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 Basics
- AND-OR Graph Search

© Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Basics

Problem Reduction: A Powerful Tool in Problem Solving

Examples for problem decompositions in S:I Introduction

- Counterfeit Coin Problem
- Tic-Tac-Toe Game

Examples of problem decomposition in algorithms: divide-and-conquer

- Mergesort.
  Divide list of objects in two subsets, sort these sublists separately, and merge them; requires $\mathcal{O}(n \log n)$ comparisons. [Wikipedia: Merge Sort]
- Fast Median Computation.
  Special case of the selection problem *"determine k-th element in list of numbers according to some sorting"*; algorithm by Blum, Floyd, Pratt, Rivest, and Tarjan requires at most $6n$ comparisons. [Wikipedia: Sample Median]
- Fast Integer Multiplication.
  Karatsuba algorithm $(a2^k + b)(c2^k + d) = ac2^{2k} + (ac + bd - (a - b)(c - d))2^k + bd$; requires $\mathcal{O}(n^{1.585})$ bit operations compared to $\mathcal{O}(n^2)$ required by a naive implementation.
  [Wikipedia: Karatsuba algorithm]
- Matrix Multiplication.
  Strassen algorithm requires $\mathcal{O}(n^{2.808})$ arithmetic operations compared to $\mathcal{O}(n^3)$ required by a naive implementation. [Wikipedia: Strassen algorithm]

©Stein/Lettmann/Hagen 1998–2021

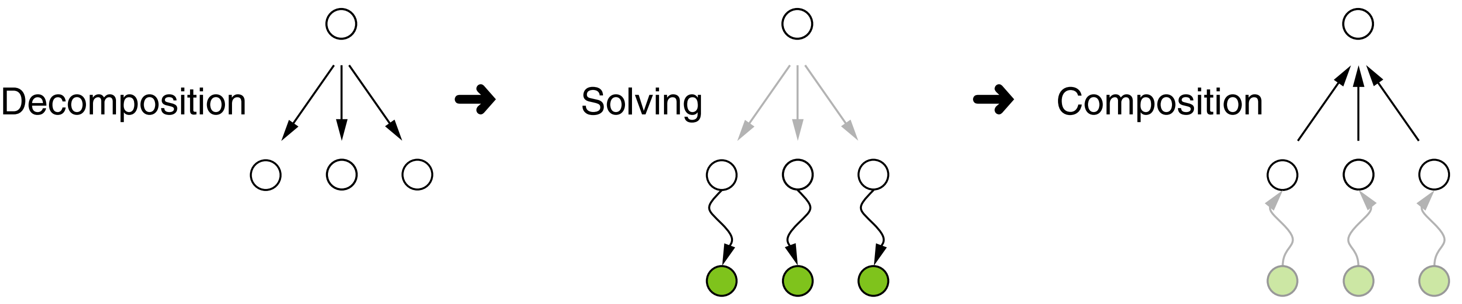
AND-OR Graph Basics

Building Blocks of Problem Reduction

- **Decomposition into subproblems.**
  A problem at hand is decomposed into a finite set of independently solvable subproblems.

- **Solving all subproblems.**
  All the subproblems are solved independently (i.e., without interactions between solution processes for different subproblems).

- **Solution composition.**
  Solutions to the subproblems are used to construct a solution of the problem at hand in a (relatively) simple way.



 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Basics

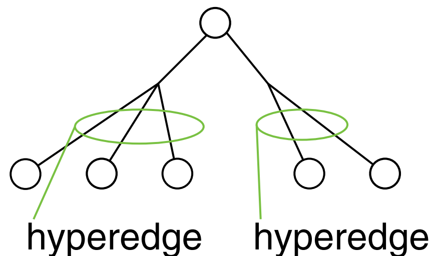
Graph Representations of Problem Reduction

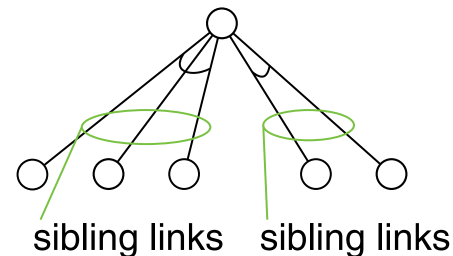
- **Directed Hyperedges.**

  A directed hyperedge $(n, \{n'_1, \ldots, n'_k\})$ with single start node n and multiple end nodes $n'_1, \ldots, n'_k$ represents the decomposition of the problem represented by n into subproblems represented by $n'_1, \ldots, n'_k$.

- **Families of AND links (AND edges).**

  A family of directed edges $(n, n'_1), \ldots, (n, n'_k)$ represents the decomposition into subproblems. In graphical representations sibling AND links $(n, n'_1), \ldots, (n, n'_k)$ are connected by an arc.





Q. Is there a simple way to integrate the concept of hyperedges or families of AND links into state-space graphs?

 ©Stein/Lettmann/Hagen 1998–2021

Remarks:

- The above hyperedges are directed forward hyperedges, also called F-arcs (forward arcs). [Gallo 1993]

- The concept of (cyclic/acyclic) paths can be extended to directed hypergraphs in a straightforward way. The definition of hyperpaths, however, includes the property of being acyclic and minimal. [Nielsen/Andersen/Pretolani 2005]

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Basics

Problem Reduction: Integration into State-Space Graphs

Idea: Distinguish nodes instead of links.

1. AND nodes.

   Represent a state where only a specific problem decomposition is possible. AND nodes have a single family of AND links as outgoing links.

2. OR nodes.

   Represent a state where only state transitions are possible or a decision to perform a specific problem decomposition. OR nodes have only OR links as outgoing links.

Definition 15 ((Canonical) Problem-Reduction Graph, AND-OR Graph)

Let S be a state-space and $l : S \rightarrow \{\text{AND}, \text{OR}\}$ be a labeling of states in S that defines disjoint sets of AND-states resp. OR-states. A graph $G = (S, E)$ with node set S and edge set $E \subseteq S \times S$ together with a labeling l and an OR-state s as start node is called a (canonical) problem-reduction graph or an AND-OR graph.

Usually, the labeling l will not be mentioned for AND-OR graphs.

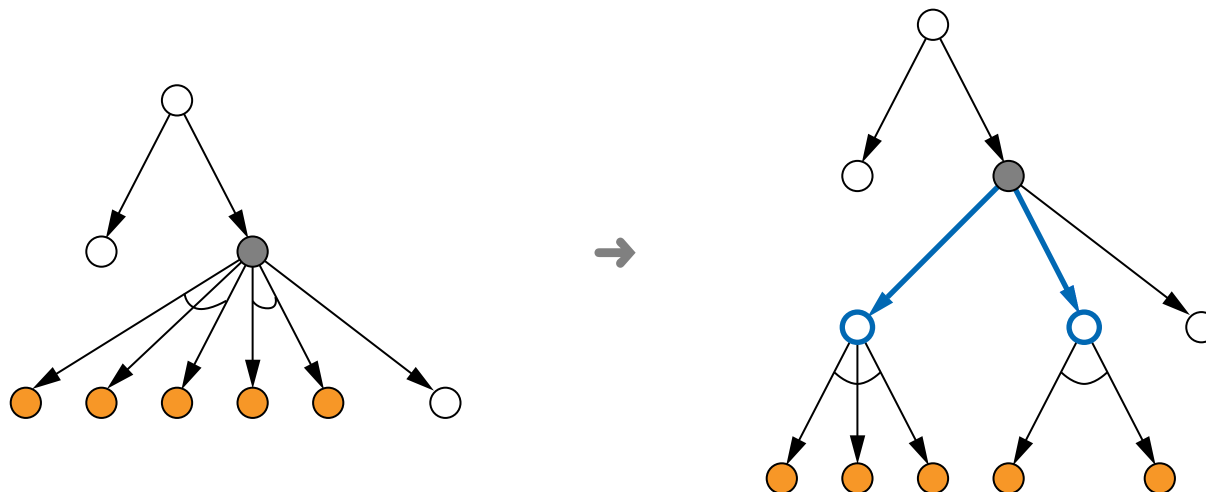
©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Basics

Canonical Representation of AND-OR Graphs

Graph transformation.

1. Label all nodes as OR-nodes.
2. As long as there is an OR-node n with a family of outgoing AND links $(n, n'_1), \ldots, (n, n'_k)$, introduce a new intermediate AND-node n' and replace the links by a new OR link (n, n') and AND links $(n', n'_1), \ldots, (n', n'_k)$.



Since a canonical representation can always be generated, we will consider only AND-OR graphs.

©Stein/Lettmann/Hagen 1998–2021

Remarks:

- Outgoing links of an OR-node are considered as alternative transformations of a problem.
- Outgoing links of an AND-node are considered as a family of AND-links representing a problem decomposition (e.g., a subset splitting).
- The canonical representation of AND-OR graphs does not require alternating node types. In particular, it is not really necessary to require that the root node is of type OR.
- As solutions to subproblems of a problem decomposition are combined to form the solution to the original problem, solutions have to be represented by more complex structures than paths in state space graphs.
- A state space graph is an AND-OR graph without AND nodes.

© Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Basics

AND-OR Graph Properties

Well-known concepts from graph theory (path, tree, . . .) can be used as well in context of AND-OR graphs.

Example: Acyclic AND-OR Graph

Let $G = (S, E)$ with labeling l be an AND-OR-graph. The AND-OR graph is called acyclic or cycle-free if the underlying graph G is acyclic.

If the special semantics of AND-nodes as starting point of a problem decomposition is involved, redefinitions would be necessary.

Example: AND-OR Subgraph

Let $G = (S, E)$ with labeling l be an AND-OR-graph. A subgraph $G' = (S', E')$ of G with labeling $l|_{S'}$ is an AND-OR subgraph of G with l, if G' contains for each AND node in G either all or none of its outgoing edges in G.

Usage:

To avoid misunderstandings, only standard concepts from graph theory are used.

Consequences resulting from the semantics of AND-nodes are always mentioned explicitly.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Basics

Solutions in AND-OR Graphs

Desirable characteristics:

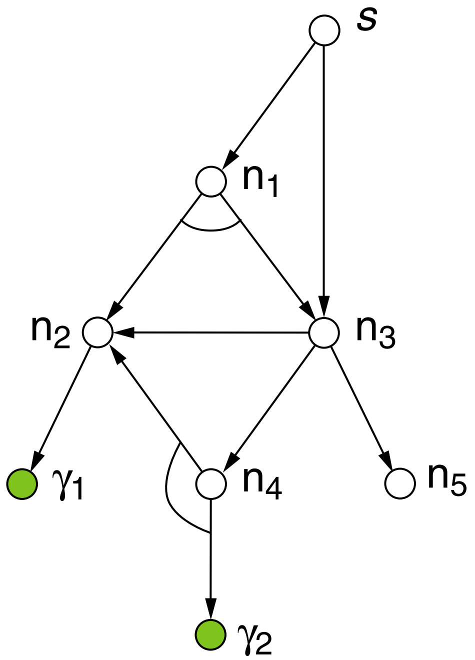
- Compatibility.
  Paths should be permitted as solutions if only OR nodes are contained.
- Constructability.
  There is a procedure to construct a solution stepwise.
- Representability.
  Solutions can be represented as AND-OR trees.
- Totality.
  No (sub-)problem in a solution should remain unsolved.
- Finiteness.
  Solutions should be finite structures that contain for any subproblem a finite number of steps to reach goal nodes.
- Reuse.
  Solutions should provide only one (sub-)solution for problems that occur multiple times.
- Minimality.
  Solutions should not contain superfluous parts.

©Stein/Lettmann/Hagen 1998–2021

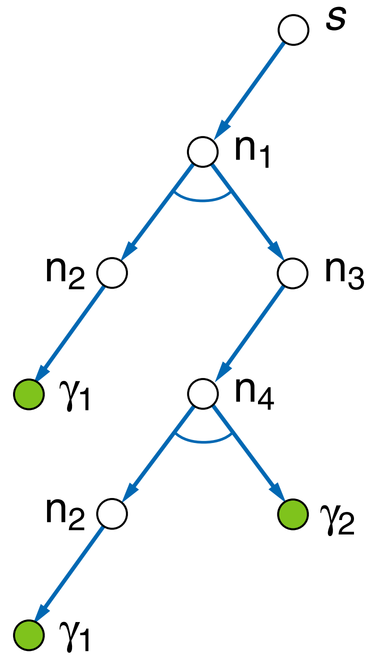
AND-OR Graph Basics

Solution Trees: A Generalisation of Solution Paths

AND-OR graph example:



Solution tree:



➜ Solution as set of paths: $\{(s, n_1, n_2, \gamma_1), (s, n_1, n_3, n_4, n_2, \gamma_1), (s, n_1, n_3, n_4, \gamma_2)\}$
Compact representation: AND-OR solution tree

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Basics

Solution Trees in AND-OR Graphs [Solution Path in OR Graphs]

Definition 16 (Solution Tree for an AND-OR Graph)

Let G be an AND-OR graph, and let n be a node in G. Also, let some additional solution constraints be given. An AND-OR graph H is called a *solution tree for* n *in* G iff ($\leftrightarrow$) the following conditions hold:

1. H is a finite tree.
2. H contains the node n as root node.
3. If H contains an inner OR node n', which is an instantiation of a node in G, then H contains an instantiation of a successor node of n' in G and the corresponding link.
4. If H contains an inner AND node n', which is an instantiation of a node in G, then H contains instantiations of all of the successor nodes of n' in G and all corresponding links.
5. The leaf nodes (terminal nodes) in H are instantiations of goal nodes in G.
6. H satisfies the solution constraints.

©Stein/Lettmann/Hagen 1998–2021

Remarks:

- For compatibility of solutions in AND-OR graphs with solution paths in OR graphs we allow multiple instances of nodes in a solution tree for representing loops. This conflicts with the request for reusing a solution for every instance of a problem.

  Best-First search algorithms for OR graphs prune cyclic paths. Therefore, compatibility could be restricted to cycle-free solution paths: Any path from s to a leaf node in a solution tree for s for an AND-OR graph could be restricted to be cycle-free.

  Q. Does this restriction solve the reuse conflict?

©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Differences to OR Graphs Search

Besides checking additional solution constraints, the different search space representations entail different termination tests:

- State-space graph: analyze a single node (leaf node of a solution base).
- AND-OR graph: analyze a set of nodes (leaf nodes of a particular solution-tree base).

© Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Differences to OR Graphs Search

Besides checking additional solution constraints, the different search space representations entail different termination tests:

- State-space graph: analyze a single node (leaf node of a solution base).
- AND-OR graph: analyze a set of nodes (leaf nodes of a particular solution-tree base).

Recall the termination test in AND-OR graph search:

- We have to answer the question:
  Does the AND-OR graph G allow for the instantiation of a solution tree?
- To do: Collect knowledge about nodes representing solved or unsolvable problems and its implications for ancestor nodes.

Approach

- Adapt [DFS] (or [BFS]) for the propagation of labels "solved" and "unsolvable".

©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Solved Labeling

The question whether or not an AND-OR tree G has a solution tree can be answered by applying the following (bottom-up) labeling rules.

Definition 17 (Solved-Labeling Procedure)

Let G be an AND-OR tree. A node in G is labeled as "solved" resp. "unsolvable" if one of the following conditions is fulfilled.

1. A terminal node (leaf node) is labeled as "solved" if it is a goal node (solved rest problem); otherwise it is labeled as "unsolvable".
2. A non-terminal OR node is labeled as "solved" if one of its successor nodes is labeled as "solved"; it is labeled as "unsolvable" if all of its successor nodes are labeled as "unsolvable".
3. A non-terminal AND node is labeled as "solved" if all of its successor nodes are labeled as "solved"; it is labeled as "unsolvable" if one of its successor nodes is labeled as "unsolvable".

Note: No additional solution constraints are considered here.

©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Solved Labeling Using DFS

Applicability.

- Algorithm DFS performs a tree unfolding of the search space graph G. This works even if G is an AND-OR graph. Therefore, the above labeling rules can be used in DFS.

Additional initializations.

- All nodes are unlabeled when generated. This includes the start node s.
- If a node n is expanded, the number of its successors is stored as *unlabeled_succ*(n).

Leaf node processing.

- If a goal node is found, label "solved" is propagated along the back-pointers instead of returning a solution path.
- If a dead end is found, label "unsolvable" is propagated along the back-pointers.
- Termination occurs as soon as s is labeled.

Processing of non-terminal nodes.

- Nodes are labeled according to labeling rules. If a label is set, this label is propagated further.

©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs [DFS]

Solved Labeling Using DFS (continued)

```
solved_labeling_DFS(s, successors, ⋆, ⊥, k)

 1.  push(s, OPEN);
 2.  LOOP
 3.    IF (OPEN = ∅) THEN RETURN(Fail);
 4.    n = pop(OPEN); push(n, CLOSED);
 5.    IF (depth(n) < k)
       THEN
         FOREACH n' IN successors(n) DO   // Expand n.
           set_backpointer(n', n);
           IF is_goal_node(n') THEN set_label(n', "solved");
           IF ⊥(n') THEN set_label(n', "unsolvable");
           IF is_labeled(n')
           THEN
             propagate_label(n'); IF is_labeled(s) THEN RETURN(label(s));
           ELSE
             push(n', OPEN);
           ENDIF
         ENDDO
       ENDIF
       cleanup_closed();
 6.  ENDLOOP
```

 ©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Solved Labeling Using DFS (continued)

Label propagation is done via back-pointer references. For the start node s the back-pointer reference is assumed to be `null`.

```
propagate_label(n)
  l = label(n);  p = backpointer_parent(n);
  WHILE p ≠ null AND is_unlabeled(p) DO    // Process parent node p.
    IF is_OR_node(p) AND l = "solved" THEN set_label(p, "solved");
    IF is_OR_node(p) AND l = "unsolvable"
    THEN
      unlabeled_succ(p) = unlabeled_succ(p) − 1;
      IF unlabeled_succ(p) = 0 THEN set_label(p, "unsolvable");
    ENDIF
    ...    // Case is_AND_node(p) analogously.
    IF is_labeled(p)
    THEN
      l = label(p);  p = backpointer_parent(p);
    ELSE
      p = null;
    ENDIF
  ENDDO
```


©Stein/Lettmann/Hagen 1998–2021

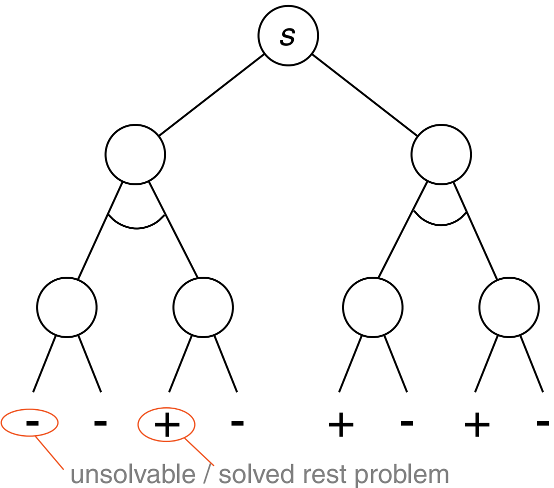
Remarks:

- Function *cleanup_closed*(n) will remove all nodes that have been labeled together with all nodes for which such a node is an ancestor (via back-pointers). Thereby also some OPEN nodes may be removed.
- In solved_labeling_DFS only a single path (i.e., a sequence of nodes) is processed in function *propagate_label*(n). However, the label of each node processed by AND-OR_labeling_DFS will be set at most once. Therefore, the overall effort in labeling is linear in the number of nodes (node instantiations) processed by solved_labeling_DFS.

