TDDC17

Seminar II
Search I
Physical Symbol Systems
Uninformed Search

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

Physical Symbol System Hypothesis

Computer Science as Empirical Enquiry: Symbols and Search
Newell and Simon (1976)

Newell and Simon are trying to lay the foundational basis for the science of artificial intelligence.

What are the structural requirements for intelligence?

Can we define laws of qualitative structure for the systems being studied?

Symbol

What is a symbol, that intelligence may use it, and intelligence, that it may use a symbol? (McCulloch)

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

Physical Symbol Systems

The adjective "physical" denotes two important aspects:
- Such systems clearly obey the laws of physics -- they are realizable by engineered systems made of engineered components.
- The use of the term "symbol" is not restricted to human symbol systems.

A physical symbol system consists of:
- a set of entities called symbols which are physical patterns that can occur as components of another type of entity called an expression (or symbol structure).
- At any instant of time the system will contain a collection of symbol structures.
- The system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction, and destruction.

A physical-symbol system is a machine that produces through time an evolving collection of symbol structures and exists in a world of objects wider than just those symbol structures themselves.

Designation and Interpretation

There are two concepts central to these structures of expressions, symbols and objects:

Designation - An expression designates an object if, given the expression, the system can either effect the object itself or behave in ways depending on the object.

Interpretation - The system can interpret an expression if the expression designates a process and if, given the expression, the system can carry out the process.

Some additional requirements in the paper

Physical-Symbol System Hypothesis

The Physical-Symbol System Hypothesis - A physical-symbol system has the necessary and sufficient means for general intelligent action.

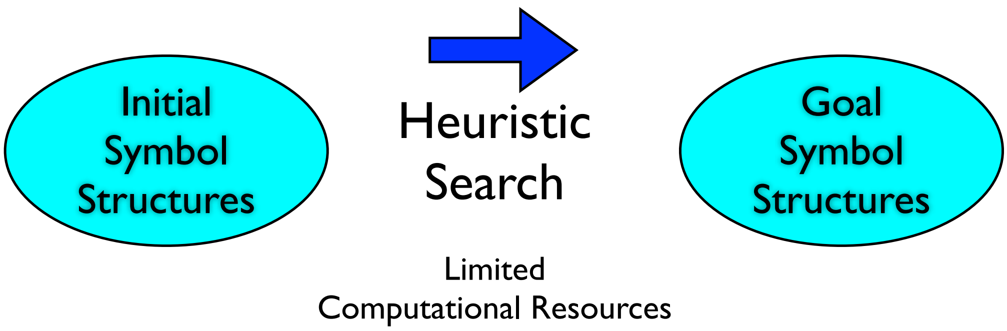
necessary - any system exhibiting intelligence will prove upon analysis to be a physical symbol system.

sufficient - any physical-symbol system of sufficient size can be organized further to exhibit general intelligence.

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

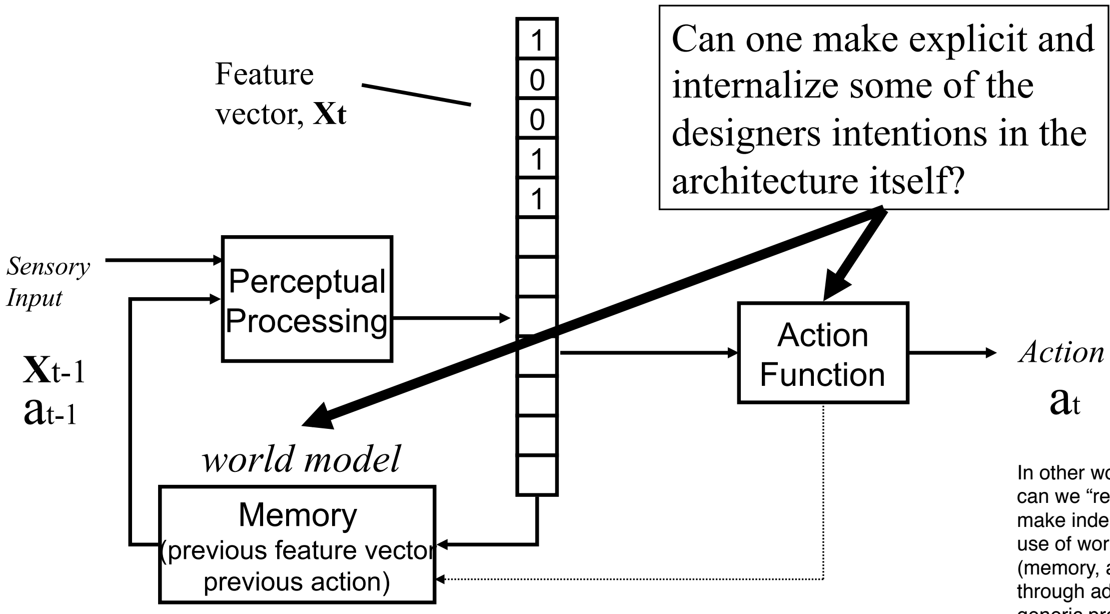
Heuristic Search Hypothesis

Heuristic Search Hypothesis - The solutions to problems are represented as symbol structures. A physical-symbol system exercises its intelligence in problem solving by search -- that is, by progressively modifying symbol structures until it produces a solution structure.



