Chapter S:II (continued)

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

©STEIN/LETTMANN 1998-2018

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 is preferred.

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

  If a node is explored (= reached), *all* of its direct successors $n_1, \ldots, n_k$ are generated at once. Among the $n_1, \ldots, n_k$ *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.

©STEIN/LETTMANN 1998-2018

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 BFS] [OPEN list UCS]

- The depth in trees is a naturally defined: The most recently generated node is also a deepest node.

Depth-First Search

Algorithm: DFS

Input:
s. Start node representing the initial problem.
successors(n). Returns the successors of node n.
$\star(n)$. Predicate that is *True* if n represents a solution path.
$\perp(n)$. Predicate that is *True* if n is a dead end.
k. Depth-bound.

Output: A goal node or the symbol *Fail*.

Depth-First Search [Basic_OR_Search] [BT]

```
DFS($s$, $successors$, $\star$      )    // Basic version.

  1.  $push(s, \text{OPEN})$;
  2.  LOOP
  3.    IF ($\text{OPEN} = \emptyset$) THEN RETURN($Fail$);
  4.    $n = pop(\text{OPEN})$;    // Select and remove front element in OPEN.
        $push(n, \text{CLOSED})$;
  5.

          FOREACH $n'$ IN $successors(n)$ DO    // Expand $n$.
            $add\_backpointer(n', n)$;
            IF $\star(n')$ THEN RETURN($n'$);
            $push(n', \text{OPEN})$;



          ENDDO
          IF ($successors(n) = \emptyset$) THEN $cleanup\_closed()$;

  6.  ENDLOOP
```

Remarks:

- The start node s is usually not a goal node. If this possibility shall be explicitly allowed, change the first line as follows:

```
1.  IF $\star(s)$ THEN RETURN($s$); push($s$, OPEN);
```

- The function *cleanup_closed* deletes nodes from the CLOSED list that are no longer required. It is based on the following principles:

  1. Nodes that fulfill $\perp$ (= that are dead ends) can be discarded.
  2. When a node n is discarded, check if n has predecessors that are still part of a solution path. A node is part of a solution path if it has a successor on the OPEN list.
     Those predecessors that are not part of a solution path (= that have no successor on OPEN) can be discarded.

  As a consequence, 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 $n$.
          add_backpointer($n'$, $n$);
          ...
```

Depth-First Search [Basic_OR_Search] [BT]

```
DFS($s$, $successors$, $\star$ , $k$)    // Basic version with depth bound.

  1.  push($s$, OPEN);
  2.  LOOP
  3.    IF (OPEN $= \emptyset$) THEN RETURN(Fail);
  4.    $n$ = pop(OPEN);    // Select and remove front element in OPEN.
        push($n$, CLOSED);
  5.    IF (depth($n$) $= k$)
        THEN cleanup_closed()
        ELSE
          FOREACH $n'$ IN successors($n$) DO    // Expand $n$.
            add_backpointer($n'$, $n$);
            IF $\star(n')$ THEN RETURN($n'$);
            push($n'$, OPEN);

          ENDDO
          IF (successors($n$) $= \emptyset$) THEN cleanup_closed();
        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 is violated even for large finite graphs.

  Instead of using a high depth bound it is advisable to increase the bound gradually:

```
WHILE $k < K$ DO
  $result$ = DFS($s$, $successors$, $\star$, $k$);
  IF ($result$ IS $Fail$)
  THEN
    $k = 2 \cdot k$;
  ELSE
    ...    // Do something with the solution found.
  ENDIF
ENDDO
```

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

Depth-First Search [Basic_OR_Search] [BT]

```
DFS($s$, $successors$, $\star$, $\perp$, $k$)   // Basic version with depth bound + dead end test.

 1.  push($s$, OPEN);
 2.  LOOP
 3.    IF (OPEN $= \emptyset$) THEN RETURN(Fail);
 4.    $n$ = pop(OPEN);   // Select and remove front element in OPEN.
       push($n$, CLOSED);
 5.    IF (depth($n$) $= k$)
       THEN cleanup_closed()
       ELSE
         FOREACH $n'$ IN successors($n$) DO   // Expand $n$.
           add_backpointer($n'$, $n$);
           IF $\star(n')$ THEN RETURN($n'$);
           push($n'$, OPEN);
           IF $\perp(n')$
           THEN
             pop(OPEN);
             cleanup_closed();
           ENDIF
         ENDDO
         IF (successors($n$) $= \emptyset$) THEN cleanup_closed();
       ENDIF
 6.  ENDLOOP
```

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 $\perp(n) =$ *False*
- ❑ Example definition of a very simple dead end predicate (its look-ahead is 0) :

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

- ❑ 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

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 $\perp(n) =$ *True*, trigger a jump (backtrack) to the deepest alternative not violating the depth bound.

Depth-First Search

Discussion (continued)

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

(a) We are given plenty of equivalent solutions.

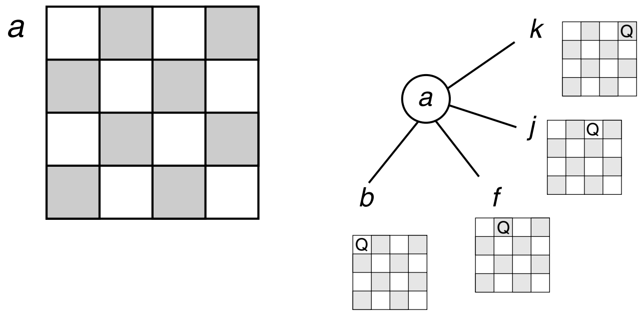
(b) Dead ends can be recognized early (i.e., with a considerable look-ahead).

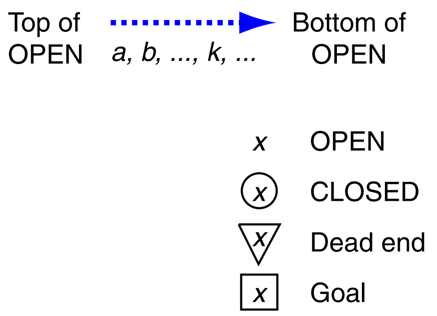
(c) There are no cyclic or infinite paths resp. cyclic or infinite paths can be avoided.

