

Chapter S:II

II. Basic Search Algorithms

- ❑ Systematic Search
- ❑ Graph Theory Basics
- ❑ State Space Search
- ❑ Depth-First Search
- ❑ Backtracking
- ❑ Breadth-First Search
- ❑ Uniform-Cost Search
- ❑ AND-OR Graph Basics
- ❑ Depth-First Search of AND-OR Graphs
- ❑ AND-OR Graph Search Basics
- ❑ AND-OR Graph Search

Depth-First Search (DFS)

Depth-first search is an **uninformed (systematic)** search strategy.

DFS characteristics:

- ❑ Nodes at deeper levels in G are preferred.

A solution base that is most complete (= longest) is preferred.

- ❑ The smallest, indivisible (= atomic) step is node expansion:

If a node is explored (= reached), *all* of its direct successors n_1, \dots, n_k are generated at once. Among the n_1, \dots, n_k *one* node is chosen for expansion in the next step.

- ❑ If node expansion comes to an end, backtracking is invoked:

Node expansion is continued with the deepest node that has non-explored alternatives.

- ❑ Terminates (on finite, cyclefree graphs) with a solution, if one exists.

Caveat: Cyclic paths or infinite paths cannot be handled properly.

Remarks:

❑ Operationalization of DFS:

The OPEN list is organized as a stack, i.e., nodes are explored in a LIFO (last in first out) manner. [[OPEN list in BFS](#)] [[OPEN list in BF](#)]

Backtracking is a consequence of the organization of solution bases in OPEN; it is not the result of an active checking of solution bases.

❑ Depth of a **node** is defined with respect to the traversal tree: $depth(n)$ is the length of the back-pointer path for n in the traversal tree. A most recently generated node in DFS will therefore be a deepest node in the traversal tree.

Depth of a state s' in the state-space G can only be defined as minimum distance between start state s and s' .

Therefore, the traversal tree may contain several nodes referencing the same state s' but with different depth values for the nodes in the traversal tree.

Depth-First Search

Specification of DFS Algorithm Family

Algorithm: DFS

Input: s . Start node representing the initial state (problem) in G .
 $\text{successors}(n)$. Returns *new instances of* nodes for the successor states in G .
 $\star(n)$. Predicate that is *True* if n represents a goal state in G .
 $\text{constraints}(n)$. Predicate that is *True* if path repr. by n satisfies solution constraints.
 $\perp(n)$. Predicate that is *True* if n is a dead end.
 maxDepth . Depth-bound for nodes in G to consider ($\text{maxDepth} \geq 0$).
 visitedDepth . Returns maximum depth for nodes in G observed (call by reference).
Output: A node γ representing a solution path for s in G or the symbol *Fail*.

Specialization of Basic-OR.

Variants:

Basic-DFS: a minimal DFS version.

DFS: DFS with dead end recognition.

DL-DFS: DFS with fixed depth limit.

ID-DFS: an iterative deepening framework for DFS.

Compare: BT versions and BFS variants.

General Information: [\[Wikipedia\]](#)

Depth-First Search

Core Version of DFS

- ❑ Please note that OPEN is organized as a stack, indicated by functions *push(.)* and *pop(.)*.
- ❑ Memory consumption will be controlled explicitly using the function *cleanup_closed()*.

```
Basic-DFS(s, successors, ★, constraints)    // A deterministic variant of Basic-OR.

0.  s.parent = null;
    IF ★(s) THEN IF constraints(s) THEN RETURN(s); // Check path repr. by s.
1.  push(s, OPEN);    // Add node s at the front of OPEN.
2.  LOOP
3.    IF (OPEN ==  $\emptyset$ ) THEN RETURN(Fail);
4.    n = pop(OPEN); add(n, CLOSED); // Get + remove front node n from OPEN.
5.    FOREACH n' IN successors(n) DO    // Expand n.
        n'.parent = n;
        IF ★(n') THEN    // Check whether n' represents a solution path.
            IF constraints(n') THEN RETURN(n');
            push(n', OPEN);    // Add node n' at the front of OPEN.
        ENDDO
        IF (successors(n) ==  $\emptyset$ ) THEN cleanup_closed();    // Remove dead ends.
6.  ENDLOOP
```

Remarks:

- ❑ Function *cleanup_closed* deletes nodes from CLOSED that are no longer required:

A node that cannot become part of a solution path can be safely discarded.

From algorithm Basic-OR we know already that only solution bases represented by nodes in OPEN can become part of a solution path. Therefore, all nodes in the traversal tree that are in CLOSED and have no descendant in OPEN can be safely discarded. Nodes of this type are:

1. Nodes n' that fulfill $\perp (n')$ (= nodes that are dead ends) can be discarded. (They will never enter OPEN, see DFS).
2. Nodes in CLOSED without known successors (because the state is terminal or because of previous *cleanup_closed* calls) can be safely discarded.

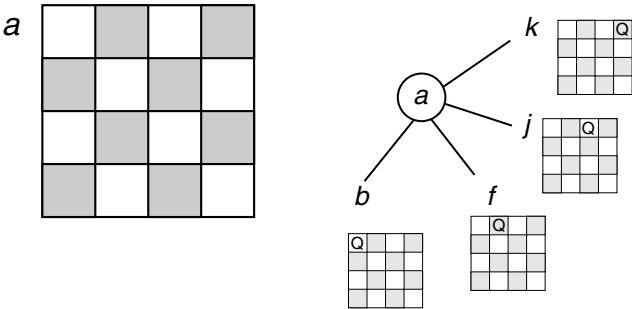
In order to find nodes of these types in CLOSED a reference count can be used.


- ❑ In depth-first search, the CLOSED list forms a path from s to the most recently expanded node with direct successors in OPEN.
- ❑ The node expansion under Step 5 exhibits the uninformed nature of DFS. A kind of “partial informedness” is achieved by introducing a heuristic h to sort among the direct successors:




```
5.    FOREACH  $n'$  IN sort(successors( $n$ ),  $h$ ) DO    // Expand  $n$ .  
       $s.parent = null$ ;  
      ...
```

