بسم اللّه الرحمن الرحیم

Systematic Search Algorithms (Chapter 3)

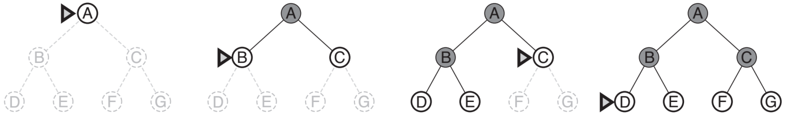
Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

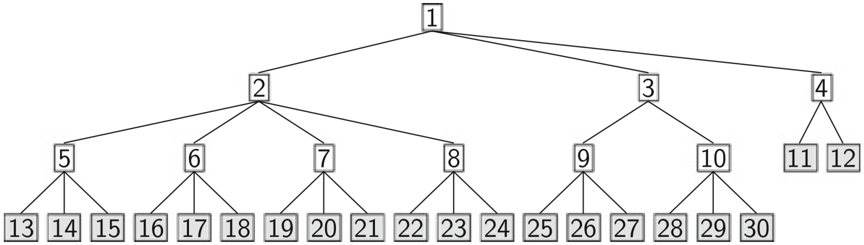
☞ The shallowest unexpanded node is chosen for expansion.
☞ A FIFO queue for the frontier.
☞ New nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.
☞ For BFS, the set frontier is realized as a FIFO queue: Enqueue & Dequeue

```
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)
```

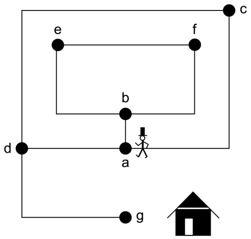
Breadth-first search on a graph.



BFS during the expansion of the third-level nodes. The nodes are numbered according to the order they were generated. The successors of nodes 11 and 12 have not yet been generated.



Example

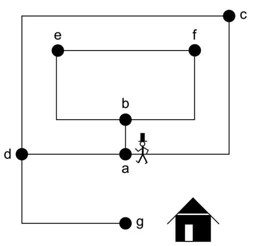


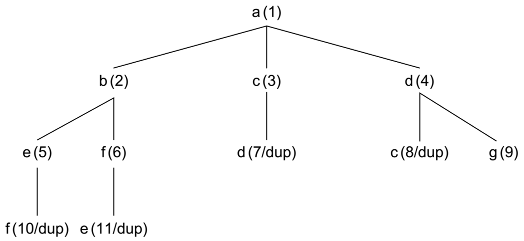
(with duplicate detection)

Step	Selection	Open	Closed	Remarks
1	{}	{a}	{}	
2	a	{b,c,d}	{a}	
3	b	{c,d,e,f}	{a,b}	
4	c	{d,e,f}	{a,b,c}	d is duplicate
5	d	{e,f,g}	{a,b,c,d}	c is duplicate
6	e	{f,g}	{a,b,c,d,e}	f is duplicate
7	f	{g}	{a,b,c,d,e,f}	e is duplicate
8	g	{}	{a,b,c,d,e,f,g}	Goal reached

اینجا، وقتی قرار است فرزندان نود ساخته شوند، چک می‌شود که نود خودش goal هست یا نه. اما در شبه‌کد صفحه ۳، به‌محض ساخته شدن یک نود، goal بودن یا نبودن آن چک می‌شود.

Example





How does BFS rate according to the four criteria?

Completeness: We can easily see that it is complete—if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite). Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test.

Optimality: The shallowest goal node is not necessarily the optimal one; technically, breadth-first search is optimal if the path cost is a nondecreasing function of the depth of the node. The most common such scenario is that all actions have the same cost.

The news about time and space is not so good!

Time Complexity: Imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d.In the worst case, it is the last node generated at that level. Then the total number of nodes generated is $b + b^2 + b^3 + \cdots + b^d = O(b^d)$.

(If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $O(b^{d+1})$.)

Space complexity: Every node generated remains in memory. There will be $O(b^{d-1})$ nodes in the explored set and $O(b^d)$ nodes in the frontier, so the space complexity is $O(b^d)$, i.e., it is dominated by the size of the frontier. Switching to a tree search would not save much space, and in a state space with many redundant paths, switching could cost a great deal of time.