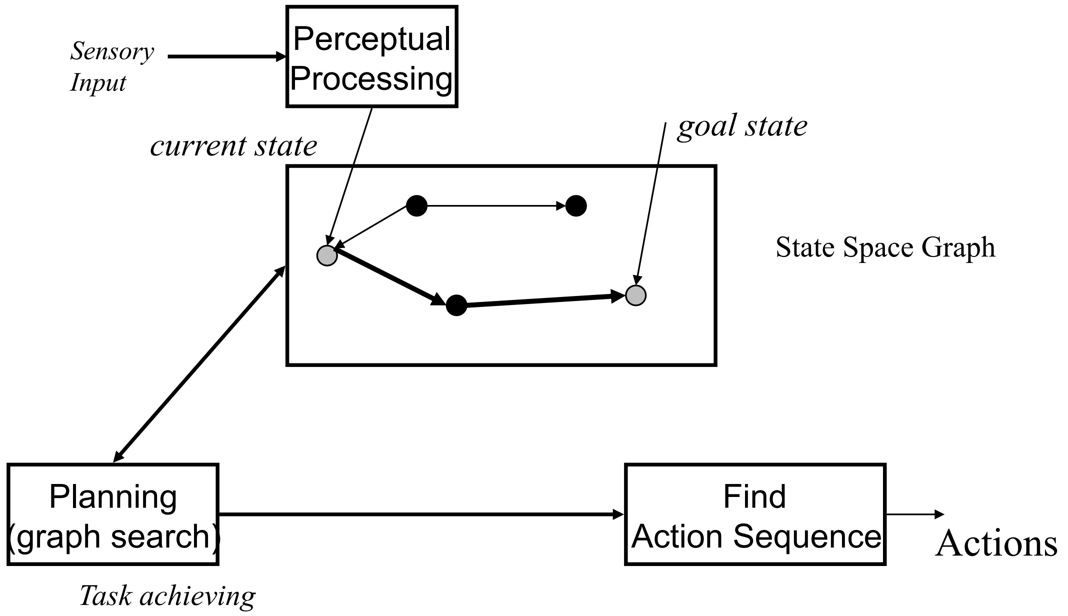
Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

Recall: State Machine Agent



In other words, can we “reify” and make independent use of world models (memory, actions) through additional generic processes which act on those world models?

Problem-Solving Agent (Version I)

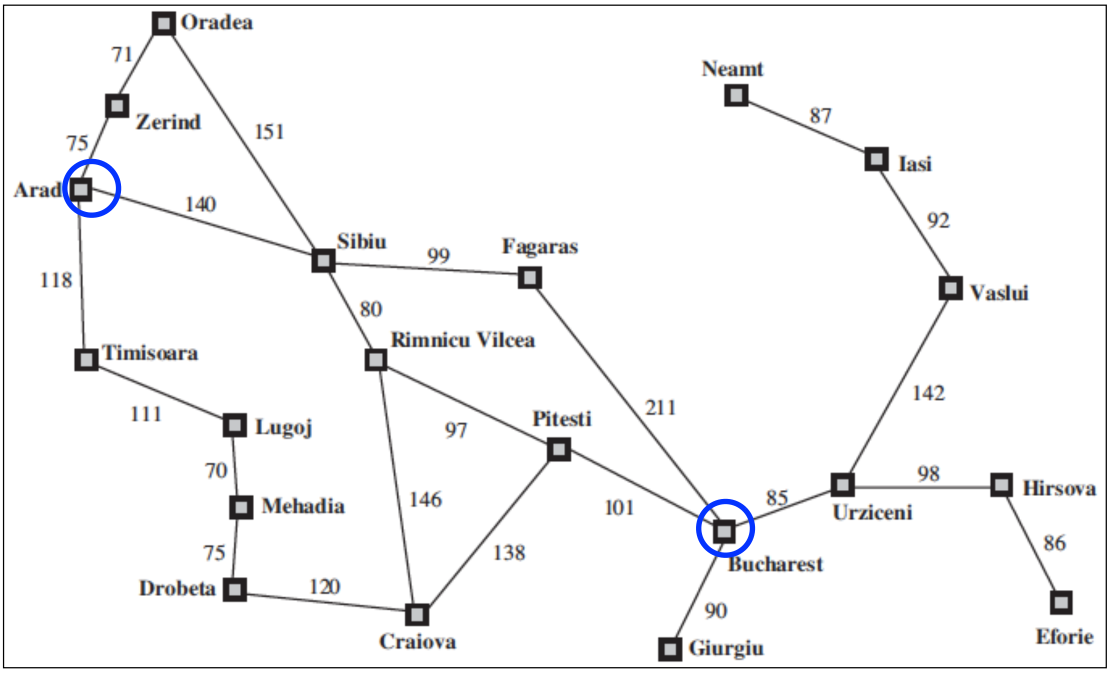


Simple Problem-Solving Agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              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)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```


Romania Route Finding Problem

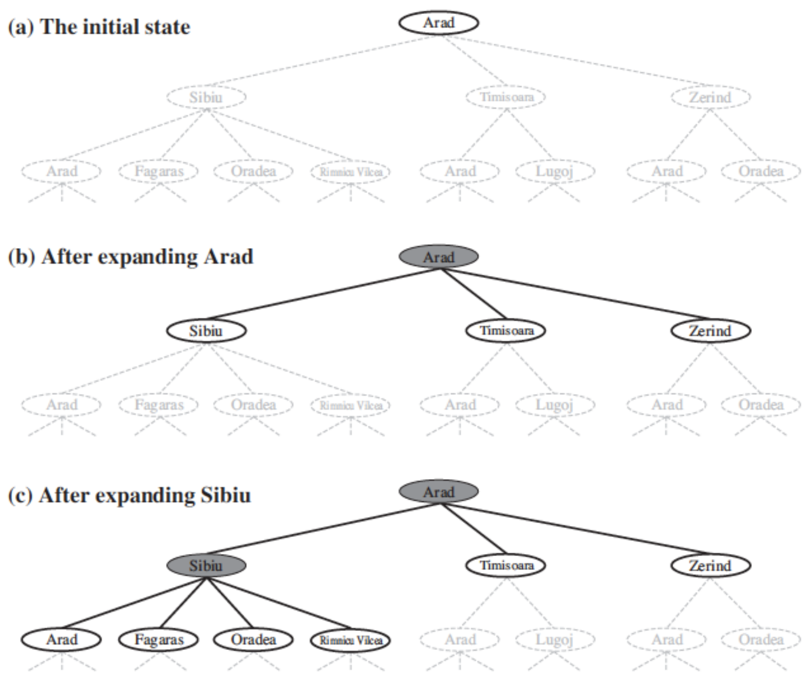


Problem Formulation

- Initial State - The state the agent starts in
  - In(Arad)
- **Actions**(*State*) - A description of what actions are available in each state.
  - **Actions**(In(Arad))= {Go(Sibiu), Go(Timisoara), Go(Zerind)}
- **Result**(*State, Action*) - A description of what each action does (Transition function)
  - **Result**(In(Arad), Go(Zerind))= In(Zerind)
- Goal Test - Tests whether a given state is a goal
  - Often a set of states: { In(Bucharest)}
- Path Cost - A function that assigns a cost to each path
  - # of actions, sum of distances, etc.
- Solution - A path from the start state to the goal state

