Uninformed (also called blind) search algorithms

This Lecture
Read Chapter 3.1-3.4

Next Lecture
Read Chapter 3.5-3.7

(Please read lecture topic material before and after each lecture on that topic)

You will be expected to 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)

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)

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
  - *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?
- 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*, Fig. 3.7) nodes?
  - Remember them?

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? When generated? When popped?

- 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 inserted.
  - E.g., Breadth-first Search, Depth-first Search, or Uniform Cost search when cost is a non-decreasing function of depth only (which is equivalent to Breadth-first Search).
- REASON ABOUT your search space & problem.
  - How could I possibly find a non-optimal goal?

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 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

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** SOLUTION(*node*)
if STATE[*node*] is not in *closed* **then**
add STATE[*node*] to *closed*
fringe ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)

Goal test after pop

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)

Goal test before push

Figure 3.11 Breadth-first search on a graph.

Uniform cost search: sort by *g*

A* is identical but uses *f*=*g*+*h*

Greedy best-first is identical but uses *h*

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

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.

Depth-limited search & IDS

```
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 before push

```
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
```

When to do Goal-Test? Summary

- For DFS, BFS, DLS, and IDS, the goal test is done when the child node is generated.
  - These are not optimal searches in the general case.
  - BFS and IDS are optimal if cost is a function of depth only; then, optimal goals are also shallowest goals and so will be found first

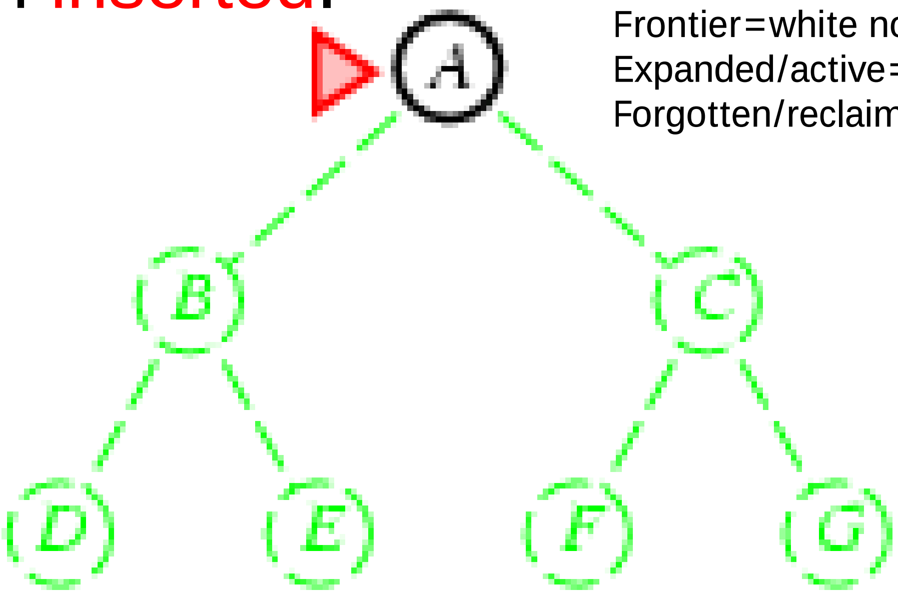
- For GBFS the behavior is the same whether the goal test is done when the node is generated or when it is removed
  - h(goal)=0 so any goal will be at the front of the queue anyway.

- For UCS and A* the goal test is done when the node is removed from the queue.
  - This precaution avoids finding a short expensive path before a long cheap path.

Breadth-first search

- Expand shallowest unexpanded node
- *Frontier* (or fringe): nodes in queue to be explored
- *Frontier* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.
- *Goal-Test* when inserted.

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



Initial state = A
Is A a goal state?

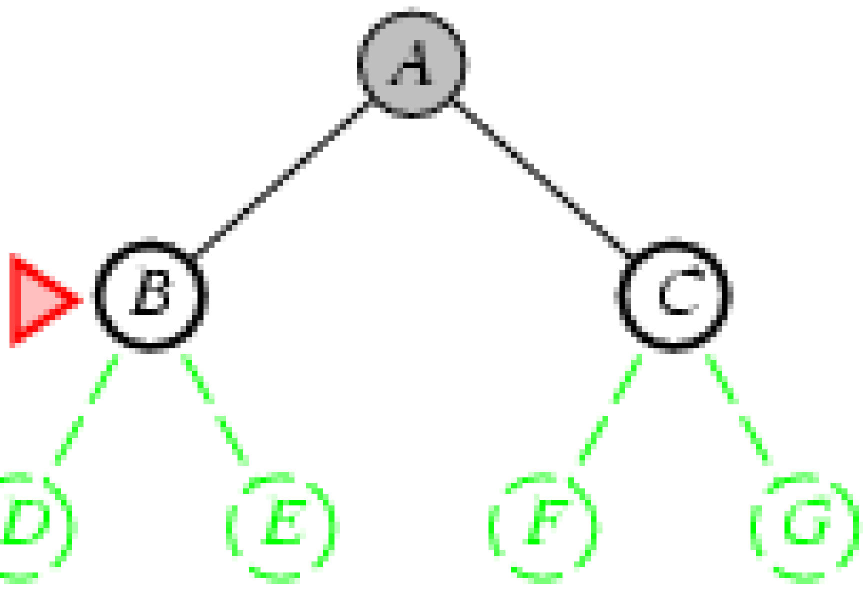
Put A at end of queue.
frontier = [A]

Breadth-first search

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

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

Put B, C at end of queue.
frontier = [B,C]

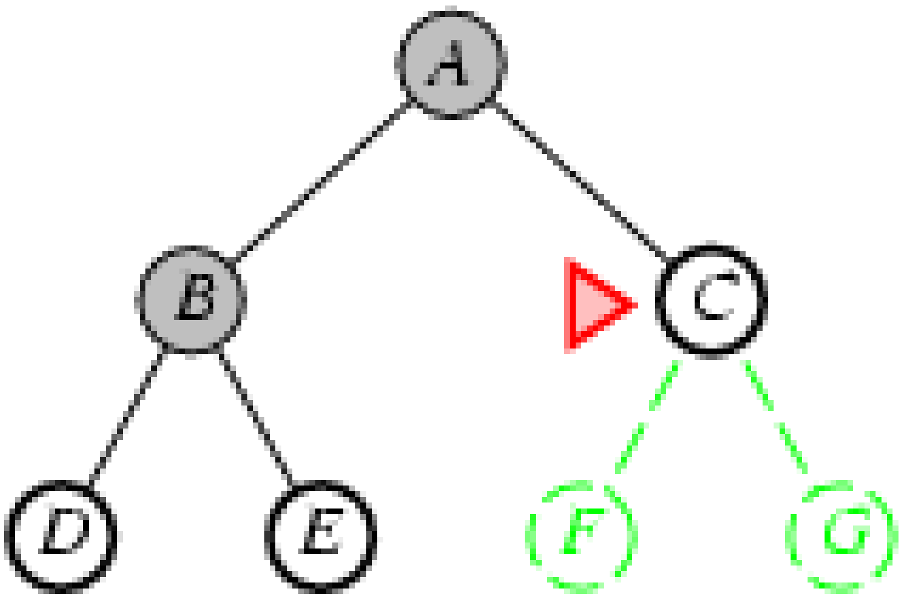


Breadth-first search

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

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

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