©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

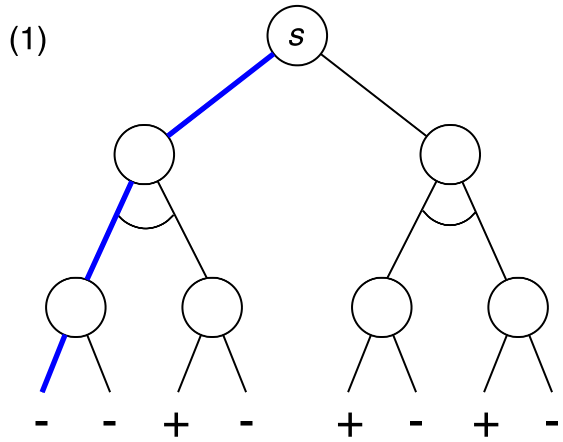
Example: Solved Labeling Using DFS



 ©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

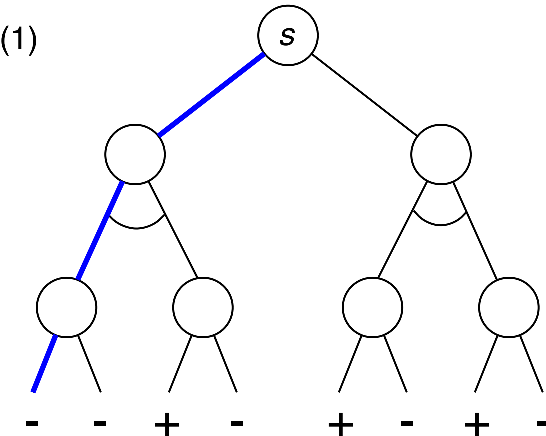
Example: Solved Labeling Using DFS (continued)

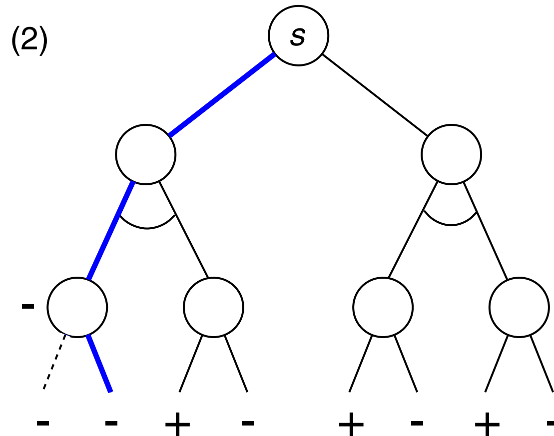


 © Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Example: Solved Labeling Using DFS (continued)

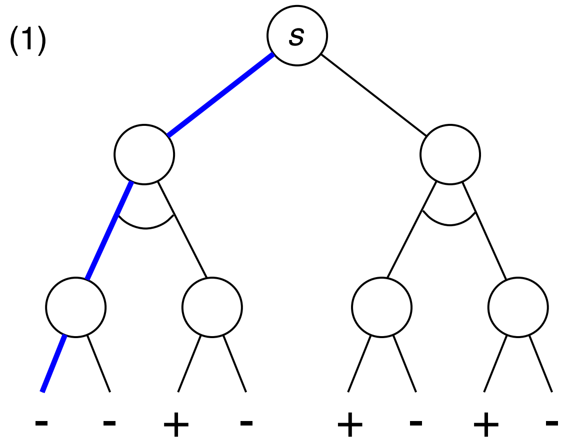


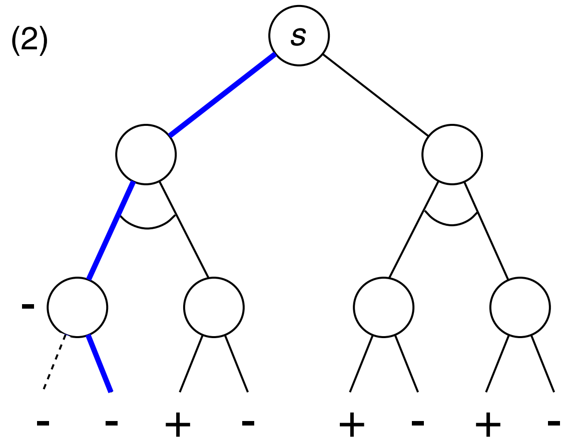


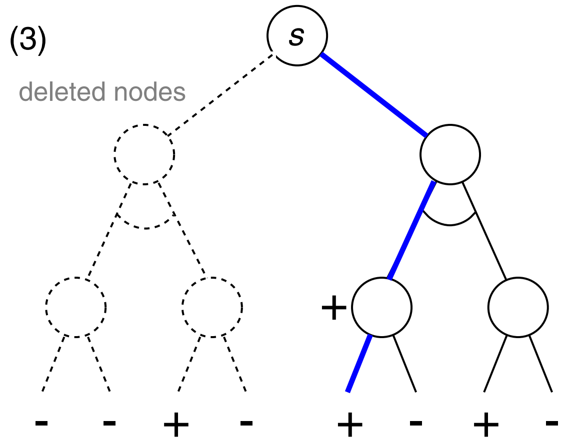
©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Example: Solved Labeling Using DFS (continued)



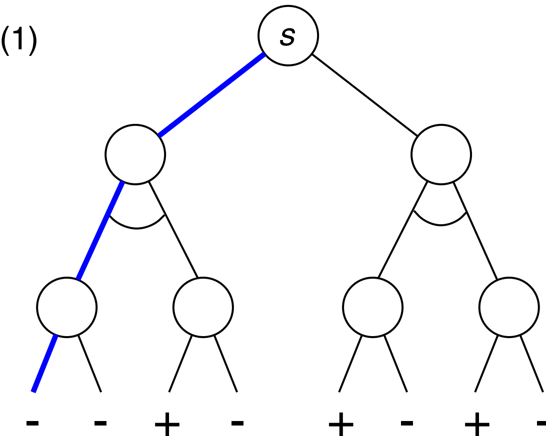


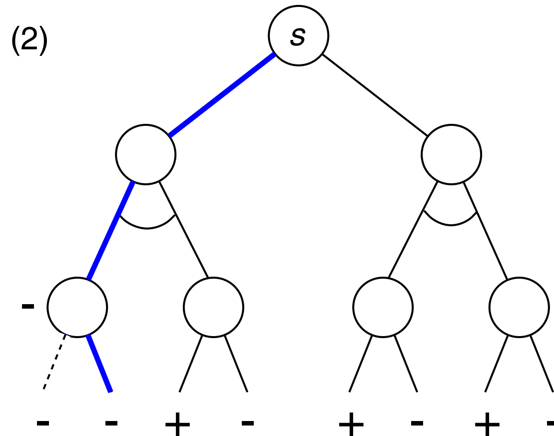


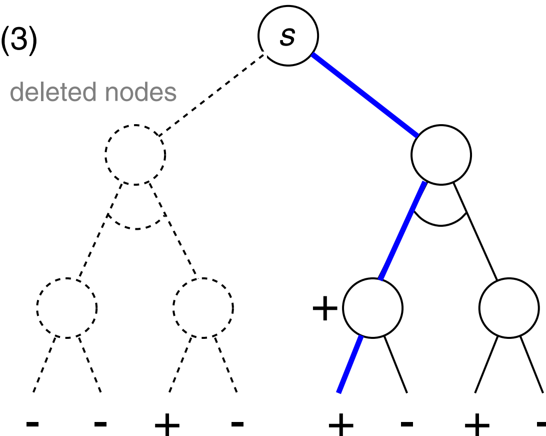
© Stein/Lettmann/Hagen 1998–2021

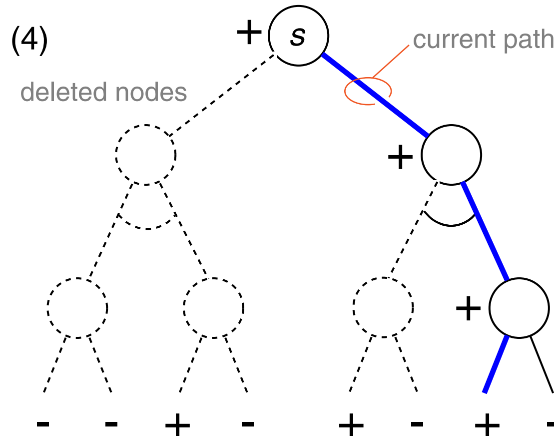
Depth-First Search of AND-OR Graphs

Example: Solved Labeling Using DFS (continued)









©Stein/Lettmann/Hagen 1998–2021

Remarks:

- Information “solved” resp. “unsolvable” for a node is propagated to its parent node following the back-pointer. Once the final label is determined, it can be propagated upwards and search of sibling nodes can in some cases be omitted (see transition from (2) to (3) in the above example).
- Depth-first search realizes a recursive solution tree search, similar to the inductive part of the solution tree definition.
- If a solution tree has to fulfill additional constraints (due to optimality requirements or particularities of the domain), a DFS-based solved labeling can be applied only if the imposed constraint fulfillment can be checked recursively as well.

© Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Solution Tree Labeling

- Information collected during DFS search:

  Solved-labeling procedure:
  The information propagated is Boolean: either “unsolvable” or “solved”.

  Solution-labeling procedure:
  The information propagated is a proof tree or `null`.

- Scope of the analysis by DFS search:

  Solved-labeling procedure:
  The label of the start node s indicates the existence of solution trees.
  ➜ OR-nodes are not fully analyzed once “solved” is known.

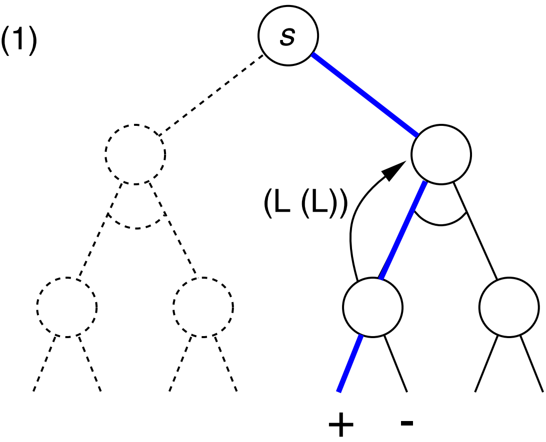
  Solution-labeling procedure:
  If more than one solution tree is required, the information propagated is a list of all possible proof trees. This list may be empty.
  ➜ OR-nodes have to be fully analyzed.

Note: No additional solution constraints are considered here.

©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

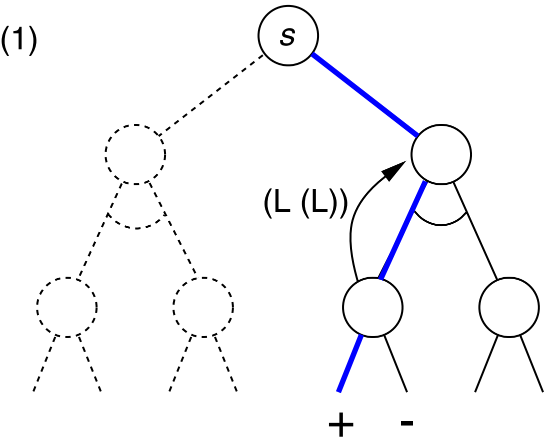
Example: *Solution Labeling* Using DFS

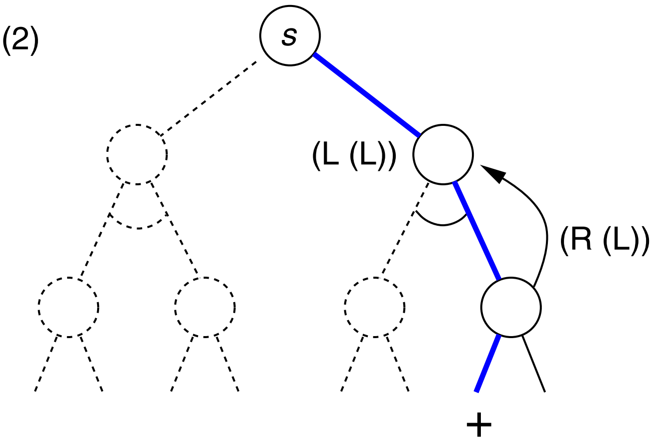


 © Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Example: *Solution Labeling* Using DFS (continued)

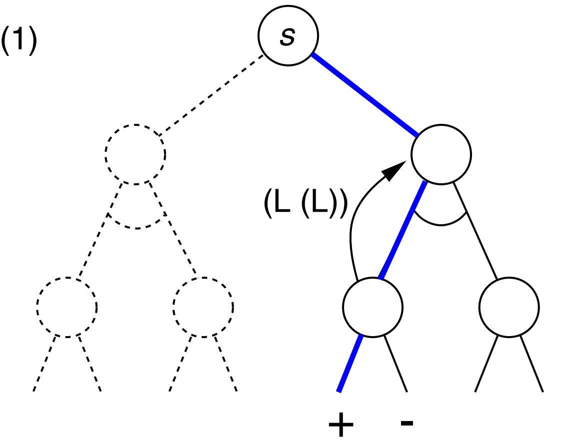


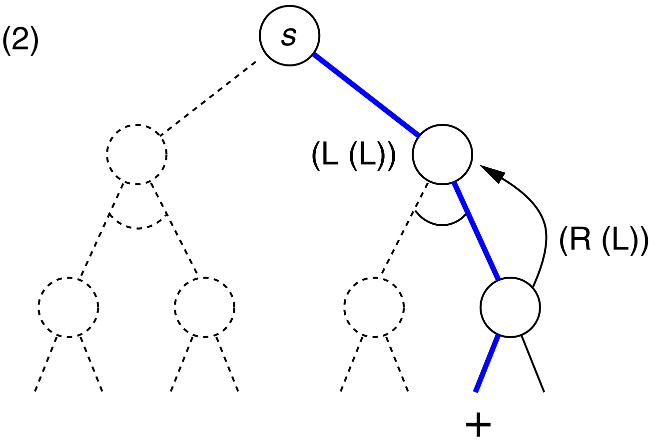


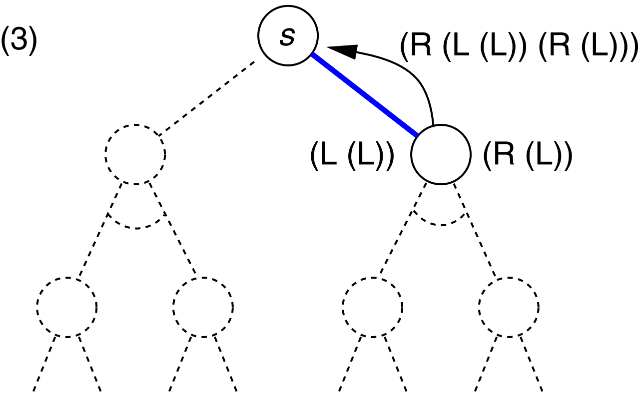
©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Example: *Solution Labeling* Using DFS (continued)



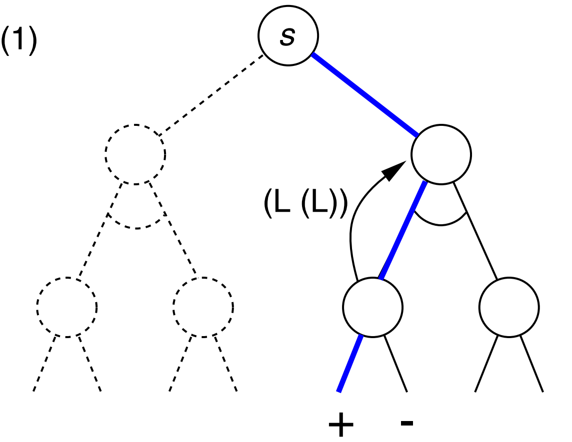


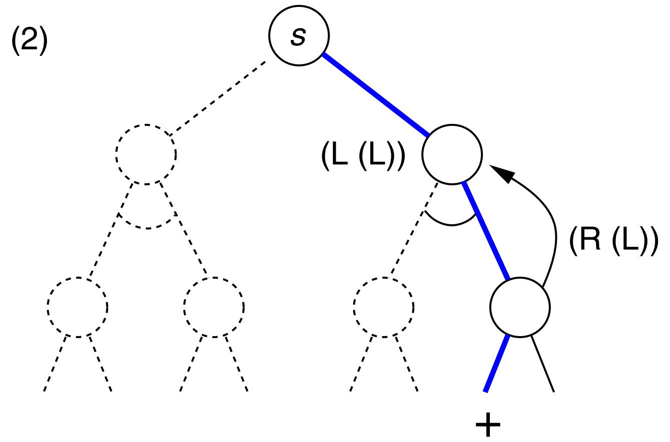


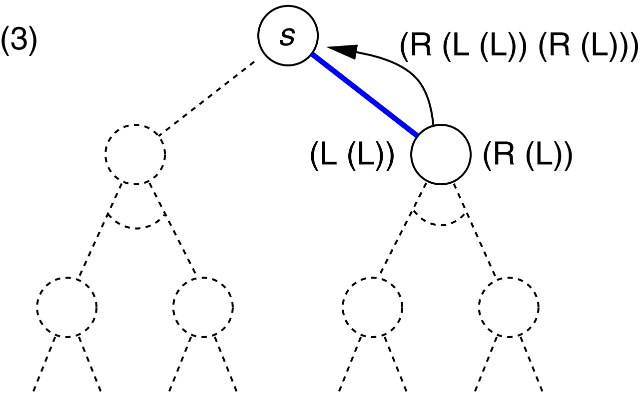
© Stein/Lettmann/Hagen 1998–2021

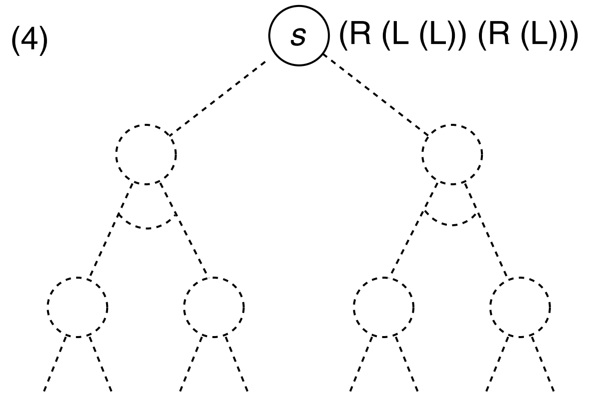
Depth-First Search of AND-OR Graphs

Example: *Solution Labeling* Using DFS (continued)









 © Stein/Lettmann/Hagen 1998–2021

Remarks:

- If in solved_labeling_DFS nodes are discarded soon after their labels have been propagated, a code for reconstructing the solution graph needs to be propagated as well.

