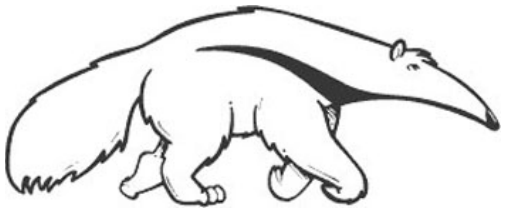
Uninformed Search

CS171, Summer 1 Quarter, 2019
Introduction to Artificial Intelligence
Prof. Richard Lathrop



Read Beforehand: R&N 3.4

Uninformed search strategies

- Uninformed (blind):
  - You have no clue whether one non-goal state is better than any other. Your search is blind. You don't know if your current exploration is likely to be fruitful.

- Various blind strategies:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Iterative deepening search (generally preferred)
  - Bidirectional search (preferred if applicable)

Basic graph/tree search scheme

- We have 3 kinds of states:
  - [only for graph search: explored (past states; = closed list)]
  - frontier (current nodes; = open list, fringe, queue) [nodes now on the queue]
  - unexplored (future nodes) [implicitly given]
- Initially, frontier = MakeNode(start state)
- Loop until solution is found or state space is exhausted
  - pick/remove first node from queue/frontier/fringe/open using search strategy
  - if node is a goal then return node
  - [only for graph search: add node to explored/closed]
  - expand this node, add children to frontier only if not already in frontier
    - [only for graph search: add children only if their state is not in explored/closed list]

- Question:
  - what if a better path is found to a node already in frontier or on explored list?

Search strategy evaluation

- A search strategy is defined by 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 (always finite)
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)
  - (for UCS: C*: true cost to optimal goal; $\varepsilon > 0$: minimum step cost)

Uninformed search design choices

- Queue for Frontier:
  - FIFO? LIFO? Priority? If Priority, what sort function?
- Goal-Test:
  - Do goal-test when node inserted into *Frontier*?
  - Do goal-test when node removed?
- Tree Search, or Graph Search:
  - Forget *Expanded* (or *Explored*, *Closed,* Fig. 3.7) nodes?
    - = Tree Search: Smaller memory cost, but larger search time
  - Or remember them?
    - = Graph Search: Smaller search time, but larger memory cost
  - Classic space/time computational tradeoff

Queue for Frontier

- FIFO (First In, First Out)
  - Results in Breadth-First Search
- LIFO (Last In, First Out)
  - Results in Depth-First Search
- Priority Queue sorted by path cost so far
  - Results in Uniform Cost Search
- Iterative Deepening Search uses Depth-First
- Bidirectional Search can use either Breadth-First or Uniform Cost Search

When to do goal test? (General)

- Do Goal-Test when node is popped from queue:

  IF you care about finding the optimal path

  AND your search space may have both short expensive and long cheap paths to a goal.
  - Guard against a short expensive goal.
  - E.g., Uniform Cost search with variable step costs.
- Otherwise, do Goal-Test when is node generated and inserted.
  - Usually, most of the search cost goes into creating the children (storage allocation, data structure creation, etc.), while the goal-test is usually fast and light-weight (am I in Bucharest? even the complicated 'check-mate?' goal-test in chess usually is fast because it does little or no storage allocation or data structure creation).
  - So most efficient search does goal-test as soon as nodes are generated.
- REASON ABOUT your search space & problem.
  - How could I possibly find a non-optimal goal?

When to do Goal-Test? (Summary)

- For BFS, the goal test is done when the child node is generated.
  - Not an optimal search in the general case.
- For DLS, IDS, and DFS as in Fig. 3.17, goal test is done in the recursive call.
  - Result is that children are generated then iterated over. For each child DLS, is called recursively, goal-test is done first in the callee, and the process repeats.
  - More efficient search goal-tests children as generated. We follow your text.
- For DFS as in Fig. 3.7, goal test is done when node is popped.
  - Search behavior depends on how the LIFO queue is implemented.
- For UCS and A*(next lecture), goal test when node removed from queue.
  - This avoids finding a short expensive path before a long cheap path.
- Bidirectional search can use either BFS or UCS.
  - Goal-test is search fringe intersection, see additional complications below
- For GBFS (next lecture) the behavior is the same either way
  - h(goal)=0 so any goal will be at the front of the queue anyway.

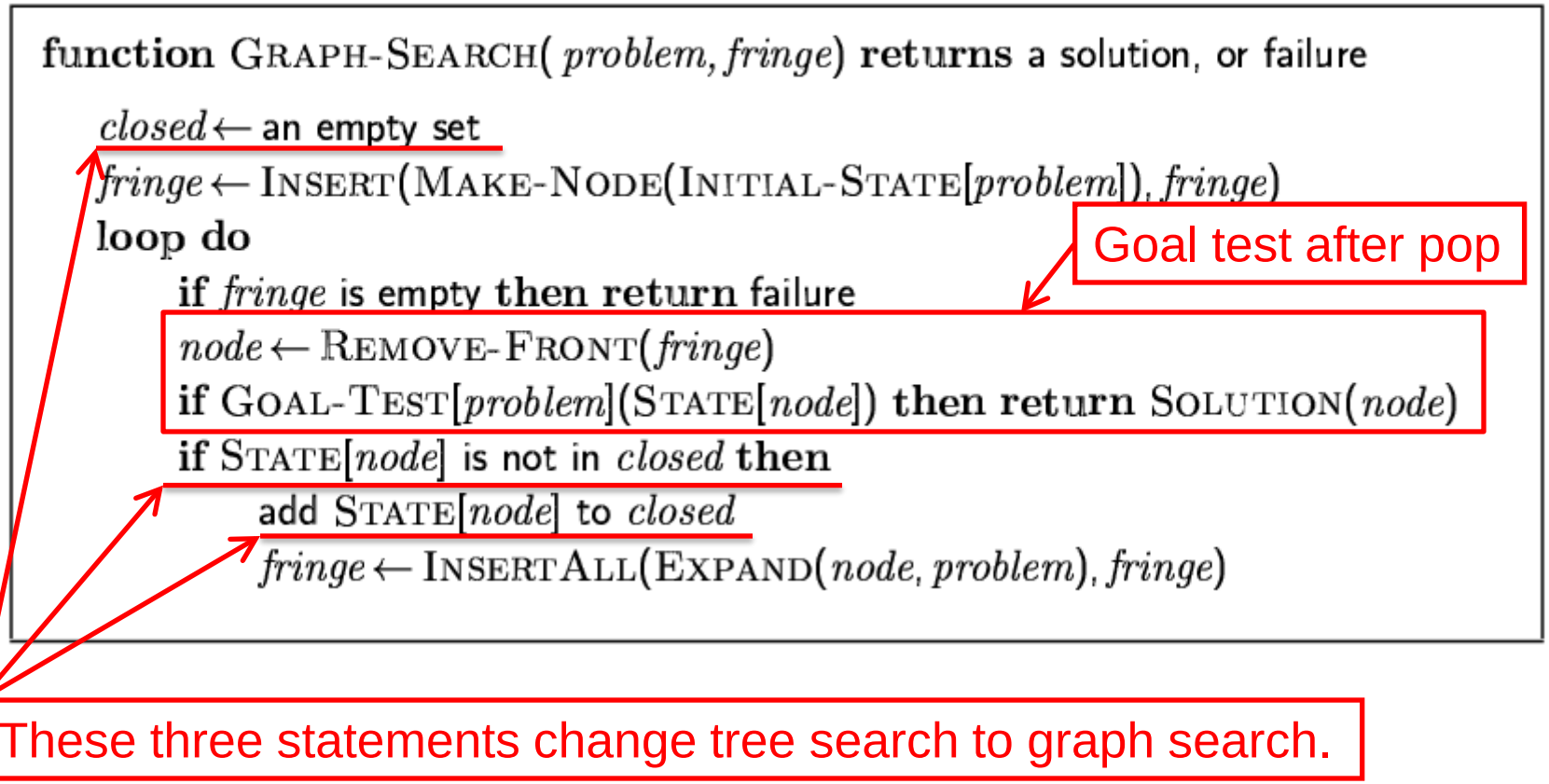
General tree search (R&N Fig. 3.7)
Do not remember visited nodes

```
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 SOLUTION(node)
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Goal test after pop

```
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
```

General graph search (R&N Fig. 3.7)
Do remember visited nodes



Tree-Search vs. Graph-Search

- Example : Assemble 5 objects {a, b, c, d, e}
- A state is a bit-vector (length 5), 1=object in assembly, 0= not in assembly
  - 11010 = a=1, b=1, c=0, d=1, e=0
  - $\Rightarrow$ a, b, d in assembly; c, e not in assembly
- State space:
  - Number of states = 2^5 = 32
  - Number of undirected edges = (2^5)·5·½ = 80

- Tree search space:
  - **Number of nodes** = number of paths = 5! **= 120**
  - States can be reached in multiple ways
    - 11010 can be reached by a+b+d or by a+d+b or by … etc.
  - Often requires much more time, but much less space, than graph search
- Graph search space:
  - **Number of nodes** = choose(5,0) + choose(5,1) + choose(5,2) + choose(5,3) + choose(5,4) + choose(5,5) = 1 + 5 + 10 + 10 + 5 + 1 **= 32**
  - States are reached in only one way, redundant paths are pruned
    - Question: What if a better path is found to a state that already has been explored?
  - Often requires much more space, but much less time, than tree search

Checking for identical nodes (1)
Check if a node is already in fringe-frontier

- It is "easy" to check if a node is already in the fringe/frontier (recall fringe = frontier = open = queue)
  - Keep a hash table holding all fringe/frontier nodes
    - Hash size is same O(.) as priority queue, so hash does not increase overall space O(.)
    - Hash time is O(1), so hash does not increase overall time O(.)
  - When a node is expanded, remove it from hash table (it is no longer in the fringe/frontier)
  - For each resulting child of the expanded node:
    - If child is not in hash table, add it to queue (fringe) and hash table
    - Else if an old lower- or equal-cost node is in hash, discard the new higher- or equal-cost child
    - Else remove and discard the old higher-cost node from queue and hash, and add the new lower-cost child to queue and hash

Always do this for tree or graph search in BFS, UCS, GBFS, and A*

Checking for identical nodes (2)

Check if a node is in explored/expanded

- It is memory-intensive [$O(b^d)$ or $O(b^m)$]to check if a node is in explored/expanded (recall explored = expanded = closed)
  - Keep a hash table holding all explored/expanded nodes (hash table may be HUGE!!)
- When a node is expanded, add it to hash (explored)
- For each resulting child of the expanded node:
  - If child is not in hash table or in fringe/frontier, then add it to the queue (fringe/frontier) and process normally (BFS normal processing differs from UCS normal processing, but the ideas behind checking a node for being in explored/expanded are the same).
  - Else discard any redundant node.

Always do this for graph search

Checking for identical nodes (3)

Quick check for search being in a loop

- It is “moderately easy” to check for the search being in a loop
  - When a node is expanded, for each child:
    - Trace back through parent pointers from child to root
    - If an ancestor state is identical to the child, search is looping
      - Discard child and fail on that branch
    - Time complexity of child loop check is O(depth(child))
    - Memory consumption is zero
      - Assuming good garbage collection
- Does NOT solve the general problem of repeated nodes — only the specific problem of looping
- For quizzes and exams, we will follow your textbook and NOT perform this loop check

Breadth-first graph search (R&N Fig. 3.11)