State Space

Search Space: Route Finding



Expanded

Generated

Not Expanded

Problem Formulation for the Vacuum Cleaner World

- World states:

  2 positions, dirt or no dirt
  → 8 world states

- Actions:

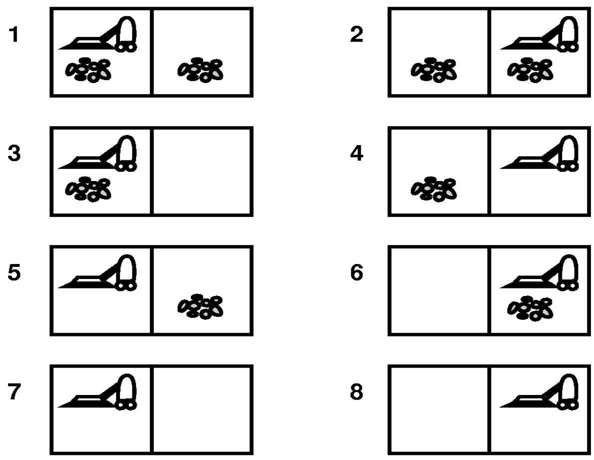
  Left (L), Right (R), or Suck (S)

- Transition model: next slide
- Initial State: Choose.
- Goal Test:

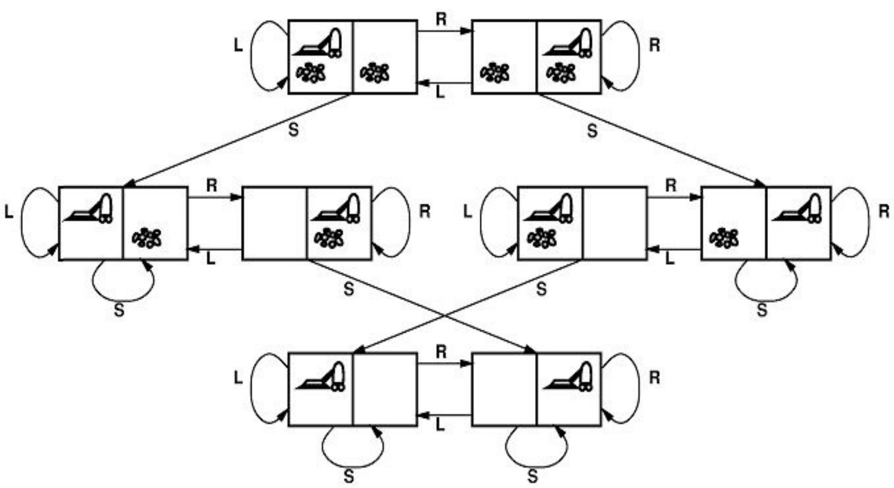
  no dirt in the rooms

- Path costs:

  one unit per action



Vacuum World: State Space



If the environment is completely accessible, the vacuum cleaner always knows where it is and where the dirt is. The solution then can be found by searching for a path from the initial state to the goal state.

States for the search: The world states 1-8.

Example: Missionaries and Cannibals

Informal problem description:

- Three missionaries and three cannibals are on one side of a river that they wish to cross.
- A boat is available that can hold at most two people.
- You must never leave a group of missionaries outnumbered by cannibals on the same bank.

➡ How should the state space be represented?
➡ What is the initial state?
➡ What is the goal state?
➡ What are the actions?

Formalization of the M&C Problem

States: triple (x,y,z) with $0 \le x,y,z \le 3$, where x,y, and z represent the number of missionaries, cannibals and boats currently on the original bank.

Initial State: (3,3,1)

Successor function: from each state, either bring one missionary, one cannibal, two missionaries, two cannibals, or one of each type to the other bank.

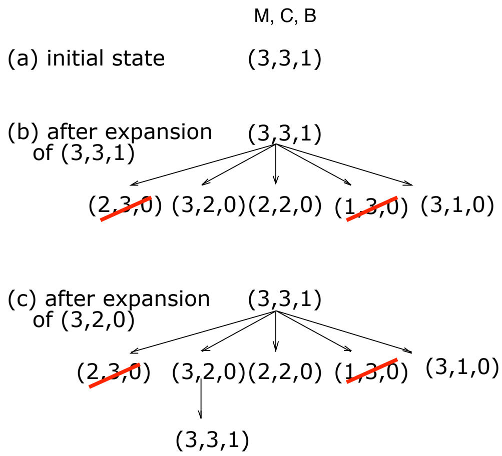
Note: not all states are attainable (e.g., (0,0,1)), and some are illegal.

Goal State: (0,0,0)

Path Costs: 1 unit per crossing

General Search

From the initial state, produce all successive states step by step → search tree.



Examples of Real-World Problems

- Route Planning, Shortest Path Problem
  - -Routing video streams in computer networks, airline travel planning, military operations planning...
- Travelling Salesperson Problem (TSP)
  - -A common prototype for NP-complete problems
- VLSI Layout
  - -Another NP-complete problem
- Robot Navigation (with high degrees of freedom)
  - -Difficulty increases quickly with the number of degrees of freedom. Further possible complications: errors of perception, unknown environments
- Assembly Sequencing
  - -Planning of the assembly of complex objects (by robots)

Implementing the Search Tree

Data structure for nodes in the search tree:

State: state in the state space
Parent-Node: Predecessor nodes
Action: The operator that generated the node
Depth: number of steps along the path from the initial state
Path Cost: Cost of the path from the initial state to the node

Operations on a queue:

Make-Queue(Elements): Creates a queue
Empty?(Queue): Empty test
First(Queue): Returns the first element of the queue (Non-destructive)
Remove-First(Queue): Returns the first element
Insert(Element, Queue): Inserts new elements into the queue
(various possibilities)
Insert-All(Elements, Queue): Inserts a set of elements into the queue

