Chapter S:II

II. Basic Search Algorithms

- Systematic Search
- Graph Search Basics
- Depth-First Search
- Backtracking
- Breadth-First Search
- Uniform-Cost Search

© STEIN/LETTMANN 1998-2017

Systematic Search

Types of Problems

Each of the illustrated problems defines a search space S comprised of objects called solution candidates:

- board configurations,
- move sequences,
- travel tours.

In particular, the desired solution is in S.

Systematic Search

Types of Problems

Each of the illustrated problems defines a search space S comprised of objects called solution candidates:

- board configurations,
- move sequences,
- travel tours.

In particular, the desired solution is in S.

Distinguish two problem types:

1. Constraint satisfaction problems.
   A solution has to fulfill constraints and shall be found with minimum search effort.

2. Optimization problems.
   A solution has to fulfill constraints and stands out among all other candidates with respect to a special property.

Remarks:

- For example, a constraint for a solution would be that it costs C maximum.
- We require a function $\star($*solution_candidate*$)$which can test whether the constraints for a solution are fulfilled.
- We will use solution cost as optimization criterion.
- Q. Is it possible to pose the 8-queens problem as (special case of) an optimization problem?

Systematic Search

Search Building Blocks

Given a search space S with solution candidates. Then *problem solving* means to find an object with specific properties within S.

Prerequisites to “algorithmize” problem solving:

1. Encoding.

   A symbol structure or code that can represent both fully-specified and partially-specified solution candidates in S.

2. Operators.

   A mechanism for completing the encodings of partially-specified solution candidates by code transformations.

3. Control strategy.

   A mechanism to schedule (= order) transformations.

   Objective is to maximize effectiveness: find the desired object as quickly as possible.

Systematic Search

Definition 1 (Systematic Control Strategy)

Given a search space S with solution candidates. A control strategy is called *systematic* if

(a) all objects in S are considered,

(b) each objects in S is considered only once.

A search that employs a systematic control strategy is called *systematic search*.

Condition (a) implies completeness, condition (b) implies efficiency.

Remarks:

- Of course, condition (a) of a systematic search is particularly important if solutions are rare. No part of the search space should be excluded a priori.

Systematic Search

Definition 2 (State Transition System)

A *state transition system* is quadruple, $\mathcal{T} = (S, T, s, F)$, consisting of

- S, a set of states,
- $T \subseteq S \times S$, a transition relation,
- $s \in S$, a start state, and
- $F \subseteq S$, a set of final states.

A state s_1 is *reachable* from s_0 iff either $s_0 = s_1$ or there is a state s' such that s' is reachable from s_0 and $(s', s_1) \in T$.

Observation

- Transitions (s_0, s_1) are entirely local. A transition can be used to effect a state change regardless of which transition to the s_0 state resulted or which transitions will be used to change s_1.

- The reachability relation on S is the transitive closure of T.

  A justification for the fact that a state can be reached from another is the disclosure of a finite sequence of transitions that achieves this (e.g. given as sequence of intermediate states).

Systematic Search

Modeling as Reachability Problem for STS

Systems and processes can be modeled as state transition systems, whereby specific questions and related knowledge can be captured.

States	are certain situations (states) of a system or a process that can be characterized by unique descriptions.
Transitions	are state changes of a system or a process that are initiated by some rules or operations or actions.
Final States	are certain states of a system or a process, not necessarily terminal states.
Problem	is to decide for state s whether a final state can be reached under certain constraints (solution constraints).
Solution	is then a suitable sequence of transitions.
Candidates	are finite sequences of states starting in s.
Search Space	is the set of all solution candidates.

Systematic Search

Modeling as Reachability Problem for STS

Systems and processes can be modeled as state transition systems, whereby specific questions and related knowledge can be captured.

States	are certain situations (states) of a system or a process that can be characterized by unique descriptions.
Transitions	are state changes of a system or a process that are initiated by some rules or operations or actions.
Final States	are certain states of a system or a process, not necessarily terminal states.
Problem	is to decide for state s whether a final state can be reached under certain constraints (solution constraints).
Solution	is then a suitable sequence of transitions.
Candidates	are finite sequences of states starting in s.
Search Space	is the set of all solution candidates.

© STEIN/LETTMANN 1998-2017

Systematic Search

Modeling as Reachability Problem for STS

Systems and processes can be modeled as state transition systems, whereby specific questions and related knowledge can be captured.

States	are certain situations (states) of a system or a process that can be characterized by unique descriptions.
Transitions	are state changes of a system or a process that are initiated by some rules or operations or actions.
Final States	are certain states of a system or a process, not necessarily terminal states.
Problem	is to decide for state s whether a final state can be reached under certain constraints (solution constraints).
Solution	is then a suitable sequence of transitions.
Candidates	are finite sequences of states starting in s.
Search Space	is the set of all solution candidates.

Remarks:

- The simplest constraint for a solution is "no constraint", i.e. any sequence of transitions from the start node to a goal node is acceptable.
- Some sources define state transition systems by states and transitions, only. The difference is whether we see s and F as part of the setting or as part of the reachability question.
- In general, the modeling of real world problems as state transition systems requires discretization of continuous transitions and simplification of dependencies.
- A state description can also be considered as encoding of a problem description. Differences to descriptions of final states define the problem of reaching a final state from this state. Accordingly, states reached by (a sequence of) transitions encode remaining problems, final states encode solved problems.
- In theoretical computer science, reachability problems occur in many contexts, e.g. the halting problem for turing machines or the problem whether a grammar for a formal language generates any terminal strings at all.
- Q. What can state transition systems look like for the problems from the introduction part?

Graph Search Basics

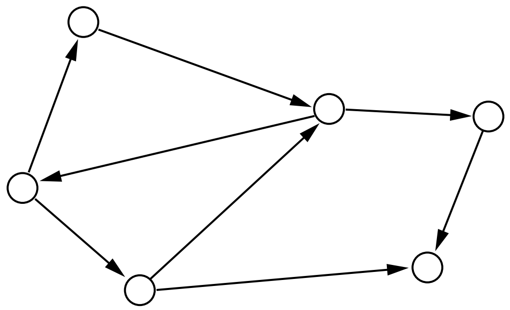
Definition 3 (Directed Graph)

A directed graph G is a tuple $\langle V, E\rangle$, where V denotes a nonempty set and $E \subseteq V \times V$ denotes a set of pairs.

The elements $n \in V$ are called nodes, the elements $e = (n, n') \in E$ are called directed (from n to n') edges, links, or arcs.

Given two nodes n, n' in V that are connected with an edge $e = (n, n')$, then the node n' is called direct successor or son of n; the node n is called direct predecessor, parent, or father of n'.

Given an edge $e = (n, n')$, then the nodes n, n' are adjacent, and the node-edge combination n, e and n', e are called incident respectively.



©STEIN/LETTMANN 1998-2017

Graph Search Basics

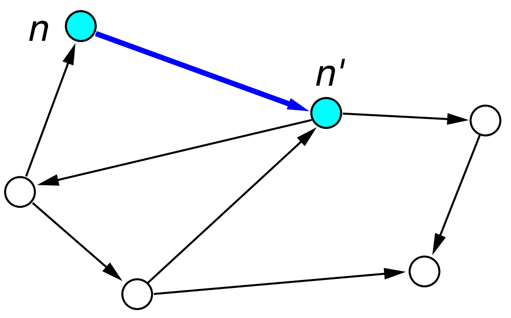
Definition 3 (Directed Graph)

A directed graph G is a tuple $\langle V, E \rangle$, where V denotes a nonempty set and $E \subseteq V \times V$ denotes a set of pairs.

The elements $n \in V$ are called nodes, the elements $e = (n, n') \in E$ are called directed (from n to n') edges, links, or arcs.

Given two nodes n, n' in V that are connected with an edge $e = (n, n')$, then the node n' is called direct successor or son of n; the node n is called direct predecessor, parent, or father of n'.

Given an edge $e = (n, n')$, then the nodes n, n' are adjacent, and the node-edge combination n, e and n', e are called incident respectively.



Remarks:

- ❑ Graphs are not restricted to be finite, i.e. graphs can have an infinite set of nodes V and, in this case, an infinite set of edges E as well.
- ❑ Here, we restrict ourselves to simple graphs, i.e. we do not allow multiple edges connecting a node n to a successor node n' in a graph as it may occur in multigraphs. If multiple edges are needed, they can be simulated by introducing unique intermediate nodes in such edges.

Graph Search Basics

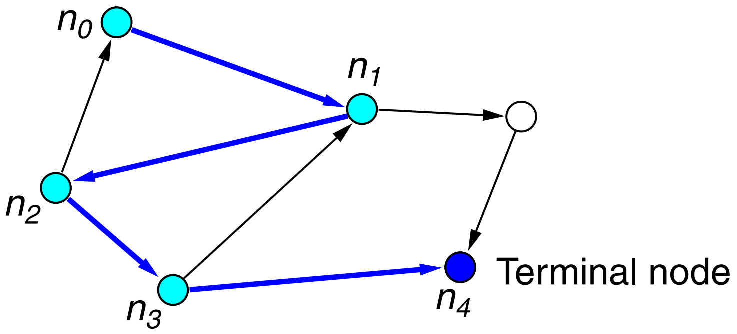
Definition 4 (Path, Node Types, Outdegree)

Let $G = (V, E)$ be a directed graph.

A sequence of nodes, $P = (n_0, n_1, \ldots, n_k)$, where each n_i, $i = 1, \ldots, k$, is direct successor of n_{i-1}, is called directed path or trail of length k from node n_0 to the node n_k.

The nodes $n_1, \ldots, n_k$ are called successors or descendants of the node n_0. The nodes $n_0, \ldots, n_{k-1}$ are called predecessors or ancestor of the node n_k. A node without a successor is called a terminal node.

The number b of all direct successors of a node n is called its outdegree. A graph G is called locally finite iff ($\leftrightarrow$) the outdegree of each node in G is finite.



© STEIN/LETTMANN 1998-2017

Graph Search Basics

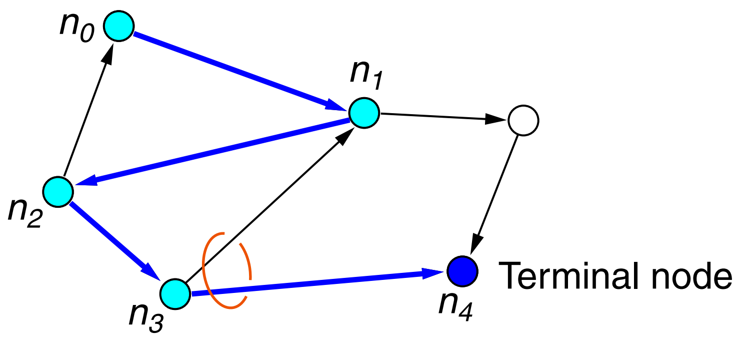
Definition 4 (Path, Node Types, Outdegree)

Let $G = (V, E)$ be a directed graph.

A sequence of nodes, $P = (n_0, n_1, \ldots, n_k)$, where each n_i, $i = 1, \ldots, k$, is direct successor of n_{i-1}, is called directed path or trail of length k from node n_0 to the node n_k.

The nodes $n_1, \ldots, n_k$ are called successors or descendants of the node n_0. The nodes $n_0, \ldots, n_{k-1}$ are called predecessors or ancestor of the node n_k. A node without a successor is called a terminal node.

