Comparing DFS and BFS

- Same worst-case time Complexity, but
  - In the worst-case BFS is always better than DFS (because depth d nodes are expanded before d+1 nodes).
  - Sometime, on the average DFS is better if:
    - many goals, no loops and no infinite paths
- BFS is much worse memory-wise
    - DFS is linear space (linear in depth d and branching factor m)
    - BFS may store the whole search space.
- In general
    - BFS is better if goal is not deep, if infinite paths, if many loops, if small search space
    - DFS is better if many goals, not many loops
    - DFS is much better in terms of memory

Depth-limited DFS

```
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
```

Figure 3.17 A recursive implementation of depth-limited tree search.

Iterative Deepening (DFS)

- Every iteration is a DFS with a depth cutoff.

Iterative deepening (ID)

1. i = 1
2. While no solution do

   DFS from initial state S_0 with cutoff I

   If found goal, stop and return solution

   else, increment cutoff

Comments:

- ID implements BFS with DFS
- Only one path in memory
- So it combines the better features of BFS and DFS at a slightly higher cost than

Iterative deepening search

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

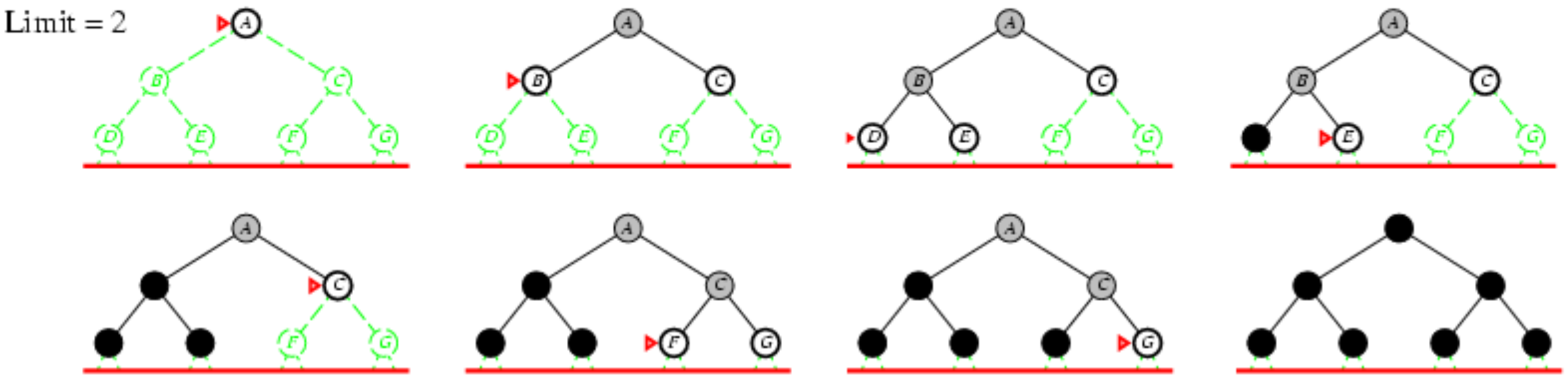
Figure 3.18 The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