```
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)
```

Figure 3.11 Breadth-first search on a graph.

Goal test before push

Avoid redundant frontier nodes

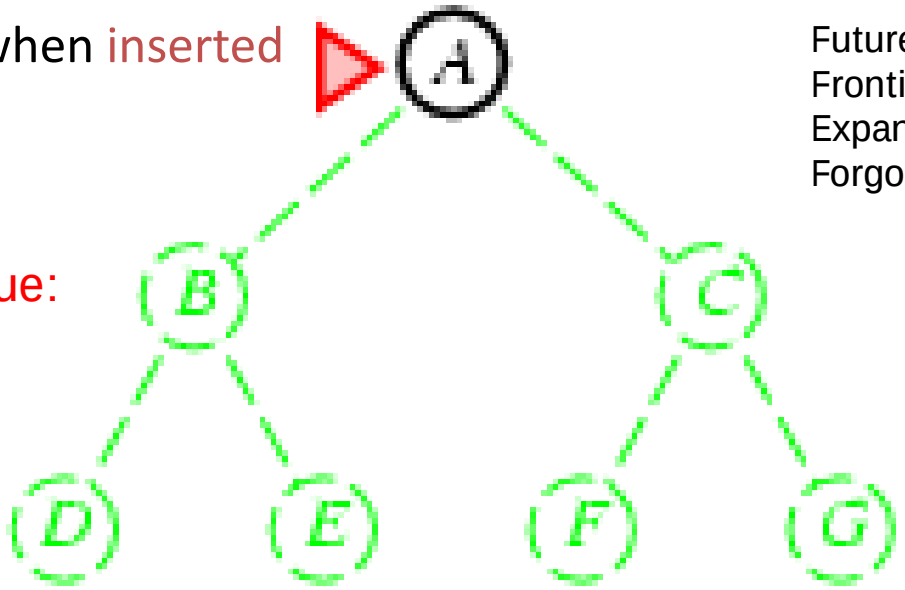
These three statements change tree search to graph search.

Breadth-first search

- Expand shallowest unexpanded node
- Frontier: nodes waiting in a queue to be explored
  - also called Fringe, or OPEN
- Implementation:
  - *Frontier* is a first-in-first-out (FIFO) queue (new successors go at end)
  - Goal test when inserted

Initial state = A
Is A a goal state?

Put A at end of queue:
Frontier = [A]



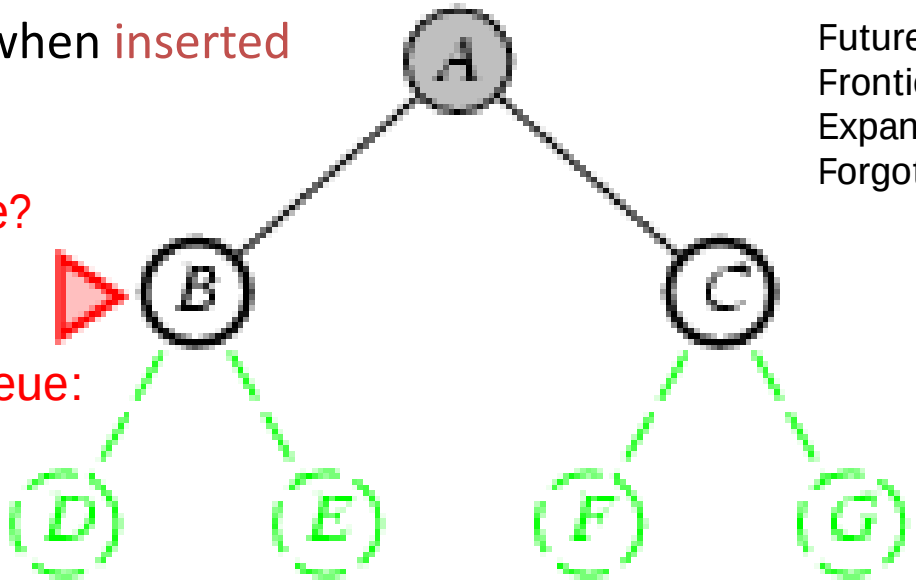
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Breadth-first search

- Expand shallowest unexpanded node
- Frontier: nodes waiting in a queue to be explored
  - also called Fringe, or OPEN
- Implementation:
  - *Frontier* is a first-in-first-out (FIFO) queue (new successors go at end)
  - Goal test when inserted

Expand A to B,C
Is B or C a goal state?

Put B,C at end of queue:
Frontier = [B,C]



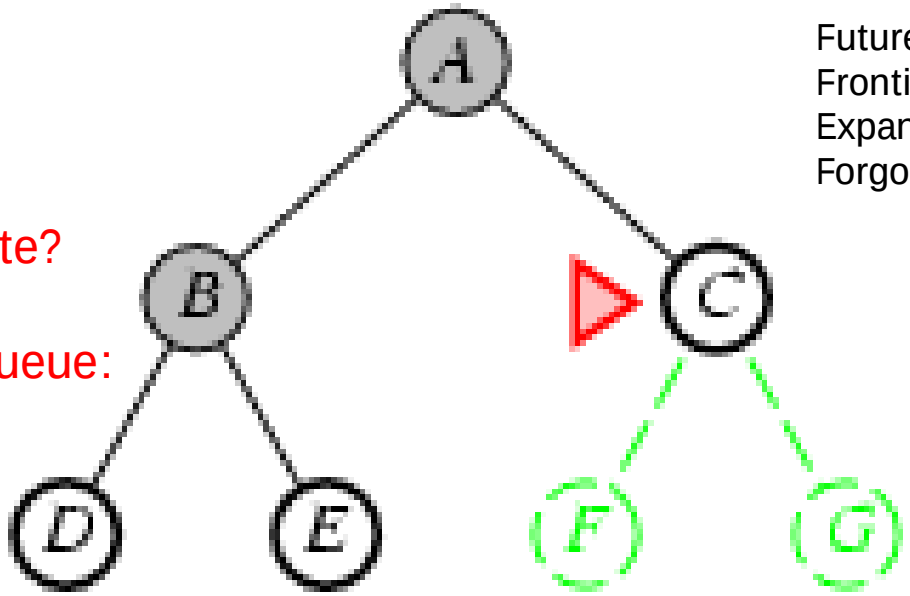
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Breadth-first search

- Expand shallowest unexpanded node
- Frontier: nodes waiting in a queue to be explored
  - also called Fringe, or OPEN
- Implementation:
  - *Frontier* is a first-in-first-out (FIFO) queue (new successors go at end)
  - Goal test v

Expand B to D,E
Is D or E a goal state?

Put D,E at end of queue:
Frontier = [C,D,E]



Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

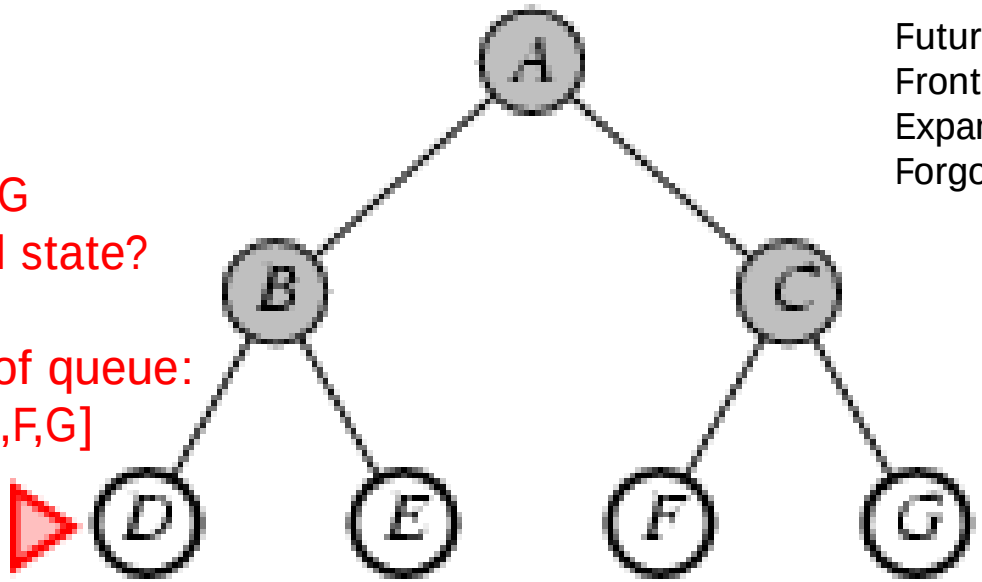
Breadth-first search

- Expand shallowest unexpanded node
- Frontier: nodes waiting in a queue to be explored
  - also called Fringe, or OPEN
- Implementation:
  - *Frontier* is a first-in-first-out (FIFO) queue (new successors go at end)
  - Goal te

Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Expand C to F, G
Is F or G a goal state?

Put F,G at end of queue:
Frontier = [D,E,F,G]

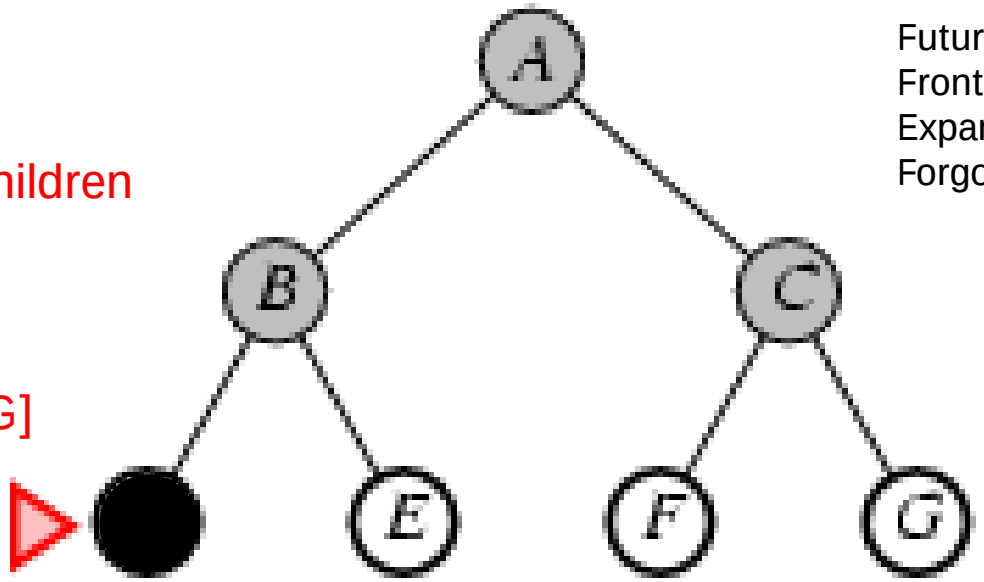


Breadth-first search

- Expand shallowest unexpanded node
- Frontier: nodes waiting in a queue to be explored
  - also called Fringe, or OPEN
- Implementation:
  - *Frontier* is a first-in-first-out (FIFO) queue (new successors go at end)
  - Goal te

Expand D; no children
Forget D

Frontier = [E,F,G]



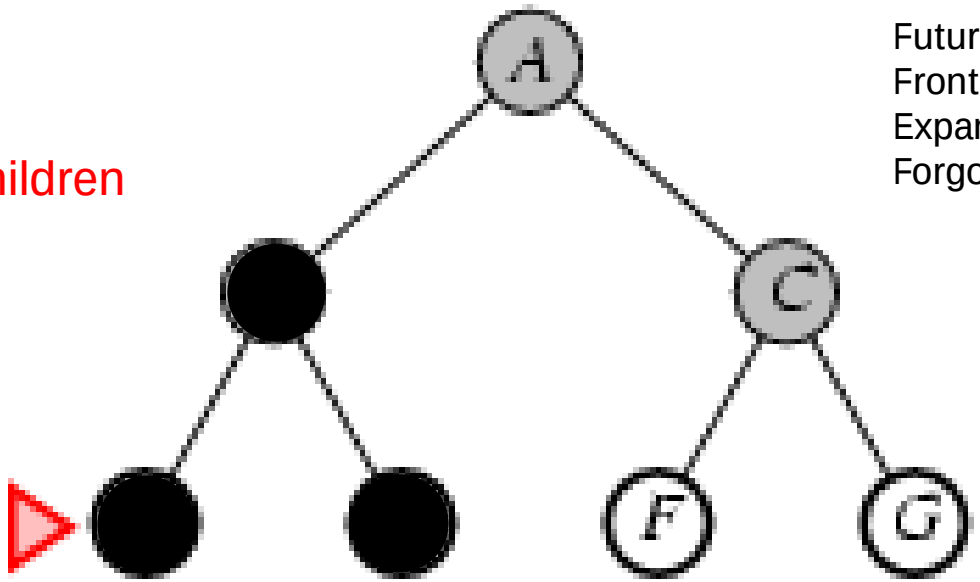
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Breadth-first search

- Expand shallowest unexpanded node
- Frontier: nodes waiting in a queue to be explored
  - also called Fringe, or OPEN
- Implementation:
  - *Frontier* is a first-in-first-out (FIFO) queue (new successors go at end)
  - Goal te

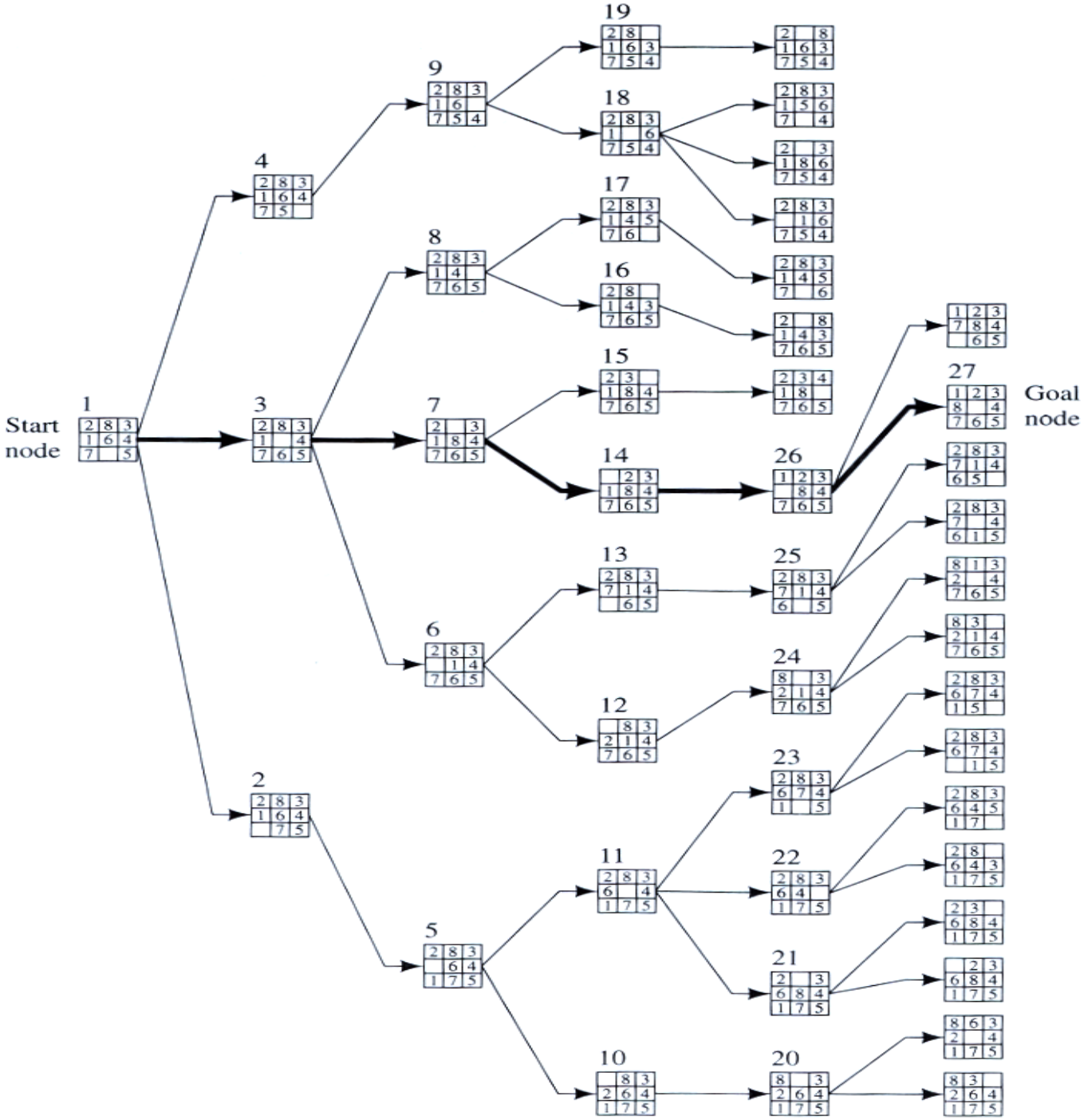
Expand E; no children
Forget E; B

Frontier = [F,G]



Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Example BFS for 8-puzzle



Properties of breadth-first search

- Complete? Yes, it always reaches a goal (if *b* is finite)
- Time? $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

  (this is the number of nodes we generate)
- Space? $O(b^d)$

  (keeps every node in memory, either in frontier or on a path to frontier).
- Optimal? No, for general cost functions.

  Yes, if cost is a non-decreasing function only of depth.
  - With $f(d) \geq f(d-1)$, e.g., step-cost = constant:
    - All optimal goal nodes occur on the same level
    - Optimal goals are always shallower than non-optimal goals
    - An optimal goal will be found before any non-optimal goal

- Usually Space is the bigger problem (more than time)

BFS: Time & Memory Costs

Depth of Solution	Nodes Expanded	Time	Memory
0	1	5 microseconds	100 bytes
2	111	0.5 milliseconds	11 kbytes
4	11,111	0.05 seconds	1 megabyte