The number b of all direct successors of a node n is called its outdegree. A graph G is called locally finite iff ($\leftrightarrow$) the outdegree of each node in G is finite.



©STEIN/LETTMANN 1998-2017

Remarks:

- ❑ If startnode and endnode of a path $(n_0, n_1, \ldots, n_k)$ are of interest, we denote such a path by $P_{n_0-n_k}$.
- ❑ Some sources define paths as special type of walks.

  A walk is an alternating sequence of vertices and edges, starting and ending at a vertex, in which each edge is adjacent in the sequence to its two endpoints. [Wikipedia]

  As we are dealing with simple graphs, the edges traversed are immediately clear from the sequence of nodes.
- ❑ As there is no uniform terminology for paths, we explicitly state that multiple occurrences of nodes are allowed in our context. So, paths can be cyclic.
- ❑ The concept of a path can be extended to cover infinite paths by defining them as an infinite sequence of nodes $(n_0, n_1, n_2 \ldots)$. An infinite path has no endnode.
- ❑ The outdegree of a graph is the maximum of the outdegrees of its nodes.

  The fact that a graph is locally finite does not entail that its outdegree is finite.

Graph Search Basics

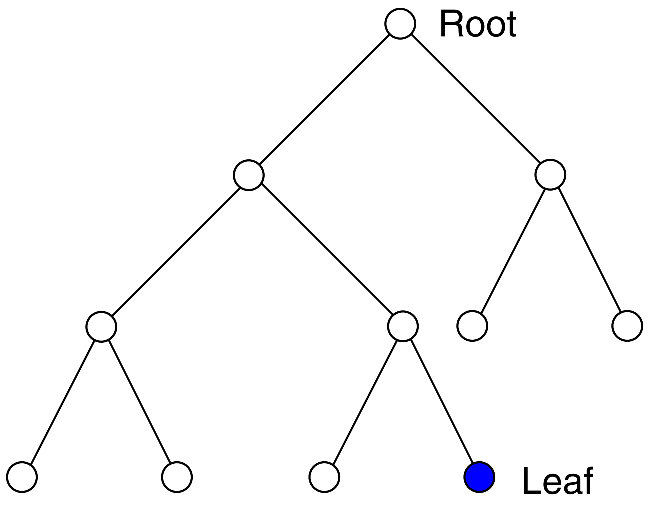
Definition 5 (Directed Tree, Uniform Tree)

Let $G = \langle V, E \rangle$ be a directed graph. G is called a directed tree with root $s \in V$, if $|E| = |V| - 1$ and if there is a directed path from s to each node in V.

The length of a path from s to some node $v \in V$ is called the depth of v.

The terminal nodes of a tree are also called leaf nodes, or leafs for short.

A uniform tree of depth h is a tree where each node of depth smaller than h has the same degree, and where all nodes of depth h are leaf nodes.



©STEIN/LETTMANN 1998-2017

Remarks:

- For trees, the indication of the edge direction can be omitted since all edges are equally oriented, from the root node toward the leaf nodes.

Graph Search Basics

Modeling as Path-Seeking Problem

- ❑ Systems and processes can be modeled as state transition systems.
- ❑ A state transition system $\mathcal{T} = (S, T, s, F)$ defines a graph (S, T) of possible transitions.

➜ Deciding, whether a final state is reachable from s, is a path-seeking problem in the graph (S, T).

Nodes	represent states of a system or a process.
Edges	represent state transitions of a system or a process.
Goal Nodes	represent final states.
Problem	is to decide for node s whether a goal node is reachable under certain constraints (solution constraints).
Solution	is then a suitable path from s to a goal node.
Candidates	are finite paths starting from s.
Search Space	is the set of all solution candidates.

Graph Search Basics

Modeling as Path-Seeking Problem (continued) [Problem-Reduction]

Definition 6 (State-Space, State-Space Graph)

Given a state transition system $\mathcal{T} = (S, T, s, F)$, the set of states S is called *state-space*. The graph $G = (S, T)$ defined over S by T is called *state-space graph*.

Naming conventions:

- ❑ The links in a state-space graph define alternatives how the problem at a parent node can be simplified. The links are called OR links.
- ❑ Nodes with only outgoing OR links are called OR nodes.
- ❑ The state-space graph is an OR graph, i.e., a graph that contains only OR links.

Modeling

- ❑ The possible states of a system or a process form the state space.
- ❑ The possible state transitions form the state-space graph.

© STEIN/LETTMANN 1998-2017

Remarks:

- ❑ A state-space graph is a variant of the more general concept of a search space graph.
- ❑ If state transitions in a system or a process result from the application of an operation, rule, or action, the edge in the state-space graph can be labeled accordingly.

© STEIN/LETTMANN 1998-2017

Graph Search Basics

Modeling as Path-Seeking Problem (continued) [Problem-Reduction]

Definition 7 (Solution Path, Solution Base, State-Space Search)

Let G be a state-space graph containing a node n, and let γ, $\gamma \in G$ be a goal node. Also, let some solution constraints be given. Then a path P in G from n to γ satisfying these constraints is called *solution path for* n. A path P in G from n to some node n' in G is called *solution base for* n.

A search for a solution graph in a state space graph is called *state-space search*.

Usage:

- Solution Paths

  We are interested in finding a solution path P for the start node s in G.

- Solution Bases

  Algorithms maintain and extend a set of promising solution bases until a solution path is found.

Problem

➜ How to search for a solution path in an infinite state-space graph?

© STEIN/LETTMANN 1998-2017

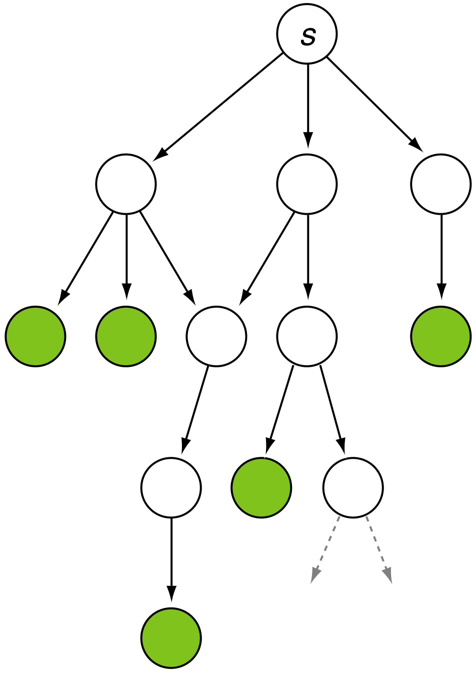
Remarks:

- ❑ Obviously, all solution paths contained in G are also solution bases.
- ❑ While solution paths are often discussed with respect to the underlying state-space graph, solution bases are only considered within a finite subgraph of the state-space graph which was explored so far by an algorithm.
- ❑ If P is a solution base for a node n in G and if n' is some node in P, then the subpath P' of P starting in n' and ending in the endnode of P is a solution base for a node n' in G. This subpath P' is sometimes called the *solution base in* P *induced by* n'.

Graph Search Basics

Illustration of Solution Paths and Solution Bases

State-space graph:



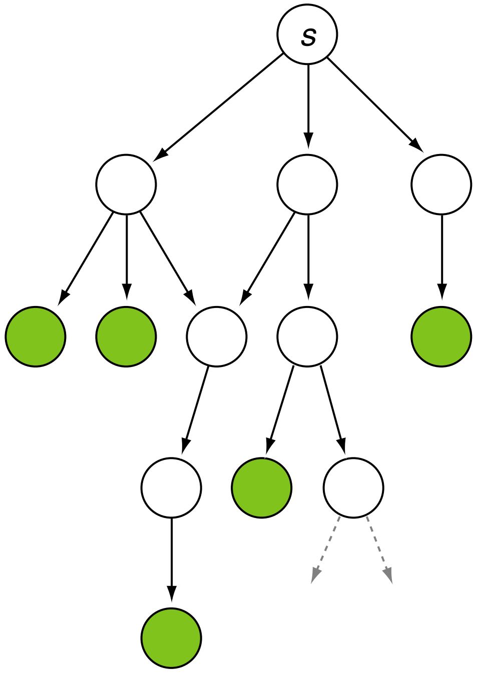
● Goal node (solved rest problem)

© STEIN/LETTMANN 1998-2017

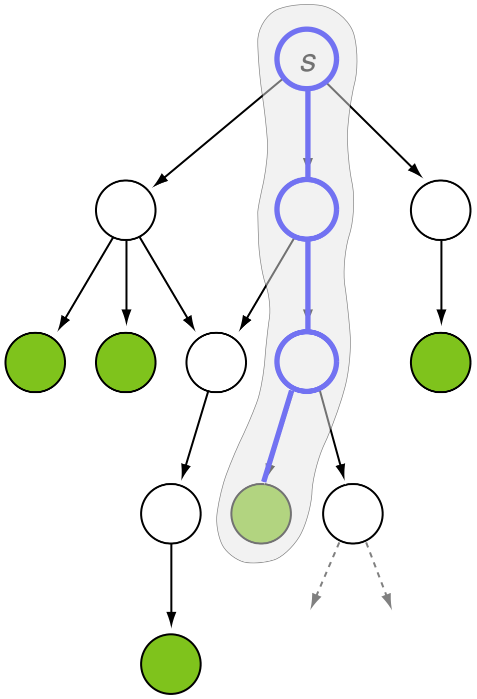
Best-First Search for State-Space Graphs

Illustration of Solution Paths and Solution Bases

State-space graph:



Solution path for node *s*:



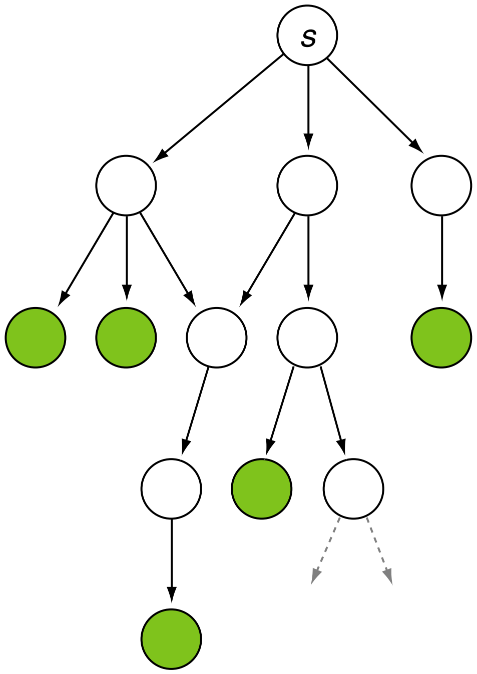
● Goal node (solved rest problem)

© STEIN/LETTMANN 1998-2017

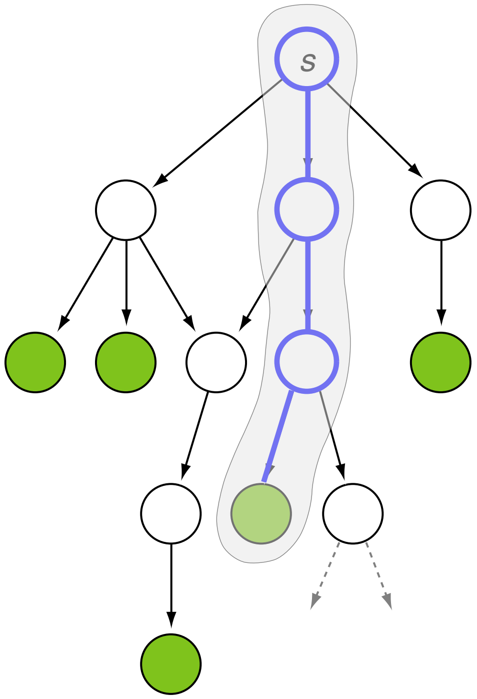
Best-First Search for State-Space Graphs