Depth-First Search

Example: 4-Queens Problem

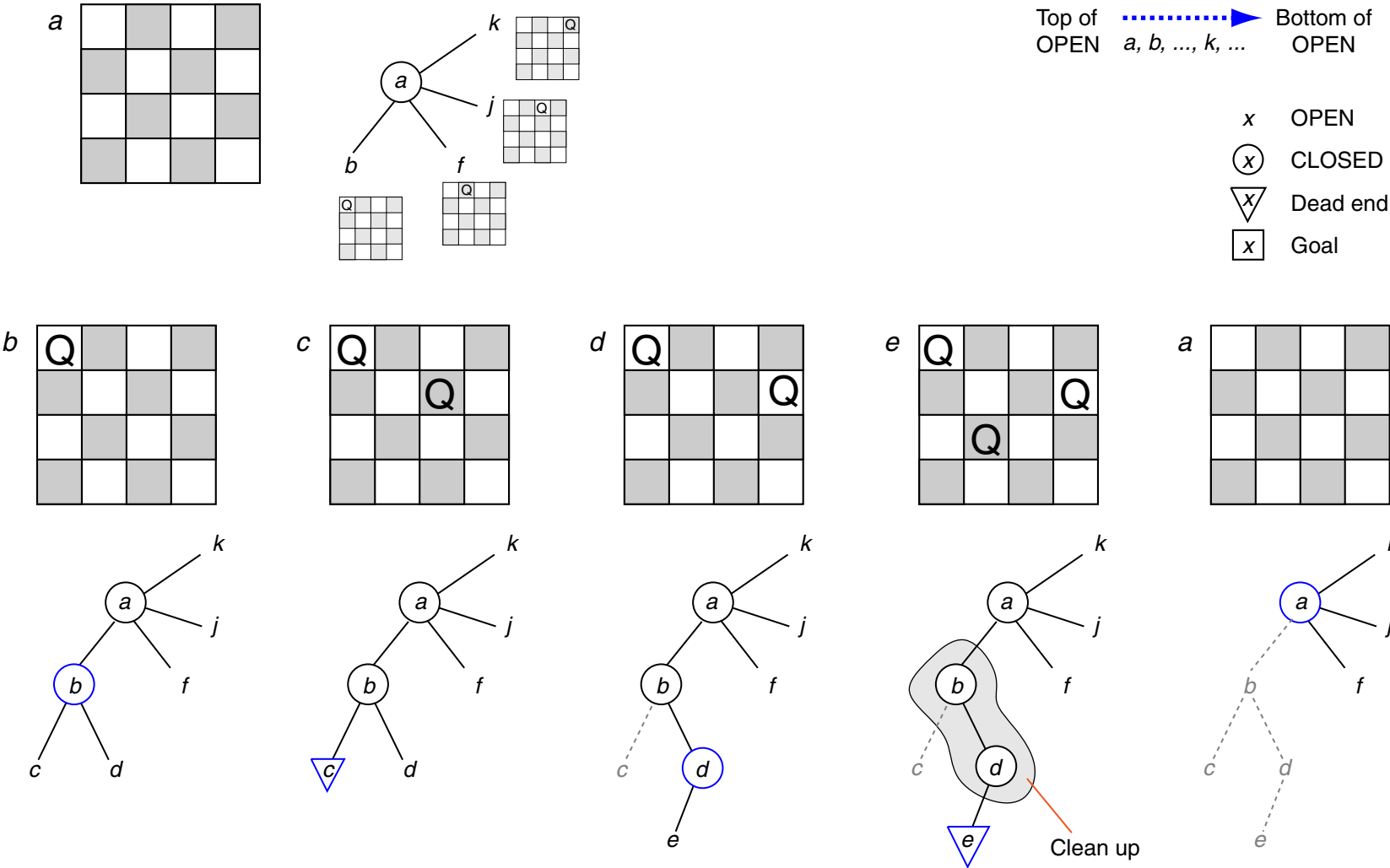


Top of OPEN  Bottom of OPEN
a, b, ..., k, ...

- x* OPEN
-  CLOSED
-  Dead end
-  Goal

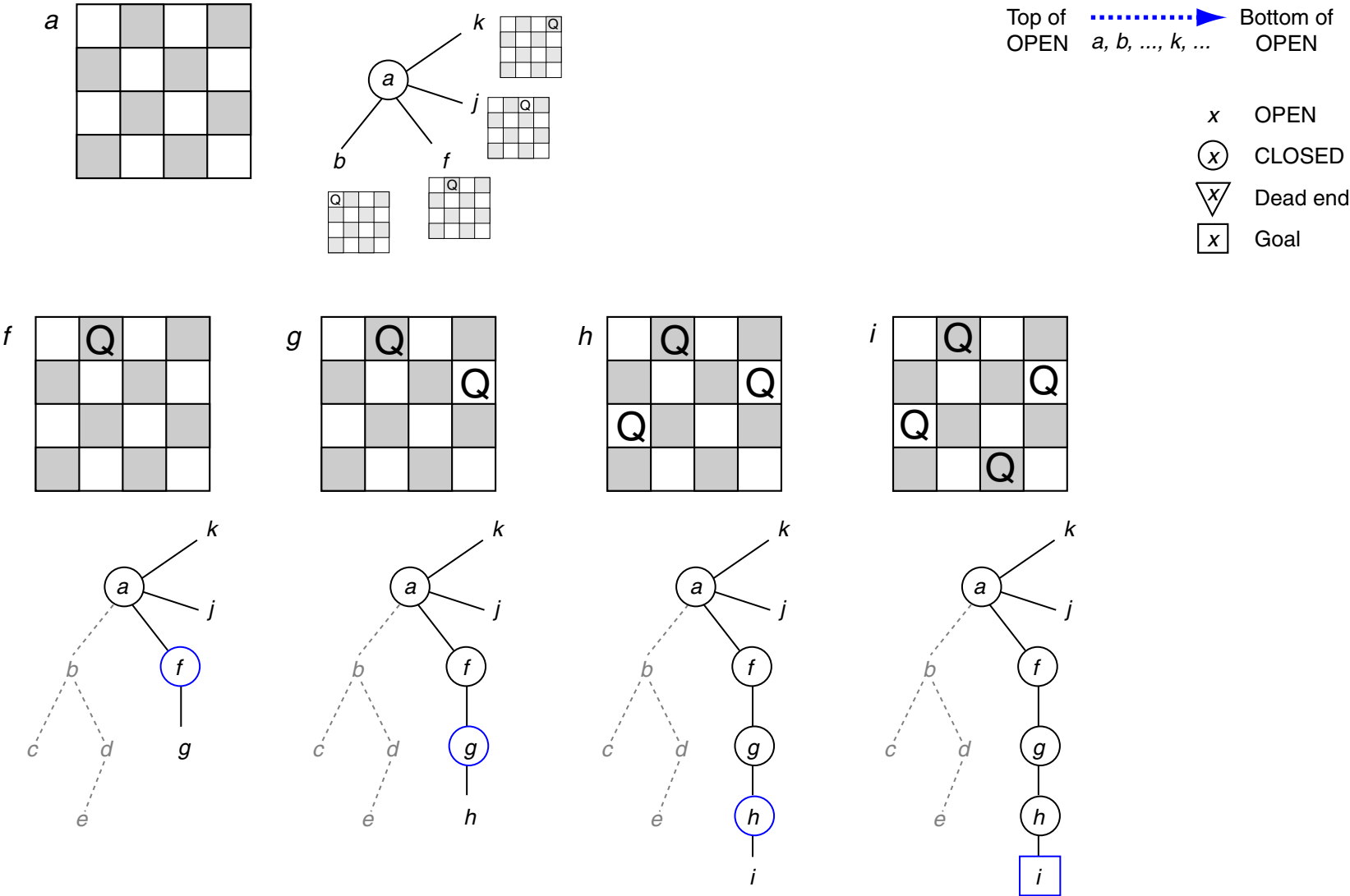
Depth-First Search

Example: 4-Queens Problem (continued)



Depth-First Search

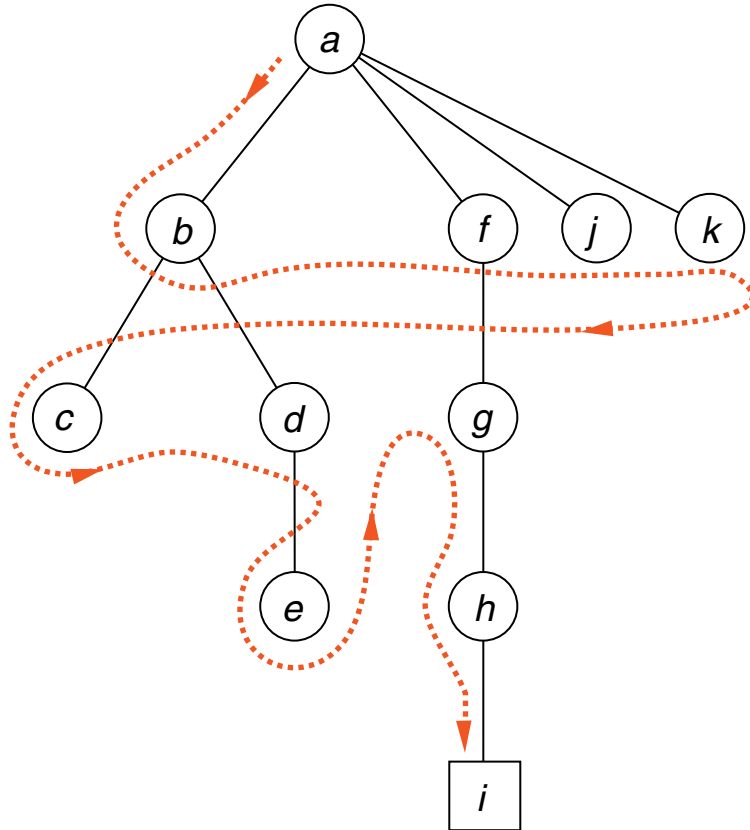
Example: 4-Queens Problem (continued)



