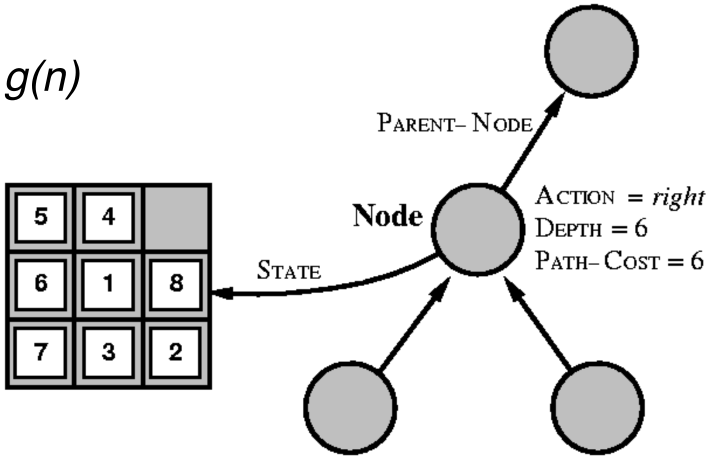
Infrastructure for search algorithms

- Data structures for search trees
  - Many ways to represent nodes, here:
  - Node has 5 components
    - STATE: State in the state space to which the node corresponds
    - PARENT- NODE: the node in the search tree that generated this node
    - ACTION: the action that was applied to the parent to generate the node
    - PATH-COST : the cost of the path from the initial state to the node, as indicated by the parent pointers. Traditionally denoted by *g(n)*
    - DEPTH: the number of steps along the path from the initial state

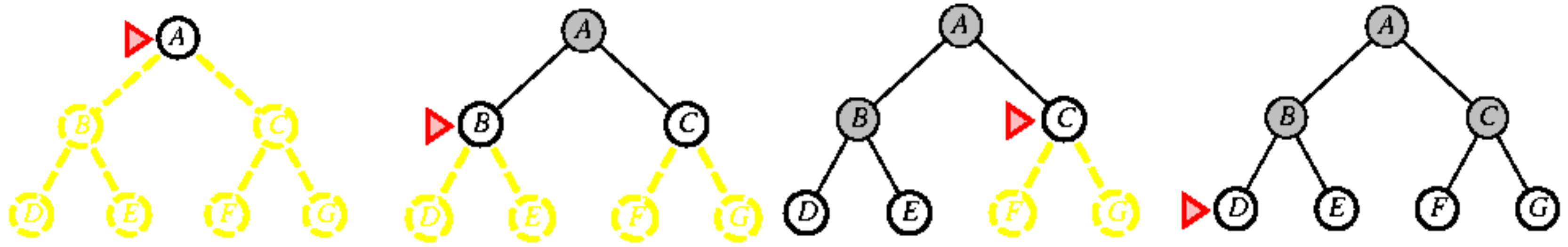


Measuring problem-solving performance

- Lots of work has been invested to find good search strategies. Evaluation criteria:
  - **Completeness**: is the algorithm guaranteed to find a solution when there is one?
  - **Optimality**: does the strategy find the optimal solution?
  - **Time complexity**: how long does it take to find a solution?
  - **Space complexity**: how much memory does it need to perform the search?
  - Optimal solution?
    - has the lowest path cost among all solutions
- Following: 6 search strategies known as uninformed search.