Illustration of Solution Paths and Solution Bases

State-space graph:

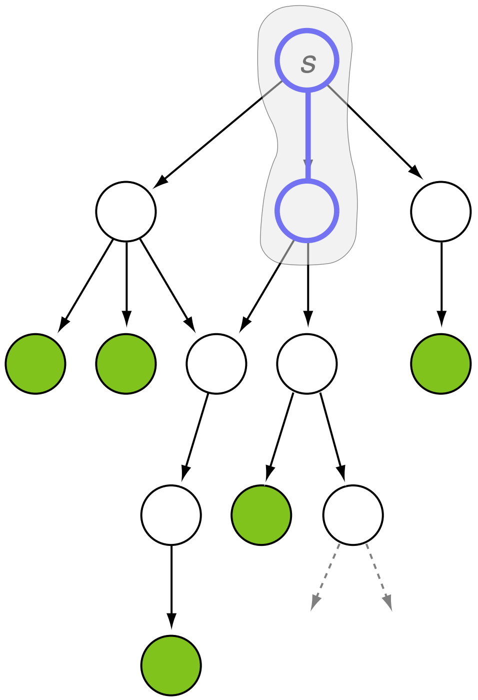


Solution path for node s:



● Goal node (solved rest problem)

Solution base for *s*:



Graph Search Basics

Graph Representation

Explicit Graph (Representation):

- A graph $G = (V, E)$ is specified by listing all nodes in V and all edges in E explicitly.

➜ Algorithms using an explicit graph representation can handle finite graphs, only.

Implicit Graph (Representation):

- A graph G is specified by listing a set S of start nodes and giving a function *successors*(n) returning the set of direct successors of node n in G.

- A graph G is specified by listing a set S of start nodes and giving a function *next_successor*(n) returning an unseen direct successor of node n in G.

➜ Algorithms using an implicit graph representation can handle graphs with finite set S and computable function *successors*(n) resp. *next_successor*(n).

We restrict ourselves to singleton sets S, i.e. we have a single start node s.

© STEIN/LETTMANN 1998-2017

Remarks:

- Function *next_successor*(n) generates the direct successors of n in G one by one. In order to simplify notation and algorithms, we assume that a function *next_successor*(n) itself stores which successors have been generated so far.

- An explicit representation of a graph G can be determined from its implicit representation in the following way.

$$\begin{aligned}
V_0 &:= S \\
V_{i+1} &:= \{n' \mid n' \in \textit{successors}(n), n \in V_i\} \\
V &:= \bigcup_{i=0}^{\infty} V_i \\
E_0 &:= \emptyset \\
E_{i+1} &:= \{(n, n') \mid n' \in \textit{successors}(n), n \in V_i\} \\
E &:= \bigcup_{i=0}^{\infty} E_i
\end{aligned}$$

  Then $G = (V, E)$.

- Obviously, there are graphs for which no implicit representations with finite set S of start nodes exist. Examples are graphs with infinite node set, but empty edge set.

- Let $G = (V, E)$ be an explicitly defined graph and let graph G' be implicitly defined by a start node $s \in V$ and an appropriate function *successors*(n). In general, these two graphs will not be the same. Without further notice, we will restrict ourselves to the subgraph of G induced by the set of nodes that are reachable from s.

Graph Search Basics

Definition 8 (Node Generation)

Let G be a graph, implicitly defined by a start node s and a function *next_successor*(n). Let A be an algorithm working on G, given s and *next_successor*(n).

Applying function *next_successor*(n) and thereby generating an unseen direct successors of a node is called *node generation*.

Definition 9 (Node Expansion)

Let G be a graph, implicitly defined by a start node s and a function *successors*(n). Let A be an algorithm working on G, given s and *successors*(n).

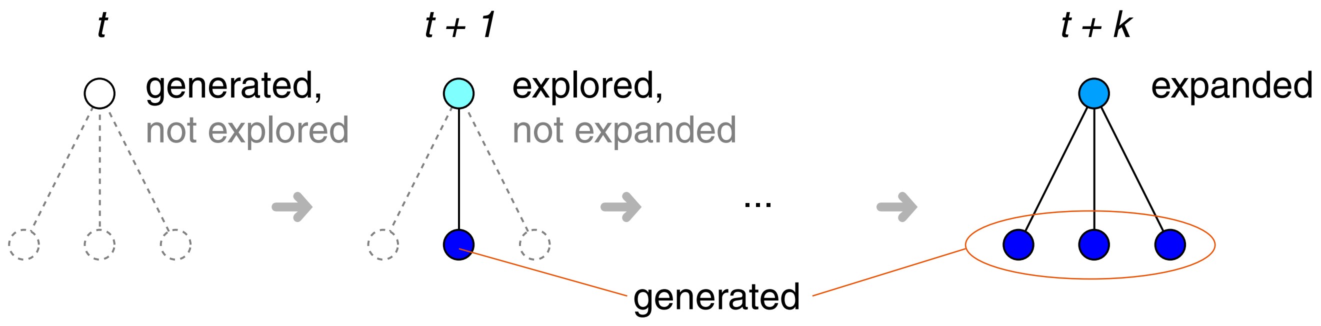
Applying function *successors*(n) and thereby generating all direct successors of a node in one (time) step is called *node expansion*.

- ❑ Node expansion can be seen as the iterated use of node generation without interruption until all successors have been generated.
- ➜ Except for algorithm Backtracking (BT) all algorithms considered will use node expansion as basic step.

©STEIN/LETTMANN 1998-2017

Graph Search Basics

Node Generation as Basic Step



Node status:

- The status of the newly generated successor is called *generated*.
- The status of the parent node is called *explored*.
- The status of the parent node is called *expanded* if all of its direct successors are generated.

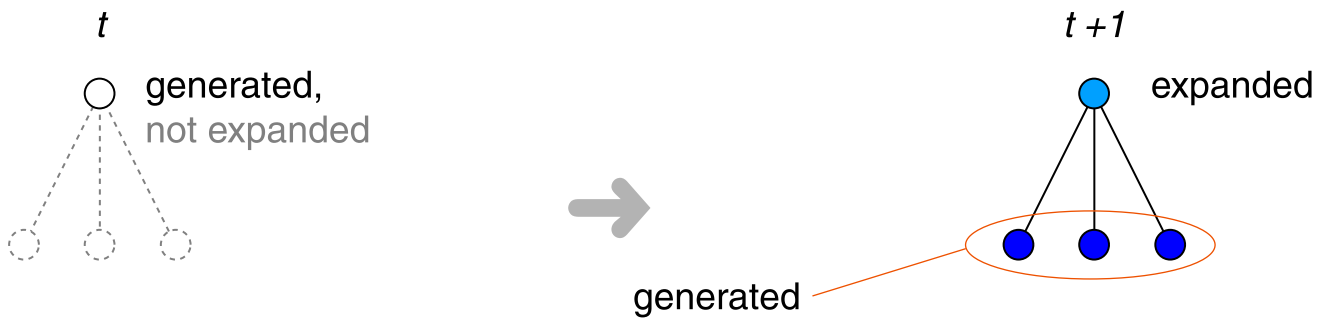
Algorithms based on node generation generate direct successors of a node one by one as needed.

Disadvantage: *next_successor*(n) has to keep track of generated successors.
Advantage: An algorithm has to store less nodes.

© STEIN/LETTMANN 1998-2017

Graph Search Basics

Node Expansion as Basic Step



Node status:

- The status of the newly generated successor is called *generated*.

- The status of the parent node is called *expanded* if all of its direct successors are generated.

Algorithms based on node expansion generate all direct successors of a node in one time step.

Advantage: *successors*(n) is easier to implement.
Disadvantage: An algorithm has to store more nodes.

Remarks:

- Using node generation as basic step requires additional accounting: Because of the step-wise expansion, those operators (transitions) that were applied already need to be marked.
- Knowledge about operator application can be maintained by appending a label to the encoding of a state.

Graph Search Basics

Explicit Parts of State-Space Subgraphs

Let A be an algorithm working on G, given by s and *successors*(n).

- What A could know about G after t node expansions:

  Explored (sub)graph H_t of G after t node expansions.

  Inductive Definition: $H_0 = (\{s\}, \{\})$

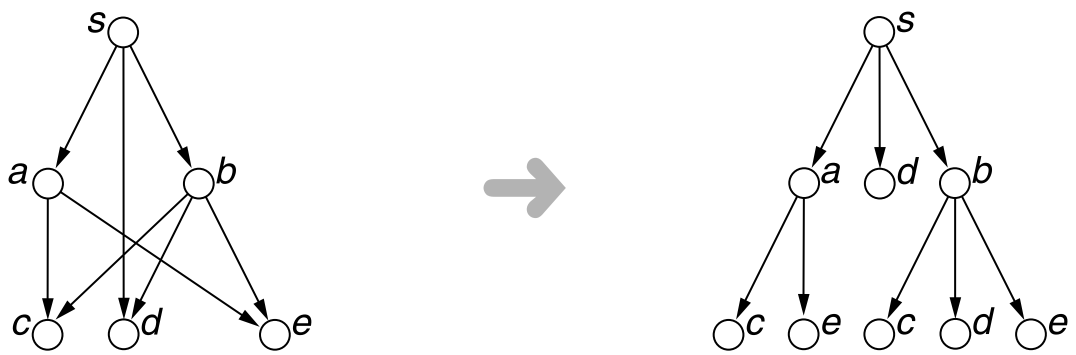
  $H_{t+1} = (V(H_{t+1}), E(H_{t+1}))$ with

  $V(H_{t+1}) = V(H_t) \cup \textit{successors}(n)$
  $E(H_{t+1}) = E(H_t) \cup \{(n, n') \mid n' \in \textit{successors}(n)\}$

Computing successors of a node will return new instantiations (clones) of nodes.

- What A could know if A does not care for identical nodes:

  Tree unfolding T_t of H_t.



Graph Search Basics

Requirements for an Algorithmization of State-Space Search

Properties of Search Space Graphs

1. G is a state-space graph (directed OR graph).
2. G is locally finite.
3. G is implicitly defined by
   (a) a single start node s and
   (b) a function *successors*(n) or *next_successor*(n) returning successors of a node.
4. Computing successors always returns new clones of nodes.
5. G has a set Γ of goal nodes γ.
6. G has a function $\star(.)$ returning true if a solution base is a solution path.

Task:

- Determine in G a solution path for s.

Remarks:

- The terms search space, search space graph, and search graph will be used synonymously.
- Minimum requirement for a solution path is that the terminal node of a solution base is a goal node.
- The property of a graph of being locally finite does not imply an upper bound for a node's degree.
- The property of a graph of being locally finite does not imply an upper bound for the size of the node set that can be generated.
- Two nodes in an OR graph can be connected by two (opposite) directed links, which then may be considered as a single undirected edge. Undirected edges represent a kind of "invertible" operators.
- Functions *successors*(n) resp. *next_successor*(n) return new clones for each successor node of an expanded node. An algorithm that does not care whether two nodes represent the same state will, therefore, consider an unfolding of the state-space graph to a tree with root s.