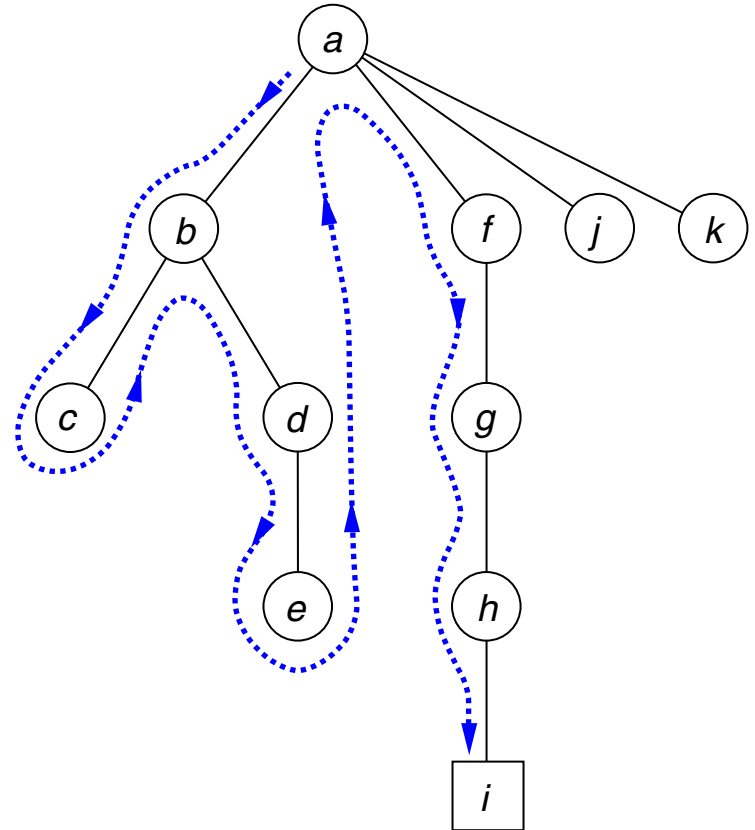
Depth-First Search

Example: 4-Queens Problem (continued)

DFS node processing sequence:



Node generation

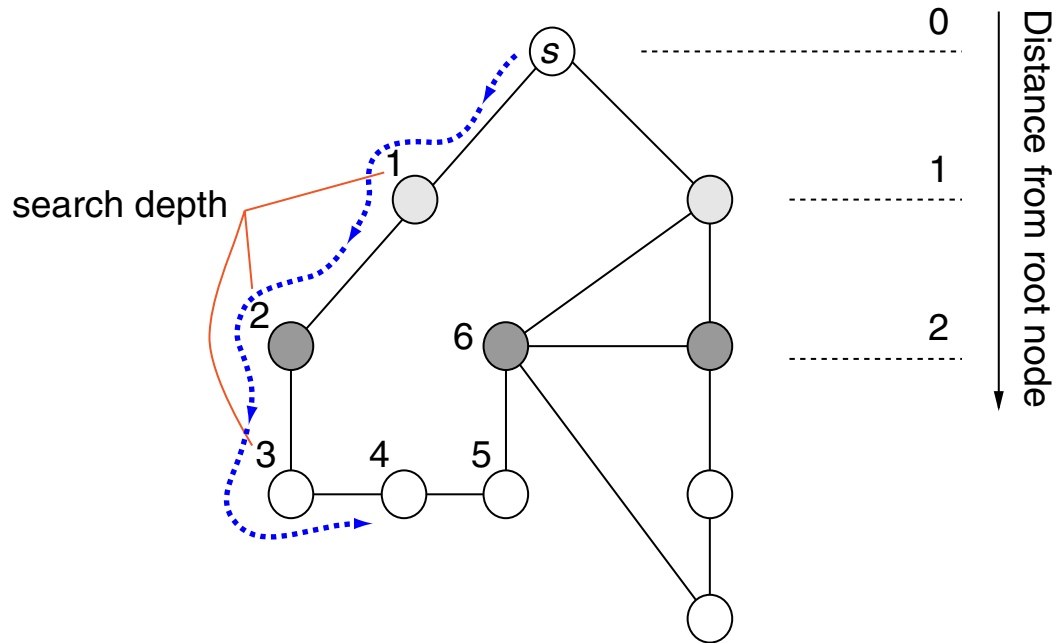


Node expansion

Depth-First Search

Search Depth

For graphs that are not trees, the search depth is not naturally defined:



In the search space graph G depth of a node n is defined as minimum length of a path from s to n .

DFS also uses this definition, but with respect to its finite knowledge about G which is a tree rooted in s .

Remarks:

- ❑ The DFS paradigm requires that a node will not be expanded as long as a deeper node is on the OPEN list.
- ❑ Q. How to define the depth of a node n ?

A. Ideally, $depth(n') = 1 + depth(n)$, where n is the highest parent node of n' in the search space graph G (= parent node that is nearest to s). Together with the base case $depth(s) = 0$ we have, depth of n is minimum length of a path from s to n .

DFS stores a finite tree rooted in s , which is a tree-unfolding of the part of the explored subgraph of G . Each instantiation of a node n has a depth in that tree, which is the length of the unique path from s to n in that tree.

As for DFS it is impossible to compute the depth of a node n in G (see [illustration](#)), DFS uses the depth of n in its stored tree. The DFS paradigm is then realized by the LIFO principle used for OPEN.

Depth-First Search

Discussion

Basic-DFS is complete for finite state-space graphs G without cycles.

DFS issue:

- ❑ Search may run deeper and deeper and follow some fruitless path.

Workaround 1:

- ❑ Install a depth-bound.
- ❑ When reaching the bound, trigger a jump (backtrack) to the deepest alternative not violating this bound.

Workaround 2:

- ❑ Check for dead ends with a “forecasting” dead end predicate $\perp(n)$.
- ❑ If $\perp(n) = \text{True}$, trigger a jump (backtrack) to the deepest alternative not violating the depth bound.

Depth-First Search

Depth-limited DFS

A depth limit *maxDepth* helps to avoid infinite paths, a call by reference parameter allows the interpretation of negative results ($(visitedDepth < maxDepth) \Rightarrow$ no solution available).

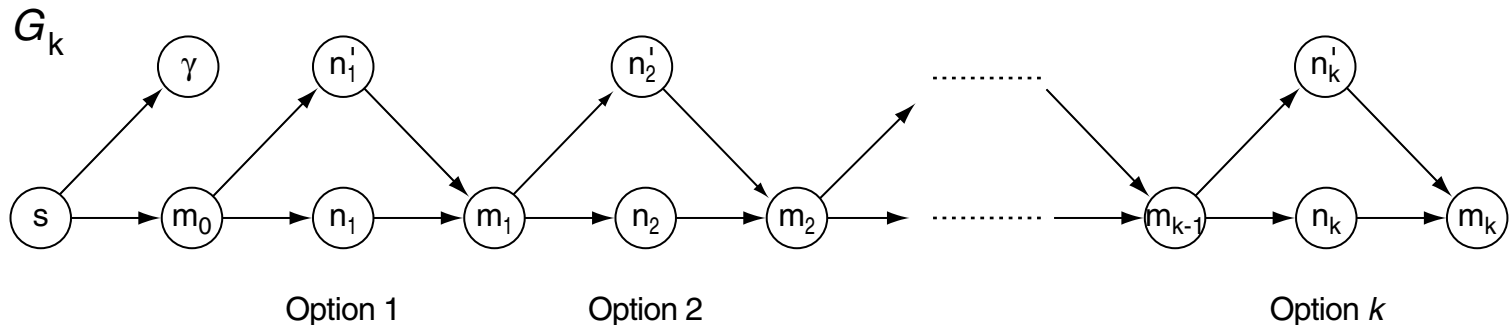
```
DL-DFS(s, successors, ★, constraints, maxDepth, visitedDepth)    // Extends Basic-DFS.

1.  s.parent = null; push(s, OPEN); depth(s) = 0; visitedDepth = 0;
2.  LOOP
3.    IF (OPEN == ∅) THEN RETURN(Fail);
4.    n = pop(OPEN); add(n, CLOSED);
      IF (depth(n) > visitedDepth) THEN visitedDepth = depth(n);
5.    IF (depth(n) >= maxDepth)    // Do not expand nodes of highest depth.
      THEN
        cleanup_closed();    // Remove unref. nodes from CLOSED, e.g., n.
      ELSE
        FOREACH n' IN successors(n) DO    // Expand n.
          n'.parent = n;
          depth(n') = depth(n) + 1;
          IF ★(n') THEN IF constraints(n') THEN RETURN(n');
          push(n', OPEN);    // Add node at the front of OPEN.
        ENDDO
        IF (successors(n) == ∅) THEN cleanup_closed();    // Remove dead ends.
      ENDIF
6.  ENDLOOP
```

Remarks:

- ❑ Reaching the depth bound at node n is treated as if n had no successors.
- ❑ Using a depth bound means that completeness of DL-DFS can be violated even for finite graphs without cycles. The reason is that the depth bound could be chosen less than minimum solution path length for s .
- ❑ Even if short solution paths for s are available the runtime of DL-DFS can be exponential in the size of G , since many long non-solution paths can be available. Since the order on which solution bases will be considered is not known, it is advisable to increase the bound gradually instead of using a high depth bound directly (see [ID-DFS](#)).