8	10^8	9.25 minutes	11 gigabytes
12	10^{12}	64 days	111 terabytes

Assuming b=10; 200k nodes/sec; 100 bytes/node

Uniform-cost search

Breadth-first is only optimal if path cost is a non-decreasing function of depth, i.e., g(d) ≥ g(d-1); e.g., constant step cost, as in the 8-puzzle.

Can we guarantee optimality for variable positive step costs $\geq\varepsilon$?

(Why $\geq\varepsilon$? To avoid infinite paths w/ step costs 1, ½, ¼, …)

Uniform-cost Search:

Expand node with smallest path cost g(n).

- *Frontier* is a priority queue, i.e., new successors are merged into the queue sorted by g(n).
  - Can remove successors already on queue w/higher g(n).
    - Saves memory, costs time; another space-time trade-off.
- *Goal-Test* when node is popped off queue.

Uniform cost search (R&N Fig. 3.14) [A* is identical except queue sort = f(n)]

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

Goal test after pop

Avoid redundant frontier nodes

Avoid higher-cost frontier nodes

Figure 3.14 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for frontier needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

These three statements change tree search to graph search.

Uniform-cost search

Proof of Completeness:

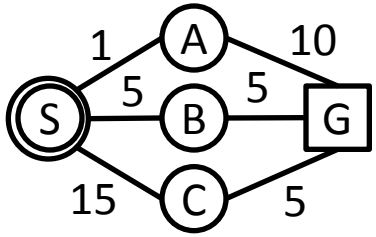
Assume (1) finite max branching factor = *b;* (2) min step cost $\geq \varepsilon > 0$; (3) cost to optimal goal = C^*. Then a node at depth $\lfloor 1+C^*/\varepsilon \rfloor$ must have a path cost > C*. There are $O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$ such nodes, so a goal will be found.

Proof of Optimality (given completeness):

Suppose that UCS is not optimal. Then there must be an (optimal) goal state with path cost smaller than the found (suboptimal) goal state (invoking completeness). However, this is impossible because UCS would have expanded that node first, by definition. Contradiction.

(Search tree version)

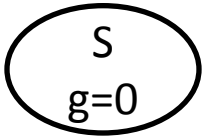
Example: Uniform-cost search



Order of node expansion: _____

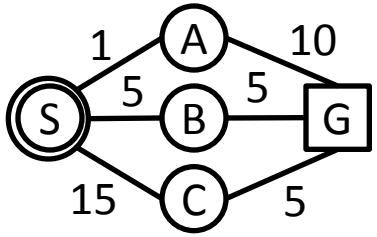
Path found: _____ Cost of path found: _____

Route finding problem.
Steps labeled w/cost.



(Search tree version)

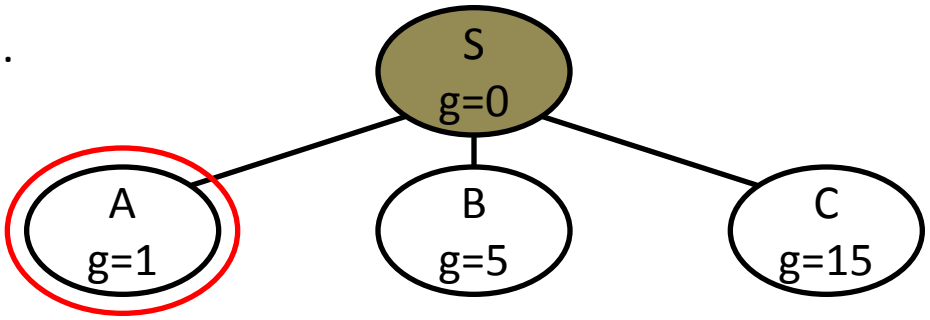
Example: Uniform-cost search



Order of node expansion: S

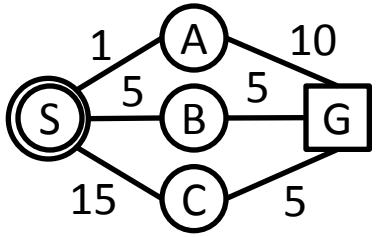
Path found: _____ Cost of path found: _____

Route finding problem.
Steps labeled w/cost.



(Search tree version)

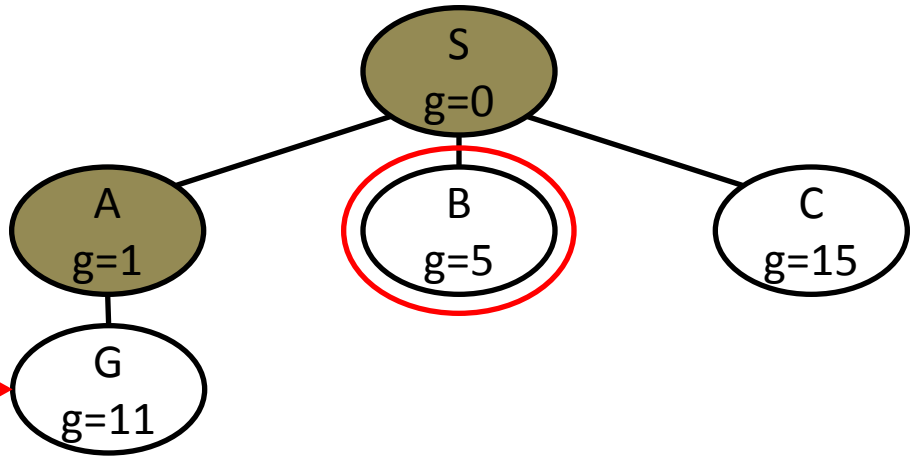
Example: Uniform-cost search



Order of node expansion: S A

Path found: _____ Cost of path found: _____

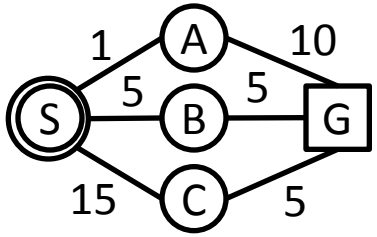
Route finding problem.
Steps labeled w/cost.



This early expensive goal node will go back onto the queue until after the later cheaper goal is found.

(Search tree version)

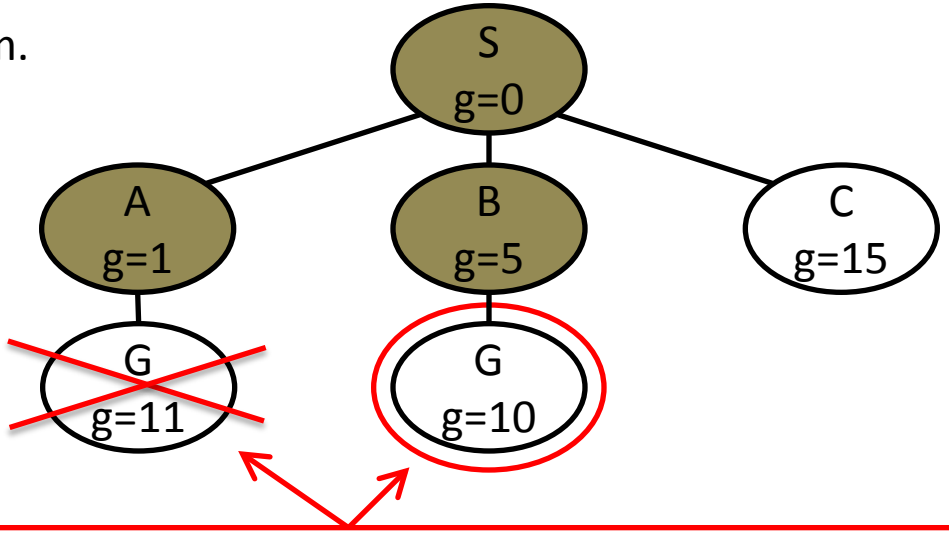
Example: Uniform-cost search



Order of node expansion: S A B

Path found: _____ Cost of path found: _____

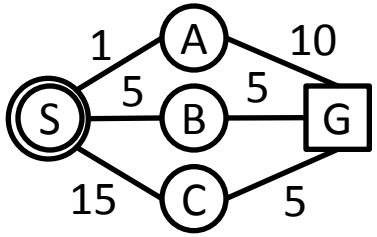
Route finding problem.
Steps labeled w/cost.



Remove the higher-cost of identical nodes on the queue and save memory. However, UCS is optimal even if this is not done, since lower-cost nodes sort to the front.

(Search tree version)

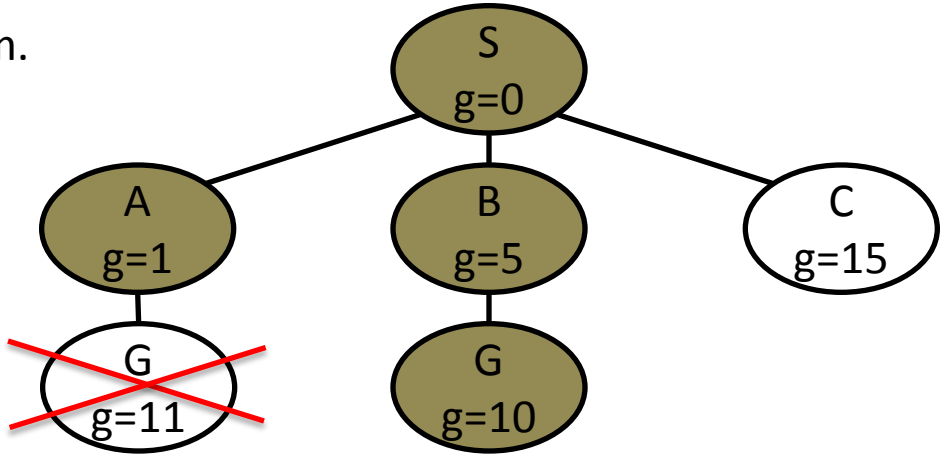
Example: Uniform-cost search



Order of node expansion: S A B G

Path found: S B G Cost of path found: 10

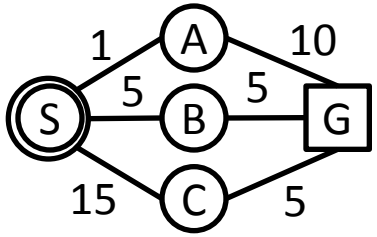
Route finding problem.
Steps labeled w/cost.



Technically, the goal node is not really expanded, because we do not generate the children of a goal node. It is listed in “Order of node expansion” only for your convenience, to see explicitly where it was found.

(Virtual queue version)

Example: Uniform-cost search