©STEIN/LETTMANN 1998-2018

Depth-First Search

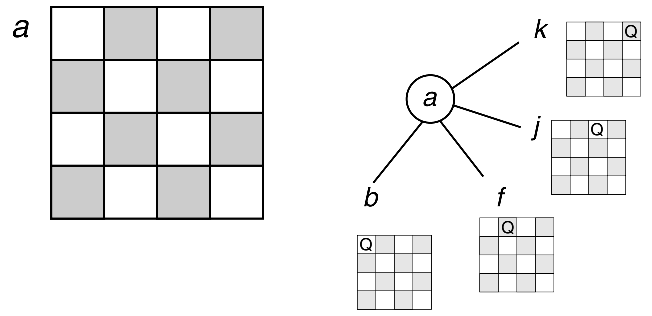
Example: 4-Queens Problem

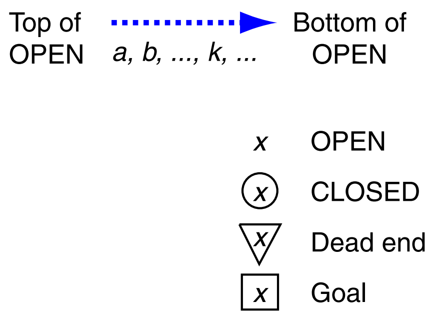


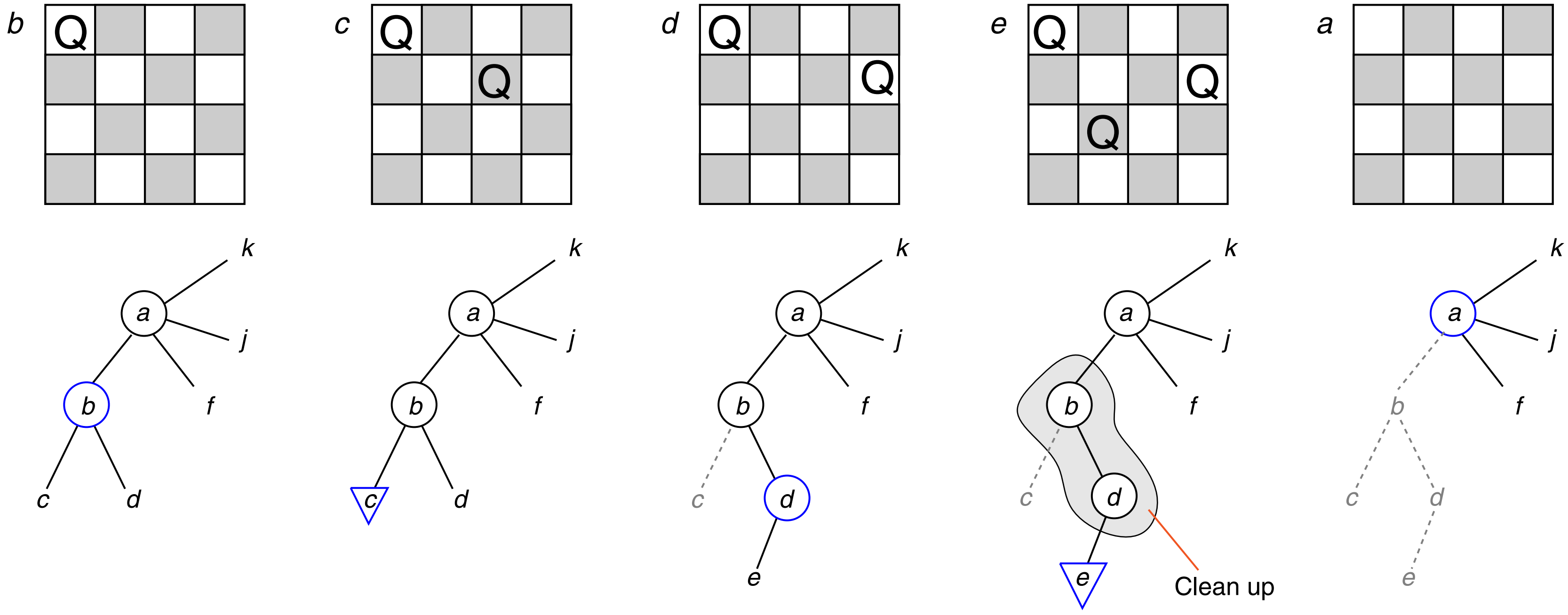


Depth-First Search

Example: 4-Queens Problem (continued)