Remarkss (continued) :

- Questions:
  1. Which of the introduced search problems give rise to undirected edges?
  2. How can multiple start nodes be handled?
  3. Is it possible to combine multiple goal nodes into a single goal node?
  4. For which kind of problems may the search space graph not be locally finite?
  5. How can we deal with problems whose search space graph is not locally finite?
  6. What will happen if none of the goal nodes in Γ can be reached?

Graph Search Basics

Algorithm: Generic_Search (Compare [GTS])

Input: s. Start node representing the initial state (problem).
successors(n). Returns the successors of node n.
$\star(n_0, \ldots, n_k)$. Predicate that is *True* if $(n_0, \ldots, n_k)$ is a solution path.

Output: A node γ representing a solution path or the symbol *Fail*.

```
Generic_Search($s$, successors, $\star$)

  1.  IF $\star((s))$ THEN RETURN($(s)$);  // Check if sol. base $(s)$ is a sol. path.
  2.  push($(s)$, OPEN);    // Store $b'$ on OPEN waiting for extension.
  3.  LOOP
  4.    IF (OPEN $= \emptyset$) THEN RETURN(Fail);
  5.    $b$ = choose(OPEN);    // Choose a solution base $b$ from OPEN.
        remove($b$, OPEN);    // Delete $b$ from OPEN.
        $n$ = last_node($b$);    // Determine last node in $b$.
  6.    FOREACH $n'$ IN successors($n$) DO    // Expand $n$.
          $b'$ = append($b, n'$);    // Extend solution base $b$.
          IF $\star(b')$ THEN RETURN($b'$);    // Check if base $b'$ is a solution path.
          push($b'$, OPEN);    // Store $b'$ on OPEN waiting for extension.
        ENDDO
  7.  ENDLOOP
```

©STEIN/LETTMANN 1998-2017

Remarks:

- Algorithm Generic_Search takes a solution-base-oriented perspective:

  Generic_Search maintains and manipulates solution bases (which are sequences of nodes).

- The last node of a solution base represents a remaining problem which still is to be solved. So, a solution base represents some open problem.

  Therefore, the solution base storage is called OPEN list.

- For an OPEN list some appropriate data structure can be used.

- Function *append*(b, n') returns a representation of the extended solution base. For that purpose, b might be copied. Then, each solution base is represented separately, although they often share initial subsequences.

- Information on the explored part of the search space graph is stored within the solution bases. Node are contained in solution bases and edges are stored as pairs of nodes (occurring in solution bases).
  In fact, after t node expansions the solution bases store parts of the explicit part H_t of the search base graph by storing all maximal paths in H_t, i.e. all paths from s to a terminal node in H_t, that still have to be considered.

- For algorithm Generic_Search it is not important that function *successors*(n) returns new clones for each successor node of an expanded node. There is no need for an unique node representing a state.

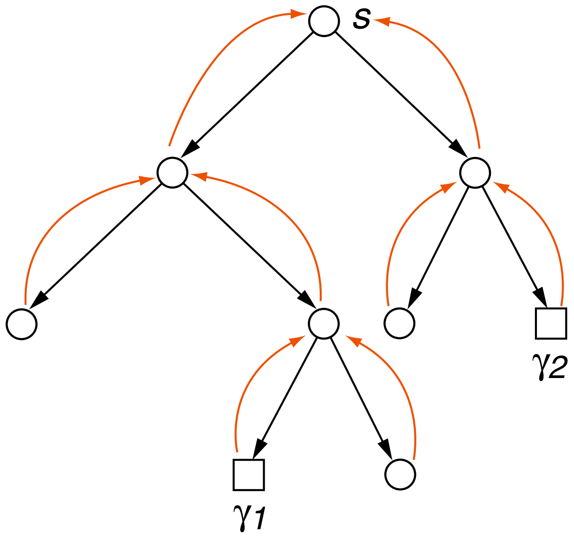
Graph Search Basics

Efficient Storage of Solution Bases

```
6.    FOREACH $n'$ IN $successors(n)$ DO    // Expand $n$.
        $b' = append(b, n')$;    // Extend solution base $b$.
        ...
```

Idea: A solution base can be represented by its last node and a reference to the previous solution base.

➜ Backpointer: a refrence at each newly generated node, pointing to its parent.



➜ A solution base is represented by a backpointer path.

©STEIN/LETTMANN 1998-2017

Remarks:

- The backpointer of start node s has value `null`.
- The backpointer idea allows for an efficient recovery of a solution base given its terminal node if there is at most one backpointer per node.

  If an algorithm wants to store more than one backpointer per node, multiple solution bases can be represented by a node.
- Within an algorithm a node is discarded if the node was removed from the algorithms data structures and if the node is not accessible from some other node in the data structures.

Graph Search Basics

Efficient Storage of Solution Bases (continued)

- Algorithms that only store or discard nodes but do not change backpointers once they are set, store a tree-like graph any point in time.
- If the edges (backpointers) were reversed, it would be a tree unfolding of some part of the explored part of G.

Graph Search Basics

Efficient Storage of Solution Bases (continued)

- Algorithms that only store or discard nodes but do not change backpointers once they are set, store a tree-like graph any point in time.
- If the edges (backpointers) were reversed, it would be a tree unfolding of some part of the explored part of G.

Definition 10 (Traversal Tree, Backpointer Path)

Let A be an algorithm working on G using its implicit definition. Let A store with each generated node a backpointer to its parent node, hereby defining at any point in time t a graph $G_{A,t}$ with nodes in G and backpointers as edges.

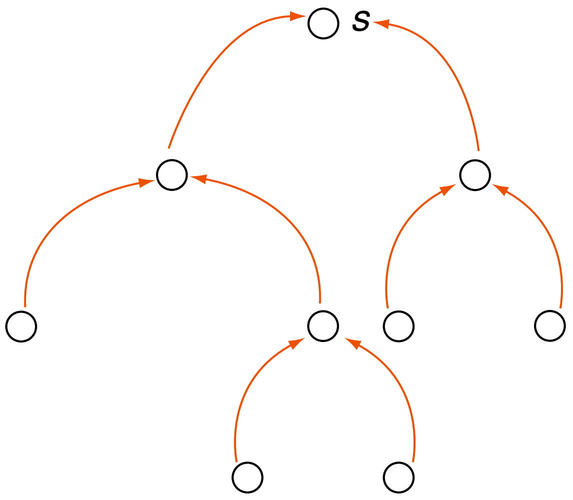
If A does not change backpointer edges in $G_{A,t}$, we call $G_{A,t}$ with its edges reversed the *traversal tree* maintained by A at time t.

For a path $(n, \ldots, s)$ in $G_{A,t}$ the reversed sequence of nodes $(s, \ldots, n)$ defines a s path in the traversal tree, the so-called *backpointer path* for n.

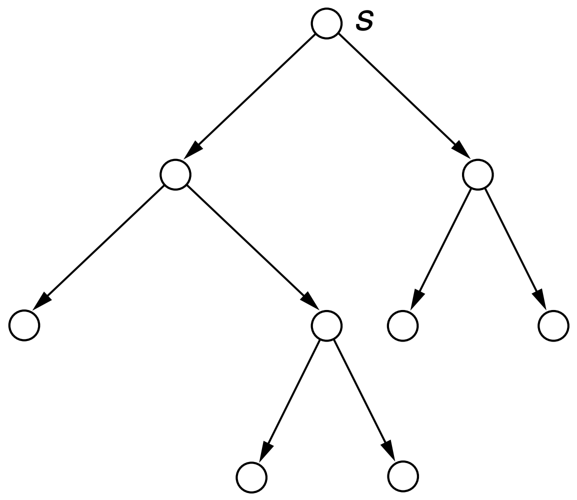
Graph Search Basics

Efficient Storage of Solution Bases (continued)

- A traversal tree is not directly maintained by an algorithm.
- A traversal tree is finite at any point in time.
- A traversal tree is a tree-unfolding of a part of the explored subgraph of G.



Graph maintained by algorithm *A*:
nodes from *G* + backpointers

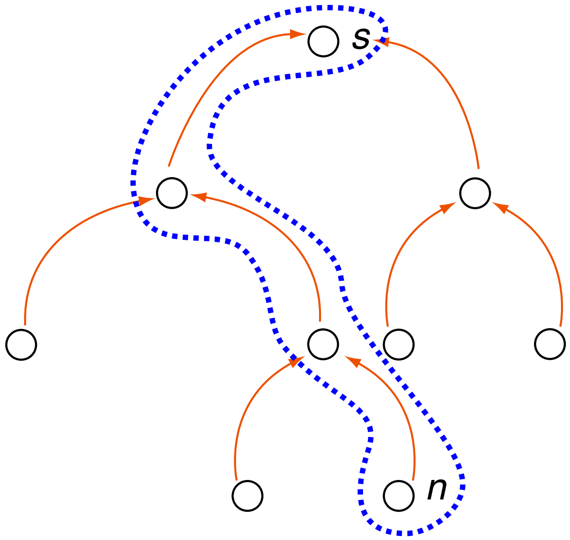


Traversal Tree

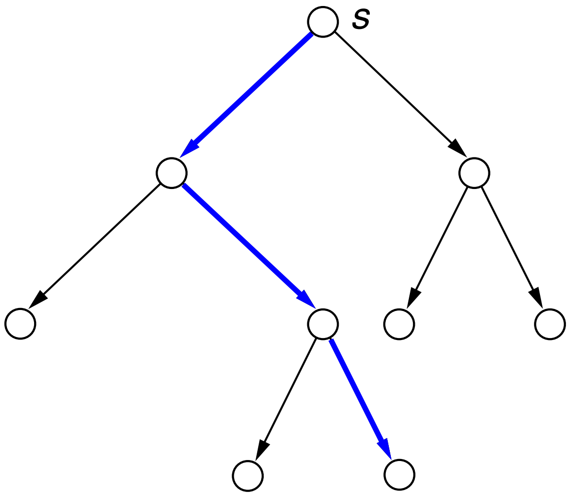
Graph Search Basics

Efficient Storage of Solution Bases (continued)

- A backpointer path is a solution base which can be uniquely identified by the terminal node n.
- Using the backpointer concept, an algorithm can use nodes instead of solution bases. Even returning a solution can be done by returning the goal node (assuming that backpointers are stored within the nodes).



Path defined by node *n*



Backpointer Path

Graph Search Basics

Schema for Search Algorithms

... from a graph-oriented perspective
... using either node expansion or node generation as basic step:

1. Initialize node storage.
2. Loop.
   (a) Using some strategy select an unexpanded node to be explored.
   (b) Start resp. continue the exploration of that node.
   (c) Determine whether a solution graph resp. solution path is found.

Usage:

- Search algorithms maintain sets of nodes.
- Initially, only the start node s is available.
- Search algorithms can make use of graph nodes stored in OPEN and CLOSED and of graph edges stored in form of backpointers.

Remarks:

- The search space graph, which is defined by the possible states and operators, is called *underlying (search space) graph*. An underlying graph is usually denoted by G. The underlying graph is not directly accessible to search algorithms.
- Graph search starts from a single node s and uses node expansion or node generation as a basic step.