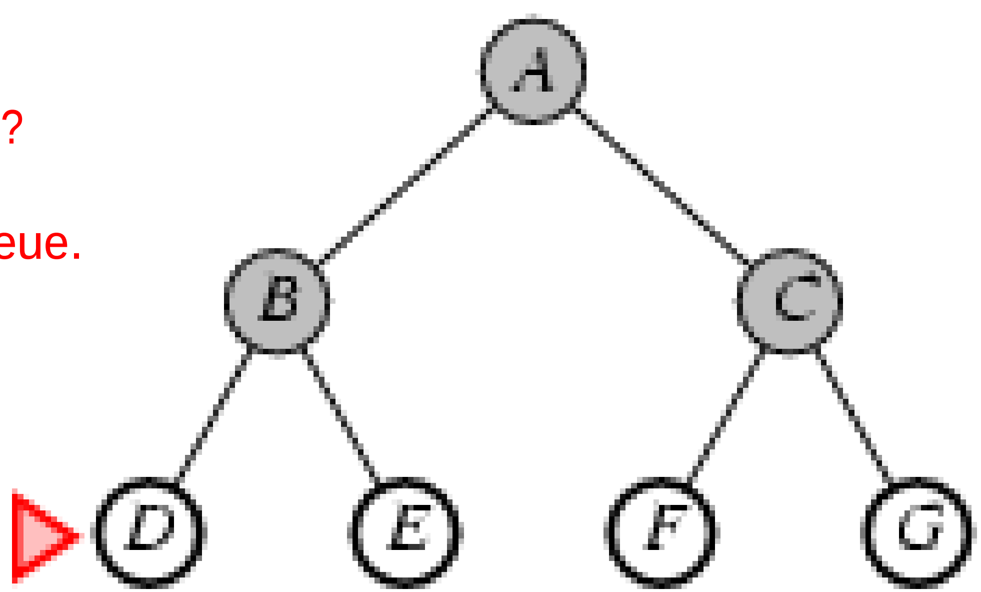


Breadth-first search

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

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

Put F, G at end of queue.
frontier = [D,E,F,G]

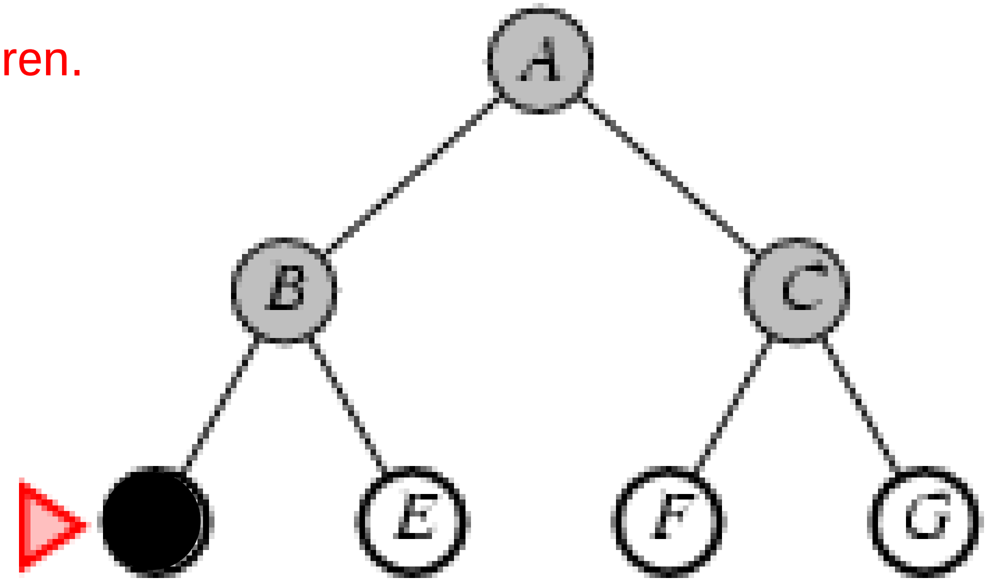


Breadth-first search

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

Expand D to no children.
Forget D.

frontier = [E,F,G]

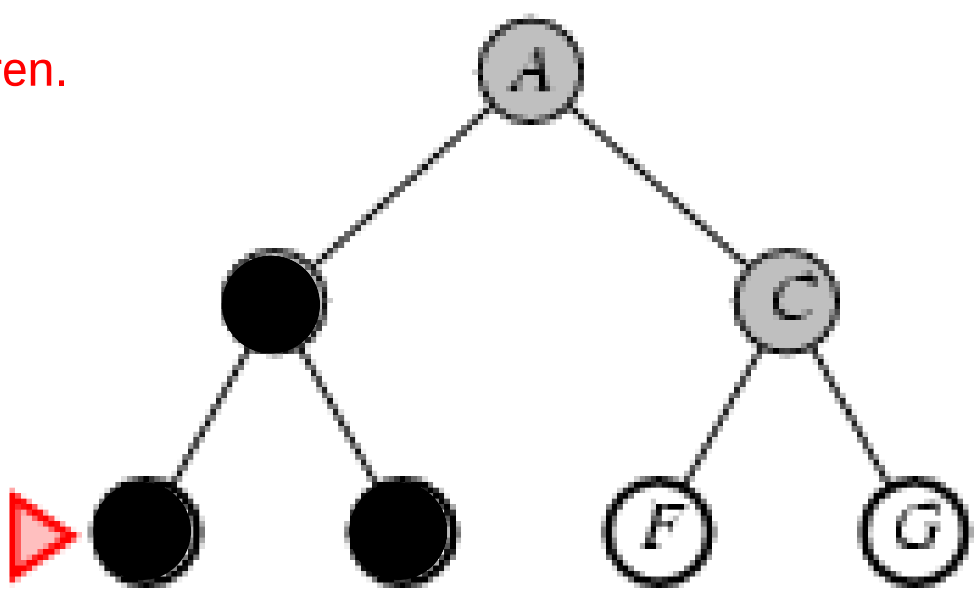


Breadth-first search

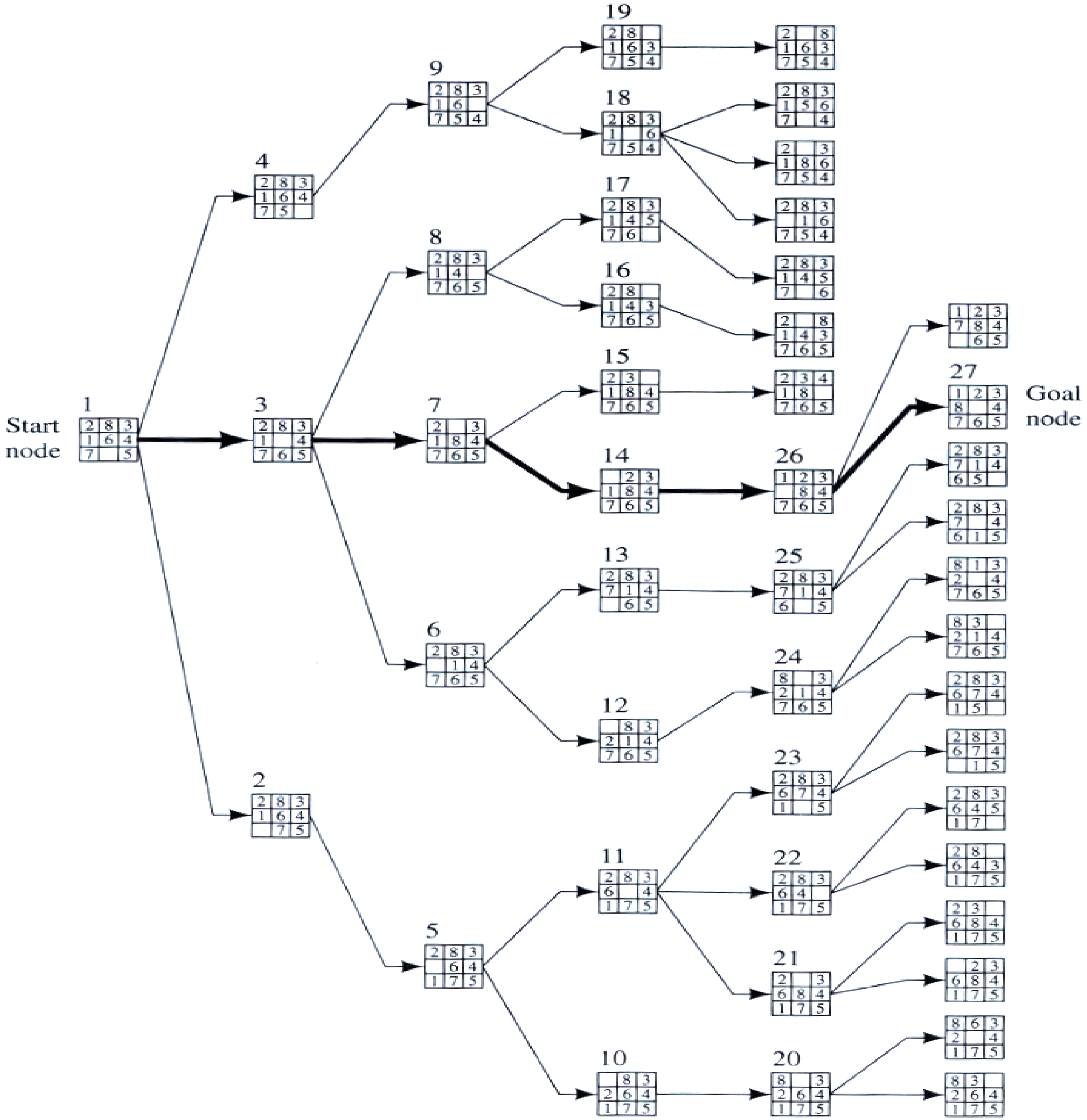
- Expand shallowest unexpanded node
- *Frontier* is a FIFO queue, i.e., new successors go at end

Expand E to no children.
Forget B,E.

frontier = [F,G]



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 fringe or on a path to fringe).
- 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
- Space is the bigger problem (more than time)

Uniform-cost search

Expand node with smallest path cost g(n).

Breadth-first is only optimal if path cost is a non-decreasing function of depth, i.e., $f(d) \geq f(d-1)$; 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

Expand node with smallest path cost g(n).

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 loops)

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

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

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

Uniform-cost search

Expand node with smallest path cost g(n).

Proof of Completeness:

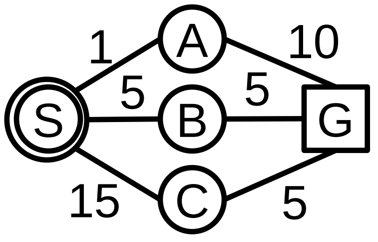