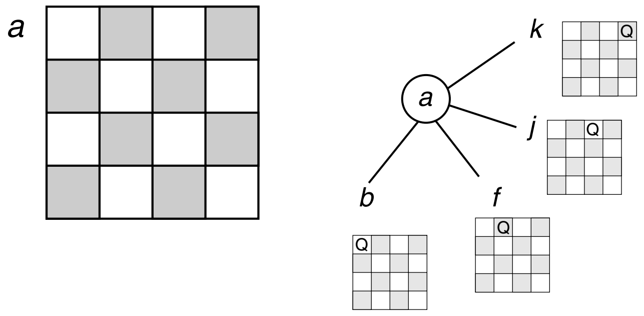


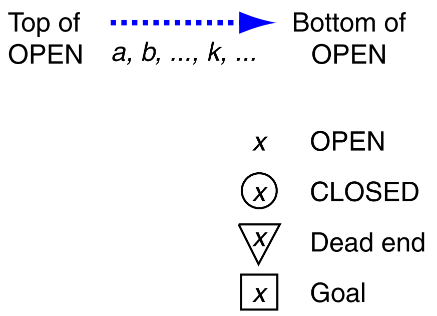


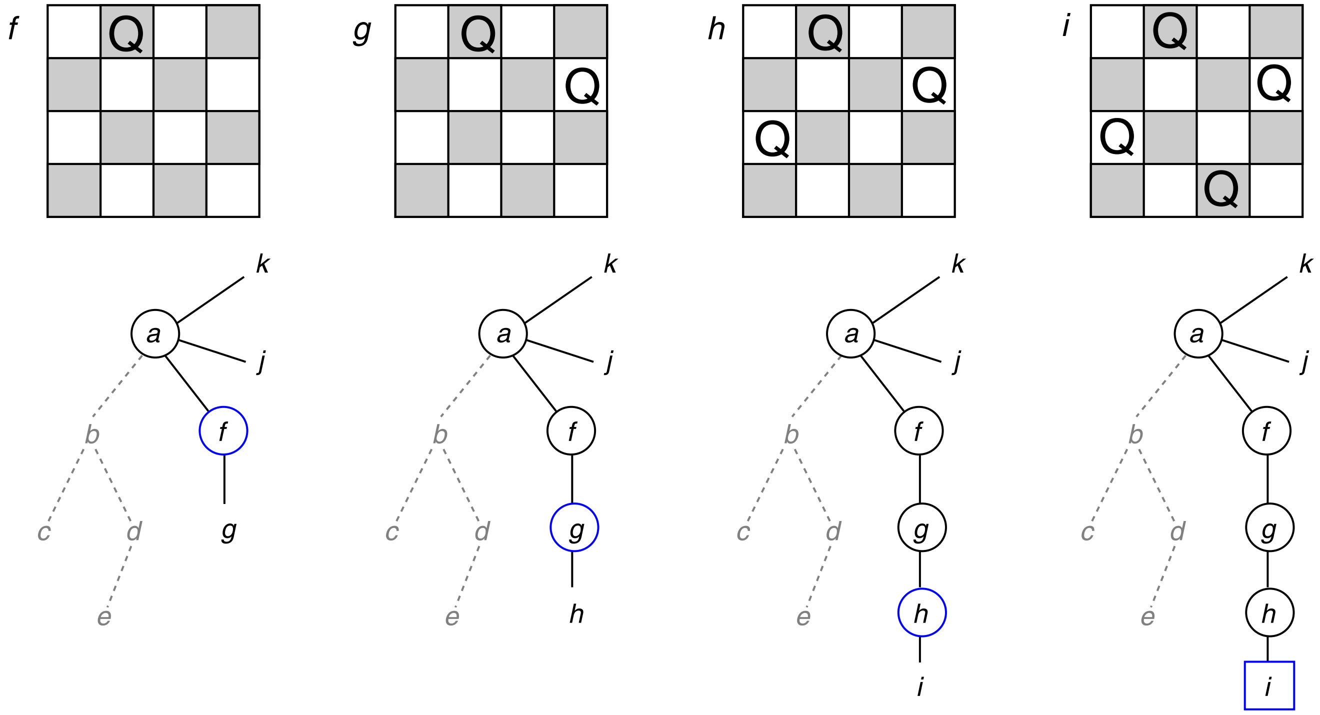
©STEIN/LETTMANN 1998-2018

Depth-First Search

Example: 4-Queens Problem (continued)



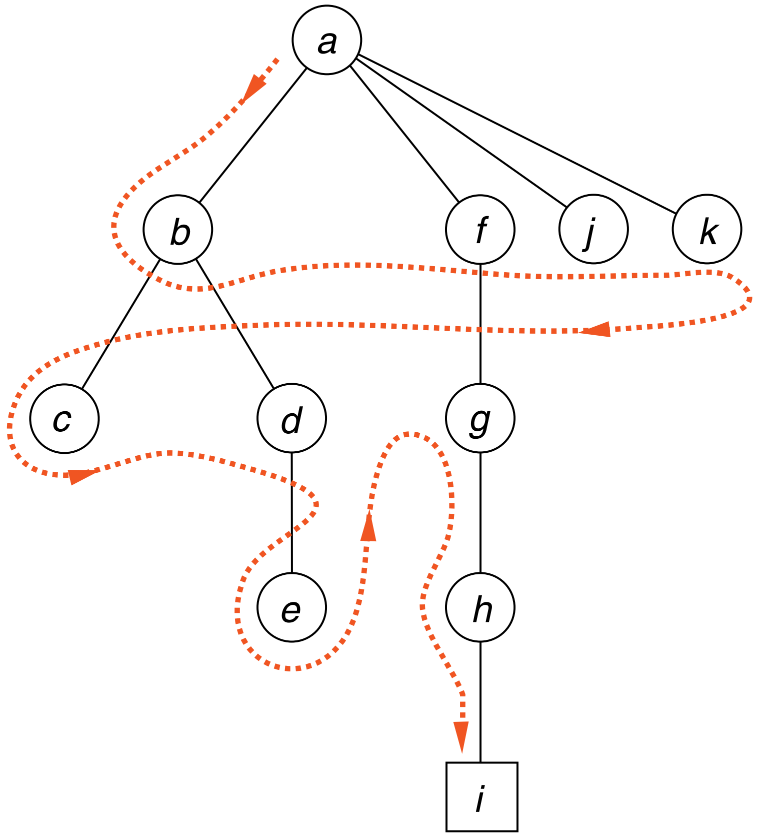




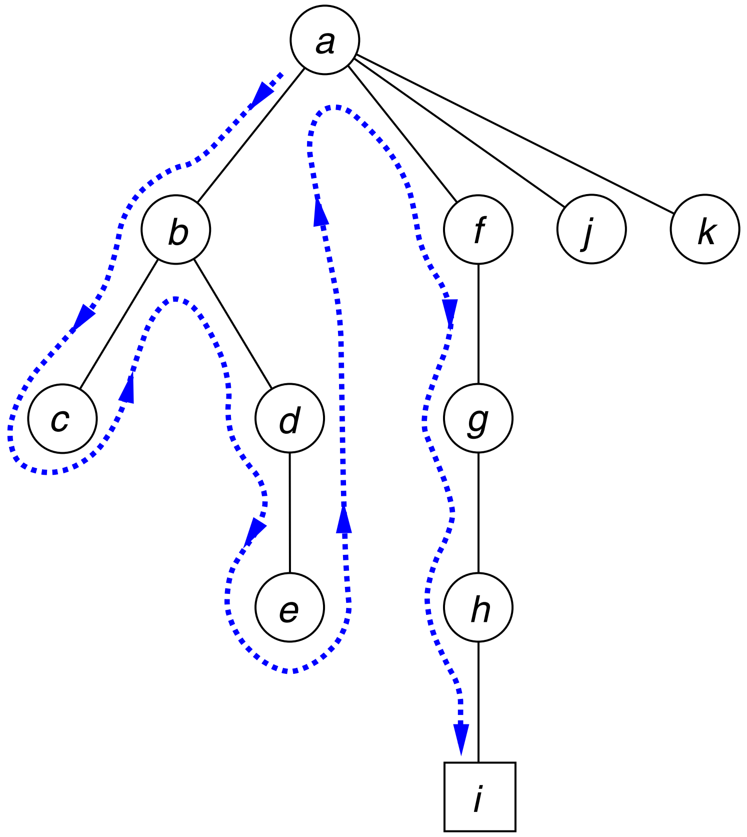
Depth-First Search

Example: 4-Queens Problem (continued)

DFS node processing sequence:



Node generation



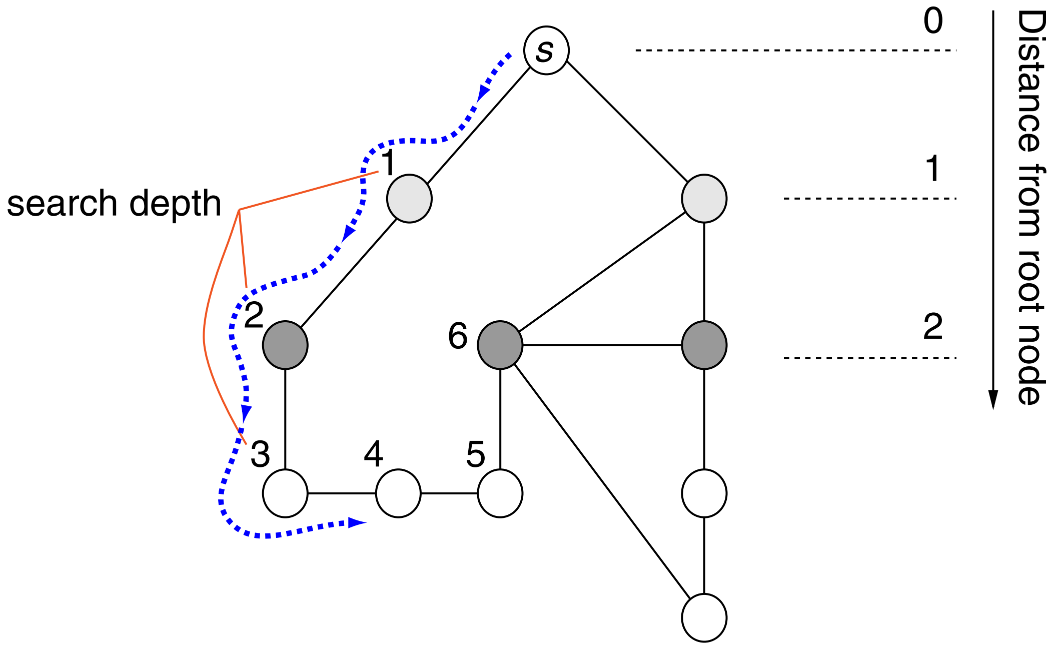
Node expansion

©STEIN/LETTMANN 1998-2018

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*$(\hat{n})$, where $\hat{n}$ is the highest parent node of n in the search space graph G (= parent node that is nearest to s). So, 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.

Remarks (continued) :

- 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 instantiations of a node n can be available at the same time in the 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 in $\perp(n)$ whether an instantiation of n is currently available in OPEN or CLOSED.

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

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., $\perp(n') =$ *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 problem. *next_successor*(n). Returns the next successor of node n. *expanded*(n). Predicate that is *True* if n is expanded. $\star(n)$. Predicate that is *True* if n represents a solution path. $\perp(n)$. Predicate that is *True* if n is a dead end. k. Depth-bound.
Output:	A goal node or the symbol *Fail*.