Order of node expansion: _____

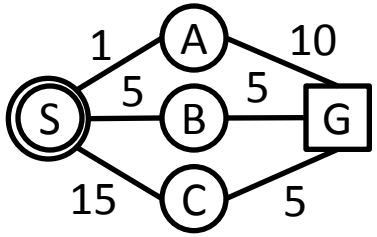
Path found: _____ Cost of path found: _____

Route finding problem.
Steps labeled w/cost.

Expanded:
Next:
Children:
Queue: S/g=0

(Virtual queue version)

Example: Uniform-cost search



Order of node expansion: S

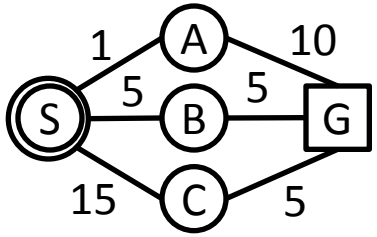
Path found: _____ Cost of path found: _____

Route finding problem.
Steps labeled w/cost.

Expanded: S/g=0
Next: S/g=0
Children: A/g=1, B/g=5, C/g=15
Queue: S/g=0, A/g=1, B/g=5, C/g=15

(Virtual queue version)

Example: Uniform-cost search



Order of node expansion: S A

Path found: _____ Cost of path found: _____

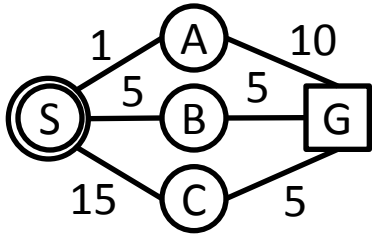
Route finding problem.
Steps labeled w/cost.

Expanded: S/g=0, A/g=1
Next: A/g=1
Children: G/g=11
Queue: S/g=0, A/g=1, B/g=5, C/g=15, G/g=11

Note that in a proper priority queue in a computer system, this queue would be sorted by g(n). For hand-simulated search it is more convenient to write children as they occur, and then scan the current queue to pick the highest-priority node on the queue.

(Virtual queue version)

Example: Uniform-cost search



Order of node expansion: S A B

Path found: _____ Cost of path found: _____

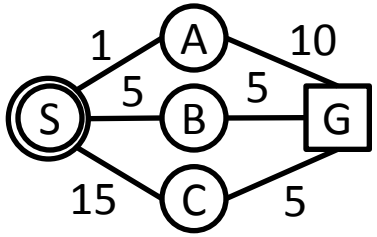
Route finding problem.
Steps labeled w/cost.

Expanded: S/g=0, A/g=1, B/g=5
Next: B/g=5
Children: G/g=10
Queue: S/g=0, A/g=1, B/g=5, C/g=15, ~~G/g=11~~, G/g=10

Remove the higher-cost of identical nodes on the queue and save memory. However, UCS is optimal even if this is not done, since lower-cost nodes sort to the front.

(Virtual queue version)

Example: Uniform-cost search



Route finding problem.
Steps labeled w/cost.

Order of node expansion: S A B G

Path found: S B G Cost of path found: 10

The same "Order of node expansion", "Path found", and "Cost of path found" is obtained by both methods. They are formally equivalent to each other in all ways.

Expanded: S/g=0, A/g=1, B/g=5, G/g=10
Next: G/g=10
Children: none
Queue: S/g=0, A/g=1, B/g=5, C/g=15, ~~G/g=11~~, G/g=10

Technically, the goal node is not really expanded, because we do not generate the children of a goal node. It is listed in "Order of node expansion" only for your convenience, to see explicitly where it was found.

Uniform-cost search

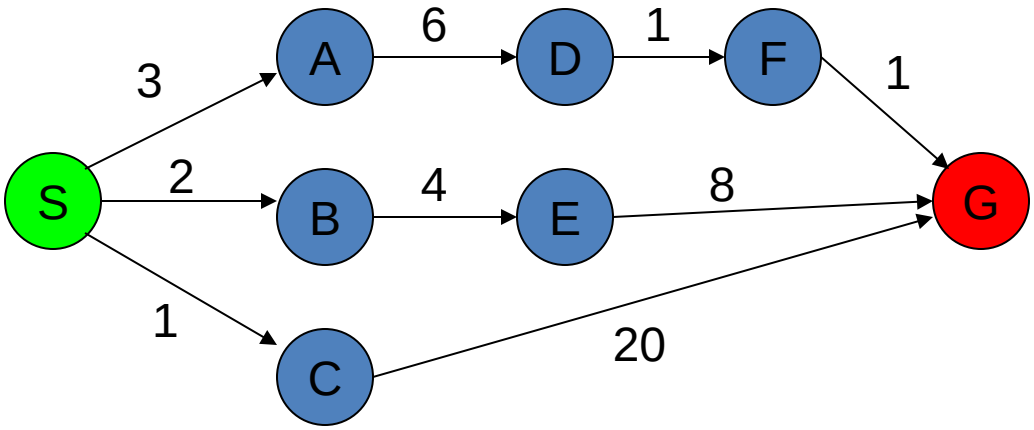
Implementation: *Frontier* = queue ordered by path cost.
Equivalent to breadth-first if all step costs all equal.

- Complete? Yes, if b is finite and step cost $\geq \varepsilon > 0$.
  (otherwise it can get stuck in infinite regression)

- Time? # of nodes with path cost ≤ cost of optimal solution.
  $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$

- Space? # of nodes with path cost ≤ cost of optimal solution.
  $O(b^{\lfloor 1+C^*/\varepsilon \rfloor}) \approx O(b^{d+1})$.

- Optimal? Yes, for step cost $\geq \varepsilon > 0$.



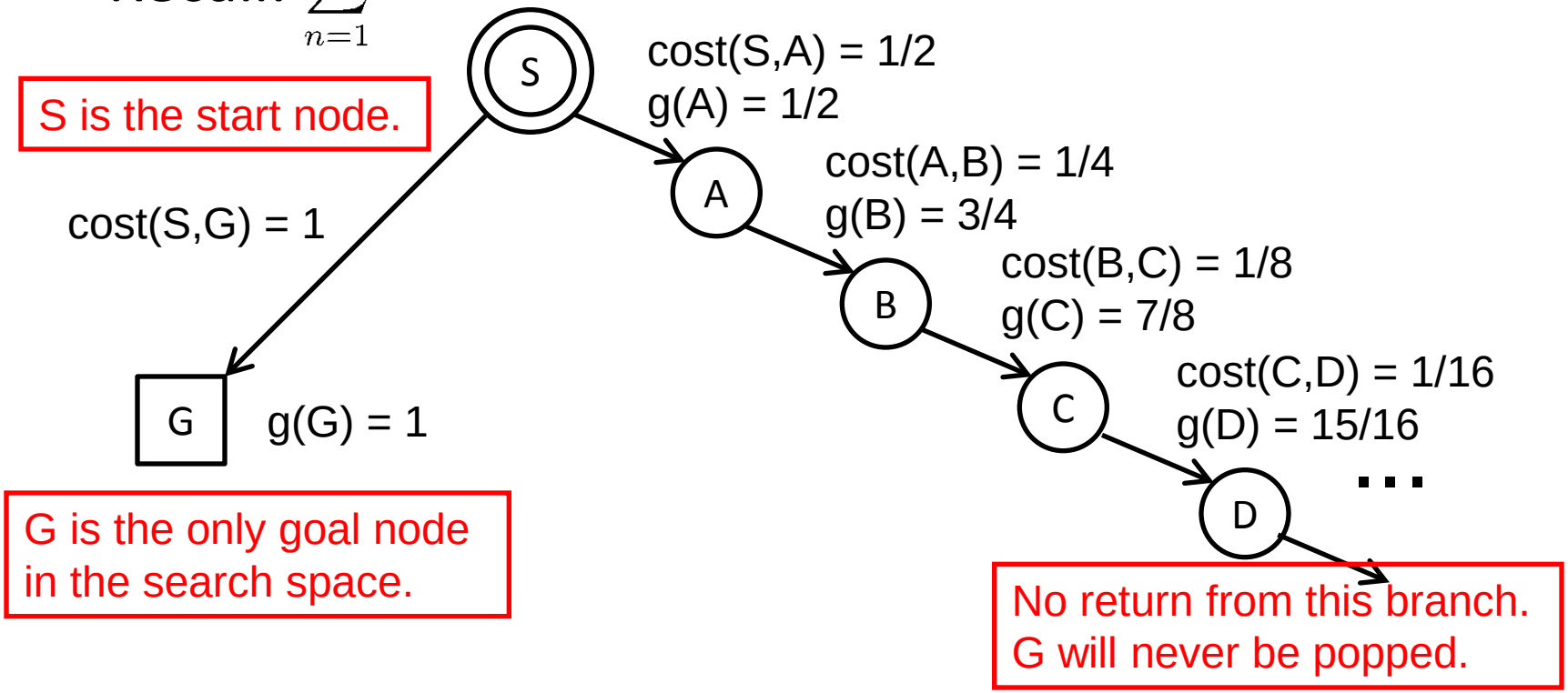
The graph above shows the step-costs for different paths going from the start (S) to the goal (G).

Use uniform cost search to find the optimal path to the goal.

Exercise for home

Uniform cost search

- Why require step cost $\geq \varepsilon > 0$?
  - Otherwise, an infinite regress is possible.
  - Recall: $\sum_{n=1}^{\infty} 2^{-n} = 1$



Iterative Deepening Search

- To avoid the infinite depth problem of DFS:
  - Only search until depth L
  - i.e, don't expand nodes beyond depth L
  - Depth-Limited Search
- What if solution is deeper than L?
  - Increase depth iteratively
  - Iterative Deepening Search
- IDS — GENERALLY THE PREFERRED UNINFORMED SEARCH
  - Inherits the memory advantage of depth-first search
  - Has the completeness property of breadth-first search

Depth-limited search & IDS (R&N Fig. 3.17-18)