Assume (1) finite max branching factor = *b;* (2) min step cost $\geq \varepsilon > 0$; (3) cost to optimal goal = *C**. Then a node at depth $\lfloor 1+C^*/\varepsilon \rfloor$ must have a path cost > C*. There are O(b^($\lfloor 1+C^*/\varepsilon \rfloor$) such nodes, so a goal will be found.

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.

Uniform-cost search (graph search)

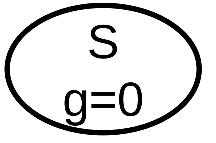
Example hand-simulated search: Search tree method



Order of node expansion: ____________________________

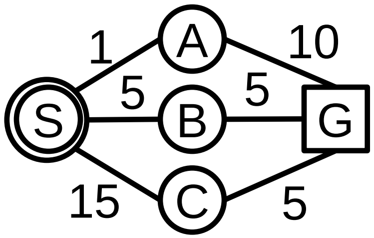
Path found: _______________ Cost of path found: ______

Route finding problem.
Steps labeled w/cost.



Uniform-cost search (graph search)

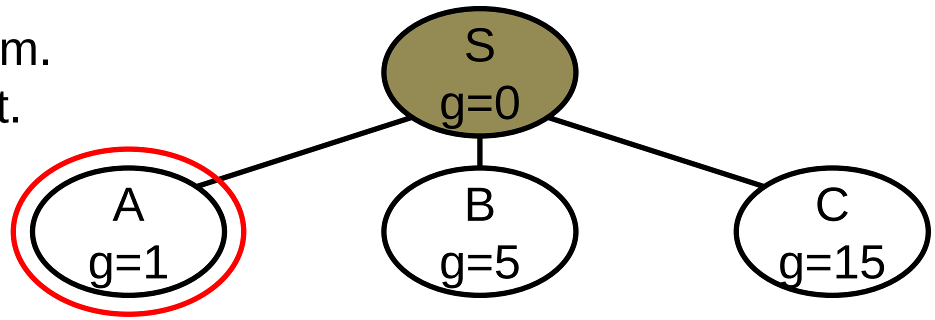
Example hand-simulated search: Search tree method



Route finding problem.
Steps labeled w/cost.

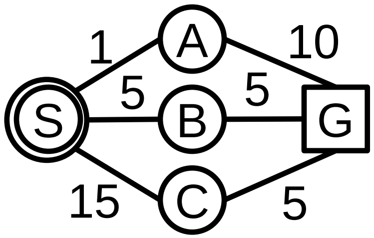
Order of node expansion: S______________________

Path found: _______________ Cost of path found: _______



Uniform-cost search (graph search)

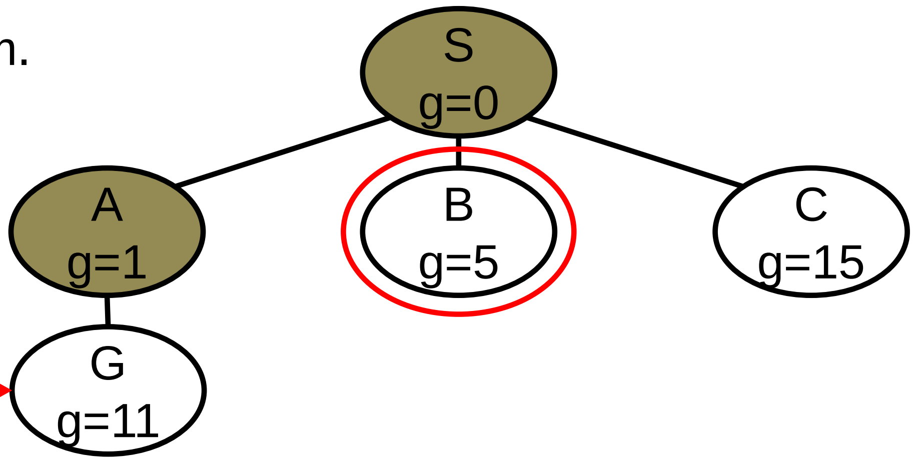
Example hand-simulated search: Search tree method



Order of node expansion: S A ______________________

Path found: ______________ Cost of path found: ______

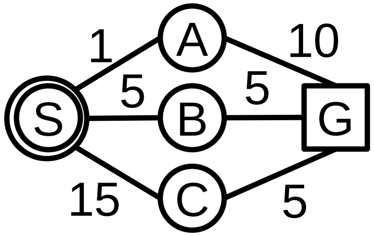
Route finding problem.
Steps labeled w/cost.



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

Uniform-cost search (graph search)

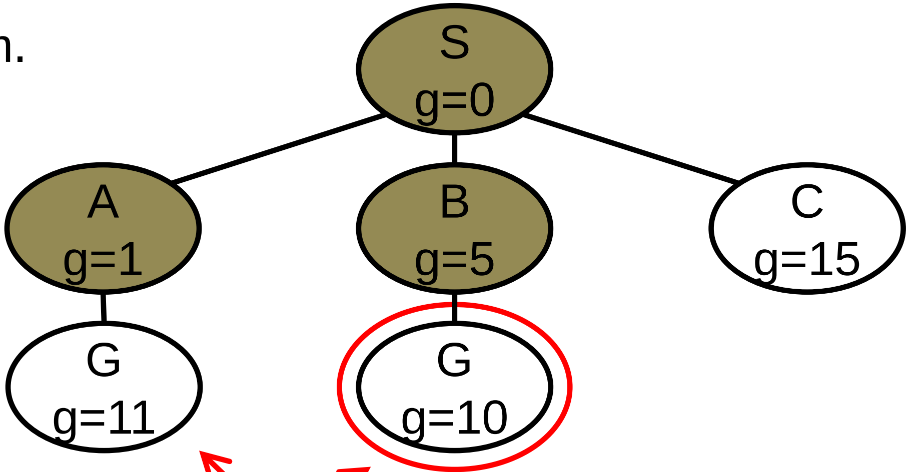
Example hand-simulated search: Search tree method



Order of node expansion: _S A B________________

Path found: _______________ Cost of path found: ______

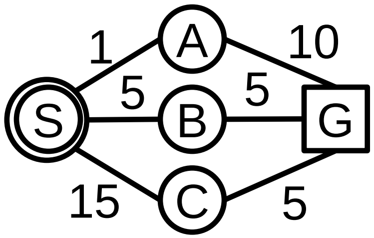
Route finding problem.
Steps labeled w/cost.