Iterative deepening search L=0

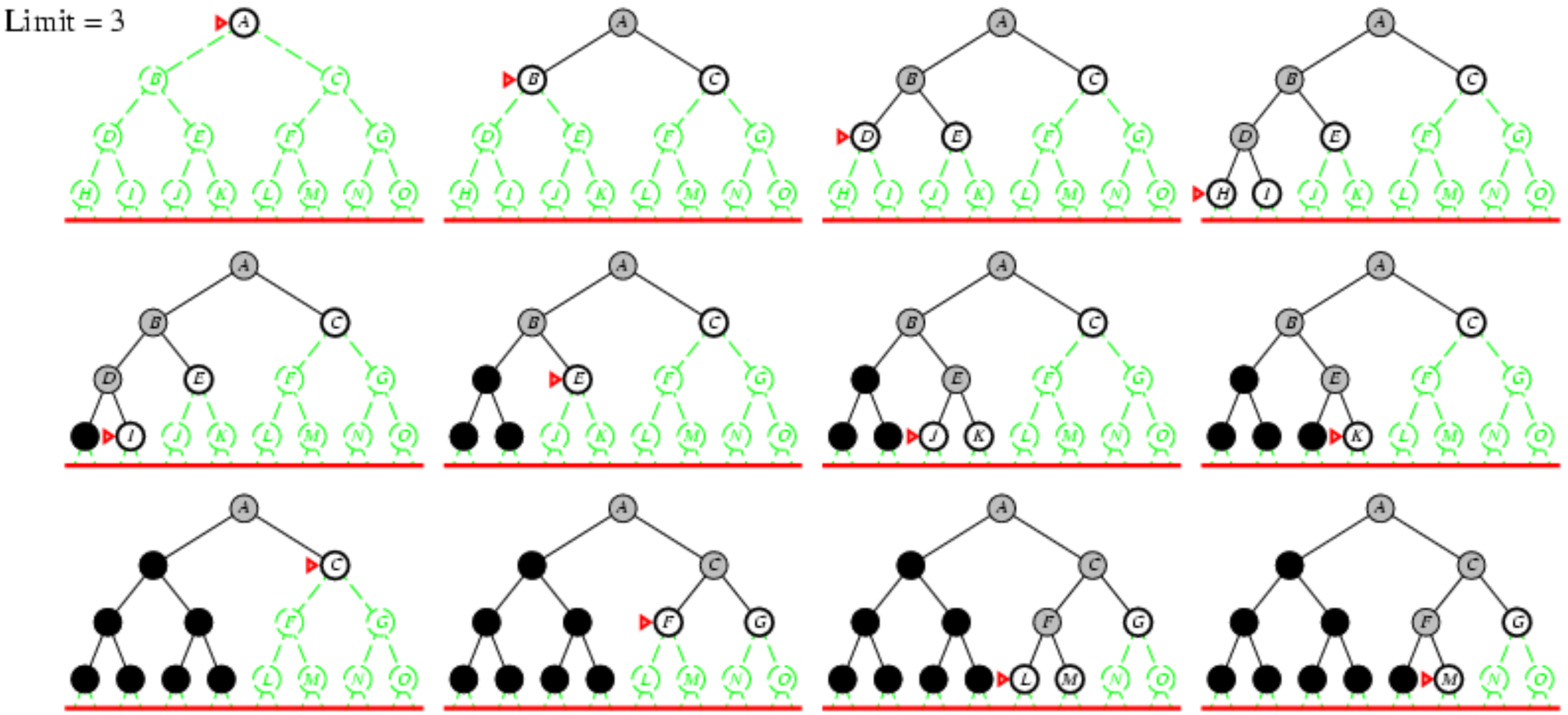
Limit = 0

Iterative deepening search *L*=1

Iterative deepening search L=2



Iterative Deepening Search L=3



Iterative deepening search

Properties of iterative deepening search

- Complete? Yes
- Time? $O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1 or increasing function of depth.

Iterative Deepening Time (DFS)

- Time:

  - BFS time is $O(b^n)$
  - b is the branching degree
  - ID is asymptotically like BFS
  - For b=10 d=5 d=cut-off
  - DFS = 1+10+100,…,=111,111
  - IDS = 123,456

Comments on Iterative Deepening Search

- Complexity
  - Space complexity = O(bd)
    - (since its like depth first search run different times)
  - Time Complexity
    - $1 + (1+b) + (1 + b+b^2) + \ldots\ldots.(1 + b+\ldots.b^d)$
    - $= O(b^d)$
      (i.e., asymptotically the same as BFS or DFS in the worst case)

    - The overhead in repeated searching of the same subtrees is small relative to the overall time
      - e.g., for b=10, only takes about 11% more time than BFS

- A useful practical method
  - combines
    - guarantee of finding an optimal solution if one exists (as in BFS)
    - space efficiency, O(bd) of DFS
    - But still has problems with loops like DFS

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
    - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?
- Complexity
  - time complexity is best: $O(2\ b^{(d/2)}) = O(b^{(d/2)})$, worst: $O(b^{d+1})$
  - memory complexity is the same

