

Artificial Intelligence

Prof. Björn Ommer
HCI & IWR

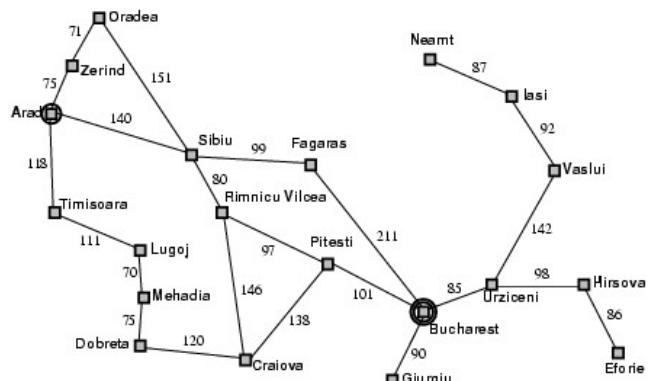


Example: Romania

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest
- Formulate goal:
 - be in Bucharest
- Formulate problem:
 - **states:** various cities
 - **actions:** drive between cities
- Find solution:
 - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

B. Ommer | ommer@uni-heidelberg.de

Example: Romania



B. Ommer | ommer@uni-heidelberg.de

Outline – Solving problems by search

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
- Now: Uninformed Search
- Next Chapter: Informed Search