```
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 SOLUTION(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
```

Goal test in recursive call, one-at-a-time

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

At *depth* = 0, IDS only goal-tests the start node. The start node is is not expanded at *depth* = 0.

Iterative Deepening Search, L=0

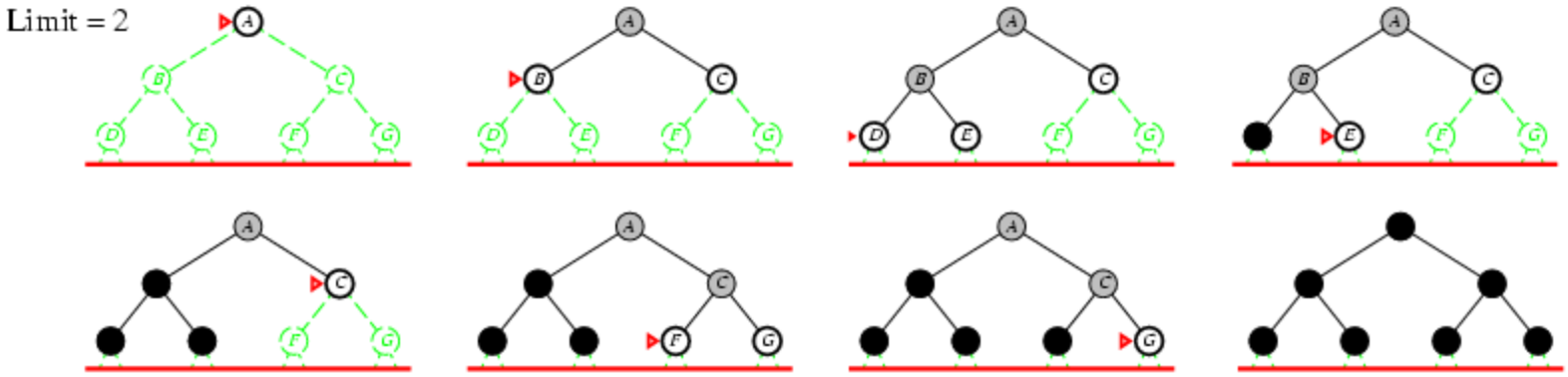
Limit = 0

At L=0, the start node is goal-tested but no nodes are expanded. This is so that you can solve trick problems like, “Starting in Arad, go to Arad.”

Iterative Deepening Search, L=1

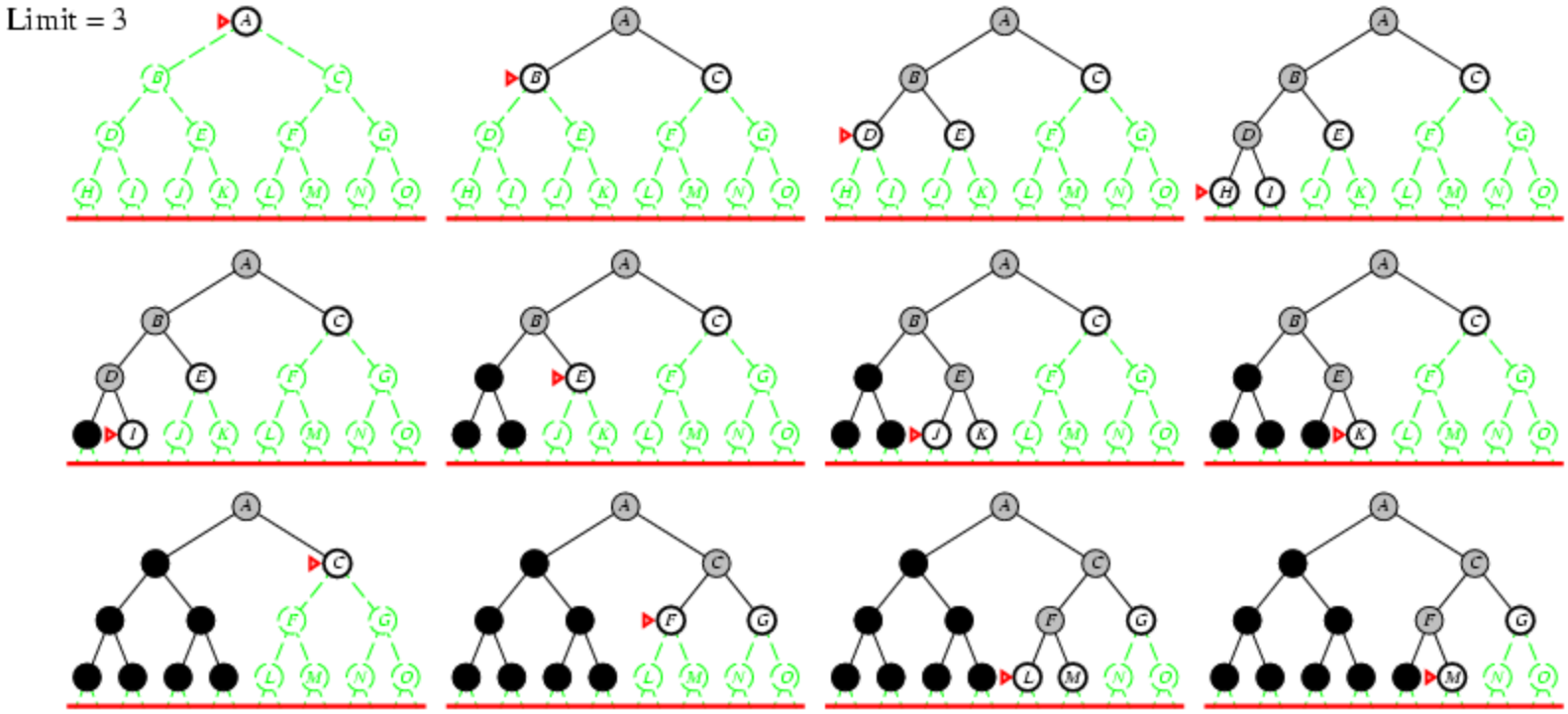
At L=1, the start node is expanded. Its children are goal-tested, but not expanded. Recall that to expand a node means to generate its children.

Iterative Deepening Search, L=2



At L=2, the start node and its children are expanded. Its grand-children are goal-tested, but not expanded.

Iterative Deepening Search, L=3



At L=3, the start node, its children, and its grand-children are expanded. Its great-grand-children are goal-tested, but not expanded.

Iterative Deepening Search

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

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

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

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

$= O(b^d)$

- For *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$

[Ratio: b/(b-1)]

Properties of iterative deepening search

- Complete? Yes
- Time? $O(b^d)$
- Space? *O(bd)*
- Optimal? No, for general cost functions.
  Yes, if cost is a non-decreasing function only of depth.

Generally the preferred uninformed search strategy.

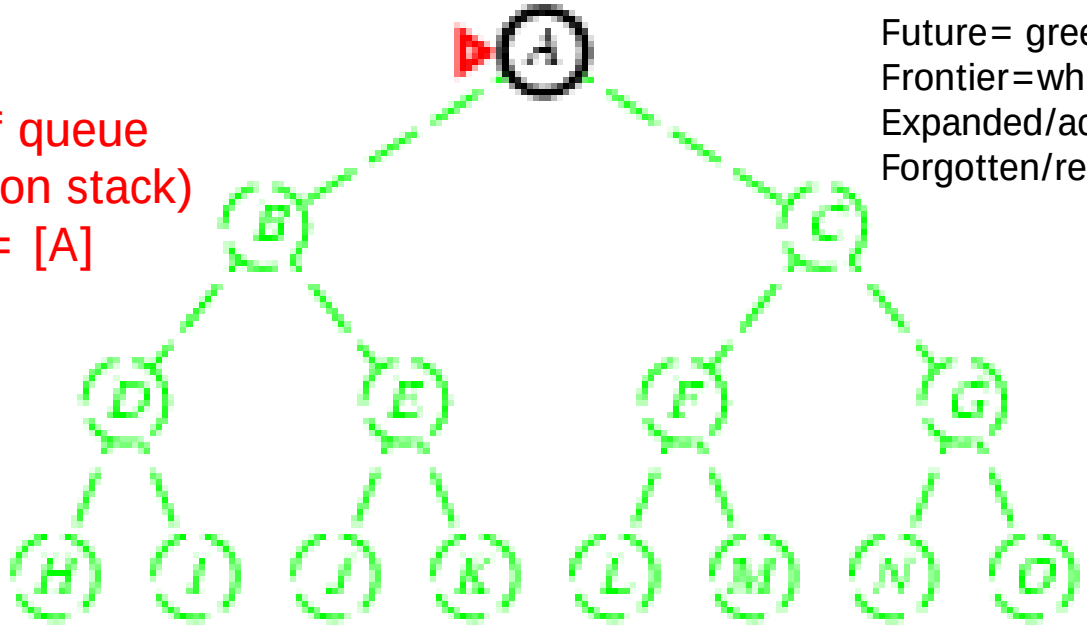
Depth-First Search (R&N Section 3.4.3)

- Your textbook is ambiguous about DFS.
  - The second paragraph of R&N 3.4.3 states that DFS is an instance of Fig. 3.7 using a LIFO queue. Search behavior may differ depending on how the LIFO queue is implemented (as separate pushes, or one concatenation).
  - The third paragraph of R&N 3.4.3 says that an alternative implementation of DFS is a recursive algorithm that calls itself on each of its children, as in the Depth-Limited Search of Fig. 3.17 (above).
- **For quizzes and exams, we will follow Fig. 3.17.**

Depth-first search

- Expand *deepest* unexpanded node
