CSC384 Summer 2018

Week 2 - Search

Ilir Dema

University of Toronto

May 14, 2018

OVERVIEW

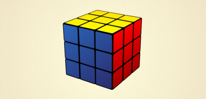
SEARCH

- One of the most fundamental techniques in AI
    - Underlying sub-module in many AI systems
- Can solve many problems that humans are not good at.
- Can achieving super-human performance on other problems (Chess, go)
- Very useful as a general algorithmic technique for solving problems (both in AI and in other areas)

HOW DO WE SOLVE RUBIK'S CUBE?

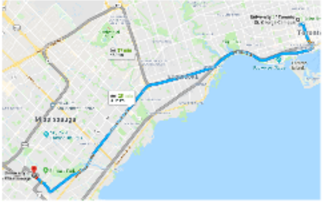
- Possible states: 43,252,003,274,489,856,000
- We must take into account various available tools and constraints to develop a plan.
- An important technique in developing such a plan is "hypothetical" reasoning.
- Example:
  - Label faces: F,B,U,D,L,R
  - Clockwise rotations by $\pi/2$ (F), counterclockwise rotations (f), individual piece (FUR), ...
  - Develop a sequence of moves: (example: R2drLF2lRU2DR2) ...
  - Will we solve it in our lifetime ?

How do we solve Rubik's cube?



- This kind of hypothetical reasoning involves asking
- what state will the cube be in after taking certain actions, or after certain sequences of moves?
- From this we can reason about particular sequences of actions one should execute to achieve a desirable state.
- Search is a computational method for capturing a particular version of this kind of reasoning.

MANY PROBLEMS CAN BE SOLVED BY SEARCH:



Many problems can be solved by search:

Deepblue 1997

beats Kasparov world champion Chess player

AlphaGo 2016

beats Lee Sedol 9th dan Go player

2017 beats Ke Jie World #1 ranked player

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 specific algorithms for solving them. Casting 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).
  - Problem specific heuristics provides search with a way of exploiting extra knowledge.
- Some critical aspects of intelligent behaviour, e.g., planning, can be naturally cast as search.

Limitations of search

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

Representing a problem: The Formalism

To formulate a problem as a search problem we need the following components:

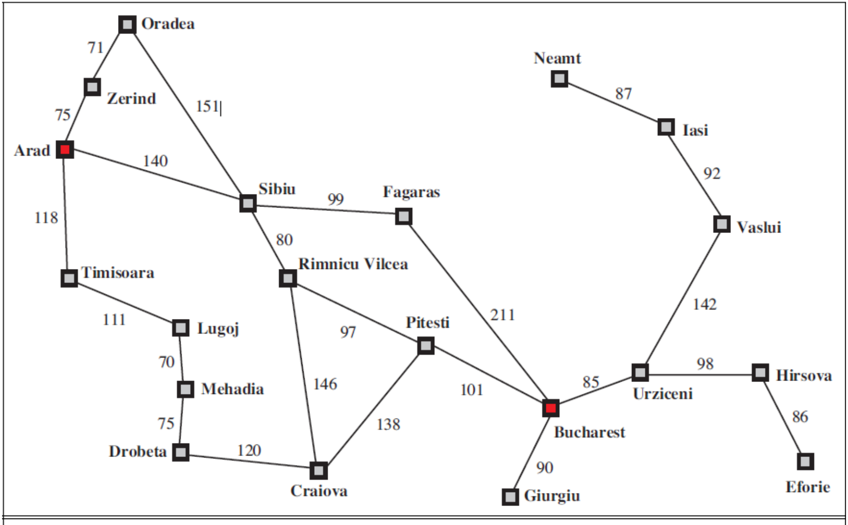
1. **STATE SPACE:** Formulate a state space over which we perform search. The state space is a way or representing in a computer the states of the real problem.
2. **ACTIONS** or **STATE SPACE Transitions:** Formulate actions that allow one to move between different states. The actions reflect the actions one can take in the real problem but operate on the state space instead.
3. **INITIAL** or **START STATE** and **GOAL:** Identify the initial state that best represents the starting conditions, and the goal or condition one wants to achieve.
4. **HEURISTICS:** 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 a state where your desired condition holds.

EXAMPLE 1

Currently in **Arad**, need to get to **Bucharest** by tomorrow to catch a flight. What is the **State Space**?



EXAMPLE 1

- State space.
    - **States**: the various cities you could be located in.
        - Our abstraction: 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

- Water Jugs
  - We have a 3 gallon jug and a 4 gallon jug. We can fill either jug to the top from a tap, we can empty either jug, or we can pour one jug into the other (at least until the other jug is full).
  - States: pairs of numbers (gal3, gal4)
    gal3 = the number of gallons in the 3 gallon jug
    gal4 = the number of gallons in the 4 gallon jug.
  - Actions: Empty-3-Gallon, Empty-4-Gallon, Fill-3-Gallon, Fill-4- Gallon, Pour-3-into-4, Pour 4-into-3.
  - Initial state: Various, e.g., (0,0)
  - Desired condition (Goal): Various, e.g., (0,2) or (*, 3) where * means we don't care.