©STEIN/LETTMANN 1998-2018

Backtracking [Basic_OR_Search] [DFS]

```
BT($s$, $\textit{next\_successor}$, $\textit{expanded}$, $\star$, $\bot$, $k$)

  1.  $\textit{push}(s, \texttt{OPEN})$;
  2.  LOOP
  3.    IF ($\texttt{OPEN} = \emptyset$) THEN RETURN($\textit{Fail}$);
  4.    $n$ = $\textit{top}(\texttt{OPEN})$;     // Select front element in OPEN, no removing.
  5.    IF (($\textit{depth}(n) = k$) OR $\textit{expanded}(n)$)
        THEN $\textit{pop}(\texttt{OPEN})$;
        ELSE
          $n' = \textit{next\_successor}(n)$;     // Get a new successor of $n$.
          $\textit{add\_backpointer}(n', n)$;
          IF $\star(n')$ THEN RETURN($n'$);
          $\textit{push}(n', \texttt{OPEN})$;
          IF $\bot(n')$ THEN $\textit{pop}(\texttt{OPEN})$;
        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.

©STEIN/LETTMANN 1998-2018

Backtracking

Discussion (continued)

Backtracking versus depth-first search:

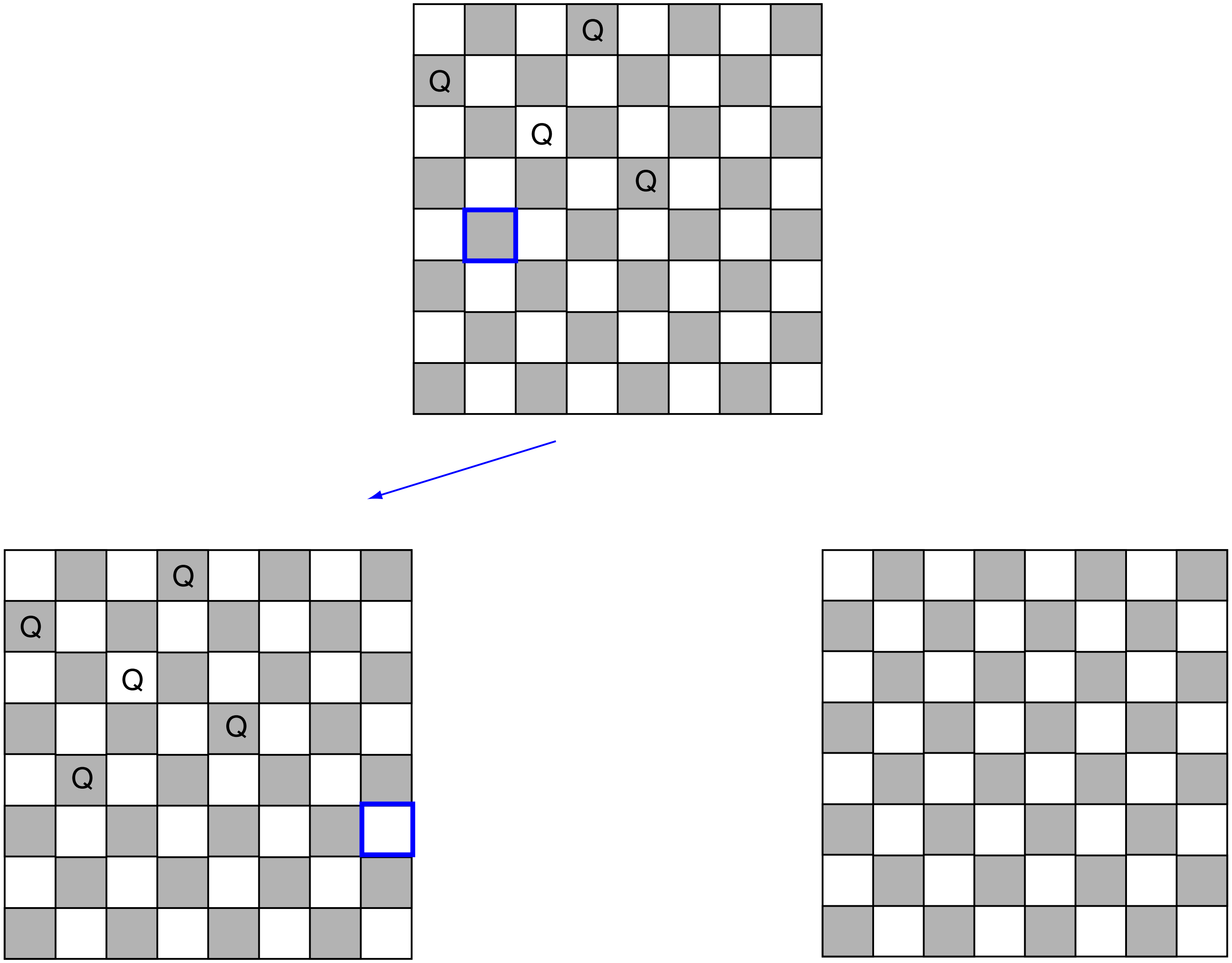
- Backtracking requires only an OPEN list, which serves as both LIFO stack and traversal path storage.

