Uninformed Search

Ian Sanders

Second Semester, 2024

Why do we cover these approaches?

- History – most of them were originally developed to solve AI problems – planning, game playing, etc.
- They have become part of mainstream Computer Science – in applications that are not necessarily seen as AI problems
- GPS Navigation, Town Planning, Intelligent Tutoring Systems, Logistics
- Obviously still important in AI – search based artificially intelligent agents.
  Agents that seek to achieve a goal (e.g. build a house, win a board game, etc.)
  Search to find a solution to reach a goal
  What is important is how we formulate the problem.

Definitions

- Agent – a decision-making entity e.g. robot, software code, etc.
- State – a representation of the agent's world/environment at a given time
- State space – the space of all possible states
- Successor function – what actions the agent can take at a given time, the cost of doing so and what happens next
- Goal test – a function that, given the current state, returns true if a goal condition has been met
- Reflexive agent – makes decisions based on current state only
- Planning agent – makes decisions taking into account future consequences of its actions

Block layout

This involves a *problem-solving agent*.

The agent is being asked to solve the problem of laying out seven identical blocks into a "C" shape.

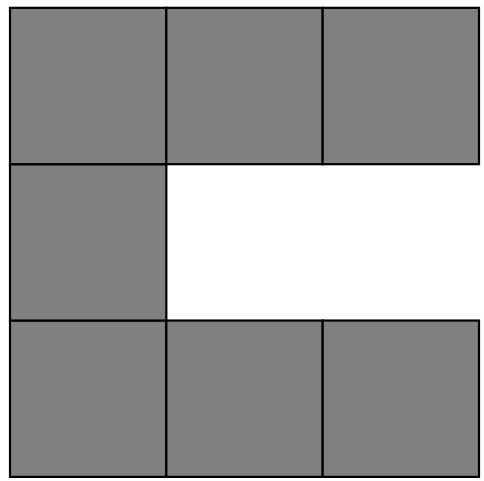


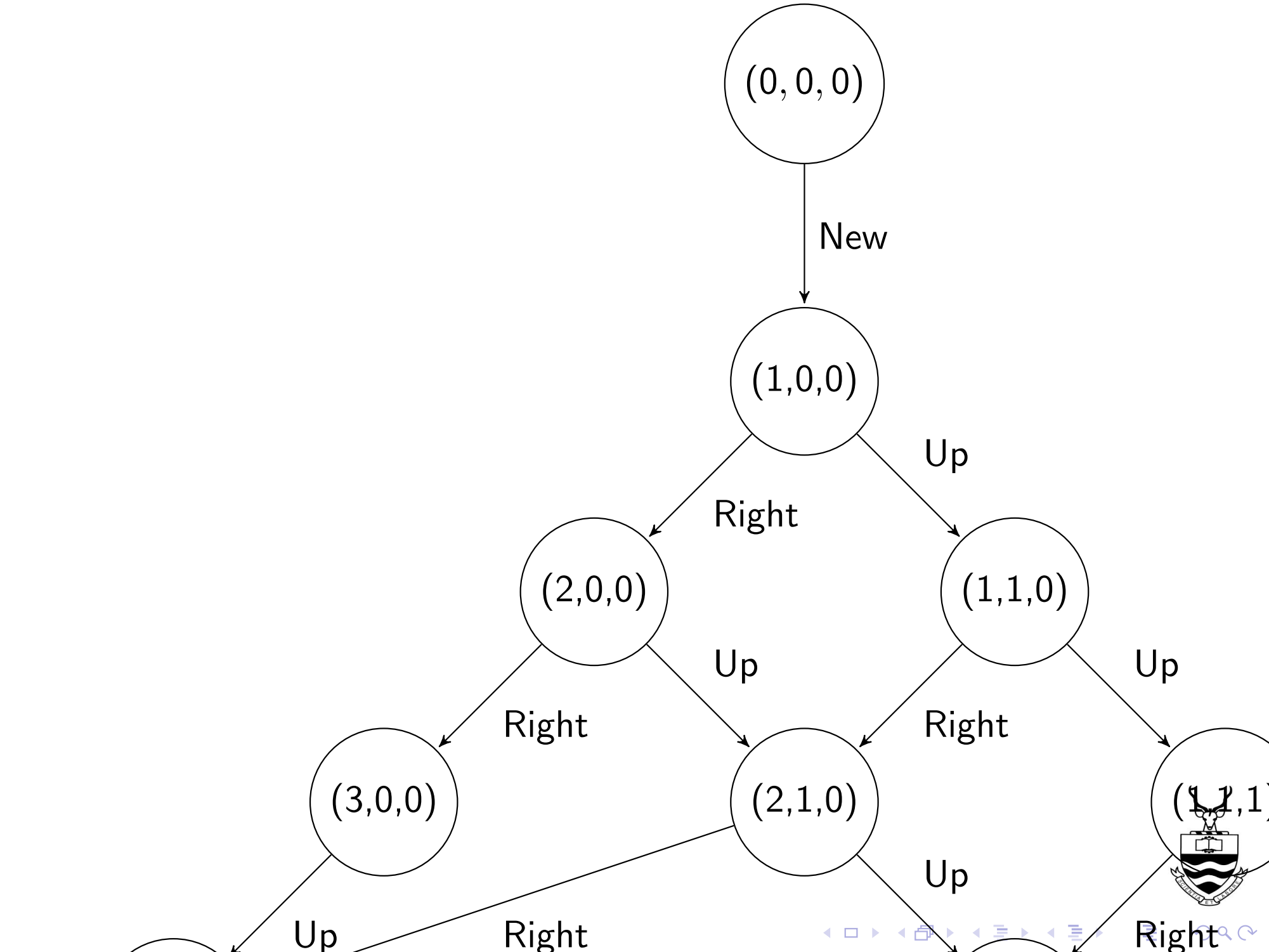
Figure: The desired result of block layout as seen from above