- In the shown illustration the code describes the chosen operators (L or R), whereas the nesting level of the parentheses indicates the depth of the tree.

- Instead of propagating a single solution tree, solved_labeling_DFS could also propagate a list of all possible solution trees. In inner OR nodes, unions of these lists have to be built; in inner AND nodes, a Cartesian product is necessary.

- If a solution tree has to fulfill additional constraints (due to optimality requirements or particularities of the domain), a DFS-based solution labeling can be applied only if the imposed constraint fulfillment can be checked recursively as well.

  If a recursive constraint analysis is infeasible, we have to determine all solution trees in solution labeling and check them one by one.

©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Redundant Problem Solving

DFS cannot exploit the compact encoding of problem-reduction representations.

Reason: AND-OR graphs contain only a single instance of identical subproblems, DFS maintains only a *path* from s to the current node.

- ➜ Identical subproblems may be encountered on different paths and solved several times by DFS.
- ➜ If AND-OR graphs are searched with DFS, redundancy cannot be avoided.

© Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

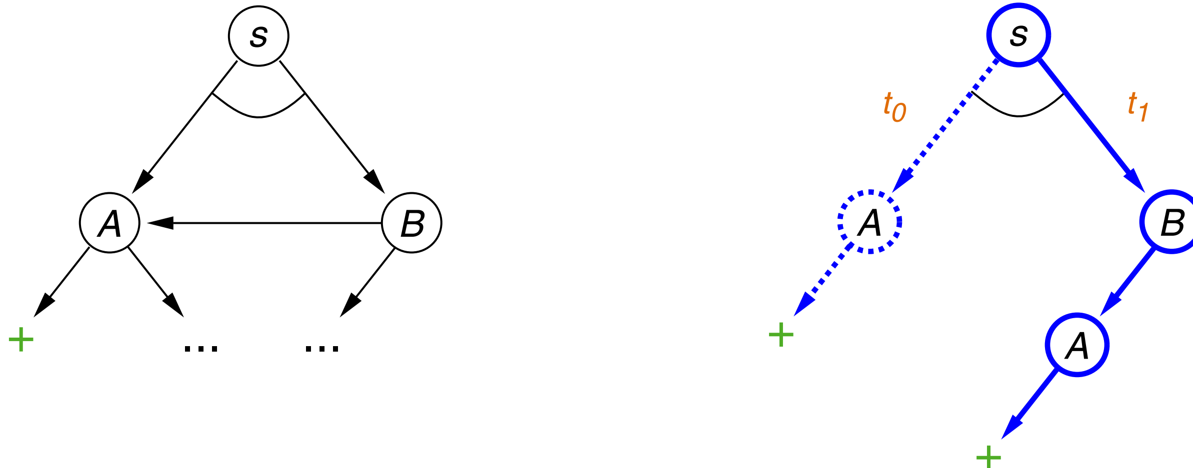
Redundant Problem Solving

DFS cannot exploit the compact encoding of problem-reduction representations.

Reason: AND-OR graphs contain only a single instance of identical subproblems, DFS maintains only a *path* from s to the current node.

- ➜ Identical subproblems may be encountered on different paths and solved several times by DFS.
- ➜ If AND-OR graphs are searched with DFS, redundancy cannot be avoided.

AND-OR graph (left) and its exploration (= unfolding) with DFS (right):



 ©Stein/Lettmann/Hagen 1998–2021

Remarks:

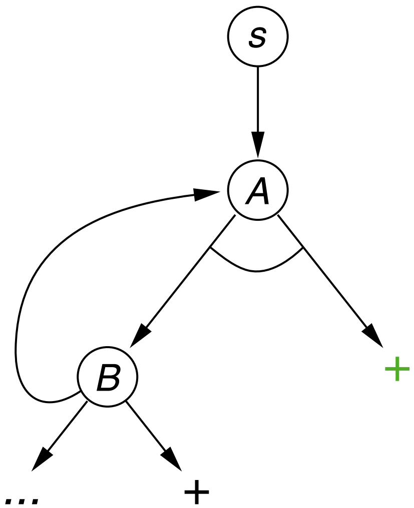
- With a complete occurrence check (i.e., an occurrence check that considers all explored nodes so far), recurring subproblems can be identified.
- A complete occurrence check entails a memory consumption similar to BFS, which renders such a check very unattractive—if not impossible. In most cases, the small memory footprint of DFS along with a usually manageable effort for resolving (a small number of) recurring problems makes DFS superior to BFS when searching AND-OR graphs.
- Recall that by omitting an occurrence check, we cannot identify loops in the search space graph.
- The path from s to the current node (i.e., the partial solution path), is also called *traversal path* or back-pointer path [Pearl 84]. Recall that DFS stores the partial solution path on the CLOSED list (with exception of the last node which is in OPEN).
- A *partial occurrence check*, which is limited to the nodes on the traversal path, is both computationally manageable and prohibits infinite loops caused by recurrent problem solving: Nodes that are encountered twice on the traversal path are discarded.

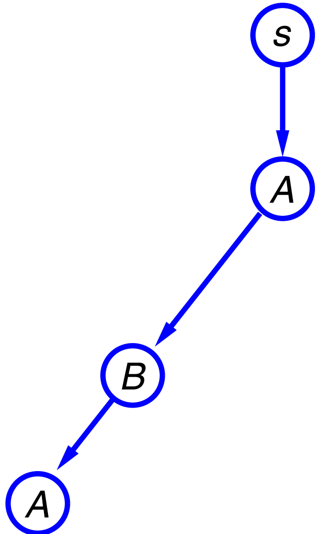
©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Dealing with Cycles (1)

AND-OR graph (left) and its exploration (= unfolding) with DFS (right):



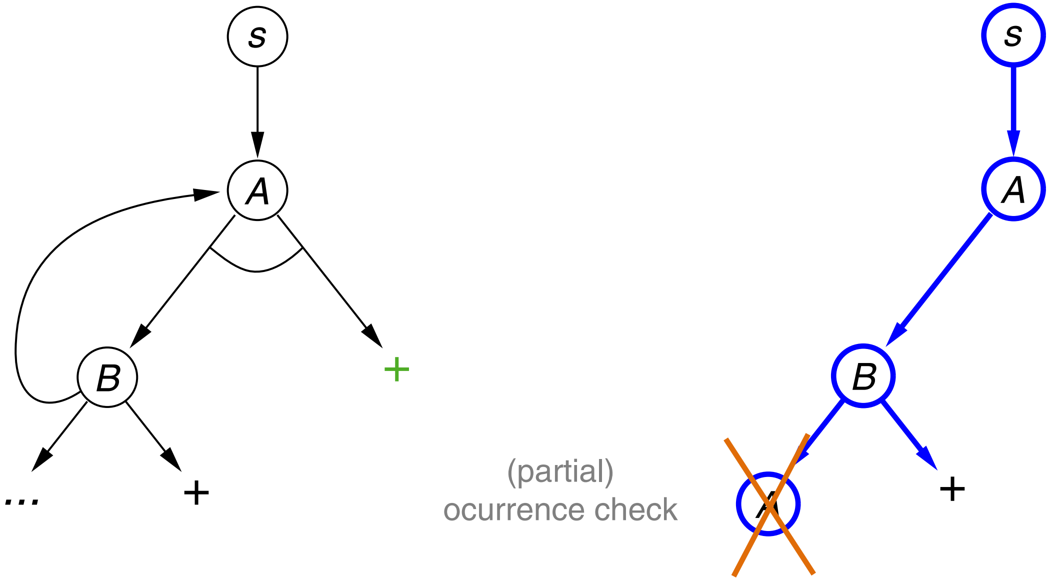


 ©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Dealing with Cycles (1)

AND-OR graph (left) and its exploration (= unfolding) with DFS (right):

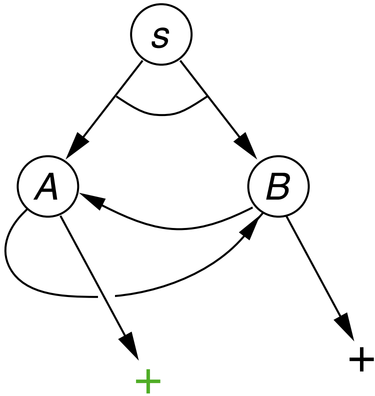


 ©Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Dealing with Cycles (2)

AND-OR graph (left) and its exploration (= unfolding) with DFS (right):

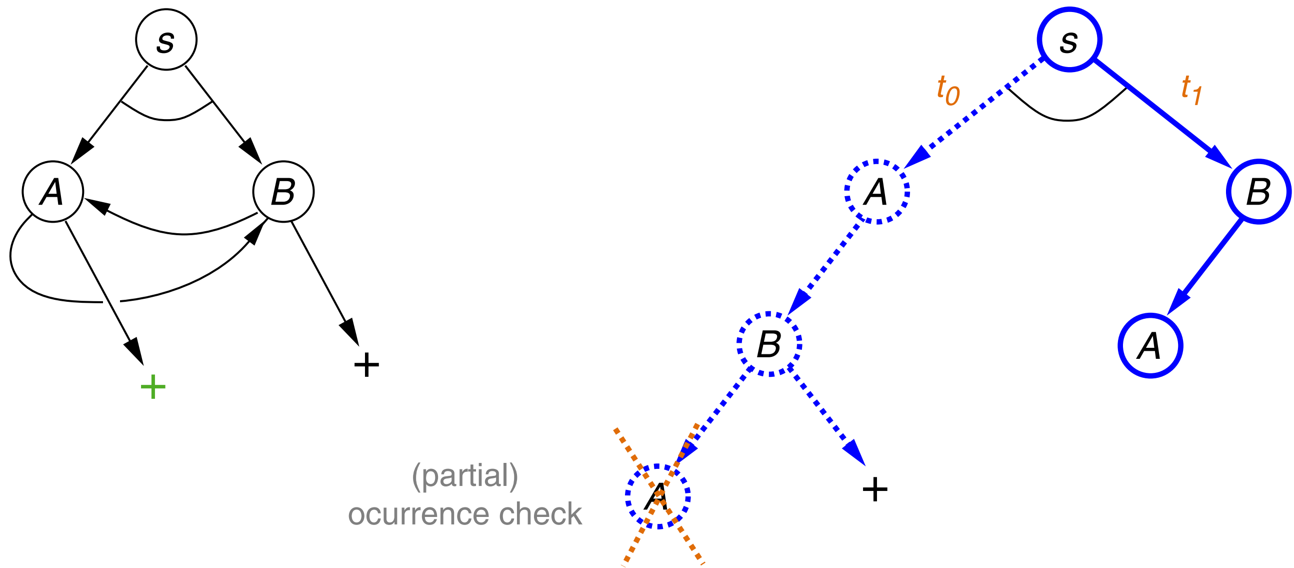


 © Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Dealing with Cycles (2)

AND-OR graph (left) and its exploration (= unfolding) with DFS (right):



 © Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Dealing with Cycles (2)

AND-OR graph (left) and its exploration (= unfolding) with DFS (right):

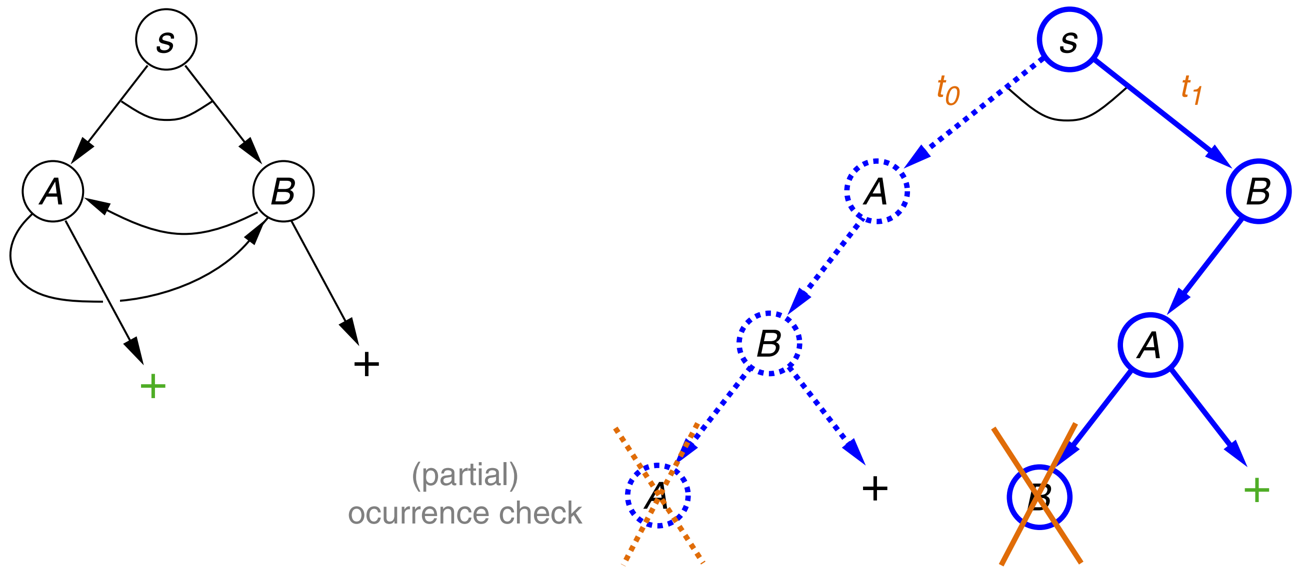
(partial)
ocurrence check

© Stein/Lettmann/Hagen 1998–2021

Depth-First Search of AND-OR Graphs

Dealing with Cycles (2)

AND-OR graph (left) and its exploration (= unfolding) with DFS (right):



 © Stein/Lettmann/Hagen 1998–2021

Remarks:

- Q. Why does the discarding of recurring nodes (partial occurrence check) not compromise the completeness of solved-labeling?
- Recapitulation. When searching an AND-OR graph with DFS, a partial occurrence check cannot address the issue of redundant problem solving in general.
- Given the previous AND-OR graph (2), breadth-first search would have found a solution in depth 2 already.

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search Basics

Solution-Tree Bases in AND-OR Graphs

Definition 18 (Base of a Solution Tree for an AND-OR Graph)

Let G be an AND-OR graph, and let n be a node in G. A *solution-tree base H for n in G* is defined in the same way as a solution tree for n in G except for Condition 5 on leaf nodes and Condition 6 on additional solution constraints. Leaf nodes must not be dead ends.

Usage:

- Solution Trees

  We are interested in finding a solution tree H for the start node s in G.

- Solution-Tree Bases

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

Disadvantage:

- Multiple Occurrences of Nodes

  The same problem (represented by a node $n \in G$) can have to be solved multiple times (in different ways), even in a single solution tree.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search Basics

Requirements for an Algorithmization of AND-OR Graph Search

Properties of Search Space Graphs

1. G is a canonical problem-reduction graph (directed AND-OR graph).
2. 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.
3. For each node n its type $l(n)$ (AND node / OR node) is known.
4. Computing successors always returns new clones of nodes.
5. G is locally finite.
6. For G a set Γ of goal nodes is given; in general Γ will not be singleton. Goal nodes are terminal nodes (leaf nodes) in G.
7. G has a function $\star(.)$ returning true if a solution base is a solution tree.

Task:

- Determine in G a solution tree for s.

 ©Stein/Lettmann/Hagen 1998–2021

Remarks:

- Node expansion works in the same way for both types of nodes.
- Minimum requirement for a solution tree is that each terminal node of a solution-tree base is a goal node.
- 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 problem-reduction graph to a tree with root s.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search Basics

Generic Schema for AND-OR-Graph Tree Search Algorithms

... from a solution-tree-base-oriented perspective:

1. Initialize solution-tree-base storage.
2. Loop.
   (a) Using some strategy select a solution-tree base to be extended.
   (b) Using some strategy select an unexpanded node in this base.
   (c) According to the node type extend the solution-tree base by successor nodes in any possible way and store the new candidates.
   (d) Determine whether a solution tree is found.