States and Nodes

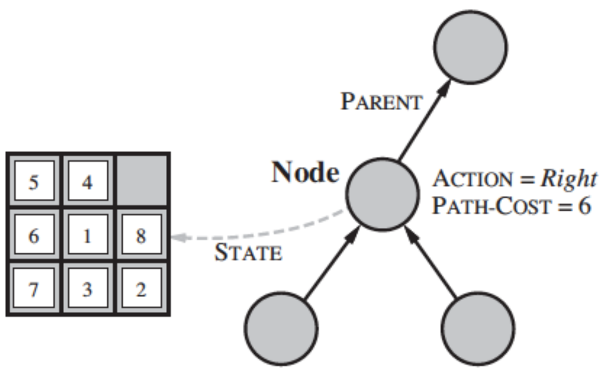


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Finite set of states but sometimes infinite nodes in a search tree

Tree/Graph Search Algorithm

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Avoids redundant paths and loops

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Problem:
- initial state
- actions/result. fn
- goal test
- path cost

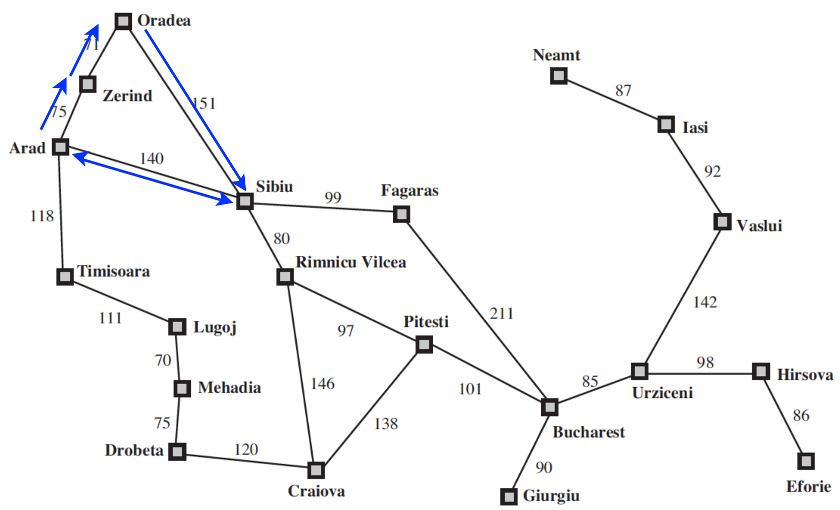
Romanian Roadmap

Arad-Sibiu-Arad

Loopy Path- makes the complete search space infinite even though there are only 20 states

Arad-Sibiu
Arad- Zerind-Oradea-Sibiu

Redundant Path- more than one way to get from one state to another



Search Strategies

- A **Strategy** is defined by picking the *order of node expansion*
- Strategies are evaluated along the following dimensions:
  – **Completeness** – does it always find a solution if one exists?
  – **Time Complexity** – number of nodes generated/expanded
  – **Space Complexity** – maximum number of nodes in memory
  – **Optimality** – does it always find a least cost solution
- Time & space complexity are measured in terms of
  – b – maximum branching factor of the search tree
  – d – depth of the least cost solution
  – m – maximum length of any path in the state space (possibly infinite)

Some Search Classes

- Uninformed Search (Blind Search)
  – No additional information about states besides that in the problem definition.
  – Can only generate successors and compare against goal state.
  – Some examples
    - Breadth first search, Depth first search, iterative deepening DFS
- Informed Search (Heuristic Search)
  – Strategies have additional information whether non-goal states are more promising than others.
  – Some examples
    - Greedy Best-First search, A* search,

Breadth-First Graph Search

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```


Breadth-First Search

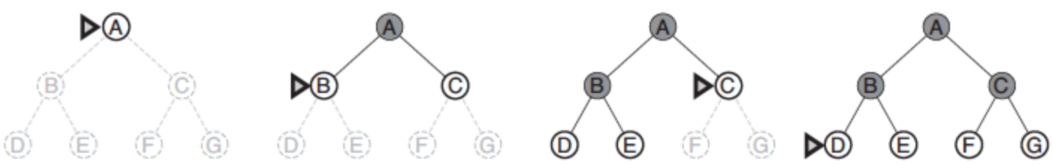


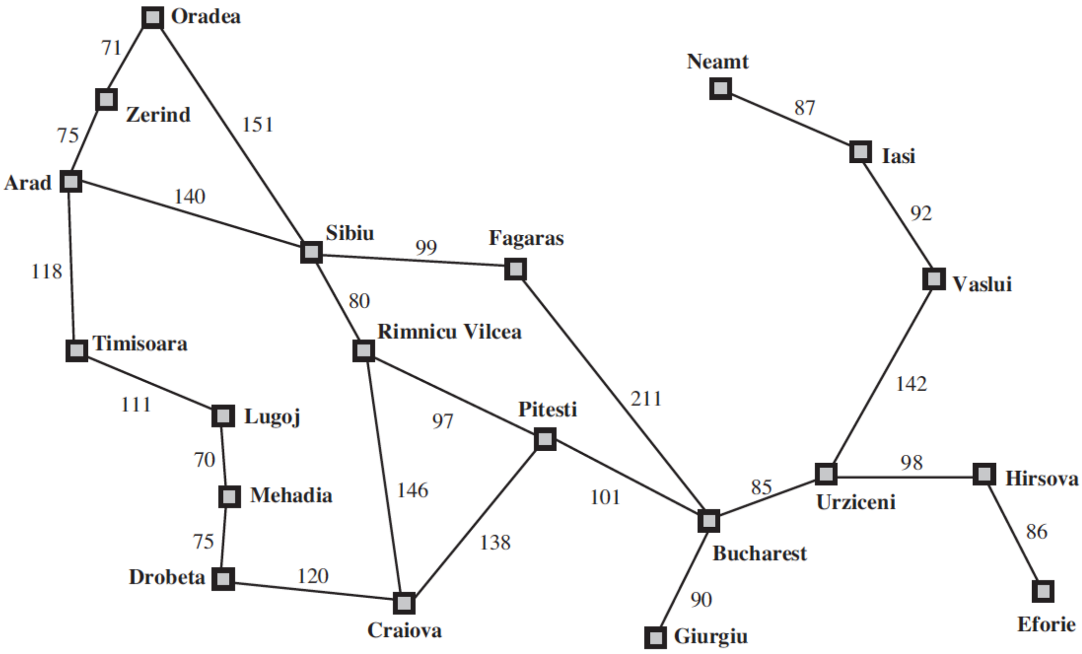
Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