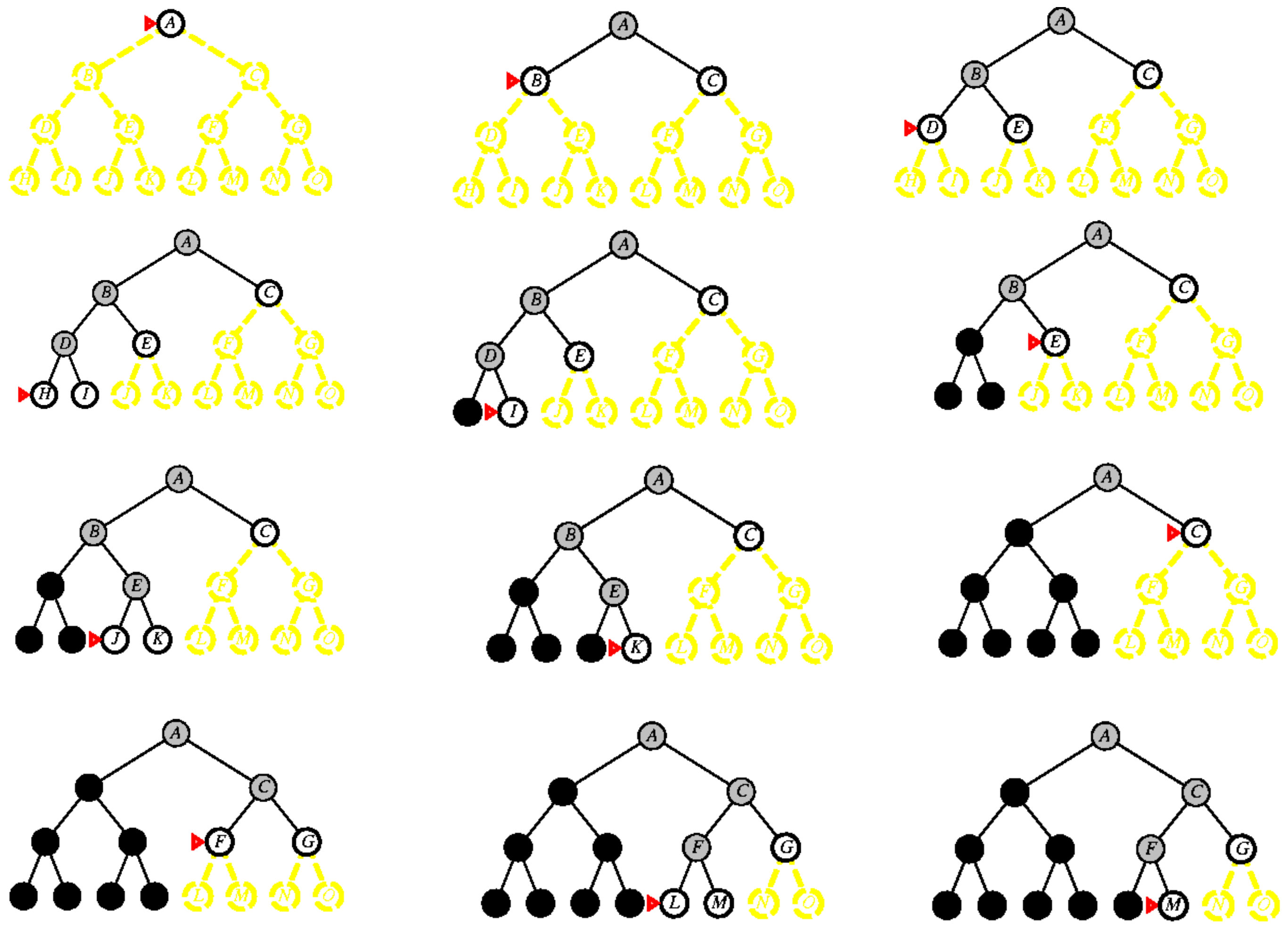
Breadth-first search

- Expand root first, then all nodes that have been created by the root and so forth
- All nodes of depth *d* are expanded before the first node of depth *d+1* will be created
- Can be implemented with GeneralSearch and a Queue.
- Time complexity: $O(b^d)$, here $b=2$.



Depth-First

- Depth-first search always expands the deepest node in the current fringe of the search tree.
- This strategy can be implemented by TREE - SEARCH with a last-in-first-out (LIFO).



Depth-First (2)

- Discussion:
  - Moderate space complexity (only single path in memory), max. depth *m* and branching factor *b = O(bm)* nodes.
  - Comparison to breadth-first: 156 KB instead of 10 EB with depth 16, a factor of 1:7 trillion times less memory.
  - Time: O(b^m).
  - Better as breath-first for problems with lots of solutions
  - But: you can get stuck if wrong choice leads to very long path (or even infinite)
  - $\rightarrow$ Depth-first not complete and not optimal.
  - Thus: avoid using depth-first with large or infinite maximal depths

Depth-First (3)

- Backtracking:
  - A variant of depth-first search called backtracking search uses still less memory.
  - Only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next.
  - In this way, only *O(m)* memory is needed rather than *O(bm)*
  - Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by *modifying* the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and *O(m)* actions.
  - For this to work, we must be able to undo each modification when we go back to generate the next successor. For problems with large state descriptions, such as robotic assembly, these techniques are critical to success.