Usage:

- Search algorithms following this schema maintain a set of solution-tree bases.
- Initially, only the start node s is available; node expansion is the basic step.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search Basics

Algorithm: Generic_AND-OR_Tree

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

Output: A solution tree b or the symbol *Fail*.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Basics [Generic_OR]

```
Generic_AND-OR_Tree(s, successors, ⋆)

1.  b = solution_tree_base(s);   // Initialize solution-tree base b.
    IF ⋆(b) THEN RETURN(b);   // Check if b is solution tree.
2.  push(b, OPEN);    // Store b on OPEN waiting for extension.
3.  LOOP
4.    IF (OPEN = ∅) THEN RETURN(Fail);
5.    b = choose(OPEN);    // Choose a solution-tree base b from OPEN.
      remove(b, OPEN);    // Delete b from OPEN.
      n = choose(b);    // Choose unexpanded tip node in b.
6.    IF (is_OR_node(n))
      THEN
        FOREACH n' IN successors(n) DO    // Expand n.
          b' = add(copy(b), n, {n'});    // Extend b by node n', edge (n, n').
          IF ⋆(b') THEN RETURN(b');    // Check if b' is solution tree.
          push(b', OPEN);    // Store b' on OPEN waiting for extension.
        ENDDO
      ELSE
        b' = add(copy(b), n, successors(n));  // Extend b by ALL succ. nod./edges.
        IF ⋆(b') THEN RETURN(b');    // Check if b' is solution tree.
        push(b', OPEN);    // Store b' on OPEN waiting for extension.
      ENDIF
7.  ENDLOOP
```


©Stein/Lettmann/Hagen 1998–2021

Remarks:

- Algorithm Generic_AND-OR_Tree takes a solution-tree-base-oriented perspective.

  Generic_AND-OR_Tree maintains and manipulates solution-tree bases (which are AND-OR trees with root s).

- Remaining problems which still are to be solved can be found only among the tip nodes. So, a solution-tree base may contain multiple open problems.

- Function *add*$($*copy*$(b), n, .)$ returns a representation of the extended solution-tree base: node instances in *successors*(n) are added and the edges from n to these nodes. For that purpose, b is copied. So each solution-tree base is represented separately, although they often share initial subtrees.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Efficient Storage of Solution-Tree Bases

OR-graph search:

- Solution bases are paths.
- Backpointers allow the recovery of a path starting from its tip node.

AND-OR-graph search:

- Idea: Reusing the back-pointer concept of `Basic_OR`

➜ The back-pointer structure for a single solution-tree may already have several leaf nodes.

➜ Problem: A solution-tree base is not identified by a single tip node.

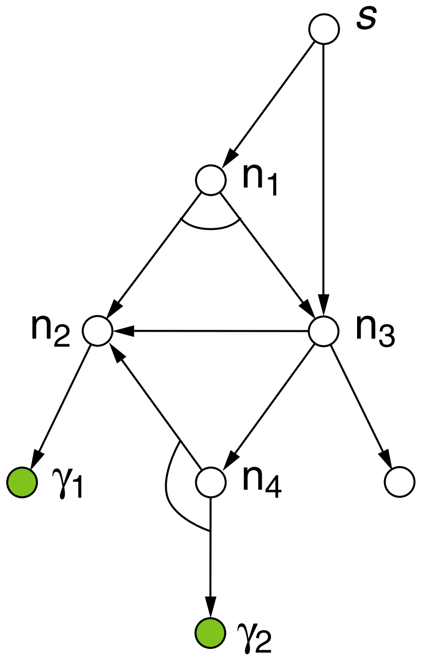
- For each expanded node instance forward pointer to the successors node instances in the solution-tree base are stored, i.e., information about outgoing edges in G is stored.

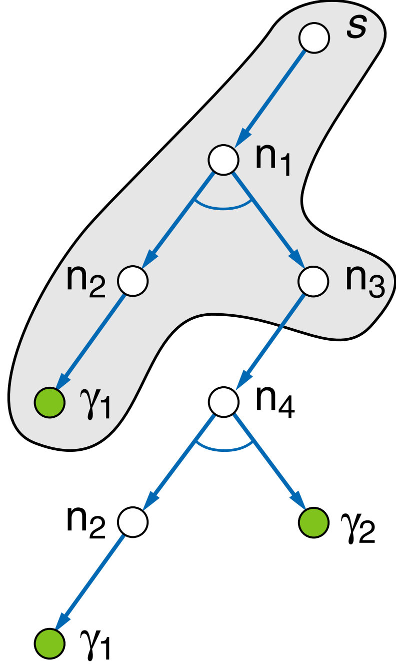
©Stein/Lettmann/Hagen 1998–2021

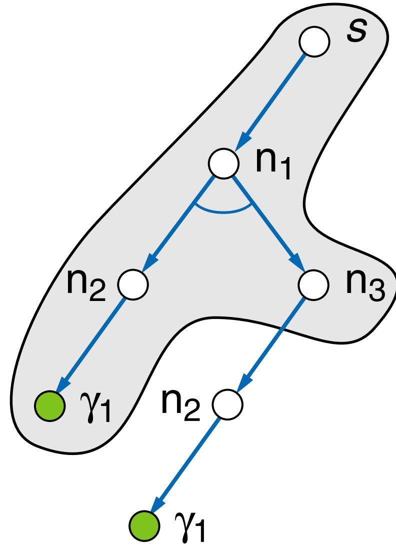
AND-OR Graph Search

Efficient Storage of Solution-Tree Bases (continued)

Example: Solution trees for AND-OR graphs







Q. Which node instances might be shared, which are kept separate when using back-pointers?

A. Doesn't matter at the moment, just use back-pointers.

 © Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Efficient Storage of Solution-Tree Bases (continued)

OR-graph search:

- All solution bases are stored in a single back-pointer structure.
- By sharing of initial parts, the maintained solution bases form a traversal tree.
- A solution base is uniquely identified by the corresponding tip node in the traversal tree.

AND-OR-graph search:

- All solution-tree bases are maintained in a shared storage that is a combination of
  - a single back-pointer structure containing all necessary node instances of all solution-tree bases as well as the corresponding back-pointers and
  - an explicit graph G_e containing all node instances of all solution-tree bases and the corresponding edges to successor node instances for expanded node instances (finite initial part of a tree unfolding of G = traversal tree).

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Identifying Solution Trees and Solution-Tree Bases

AND-OR-graph search:

➜ Problem: How is a solution-tree base identified in G_e?

- solved_labeling_DFS can be adapted to determine solution-tree bases in G_e: nodes labeled "*unsolvable*" are considered as dead ends.

➜ Problem: How is detected whether a solution-tree that is contained in G_e?

- solved_labeling_DFS propagates "*solved*" resp. "*unsolvable*" information along back-pointers.
  If there are no further solution constraints, a solution tree is contained if the start node s is labeled "*solved*".

➜ Problem: How is a solution tree identified in G_e?

- solved_labeling_DFS can be adapted to determine solution trees in G_e: nodes not labeled "*solved*" are considered as dead ends.
  If there are no further solution constraints, the solution-tree bases determined are solution trees.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Algorithm: Basic_AND-OR_Tree

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.

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

Subroutines: *is_solved*(n). Predicate that is *True* if n is labeled "solved" or n is a goal node.
propagate_label(n). Function that propagates node label "solved" along back-pointers.

© Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search [Basic_OR]

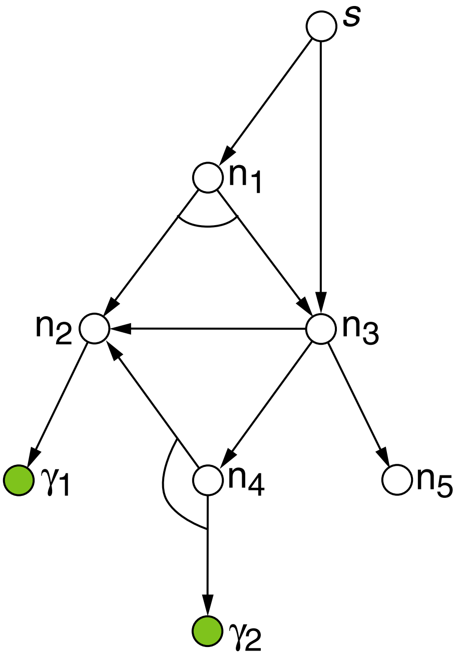
```
Basic_AND-OR_Tree(s, successors, is_solved)

  1.   insert(s, OPEN);  add_node(s, G_e);    // G_e is the explored part of G.
  2.   LOOP
  3.     IF (OPEN = ∅) THEN RETURN(Fail);
 4.a     H = choose_solution_tree_base(s, G_e);    // Solution base for s in G_e.
 4.b     n = choose(OPEN ∩ H);    // Choose OPEN non-goal tip node in H.
         remove(n, OPEN);  push(n, CLOSED);
  5.     FOREACH n' IN successors(n) DO

           add_node(n', G_e);    // n' encodes a new instance of a node in G.
           insert(n', OPEN);

           add_edge((n, n'), G_e);  add_backpointer(n', n);
           IF is_solved(n')    // Is n' goal node or labeled solvable?
           THEN
             propagate_label(n');
             IF is_solved(s) THEN RETURN(compute_solution_graph(G_e));
           ENDIF

         ENDDO
  6.   ENDLOOP
```


©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Illustration of Basic_AND-OR

AND-OR graph G



Maintained graph G_e

Backpointer structure

Goal node

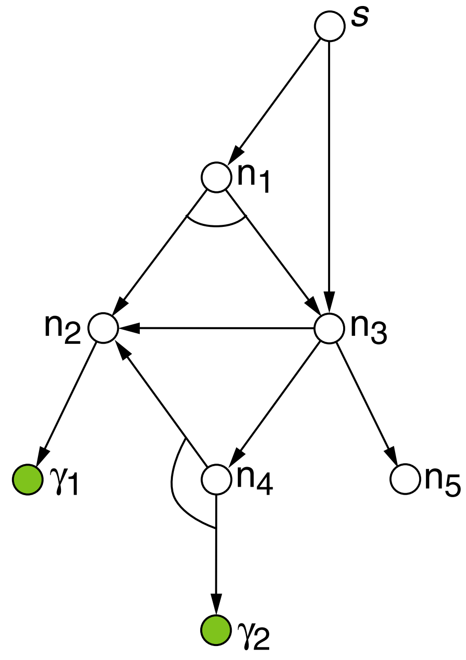
OPEN node / CLOSED node

© Stein/Lettmann/Hagen 1998–2021

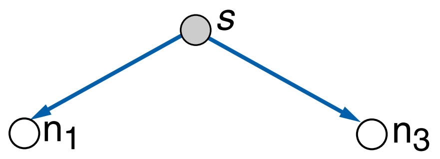
AND-OR Graph Search

Illustration of Basic_AND-OR_Tree (continued)

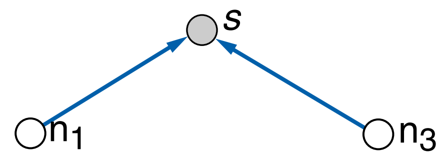
AND-OR graph G



Maintained graph G_e



Backpointer structure



Goal node   OPEN node / CLOSED node

Which solution-tree bases are maintained in G_e?

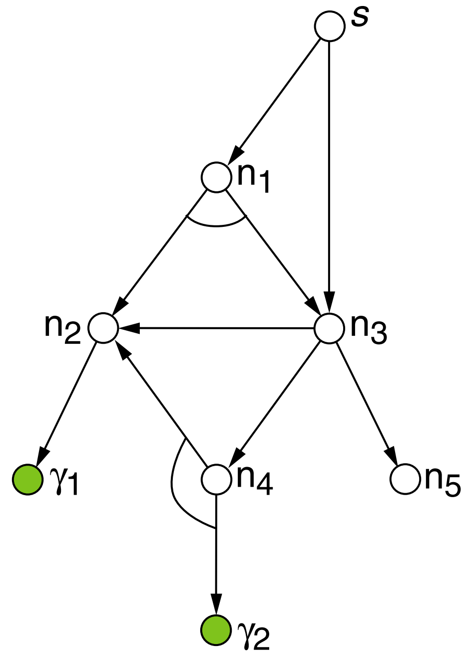
Which solution-tree base and which node in it was selected?

 © Stein/Lettmann/Hagen 1998–2021

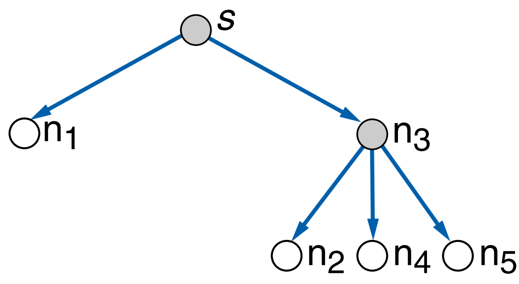
AND-OR Graph Search

Illustration of Basic_AND-OR_Tree (continued)

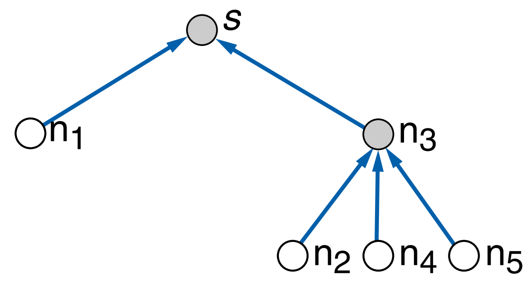
AND-OR graph G



Maintained graph G_e



Backpointer structure



Goal node

OPEN node / CLOSED node

Which solution-tree bases are maintained in G_e?

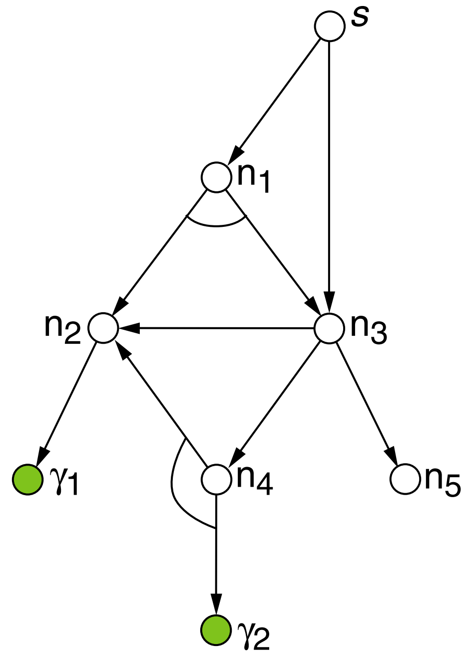
Which solution-tree base and which node in it was selected?

©Stein/Lettmann/Hagen 1998–2021

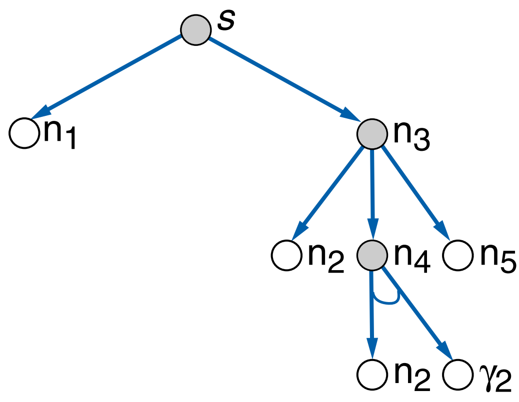
AND-OR Graph Search

Illustration of Basic_AND-OR_Tree (continued)

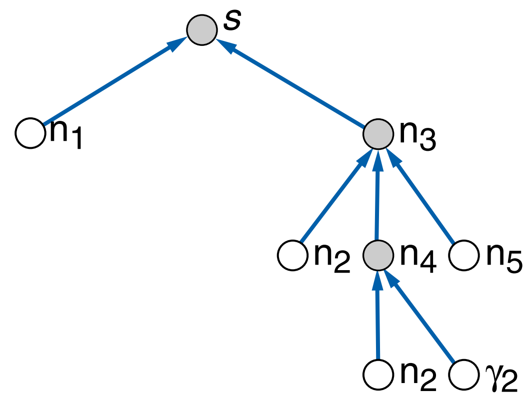
AND-OR graph G



Maintained graph G_e



Backpointer structure



Goal node

OPEN node / CLOSED node

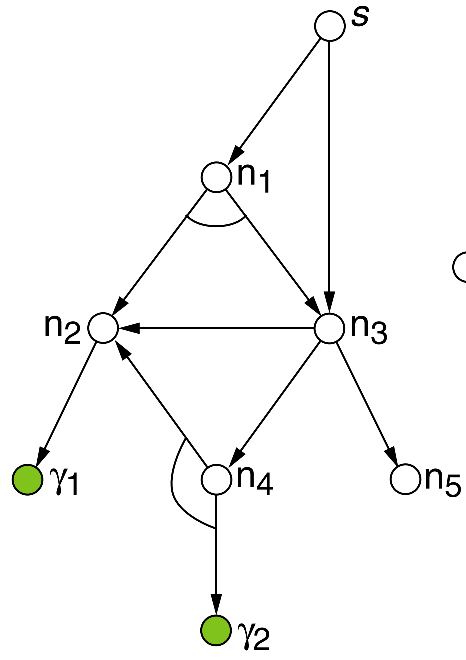
Which solution-tree bases are maintained in G_e?

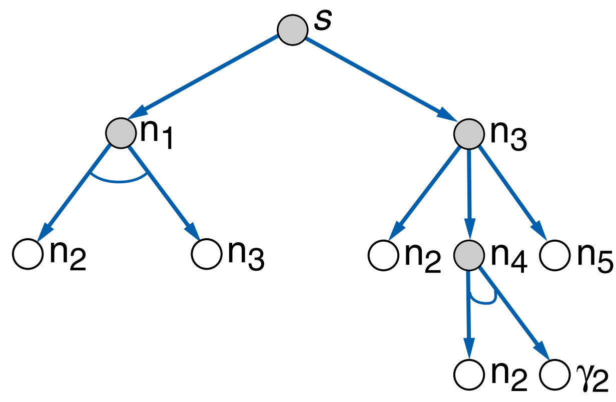
Which solution-tree base and which node in it was selected?

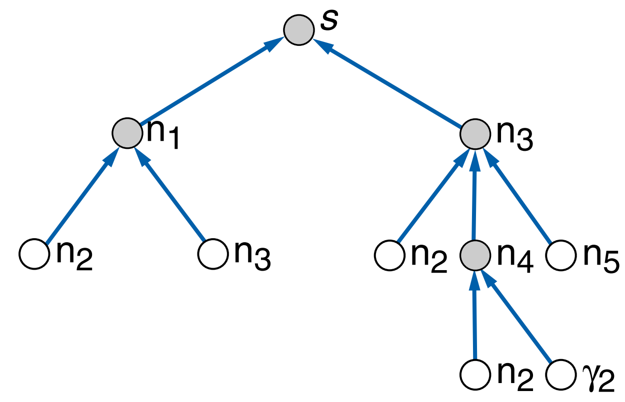
©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Illustration of Basic_AND-OR_Tree (continued)







Goal node

OPEN node / CLOSED node

Which solution-tree bases are maintained in G_e?