	Frontier (FIFO queue)	Explored
INIT	A	
POP		A'
Add Children	B, C	
POP	C	A',B'
Add Children	C,D,E	
POP	D,E	A',B',C'
Add Children	D,E,F,G	
POP	E,F,G	A',B',C',D'
POP	F,G	A',B',C',D',E'
POP	G	A',B',C',D',E',F'
POP		A',B',C',D',E',F',G'

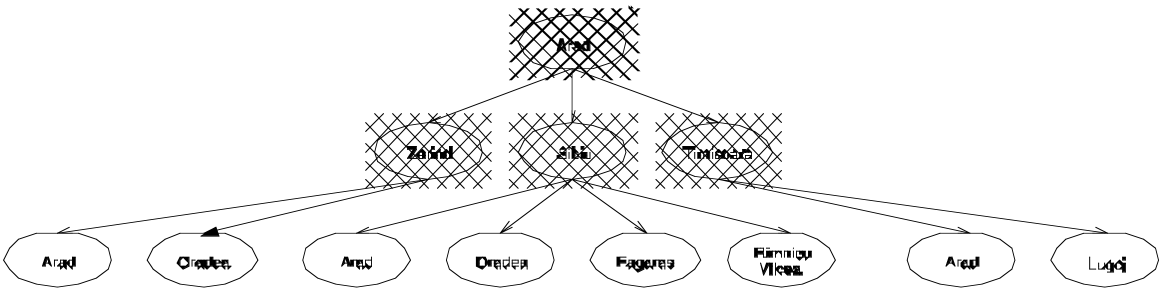
X' - node state

Shallowest nodes in the front of the frontier

Romanian Roadmap



Romania BFS



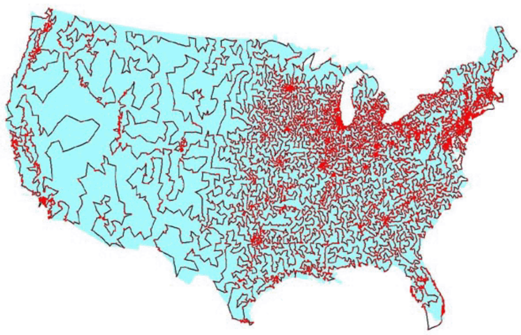
Note the potential loop to Arad!

• This makes the complete search tree infinite even though the state space has only 20 states

Depth 2, BFS animated

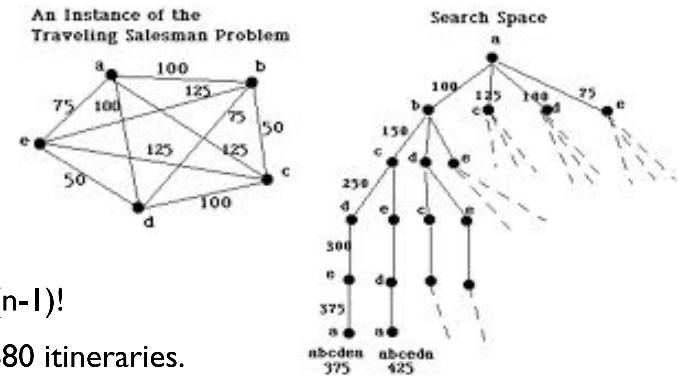
Computational Complexity Theory

Traveling Salesman Problem



- The Traveling Salesman Problem is one of the most intensively studied problems in computational mathematics.
- A traveling salesman has *n* number of cities to visit. He wants to know the <u>shortest route</u> which will allow him to visit <u>all</u> cities <u>one</u> time and <u>return</u> to his starting point.
- Solving this problem becomes MUCH harder as the number of cities increases; the figure in the middle shows the solution for the 13,509 cities and towns in the US that have more than 500 residents.

Traveling Salesman Problem



- Suppose there are n cities to visit.
- The number of possible itineraries is (n-1)!
  - For n=10 cities, there are 9!=362,880 itineraries.
- What if n=40?
  - There are now 39! itineraries to check which is greater then 10^{45}
  - Examining 10^{15} tours per second, the required time would be several billion lifetimes of the universe
  - In fact, no supercomputer, existing or projected can run this fast.
- (n-1)! grows faster than 2^n. So the time it takes to solve the problem grows exponentially with the size of the input.

Sequential Search: Telephone Book

- Suppose a telephone book has N=`1000000` entries.
- Given the name Y, search the telephone book sequentially for Y's telephone number
  - Entries $\langle X_1, T_1\rangle, \langle X_2, T_2\rangle, \ldots \langle X_{1000000}, T_{1000000}\rangle$
  - At each iteration Y is compared with X_i
  - Assume time increases relative to the number of comparisons, so we are counting comparison instructions. (there may be other instructions...)
- In the worst case, 1000000 comparisons may have to be made.
- Call the algorithm A. We say it has a worst case running time which is on the order of *N*.
- A runs in time $O(N)$ in the worst case, where *N* is the number of entries in the telephone book.
  - In other words, the time complexity of A is dependent on the size of the input.
  - A has worst case behavior which is linear in the size of the input to A.

Big-O Notation

- We do not care whether the algorithm takes time N, $3N$, $100N$, or even a fraction of N: $N/6$
  - The only thing that matters is that the running time of the algorithm grows linearly with N.
  - In other words, there is some constant k such that the algorithm runs in time that is no more than $k \times N$ in the worst case
- Let T(n) be a function on n (the size of the input to an algorithm) then T(n) is on the order of $f(n)$ if:

T(n) characterizes the running time of the algorithm, i.e, lines of code, # of additions, etc. as a function of input n.