If we were doing graph search we would remove the higher-cost of identical nodes and save memory. However, UCS is optimal even with tree search, since lower-cost nodes sort to the front.

Uniform-cost search (graph search)

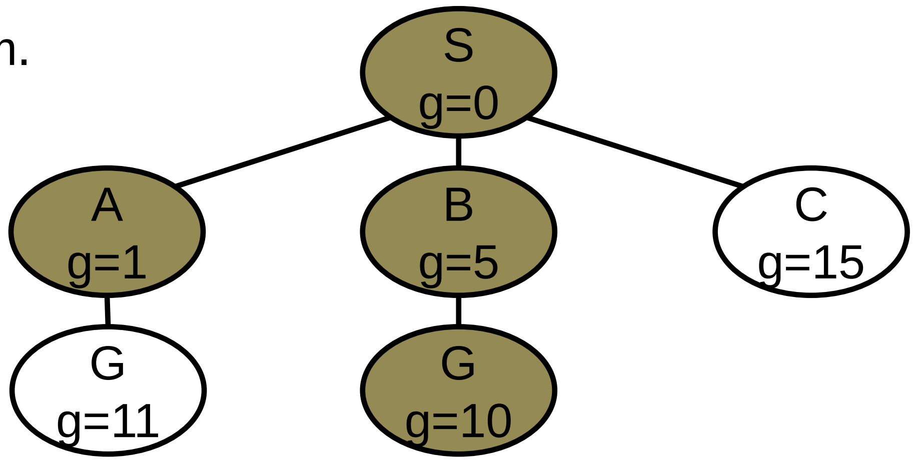
Example hand-simulated search: Search tree method



Route finding problem.
Steps labeled w/cost.

Order of node expansion: S A B G

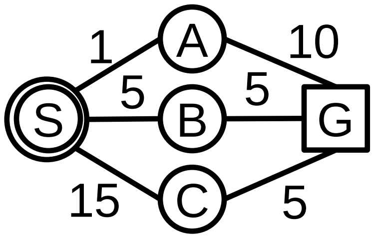
Path found: S B G Cost of path found: 10



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 (graph search)

Example hand-simulated search: Virtual queue method



Order of node expansion: ______________________

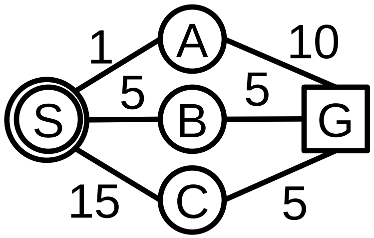
Path found: ____________ Cost of path found: _____

Route finding problem.
Steps labeled w/cost.

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

Uniform-cost search (graph search)

Example hand-simulated search: Virtual queue method



Order of node expansion: _S_______________________

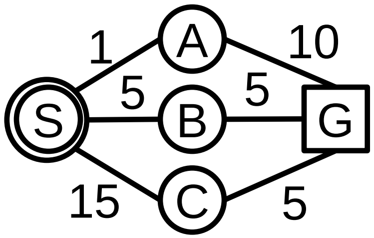
Path found: ______________ Cost of path found: ______

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

Uniform-cost search (graph search)

Example hand-simulated search: Virtual queue method



Route finding problem.
Steps labeled w/cost.

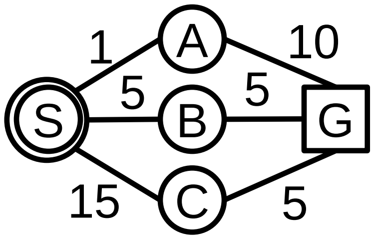
Order of node expansion: S A
Path found: ______________ Cost of path found: ______

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.

Uniform-cost search (graph search)

Example hand-simulated search: Virtual queue method



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

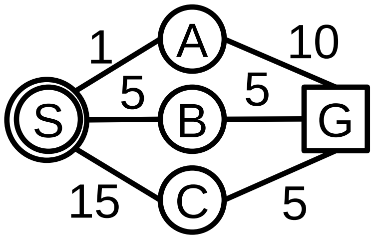
Next: B/g=5

Children: G/g=10

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