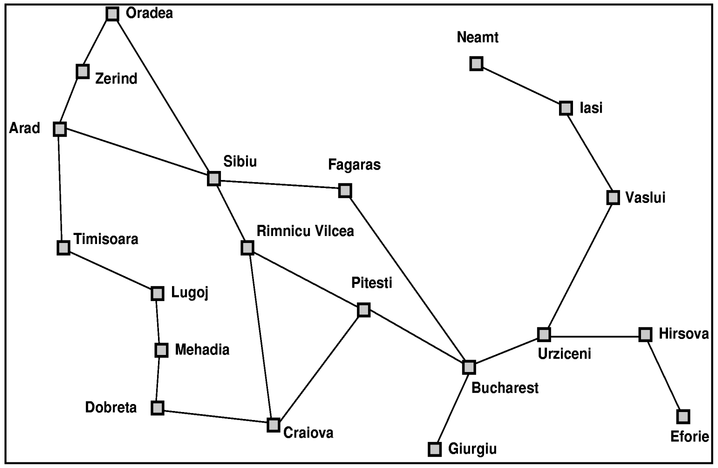
Depth-limited search

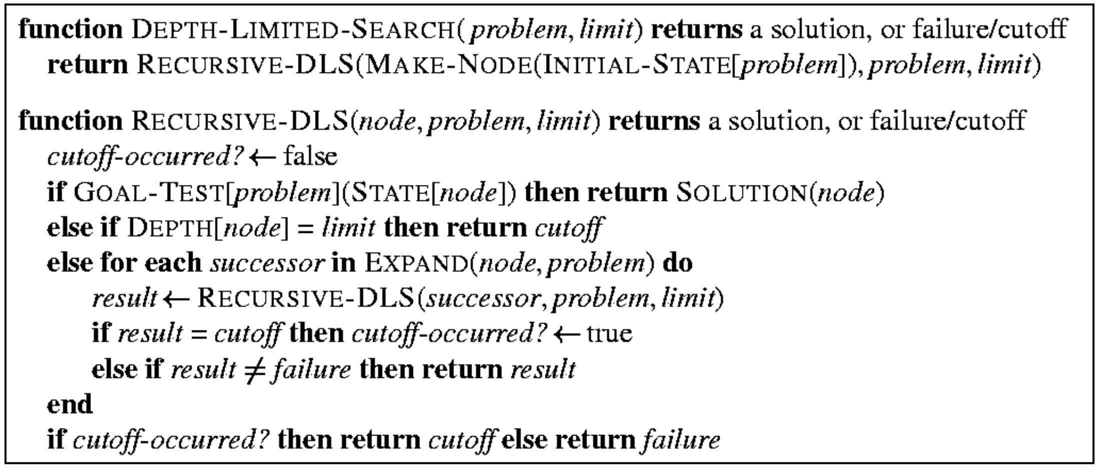
- Avoids disadvantage of depth-first through cut off max. depth of path
- Can be implemented with special algorithm or with GeneralSearch that includes operators, which save current.
- Example: Search for solution of a path having 20 cities. We know that max. depth=19
  - New operator: Suppose you are in city A and you have done < 19 steps. We then generate a new state in city B with path length that is increased by one.
- We find the solution guaranteed (if there is one) but algorithm is not optimal.
- Time and space complexity resembles depth-first search

- → **Depth-limited search is complete but not optimal**

Depth-limited search (2)

- Path length 19 is obvious, so *l=19* (max. path length). But diameter=9 if we have a closer look at map
- **Diameter** of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search.
- But: mostly, we will not know a good depth limit until we have solved the problem.