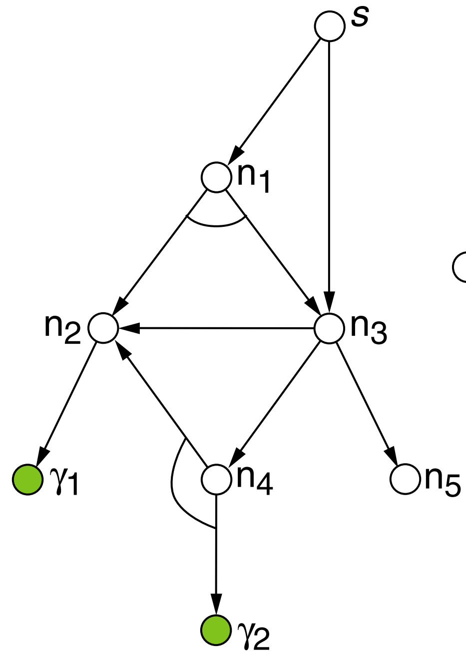
Which solution-tree base and which node in it was selected?

©Stein/Lettmann/Hagen 1998–2021

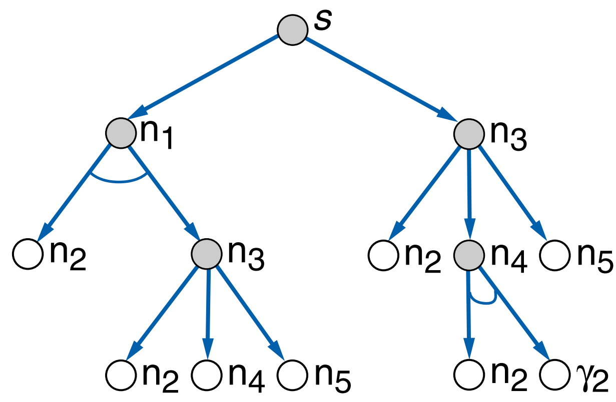
AND-OR Graph Search

Illustration of Basic_AND-OR_Tree (continued)

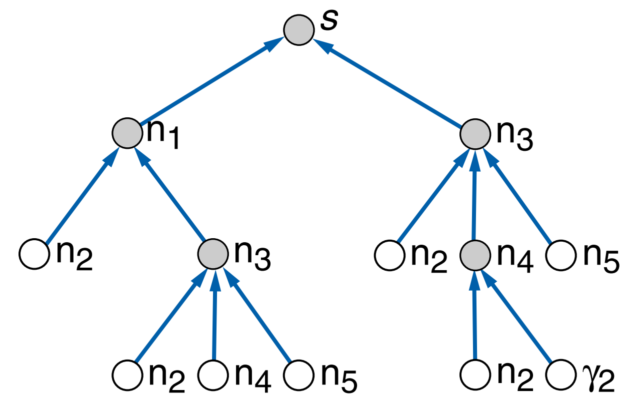
AND-OR graph G



Maintained graph G_e



Backpointer structure



Goal node

OPEN node / CLOSED node

Which solution-tree bases are maintained in G_e?

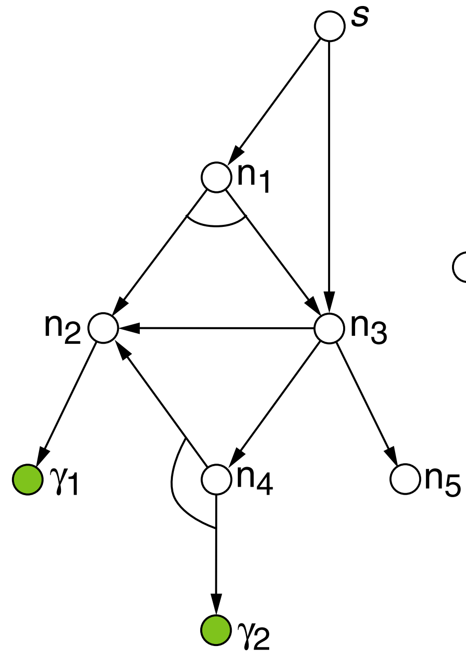
Which solution-tree base and which node in it was selected?

©Stein/Lettmann/Hagen 1998–2021

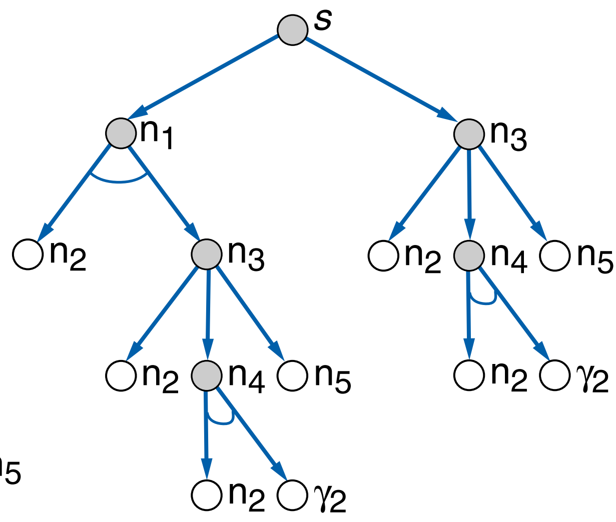
AND-OR Graph Search

Illustration of Basic_AND-OR_Tree (continued)

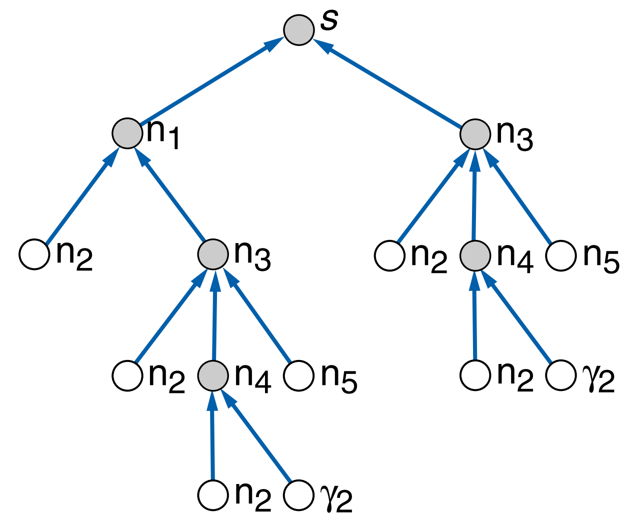
AND-OR graph G



Maintained graph G_e



Backpointer structure



Goal node

OPEN node / CLOSED node

Which solution-tree bases are maintained in G_e?

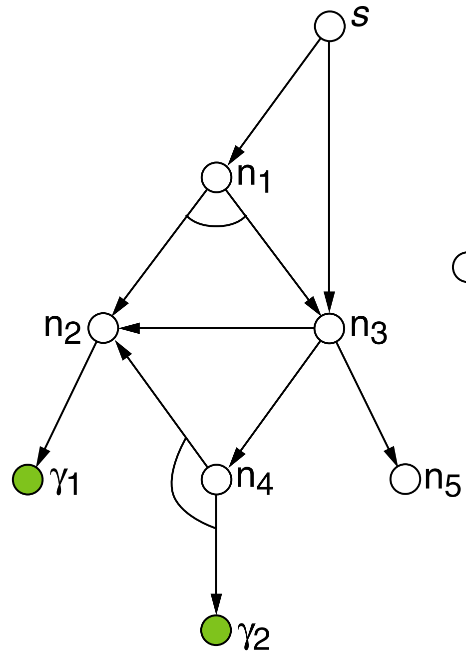
Which solution-tree base and which node in it was selected?

©Stein/Lettmann/Hagen 1998–2021

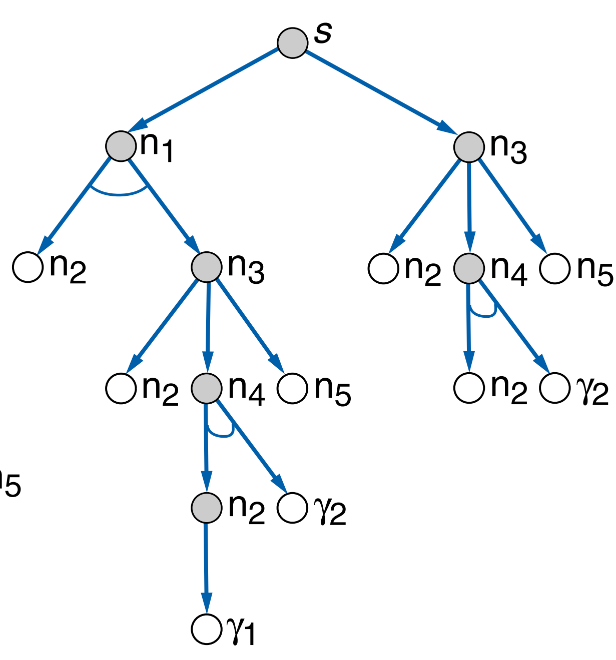
AND-OR Graph Search

Illustration of Basic_AND-OR_Tree (continued)

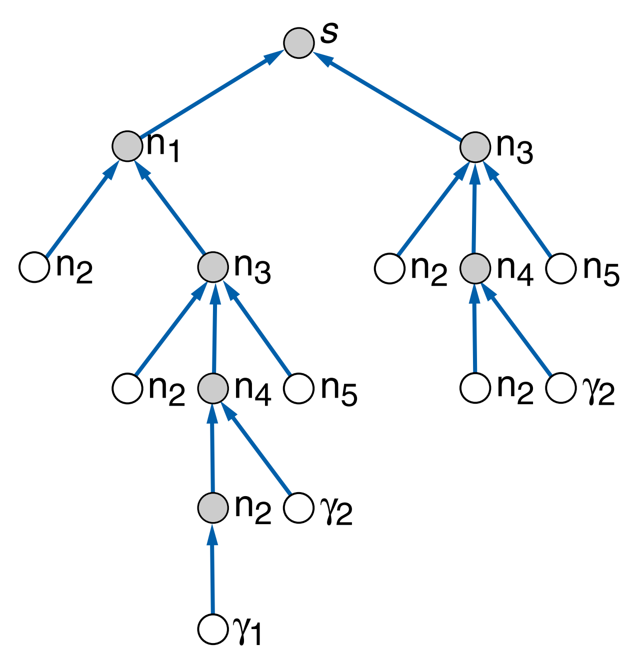
AND-OR graph G



Maintained graph G_e



Backpointer structure



Goal node

OPEN node / CLOSED node

Which solution-tree bases are maintained in G_e?

Which solution-tree base and which node in it was selected?

 ©Stein/Lettmann/Hagen 1998–2021

Remarks:

- Due to the structure sharing approach in algorithm `Basic_AND_OR_Tree`, node expansions modify all solution-tree bases that share the expanded node.
- Solution-tree bases are identified in G_e by an adaption of `solved_labeling_DFS` on the fly. Leaf nodes of the solution-tree bases are leaf nodes in G_e.
- A solution-tree base H for s that needs expansion is computed from G_e. H is maximal in the sense that G_e contains no solution-tree base H' for s such that H is a proper subgraph of H'. So tip nodes in H are tip nodes in G_e.
- Of course, labels "*solved*" resp. "*unsolvable*" can be propagated using the back-pointer structure maintained in `Basic_AND_OR_Tree`. A separate DFS search in G_e is not required.

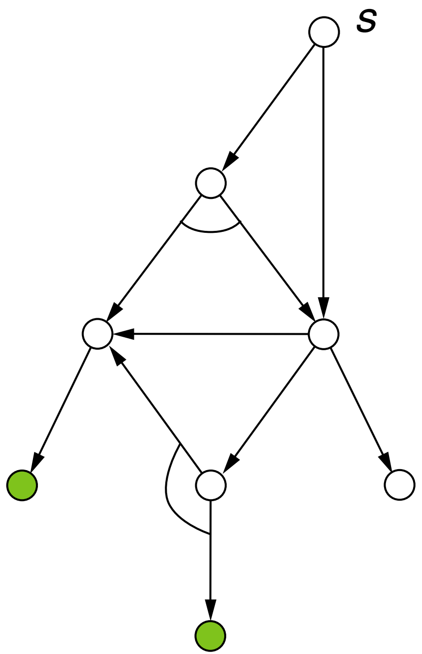
 © Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

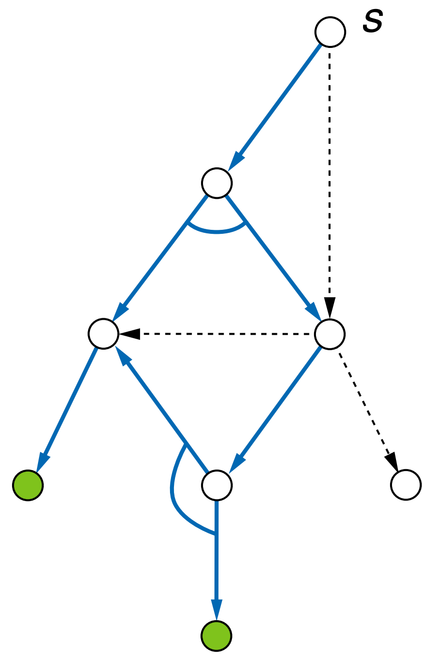
Enfolding of Solution Trees: Compact Representation as AND-OR Graph

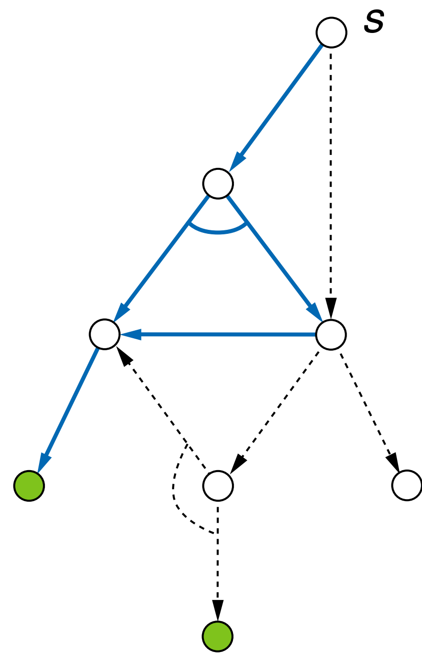
Advantage: Avoiding multiple solutions to the same problem.

AND-OR graph example:



Solution graphs generated by enfolding:





Idea: Multiple instances of nodes from the AND-OR graph G are merged.

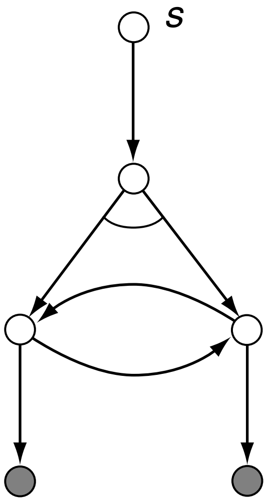
➜ The resulting AND-OR graph is a subgraph of the AND-OR graph G.

 ©Stein/Lettmann/Hagen 1998–2021

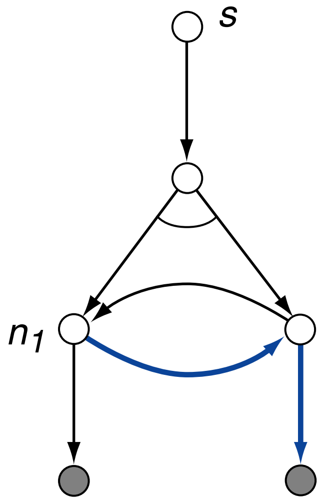
AND-OR Graph Search

Enfolding of Solution Trees

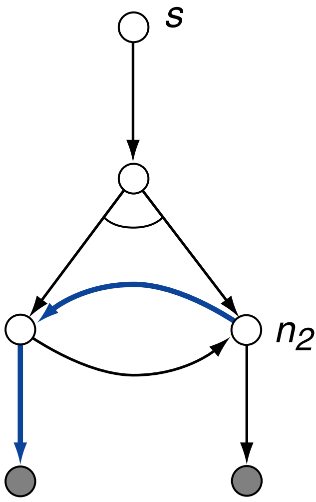
Enfolding of solution trees can result in cyclic AND-OR graphs (no unique structure):



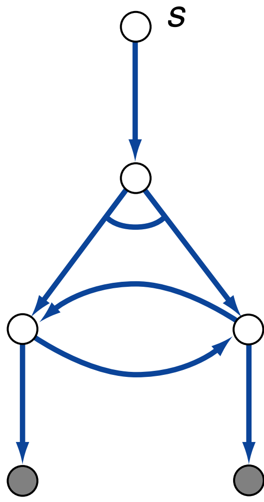
AND-OR graph



Solution graph for n_1



Solution graph for n_2



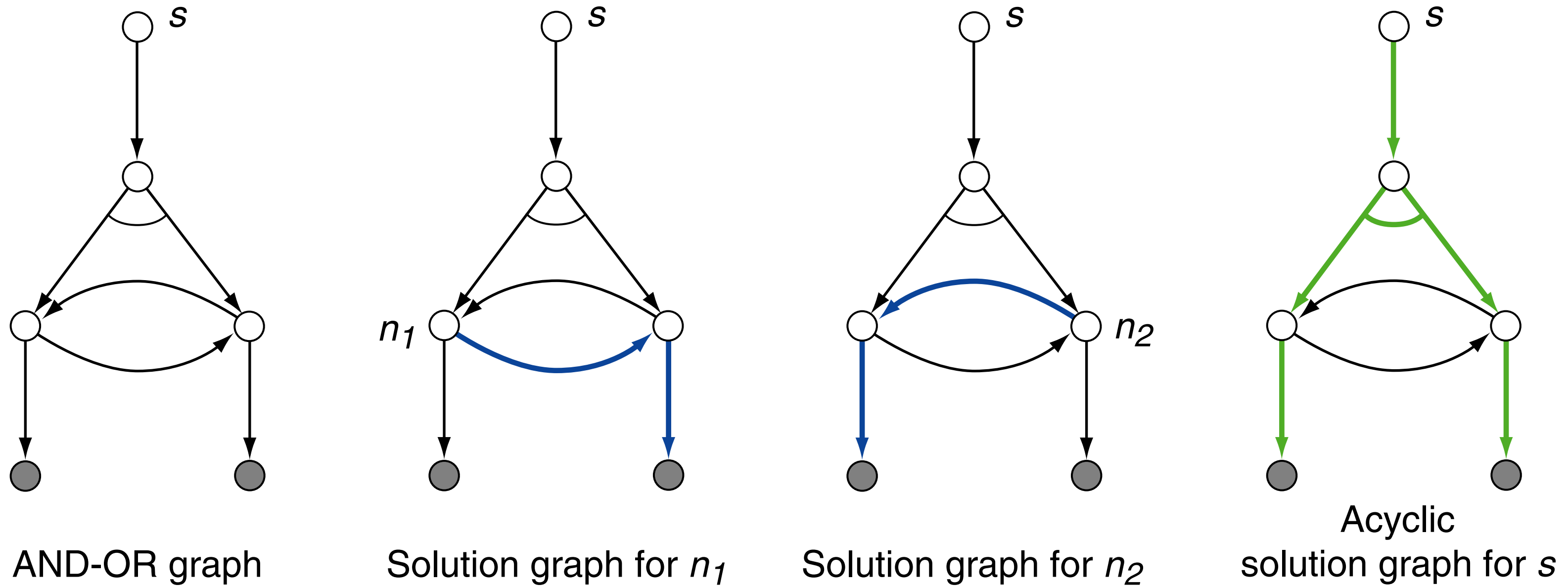
Solution graph for s ??