Uniform-cost search (graph search)

Example hand-simulated search: Virtual queue method



Order of node expansion: S A B G

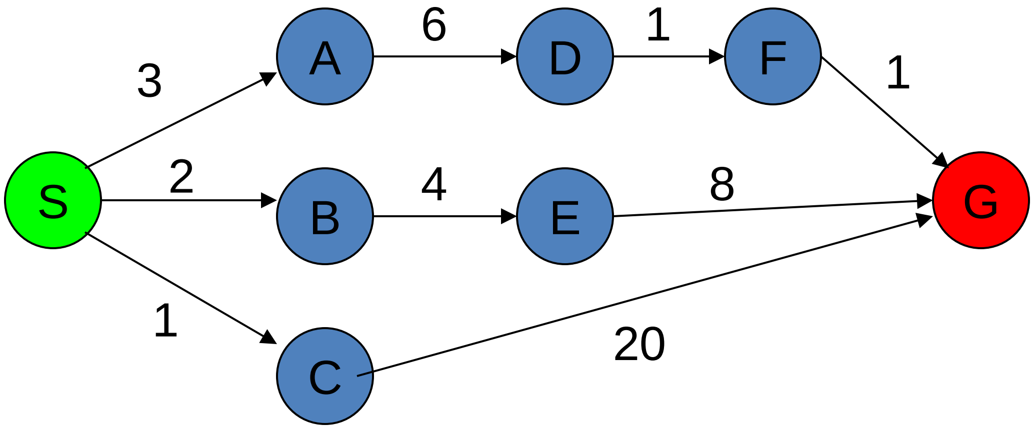
Path found: S B G Cost of path found: 10

Route finding problem.
Steps labeled w/cost.

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.



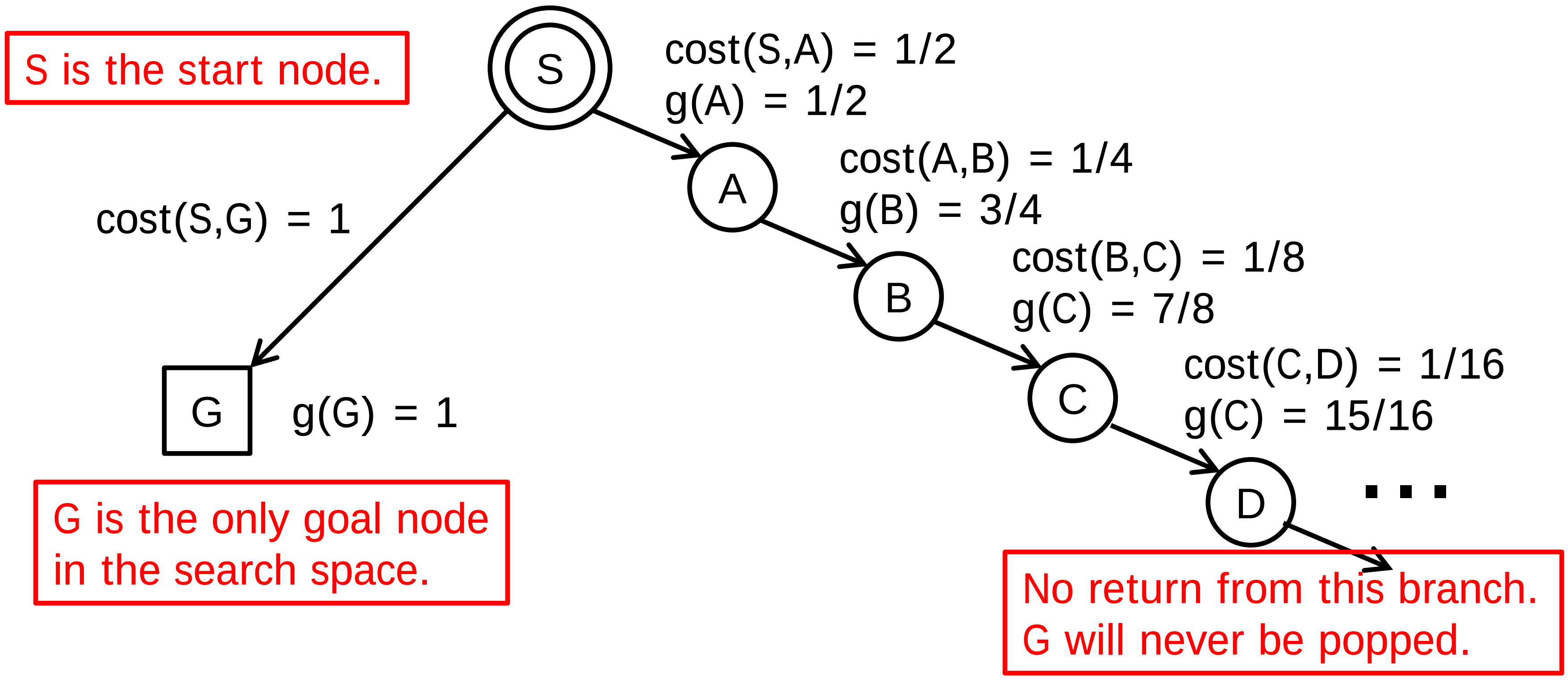
The graph above shows the step-costs for different paths going from the start (S) to the goal (G).

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

Exercise for at home

Uniform-cost search
Why require step cost ≥ ε > 0?

- Otherwise, an infinite regress is possible.
- Consider that $\Sigma_{n \text{ from } 1 \text{ to } \infty} 2^{-n} = 1$.



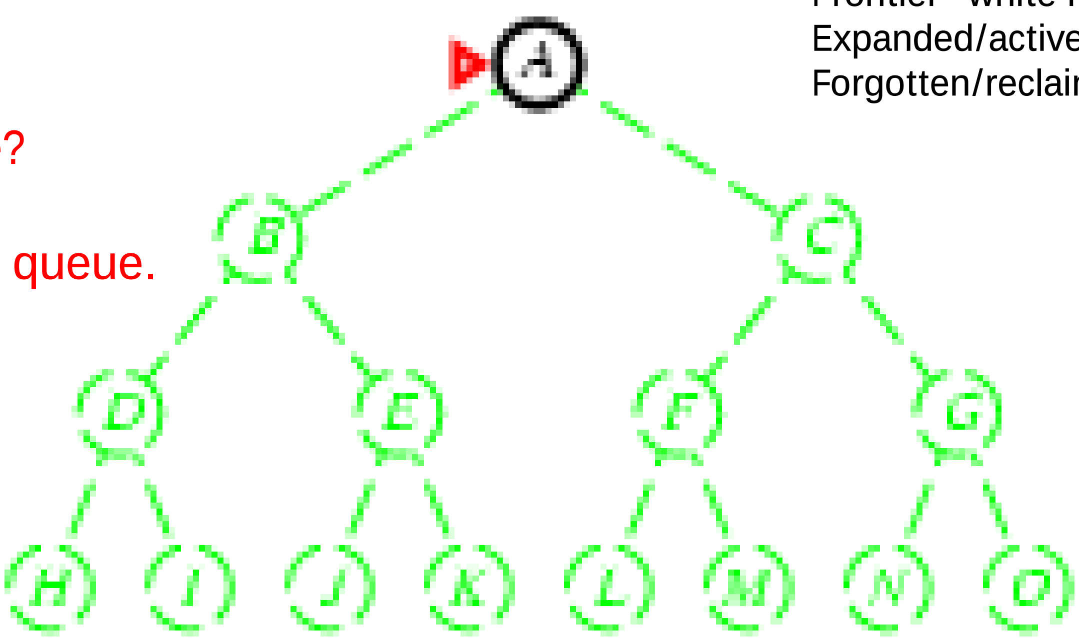
Depth-first search

- Expand *deepest* unexpanded node
- *Frontier* = Last In First Out (LIFO) queue, i.e., new successors go at the front of the queue.
- *Goal-Test* when inserted.

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

Initial state = A
Is A a goal state?

Put A at front of queue.
frontier = [A]

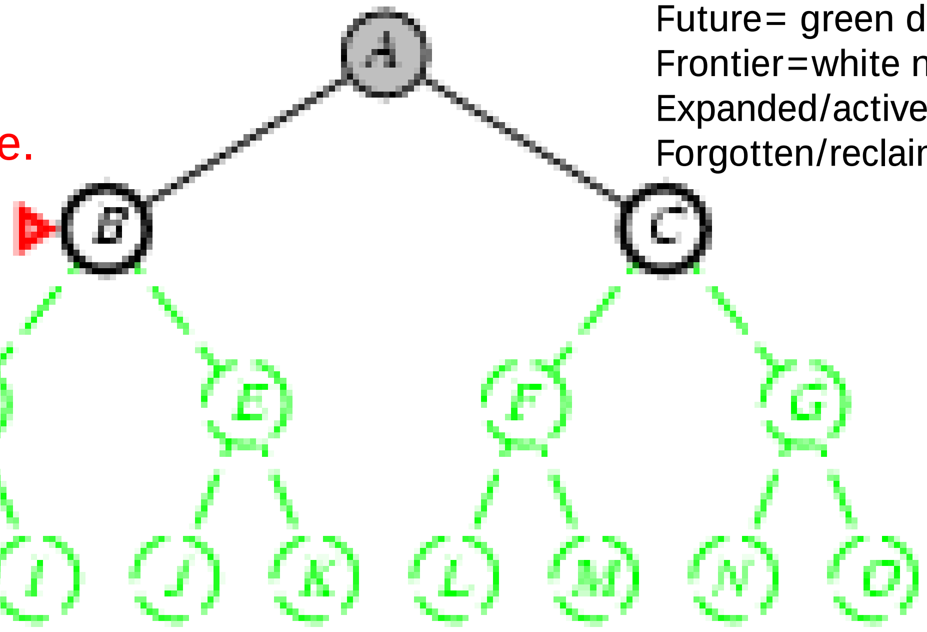


Depth-first search

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

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

Put B, C at front of queue.
frontier = [B,C]



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

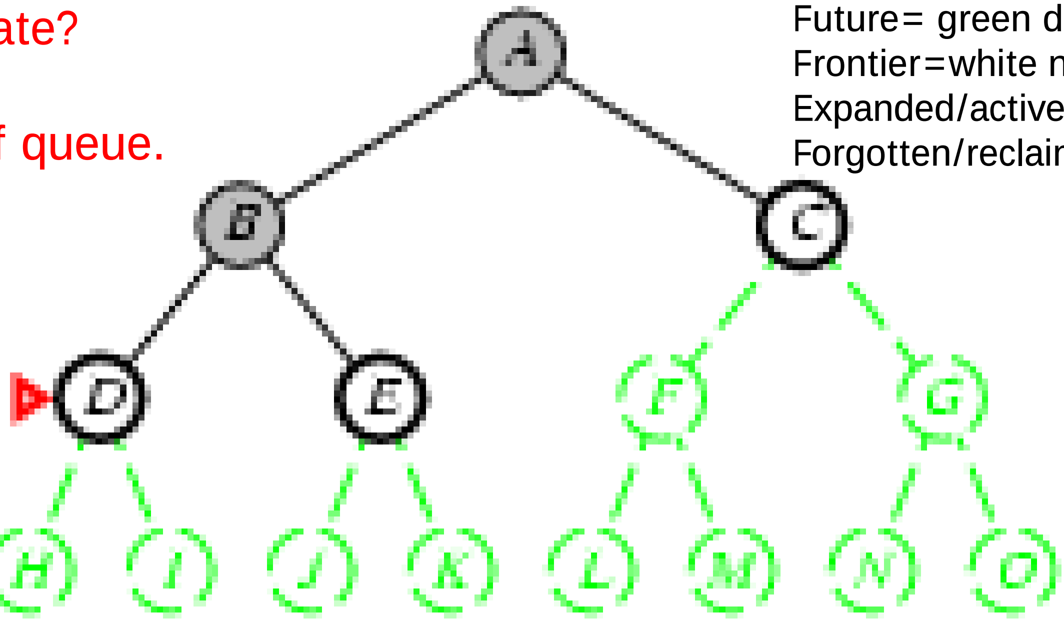
Note: Can save a space factor of *b* by generating successors one at a time.
See **backtracking search** in your book, p. 87 and Chapter 6.

Depth-first search

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

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