B. Ommer | ommer@uni-heidelberg.de

Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  static: seq, an action sequence, initially empty
  state, some description of the current world state
  goal, a goal, initially null
  problem, a problem formulation
  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then do
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
    action  $\leftarrow$  FIRST(seq)
    seq  $\leftarrow$  REST(seq)
  return action
```

Note: this is online problem solving; solution executed "eyes closed."
Online problem solving involves acting without complete knowledge.

B. Ommer | ommer@uni-heidelberg.de

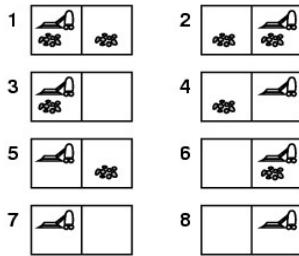
Problem types

- Deterministic, fully observable → **single-state problem**
 - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → **sensorless problem (conformant problem, multi-state)**
 - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → **contingency problem**
 - percepts provide **new** information about current state
 - often **interleave** search, execution
- Unknown state space → **exploration problem ("online")**

B. Ommer | ommer@uni-heidelberg.de

Example: vacuum world

- Single-state, start in #5.
Solution?



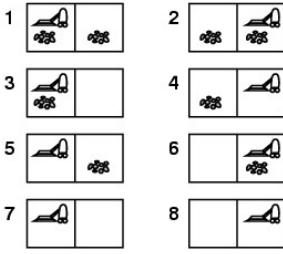
B. Ommer | ommer@uni-heidelberg.de

Example: vacuum world

- Single-state, start in #5.

Solution? [Right, Suck]

- Sensorless, start in $\{1,2,3,4,5,6,7,8\}$ e.g.,
Right goes to $\{2,4,6,8\}$
Solution?



B. Ommer | ommer@uni-heidelberg.de

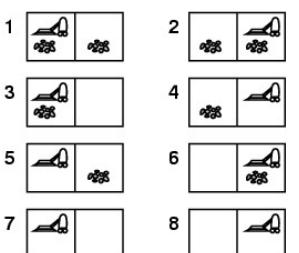
Example: vacuum world

- Sensorless, start in

 $\{1,2,3,4,5,6,7,8\}$ e.g.,
Right goes to $\{2,4,6,8\}$ **Solution?**

[Right, Suck, Left, Suck]

- Contingency
 - Nondeterministic: *Suck* may dirty a clean carpet
 - Partially observable: can observe dirt at current location only
 - Percept: [*L*, *Clean*], i.e., start in #5 or #7
Solution?



B. Ommer | ommer@uni-heidelberg.de

Example: vacuum world

- Sensorless, start in

 $\{1,2,3,4,5,6,7,8\}$ e.g.,*Right* goes to $\{2,4,6,8\}$ **Solution?**

[Right, Suck, Left, Suck]

- Contingency
 - Nondeterministic: *Suck* may dirty a clean carpet
 - Partially observable: location, dirt at current location.
 - Percept: [*L*, *Clean*], i.e., start in #5 or #7
Solution? [Right, if dirt then Suck]

B. Ommer | ommer@uni-heidelberg.de

Single-state problem formulation

A **problem** is defined by four items:

1. **initial state** e.g., "at Arad"
 2. **actions or successor function** $S(x) = \text{set of action-state pairs}$
e.g., $S(\text{Arad}) = \{\langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots\}$
 3. **goal test**, can be
 - **explicit**, e.g., $x = \text{"at Bucharest"}$
 - **implicit**, e.g., $\text{NoDirt}(x)$
 4. **path cost** (additive)
 - e.g., sum of distances, number of actions executed, etc.
 - $c(x,a,y)$ is the **step cost**, assumed to be ≥ 0
- A **solution** is a sequence of actions leading from the initial state to a goal state

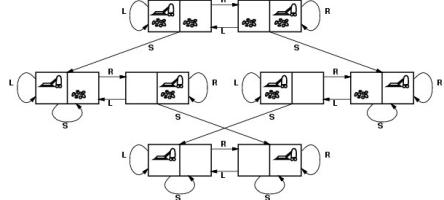
B. Ommer | ommer@uni-heidelberg.de

Selecting a state space

- Real world is absurdly complex
→ state space must be **abstracted** for problem solving
- **(Abstract) state** = set of real states
- **(Abstract) action** = complex combination of real actions
 - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, **any** real state "in Arad" must get to **some** real state "in Zerind"
- **(Abstract) solution** = set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

B. Ommer | ommer@uni-heidelberg.de

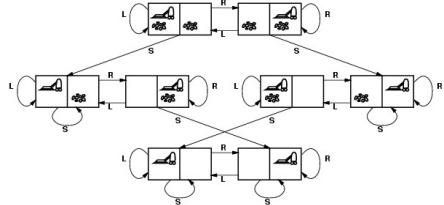
Vacuum world state space graph



- states?
- actions?
- goal test?
- path cost?

B. Ommer | ommer@uni-heidelberg.de

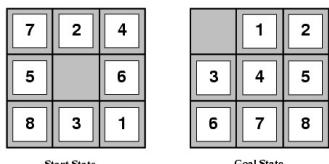
Vacuum world state space graph



- states? integer dirt and robot location (ignore dirt amount)
- actions? Left, Right, Suck, NoOP
- goal test? no dirt at all locations
- path cost? 1 per action

B. Ommer | ommer@uni-heidelberg.de

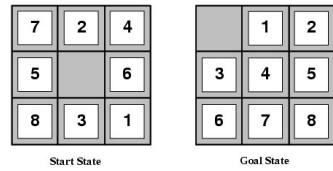
Example: The 8-puzzle



- states?
- actions?
- goal test?
- path cost?

B. Ommer | ommer@uni-heidelberg.de

Example: The 8-puzzle

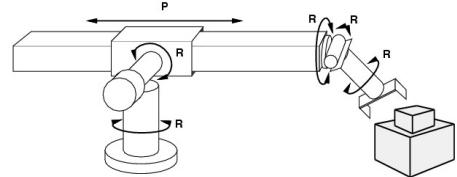


- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

[Note: optimal solution of n -Puzzle family is NP-hard]

B. Ommer | ommer@uni-heidelberg.de

Example: robotic assembly



- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

B. Ommer | ommer@uni-heidelberg.de

Tree search algorithms

- Basic idea:

- offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. **expanding states**)

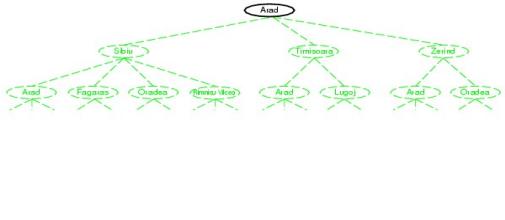
```