Example 2

- Water Jugs
  - If we start off with gal3 and gal4 as integer, can only reach integer values.
  - Some values, e.g., (1,2) are not reachable from some initial state, e.g., (0,0).
  - Some actions are no-ops. They do not change the state, e.g.,
  - (0,0) Empty-3-Gallon (0,0)

Example 3. The 8-puzzle

Rule: Can slide a tile into the blank spot.
Alternative view: move the blank spot around.

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

EXAMPLE 3. 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:** e.g., state 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.

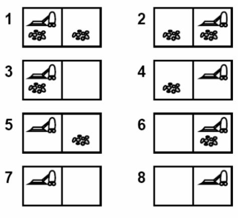
Example 3. The 8-puzzle

- Although there are 9! different configurations of the tiles (362,880) in fact the state space is divided into two disjoint parts.
    - states where the goal is reachable after a finite number of actions.
    - states where there does not exist a sequence of actions that leads to the goal.
- It can be shown that if the initial state (ignoring the hole) belongs to $\mathcal{A}_8$ the goal is reachable.
- That is, if the initial state can be generated by an even number of inversions in $\mathcal{S}_8$, the puzzle is solvable.

Example 4: Vacuum World

- In the previous examples, a state in the search space represented some a particular state of the world.
- However, states need not map directly to world configurations. Instead, a state could map to knowledge states.
- If you know the exact state of the world your knowledge state is a single unique state.
- If you don't know some things, then your knowledge state is a set of world state - every world state that you believe to be possible.

EXAMPLE 4: VACUUM WORLD



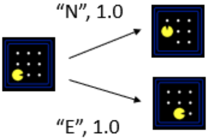
Goal is to have all rooms clean

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

EXAMPLE 5. PACMAN

- A state space

- A successor function (with actions, costs)

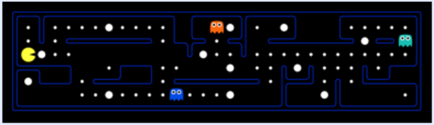


- A start state and a goal test

- A **solution** is a sequence of actions (a plan) which transforms the start state to a goal state

Example 5. Pacman

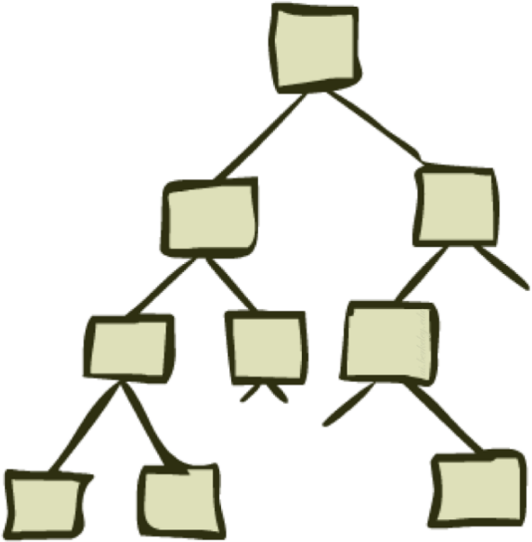
The world state includes every last detail of the environment



A search state keeps only the details needed for planning (abstraction)

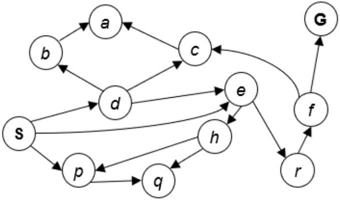
- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END
- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

State Space Graphs and Search Trees



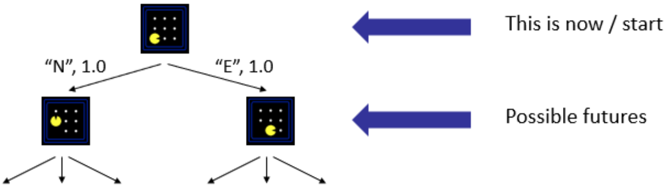
State Space Graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a search graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



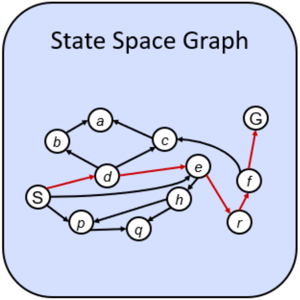
Tiny search graph for a tiny search problem

Search Trees



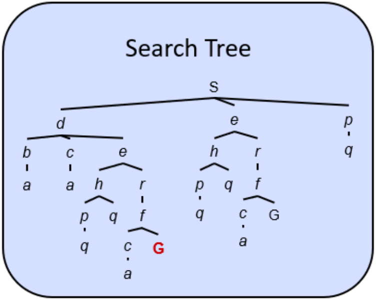
- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
  - For most problems, we can never actually build the whole tree

State Space Graphs vs. Search Trees



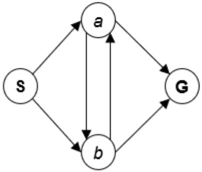
Each NODE in in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.