  During graph search, only the finite explored (sub)graph is accessible at a point in time. An explored subgraph is also denoted by G, except in cases where the underlying search space graph is addressed as well.
- An explored subgraph G is rooted at node s, the root node of the underlying search space graph. G is connected, more precisely, all nodes in G can be reached from s.
- In a narrower sense, exploration of a generated node n starts with the $\star(n)$ function and not with the generation of a first successor node.

Graph Search Basics

Searching a Search Space Graph

At any time of a search the nodes of a search space graph G can be divided into the following four sets:

1. Nodes that are not generated yet.
2. Nodes that are generated but not explored yet.
3. Nodes that are explored but not expanded yet.
4. Nodes that are expanded.

The distinction gives rise to the well-known sets to organize graph search:

Graph Search Basics

Searching a Search Space Graph

At any time of a search the nodes of a search space graph G can be divided into the following four sets:

1. Nodes that are not generated yet.
2. Nodes that are generated but not explored yet.
3. Nodes that are explored but not expanded yet.
4. Nodes that are expanded.

The distinction gives rise to the well-known sets to organize graph search:

- Generated or explored but not expanded nodes are called “open”.
  They are maintained in a so-called OPEN list.

©STEIN/LETTMANN 1998-2017

Graph Search Basics

Searching a Search Space Graph

At any time of a search the nodes of a search space graph G can be divided into the following four sets:

1. Nodes that are not generated yet.
2. Nodes that are generated but not explored yet.
3. Nodes that are explored but not expanded yet.
4. Nodes that are expanded.

The distinction gives rise to the well-known sets to organize graph search:

- Generated or explored but not expanded nodes are called “open”.
  They are maintained in a so-called OPEN list.

- Expanded nodes are called “closed”.
  They are maintained in a so-called CLOSED list.

© STEIN/LETTMANN 1998-2017

Remarks:

- The situation characterized by Property 1 (“not generated”) is indistinguishable from situations where a node had already been generated and was discarded later on.
- The node sets that are induced by the properties 2, 3, and 4, the generated nodes, the explored nodes, and the expanded nodes, form a subset-hierarchy: only generated nodes can be explored, only explored nodes can be expanded.

Graph Search Basics

Algorithm: GTS (Generic_Tree_Search) (Compare [Generic_Search] [DFS] [BT])

Input: s. Start node representing the initial state (problem).
successors(n). Returns the successors of node n.
$\star(n)$. Predicate that is *True* if n represents a solution path.

Output: A solution path or the symbol *Fail*.

```
GTS($s$, successors, $\star$)

  1. IF $\star(s)$ THEN RETURN($s$);    // Check if $s$ represents a solution path.
  2. push($s$, OPEN);    // Store $s$ on OPEN waiting for extension.
  3. LOOP
  4.   IF (OPEN $= \emptyset$) THEN RETURN(Fail);
  5.   $n$ = choose(OPEN);    // Choose a solution base represented by $n$.
       remove($n$, OPEN);    // Delete $n$ from OPEN.
       push($n$, CLOSED);    // Store $n$ on CLOSED for easy access.
  6.   FOREACH $n'$ IN successors($n$) DO    // Expand $n$.
         add_backpointer($n'$, $n$);    // Extend solution base represented by $n$.
         IF $\star(n')$ THEN RETURN($n'$);    // Check if $n'$ repr. a solution path.
         push($n'$, OPEN);    // Store $bn'$ on OPEN waiting for extension.
       ENDDO
  7. ENDLOOP
```

©STEIN/LETTMANN 1998-2017

Remarks:

- ❑ Algorithm GTS takes a graph-oriented perspective:

  GTS maintains and manipulates nodes and backpointers.

- ❑ A solution base is stored via its terminal nodes that are connected via backpointers to their predecessors in the solution base.
  Therefore, OPEN is now a list of nodes.

- ❑ The node list CLOSED is kept for garbage collection purposes. Since nodes are shared in backpointer paths, a node can be discarded not before its last successor was discarded.

- ❑ Information on the explored part of the search space graph is stored in lists of nodes, OPEN and CLOSED. Information on edges is stored in the backpointers of the nodes. A backpointers represents an edge with opposite direction.

- ❑ For a CLOSED list some appropriate data structure can be used.

Graph Search Basics

Searching a Search Space Graph (continued)

A *search strategy* decides which node to expand next. A search strategy should be systematic, i.e. it should consider any solution base, but none twice.

[control strategy]

Search strategies are distinguished with regard to the regime the nodes in the search space graph G are analyzed:

1. Blind or uninformed.

   The order by which nodes in G are chosen for exploration / expansion depends only on information collected during the search up to this point. I.e., the not explored part of G, as well as information about goal criteria, is not considered.

2. Informed, guided, or directed.

   In addition to the information about the explored part of G, information about the position (direction, depth, distance, etc.) of the goal nodes Γ, as well as knowledge from the domain is considered.

©STEIN/LETTMANN 1998-2017

Graph Search Basics

Uninformed Systematic Search

The key for systematic search is to keep alternatives in reserve, leading to the following trade-off:

Space (for remembering states) versus Time (for re-generating states)

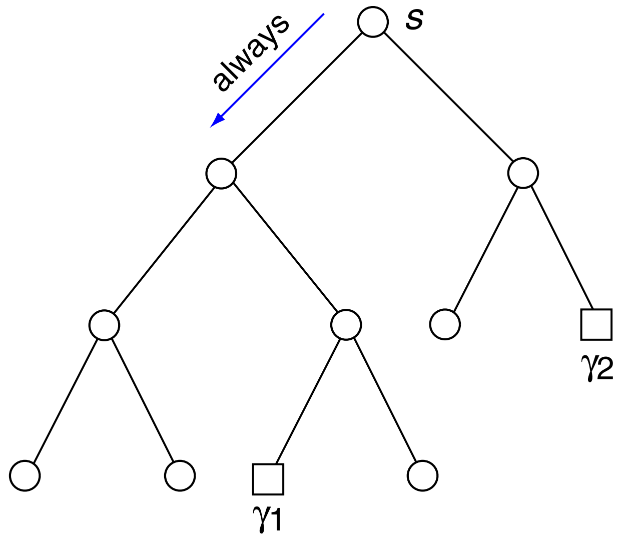
Graph Search Basics

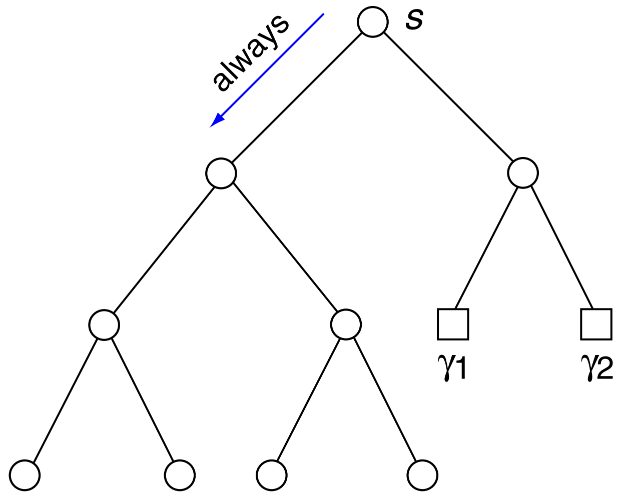
Uninformed Systematic Search

The key for systematic search is to keep alternatives in reserve, leading to the following trade-off:

Space (for remembering states) versus Time (for re-generating states)

Uninformed (blind) search will expand nodes *independent from knowledge*, e.g. knowledge about goal nodes in G.





©STEIN/LETTMANN 1998-2017

Remarks:

- ❑ Search algorithm schemes that are uninformed—and systematic at the same time—are also called “tentative strategies”.
- ❑ Known representatives of uninformed (systematic) search strategies are depth-first search, backtracking, breadth-first search, and uniform-cost search.
- ❑ Disclaimer (as expected): real-world problems cannot be tackled with uninformed 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 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 correctly.

©STEIN/LETTMANN 1998-2017

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*.

© STEIN/LETTMANN 1998-2017

Depth-First Search [GTS] [BT]

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

  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.

          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();

  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 [GTS] [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
```

©STEIN/LETTMANN 1998-2017

Remarks:

- Reaching the depth bound 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 [GTS] [BT]