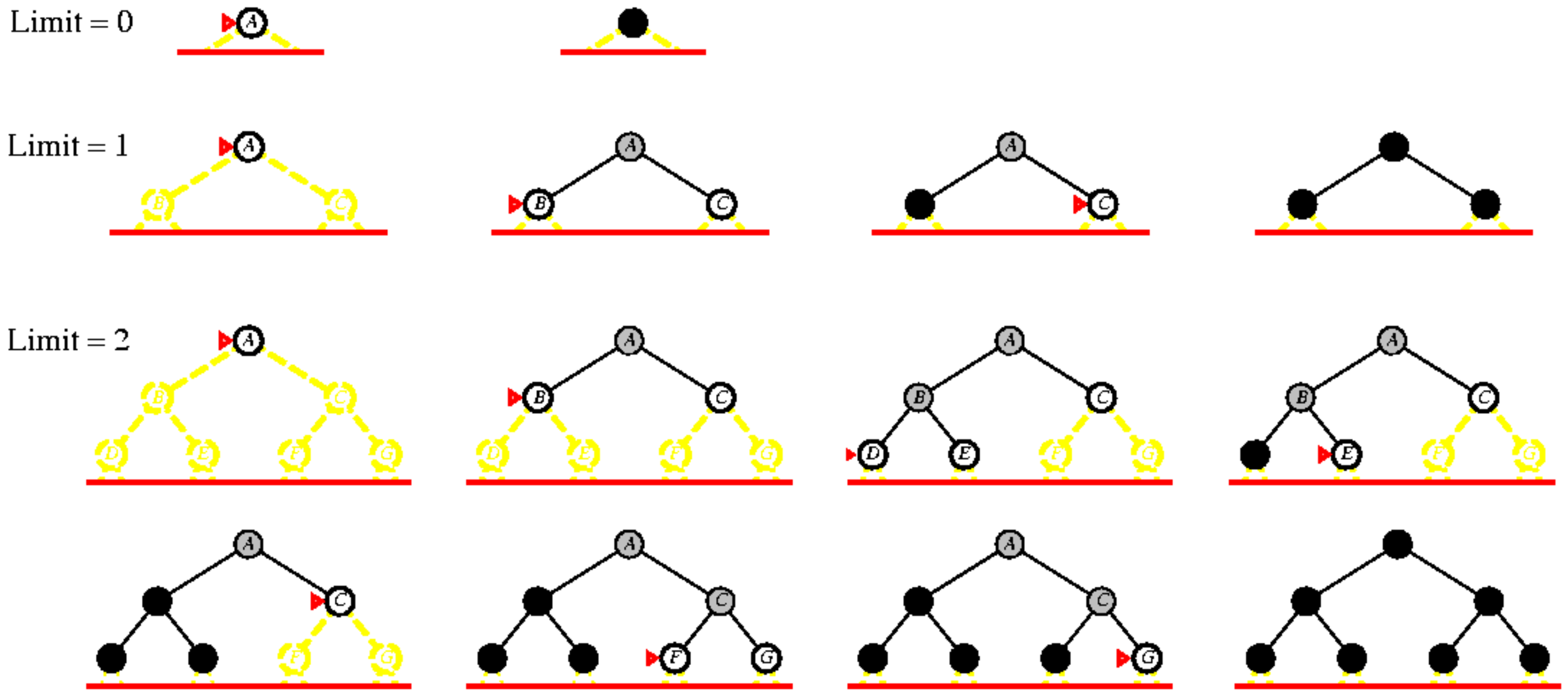




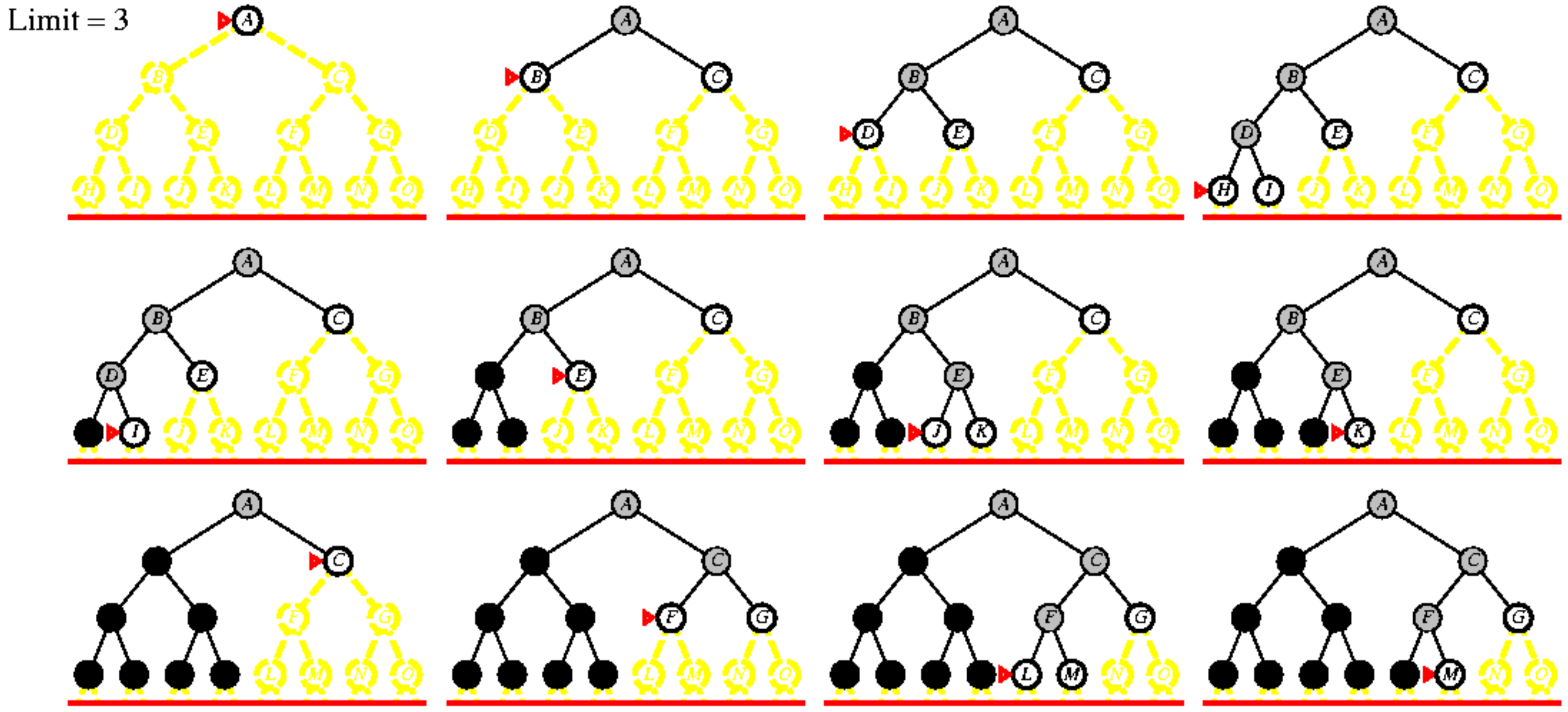
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
   return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/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
   end
   if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening depth-first search

- General strategy often used in combination with depth-first search, that finds the best depth limit.