Example:



We assume, γ is the only goal state and successors of s are processed in the order m_0, γ .
 k independent options for “bottom” and “top” provide $2^k + 1$ different paths in G_k .

- ❑ Q. What is the asymptotic space requirement of DFS with depth bound?

Depth-First Search

Iterative Deepening Framework using DL-DFS

- If there is a chance for a short solution, do not call DFS with the maximal depth bound.

Algorithm: ID-DFS (Iterative-Deepening DFS) [[DFS family](#), [ID-A*](#), [Wikipedia](#)]

Input: s . Start node representing the initial state (problem) in G .
 $successors(n)$. Returns *new instances of* nodes for the successor states in G .
 $\star(n)$. Predicate that is *True* if n represents a goal state in G .
 $constraints(n)$. Predicate that is *True* if path repr. by n satisfies solution constraints.
initDepth. Initial depth bound.
maxDepthBound. Maximum depth bound to consider.

Output: A node γ representing a solution path for s in G or the symbol *Fail*.

ID-DFS(s , $successors$, \star , $constraints$, *initDepth*, *maxDepthBound*)

1. $curDepth = initDepth$; $visitedDepth = 0$;
2. **LOOP**
3. IF ($curDepth > maxDepth$) THEN RETURN(*Fail*);
4. $result = DL-DFS(s, successors, \star, constraints, curDepth, visitedDepth)$;
// *visitedDepth* is a call by reference parameter.
IF ($result \neq Fail$) THEN RETURN($result$);
IF ($visitedDepth < curDepth$) THEN RETURN(*Fail*); // *result* is reliable.
 $curDepth = increment(curDepth)$; // Increase *curDepth* reasonably.
5. **ENDLOOP**

Depth-First Search

DFS with Pruning

Information about solvability of a rest problem (state) represented by a node allows for pruning.

A domain-specific predicate $\perp(n)$ (dead end function) serves this purpose.

DFS(s , *successors*, \star , *constraints*, \perp) // Extends Basic-DFS

```
1.  s.parent = null;  push(s, OPEN);
2.  LOOP
3.    IF (OPEN ==  $\emptyset$ ) THEN RETURN(Fail);
4.    n = pop(OPEN);  add(n, CLOSED);
5.    FOREACH n' IN successors(n) DO      // Expand n.
        n'.parent = n;
        IF  $\star(n')$  THEN IF constraints(n') THEN RETURN(n');
        IF  $\perp(n')$       // Detect and remove dead ends in G.
        THEN
            add(n', CLOSED);
            cleanup_closed();    // Remove unref. nodes from CLOSED, e.g., n'.
        ELSE
            push(n', OPEN);    // Add node at the front of OPEN.
        ENDIF
    ENDDO
    IF (successors(n) ==  $\emptyset$ ) THEN cleanup_closed();    // Remove dead ends.
6.  ENDLLOOP
```

Remarks:

- ❑ Function $\perp(n)$ is part of the problem definition and not part of the search algorithm. We assume that $\perp(n)$ is computable.
- ❑ For each node n on a solution path we have $\perp(n) = \text{False}$
- ❑ Example definition of a very simple dead end predicate (its look-ahead is 0) :

$$\perp(n) = \text{True} \quad \Leftrightarrow \quad (\star(n) = \text{False} \wedge \text{successors}(n) = \emptyset)$$

- ❑ Whether a depth bound is reached, could easily be tested in $\perp(n)$ as well. But then a depth bound becomes part of the problem setting: “Find a solution path with at most length k .” Instead, we see the depth bound as part of the search strategy that excludes solution bases longer than k from further exploration.

Depth-First Search

Discussion (continued)

Depth-first search can be the favorite strategy in certain situations:

- (a) We are given plenty of equivalent solutions.
- (b) Dead ends can be recognized early (i.e., with a considerable look-ahead).
- (c) There are no cyclic or infinite paths resp. cyclic or infinite paths can be avoided.

Corollary 12 (Termination and Completeness of Basic-DFS for Finite Graphs without Cycles)

Basic-DFS terminates and is complete for finite graphs G without cycles that have the $Prop_0(G)$ properties.

(See completeness of Basic-OR.)

Remarks:

- Q. How to avoid the multiple expansion of a node, i.e., the expansion of allegedly different nodes all of which represent the same state (recall the 8-Puzzle problem)?

A. Multiple nodes referencing the same state can be available at the same time in the traversal tree stored by DFS. Each of these instantiations represents a different solution base. It makes sense to avoid multiple expansions of a node if constraints for solution paths are not too restrictive. This is for example the case if $\star(n)$ only checks, whether n is a goal node.

Then a solution base represented by one instantiation of n can be completed to a solution path if and only if a solution base represented by some other instantiation of n can be completed to a solution path.

In such cases, some of the multiple extensions can be avoided by checking (e.g., in $\perp(n)$ for the case of cycles) whether a node referencing the same state as n is currently available in OPEN or CLOSED.

- If multiple nodes referencing the same state can be avoided (no constraints and removing duplicates), the resulting algorithm terminates and is complete for finite graphs G with $Prop_0(G)$.

To completely eliminate multiple expansions, all nodes that have ever been expanded must be stored, which sweeps off the memory advantage of DFS.

Depth-First Search

Example: DFS for Optimization (see Basic-OR for Optimization)

Determine the minimum column sum of a matrix:

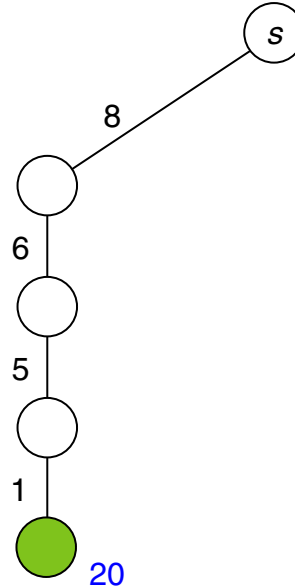
8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6

Depth-First Search

Example: DFS for Optimization (see Basic-OR for Optimization)

Determine the minimum column sum of a matrix:

8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6



6 (black) edge cost

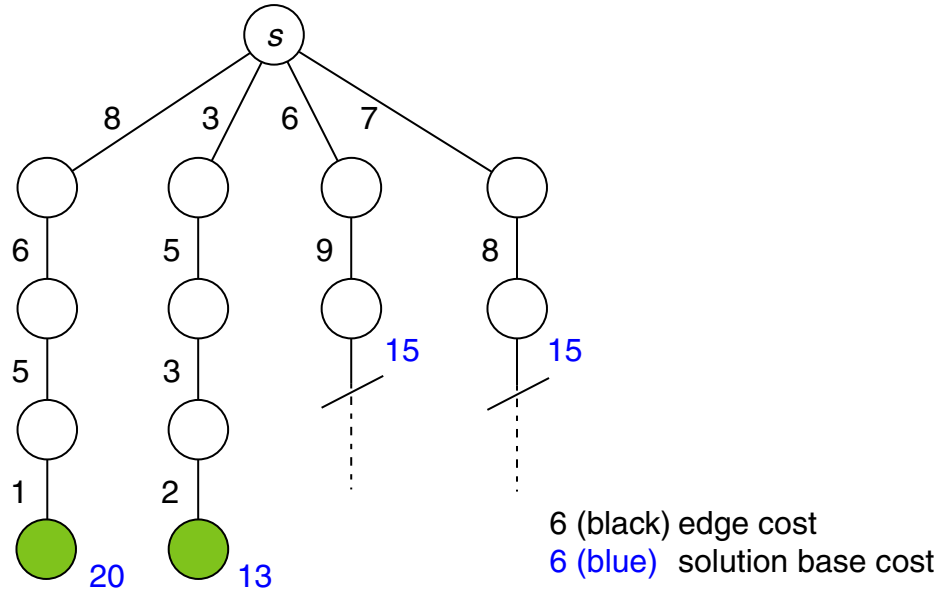
6 (blue) solution base cost

Depth-First Search

Example: DFS for Optimization (see Basic-OR for Optimization)

Determine the minimum column sum of a matrix:

8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6



Prerequisite for pruning:

Objective function *column sum* increases monotonically with the depth of a path.

Solution bases exceeding cost of a cheapest solution found so far can be pruned.

Chapter S:II

II. Basic Search Algorithms

- ❑ Systematic Search
- ❑ Graph Theory Basics
- ❑ State Space Search
- ❑ Depth-First Search
- ❑ Backtracking
- ❑ Breadth-First Search
- ❑ Uniform-Cost Search
- ❑ AND-OR Graph Basics
- ❑ Depth-First Search of AND-OR Graphs
- ❑ AND-OR Graph Search Basics
- ❑ AND-OR Graph Search

Backtracking (BT)

Backtracking is a variant of depth-first search.

BT characteristics:

- ❑ The LIFO principle is applied to node generation—as opposed to node expansion in DFS.
(I.e., the illustrated difference in time of node generation and time of node processing disappears.)
- ❑ When selecting a node n for exploration, only *one* of its direct successors n' is generated.
- ❑ If n' fulfills the dead end criterion (i.e., $\perp(n') = \text{True}$), backtracking is invoked and search is continued with the next non-expanded successor of n .

Remarks:

- ❑ The operationalization of BT can happen elegantly via recursion. Then, the OPEN list is realized as the stack of recursive function calls (i.e., the data structure is provided directly by programming language and operating system means).

Backtracking

Algorithm: BT

Input: s . Start node representing the initial state (problem) in G .
 $next_successor(n)$. Returns a *new instance of* a node for the next succ. in G .
 $\star(n)$. Predicate that is *True* if n represents a goal state in G .
 $constraints(n)$. Predicate that is *True* if path repr. by n satisfies solution constraints.

Output: A node γ representing a solution path for s in G or the symbol *Fail*.

```
BT( $s, next\_successor, \star, constraints$ )           // Recursive version of Basic-DFS.  
1.  $s.parent = null$ ;  $push(s, OPEN)$ ;    // Assuming  $s$  represents no solution path.  
2. LOOP  
3.   IF ( $OPEN == \emptyset$ ) THEN RETURN(Fail);  
4.    $n = top(OPEN)$ ;    // Get front node  $n$  from  $OPEN$ , no removing.  
5.    $n' = next\_successor(n)$ ;    // Get a next successor of  $n$ .  
      IF ( $n' == null$ )  
      THEN  
           $pop(OPEN)$ ;  $remove(n)$ ;    // Remove  $n$  from  $OPEN$  and free memory.  
      ELSE  
           $n'.parent = n$ ;  
          IF  $\star(n')$  THEN IF  $constraints(s)$  THEN RETURN( $n'$ );  
           $push(n', OPEN)$ ;  
      ENDIF  
6. ENDLOOP
```

Backtracking

Discussion

BT issue:

- ❑ Heuristic information to sort among successors is not exploited.

Workarounds:

- ❑ Generate successors temporarily.
- ❑ Assess operators — instead of the effect of operator application.

Backtracking can be the favorite strategy if we are given a large number of applicable operators.

Backtracking

Discussion (continued)

Backtracking versus depth-first search:

- ❑ Backtracking requires only an OPEN list, which serves as both LIFO stack and traversal path storage.

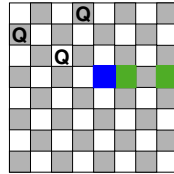
DFS employs an OPEN list to maintain alternatives and a CLOSED list to support the decision which nodes to discard. So, CLOSED stores the current path (aka. traversal path).

- ❑ Even higher storage economy compared to DFS:
 - BT will store only one successor at a time.
 - BT will never generate nodes to the “right” of a solution path.
 - BT will discard a node as soon as it is known to be expanded.

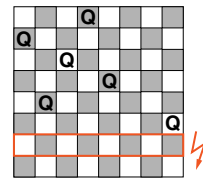
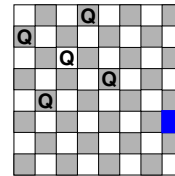
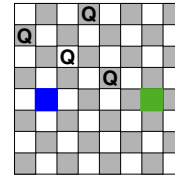
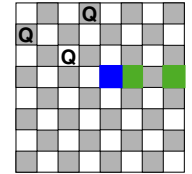
Backtracking