```
DFS($s$, $successors$, $\star$, $\bot$, $k$)    // 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 $\bot(n')$
            THEN
              pop(OPEN);
              cleanup_closed();
            ENDIF
          ENDDO
          IF (successors($n$) $= \emptyset$) THEN cleanup_closed();
        ENDIF
  6.  ENDLOOP
```

©STEIN/LETTMANN 1998-2017

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 holds: $\perp(n) =$ *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.

©STEIN/LETTMANN 1998-2017

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.

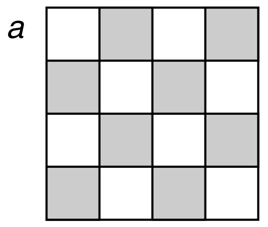
(b) Dead ends can be recognized early, i.e., with a considerable look-ahead.

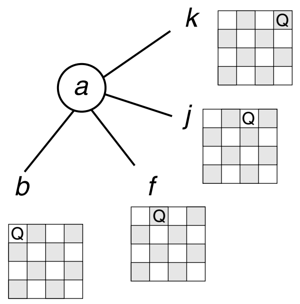
(c) There are no cyclic or infinite paths resp. cyclic or infinite paths can be avoided.

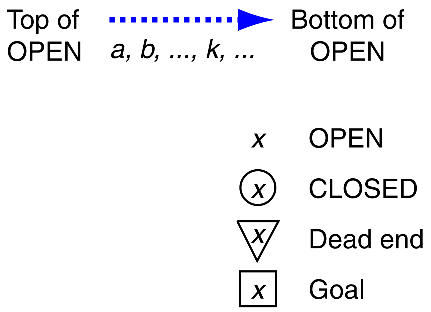
© STEIN/LETTMANN 1998-2017

Depth-First Search

Example: 4-Queens Problem



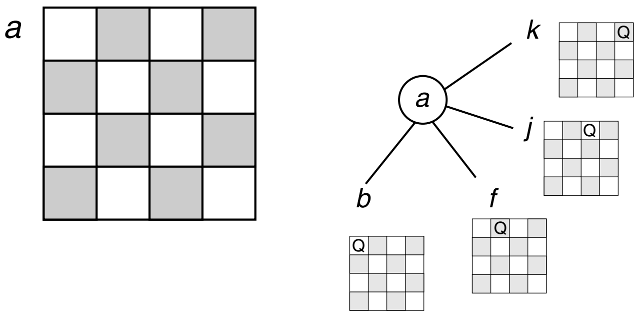


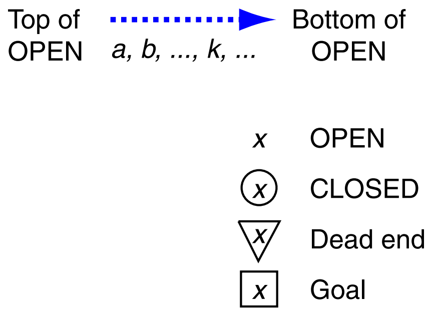


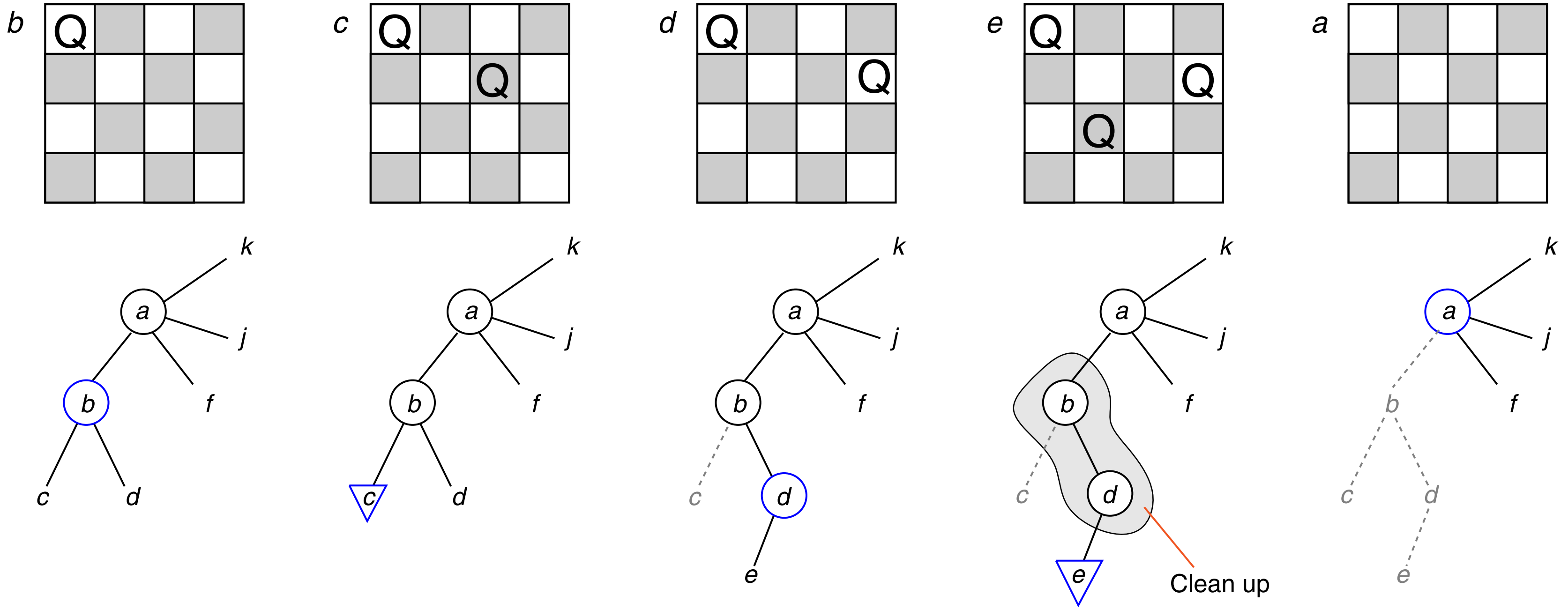
©STEIN/LETTMANN 1998-2017

Depth-First Search

Example: 4-Queens Problem (continued)



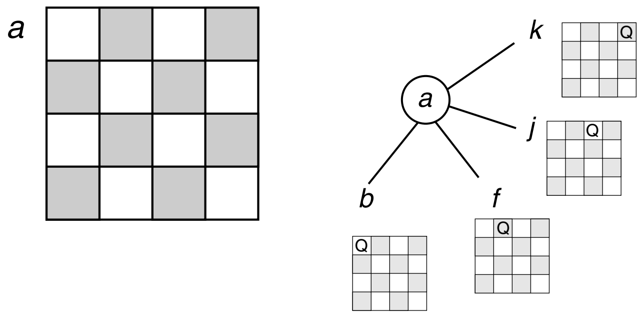


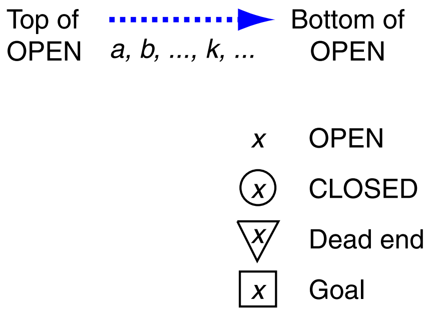


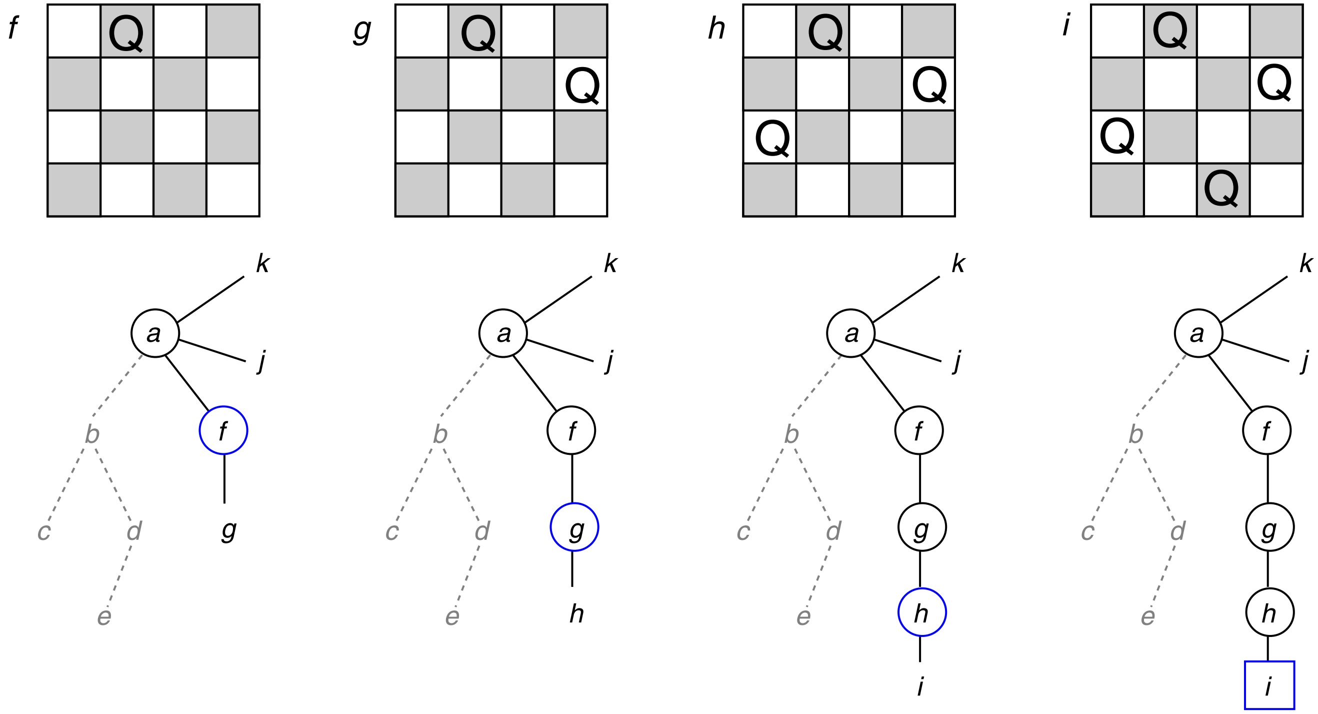
©STEIN/LETTMANN 1998-2017

Depth-First Search

Example: 4-Queens Problem (continued)





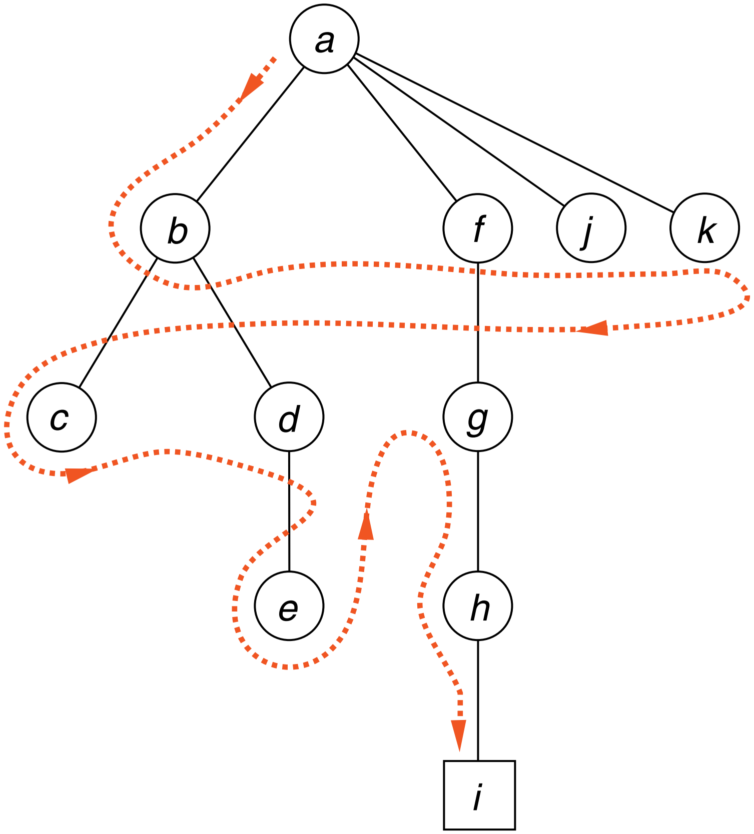


© STEIN/LETTMANN 1998-2017

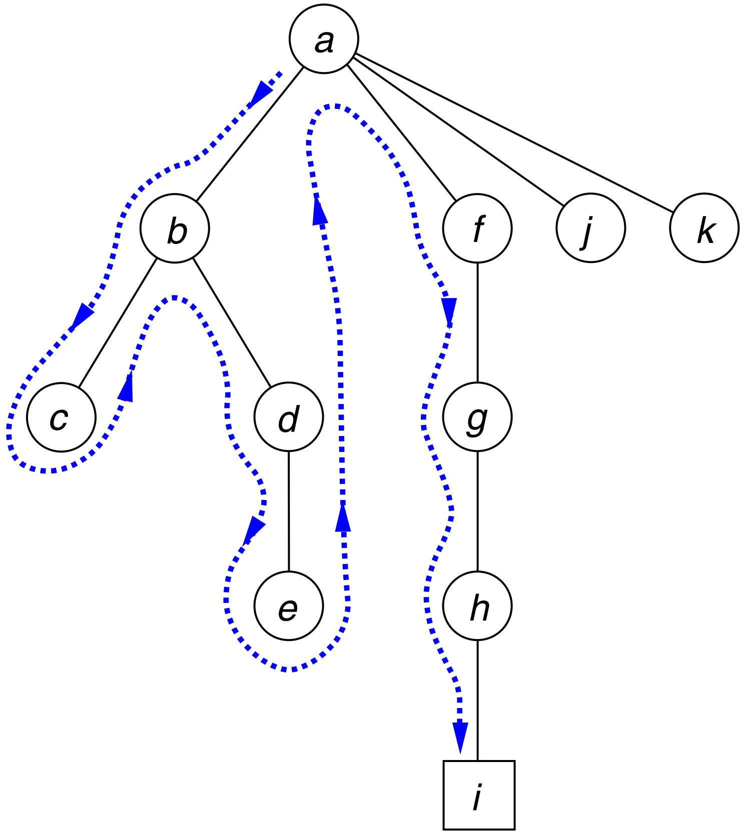
Depth-First Search

Example: 4-Queens Problem (continued)

DFS node processing sequence:



Node generation



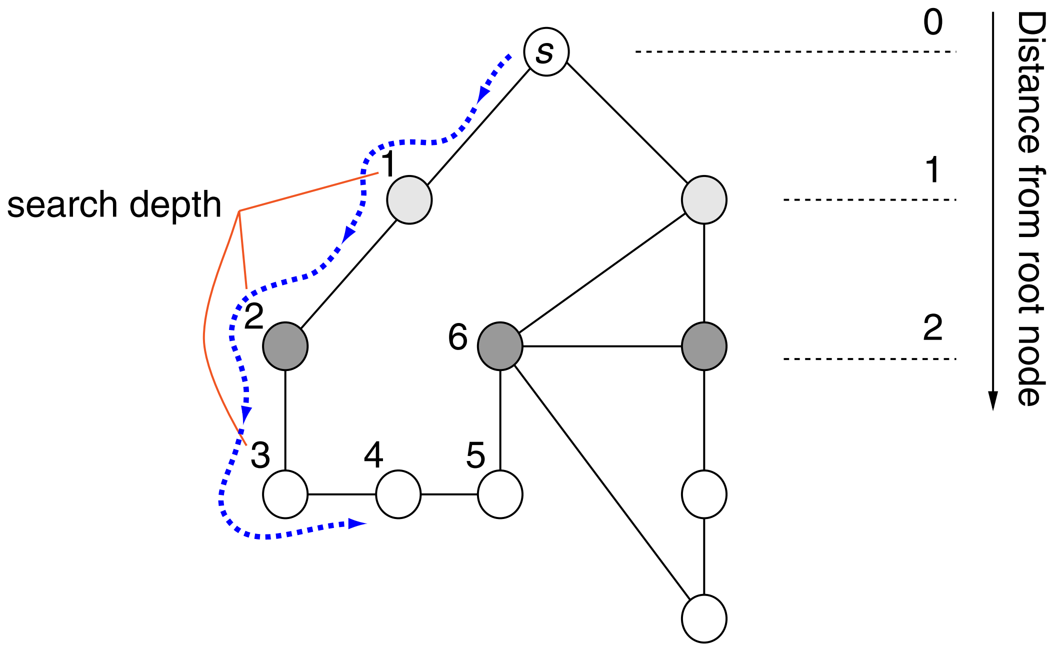
Node expansion

©STEIN/LETTMANN 1998-2017

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.

©STEIN/LETTMANN 1998-2017

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 termination criterion, $\perp(n') =$ *True*, backtracking is invoked and search is continued with the next non-expanded successor of n.

©STEIN/LETTMANN 1998-2017

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.
new_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*.

Backtracking [GTS] [DFS]