  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 expanded.

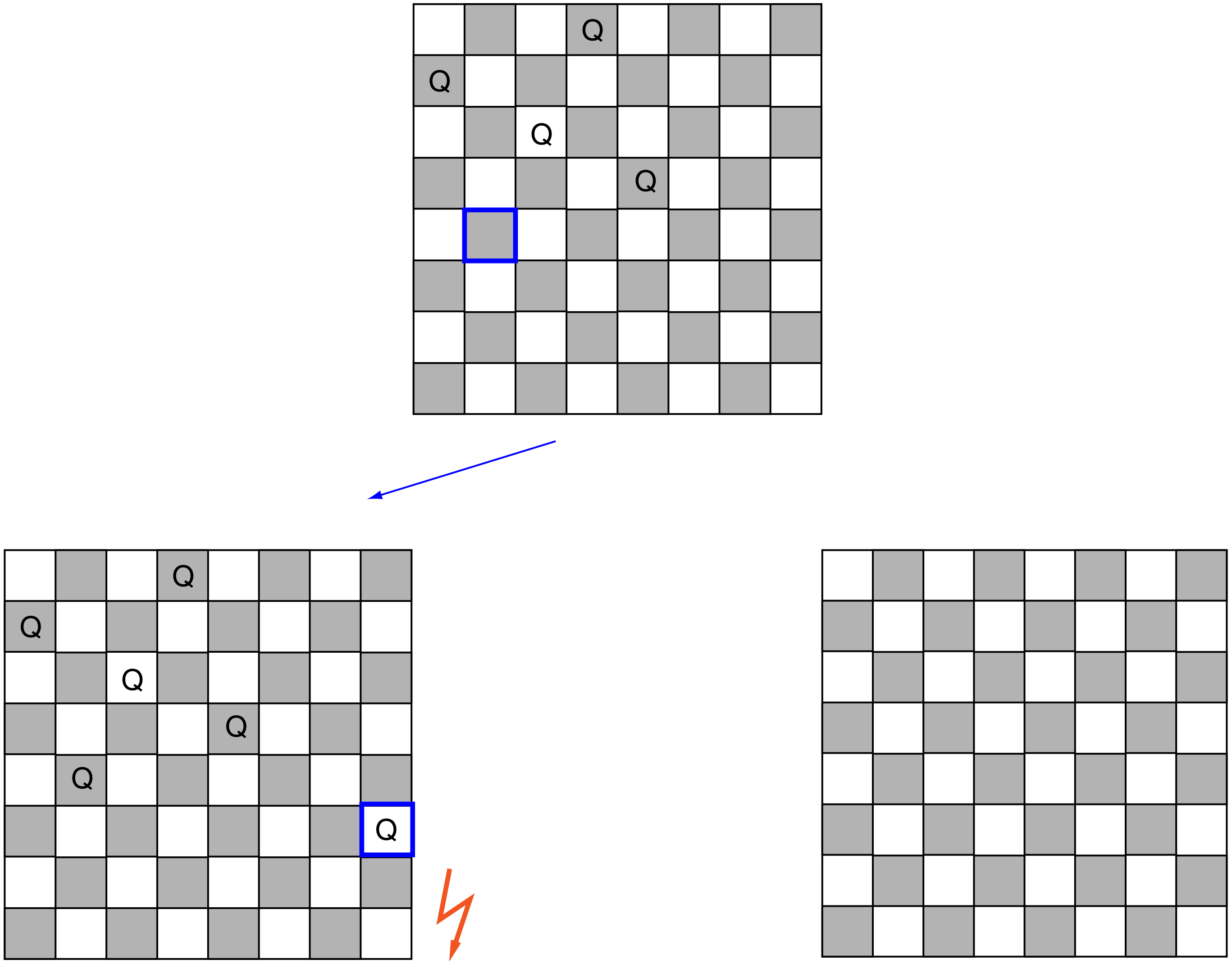
Backtracking

Monotone Backtracking: 8-Queens Problem



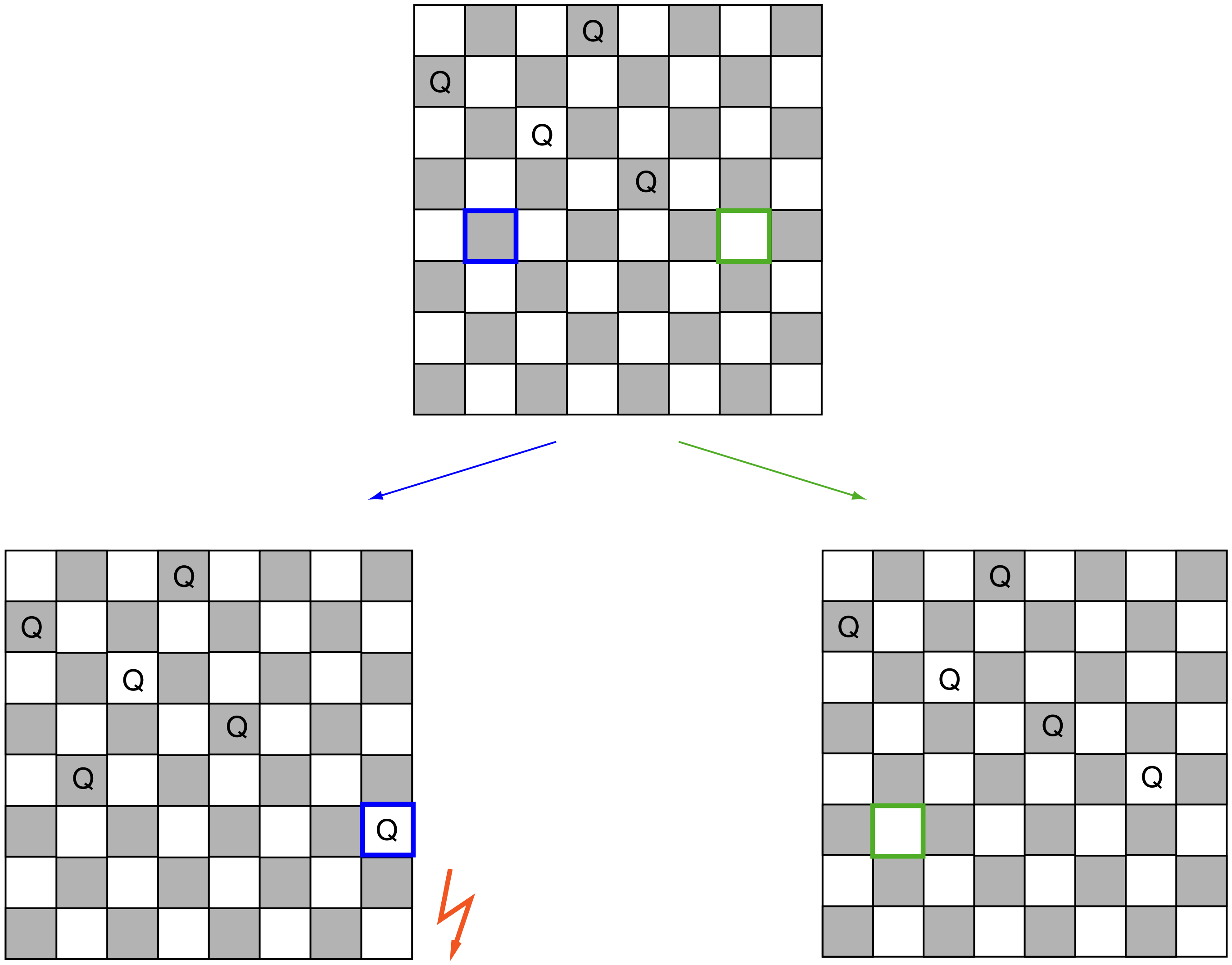
Backtracking

Monotone Backtracking: 8-Queens Problem (continued)



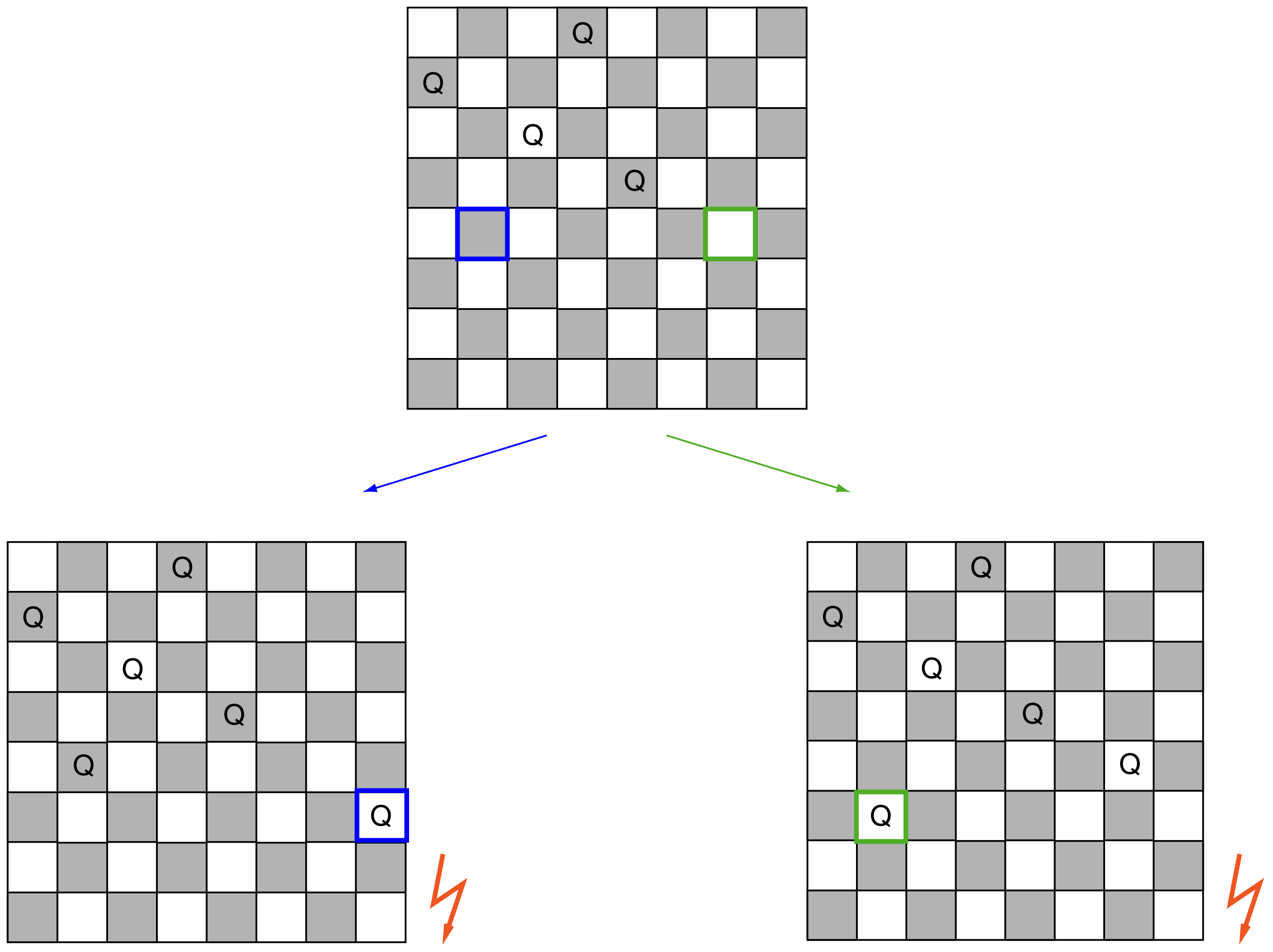
Backtracking

Monotone Backtracking: 8-Queens Problem (continued)



Backtracking

Monotone Backtracking: 8-Queens Problem (continued)



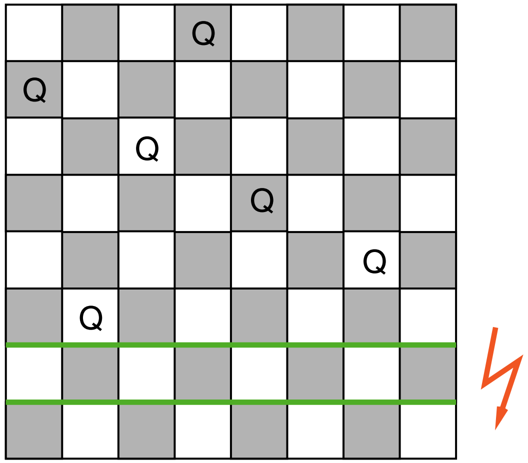
©STEIN/LETTMANN 1998-2018

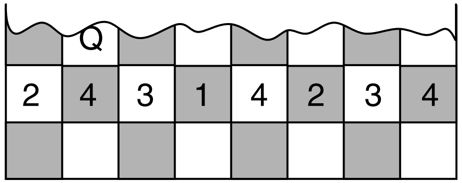
Backtracking

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.
- ❑ Guarantee 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:

- The outlined concepts and challenges lead to the research field of truth maintenance systems (TMS).

- Well-known TMS approaches:
  - justification-based truth maintenance system (JTMS)
  - assumption-based truth maintenance system (ATMS)

Backtracking