Put D, E at front of queue.
frontier = [D,E,C]



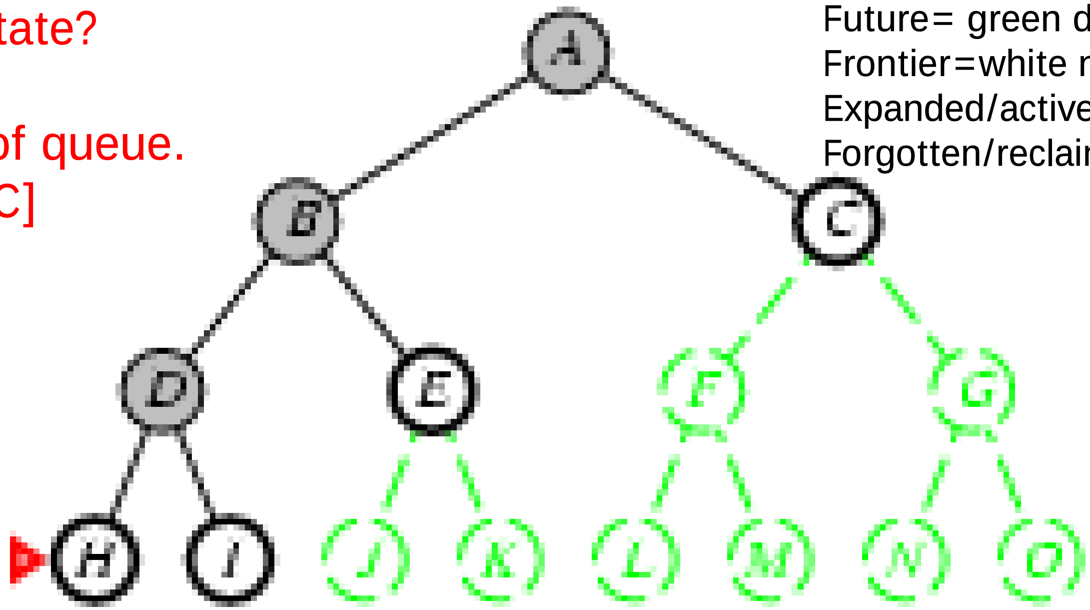
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Expand D to H, I.
Is H or I a goal state?

Put H, I at front of queue.
frontier = [H,I,E,C]



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

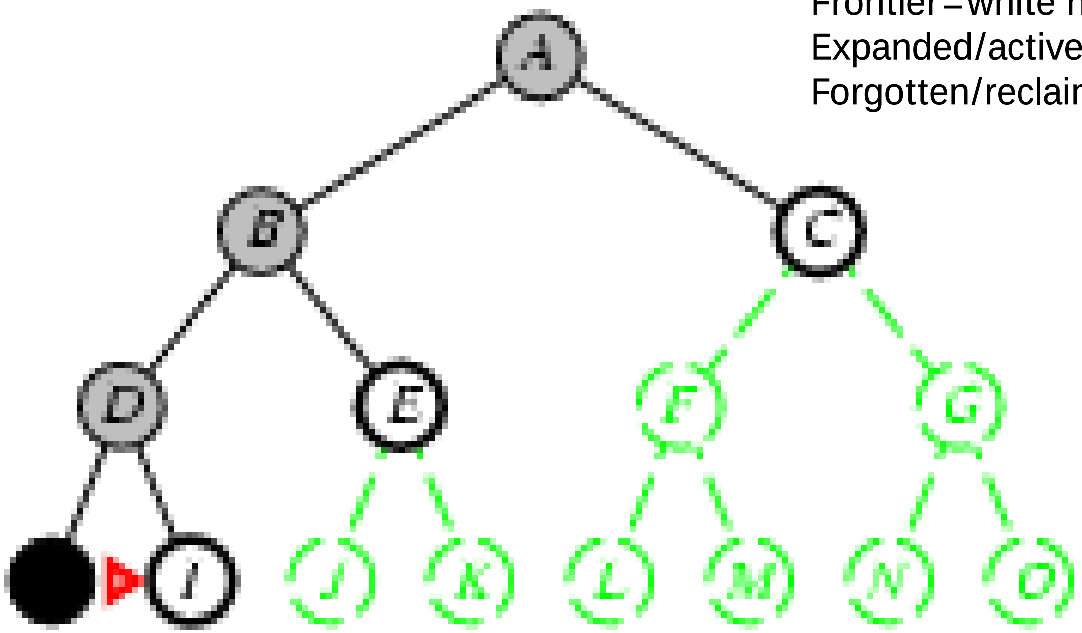
Depth-first search

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

Expand H to no children.
Forget H.

frontier = [I,E,C]

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

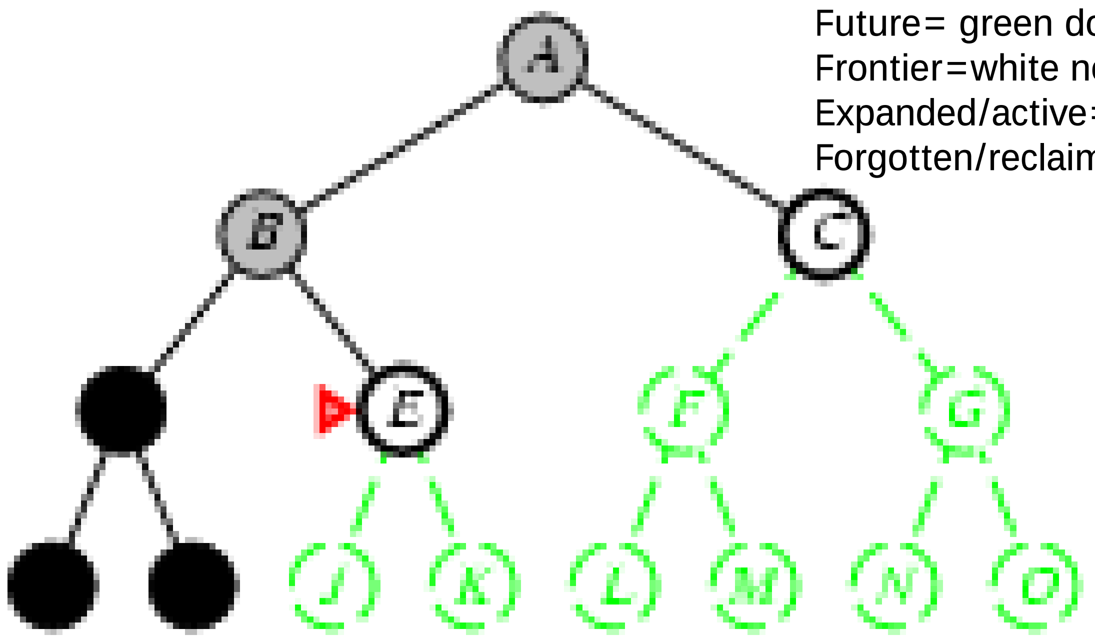


Depth-first search

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

Expand I to no children.
Forget D, I.

frontier = [E,C]



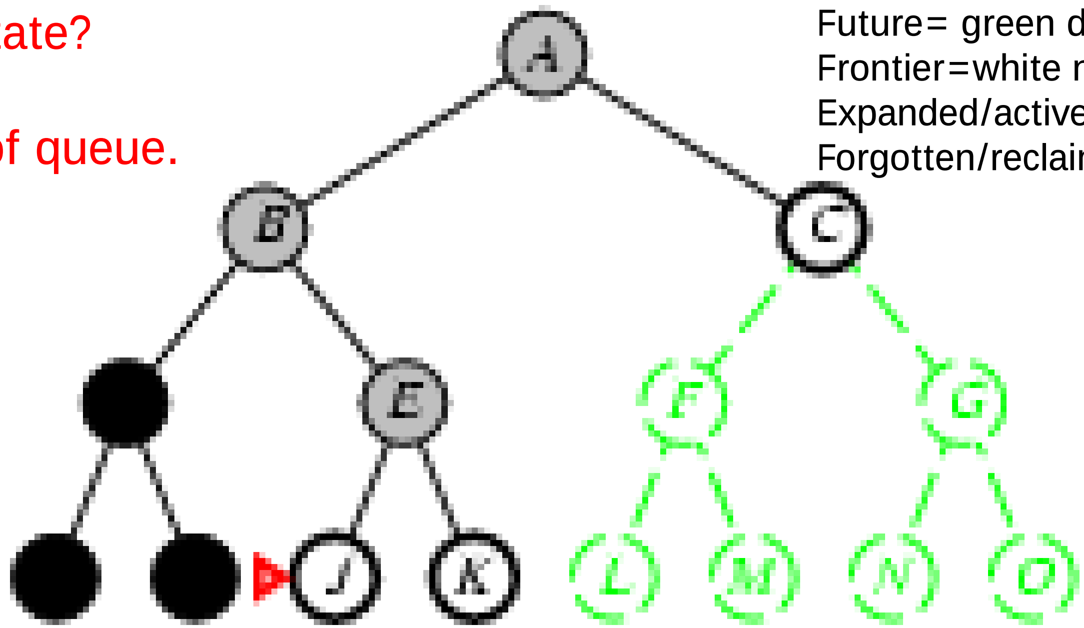
Future= green dotted circles
Frontier=white nodes
Expanded/active=gray nodes
Forgotten/reclaimed= black nodes

Depth-first search

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

Expand E to J, K.
Is J or K a goal state?

Put J, K at front of queue.
frontier = [J,K,C]



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

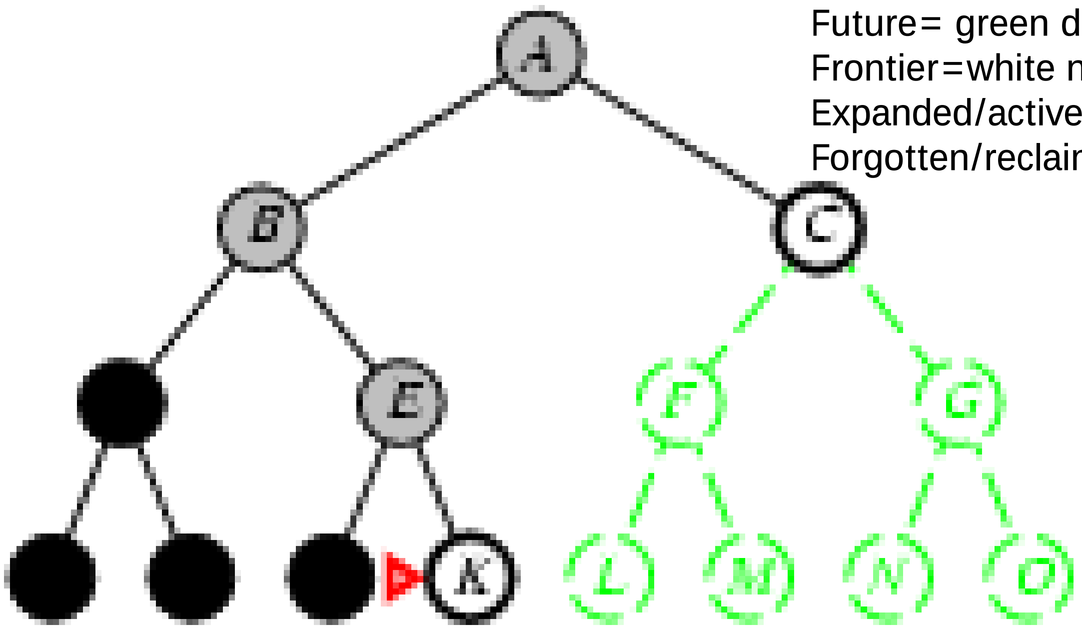
Depth-first search

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

Expand J to no children.
Forget J.

frontier = [K,C]

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



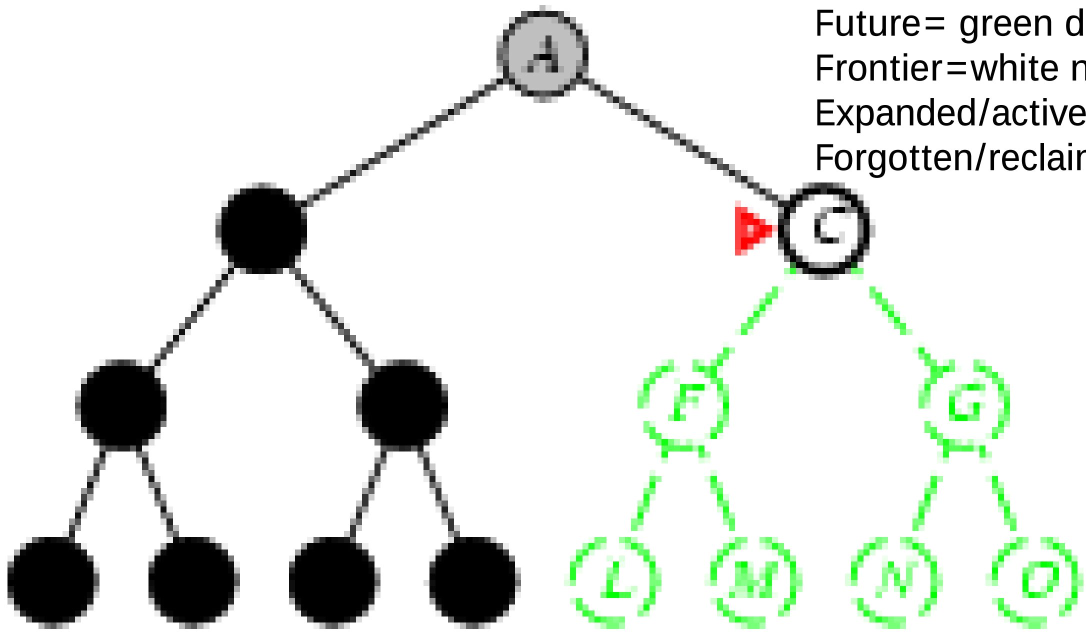
Depth-first search

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

Expand K to no children.
Forget B, E, K.

frontier = [C]

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

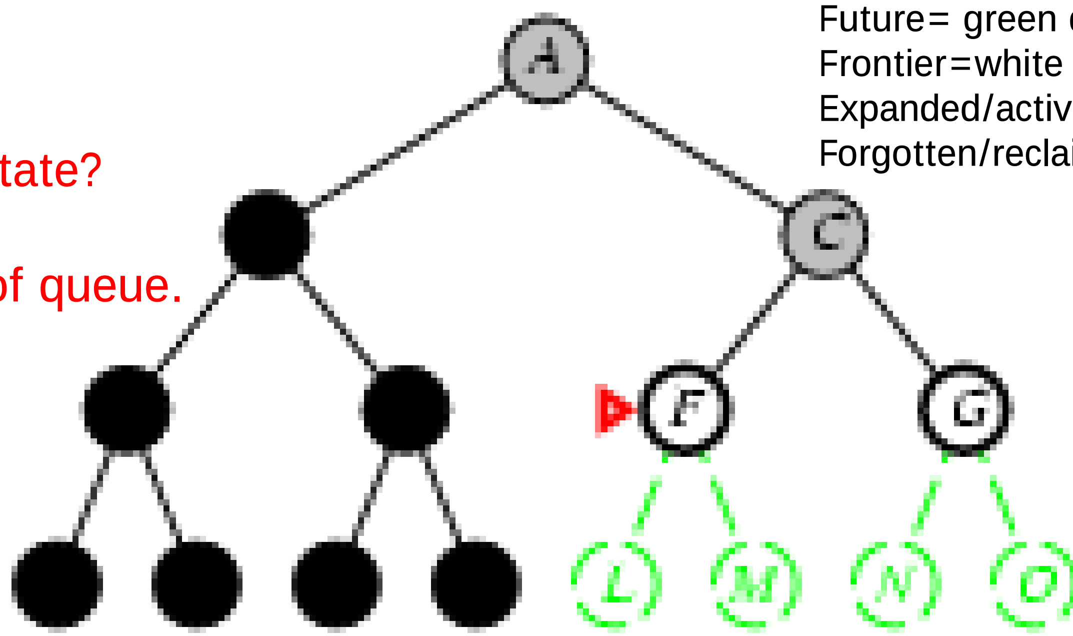


Depth-first search

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