Weighted edge case: Uniform Cost Search

- Expand lowest-cost OPEN node ($g(n)$)
- g(n) = weight of the path from start to current node
- In BFS $g(n) = depth(n)$

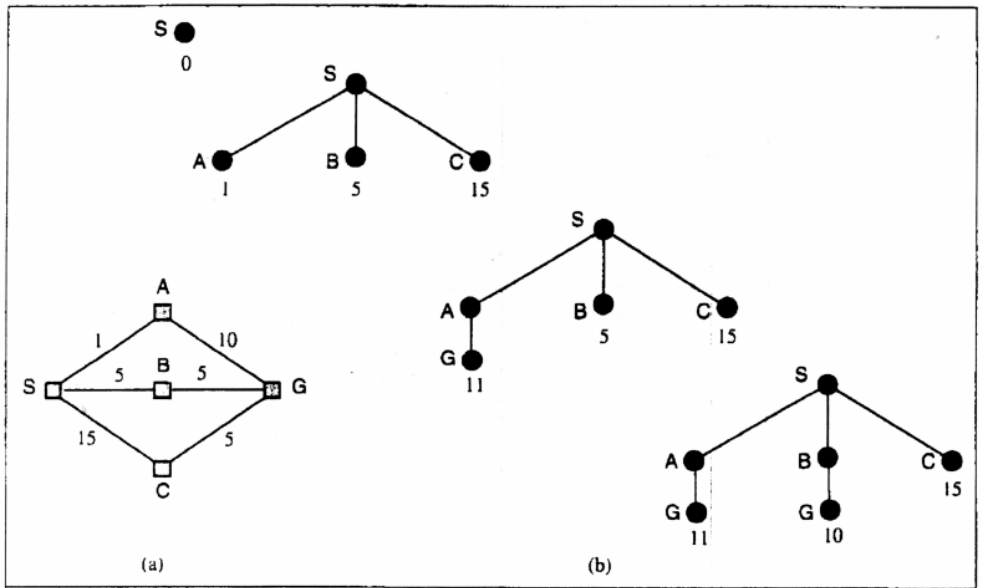


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

- Requirement
  - $g(\text{successor}(n)) \geq g(n)$

Uniform cost search – Tree version

1. Put the start node *s* on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node *n* from OPEN and place it on CLOSED.
4. If *n* is a goal node, exit successfully with the solution obtained by tracing back pointers from *n* to *s*.
5. Otherwise, <u>expand *n*</u>, generating all its successors attach to them pointers back to *n*, and put them at the *end* of OPEN ***in order of shortest cost from the root node***
6. Go to step 2.

Uniform cost search – Graph Version

1. Put the start node *s* on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node *n* from OPEN and place it on CLOSED.
4. If *n* is a goal node, exit successfully with the solution obtained by tracing back pointers from *n* to *s*.
5. Otherwise, expand *n*, generating all its successors not in CLOSED set, attach to them pointers back to *n*, and put them at the *end* of OPEN ***in order of shortest cost from the root node***
6. Go to step 2.

Uniform-cost search

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

Complete? Yes, if step cost ≥ ε
(otherwise it can get stuck in infinite loops)

Time? # of nodes with *path cost* ≤ cost of optimal solution.
(This is the number of nodes expanded)

Space? # of nodes on paths with path cost ≤ cost of optimal solution.

Optimal? Yes, for any step cost.

Comparison of Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Figure 3.18 Evaluation of search strategies. b is the branching factor; d is the depth of solution; m is the maximum depth of the search tree; l is the depth limit.

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

Summary

- A review of search
  - a search space consists of states and operators: it is a graph
  - a search tree represents a particular exploration of search space

- There are various strategies for "uninformed search"
  - breadth-first
  - depth-first
  - iterative deepening
  - bidirectional search
  - Uniform cost search
  - Depth-first branch and bound

- Repeated states can lead to infinitely large search trees
  - we looked at methods for for detecting repeated states

- All of the search techniques so far are "blind" in that they do not look at how far away the goal may be: next we will look at informed or heuristic search, which directly tries to minimize the distance to the goal. Example we saw: greedy search