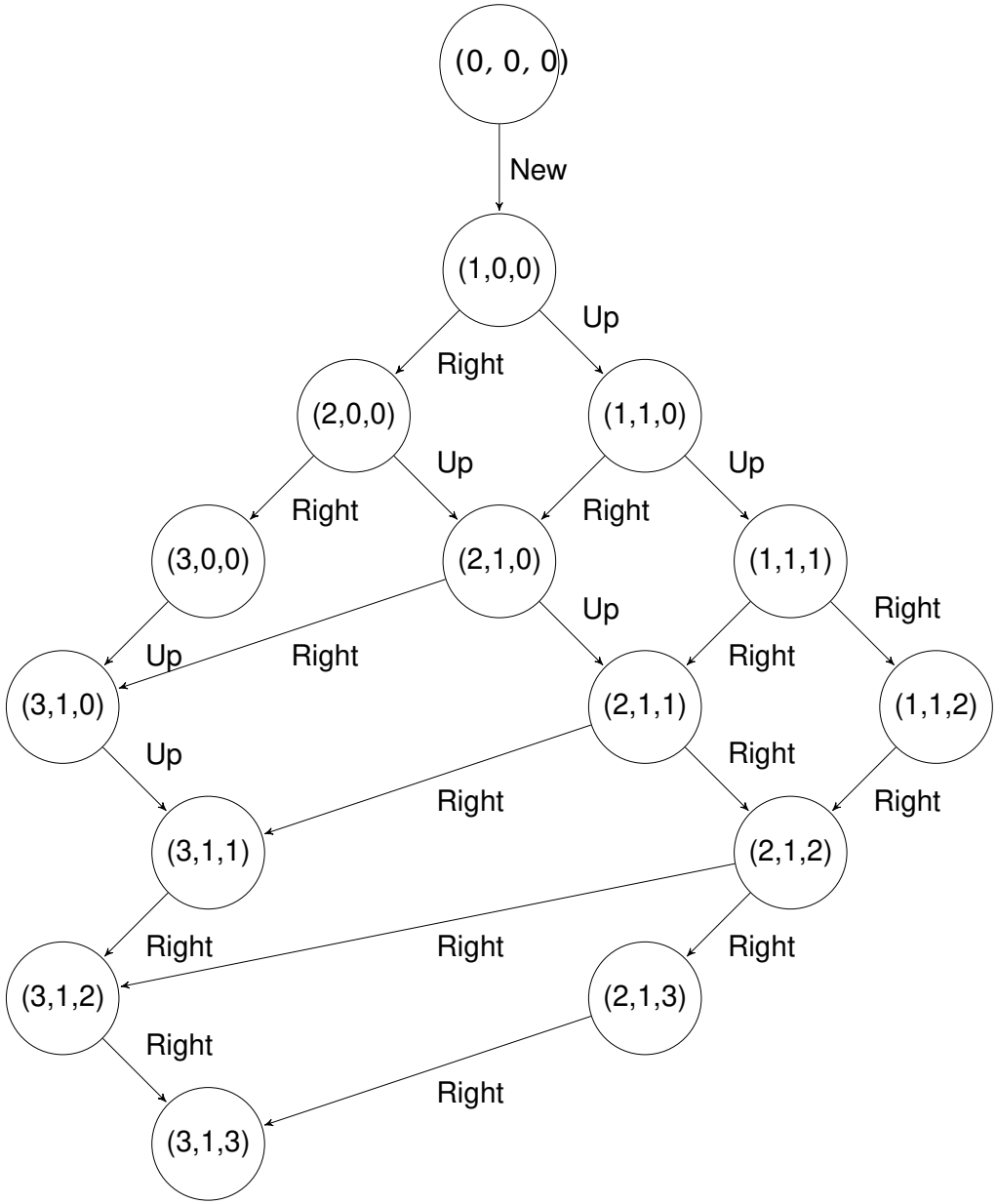
First we need to define a state representation for the problem.
A simple non-graphical representation is: $\{B, M, T\}$ –
B is a set indicating which blocks in the bottom row are filled
$\{0, 0, 0\}$ would indicate there are no blocks in the row,
M is a set indicating which blocks in the middle row are filled and
T is a set indicating which blocks in the top row are filled.
Ideally we want the most concise representation.
We would need to be sure that the level of abstraction is sufficient.
That is, that the state representation only encodes the necessary information.
Show that our representation does not encode more information or less information than is required,