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Enfolding of Solution Trees

Enfolding of solution trees can result in cyclic AND-OR graphs (no unique structure):



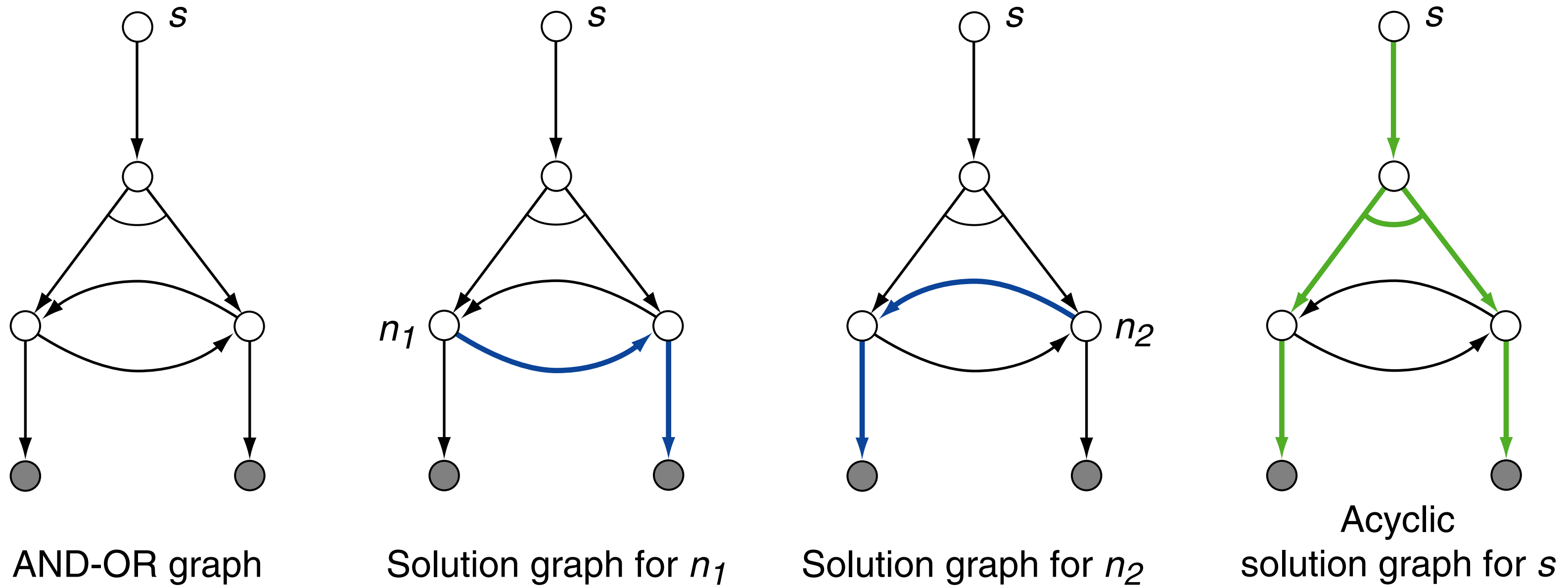
Consequence: AND-OR solution graphs are required to be acyclic.

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Enfolding of Solution Trees

Enfolding of solution trees can result in cyclic AND-OR graphs (no unique structure):



Consequence: AND-OR solution graphs are required to be acyclic.

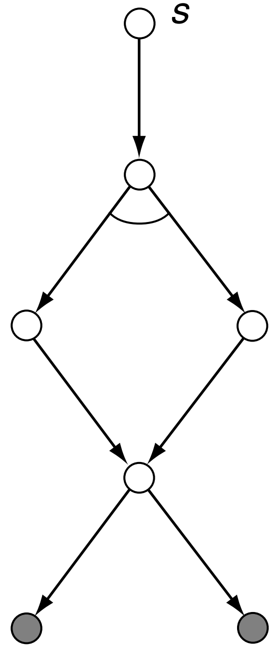
Problem: Folding AND-OR solution trees can result in cyclic AND-OR graphs.

 ©Stein/Lettmann/Hagen 1998–2021

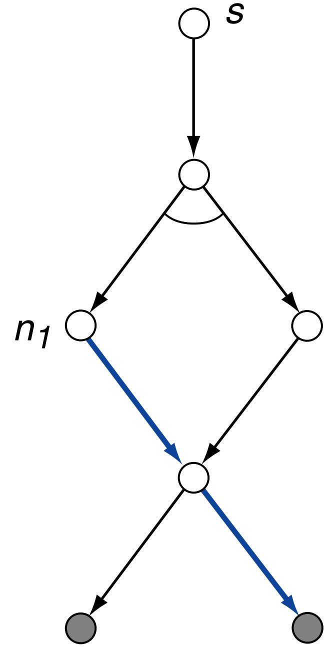
AND-OR Graph Search

Enfolding of Solution Trees (continued)

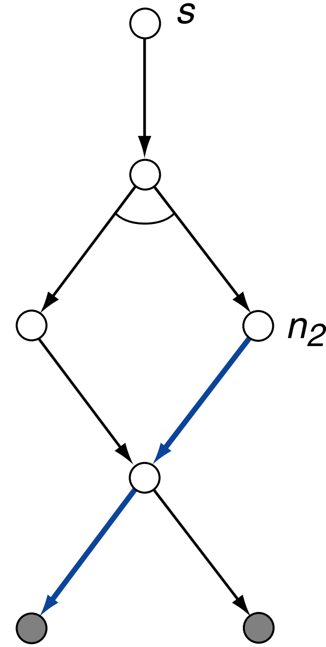
Synthesized solution graphs are not necessarily minimum:



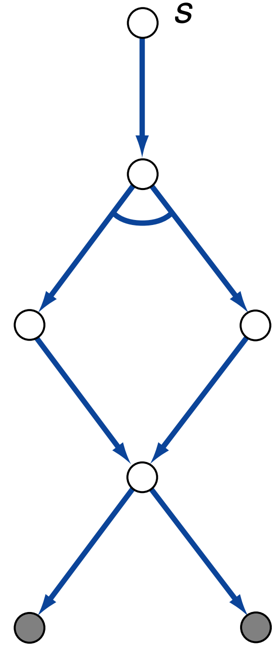
AND-OR graph



Solution graph for n_1



Solution graph for n_2



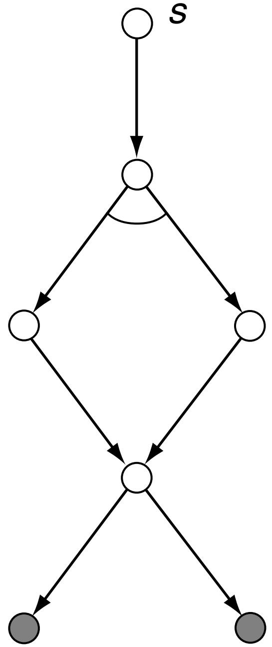
Solution graph for s ??

© Stein/Lettmann/Hagen 1998–2021

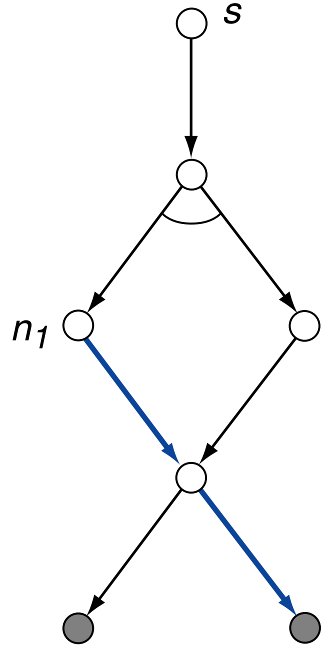
AND-OR Graph Search

Enfolding of Solution Trees (continued)

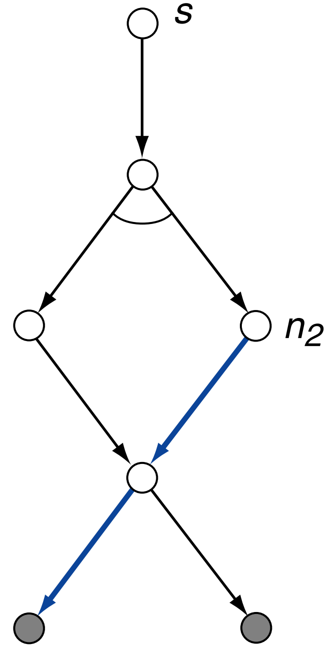
Synthesized solution graphs are not necessarily minimum:



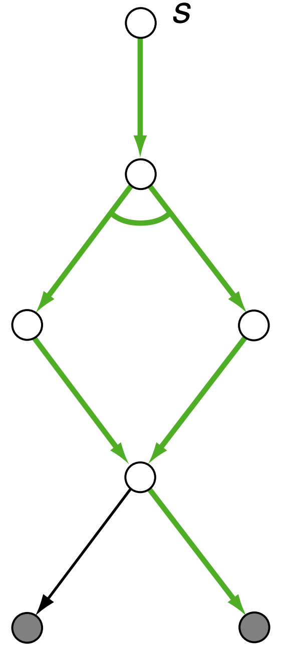
AND-OR graph



Solution graph for n_1



Solution graph for n_2



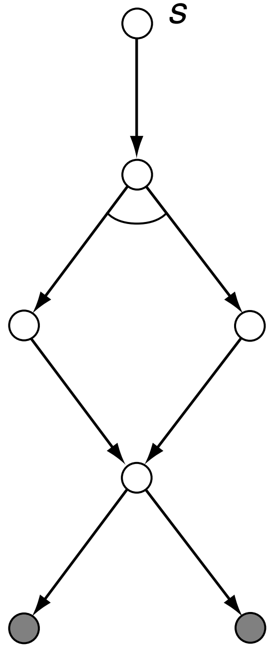
Minimum solution graph for s

©Stein/Lettmann/Hagen 1998–2021

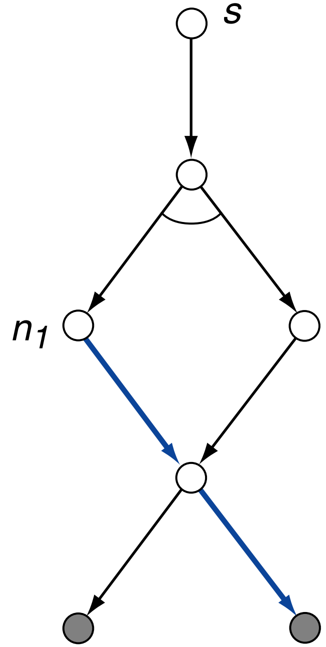
AND-OR Graph Search

Enfolding of Solution Trees (continued)

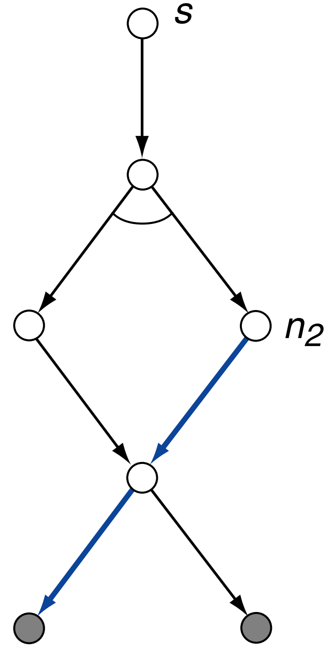
Synthesized solution graphs are not necessarily minimum:



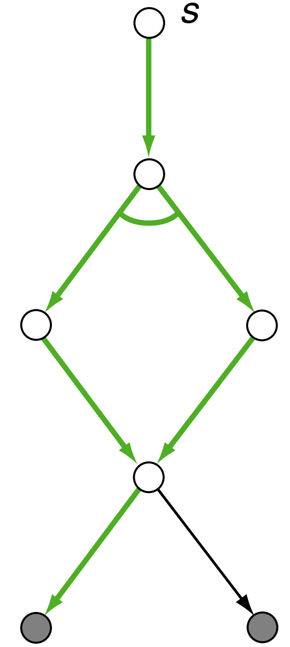
AND-OR graph



Solution graph for n_1



Solution graph for n_2



Minimum solution graph for s

Problem: A definition of solution graphs will not be constructive.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Solution Graphs in AND-OR Graphs [Solution Path in OR Graphs] [Solution Tree in AND-OR Graphs]

Definition 19 (Solution Graph / Solution Base in AND-OR Graphs)

Let G be an AND-OR graph, and let n be a node in G. Also, let some additional solution constraints be given. A graph H is called a *solution graph for* n *in* G iff ($\leftrightarrow$) the following conditions hold:

1. H is finite and acyclic.
2. H contains the node n (as start node).
3. If H contains an inner OR node n', then H contains a (single) successor node of n' of G and the corresponding edge from n to n'.
4. If H contains an inner AND node n', then H contains all the successor nodes of n' of G and all corresponding directed edges.
5. The leaf nodes (terminal nodes) in H are goal nodes of G.
6. H satisfies the solution constraints.
7. H is minimal: it contains no additional nodes or edges.

A graph H is called a *solution base for* n *in* G iff ($\leftrightarrow$) the above conditions hold except condition (5) on leaf nodes and condition (6) on solution constraints.

©Stein/Lettmann/Hagen 1998–2021

Remarks:

- A solution graph is compact in the sense that it does not contain multiple instances of identical rest problems.
- The acyclicity condition on H can be omitted if search is restricted to acyclic AND-OR graphs G.
- If H is a solution graph for a node n in G and if n' is some node in H, then the subgraph H' of H starting at n' is a solution graph for n' in G. This subgraph is called the *solution graph in* H *induced by* n'.
- Each solution graph defines a decomposition hierarchy of the problem associated with n.
- The instantiation of a solution graph is based on local decisions. For instance, a solution graph for some AND node is given by combining solution graphs of its successors.
- A legitimate solution in a problem-reduction graph must be finite. This property is implicitly fulfilled since the problem-reduction graph itself is created by a search algorithm within a finite amount of time.
- A more strict definition of solution graphs could require that there is for each node in G at most one instantiation in H. Then H is a subgraph of G. This corresponds to a definition of cycle-free solution paths in an OR graphs. Instead, the tree requirement in the definition of solution trees is replaced with the condition of acyclicity. Then the reuse of a solution graph for a problem that occurs in multiple node instances is possible, but not mandatory. It depends on the use of an occurrence check in a search algorithm whether the solutions graphs are more or less tree-like.

 ©Stein/Lettmann/Hagen 1998–2021

Remarks:

- One of the main advantages of problem decomposition approaches is that solutions to subproblems can be reused if such subproblems occur multiple times. Therefore, the constructability requirement also conflicts with the reuse aspect.

  A constructional procedure for bottom-up solution graph synthesis could be the following:

  1. If $n \in \Gamma$, i.e., n is a node that represents a solved rest problem, then $H_n := \langle \{n\}, \emptyset \rangle$ is a solution graph for n in G.
  2. If n is an OR node with successor n' in G and $H_{n'} = \langle V', E' \rangle$ is a solution graph for n' in G, then $H_n := \langle V' \cup \{n\}, E' \cup \{(n, n')\} \rangle$ is a solution graph for n in G.
  3. If n is an AND node with successors $n'_1, \ldots, n'_k$ in G and $H_{n'_i} = \langle V'_i, E'_i \rangle$ is a solution graph for n'_i in $G, i = 1, \ldots, k$, then $H_n := \langle V'_1 \cup \ldots \cup V'_k \cup \{n\}, E'_1 \cup \ldots \cup E'_k \cup \{(n, n'_1), \ldots, (n, n'_k)\} \rangle$ is a solution graph for n in G.

  Q. Why is the above process problematic?

- Following the above inductive description, a recursive procedure can be implemented that constructs such graphs. By construction these graphs will be finite and contain start node s. Iteratively eliminating edges and nodes from that graph will lead to a solution graph (apart from additional solution constraints that might not be met).

- The fact whether a problem-reduction graph G has a solution graph H for the start node s can be checked by recursively applying the propagation rules of the [solved-labeling procedure].

- An example of additional solution constraints is a maximum edge cost restriction for solution graphs.

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Solution Graphs in AND-OR Graphs (continued)

Usage.

- Solution Graphs

  We are interested in finding a solution graph H for the start node s in G.

- Solution Bases

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

Note.

- Solution graphs and solution bases are subgraphs of the graph G.

©Stein/Lettmann/Hagen 1998–2021

Remarks:

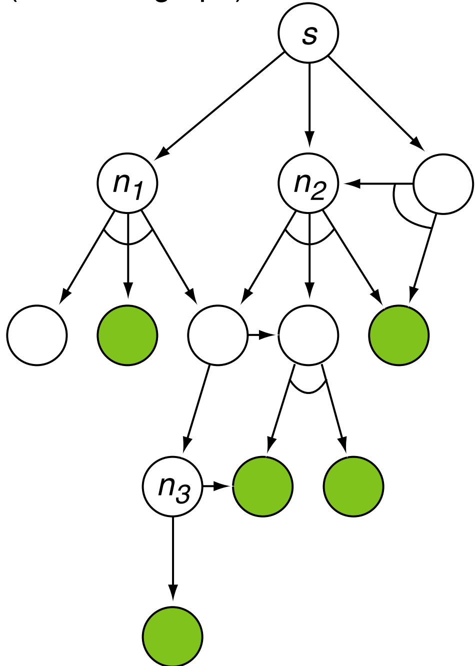
- Simply put, a subgraph of a search space graph is called solution base if it can be extended towards a solution (graph).
- Obviously, all solution graphs contained in G are also solution bases.
- If H is a solution base for a node n in G and if n' is some node in H, then the subgraph of H rooted at n', H', is a solution base for a node n' in G. This subgraph is sometimes called the *solution base in* H *induced by* n'.