State Space Graphs vs. Search Trees

Consider this 4-state graph:



How big is its search tree (from S)?

$$\infty$$

Important: Lots of repeated structure in the search tree!

ALGORITHMS FOR SEARCH

- AI search algorithms work with implicitly defined state spaces.
- There are typically an exponential number of states: impossible to explicitly represent them all.
- The space of possible configurations of a Go board is about 3361 (standard 19X19 board).
- There are even more actions than state.

ALGORITHMS FOR SEARCH

- In AI search we find solutions by constructing only those states we need to. In the worst case we will need to construct an exponential number of states - and the search will be unsuccessful.
- But often we can solve hard problems (like Go) while only examining a small fraction of the states.
- Hence the actions are given as compact functions or programs that when given a state S construct and return the states S can be transformed to by the available actions.
- This means that the state must contain enough information to allow this function to perform its computation.

ALGORITHMS FOR SEARCH

- Inputs:
  - a specified initial state (a specific world state)
  - a successor function $S(x)$ yields a 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 satisfies the goal condition.
  - An action cost function $C(x,a,y)$ which determines the cost of moving from state x to state y using action a. ($C(x,a,y) = \infty$ if a does not yield y from x). Note that different actions might generate the same move of $x \rightarrow y$.

Algorithms for search

- Output:
  - a sequence of actions that transform the initial state to a state satisfying the goal test.
  - Or just the sequence of states that arise from these actions (depends on what kind of information is most useful)
  - The sequence might be, optimal in cost for some algorithms, optimal in length for some algorithms, come with no optimality guarantees from other algorithms.
  - That is, no other sequence transforms the initial state to a goal satisfying state with lower cost (or lesser length).

ALGORITHMS FOR SEARCH

Obtaining the action sequence.

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

Algorithms for search

- The search space consists of **states** and actions that move between states.
- A **path** in the search space is a **sequence of states** connected by actions, $\langle s_0, s_1, \ldots, s_k \rangle$,
- for every s_i and its successor s_{i+1} there must exist an action a_i that transitions s_i to s_{i+1}.
- Alternately a path can be specified by
  (A) an initial state s_0, and
  (B) a sequence of actions that are applied in turn starting from s_0.

ALGORITHMS FOR SEARCH

- The search algorithms perform search by examining alternate paths of the search space. The objects used in the algorithm are called **nodes** - each node contains a path.
- In practice the path might be stored as a pointer from a node data structure to its parent node. Following those pointers to the initial state yields the path.

ALGORITHMS FOR SEARCH

- We maintain a set of nodes called the `OPEN` set (or frontier).
  - These nodes are paths in the search space that all start at the initial state.
- Initially we set `OPEN={<Start State>}`.
  - The path (node) that starts and terminates at the start state.
- At each step we select a node `n` from `OPEN`.

```
n is a path so let x be the state n terminates at.
      x = n.end_state()
We check if x satisfies the goal,
if not we add all extensions of n to OPEN
   for all successor states y of x, extend n
       to go from x->y
```

Algorithms for Search

```
Search(open, successors, goal? ):
    open.insert(<start>)
    while not open.empty():
        n = open.extract() #remove node from OPEN
        state = n.end_state()
        if (goal?(state)):
            return n #n is solution
        for succ in sucessors(state):
            open.insert(succ)
                #open could grow or shrink
    return false
```

Algorithms for search

- When a node n is extracted from open, we say that the algorithm **expands n**.
- The number of states we actually construct (as items returned by successors(), we hope is low compared to the total number of states.
- The number of states expanded depends on the order of nodes we extract from open.

Selection Rule

The order paths are selected from OPEN 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.

How to select the next path from OPEN?

- All search techniques keep OPEN as an ordered set and repeatedly execute:
  - Expand out potential plans (tree nodes)
  - Maintain OPEN
  - Try to expand as few tree nodes as possible
- How do we order the paths on OPEN?

Critical Properties of Search

- **Completeness:** will the search always find a solution if 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 (paths) than can be expanded or generated?
- **Space complexity:** what is the maximum number of nodes (paths) 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 does not change, particular properties of the search problem being solved are ignored.
- Uninformed search techniques:

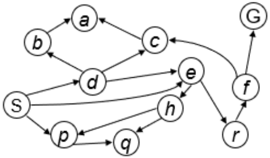
  Breadth-First, Uniform-Cost, Depth-First, Depth-Limited, and Iterative- Deepening search

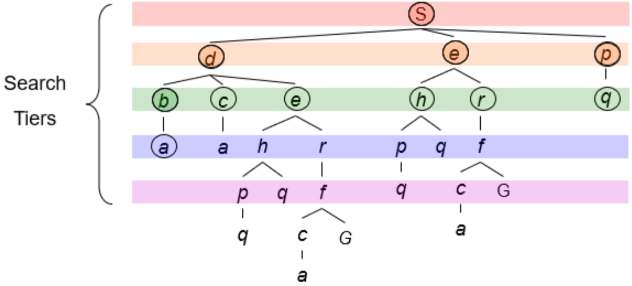
Uninformed Search

- You would have seen breadth-first search and depth-first search in CSC263/265.
  - Graph nodes = state space states
  - Graph edges = state space actions
- In that course however, it is assumed that the graph we are searching is explicitly represented as an adjacency list (or adjacency matrix).
  - This won't work when there an exponential number of and edges.
- Similarly uniform cost search is like Dijkstra's algorithm, but without an explicitly represented graph.
- All of these algorithms are simple instantiations of our implicit graph search.