What is the initial state?
$\{\{0,0,0\},\{0,0,0\},\{0,0,0\}\}$
What is the goal state?
$\{\{1,1,1\},\{1,0,0\},\{1,1,1\}\}$
Now we define the appropriate actions for this problem.
Require that the agent should start by placing a block into the bottom left postion. Call this *New*
Then using the successor function
$Result(a, S) \mapsto S'$,
show how applying each action a to some state S results in some state S'.
The applicable actions are *Up* and *Right*
$Result(Up, \{\{1,0,0\},\{0,0,0\},\{0,0,0\}\}) \mapsto$
$\{\{1,0,0\},\{1,0,0\},\{0,0,0\}\}$
$Result(Right, \{\{1,0,0\},\{0,0,0\},\{0,0,0\}\}) \mapsto$
$\{\{1,1,0\},\{0,0,0\},\{0,0,0\}$
Would need to check S' is a valid state.

The state space

Note: For convenience in the diagram below I use the number of blocks in each row instead of the set notation.
The start state is thus $(0, 0, 0)$ and the goal state is thus $(3, 1, 3)$.
This reduces the size of the nodes and thus the size of the diagram.


(0, 0, 0)
New
(1,0,0)
Right
Up
(2,0,0)
(1,1,0)
Right
Up
Right
Up
(3,0,0)
(2,1,0)
(1,1,1)
Up
Up
Right
Right



There are 14 different states – reachable in different ways.

There are 14 different states – reachable in different ways.
Note that we could have presented the state space as a *tree* instead of a graph.
There would then have been multiple *goal* states!
So how do we search this graph – from the start state to a goal state.

Search tree

- A tree that shows the future outcomes of actions
- Root is the start state
- Children are successor states
- Edges from root to a node in the tree
  - edges are the plan
  - sum of edge costs is the cost of the plan
- For most problems we can't build the full tree $O(b^d)$ where d is the maximum path length

Planning with a search tree

- Expand tree nodes (potential plans)
- Maintain a frontier of partial plans under consideration
- Try to come up with a solution while expanding as few nodes as possible

General tree 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

General tree search continued

Important ideas:

- Frontier
- Expansion
- Exploration strategy – which frontier node to choose.

Genral graph Search algorithm is similar *but* have to remember which nodes have already been visited.
Therefore have an *explored* list as well as a frontier list.
See figure 3.7 in Russell and Norvig, 3rd Edition.