Monotone Backtracking: 8-Queens Problem

Time t



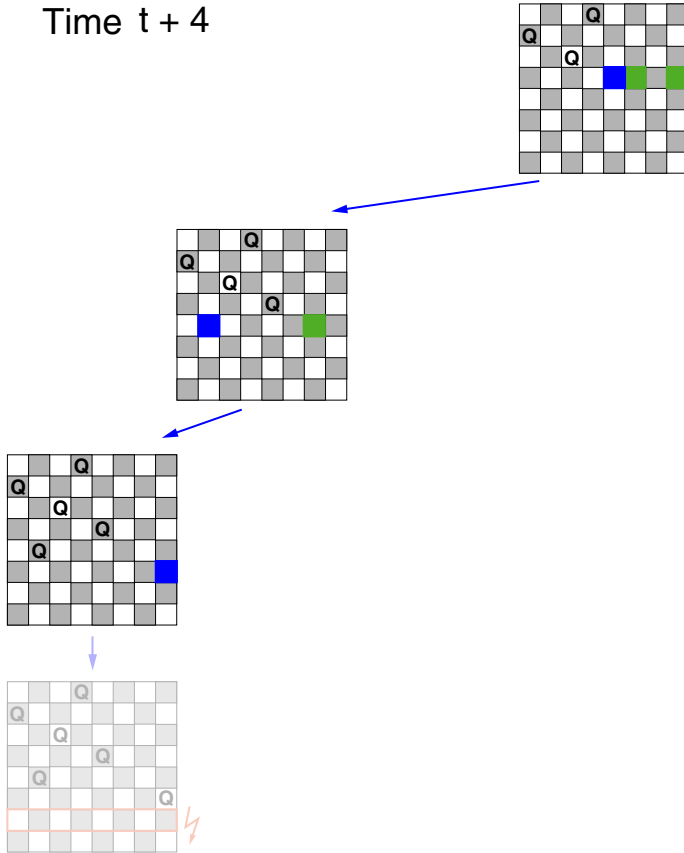
Time $t + 3$



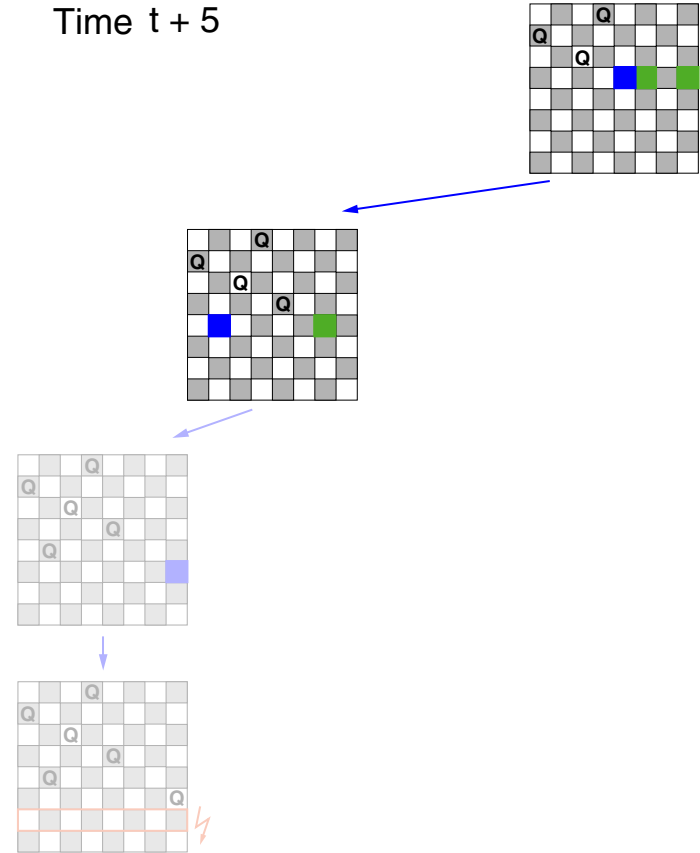
Backtracking

Monotone Backtracking: 8-Queens Problem (continued)

Time $t + 4$

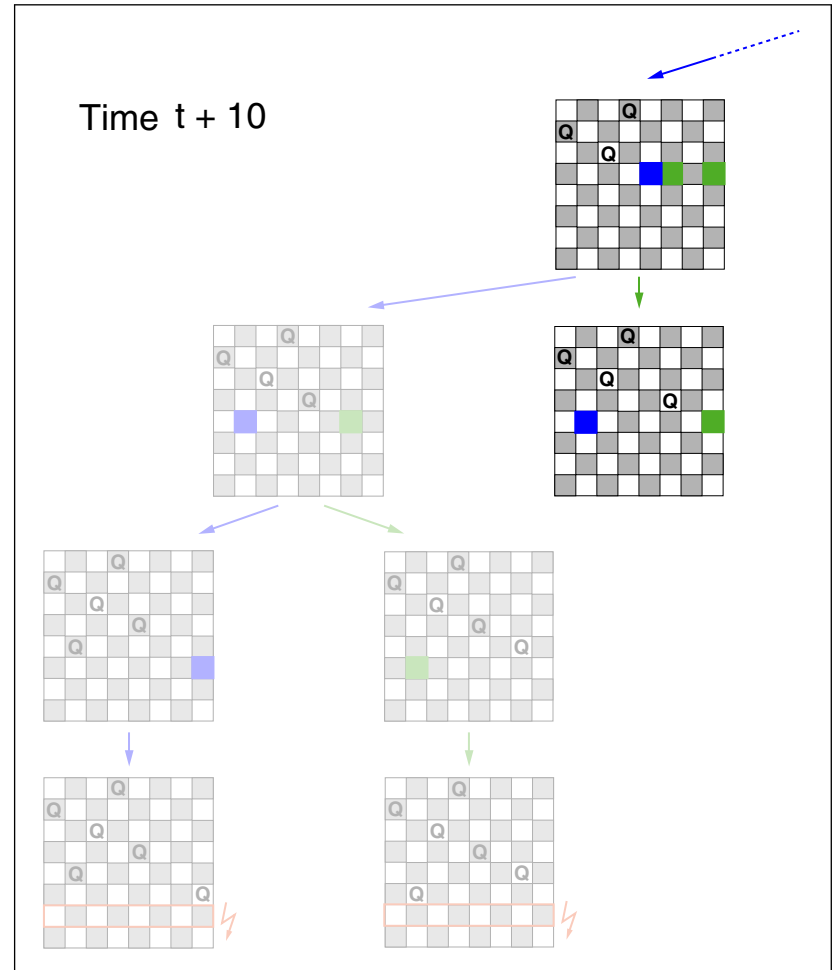
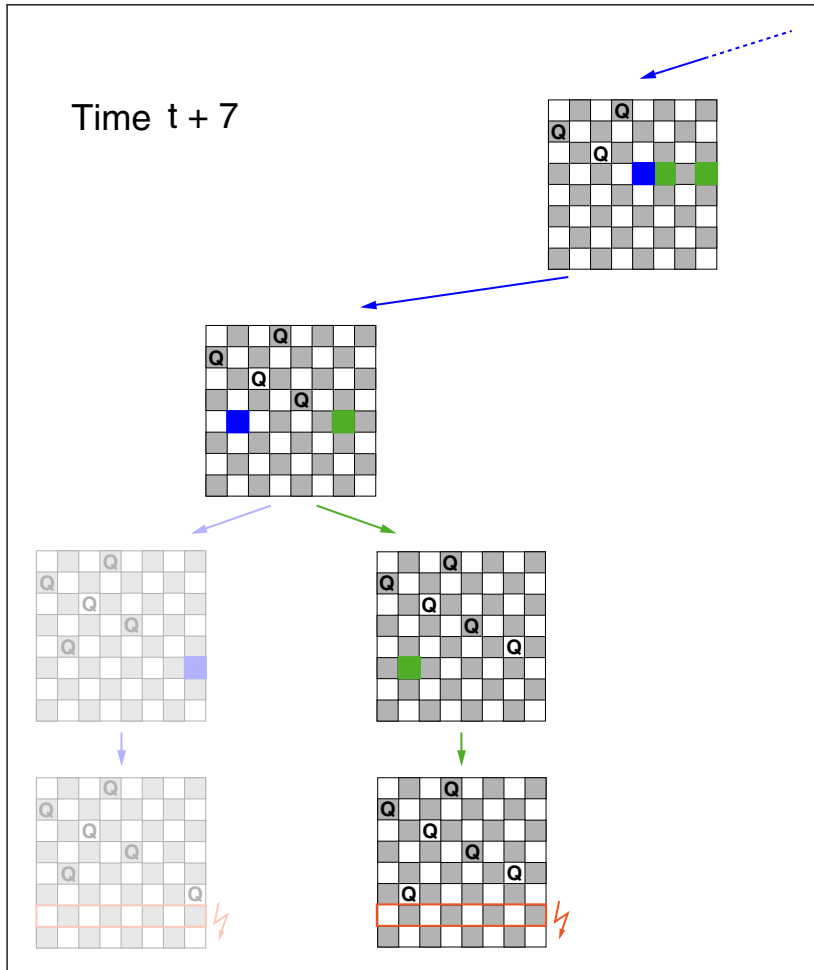


Time $t + 5$



Backtracking

Monotone Backtracking: 8-Queens Problem (continued)

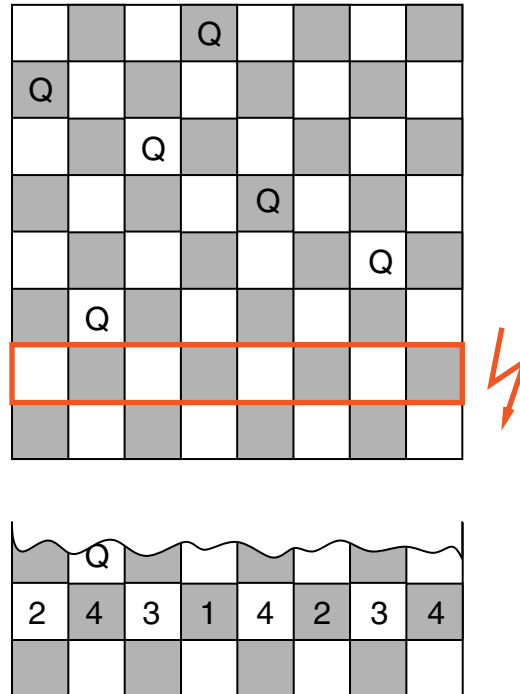


Backtracking

Non-Monotone Backtracking: 8-Queens Problem

The 8-Queens problem is handled ineffectively by monotone backtracking.

- Determine so-called “root causes” aka. “First Principles”.
- Label the attacked cells by the row number of the oldest queen:



Backtracking

Non-Monotone Backtracking: Generic

On reaching a dead end **jump back to the alleged cause** of the problem.