Breadth-First search

Strategy: expand a shallowest node first
Implementation: OPEN is a FIFO queue



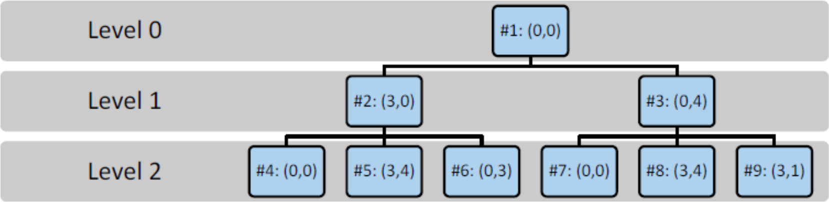


BFS - WATERJUG EXAMPLE

`Start=(0,0), Goal=(*,2).`
Place the new paths that extend the current path at the end of `OPEN`. Extract first element of `OPEN`
Red=Expanded next, Green=newly added

1. $\texttt{OPEN}=\{\langle(0,0)\rangle\}$
2. $\texttt{OPEN}=\{\langle(0,0),(3,0)\rangle,\langle(0,0),(0,4)\rangle\}$
3. $\texttt{OPEN}=\{\langle(0,0),(0,4)\rangle,\langle(0,0),(3,0),(0,0)\rangle,$ $\langle(0,0),(3,0),(3,4)\rangle,\langle(0,0),(3,0),(0,3)\rangle\}$
4. $\texttt{OPEN}=\{\langle(0,0),(3,0),(0,0)\rangle,\langle(0,0),(3,0),(3,4)\rangle,$ $\langle(0,0),(3,0),(0,3)\rangle,\langle(0,0),(0,4),(0,0)\rangle,$ $\langle(0,0),(0,4),(3,4)\rangle,\langle(0,0),(0,4),(3,1)\rangle\}$

Breadth-First Search



- Above we indicate only the state that each nodes terminates at. The path represented by each node is the path from the root to that node.
- Breadth-First explores the search space level by level.

Breadth-First Properties

- Completeness?
  - The length of the path removed from OPEN is non-decreasing.
    - We replace each expanded node n with an extension of n.
    - All shorter paths are expanded prior before any longer path.
  - Let d be the solution up to depth d.
  - Hence, eventually we must examine all paths of length d, and thus find a solution if one exists.
- Optimality?
  - By the above will find shortest length solution
    - Least cost solution?
    - Not necessarily: shortest solution not always cheapest solution if actions have varying costs

Breadth-First Search

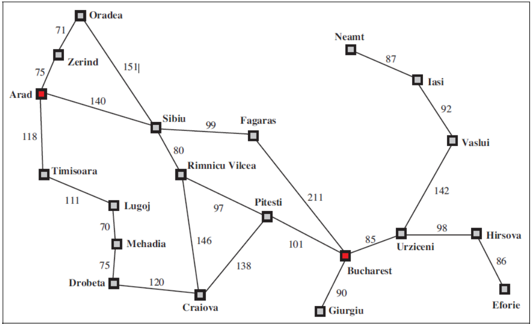


Figure 3.2 **FILES: figures/romania-distances.eps (Tue Nov 3 16:23:37 2009).** A simplified road map of part of Romania.

Breadth first Solution: Arad -> Sibiu -> Fagaras -> Bucharest

Cost: 140 + 99 + 211 = 450

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

Breadth-First Properties

- Measuring time and space complexity.
  - let b be the maximum number of successors of any node (maximal branching factor).
  - let d be the depth of the shortest solution.
  - Root at depth 0 is a path of length 1
  - So length of path $= d + 1$
- Time Complexity?

$$1 + b + b^2 + \cdots + b^{d-1} + b^d + b(b^d - 1) = O(b^{d+1})$$

- 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 OPEN when the goal node is expanded $b(b^d - 1)$.

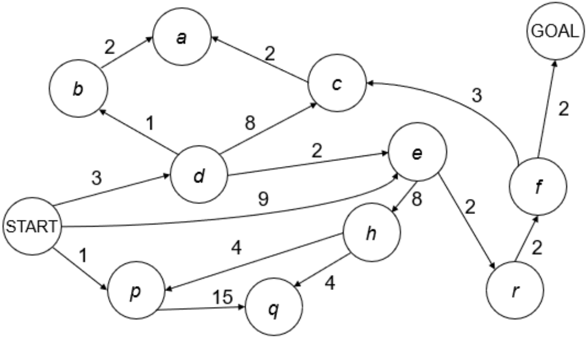
Breadth-First properties

Depth	Nodes	Time	Memory
1	1	0.01 millisec.	100 bytes
6	10^6	10 sec.	100 MB
8	10^8	17 min.	10 GB
9	10^9	3 hrs.	100 GB

Space complexity is a real problem.

