Let ni be the last node on this path that has been expanded. <Start, n0, n1, ni-1, ni, ni+1, …, nk>.
- ni+1 must be on the frontier, also c(ni+1) < c(n) since the cost of the entire path to nk is < c(n).
- But then uniform-cost would have expanded ni+1 not n!
- So every node on this path must already be expanded, i.e. this path has already been expanded. QED

Uniform-Cost Search. Proof of Optimality.

3. The first time uniform-cost expands a state, it has found the minimal cost path to it (it might later find other paths to the same state).

Proof:

- No cheaper path exists, else that path would have been expanded before.
- No cheaper path will be discovered later, as all those paths must be at least as expensive.
- So, when a goal state is expanded, the path to it must be optimal.

Depth First Search

- Place the successors of the current state at the front of the frontier.
- Frontier behaves like a stack.

Depth First Search Example

(applied to the example of Breadth First search)

{0}
{1,2}
{2,3,2}
{3,4,3,2}
{4,5,4,3,2}
{5,6,5,4,3,2}
…

- [draw search tree]

Depth First Properties

- Completeness? No!
  - Infinite paths cause incompleteness! Typically come from cycles in search space.
  - If we prune paths with duplicate states, get completeness provided the search space is finite.
- Optimality? No!
  - Can find success along a longer branch!

Depth First Properties

- Time Complexity?
  - $O(b^m)$ where m is the length of the longest path in the state space.
  - Why? In worst case, expands

  $1 + b + b^2 + \ldots + b^{m-1} + b^m = b^{m+1} - 1/b - 1 = O(b^m)$ nodes

  - Assumes no cycles.
  - Very bad if m is much larger than d, but if there are many solution paths it can be much faster than breadth first.

Depth First Backtrack Points

- At each step, all nodes in the frontier (except the head) are backtrack points (see example and draw the tree for state-space).
- These are all siblings of nodes on the current branch.

Depth First Properties

- Space Complexity?
  - $O(bm)$, linear space!
    - Only explore a single path at a time.
    - The frontier only contains the deepest states on the current path along with the backtrack points.
  - Can reduce to $O(m)$ if we generate siblings one at a time.

Depth Limited Search

- Breadth first has computational, especially, space problems. Depth first can run off down a very long (or infinite) path.
- Depth limited search.
  - Perform depth first search but only to a pre-specified depth limit L.
  - No node on a path that is more than L steps from the initial state is placed on the Frontier.
  - We “truncate” the search by looking only at paths of length L or less.
- Now infinite length paths are not a problem.
- But will only find a solution if a solution of length ≤ L exists.

Depth Limited Search

```
DLS(Frontier, Sucessors, Goal?)
   If Frontier is empty return failure
   Curr = select state from Frontier
   If(Goal?(Curr)) return Curr.
   If Depth(Curr) < L
       Frontier' = (Frontier - {Curr}) U Successors(Curr)
   Else
       Frontier' = Frontier - {Curr}
       CutOffOccured = TRUE.
   return DLS(Frontier', Successors, Goal?)
```


Iterative Deepening Search.

- Take the idea of depth limited search one step further.
- Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if no solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.

Iterative Deepening Search Example

{0} [DL = 0]

{0} [DL = 1]
{1,2}
{2}

{0} [DL = 2]
{1,2}
{2,3,2}, {3,2}, {2}
{3, 4}, {4}

{0} [DL = 3]
{1,2}
{2,3,2}
{3,4,3,2}, {4,3,2}, {3,2}
{4,5,2}, {5, 2}
Success!

Iterative Deepening Search Properties

- Completeness?
  - Yes, if solution of length d exists, will the search will find it when L = d.
- Time Complexity?
  - At first glance, seems bad because nodes are expanded many times.

Iterative Deepening Search Properties

- Time Complexity
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

    [root expanded d+1 times, level 1 nodes expanded d times, …]
  - E.g. b=4, d=10
    - $(11)*4^0 + 10*4^1 + 9*4^2 + \ldots + 2*4^9 = 815{,}555$
    - $4^{10} = 1{,}048{,}576$
    - Most nodes lie on bottom layer.
    - In fact IDS can be more efficient than breadth first search: nodes at limit are not expanded. BFS must expand all nodes until it expands a goal node.