Concepts:

- ❑ dependency-directed backtracking
- ❑ knowledge-based backtracking

Challenges:

- ❑ Book-keeping of cause-effect chains to identify the root cause.
- ❑ Guaranteeing the principles of systematic search.
Efficient maintenance of visited and non-visited nodes. Recall that we no longer apply a monotone schedule.
- ❑ Efficient reconstruction of distant states in the search space.
Keyword: *Stack Unrolling*

Remarks:

- ❑ Dependency-directed backtracking:

If we describe states (remaining problems) by facts *“A queen is placed on cell G4.”*, *“There is a threat for cell C8.”* and rules *“If there is a queen placed on a cell, there is no threat for this cell.”*, then placing a queen allows the derivation of additional facts. Derived facts depend on facts used in the derivation. Backtracking can analyze these dependencies.

Knowledge-based backtracking:

Each step in the search can be seen as decision. There is additional knowledge available, that helps in detecting a decision that has to be revised.

- ❑ The outlined concepts and challenges lead to the research field of truth maintenance systems (TMS), which gained some popularity in the 1980s.
- ❑ Well-known TMS approaches:
 - justification-based truth maintenance system (JTMS)
 - assumption-based truth maintenance system (ATMS)

Chapter S:II

II. Basic Search Algorithms

- ❑ Systematic Search
- ❑ Graph Theory Basics
- ❑ State Space Search
- ❑ Depth-First Search
- ❑ Backtracking
- ❑ Breadth-First Search
- ❑ Uniform-Cost Search
- ❑ AND-OR Graph Basics
- ❑ Depth-First Search of AND-OR Graphs
- ❑ AND-OR Graph Search Basics
- ❑ AND-OR Graph Search

Breadth-First Search (BFS)

Breadth-first search is an **uninformed, systematic** search strategy.

BFS characteristics:

- ❑ Nodes at upper levels in G are preferred.
- ❑ Node expansion happens in levels of equal depth.
- ❑ Terminates (on locally finite graphs) with a solution, if one exists.
- ❑ Determines a goal node that is closest to the start node s , measured in the number of edges on a solution path.

Remarks:

- ❑ Operationalization of BFS:

The OPEN list is organized as a queue (i.e., nodes are explored in a FIFO (first in first out) manner). [[OPEN list in DFS](#)] [[OPEN list in BF](#)]

- ❑ By enqueueing newly generated successors in OPEN (insertion at the tail) instead of pushing them (insertion at the head), DFS is turned into BFS.

Breadth-First Search

Specification of BFS Algorithm Family

Algorithm: BFS

Input: s . Start node representing the initial state (problem) in G .
 $\text{successors}(n)$. Returns *new instances of* nodes for the successor states in G .
 $\star(n)$. Predicate that is *True* if n represents a goal state in G .
 $\text{constraints}(n)$. Predicate that is *True* if path repr. by n satisfies solution constraints.
 $\perp(n)$. Predicate that is *True* if n is a dead end.
 maxDepth . Depth-bound for nodes in G to consider ($\text{maxDepth} \geq 0$).
 visitedDepth . Returns maximum depth for nodes in G observed (call by reference).

Output: A node γ representing a solution path for s in G or the symbol *Fail*.

Specialization of Basic-OR.

Variants:

Basic-BFS: a minimal BFS version.

BFS: BFS with dead end recognition.

DL-BFS: BFS with fixed depth limit.

Compare: DFS variants.

General Information: [\[Wikipedia\]](#)

Breadth-First Search

Core Version of BFS

- ❑ Note that OPEN is organized as a queue, indicated by functions *enqueue(.)* and *front(.)*.
- ❑ Memory consumption will be controlled explicitly using the function *cleanup_closed()*.

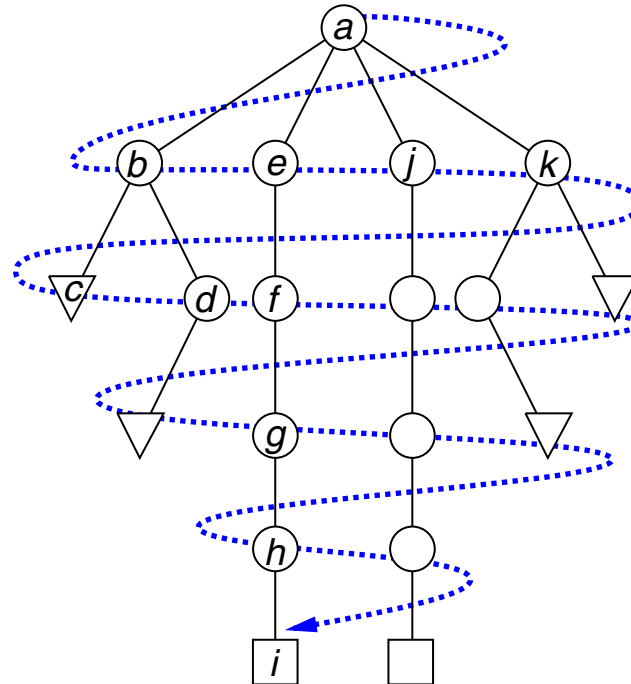
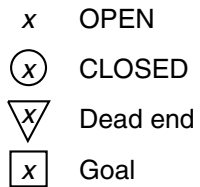
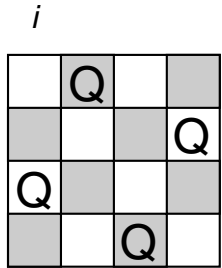
Basic-BFS(*s*, *successors*, *★*, *constraints*) // A deterministic variant of Basic-OR.

```
0.  s.parent = null;
    IF ★(s) THEN IF constraints(s) THEN RETURN(s); // Check path repr. by s.
1.  enqueue(s, OPEN); // Add node s at the end of OPEN.
2.  LOOP
3.    IF (OPEN ==  $\emptyset$ ) THEN RETURN(Fail);
4.    n = front(OPEN); remove(n, OPEN); add(n, CLOSED); // Get front node from OPEN.
5.    FOREACH n' IN successors(n) DO // Expand n.
        n'.parent = n;
        IF ★(n') THEN // Check whether n' represents a solution path.
            IF constraints(n') THEN RETURN(n');
            enqueue(n', OPEN); // Add node n' at the end of OPEN.
        ENDDO
        IF (successors(n) ==  $\emptyset$ ) THEN cleanup_closed(); // Remove dead ends.
6.  ENDLOOP
```


Breadth-First Search

Example: 4-Queens Problem

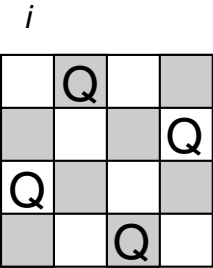
BFS node processing sequence:



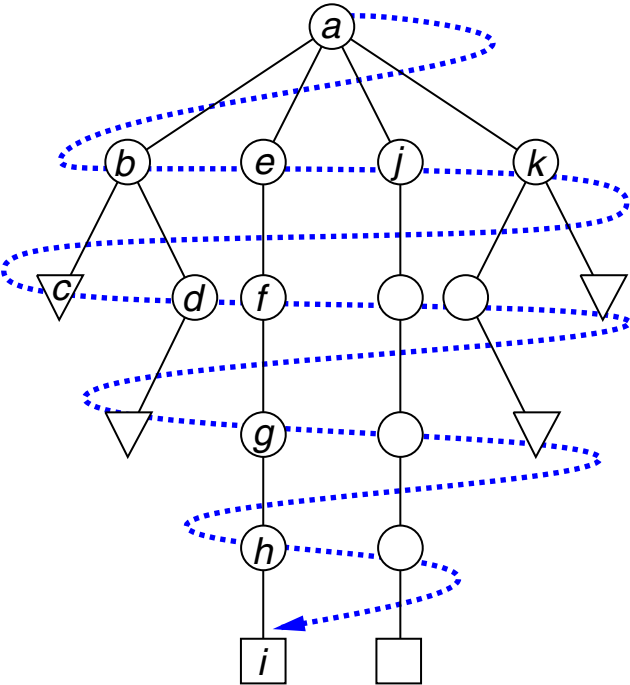
Breadth-First Search

Example: 4-Queens Problem

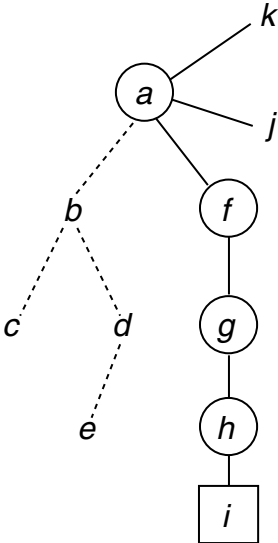
BFS node processing sequence:



- x* OPEN
- ⊙(*x*) CLOSED
- ▽(*x*) Dead end
- (*x*) Goal



Compare to the DFS strategy:



Breadth-First Search

Discussion

Basic-BFS is complete for finite state-space graphs G without cycles.