- E.g., let b = 10, and say 100,000 nodes can be expanded per second and each node requires 100 bytes of storage.
- Typically run out of space before we run out of time in most applications.

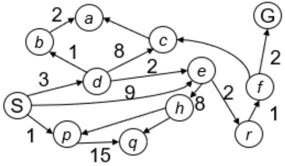
COST-SENSITIVE SEARCH

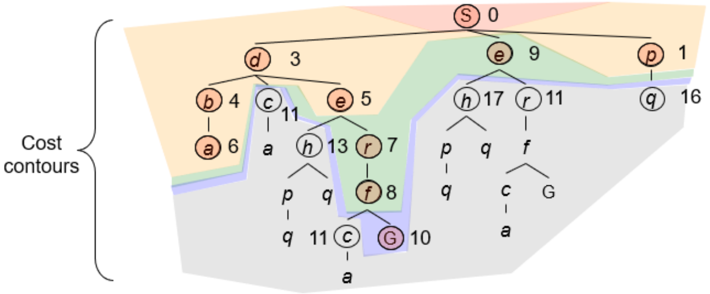


BFS finds the shortest path in terms of number of actions.
It does not find the least-cost path. We will now cover
a similar algorithm which does find the least-cost path.

UNIFORM-COST SEARCH

Strategy: expand a cheapest node first:
OPEN is a priority queue (priority: cumulative cost)





Identical to Breadth first if each action has the same cost.

Uniform-Cost Properties

- Completeness?
  - If each transition has costs $\geq \varepsilon > 0$
  - The previous argument used for breadth first search holds: the cost of the path represented by each node n chosen to be expanded must be non-decreasing.
- Optimality?
  - Finds optimal solution if each transition has costs $\geq \varepsilon > 0$
  - 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 leading to a goal

Uniform-Cost Search. Proof of Optimality

Let us prove Optimality more formally. We will reuse this argument later on when we examine Heuristic Search

Uniform-Cost Search. Proof of Optimality

Lemma 1.

Let $c(n)$ be the cost of node n on OPEN (cost of the path represented by n). If n_2 is expanded IMMEDIATELY after n_1 then $c(n_1) \leq c(n_2)$.

Proof: There are two cases:

(A) n_2 was in OPEN when n_1 was expanded:
We must have $c(n_1) \leq c(n_2)$ otherwise n_2 would have been selected for expansion rather than n_1.

(B) n_2 was added to OPEN when n_1 was expanded
Now $c(n_1) < c(n_2)$ since the path represented by n_2 extends the path represented by n_1 and thus cost at least ε more.

Uniform-Cost Search. Proof of Optimality

Lemma 2.

When node n is expanded every path in the search space with cost strictly less than $c(n)$ has already been expanded.

Proof:

- Let $n_k = \langle \text{Start}, s_1, \ldots, s_k \rangle$ be a path with cost less than $c(n)$. Let $n_0 = \langle \text{Start} \rangle, n_1 = \langle \text{Start}, s_1 \rangle, n_2 = \langle \text{Start}, s_1, s_2 \rangle, \ldots,$ $n_i = \langle \text{Start}, s_1, \ldots, s_i \rangle, \ldots, n_k = \langle \text{Start}, s_1, \ldots, s_k \rangle$. Let n_i be the last node in this sequence that has already been expanded by search.
- So, n_{i+1} must still be on OPEN: it was added to open when n_i was expanded. Also $c(n_{i+1}) \leq c(n_k) < c(n) : c(n_{i+1})$ is a subpath of n_k we have assumed that $c(n_k) < c(n)$.
- But then uniform-cost would have expanded n_{i+1} not n.
- So every node n_i including n_k must already be expanded, i.e., this lower cost path has already been expanded.

Uniform-Cost Search. Proof of Optimality

Lemma 3.

The first time uniform-cost expands a node n terminating at state S, it has found the minimal cost path to S (it might later find other paths to S but none of them can be cheaper).

Proof:

- All cheaper paths have already been expanded, none of them terminated at S.
- All paths expanded after n will be at least as expensive, so no cheaper path to S can be found later.

So, when a path to a goal state is expanded the path must be optimal (lowest cost).

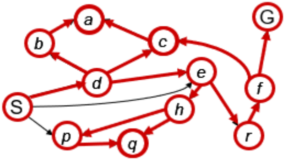
Uniform-Cost Properties

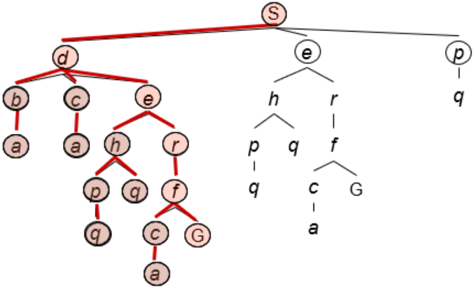
Time and Space Complexity?