Iterative deepening depth-first search (2)



Iterative deepening depth-first search (3)

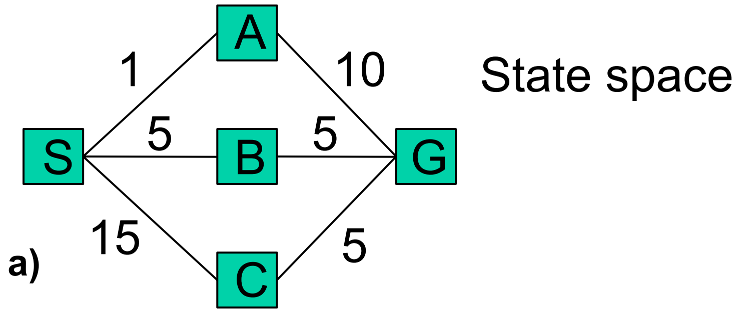
- Discussion:
  - Combination of advantages of breadth-first and depth-first search.
  - Iterative deepening depth-first search is complete and optimal[1].
  - States as in depth-first, approx. 11% more nodes.
  - Time: $O(b^d)$
  - Space: $O(bd)$
  - In general: iterative deepening if state space is large and depth of solution is unknown

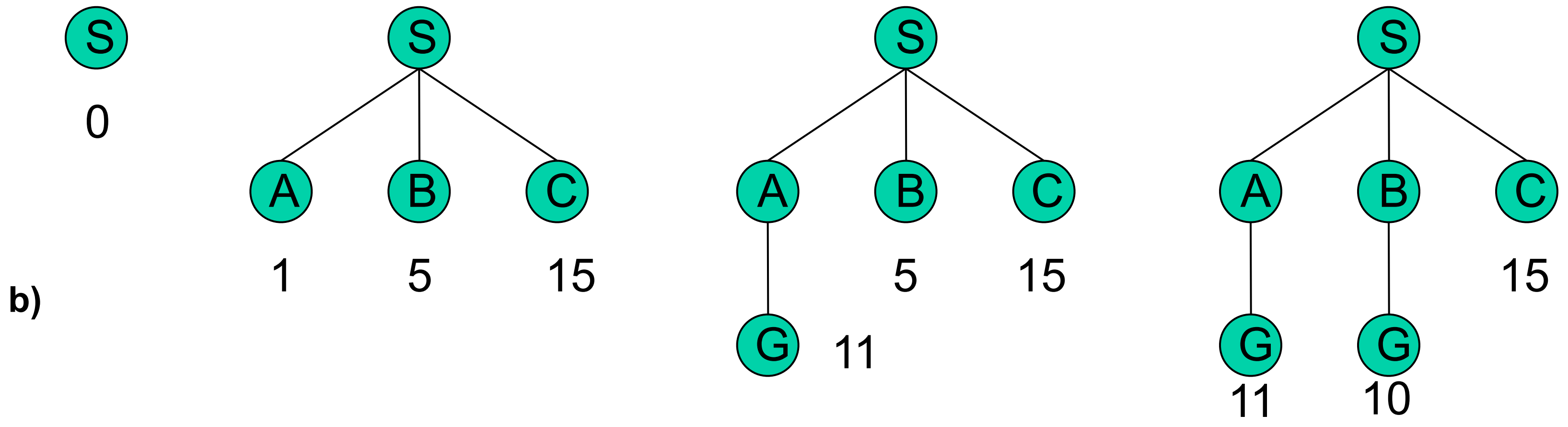
[1]: Complete, when the branching factor is finite and optimal when the path cost is a nondecreasing function of the depth of the node