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Illustration of Solution Graphs and Solution Bases

Problem-reduction graph:
(AND-OR graph)



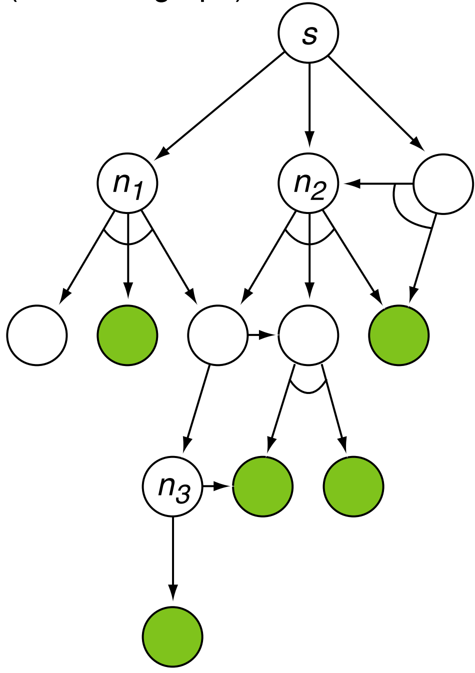
Solved rest problem

©Stein/Lettmann/Hagen 1998–2021

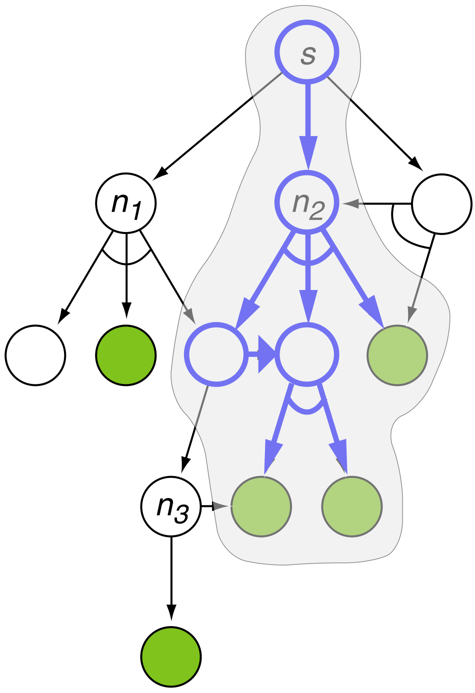
AND-OR Graph Search

Illustration of Solution Graphs and Solution Bases

Problem-reduction graph:
(AND-OR graph)



Solution graph for node s:



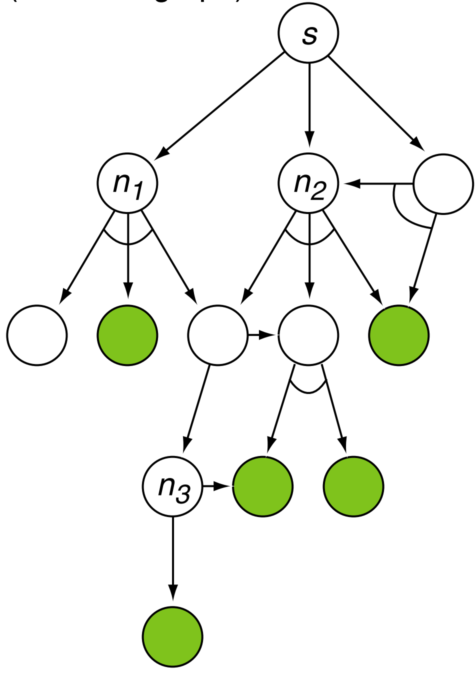
Solved rest problem

©Stein/Lettmann/Hagen 1998–2021

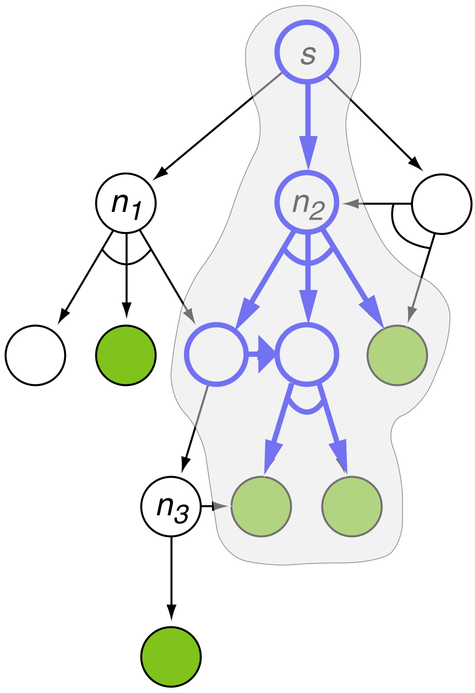
AND-OR Graph Search

Illustration of Solution Graphs and Solution Bases

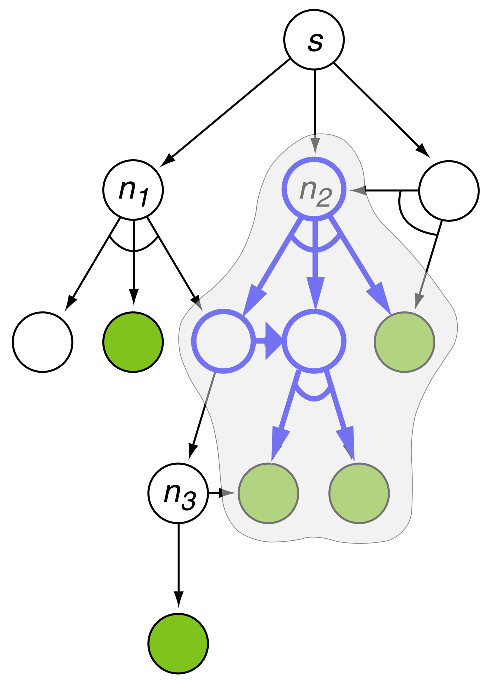
Problem-reduction graph:
(AND-OR graph)



Solution graph for node s:



Solution graph for node n_2:



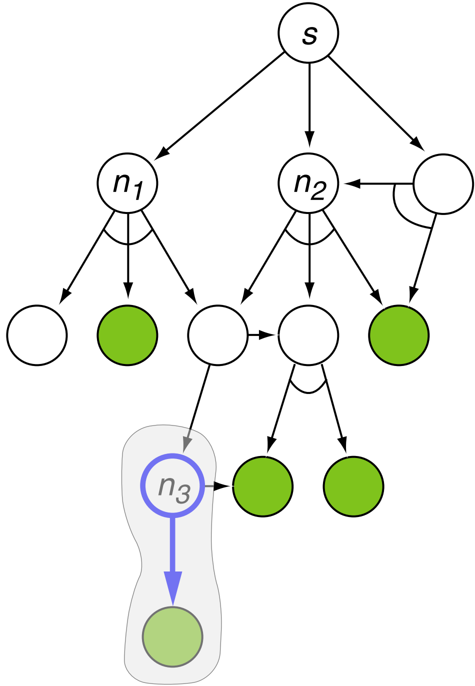
Solved rest problem

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Illustration of Solution Graphs and Solution Bases (continued)

Solution graph for n_3:

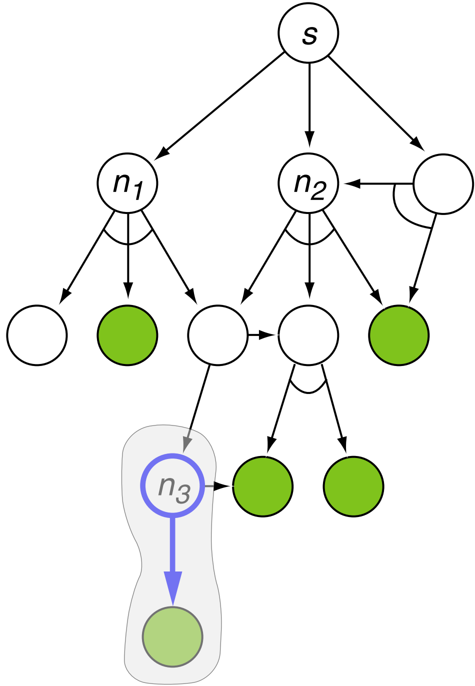


 ©Stein/Lettmann/Hagen 1998–2021

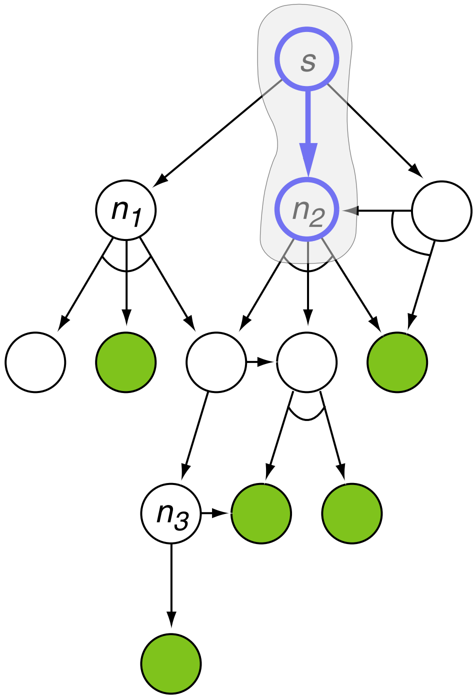
AND-OR Graph Search

Illustration of Solution Graphs and Solution Bases (continued)

Solution graph for n_3:



Solution base for s:

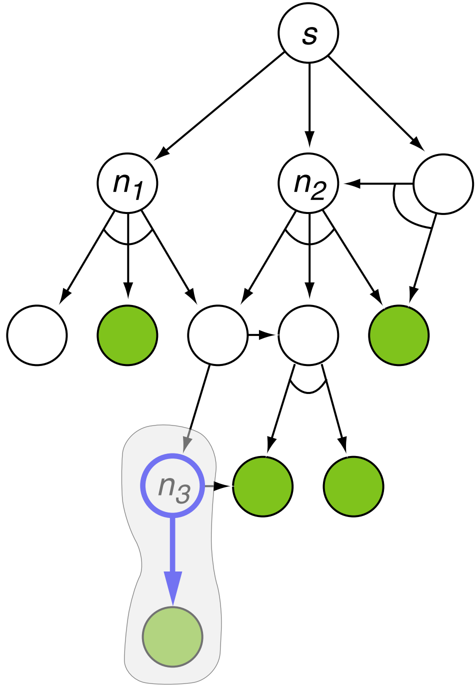


©Stein/Lettmann/Hagen 1998–2021

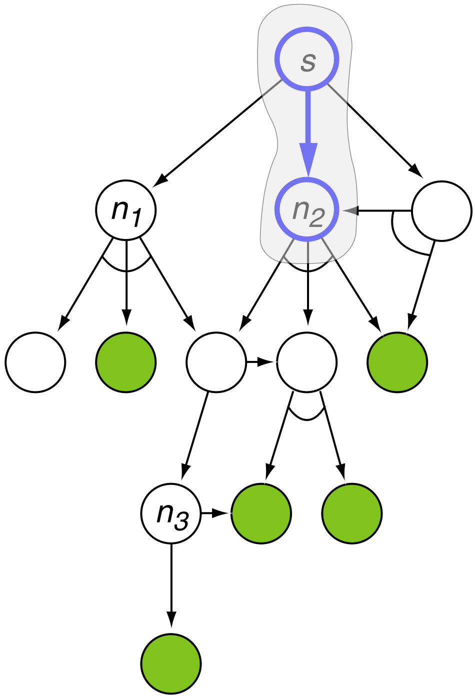
AND-OR Graph Search

Illustration of Solution Graphs and Solution Bases (continued)

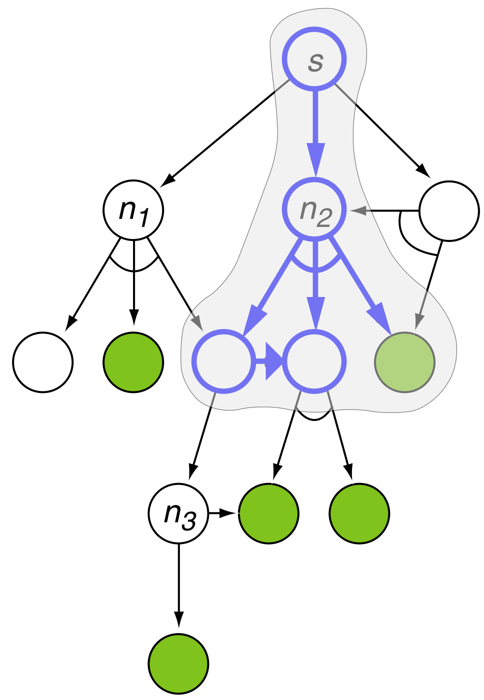
Solution graph for n_3:



Solution base for s:



Solution base for s:



Compare: Solution paths and solution bases in a state-space graphs.

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Generic Schema for AND-OR-Graph Search Algorithms

... from a solution-base-oriented perspective:

1. Initialize solution-base storage.
2. Loop.
   (a) Using some strategy select a solution base to be extended.
   (b) Using some strategy select an unexpanded node in this base.
   (c) According to the node type extend the solution base by successor nodes in any possible way and store the new candidates.
   (d) Determine whether a solution graph is found.

Usage:

- Search algorithms following this schema maintain a set of solution bases.
- Initially, only the start node s is available; node expansion is the basic step.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Algorithm: Generic_AND-OR

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

Output: A solution graph b or the symbol *Fail*.

© Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search [Generic_AND_OR_Tree] [Generic_OR]

```
Generic_AND-OR(s, successors, ⋆)

1.  b = solution_base(s);   // Initialize solution base b.
    IF ⋆(b) THEN RETURN(b);  // Check if b is solution graph.

2.  push(b, OPEN);     // Store b on OPEN waiting for extension.

3.  LOOP

4.    IF (OPEN = ∅) THEN RETURN(Fail);

5.    b = choose(OPEN);     // Choose a solution base b from OPEN.
      remove(b, OPEN);     // Delete b from OPEN.
      n = choose(b);     // Choose unexpanded tip node in b.

6.    IF (is_AND_node(n))
      THEN
        b' = add(b, n, successors(n));    // Expand n and extend b by AND edges.
        IF ⋆(b') THEN RETURN(b');    // Check if b' is solution graph.
        push(b', OPEN);    // Store b' on OPEN waiting for extension.
      ELSE
        FOREACH n' IN successors(n) DO    // Expand n.

          b' = add(b, n, {n'});    // Extend b by OR edge (n, n').
          IF ⋆(b') THEN RETURN(b');    // Check if b' is solution graph.
          push(b', OPEN);    // Store b' on OPEN waiting for extension.
        ENDDO
      ENDIF

7.  ENDLOOP
```


©Stein/Lettmann/Hagen 1998–2021

Remarks:

- Algorithm Generic_AND-OR takes a solution-base-oriented perspective.

  Generic_AND-OR maintains and manipulates solution bases (which are AND-OR graphs with root s).

- In order to keep the pseudo code short, we do not cover the case that a solution base chosen in 5. has no unexpanded tip node. Such a solution base can be discarded.

- Remaining problems which still are to be solved can be found only among the tip nodes. So, a solution base may contain multiple open problems.

- Function *add*$(b, n, .)$ returns a representation of the extended solution base: node instances in *successors*(n) will be unified with instances already contained in b, the edges to the direct successor have to be added in any case. For that purpose, b is copied. So each solution base is represented separately, although they often share subgraphs.

- To be precise, function *add*$(b, n, .)$ should include a check whether the resulting solution bases are still acyclic. Cyclic solution bases have to be discarded. (In order to avoid a test for cycles, it is often assumed that the search space graph G is acyclic.)

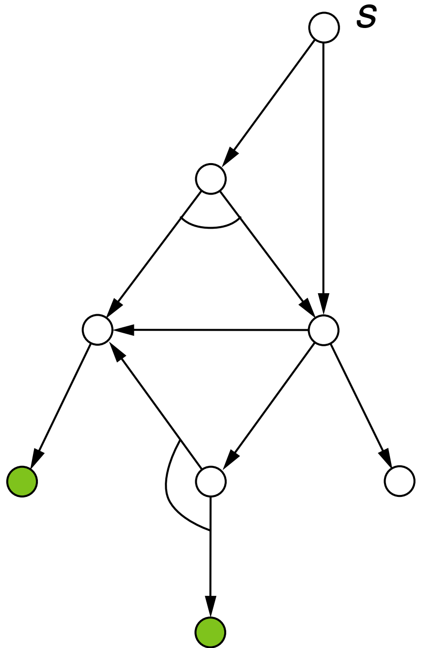
- Solution bases handled in Generic_AND-OR contain at most one instance per node of the underlying search space graph. Nevertheless we still assume that function *successors*(n) returns new instantiations (clones) for each successor node of an expanded node.