Search strategies

- Tree search algorithms are about picking the order of node expansion
- Things to consider
  - Completeness – will it always find a solution if one exists
  - Optimality – will it always find a least cost solution
  - Time complexity – number of nodes generated
  - Space complexity – max number of nodes in memory
- Complexity depends on maximum branching function, depth of optimal solution, maximum depth of state space.

Uninformed search

We use only the information in the problem definition
If we had additional domain knowledge then we could use informed/heuristic search – we will consider this later
General uniformed strategies

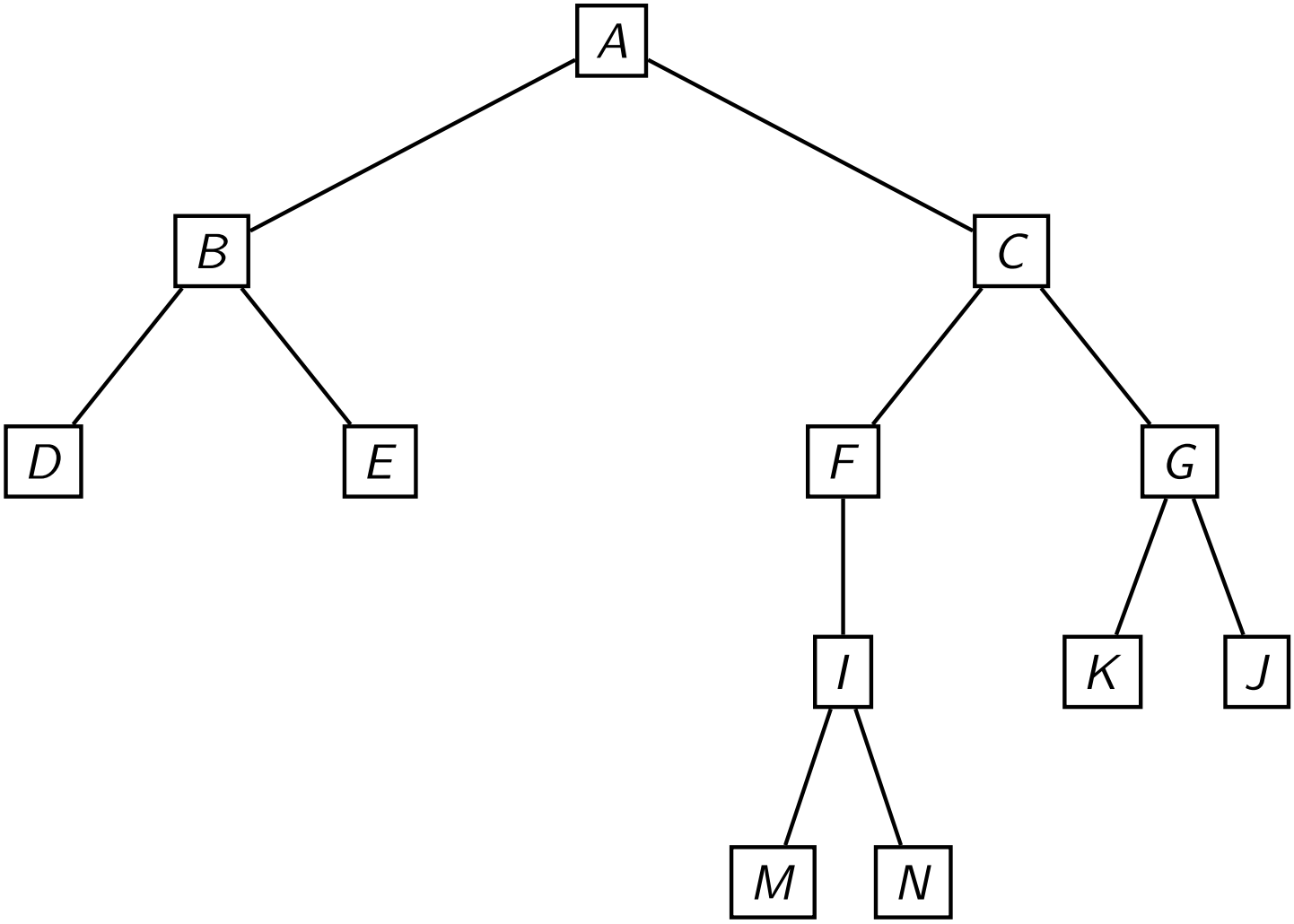
- Breadth first search
- Uniform cost search
- Depth first search