BFS issue:

- Unlike depth-first search, BFS has to store the explored part of the graph completely. Why?

DFS workarounds depth-bound and dead-end function can also limit the memory usage in BFS.

Breadth-First Search

BFS with Pruning

Information about solvability of a rest problem (state) represented by a node allows for pruning. A domain-specific predicate $\perp(n)$ (dead end function) serves this purpose.

`BFS(s, successors, \star , constraints, \perp)` // Extends Basic-BFS.

```
1.  s.parent = null; enqueue(s, OPEN);
2.  LOOP
3.    IF (OPEN ==  $\emptyset$ ) THEN RETURN(Fail);
4.    n = front(OPEN); remove(n, OPEN); add(n, CLOSED);
5.    FOREACH n' IN successors(n) DO      // Expand n.
        n'.parent = n;
        IF  $\star(n')$  THEN IF constraints(n') THEN RETURN(n');
        IF  $\perp(n')$       // Detect and remove dead ends in G.
        THEN
            add(n', CLOSED);
            cleanup_closed();    // Remove unref. nodes from CLOSED, e.g., n'.
        ELSE
            enqueue(n', OPEN);  // Add node n' at the end of OPEN.
        ENDIF
    ENDDO
    IF (successors(n) ==  $\emptyset$ ) THEN cleanup_closed();    // Remove dead ends.
6.  ENDLLOOP
```

Breadth-First Search

Depth-limited BFS

A depth limit *maxDepth* helps to avoid running out of memory, a call by reference parameter allows the interpretation of negative results ($((visitedDepth < maxDepth) \Rightarrow \text{no solution available})$).

```
DL-BFS(s, successors, ★, constraints, maxDepth, visitedDepth)    // Extends Basic-BFS.

1.  s.parent = null; enqueue(s, OPEN); depth(s) = 0; visitedDepth = 0;
2.  LOOP
3.    IF (OPEN == ∅) THEN RETURN(Fail);
4.    n = front(OPEN); remove(n, OPEN); add(n, CLOSED);
      IF (depth(n) > visitedDepth) THEN visitedDepth = depth(n);
5.    IF (depth(n) >= maxDepth)    // Do not expand nodes of highest depth.
      THEN
        cleanup_closed();    // Remove unref. nodes from CLOSED, e.g., n.
      ELSE
        FOREACH n' IN successors(n) DO    // Expand n.
          n'.parent = n;
          depth(n') = depth(n) + 1;
          IF ★(n') THEN IF constraints(n') THEN RETURN(n');
          enqueue(n', OPEN);    // Add node at the front of OPEN.
        ENDDO
        IF (successors(n) == ∅) THEN cleanup_closed();    // Remove dead ends.
      ENDIF
6.  ENDLOOP
```

Breadth-First Search

Corollary 13 (Termination of Basic-BFS for Finite Graphs without Cycles)

Basic-BFS terminates for finite cycle-free graphs G with $Prop_0(G)$ properties.

(See termination and completeness of Basic-DFS.)

Lemma 14 (Completeness of Basic-BFS)

Basic-BFS is complete for graphs G with $Prop_0(G)$ properties.

Proof (sketch)

1. Proof by induction:

When a node on OPEN represents a solution base of length $k + 2$ for some $k \in \mathbb{N} \cup \{0\}$, then all solution bases for s of at most length k have been considered by Basic-BFS.

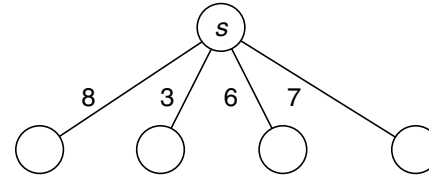
2. Let $P_{s-\gamma}$ be a solution path in G of length k . Since G has $Prop_0(G)$, the set of solution bases for s with a length at most $k + 1$ is finite.
3. At each point in time an initial part of $P_{s-\gamma}$ is available as a solution base represented by a node in OPEN.
4. Since in each run through the main loop a node (solution base) is removed from OPEN, Basic-BFS must terminate and return a solution.

Breadth-First Search

Example: BFS for Optimization (see Basic-OR and DFS for Optimization)

Determine the minimum column sum of a matrix:

8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6

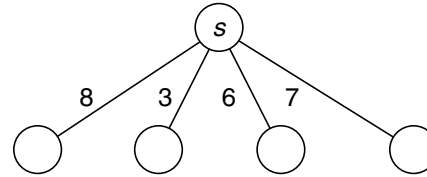


Breadth-First Search

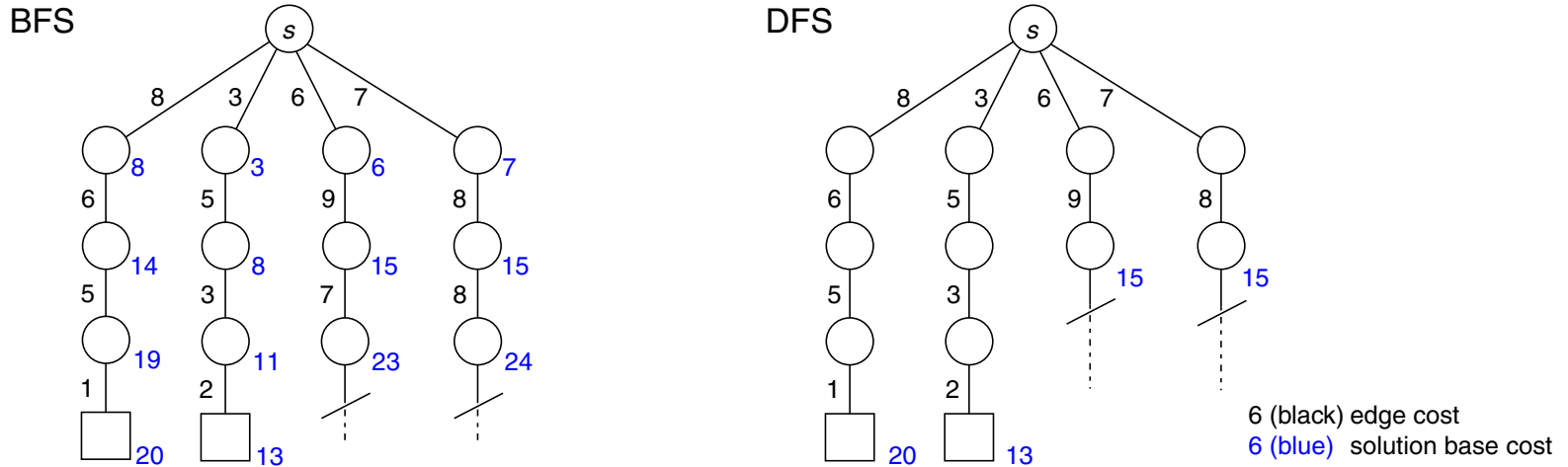
Example: BFS for Optimization (see Basic-OR and DFS for Optimization)

Determine the minimum column sum of a matrix:

8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6



Comparison of BFS and DFS for optimization with pruning:



Prerequisite for pruning:

Objective function *column sum* increases monotonically with the depth of a path.

Solution bases exceeding cost of a cheapest solution found so far can be pruned.