Uniform-cost search aka, Branch-and-Bound

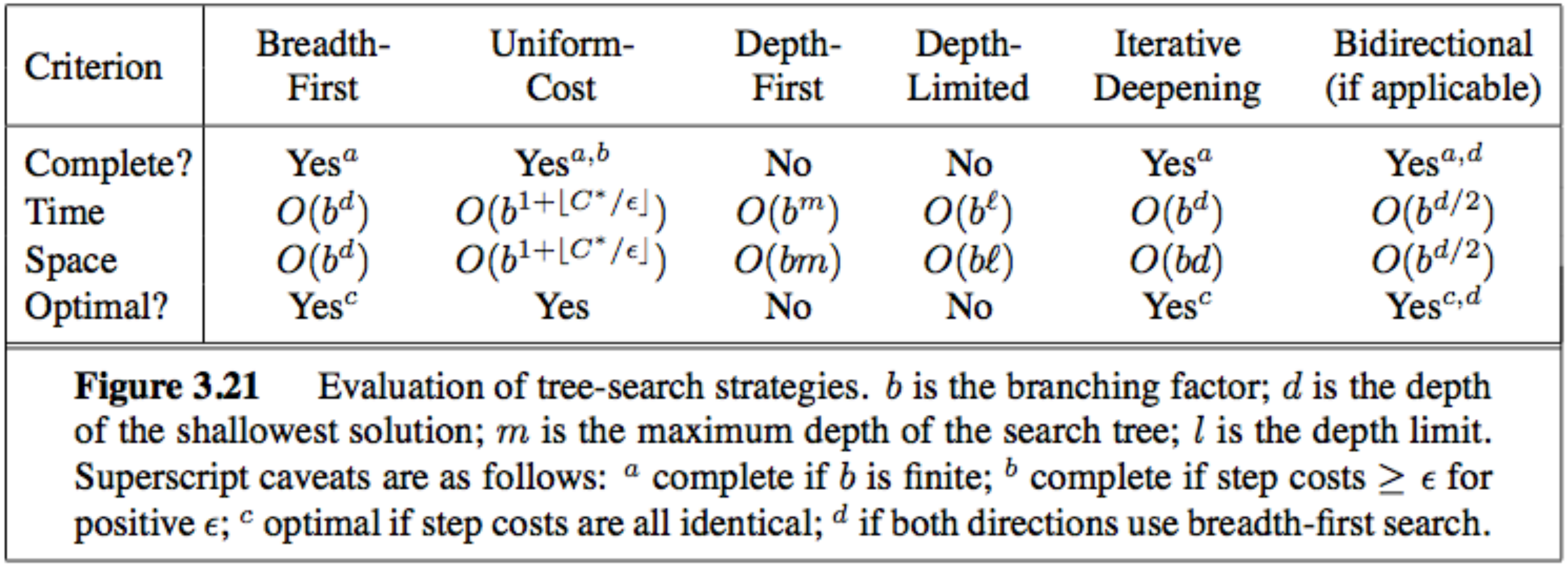
- Modified breadth-first, expands only the node with the least cost (using path *g(n)*).
- Breadth-first = uniform-cost search with *g(n)=depth(n)*.

Problem: from *S* to *G*, costs are marked (a). *SAG* is first solution (cost 11). The algorithm does not recognize this as solution, because 11 > 5 from *B*). *SBG* as final solution (b).





Comparing uninformed search strategies



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

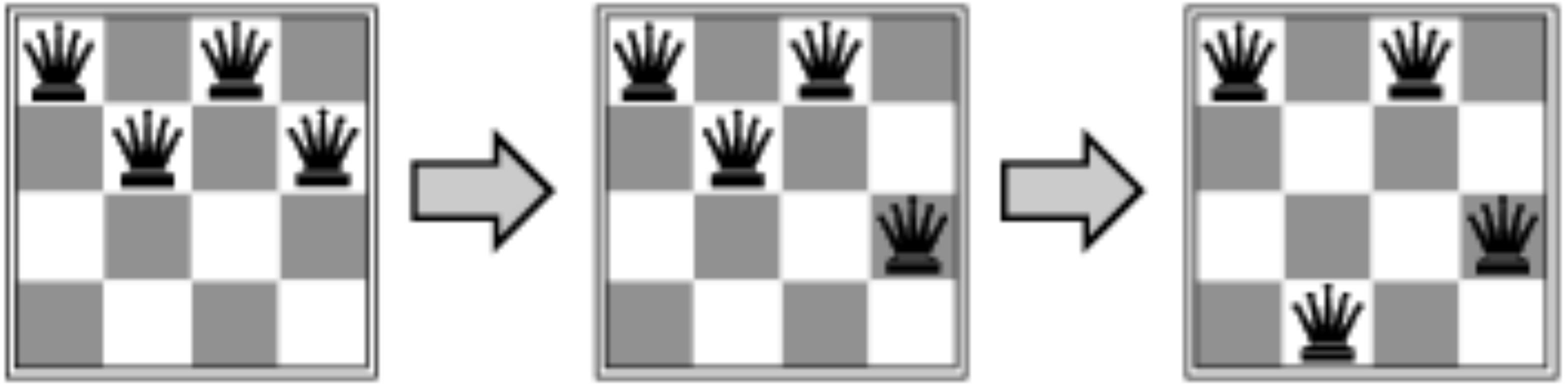
Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

Hill-climbing search

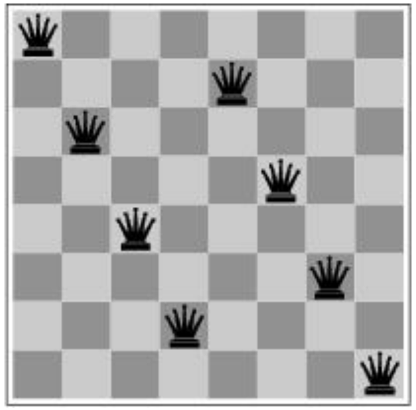
- "is a loop that continuously moves in the direction of increasing value", i.e. uphill
  - It terminates when a peak is reached.
- Hill climbing does not look ahead of the *immediate neighbors* of the current state.
- Hill-climbing chooses randomly among the set of best successors, if there is more than one.
- Hill-climbing a.k.a. *greedy local search*