function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    Which? if the node contains a goal state then return the corresponding solution
            else expand the node and add the resulting nodes to the search tree
  end

```

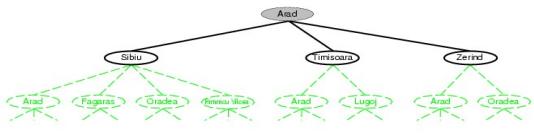
B. Ommer | ommer@uni-heidelberg.de

Tree search example



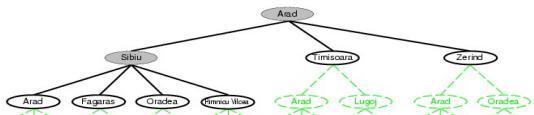
B. Ommer | ommer@uni-heidelberg.de

Tree search example



B. Ommer | ommer@uni-heidelberg.de

Tree search example

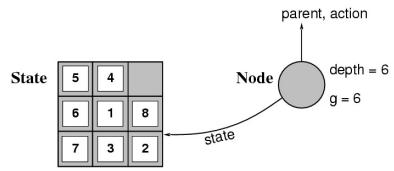


B. Ommer | ommer@uni-heidelberg.de

Implementation: states vs. nodes

- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree

- includes **state**, **parent node**, **action**, **path cost $g(x)$** , **depth**



- The **Expand** function creates new nodes, filling in the various fields and using the **SuccessorFn** of the problem to create the corresponding states.

B. Ommer | ommer@uni-heidelberg.de

General Tree & Graph Search

```

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
  
```

B. Ommer | ommer@uni-heidelberg.de

Implementation: general tree search

```

function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe ← INSERTALL(EXPAND(node, problem), fringe)
  
```