Breadth-first search

Expand the shallowest unexpanded node
Frontier as a FIFO queue (new successors get added at the end)

Breadth First 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)
```



A is start state. G is goal state.

Node	Explored	Frontier	Comment
A			From initial state of tree – root node
A			Goal test A
A		A	Add A to frontier
			Into loop
A		A	Pop A
A	A	A	add A to explored,
A	A	B	Goal test B, Add B to frontier
A	A	B C	Goal test C, Add C to frontier
B	A B	C	Pop B, add B to explored,
B	A B	C D	Goal test D, add D to frontier
B	A B	C D E	Goal test E, add E to frontier
C	A B C	D E	Pop C, add C to explored,
C	A B C	D E F	Goal test F, add F to frontier
C	A B C	D E F	Goal test G, End with success.

Note that Goal test happens when considering the children.

BFS properties

Let d be depth of shallowest solution

- Complete?
  Yes (if b is finite). Shallowest solution returned
- Time?
  $1 + b + b^2 + \ldots + b^d = O(b^d)$
- Space?
  Keeps all frontier nodes in memory: $O(b^d)$
- Optimal:
  Only if costs are constant

BFS Issues

Both time and space are $O(b^d)$
Can become very slow and require vast amounts of memory as d increases.

Uniform-cost search

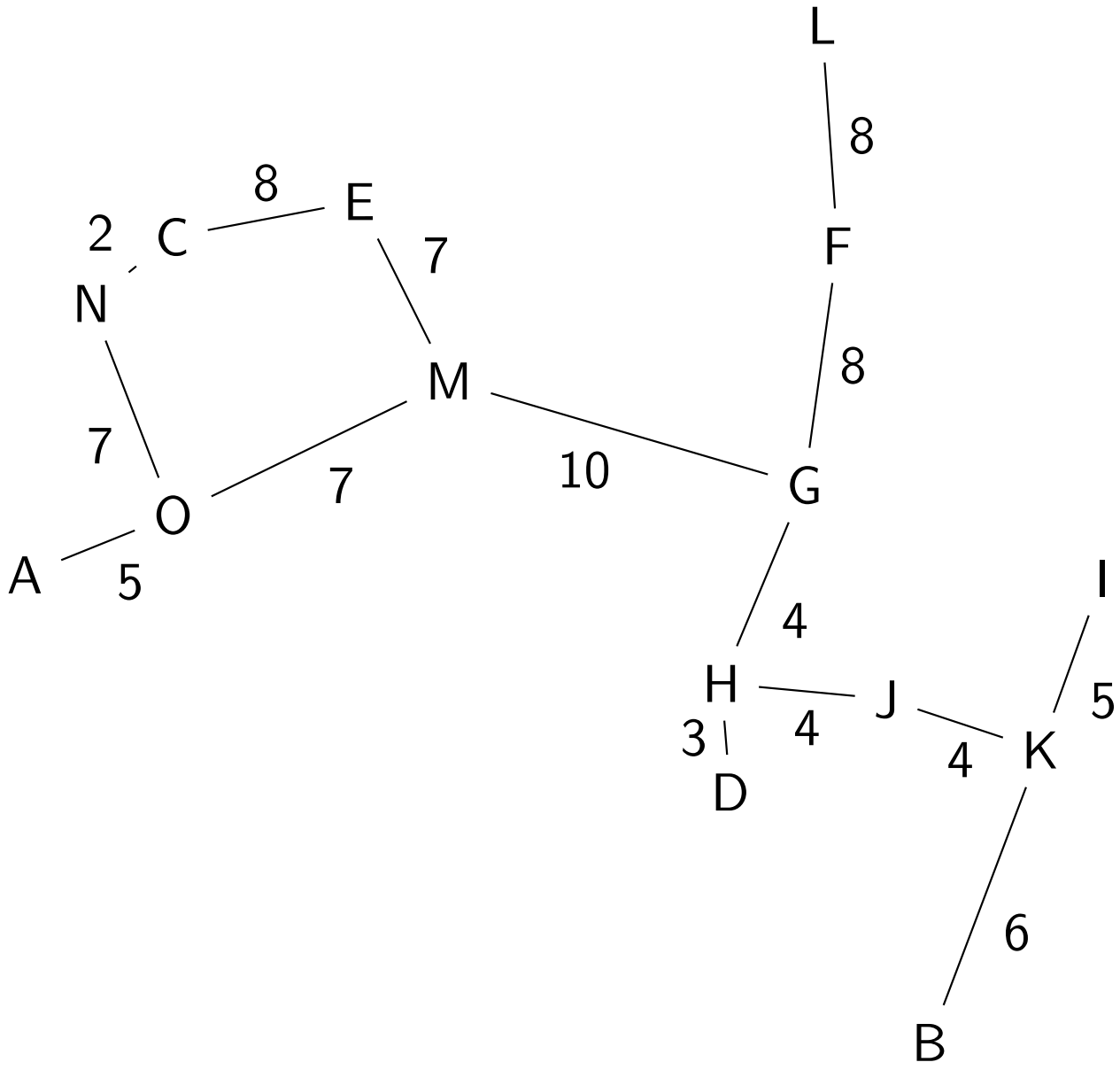
BFS finds plan with shortest length
But what if cost of plan is not optimal? i.e. a longer plan may have smaller cost overall
UCS almost same as BFS, but use priority queue instead of queue