Example: Backtracking for Optimization (see `Basic_OR_Search` for Optimization)

Determine the minimum column sum of a matrix:

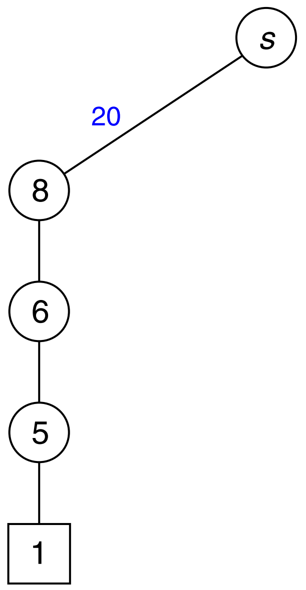
8	3	6	7
6	5	9	8
5	3	7	8
1	2	4	6

Backtracking

Example: Backtracking for Optimization (see `Basic_OR_Search` 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



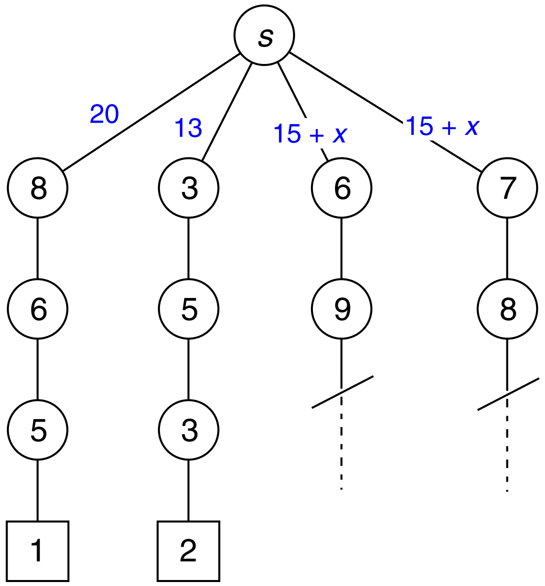
©STEIN/LETTMANN 1998-2018

Backtracking

Example: Backtracking for Optimization (see `Basic_OR_Search` 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:

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

Breadth-First Search (BFS) [UCS]

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 DFS] [OPEN list UCS]
- By enqueuing 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

Algorithm:	BFS
Input:	s. Start node representing the initial problem. *successors*(n). Returns the successors of node n. $\star(n)$. Predicate that is *True* if n is a goal node. $\perp(n)$. Predicate that is *True* if n is a dead end.
Output:	A goal node or the symbol *Fail*.

Breadth-First Search