- $O(b^{C^*/\varepsilon})$ where C^* is the cost of optimal solution.
- There are many paths with cost $\leq C^*$.
  Paths with cost lower than C^* can be as long as C^*/ε (why no longer?), there are up $b^{C^*/\varepsilon}$ paths with C^*/ε and we are to explore them all before finding an optimal cost path.

DEPTH-FIRST SEARCH

Strategy: expand a deepest node first
Implementation: OPEN is a LIFO stack





DFS - Waterjug example

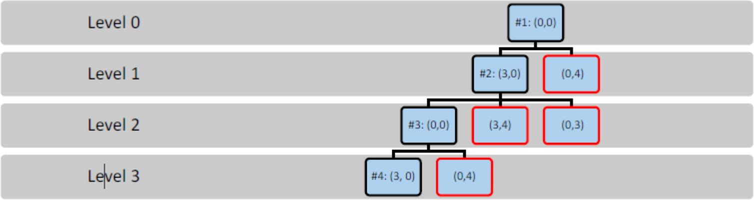
`Start=(0,0), Goal=(*,2).`

Place the new paths that extend the current path at the end of `OPEN`. Extract first element of `OPEN`

Red=Expanded next, Green=newly added

1. `OPEN=`$\{\langle(0,0)\rangle\}$
2. `OPEN=`$\{\langle(0,0),(3,0)\rangle, \langle(0,0),(0,4)\rangle\}$
3. `OPEN=`$\{\langle(0,0),(3,0),(0,0)\rangle, \langle(0,0),(3,0),(3,4)\rangle, \langle(0,0),(3,0),(0,3)\rangle, \langle(0,4),(0,0)\rangle\}$
4. `OPEN=`$\{\langle(0,0),(3,0),(0,0),(3,0)\rangle, \langle(0,0),(3,0),(0,0),(0,4)\rangle, \langle(0,0),(3,0),(3,4)\rangle, \langle(0,0),(3,0),(0,3)\rangle, \langle(0,0),(0,4)\rangle\}$

Depth-First Search



Red nodes are backtrack points (these nodes remain on open).

Depth-First Properties

- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m (tiers) is finite, takes time $O(b^m)$
- How much space does the fringe take?
  - Only has siblings on path to root, so $O(bm)$
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost
- A significant advantage of DFS
  - Only explore a single path at a time.
  - OPEN only contains the deepest node on the current path along with the backtrack points.

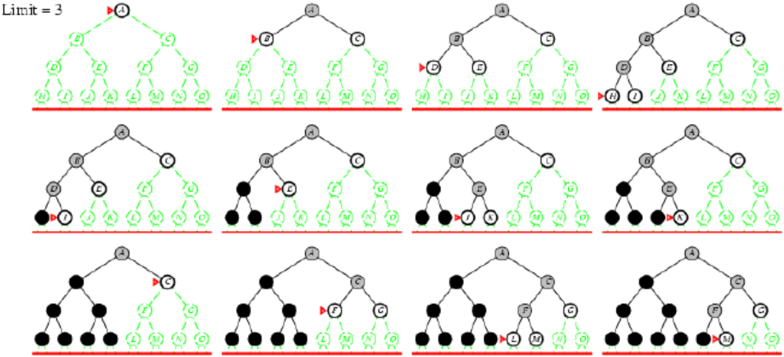
Depth Limited Search

- Breadth first has 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 D.
    The ROOT is at DEPTH 0. ROOT is a path of length 1.
  - No node representing a path of length more than D+1 is placed on OPEN.
  - We “truncate” the search by looking only at paths of length D+1 or less.
- Now infinite length paths are not a problem.
- But will only find a solution if a solution of DEPTH $\leq$ D exists.

Depth Limited Search

```
DL_Search(open, successors, goal?, maxd):
  open.insert(<start>) #OPEN MUST BE A STACK FOR DFS
  cutoff = false
  while not open.empty():
    n = open.extract() #remove node from OPEN
    state = n.end_state()
    if (goal?(state)):
      return (n,cutoff) #n is solution
    if depth(n) < maxd:
      #Only successors if depth(n) < maxd
      for succ in sucessors(state):
        open.insert(<n,succ>)
    else:
      cutoff= true. #some node was not
          #expanded because of depth
          #limit.
  return (false, cutoff)
```

DEPTH LIMITED SEARCH EXAMPLE



Iterative Deepening Search

- Solve the problems of depth-first and breadth-first by extending depth limited search
- Starting at depth limit L = 0, we iteratively increase the depth limit, performing a depth limited search for each depth limit.
- Stop if a solution is found, or if the depth limited search failed without cutting off any nodes because of the depth limit.
  - If no nodes were cut off, the search examined all paths in the state space and found no solution, this implies no solution exists.