Each node in queue ordered by cost to node
BFS = UCS when costs are constant everywhere

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 eleme
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*

Consider the graph below, perform a UCS on the graph; assume that the start node is N, and the goal node is F.



Step	Node expanded	Frontier
1		N($\hat{g} = 0$)
2	N	C-N(2), O-N(7)
3	C-N	O-N(7), E-C-N(10)
4	O-N	E-C-N(10), A-O-N(12), M-O-N(14)
5	E-C-N	A-O-N(12), M-O-N(14), [M-E-C-N(17)]
6	A-O-N	M-O-N(14)
7	M-O-N	G-M-O-N(24)
8	G-M-O-N	H-G-M-O-N(28), F-G-M-O-N(32)
9	H-G-M-O-N	D-H-G-M-O-N(31), F-G-M-O-N(32), J-H-G-M-O-N(32)
10	D-H-G-M-O-N	F-G-M-O-N(32), J-H-G-M-O-N(32)
11	F-G-M-O-N	J-H-G-M-O-N(32)

Exercise: Make all the weights on the edges in the graph above equal to 1. Then apply BFS and UCS to the graph and compare the results.

UCS properties

Expands all nodes with cost less than cheapest solution
If solution cost is C^* and each edge costs at least ϵ then the effective depth is approximately C^*/ϵ
Complete? Yes, if cost $\geq \epsilon$ (positive costs) and best solution has finite cost
Time? $O(b^{ceiling(C^*/\epsilon)})$
Space? Keeps all frontier nodes in memory: $O(b^{ceiling(C^*/\epsilon)})$
Optimal: Yes! Nodes expanded in increasing order of total cost

Depth first search

Depth first search is essentially Graph Search as below.

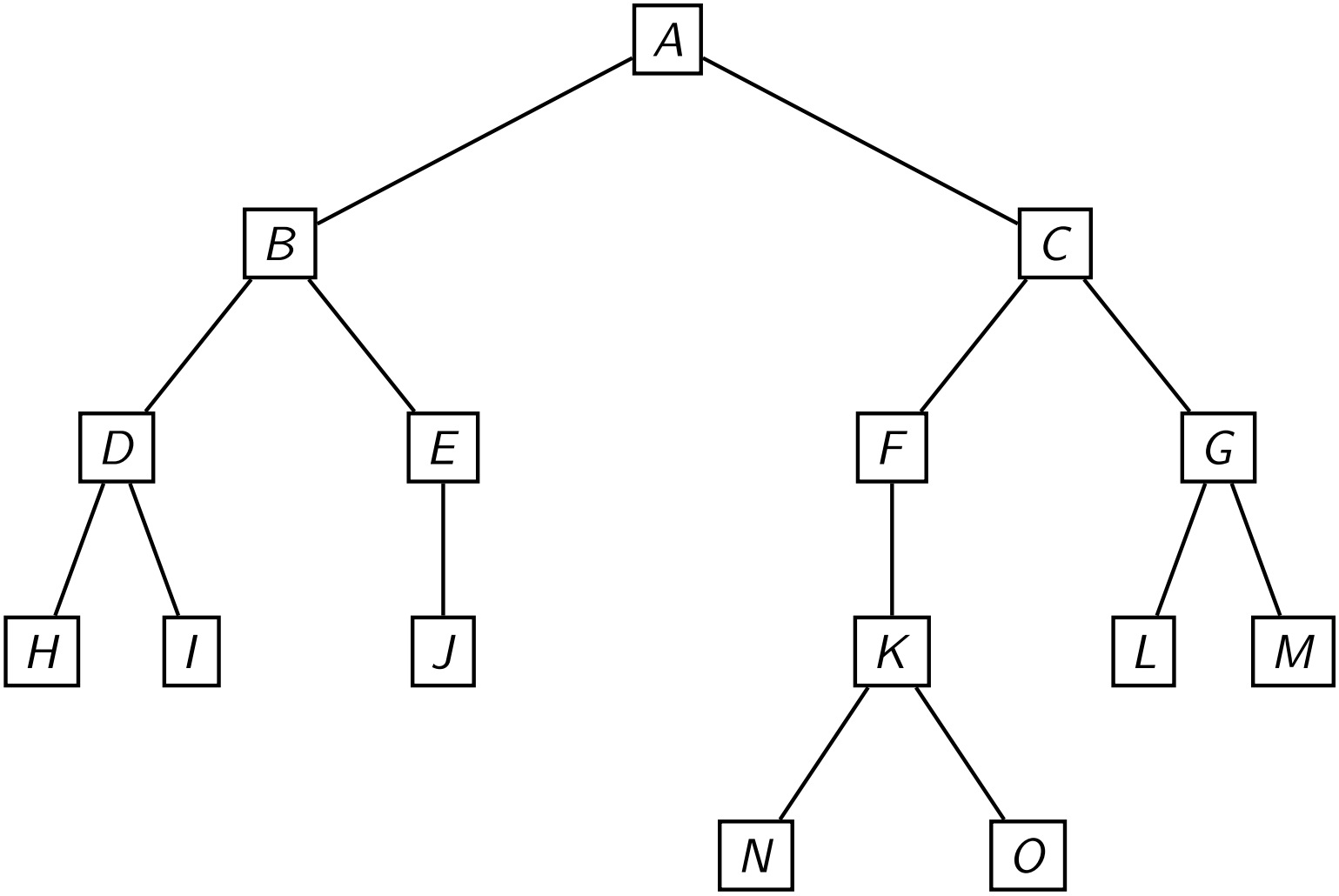