Iterative Deepening Search Properties

- Space Complexity
  - O(bd) Still linear!
- Optimal?
  - Will find shortest length solution which is optimal if costs are uniform.
  - If costs are not uniform, we can use a “cost” bound instead.
    - Only expand paths of cost less than the cost bound.
    - Keep track of the minimum cost unexpanded path in each depth first iteration, increase the cost bound to this on the next iteration.
    - This can be very expensive. Need as many iterations of the search as there are distinct path costs.

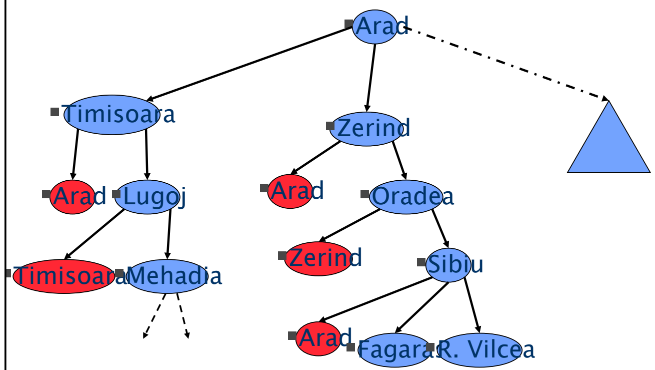
Iterative Deepening Search Properties

- Consider space with three paths of length 3, but each action having a distinct cost.

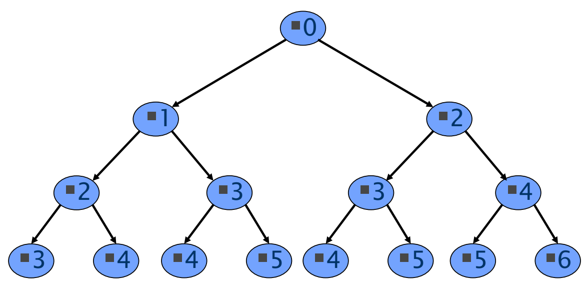
Cycle Checking

- Path checking
  - Paths are stored on the frontier (this allows us to output the solution path).
    - If $\langle S,n_1,\ldots,n_k\rangle$ is a path to node n_k, and we expand n_k to obtain child c, we have
      - $\langle S,n_1,\ldots,n_k,c\rangle$
    - As the path to "c".
  - Path checking:
    - Ensure that the state c is not equal to the state reached by any ancestor of c along this path.

Path Checking Example



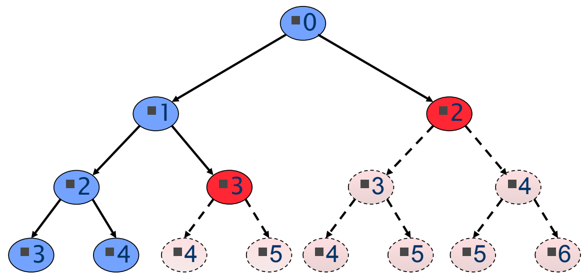
Path Checking Example



Cycle Checking

- Cycle Checking.
  - Keep track of all states previously expanded during the search.
  - When we expand n_k to obtain child c
    - ensure that c is not equal to any previously expanded state.
  - This is called cycle checking, or multiple path checking.
  - Why can't we utilize this technique with depth-first search?
    - If we use cycle checking in depth-first search what happens to space complexity.

Cycle Checking Example



Cycle Checking

- High space complexity, only useful with breadth first search.
- There is an additional issue when we are looking for an optimal solution
  - With uniform-cost search, we still find an optimal solution
    - The first time uniform-cost expands a state it has found the minimal cost path to it.
  - This means that the nodes rejected by cycle checking can’t have better paths.
  - We will see later that we don’t always have this property when we do heuristic search.