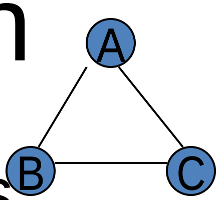
Expand C to F, G.
Is F or G a goal state?

Put F, G at front of 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

Iterative deepening search

- To avoid the infinite depth problem of DFS,
  only search until depth L,
  i.e., we don't expand nodes beyond depth L.
  → Depth-Limited Search (DLS)

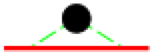
- What if solution is deeper than L? → Increase L iteratively.
  → Iterative Deepening Search (IDS)

- IDS inherits the memory advantage of Depth-first search

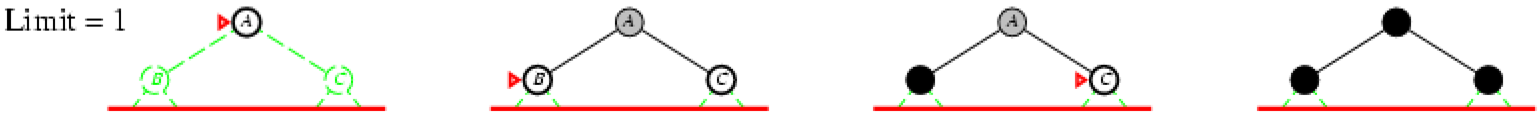
- IDS has the completeness property of Breadth-first search.

Iterative deepening search L=0

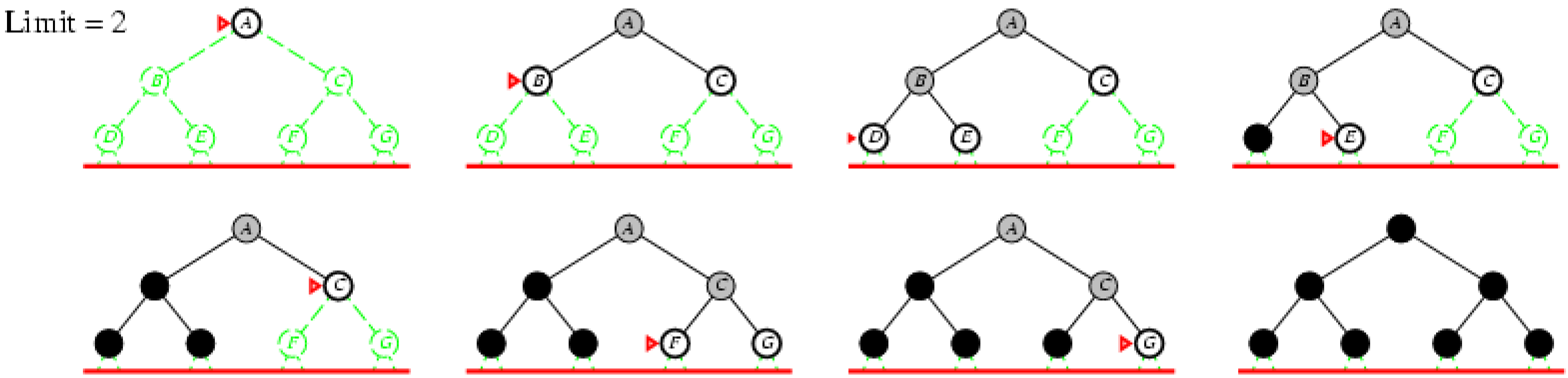
Limit = 0



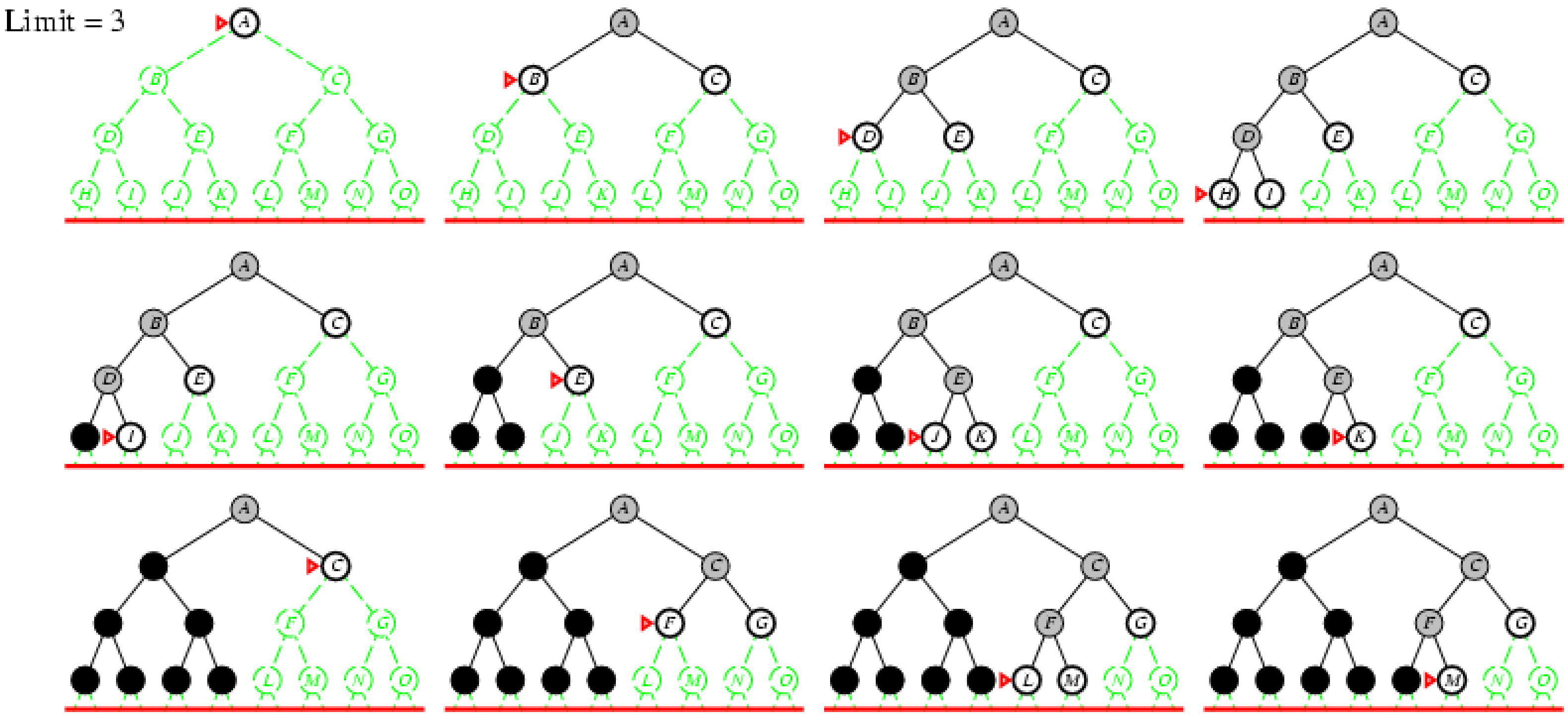
Iterative deepening search L=1



Iterative deepening search L=2



Iterative Deepening Search L=3



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

$O(b^d)$

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.

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?
  - which to take if there are multiple goal states?
  - where to start if there is only a goal test, no explicit list?

Bi-Directional Search

Complexity: time and space complexity are: $O(b^{d/2})$

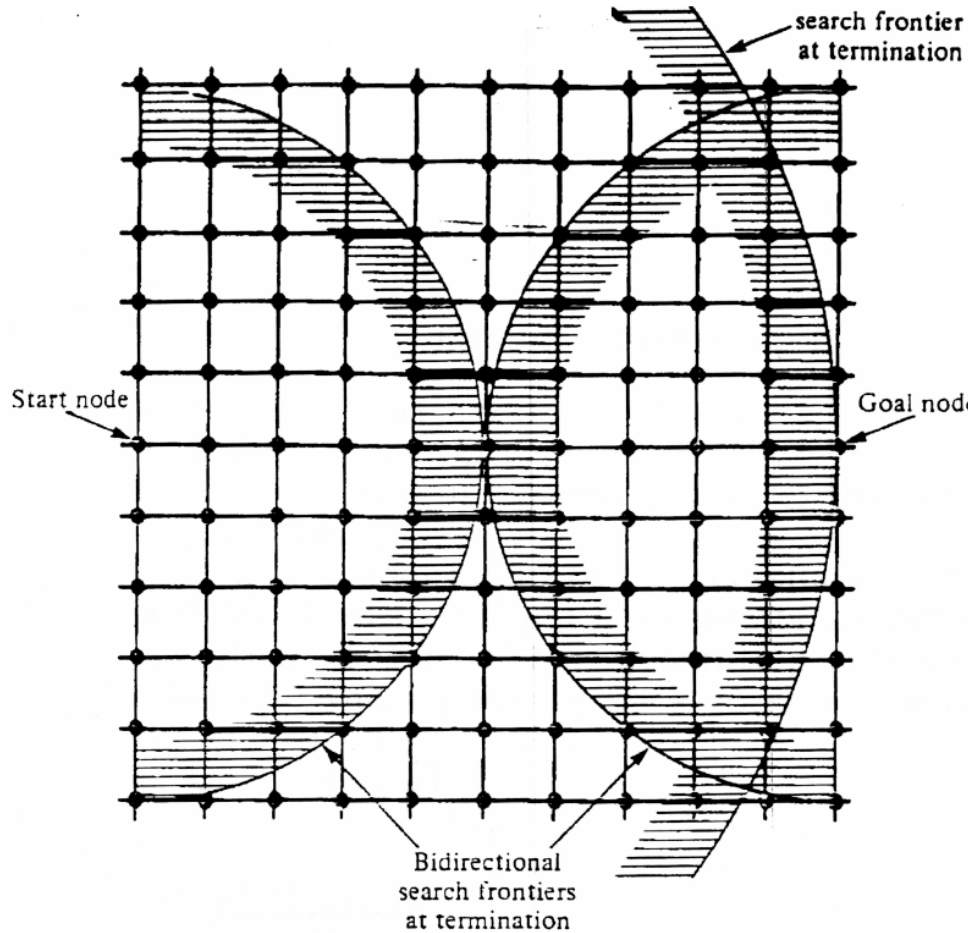


Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

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]

Generally the preferred uninformed search strategy

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

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

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)