چه از نظر پیچیدگی زمانی و چه از نظر پیچیدگی حافظه، BFS مطلوب به‌نظر نمی‌رسد.

Time and memory requirements for breadth-first search

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Two lessons can be learned from the above table

First, the memory requirements are a bigger problem for breadth-first search than is the execution time. One might wait 13 days for the solution to an important problem with search depth 12, but no personal computer has the petabyte of memory it would take. Fortunately, other strategies require less memory.

The second lesson is that time is still a major factor. If your problem has a solution at depth 16, then (given our assumptions) it will take about 350 years for breadth-first search to find it.

Uniform-cost search

When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost $g(n)$. This is done by storing the frontier as a priority queue ordered by g.

Uniform-cost search

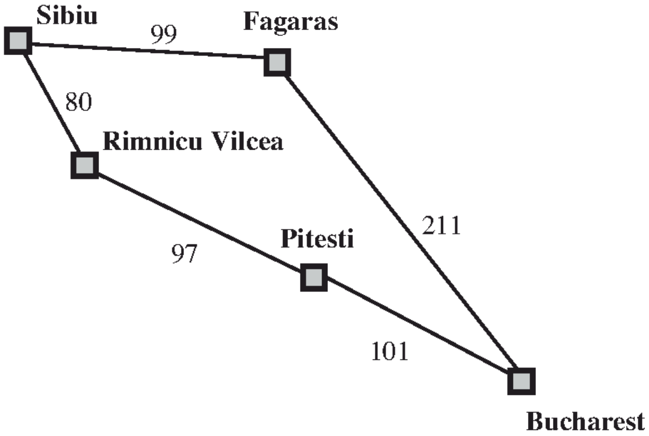
☞ In addition to the ordering of the queue by path cost, there are two other significant differences from breadth-first search. The first is that the goal test is applied to a node when it is selected for expansion rather than when it is first generated. The reason is that the first goal node that is generated may be on a suboptimal path.

☞ The second difference is that a test is added in case a better path is found to a node currently on the frontier.