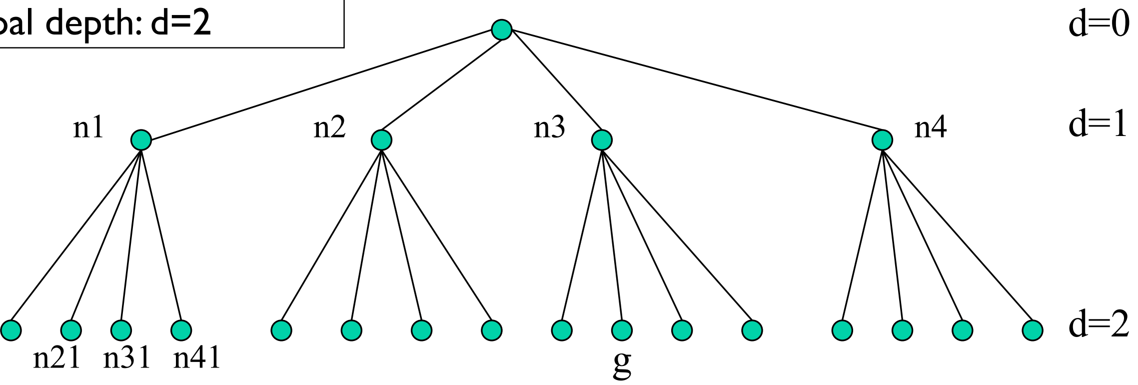
T(n) usually characterizes worst case running time.

Asymptotic Analysis

$$\boxed{T(n) \text{ is } O(f(n)) \text{ if } T(n) < k \times f(n) \text{ for some } k, \text{ for all } n > n_0}$$

Analyzing Breadth-First Search

Branching Factor: b=4
goal depth: d=2



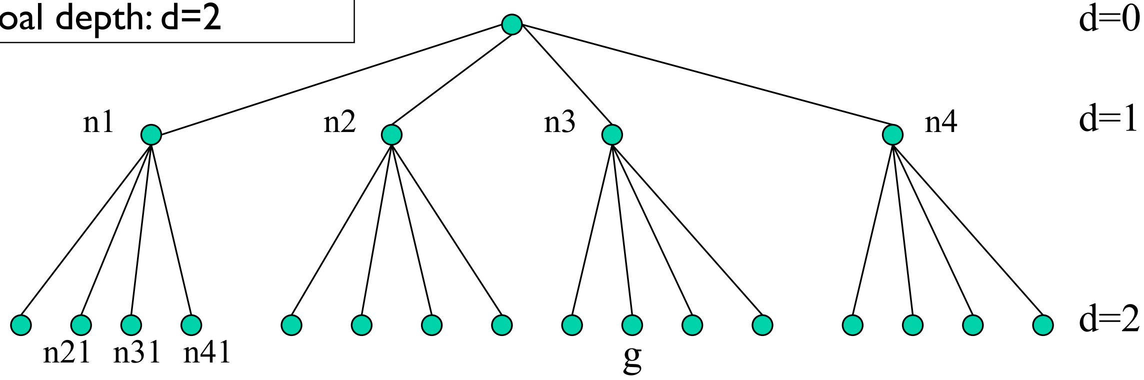
Time Complexity: O(b^d)

- When checking for a goal node at level d, at least $1 + b + b^2 + \ldots + b^{d-1}$ nodes must be generated.
- Total nodes generated and checked may be as much as $1 + b + b^2 + \ldots + b^{d-1} + b^d$

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

Analyzing Breadth-First Search

Branching Factor: b=4
goal depth: d=2



Space Complexity: $O(b^d)$

- For any graph search, every expanded node is stored in the explored set.
- There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier set.
- The space complexity is dominated by the nodes in the frontier.

Analyzing Breadth-First Search

Is it complete?

If the shallowest goal node is at some finite depth d,
BFS will eventually find it after searching all shallower nodes
(Provided the branching factor is finite)

Is it optimal?

The shallowest goal node is not necessarily optimal, but it is optimal if the path cost is a non-decreasing function of the depth of the node. Example: each action has the same cost.

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden

Exponential Complexity Bounds are Highly Problematic

Time/memory requirements for breadth-first search with branching factor b=10
1 million nodes/second; 1000bytes a node

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

1000^4

Uniform-Cost Search

We know that breadth-first search is optimal when all step-costs are equal.

This can be generalized to any step-cost function

Instead of expanding the shallowest node (FIFO queue), uniform-cost search expands the node n with the lowest path cost g(n) from the root. A priority queue on path costs of nodes is used instead of a FIFO queue.