Example: *n*-queens Problem

- Put *n* queens on an *n* × *n* board with no two queens on the same row, column, or diagonal



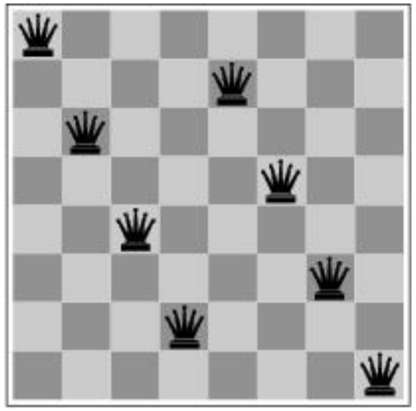
8-queens Problem Incremental or **Uninformed** Formulation



Incremental formulation: augment state description starting with an empty state) vs. complete-state formulation (starts with all 8 queens on board)

- States??
- Initial state??
- Actions??
- Goal test??

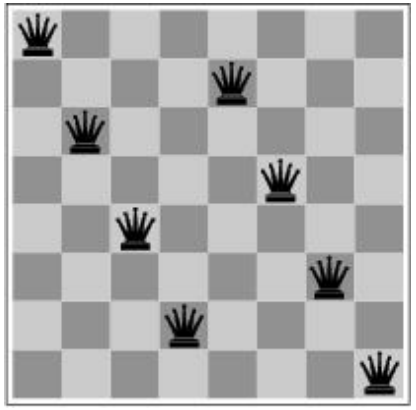
8-queens Problem Incremental or Uninformed Formulation



Incremental formulation:

- States? Any arrangement of 0 to 8 queens on the board
- Initial state? No queens on board
- Actions/Successor function? Add queen to any empty square
- Goal test? 8 queens on board and none attacked
- ➔ 64 x 63 x … x 57 possible sequences to investigate ≈ 1.8×10^{14}

8-queens Problem Incremental or **Uninformed** Formulation



Incremental formulation (alternative)

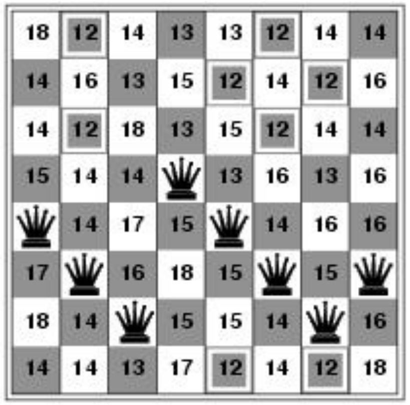
- States? *n* (0≤ *n*≤ 8) queens on the board, one per column in the *n* leftmost columns with no queen attacking another.
- Actions/Successor function? Add queen in leftmost empty column such that is not attacking other queens
- ➔ only 2057 possible sequences to investigate
- Yet makes no difference when *n*=100

8-queens problem Complete-state or **Informed** Formulation Hill-climbing example

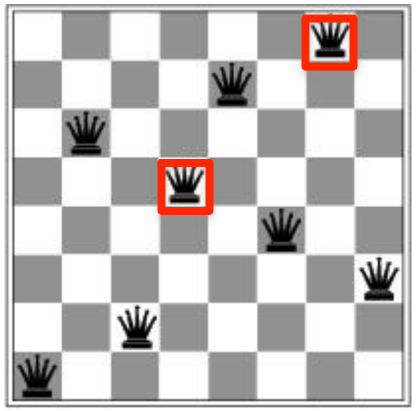
- Complete-state formulation (typically used in local searches): each state has 8 queens on board, one per column.
- **Successor function:** returns all possible states generated by moving a single queen to another square in the same column.
- **Heuristic function *h(n)*:** the number of pairs of queens that are attacking each other (directly or indirectly).

8-queens problem Hill-climbing example

a)



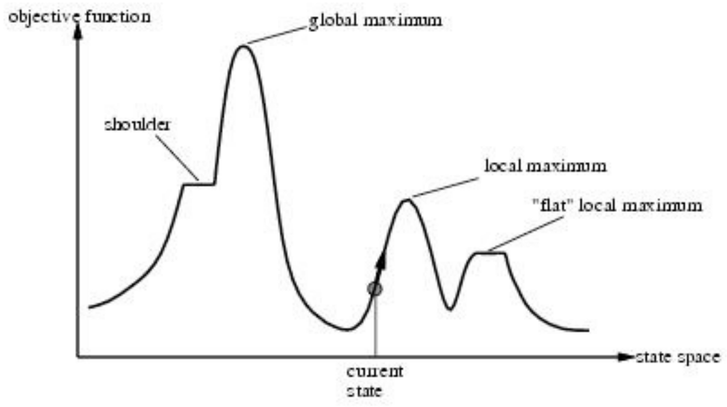
b)



a) Shows a state of *h*=17 and the *h*-value for each possible successor.

b) Shows a local minimum in the 8-queens state space (*h*=1).