```

function EXPAND(node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE(node)) do
    s ← a new NODE
    PARENT-NODE[s] ← node; ACTION[s] ← action; STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
  
```

B. Ommer | ommer@uni-heidelberg.de

Search strategies

- A search 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
 - space complexity:** maximum number of nodes in memory
 - optimality:** does it always find a least-cost solution?
- Time and space complexity are measured in terms of
 - b:** maximum branching factor of the search tree
 - d:** depth of the least-cost solution
 - m:** maximum depth of the state space (may be ∞)

B. Ommer | ommer@uni-heidelberg.de

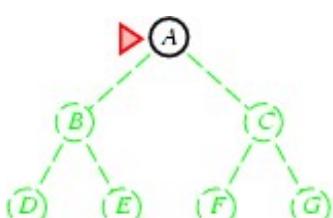
Uninformed search strategies

- Uninformed** search strategies use only the information available in the problem definition
 - Breadth-first search
 - Uniform-cost search
 - Depth-first search
 - Depth-limited search
 - Iterative deepening search

B. Ommer | ommer@uni-heidelberg.de

Breadth-first search

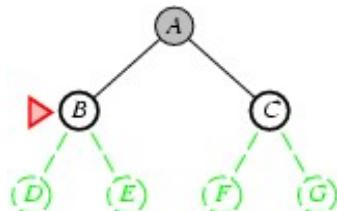
- Expand shallowest unexpanded node
 - Implementation:
 - fringe* is a FIFO queue, i.e., new successors go at end
- ```
function BREADTH-FIRST-SEARCH(problem) returns a solution or failure
 return GENERAL-SEARCH(problem,ENQUEUE-AT-END)
```



B. Ommer | ommer@uni-heidelberg.de

## Breadth-first search

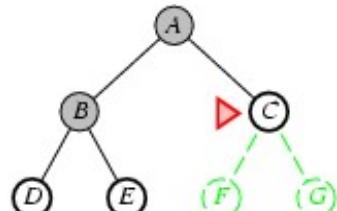
- Expand shallowest unexpanded node
- Implementation:
  - fringe* is a FIFO queue, i.e., new successors go at end



B. Ommer | ommer@uni-heidelberg.de

## Breadth-first search

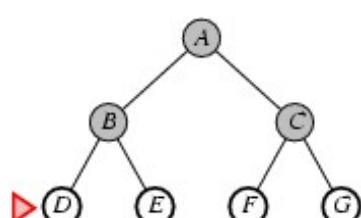
- Expand shallowest unexpanded node
- Implementation:
  - fringe* is a FIFO queue, i.e., new successors go at end



B. Ommer | ommer@uni-heidelberg.de

## Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - fringe* is a FIFO queue, i.e., new successors go at end



B. Ommer | ommer@uni-heidelberg.de

## Properties of breadth-first search

- **Complete?** Yes (if  $b$  is finite)
- **Time?**  $1+b+b^2+b^3+\dots+b^d + b(b^d-1) = O(b^{d+1})$ , i.e. exponential in  $d$
- **Space?**  $O(b^{d+1})$  (keeps every node in memory)
- **Optimal?** Yes (if cost = 1 per step); not optimal in general
- **Space** is the bigger problem (more than time)

B. Ommer | ommer@uni-heidelberg.de

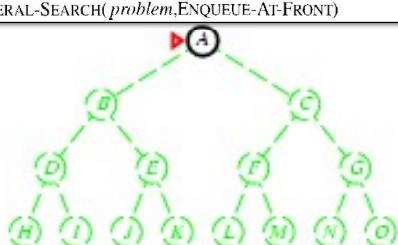
## Uniform-cost search

- Expand least-cost unexpanded node
- **Implementation:**
  - fringe = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- **Complete?** Yes, if step cost  $\geq \epsilon$
- **Time?** # of nodes with  $g \leq \text{cost of optimal solution}$ ,  $O(b^{\lceil C^*/\epsilon \rceil})$  where  $C^*$  is the cost of the optimal solution
- **Space?** # of nodes with  $g \leq \text{cost of optimal solution}$ ,  $O(b^{\lceil C^*/\epsilon \rceil})$
- **Optimal?** Yes – nodes expanded in increasing order of  $g(n)$

B. Ommer | ommer@uni-heidelberg.de

## Depth-first search

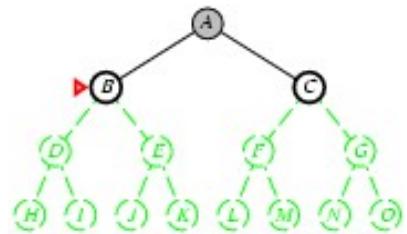
- Expand deepest unexpanded node
  - **Implementation:**
    - fringe = LIFO queue, i.e., put successors at front
- ```
function DEPTH-FIRST-SEARCH(problem) returns a solution, or failure
  GENERAL-SEARCH(problem,ENQUEUE-AT-FRONT)
```



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

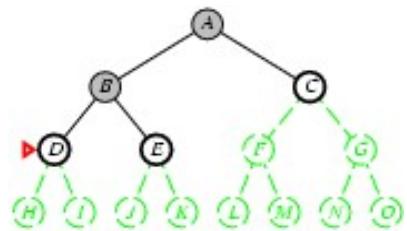
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

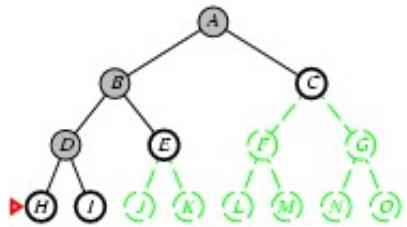
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

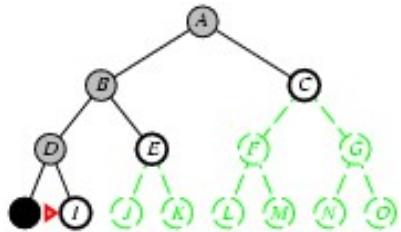
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

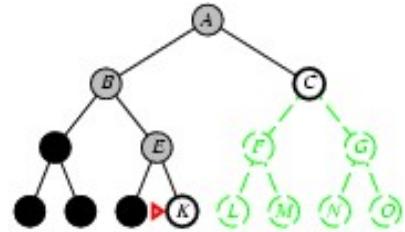
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

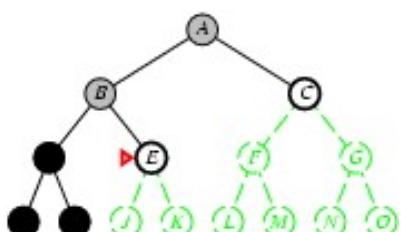
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

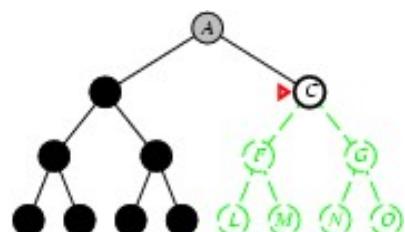
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

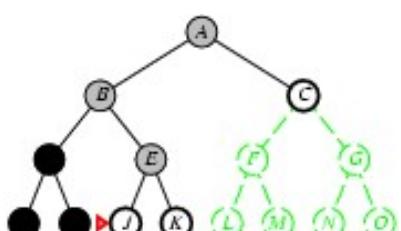
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

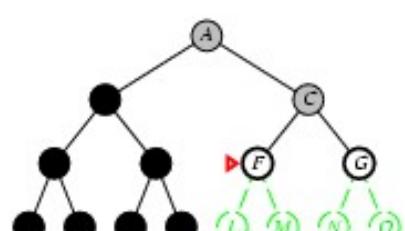
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

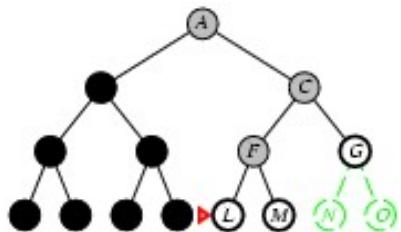
- Expand deepest unexpanded node
- **Implementation:**
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

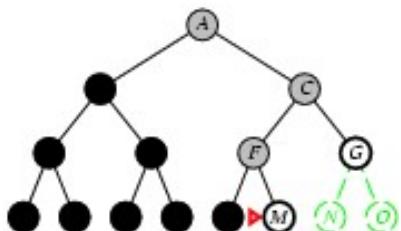
- Expand deepest unexpanded node
- Implementation:
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Depth-first search

- Expand deepest unexpanded node
- Implementation:
 - fringe = LIFO queue, i.e., put successors at front



B. Ommer | ommer@uni-heidelberg.de

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
 - Modify to avoid repeated states along path
→ complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
 - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$, i.e., linear space!
- Optimal? No
- **DFS should be avoided for search trees with large or infinite maximum depths**

B. Ommer | ommer@uni-heidelberg.de

Depth-limited search

- = depth-first search with depth limit l ,
i.e., nodes at depth l have no successors

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST(problem, STATE[node]) then return node
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result ← RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

B. Ommer | ommer@uni-heidelberg.de

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
  inputs: problem, a problem
  for depth ← 0 to  $\infty$  do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
  end
```