```
BT($s$, $\textit{new\_successor}$, $\star$, $\perp$, $k$)

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

©STEIN/LETTMANN 1998-2017

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:

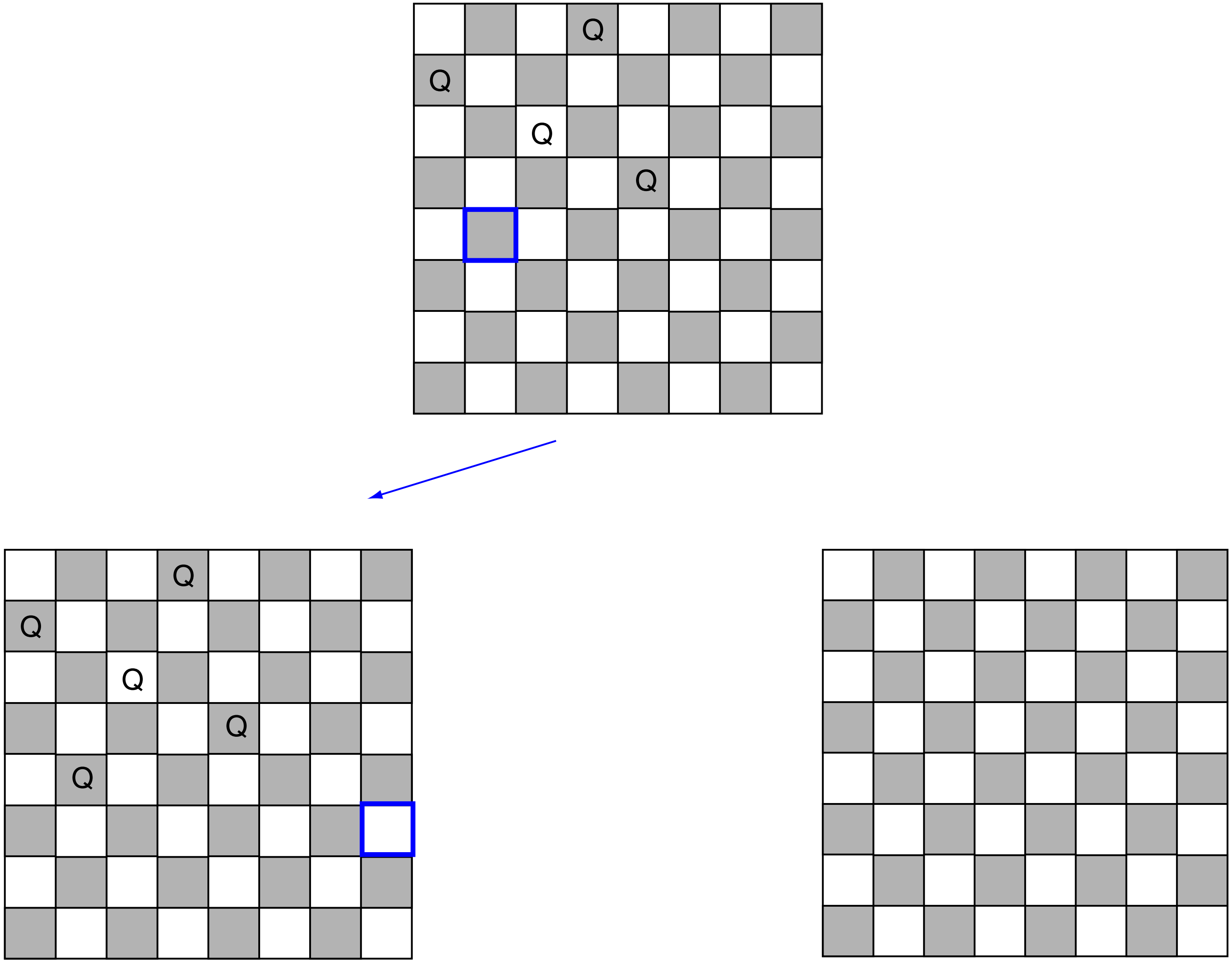
- Backtracking requires only an OPEN list, which serves as both LIFO stack and traversal path storage.

  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 expanded.

Backtracking

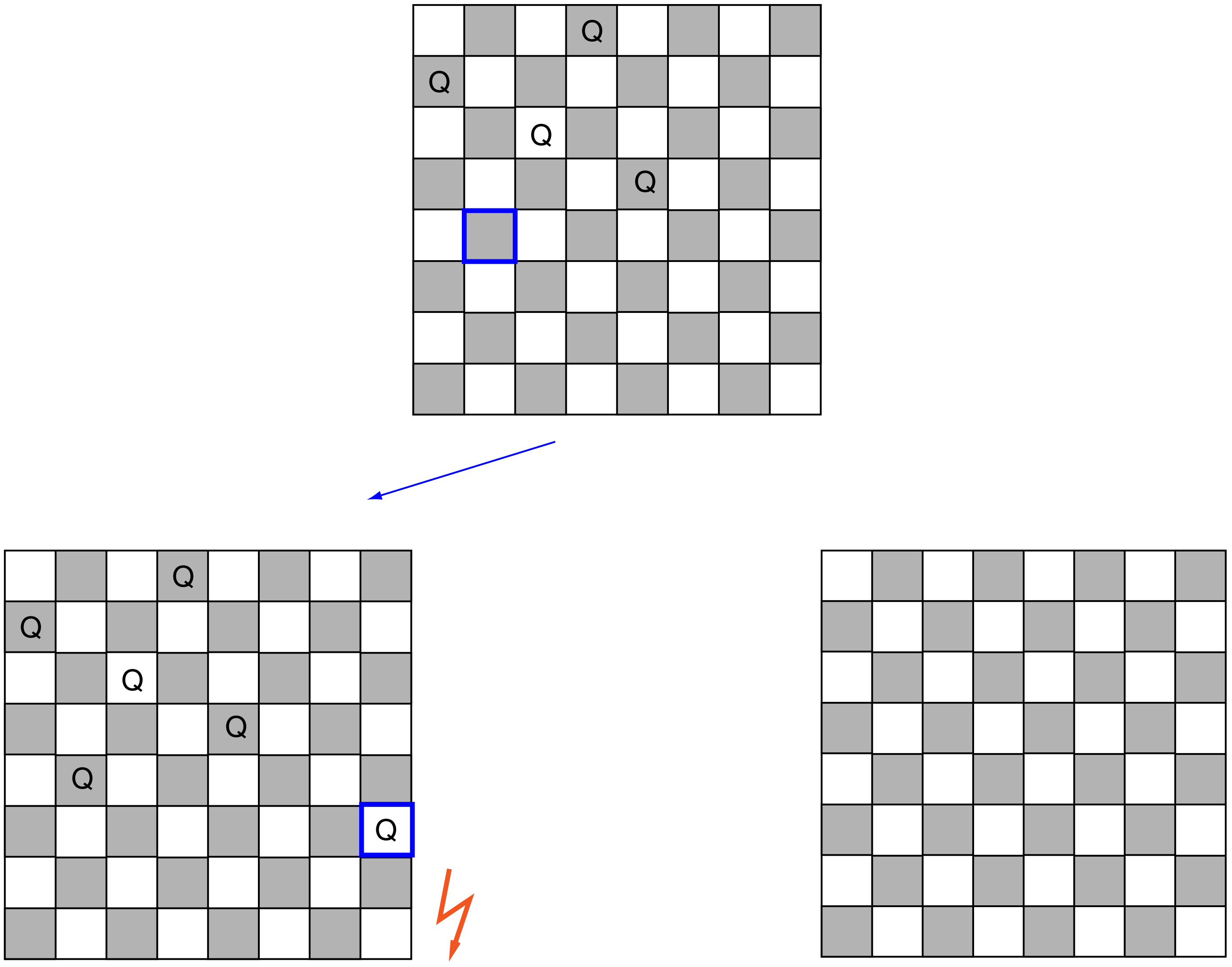
Non-Monotone Backtracking: 8-Queens Problem