```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        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
```

Expands deepest unexpanded node

The frontier is a *stack* – new successors go at the front.

DFS is often developed as a recursive function.

Depth first search



Node	Explored	Frontier
		A
A	A	B,C
B	A, B	D, E, C
D	A, B, D	H, I, E, C
H	A, B, D, H	I, E, C
I	A, B, D, H, I	E, C
E	A, B, D, H, I, E	J, C
J	A, B, D, H, I, E, J	C
C	A, B, D, H, I, E, J, C	F, G
F	A, B, D, H, I, E, J, C, F	K, G
K	A, B, D, H, I, E, J, C, F, K	N, O, G
N	A, B, D, H, I, E, J, C, F, K, N	O, G
O	A, B, D, H, I, E, J, C, F, K, N, O	G
G	A, B, D, H, I, E, J, C, F, K, N, O, G	Goal found.

Exercise: Apply DFS to the same graph that you applied BFS and UCS to. Compare the results.

DFS properties

We use a special trick for visited states – only remember states along path from root to current node

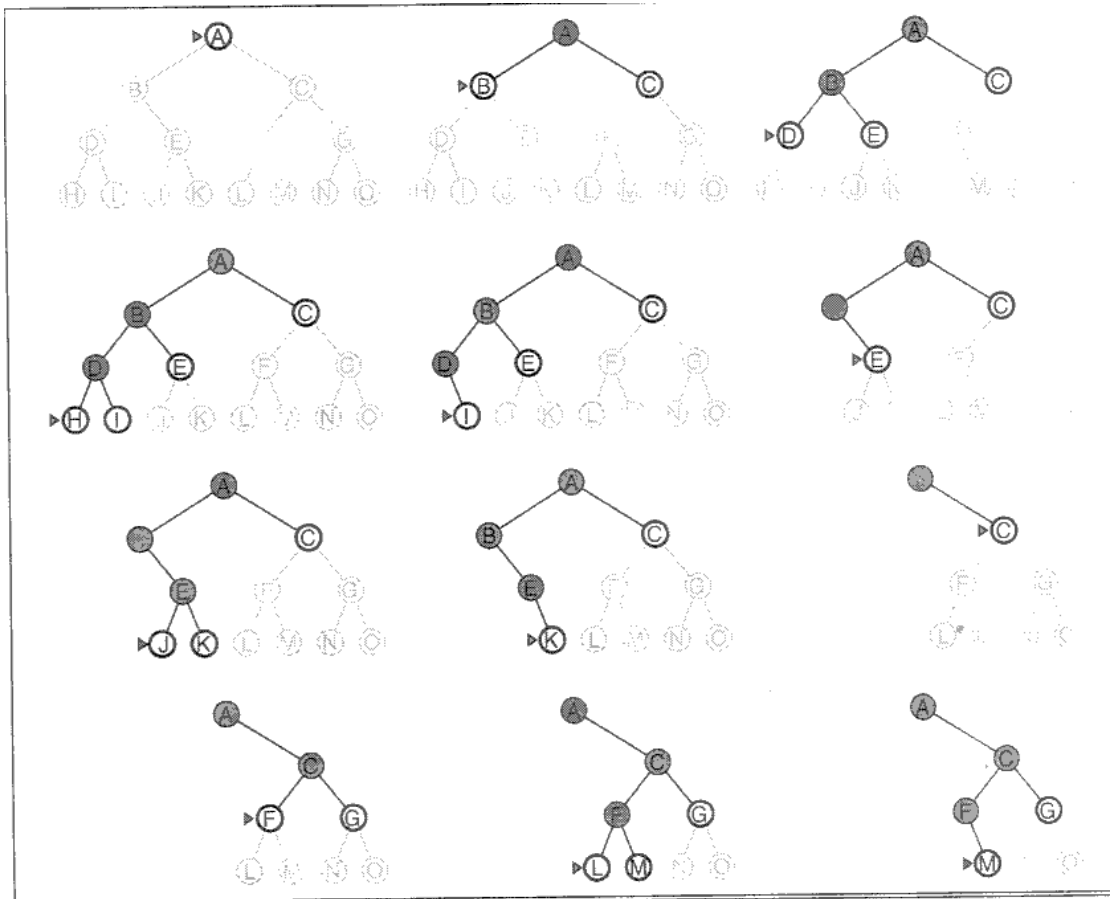


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

DFS properties

- Complete?
  Yes, for finite spaces
- Time?
  $O(b^m)$ – terrible if $m >> d$
- Space?
  Remember only path from root to current node (and unexplored siblings along path: $O(bm)$
- Optimal
  No, finds "leftmost" solution!

Depth-limited DFS

- Solution might be at finite depth, but if $m = \infty$, DFS will never find it!
- Solution: just limit max depth to 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
```