```
BFS($s$, $successors$, $\star$, $\perp$)

1. $push(s, \text{OPEN})$;
2. LOOP
3.   IF ((OPEN = $\emptyset$) OR $memory\_exhausted()$) THEN RETURN($Fail$);
4.   $n = pop(\text{OPEN})$;
     $push(n, \text{CLOSED})$;
5.   FOREACH $n'$ IN $successors(n)$ DO    // Expand $n$.
       $set\_backpointer(n', n)$;
       IF $\star(n')$ THEN RETURN($n'$);
       $enqueue(n', \text{OPEN})$;    // Insert node at the end of OPEN.




     ENDDO
6. ENDLOOP
```

Breadth-First Search

```
BFS($s$, *successors*, $\star$, $\bot$)

  1. push($s$, OPEN);
  2. LOOP
  3.   IF (($\text{OPEN} = \emptyset$) OR memory_exhausted()) THEN RETURN(Fail);
  4.   $n$ = pop(OPEN);
       push($n$, CLOSED);
  5.   FOREACH $n'$ IN successors($n$) DO     // Expand $n$.
         set_backpointer($n'$, $n$);
         IF $\star(n')$ THEN RETURN($n'$);
         enqueue($n'$, OPEN);    // Insert node at the end of OPEN.
         IF $\bot(n')$
         THEN
           remove_last(OPEN);
           cleanup_closed();
         ENDIF
       ENDDO
  6. ENDLOOP
```

Breadth-First Search

Discussion

BFS issue:

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

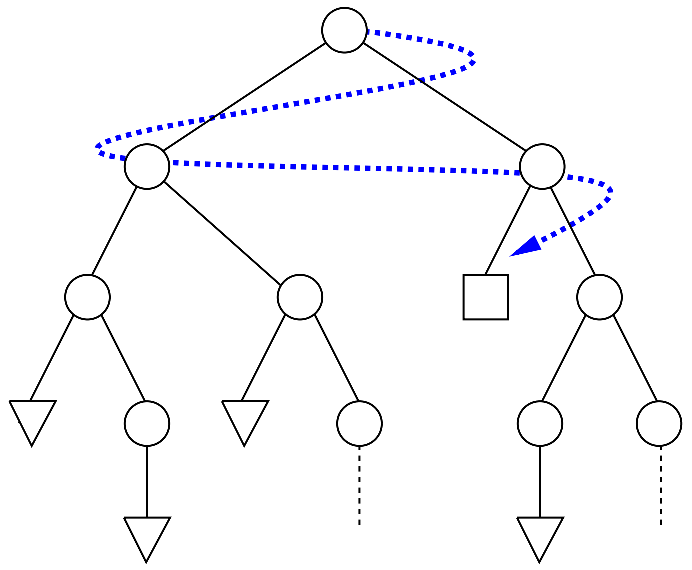
Breadth-First Search

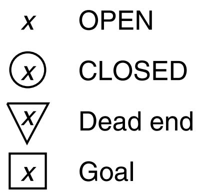
Discussion

BFS issue:

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

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

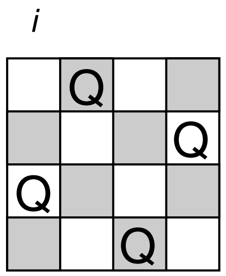


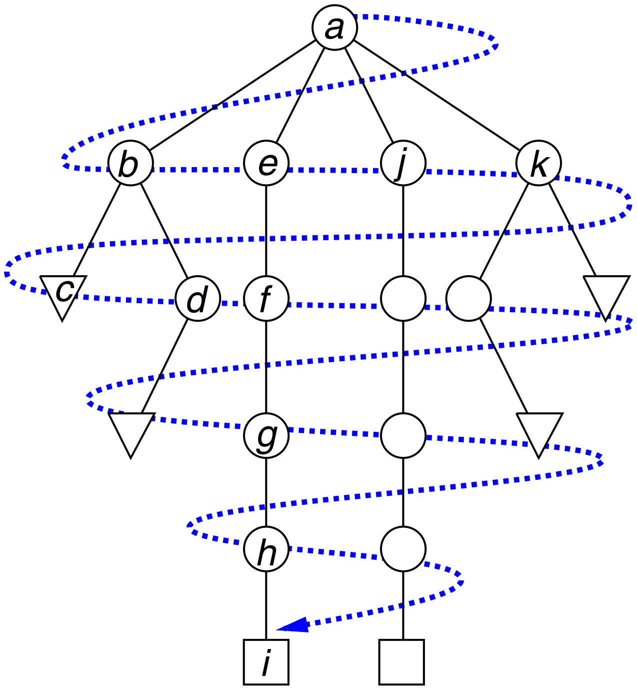


Breadth-First Search

Example: 4-Queens Problem

BFS node processing sequence:





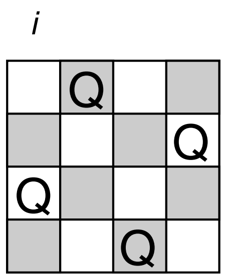
x	OPEN
$\bigcirc$ *x*	CLOSED
▽ *x*	Dead end
$\boxed{x}$	Goal

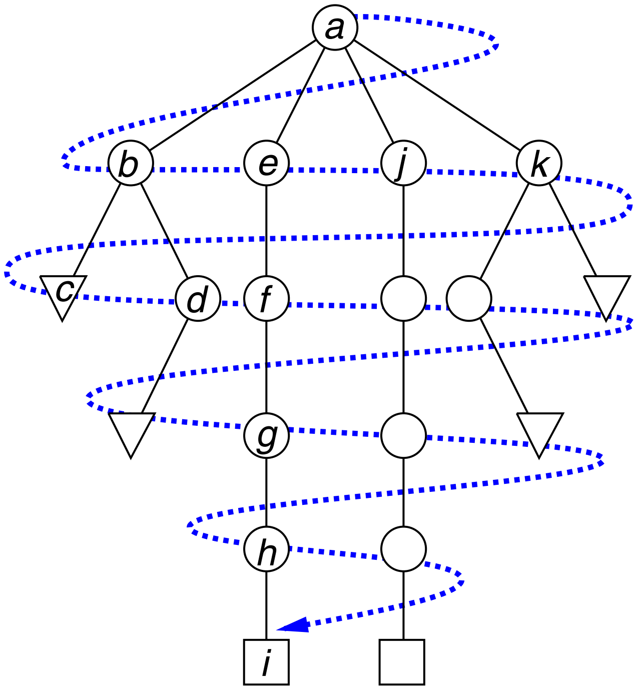
©STEIN/LETTMANN 1998-2018

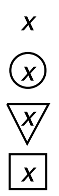
Breadth-First Search

Example: 4-Queens Problem

BFS node processing sequence:

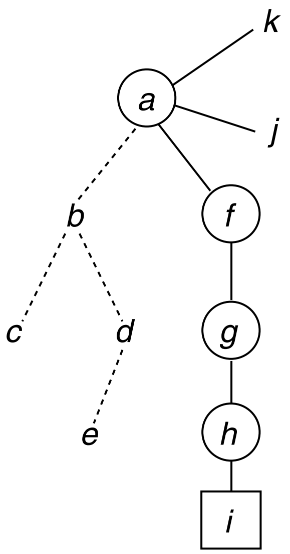






x	OPEN
ⓧ	CLOSED
▽	Dead end
$\boxed{x}$	Goal

Compare to the DFS strategy:



©STEIN/LETTMANN 1998-2018

Uniform-Cost Search

Uniform-Cost Search for Optimization (see `Basic_OR_Search` for Optimization)

Setting:

- The search space graph contains several solution paths.
- Cost values are assigned to solution paths and solution bases.

Task:

- Determine a cheapest path from s to some goal $\gamma \in \Gamma$.

Approach:

- Continue search with the cheapest solution base in OPEN.

Prerequisite:

- The cost of a solution base is a lower bound for the cheapest solution cost that can be achieved by completing the solution base.

Remarks:

- ❑ Operationalization of uniform-cost search: BFS with some modifications.
  Cost values are stored with the nodes. The OPEN list is organized as a heap, and nodes are explored wrt. the cheapest cost. [OPEN list DFS] [OPEN list BFS]
- ❑ Uniform-cost search is also called cheapest-first search.

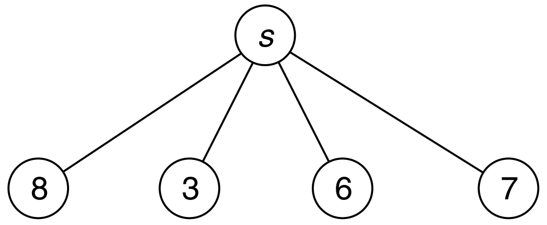
© STEIN/LETTMANN 1998-2018

Uniform-Cost Search

Uniform-Cost Search for Optimization: Example

Determine the minimum column sum of a matrix:

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



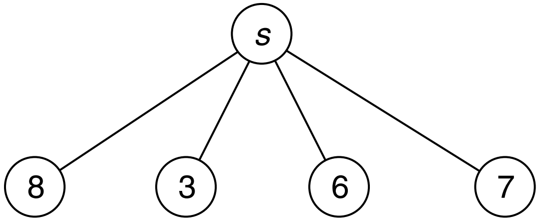
©STEIN/LETTMANN 1998-2018

Uniform-Cost Search

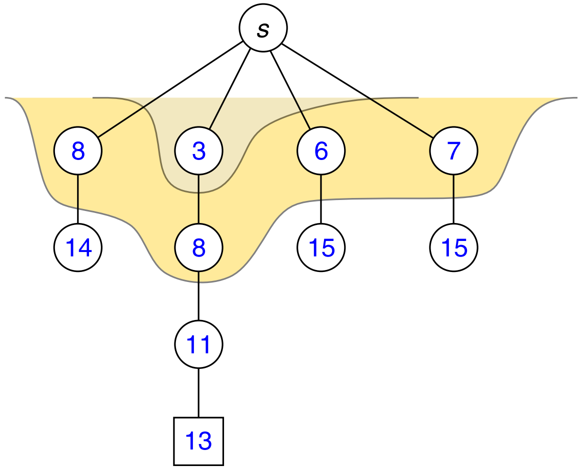
Uniform-Cost Search for Optimization: Example

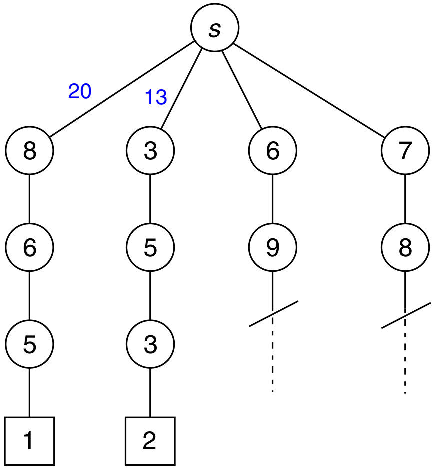
Determine the minimum column sum of a matrix:

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



Comparison of uniform-cost search (left) and BT/DFS (right):





Uniform-Cost Search [BFS]

Uniform-cost search is an uninformed (systematic) search strategy.

Uniform-cost search characteristics:

- ❑ Node expansion happens in levels of equal costs:

  A node n with cost $c(n)$ will not be expanded as long as a non-expanded node n' with $c(n') < c(n)$ resides on the OPEN list.

$\approx$ Application of the BFS strategy to solve optimization problems (using cost instead of depth).

Remarks:

- A more general cost concept is to assign cost values to edges in search space graphs. A path's cost can be calculated as the sum or as the maximum of the cost values of its edges.

  If edge cost values are limited to non-negative numbers, the path cost of a solution base is an optimistic estimate of a cheapest solution path cost achievable by continuing that solution base.