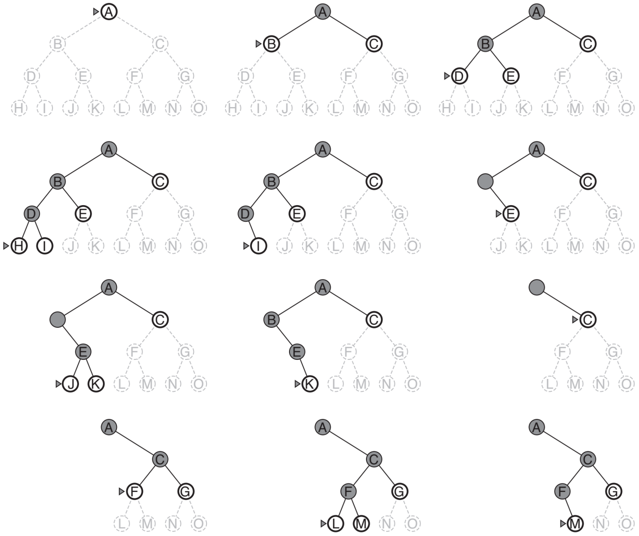
Uniform-cost search

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only 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
```

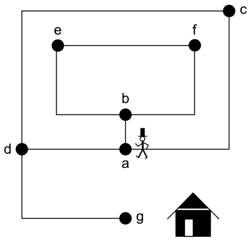
Example


Sibiu
99
Fagaras
80
Rimnicu Vilcea
211
Pitesti
97
101
Bucharest

Depth-first search (DFS)



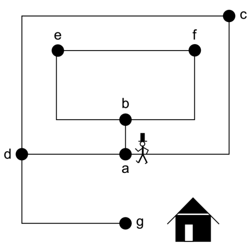
Example

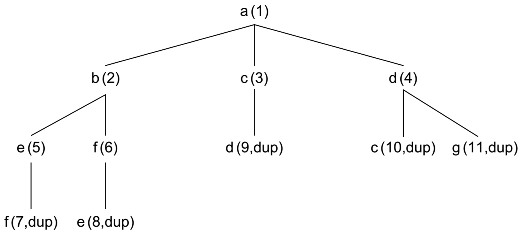


(with duplicate detection)

Step	Selection	Open	Closed	Remarks
1	{}	{a}	{}	
2	a	{b,c,d}	{a}	
3	b	{e,f,c,d}	{a,b}	
4	e	{f,c,d}	{a,b,e}	f is duplicate
5	f	{c,d}	{a,b,e,f}	e is duplicate
6	c	{d}	{a,b,e,f,c}	d is duplicate
7	d	{g}	{a,b,e,f,c,d}	c is duplicate
8	g	{}	{a,b,e,f,c,d,g}	Goal reached

Example





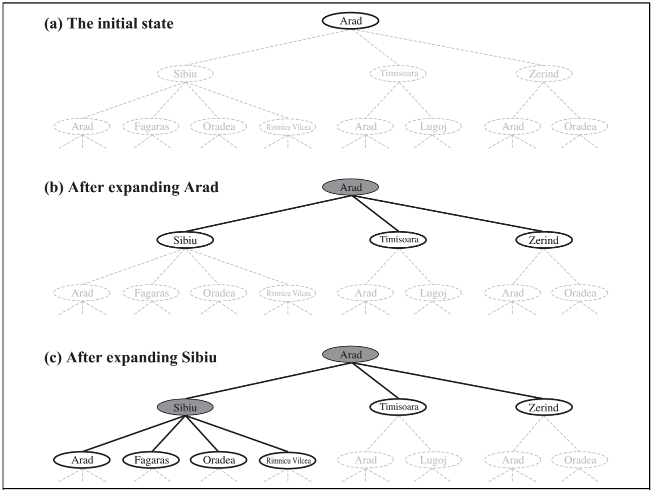
Depth-first search (DFS)

☞ Depth-first search always expands the deepest node in the current frontier of the search tree.
☞ DFS search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion.
☞ For depth-first search (DFS), the Open list is implemented as a stack (a.k.a. a LIFO, or last-in first-out queue), so that Insert is in fact a push operation and Select corresponds to a pop operation.
☞ It is easy to see that in finite search spaces DFS is complete (i.e., will find a solution path if there is some), since each node is expanded exactly once. It is, however, not optimal.
☞ Without duplicate elimination, DFS can get trapped in cycles of the problem graph and loop forever without finding a solution at all.

DFS

☞ The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node.
☞ The tree-search version, on the other hand, is not complete.
☞ In infinite state spaces, both versions fail if an infinite non-goal path is encountered. (مثلاً؟)

مثال: چرا نسخهٔ tree-search از الگوریتم DFS کامل نیست؟ (حتی در فضای حالت متناهی)



☞ **Optimality:** Both versions are nonoptimal.

☞ **Time complexity:** The time complexity of depth-first graph search is bounded by the size of the state space (which may be infinite, of course). A depth-first tree search, on the other hand, may generate all of the $O(b^m)$ nodes in the search tree, where m is the maximum depth of any node. Note that m itself can be much larger than d (the depth of the shallowest solution) and is infinite if the tree is unbounded.

☞ **Space complexity:** For a graph search, there is no advantage over BFS, but a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

☞ For a state space with branching factor b and maximum depth m, depth-first search requires storage of only $O(bm)$ nodes.

Depth-limited search

The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit ℓ. That is, nodes at depth are treated as if they have no successors. This approach is called depth-limited search. The depth limit solves the infinite-path problem.

Unfortunately, it also introduces an additional source of incompleteness if we choose $\ell < d$, that is, the shallowest goal is beyond the depth limit. (This is likely when d is unknown.) Depth-limited search will also be nonoptimal if we choose $\ell > d$. Its time complexity is $O(b^\ell)$ and its space complexity is $O(b\ell)$. Depth-first search can be viewed as a special case of depth-limited search with $\ell = \infty$.

Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so $\ell = 19$ is a possible choice.

```
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
      $cutoff\_occurred? \leftarrow$ false
      for each $action$ in $problem$.ACTIONS($node$.STATE) do
          $child \leftarrow$ CHILD-NODE($problem$, $node$, $action$)
          $result \leftarrow$ RECURSIVE-DLS($child$, $problem$, $limit - 1$)
          if $result = cutoff$ then $cutoff\_occurred? \leftarrow$ true
          else if $result \neq failure$ then return $result$
      if $cutoff\_occurred?$ then return $cutoff$ else return $failure$
```

A recursive implementation of depth-limited tree search.