- *Frontier* = Last In First Out (LIFO) queue, i.e., new successors go at the front of the queue.
- *Goal-Test* first step of recursive call (R&N, Fig. 3.17).

Initial state = A
Put A at front of queue
(note: queue is on stack)
queue/frontier = [A]



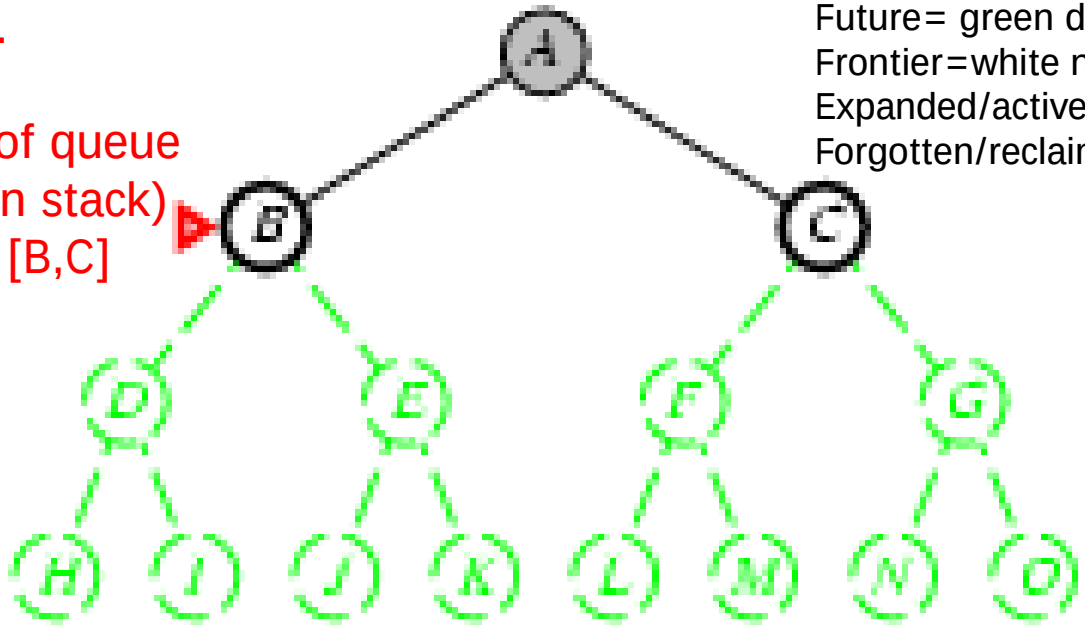
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Is A a goal state? No.
Expand A to B, C.

Put B, C at front of queue
(note: queue is on stack)
queue/frontier = [B,C]



Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

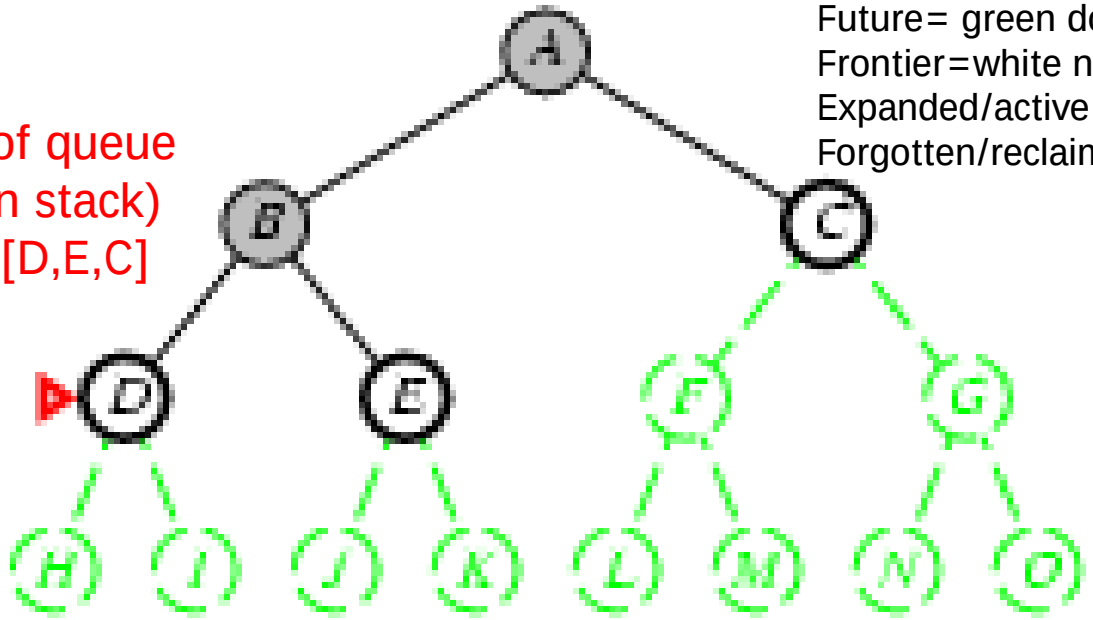
Note: Can save a space factor of *b* by generating successors one at a time.
See **backtracking search** in your book, p. 87 and Chapter 6.

Depth-first search

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

Is B a goal state? No.
Expand B to D, E.

Put D, E at front of queue
(note: queue is on stack)
queue/frontier = [D,E,C]



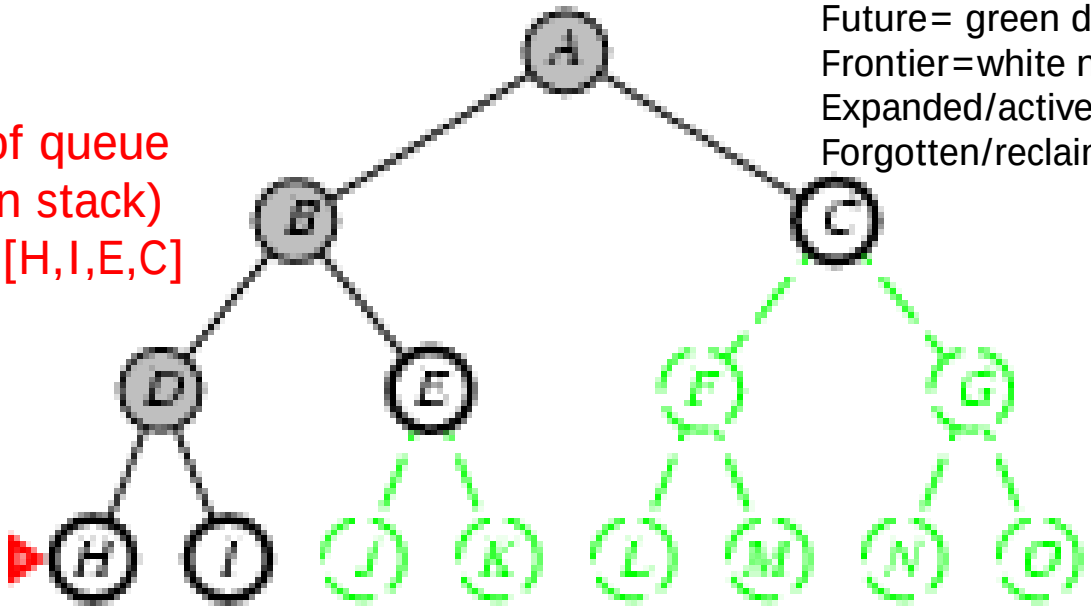
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Is D a goal state? No.
Expand D to H, I.

Put H, I at front of queue
(note: queue is on stack)
queue/frontier = [H,I,E,C]



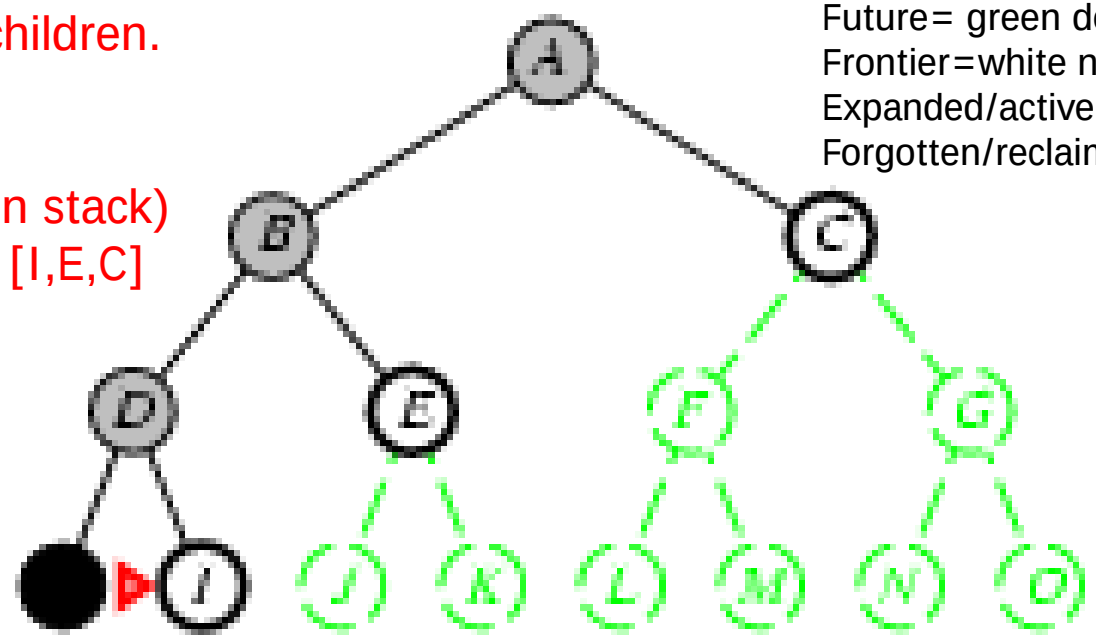
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Is H a goal state? No.
Expand H to no children.
Forget H.

(note: queue is on stack)
queue/frontier = [I,E,C]



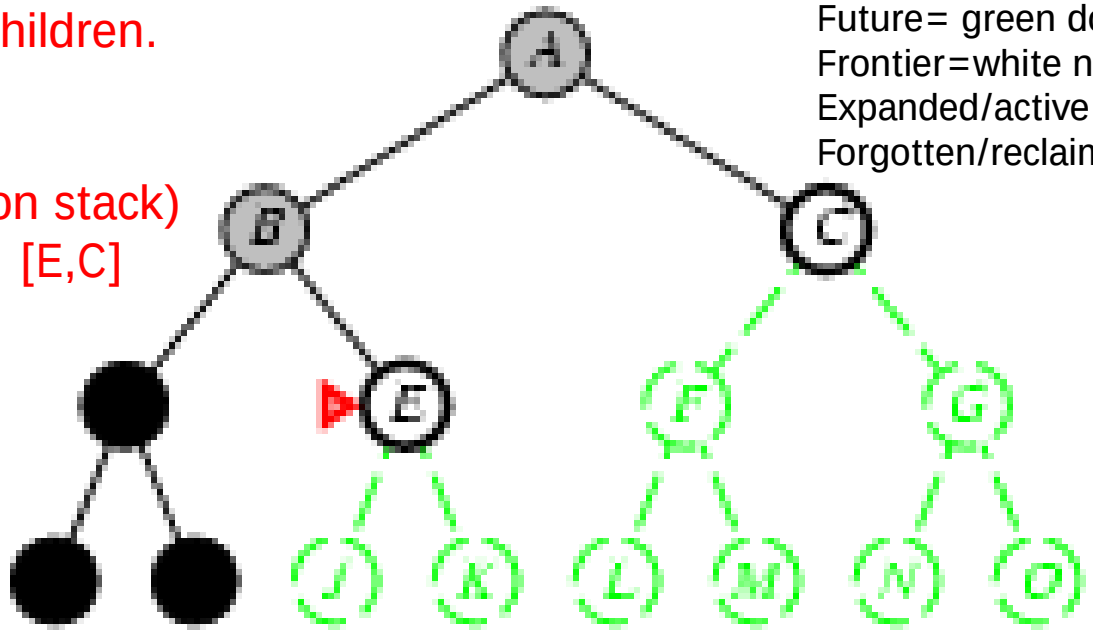
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Is I a goal state? No.
Expand I to no children.
Forget D, I.

(note: queue is on stack)
queue/frontier = [E,C]



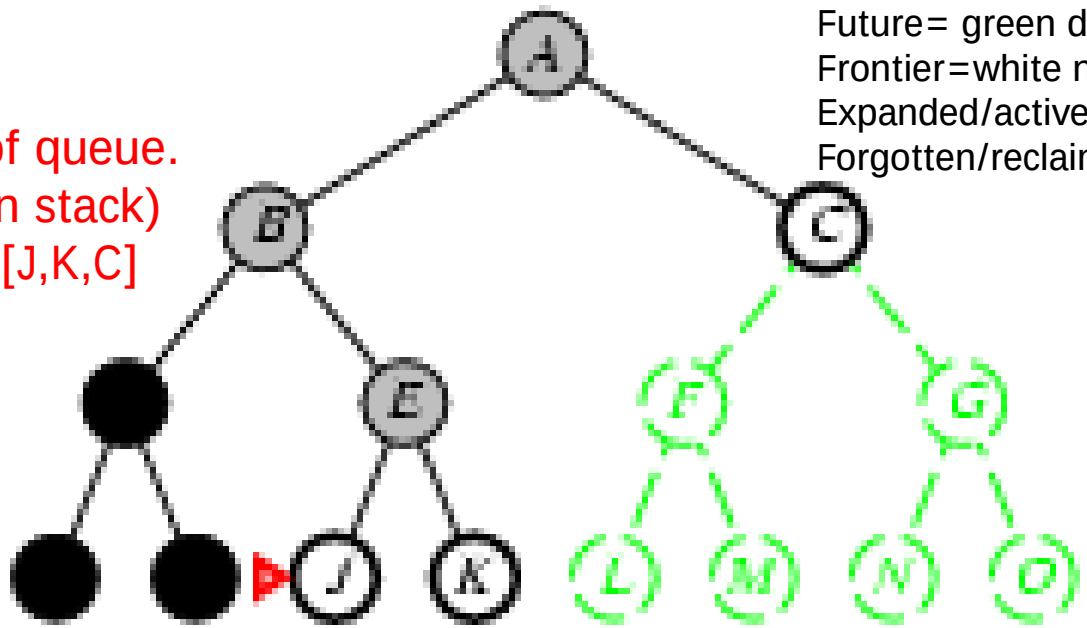
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Is E a goal state? No.
Expand E to J, K.

Put J, K at front of queue.
(note: queue is on stack)
queue/frontier = [J,K,C]



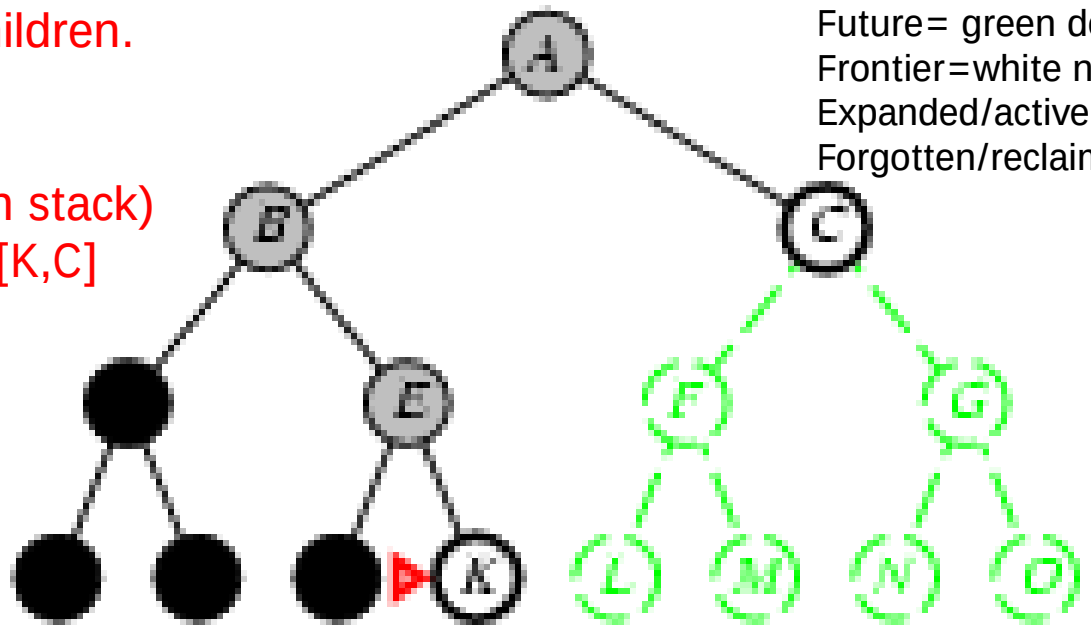
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Is J a goal state? No.
Expand J to no children.
Forget J.

(note: queue is on stack)
queue/frontier = [K,C]



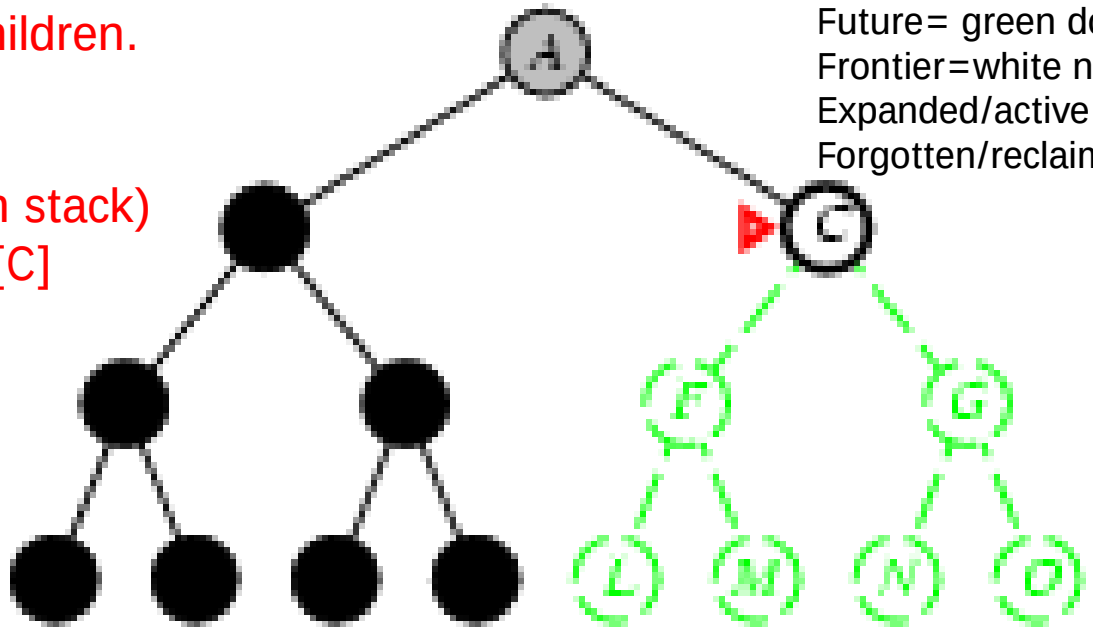
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Is K a goal state? No.
Expand K to no children.
Forget B, E, K.

(note: queue is on stack)
queue/frontier = [C]



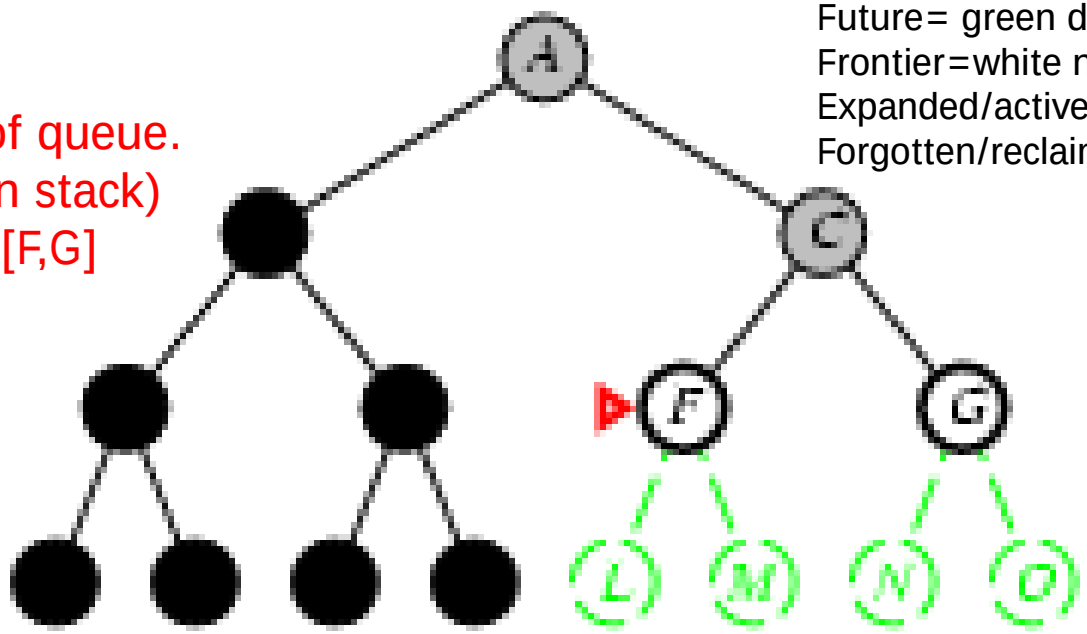
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

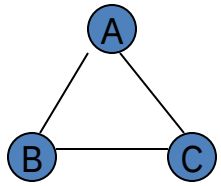
Is C a goal state? No.
Expand C to F, G.