Drawbacks

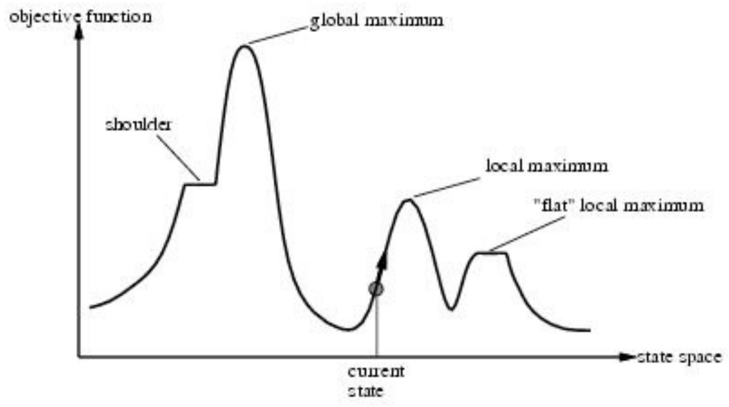


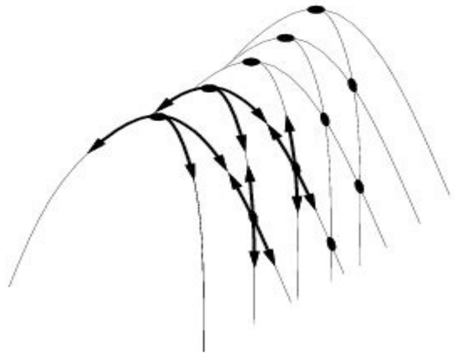
- **Local maxima:** local max is peak that is higher than each of its neighboring states, but lower than global maximum
- **Plateaux:** an area of the state space where the evaluation function is flat.

➔ **Incomplete:** Gets stuck 86% of the time, solving only 14% of problem instances

➔ Works quickly: 4 steps on average when it succeeds and 3 steps when it gets stuck (not bad for state space with 8^8 ≈ 17 million states)

Drawbacks





- **Local maxima:** local max is peak that is higher than each of its neighboring states, but lower than global maximum
- **Ridge:** sequence of local maxima difficult for greedy algorithms to navigate
- **Plateaux:** an area of the state space where the evaluation function is flat.
- Gets stuck 86% of the time, works quickly (4 steps on average when it succeeds and 3 when it gets stuck – not bad for state space with $8^8 \approx 17$ million states)

Hill-climbing variations

- Stochastic hill-climbing
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- First-choice hill-climbing
  - Stochastic hill climbing by generating successors randomly until a better one is found.
- Random-restart hill-climbing
  - Tries to avoid getting stuck in local maxima.

Local Beam Search

- Keep track of *k* states rather than just one
  Start with *k* randomly generated states
- At each iteration, all the successors of all *k* states are generated

```
If any one is a goal state,
    stop;
Else
    select the k best successors from the complete list and repeat.
```

Best-first search

- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability", i.e. measures distance to the goal

  → Expand most desirable unexpanded node

- Implementation:

  Order the nodes in fringe in decreasing order of desirability

- Special cases:
  - greedy best-first search
  - A^* search

Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from *n* to *goal*
- e.g., $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest
- Greedy best-first search expands the node that appears to be closest to goal

Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$, keeps all nodes in memory
- Optimal? No

Minimizing total path cost: A* search

- **Greedy search** minimizes the estimated cost to the goal *h(n),* and thereby cuts the search cost considerably.
  - But neither optimal nor complete
- **Uniform-cost** search minimizes the cost of the path so far *g(n)*
  - It is optimal and complete
  - But can be very inefficient
- How about **combining** these two strategies to get advantages of both?

  → A* algorithm (due to Nils Nilsson for *Shaky* the robot)

A* search

- *Best-known form of best-first search*.
- Idea: avoid expanding paths that are already expensive
- Combines the two evaluation functions (of UCS and GBFS) by summing them up
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost (so far) from start node to reach n
  - $h(n)$ = estimated cost to get from n to goal
  - $f(n)$ = estimated total cost of cheapest path solution through n to goal

Admissible heuristics

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
  - Formally, a heuristic *h(n)* is admissible if for every node *n:*
    - $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.
    - $h(G) = 0$ for any goal *G.*
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- This optimism transfers to the *f* function:

  If *h* is admissible, since *g(n)* is the exact cost to reach *n*, *f(n)* never overestimates the actual cost of the best solution through *n*.
- Theorem: If *h(n)* is admissible, A^* using `TREE-SEARCH` is optimal

Properties of A*

- Completeness: Yes
- Time complexity: exponential with path length
- Space complexity: all nodes are stored
- Optimality: Yes
  - Cannot expand f_{i+1} until f_i is finished.
  - A* expands all nodes with $f(n) < C^*$
  - A* expands some nodes with $f(n)=C^*$
  - A* expands **no** nodes with $f(n)>C^*$