©STEIN/LETTMANN 1998-2017

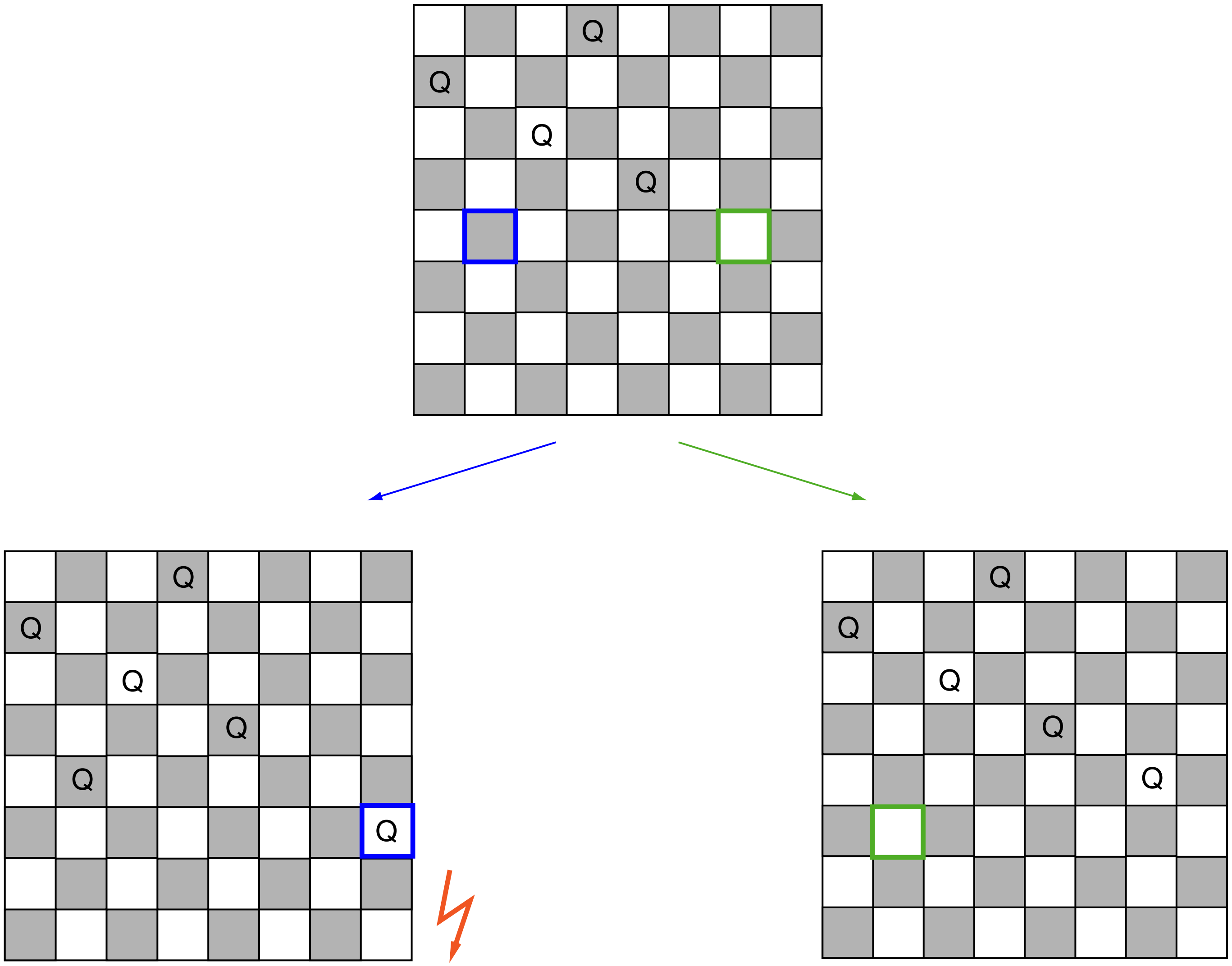
Backtracking

Non-Monotone Backtracking: 8-Queens Problem (continued)



Backtracking

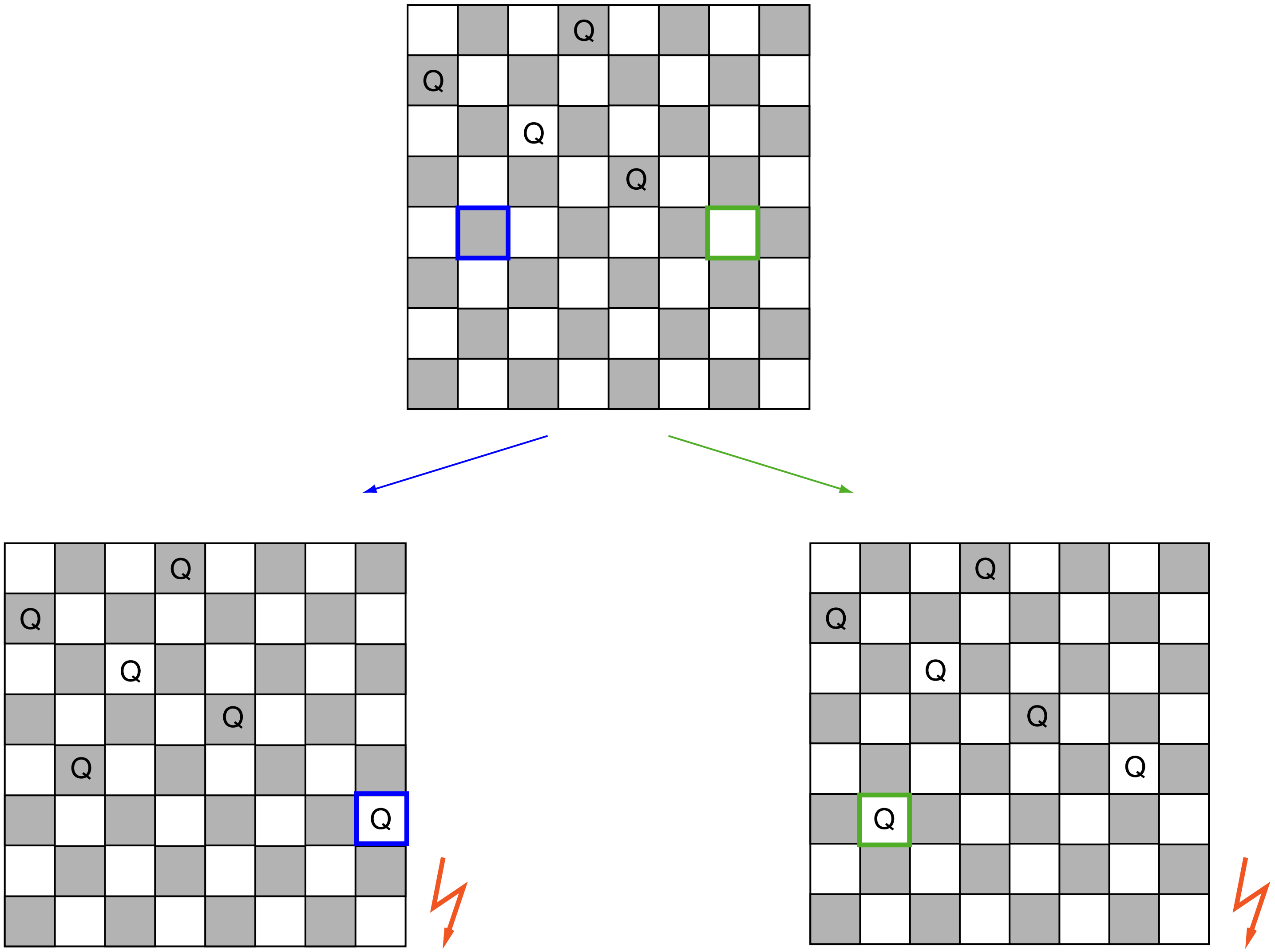
Non-Monotone Backtracking: 8-Queens Problem (continued)



©STEIN/LETTMANN 1998-2017

Backtracking

Non-Monotone Backtracking: 8-Queens Problem (continued)

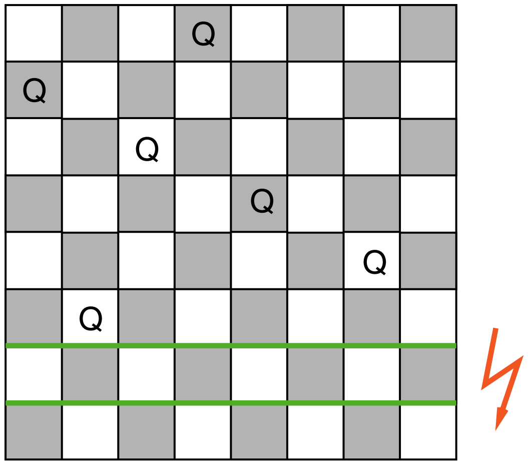


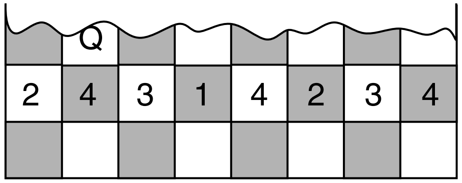
Backtracking

Non-Monotone Backtracking: 8-Queens Problem (continued)

The 8-queens CSP 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

Simple Cost Concepts

- Edge weight.

  Encode either cost values or merit values, which are accounted if the respective edges become part of the solution.

  $c(n, n')$ denotes the cost value of an edge from n to n'.

- Path cost.

  The cost of a path, C_P, results from applying a *cost measure* F, which specifies how cost of a continuing edge is combined with the cost of the rest of the path.

  Examples:

  Sum cost $:=$ the sum of all edge costs of a path P from s to n:

  $$C_P(s) = \sum_{i=0}^{k-1} c(n_i, n_{i+1}), \text{ with } n_0 = s \text{ and } n_k = n$$

  Maximum cost $:=$ the maximum of all edge costs of a path:

  $$C_P(s) = \max_{i \in \{0,\ldots,k-1\}} c(n_i, n_{i+1}), \text{ with } n_0 = s \text{ and } n_k = n$$

Backtracking

Backtracking for Optimization: Generic

Setting:

- The search space graph contains several solution paths.

Task:

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

Approach:

- Pursue a depth-first strategy.
- Continue search only until the costs of the partial solutions exceed the currently optimum cost. Keyword: *Early Pruning*

Prerequisite:

- The accumulated cost on each path increase monotonically.

Backtracking

Backtracking 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

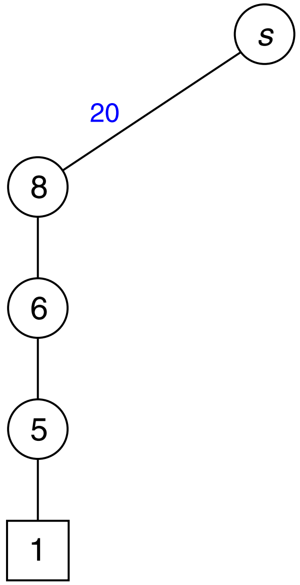
© STEIN/LETTMANN 1998-2017

Backtracking

Backtracking 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



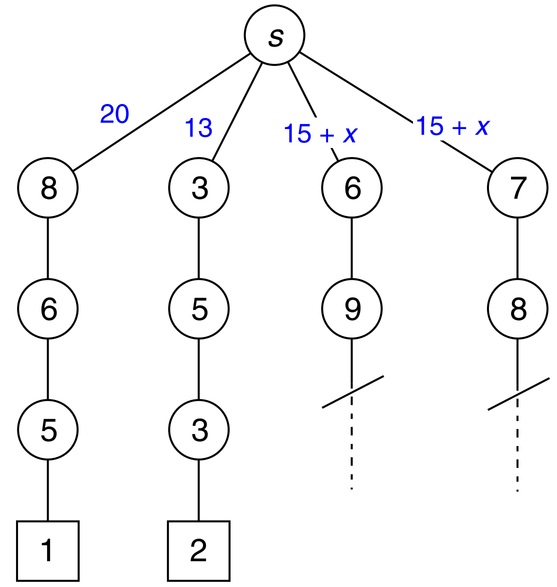
© STEIN/LETTMANN 1998-2017

Backtracking

Backtracking 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



The quality (power) of a heuristic defines the pruning gain: The earlier we find a cheap solution the larger are the search effort savings.

©STEIN/LETTMANN 1998-2017