Put F, G at front of queue.
(note: queue is on stack)
queue/frontier = [F,G]



Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Properties of depth-first search



- Complete? No: fails in loops/infinite-depth spaces
  - Can modify to avoid loops/repeated states along path
    - check if current nodes occurred before on path to root
  - Can use graph search (remember all nodes ever seen)
    - problem with graph search: space is exponential, not linear
  - Still fails in infinite-depth spaces (may miss goal entirely)
- Time? $O(b^m)$ with m =maximum depth of space
  - Terrible if m is much larger than d
  - If solutions are dense, may be much faster than BFS
- Space? $O(bm)$, i.e., linear space!
  - Remember a single path + expanded unexplored nodes
- Optimal? No: It may find a non-optimal goal first

Comparing DFS and BFS

- BFS is optimal if path cost is non-decreasing function of depth, DFS is not
- Worst-case Time Complexity: BFS = $O(b^d)$, DFS = $O(b^m)$; m may be infinite
  - In the worst-case, BFS is always better than DFS
- Sometimes, on the average, DFS is better if:
  - Many goals, no loops, and no long or infinite paths
  - Thus, DFS may luckily blunder into an early goal
- BFS is much worse memory-wise
  - BFS may store the whole search space
- DFS can be linear space
  - Stores only the nodes on the path from the current leaf to the root
- In general:
  - BFS is better if shallow goals, many long paths, many loops, small search space
  - DFS is better if many goals, not many loops (easy to check), few long or infinite paths (hard to check), huge search space
  - DFS is always much better in terms of memory

Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both "meet in the middle"
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?
- Complexity
  - time complexity is best: $O(2\ b^{(d/2)}) = O(b^{(d/2)})$
  - memory complexity is the same as time complexity

Bi-Directional Search

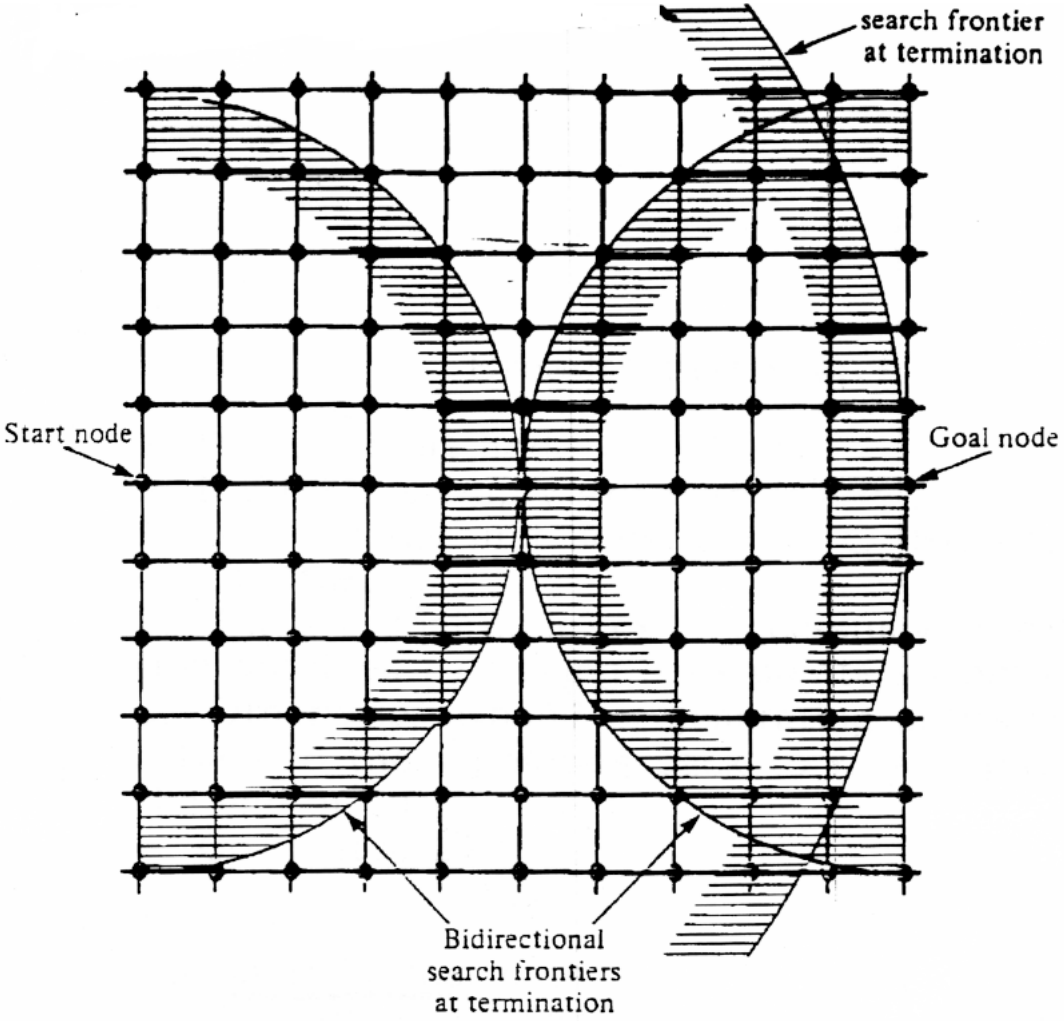


Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

Bidirectional search termination

- R&N Sec. 3.4.6 discusses the BDS termination condition for BFS.
  - To clarify it, and to handle UCS:
- For BFS, the search terminates when one fringe expands a node and discovers that one of the new children is present in the other fringe. This is quick and easy because the other fringe already maintains a hash table holding its fringe, as discussed in the lecture slides about removing duplicate nodes from the fringe, so you just look up the new child in the other fringe's hash table. If present, then you join the path from the Start to that child to the reverse of the path from the Goal to that child, and you have your path from Start to Goal. The first such solution found may not be optimal; some additional search is required to make sure there isn’t a short-cut across the gap.
- For UCS, the same applies, except that afterward you must continue searching until the sum of the costs of the nodes at the head of each queue is greater than or equal to the cost of the path you just found. This continuation guarantees that there is not a longer cheaper path somewhere in the queues. Of course, if you find a cheaper solution as the search winds down, it replaces the previous solution.

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening DLS	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{\lfloor 1+C^*/\varepsilon \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

There are a number of footnotes, caveats, and assumptions.
See Fig. 3.21, p. 91.

Generally the preferred uninformed search strategy

[a] complete if b is finite

[b] complete if step costs $\geq \varepsilon > 0$

[c] optimal if step costs are all identical
(also if path cost non-decreasing function of depth only)

[d] if both directions use breadth-first search
(also if both directions use uniform-cost search with step costs $\geq \varepsilon > 0$)

Note that $d \leq \lfloor 1+C^*/\varepsilon \rfloor$

You should know…

- Overview of uninformed search methods
- Search strategy evaluation
  - Complete? Time? Space? Optimal?
  - Max branching (b), Solution depth (d), Max depth (m)
  - (for UCS: C*: true cost to optimal goal; $\varepsilon > 0$: minimum step cost)
- Search Strategy Components and Considerations
  - Queue? Goal Test when? Tree search vs. Graph search?
- Various blind strategies:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Iterative deepening search (generally preferred)
  - Bidirectional search (preferred if applicable)

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

http://www.cs.rmit.edu.au/AI-Search/Product/
http://aima.cs.berkeley.edu/demos.html (for more demos)