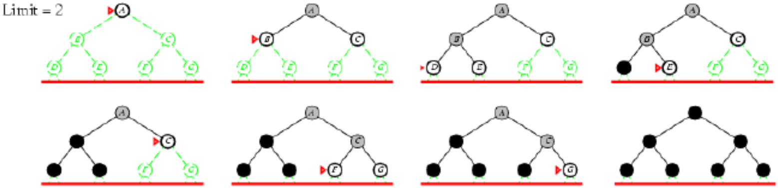
Iterative Deepening Search

```
ID_Search(open, successors, goal?):
  maxd = 0
  while true:
    (n, cutoff) =
          DL_Search(open, successors, goal?, maxd)
    if n:
      return n
    elif not cutoff: #no nodes at deeper levels exit
      return fail
    else:
      maxd = maxd + 1
```

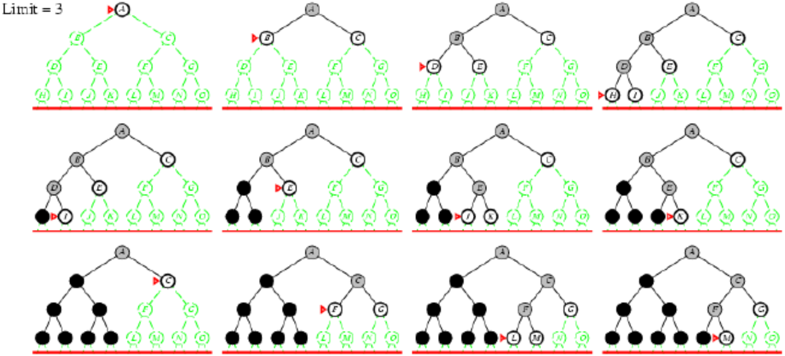
ITERATIVE DEEPENING SEARCH EXAMPLE

Limit = 1

Iterative Deepening Search Example



Iterative Deepening Search Example



Iterative Deepening Search Properties

- Completeness?
  - Yes if a minimal depth solution of depth d exists.
  - What happens when the depth limit L=d?
  - What happens when the depth limit L<d?
- Time Complexity?
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d = O(b^d)$
  - E.g. $b = 4, d = 10$
    - $11 \cdot 4^0 + 10 \cdot 4^1 + 9 \cdot 4^2 + \cdots + 4^{10} = 1,864,131$
    - Most nodes lie on bottom layer.

BFS can explore more states than IDS!

- For IDS, the time complexity is
  - $(d+1)b^0 + db^1 + (d-1)b^2 + \cdots + b^d = O(b^d)$
- For BFS, the time complexity is
  - $1 + b + b^2 + \cdots + b^d + b(b^d - 1) = O(b^{d+1})$

E.g. b=4, d=10

- For IDS
- $11 \cdot 4^0 + 10 \cdot 4^1 + \cdots + 4^{10} = 1,864,131$ (states generated)
- For BFS

  $1 + 4 + 4^2 + \cdots + 4^{10} + 4(4^{10} - 1) = 5,592,401$ (states generated)

  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. So a the bottom layer it will add many nodes to OPEN before finding the goal node.

ITERATIVE DEEPENING SEARCH PROPERTIES

Space Complexity?

- $O(bd)$

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 more expensive. Need as many iterations of the search as there are distinct path costs.

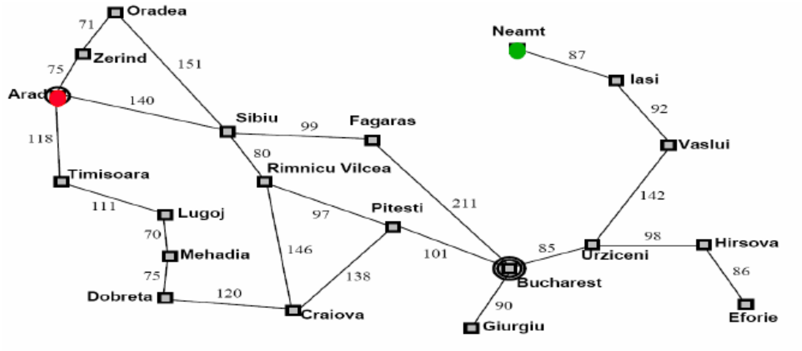
PATH CHECKING

- If n_k represents the path $\langle s_0, s_1, \ldots, s_k \rangle$ and we expand s_k to obtain child c, we have $\langle S, s_1, \ldots, s_k, c \rangle$ as the path to c.
- We write such paths as $\langle n, c \rangle$ where n is the prefix and c is the final state in the path.
- Path checking:
  - Ensure that the state c is not equal to the state reached by any ancestor of c along this path.
  - Paths are checked in isolation!