Depth-limited DFS

- Time is now $O(b^l)$ and space is $O(bl)$
- But how to pick l? And what if $l < d$?
- Could use domain knowledge e.g. if you know solution is at most k, pick $l = k$
- Or use diameter: max shortest distance between any 2 states

Iterative deepening

Instead, we can just try multiple depths!
i.e. RUN DFS with $l = 1$
If solution, great! If not, run it again with $l = 2$
Keep going until you find solution!

```
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 benefits of DFS + BFS
Seems wasteful!
But is it?

IDS Analysis

- If we ran depth-limited DFS to depth d
  $N_{DLS} = b^0 + b^1 + b^2 + ... + b^d$
- If we run IDS $l = 0, 1, \ldots, d$
  $N_{IDS} = (d+1)b^0 + db^1 + (d-1)b^2 + ... + 2b^{d-1} + b^d$
- For $b = 10$, $d = 5$
  $N_{DLS} = 111111$
  $N_{IDS} = 123456$
  11% difference!
- Rule of thumb: if search space is large and depth of solution unknown, use IDS

IDS properties

- Complete?
  Yes, it's like BFS
- Time?
  $(d+1)b^0 + db^1 + (d-1)b^2 + ... + 2b^{d-1} + b^d$
- Space?
  $O(bd)$
- Optimal:
  Only if step costs are same everywhere (like BFS)

Summary

- BFS – time and space can be issues
- DFS – time and space can be issues
- DLS – limit depth of search which could help
- IDS – can be useful