Uniform-Cost Search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
     if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           frontier ← INSERT(child, frontier)
        else if child.STATE is in frontier with higher PATH-COST then
           replace that frontier node with child
```

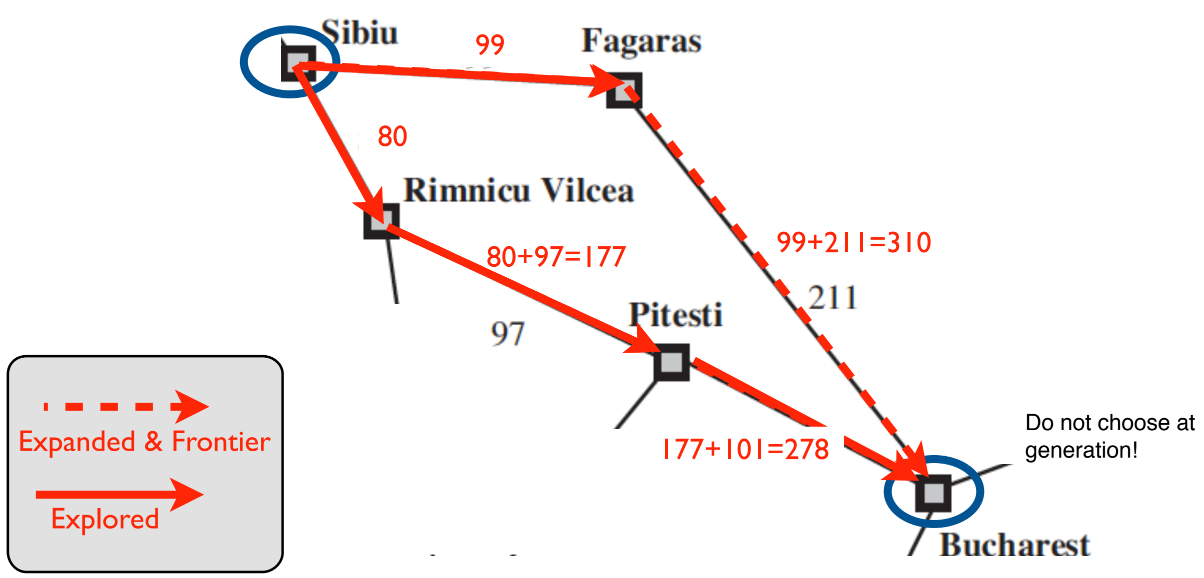
1st goal node generated may be on a sub-optimal path

goal test during selection for expansion rather than generation as in BFS

Replace, if better path to a node on the frontier is found

Generic Graph Search with modifications

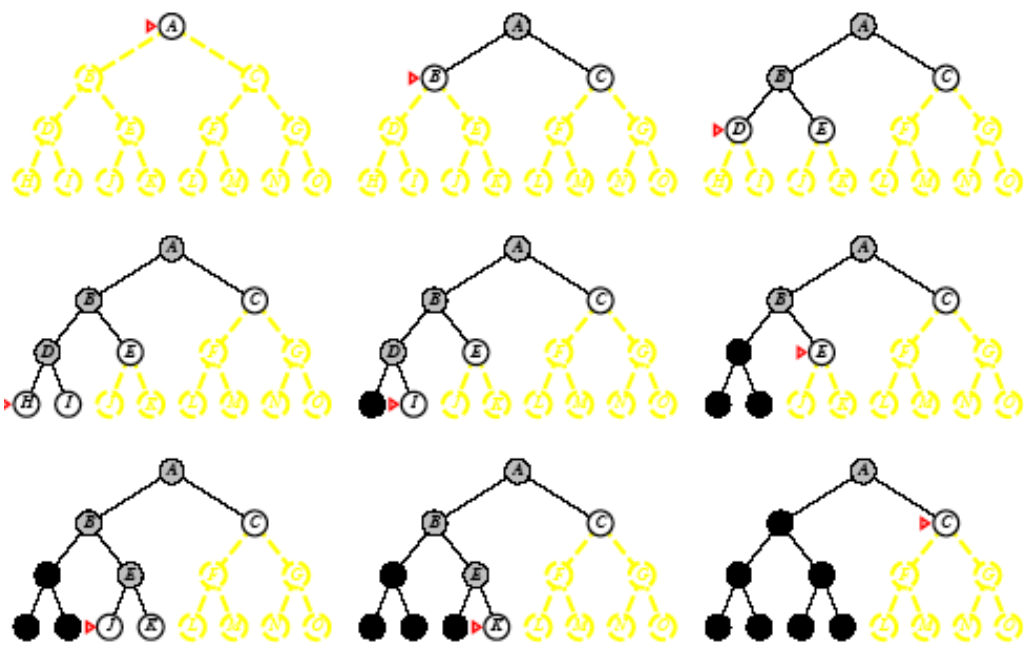
Uniform-Cost Search



Depth-First Graph Search

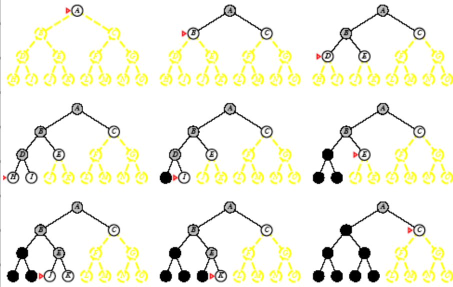
```
function Depth-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a LIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```


Depth-First Search on a Binary Tree



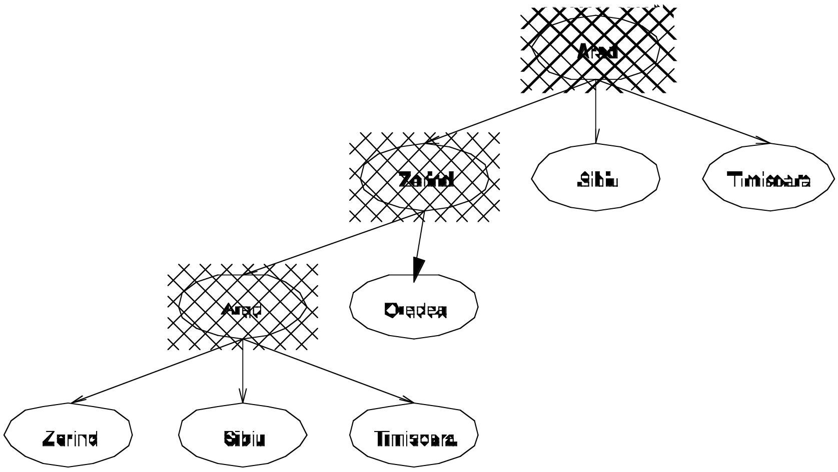
Depth-First Search

	Frontier(LIFO)	Explored
INIT	A	
POP		A’
Add Children	B, C	
POP	C	A’,B’
Add Children	D,E,C	
POP	E,C	A’,B’,D’
Add Children	H,I,E,C	
POP	I,E,C	A’,B’,D’,H’
POP	E,C	A’,B’,D’,H’I’
POP	C	A’,B’,D’,H’,I’,E’
Add Children	J,K,C	
POP	K,C	A’,B’,D’,H’,I’,E’,J’
POP	C	A’,B’,D’,H’,I’,E’,J’,K’
Add Children	F,G,	
POP	G	A’,B’,D’,H’,I’,E’,J’,K’,F
Add Children	L,M,G	
POP	M,G	A’,B’,D’,H’,I’,E’,J’,K’,F’,L’
POP	G	A’,B’,D’,H’,I’,E’,J’,K’,F’,L’,M’
POP		A’,B’,D’,H’,I’,E’,J’,K’,F’,L’,M’,G’
Add Children	N,O,	