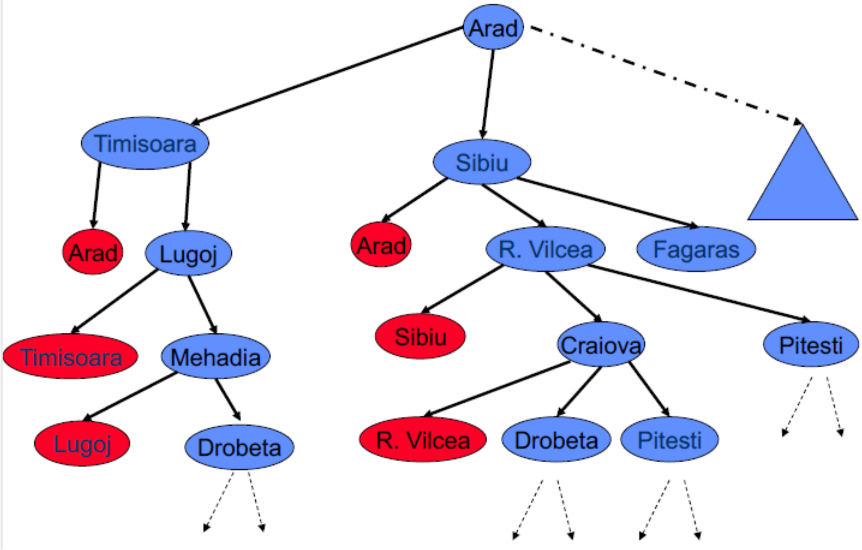
Search with Path Checking

```
Search(open, successors, goal? ):
   open.insert(<start>)
   while not open.empty():
      n = open.extract() #remove node from OPEN
      state = n.end_state()
      if (goal?(state)):
         return n #n is solution
      for succ in sucessors(state):
         if not succ in <n>: #put on OPEN
            open.insert(<n,succ>)
   return false
```

EXAMPLE


Oradea
71
Zerind
75
151
Arad
140
Sibiu
99
Fagaras
118
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Dobreta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
85
Urziceni
98
Hirsova
86
Bucharest
90
Giurgiu
Eforie

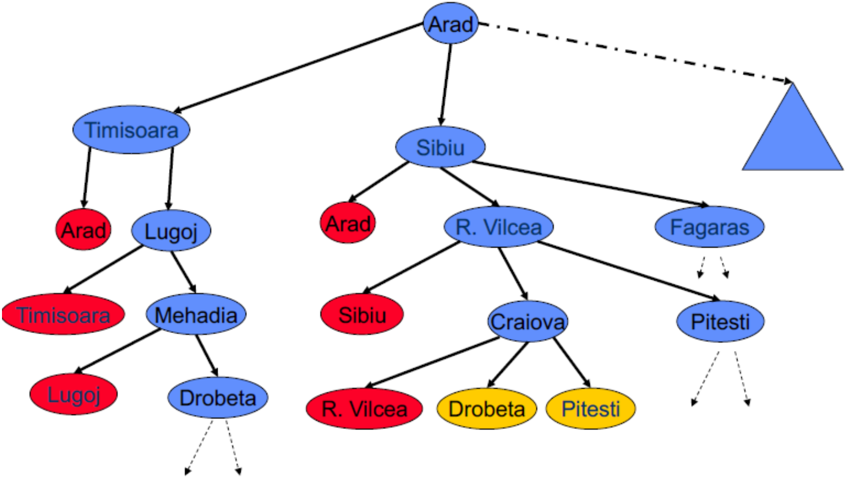
PATCH CHECKING XAMPLE



Cycle Checking

- Keep track of all states added to OPEN during the search (i.e., end state of every path added to OPEN)
- When we expand n_k to obtain successor state c
  - Ensure that c is not equal to any previously seen state.
  - If it is we do not add the path $\langle n_k, c\rangle$ to OPEN.
- This is called cycle checking, or multiple path checking.
- What happens when we utilize this technique with depth-first search?
- What happens to space complexity?

Cycle Checking Example (BFS)



CYCLE CHECKING

- Higher space complexity (equal to the space complexity of breadth-first search).
- There is an additional issue when we are looking for an optimal solution
  - If we reject a node $\langle n, c\rangle$ because we have previously seen its end state c it could be that $\langle n, c\rangle$ is a shorter path to c that we had previously seen.
  - Solution is to also keep track of the minimum cost path to each seen state.

Cycle Checking

- Keep track of each state as well as minimum known cost of a path to that state.
- If we find a longer path to a previously seen state, we don't add it to OPEN
- If we find a shorter path to a previously seen state, we add it to OPEN and
  - Remove other more expensive paths to the same state OR
  - Lazily remove these more expensive paths if and only if we decide to expand them.

Cycle Checking - ensuring optimality

```
Search(open, successors, goal? ):
  open.insert(<start>)
  seen = {start : 0} #seen is dict storing min cost
  while not open.empty():
    n = open.extract()
    state = n.end_state()
    if cost(n) <= seen[state]:
           #only expand if cheapest path
      if (goal?(state)):
        return n
      for succ in sucessors(state):
        if not succ in seen or cost(<n,succ>) <
                                              seen[succ]:
          open.insert(<n,succ>)
          seen[succ] = cost(<n,succ>)
  return false
```

References

Some of the slides were created by Dan Klein and Pieter Abbeel at UC Berkeley and Fahiem Bacchus at UofT.