B. Ommer | ommer@uni-heidelberg.de

Iterative deepening search / =0

Limit = 0

B. Ommer | ommer@uni-heidelberg.de

Iterative deepening search /=1

Limit = 1



Number of nodes generated = 1 + 1 + 2 = 4

B. Ommer | ommer@uni-heidelberg.de

Iterative deepening search

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

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

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

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

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

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

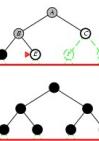
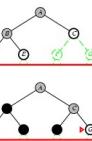
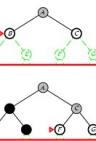
$$N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$$

$$\text{Overhead} = (123,456 - 111,111)/111,111 = 11\%$$

B. Ommer | ommer@uni-heidelberg.de

Iterative deepening search /=2

Limit = 2

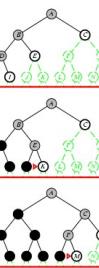
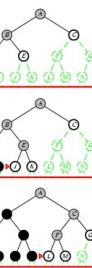
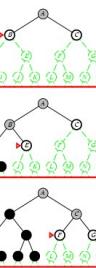


Number of nodes generated = 2 + 2 + 3 = 7

B. Ommer | ommer@uni-heidelberg.de

Iterative deepening search /=3

Limit = 3



Number of nodes generated = 3 + 3 + 4 = 10

B. Ommer | ommer@uni-heidelberg.de

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d b^1 + (d-1)b^2 + \dots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

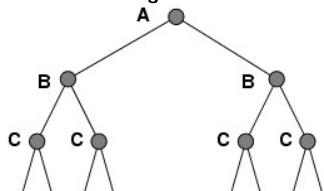
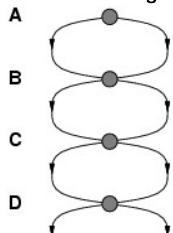
- IDS does better because other nodes at depth d are not expanded
- BFS can be modified to apply goal test when a node is generated

Criterion	Breadth-Fist	Uniform-Cost	Depth-Fist	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(bm)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	No	$O(bl)$
Optimal?	Yes	Yes	No	No	Yes

B. Ommer | ommer@uni-heidelberg.de

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Strategies:
 - Do not return to the state you just came from.
 - Do not create paths with cycles in them
 - Do not generate any state that was ever generated before



B. Ommer | ommer@uni-heidelberg.de

Graph search

```

function GRAPH-SEARCH(problem,fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]),fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem),fringe)
  end

```

B. Ommer | ommer@uni-heidelberg.de

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

B. Ommer | ommer@uni-heidelberg.de