X’ - node state

Romania DFS



Analyzing Depth-First Search

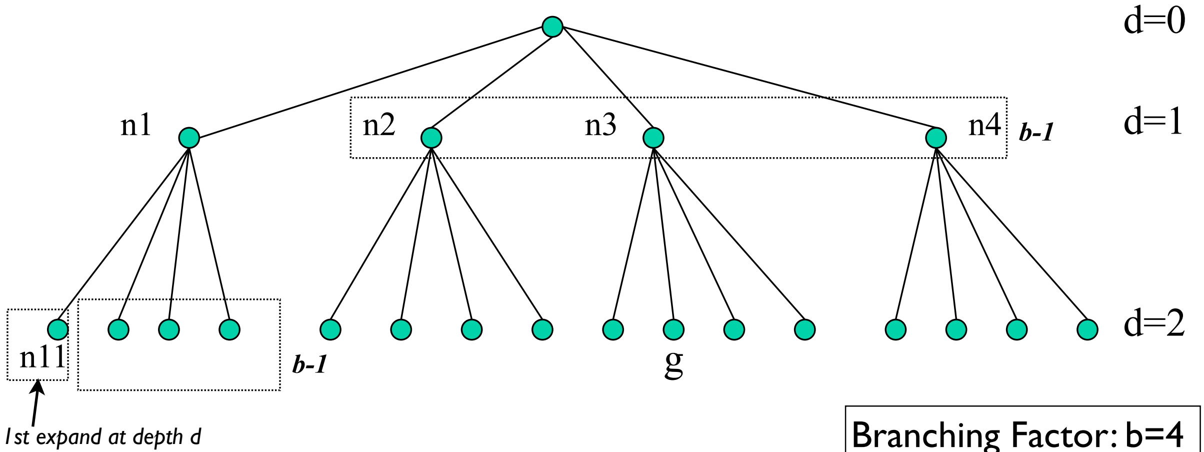
Time Complexity

Depth-First Graph search is bounded by the size of the state-space (which may be infinite).

Depth-First Tree search may generate all of the $O(b^m)$ nodes in the search tree where m is the maximum length of any path in the state space.

m can be much greater than the size of the state space and can be much larger than d, the depth of the shallowest goal node, and is infinite if the tree is unbounded.

Analyzing Depth-First Tree Search



Branching Factor: b=4
Goal Depth: d=2

Space Complexity: O(d(b-1))

•For any tree search, when checking for a goal node at level d, at most d(b-1) nodes must be stored in the frontier.

Analyzing Depth-First Search

Is it complete?

Yes, for the Graph Search version in finite state spaces. It avoids repeated states and redundant paths and will eventually expand every node.

No, for the Tree Search version. It may loop infinitely on one branch.

Is it optimal?

Both versions are non-optimal

Complexity Analysis and O Notaton

Time and space usage of a particular algorithm is often measured in terms of a function of the size of the input to the algorithm.

$T(d) = 1+b+b^2+b^{d-1}+b^d$ *Breadth-First Space Usage* $O(b^d)$

The amount of space used is <u>exponential</u> in the depth of the shallowest goal node.

$T(d) = d(b-1) + 1$ *Depth-First Space Usage* $O(d*b)$

The amount of space used is <u>linear</u> in the depth of the shallowest goal node. (tree search)

$T(n)$ is $O(f(n))$ if $T(n) < k * f(n)$ for some k, for all $n > n_0$

Asymptotic Analysis

Recursive Implementation of Depth-Limited Search

- Deals with failure of depth-first search in infinite state spaces
- Introduce a pre-determined cut-off depth limit *l*

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   else if limit = 0 then return cutoff
   else
       cutoff_occurred? ← false
       for each action in problem.ACTIONS(node.STATE) do
           child ← CHILD-NODE(problem, node, action)
           result ← RECURSIVE-DLS(child, problem, limit − 1)
           if result = cutoff then cutoff_occurred? ← true
           else if result ≠ failure then return result
       if cutoff_occurred? then return cutoff else return failure
```

If l < d then we may not find the goal and DLS is incomplete
If l > d then DLS is not optimal
DLS with l= infinity is in fact depth-first search

Time Complexity: $O(b^l)$
Space Complexity: $O(bl)$

Iterative-Deepening Depth-First Search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

- Combines the best of depth-first search and breadth-first search

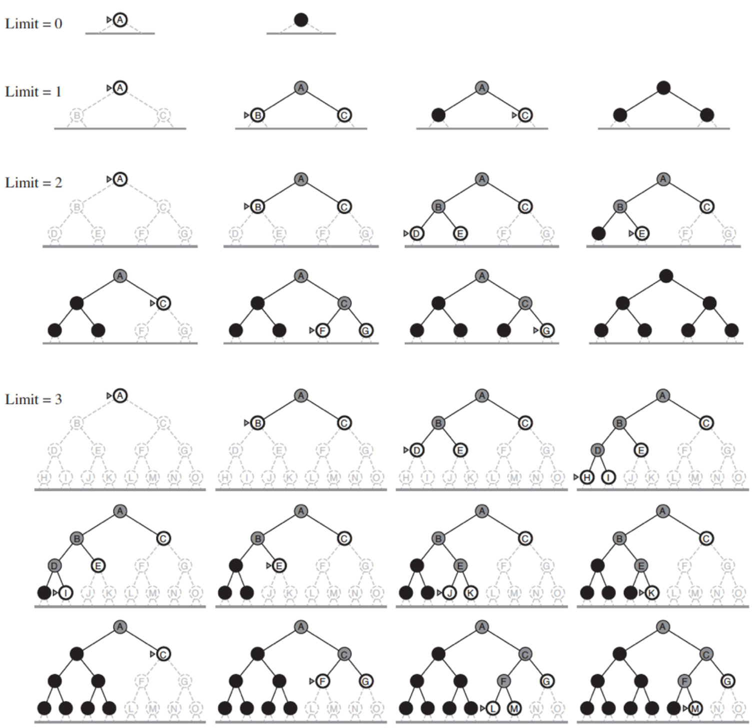
Gradually increases the depth-limit of depth-first search by increments (0,1,2...).
Each increment basically does a breadth-first search to that limit

Complete when the branching factor is finite

Optimal when the path cost is a non-decreasing function of the depth of the node

Space Complexity: O(bd)
Time Complexity: $O(b^d)$

Iterative-Deepening (4 iterations)



Summary of Analyses

For Tree-Search Versions

Criterion	Breadth-First	Depth-First	Depth-Limited	Iterative-Deepening
Complete?	Yes[a]	No	No	Yes[a]
Time	$O(b^d)$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal	Yes[c]	No	No	Yes[c]

Graph-Search Versions:
- DFS Complete for finite spaces
- Time/space complexities bounded by size of the state space

a - complete if b is finite
c - optimal if step costs are all identical

Artificial Intelligence & Integrated Computer Systems Division
Department of Computer and Information Science
Linköping University, Sweden