©Stein/Lettmann/Hagen 1998–2021

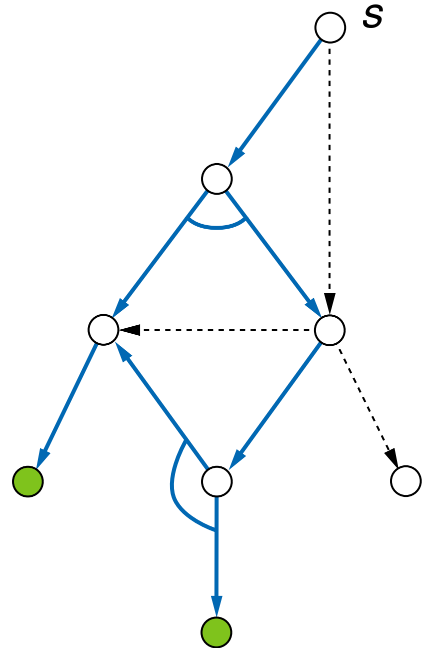
AND-OR Graph Search

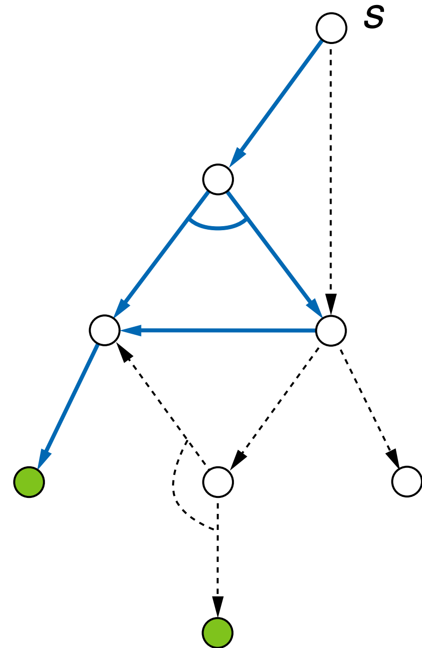
Efficient Storage of Solution Bases

AND-OR graph example:



Two possible solution graphs:





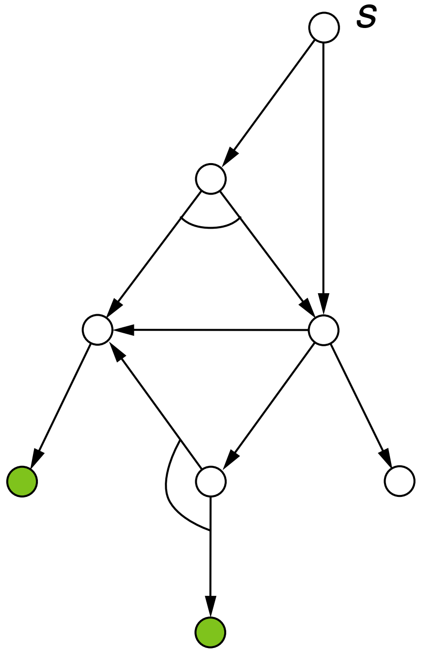
In algorithm `Generic_AND_OR_Tree` knowledge about the search space graph G and the solution-tree bases is stored in an AND-OR tree G_e and in a back-pointer structure.

©Stein/Lettmann/Hagen 1998–2021

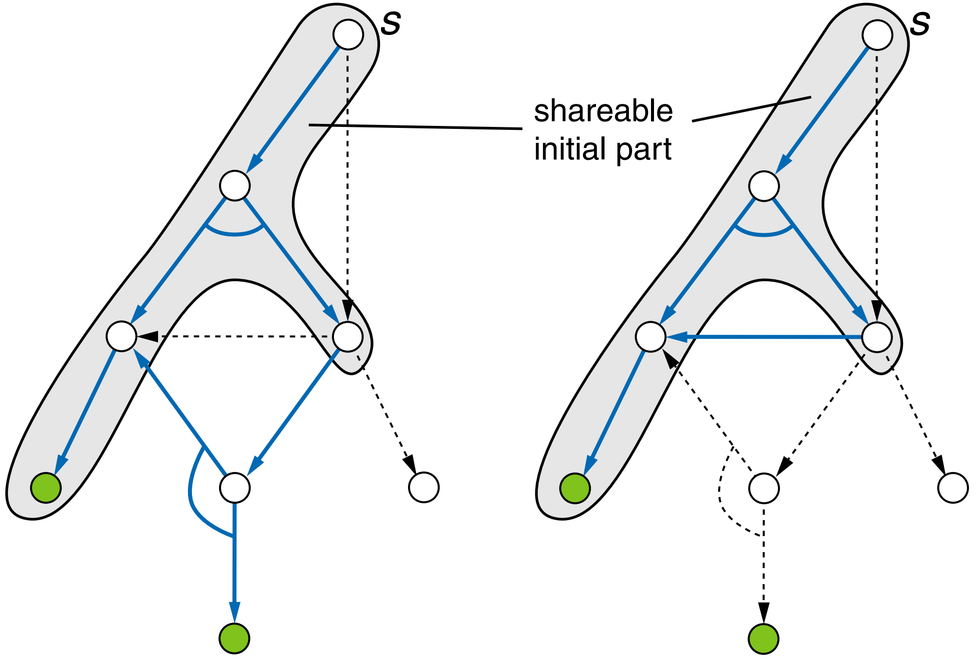
AND-OR Graph Search

Efficient Storage of Solution Bases (continued)

AND-OR graph example:



Two possible solution graphs:



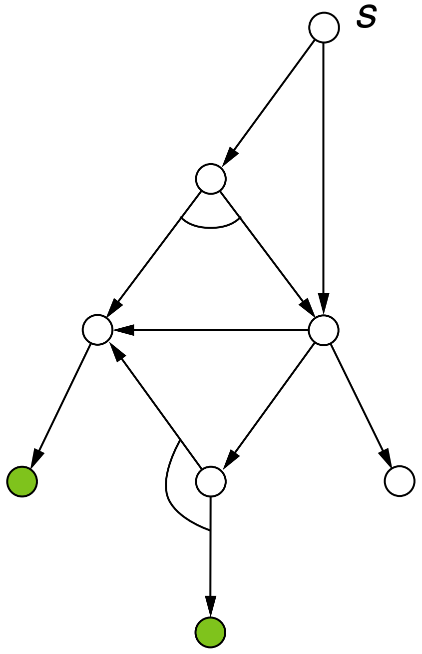
By reusing the approach from `Generic_AND_OR_Tree`, solution bases and solution graphs can also share initial parts.

©Stein/Lettmann/Hagen 1998–2021

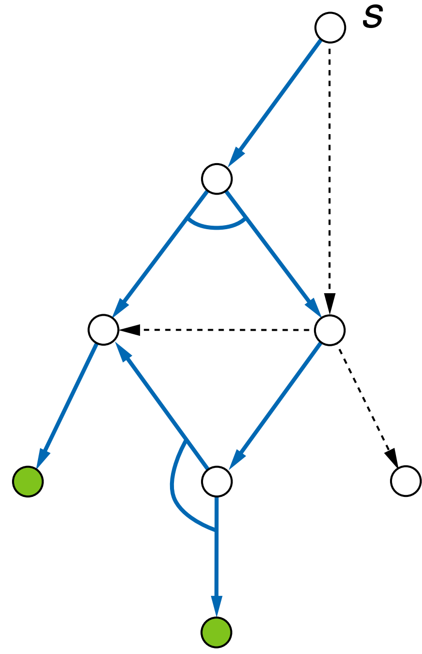
AND-OR Graph Search

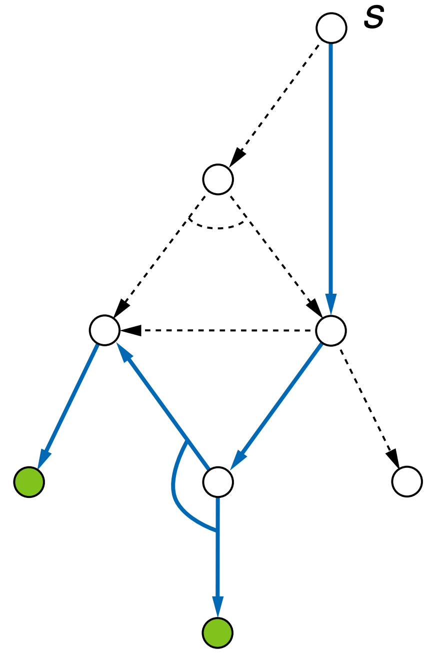
Efficient Storage of Solution Bases (continued)

AND-OR graph example:



Two possible solution graphs:





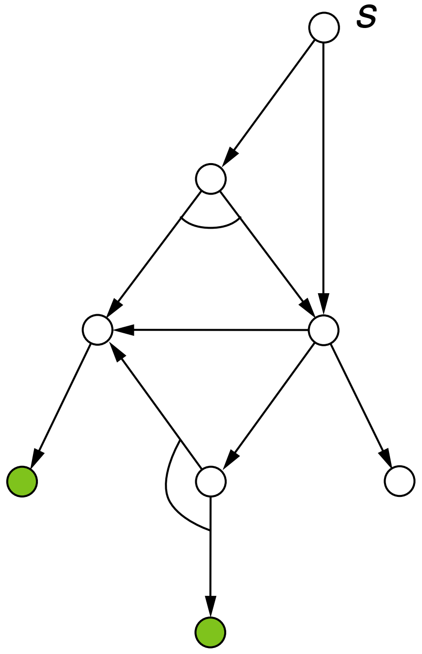
If a node has multiple occurrences in a solution tree, we want to have the same solution tree for each occurrence of that node.

 ©Stein/Lettmann/Hagen 1998–2021

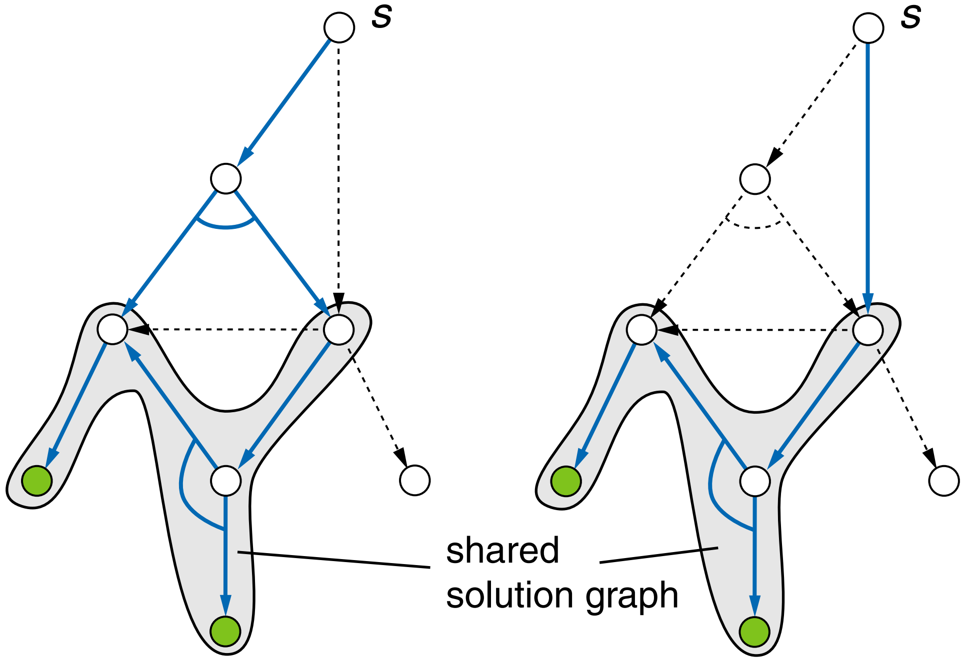
AND-OR Graph Search

Efficient Storage of Solution Bases (continued)

AND-OR graph example:



Two possible solution graphs:



In order to share a solution tree, multiple back-pointers are needed at its root node. Algorithms that use the more compact representation of solution graphs instead of solution trees have to use an occurrence check to identify previously generated instances of a node.

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Foundation of Basic AND-OR Graph Search

Basic Principle:

- The explicitly explored part G_e of an AND-OR graph G is stored during search.
- For each generated node of G there is a unique instantiation in G_e. Each time this node is found as a successor of some other node, the corresponding forward edge is added to G_e.
- Each time a node in G_e is found as a successor of some other node, a back-pointer to he parent node is added for that node. I.e., back-pointers connect each node in G_e with all of its expanded parents in G_e.
- Solution bases are identified in G_e on the fly by a DFS based search procedure so that tip nodes of solution bases are OPEN nodes.

Advantage:

- The explicitly stored graph G_e is a subgraph of G, i.e., G_e is the explored part of G.
- Extensions by node expansions become available to all solution bases that are currently contained in G_e. As a consequence, a solution graphs for a subproblems can be reused multiple times; a subproblem does not have to be solved multiple times.

Disadvantage:

- The identification of a solution base or a solution graph in G_e can require a complete DFS search in G_e with a propagation of all possible solution trees. For a compact representation some enfolding of trees will be required as well.

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Foundation of AND-OR Graph Search

General Structure.

- Initially, only the start node s is available; node expansion is the basic step.
- Unexpanded nodes are stored on OPEN, expanded nodes on CLOSED.
- For each expanded node, pointers to each of its successor nodes are stored.
- With each generated node, back-pointers to each of its parent nodes are stored.
- Solution bases maintained are the solution bases for s in G_e with tip nodes in OPEN.

Advantages.

- Algorithms must not handle multiple instances of nodes.
- Graph algorithms can be used as subroutines to process G_e.

Disadvantages:

- Solution bases / solution graphs have to be computed from G_e.
- Solution bases (e.g., cyclic solution bases), cannot be discarded separately.

Therefore, AND-OR graph search is now restricted to acyclic AND-OR graphs.

©Stein/Lettmann/Hagen 1998–2021

Remarks:

- Testing whether a graph is acyclic can be done in linear time (search for a topological sorting).

  However, G_e is the union of all solution bases under consideration. Extending one solution base by node expansion will affect multiple solution bases available in G_e. For some of them, this extension may lead to a cycle, for some others not. But there is no way to discard single solution bases from G_e. Cyclic solution bases will have to be ruled out again and again.

  So, there is no easy way to use multiple back-pointers and simultaneously to avoid cycles. This is again a justification for assuming that the search space graph G is acyclic.

 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

Algorithm: Basic_AND-OR

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.

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

Subroutines: *is_solved*(n). Predicate that is *True* if n is labeled "solved" or n is a goal node.
propagate_label(n). Function that propagates node label "solved" along back-pointers.

©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search [Basic_OR]

```
Basic_AND-OR(s, successors, is_solved)

 1.  insert(s, OPEN);  add_node(s, G_e);    // G_e is the explored part of G.
 2.  LOOP
 3.    IF (OPEN = ∅) THEN RETURN(Fail);
4.a    H = choose_solution_base(s, G_e);    // Maximal solution base for s in G_e.
4.b    n = choose(OPEN ∩ H);    // Choose OPEN non-goal tip node in H.
       remove(n, OPEN);  push(n, CLOSED);
 5.    FOREACH n' IN successors(n) DO

         IF (n' ∈ OPEN OR n' ∈ CLOSED)    // Instance of n' seen before?
         THEN   // Use old instance of n' instead.
           n' = retrieve(n', OPEN ∪ CLOSED);
         ELSE   // n' encodes an instance of a new state.
           add_node(n', G_e);  insert(n', OPEN);
         ENDIF
         add_edge((n, n'), G_e);  add_backpointer(n', n);

         IF is_solved(n')    // Is n' goal node or labeled solvable?
         THEN
           propagate_label(n');    // Propagate label along back-pointers.
           IF is_solved(s) THEN RETURN(compute_solution_graph(G_e));
         ENDIF

       ENDDO
 6.  ENDLOOP
```


©Stein/Lettmann/Hagen 1998–2021

Remarks:

- Algorithm Basic_AND-OR takes a graph-oriented perspective:
- A single OPEN node usually does not represent a solution base (if a non-terminal AND node is contained in the back-pointer path). Here, OPEN can be seen as the search frontier, the border line between the explored and the unexplored part of a solution base, but also of the underlying search space graph.
- Remaining problems which still are to be solved can be found only among the tip nodes of a solution base H. However, a solution base may contain multiple open problems.
- Expanding a node affects all solution bases that contain this node as a tip node.
- Function *propagate_label*(n) will propagate information about solved or unsolvable subproblems. Compared to [solved labeling using DFS], not only a path, but a directed acyclic graph is defined by the back-pointers starting in a node n.
- Even if the propagation of information "solved" guarantees that s is solved, the found solution graph does not have to be an extension of H. Therefore, function *compute_solution_graph*$(G_e))$ searches for a solution graph in G_e (e.g., using DFS).
- Again, no additional solution constraints are considered here.
- Because of the sharing of subgraphs, the explored subgraph G_e usually contains several solution graphs.

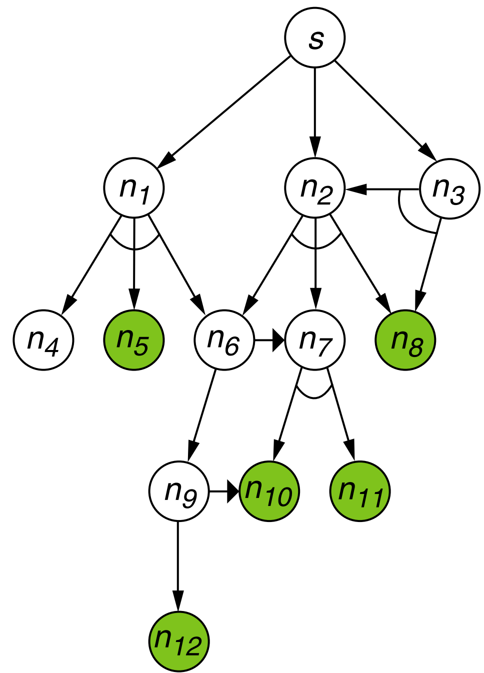
 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

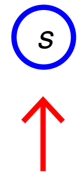
Illustration of Basic_AND-OR

Before the first run of the main loop.

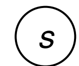
AND-OR graph G



Maintained subgraph G_e



Backpointer structure



Solved rest problem

Next solution base

OPEN node / CLOSED node

solved node / unsolvable node

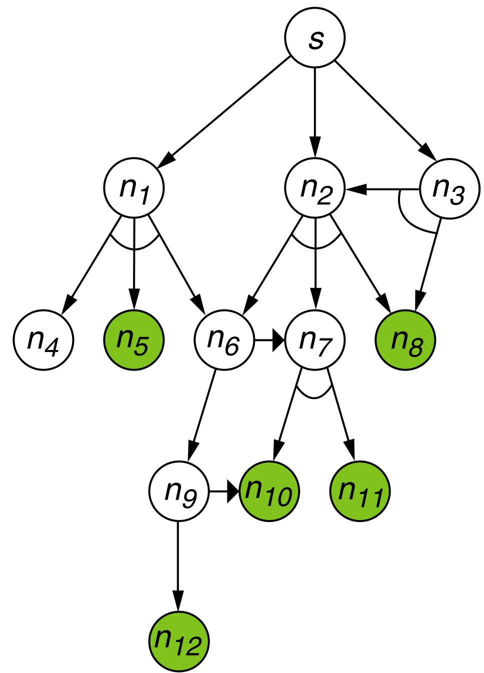
© Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

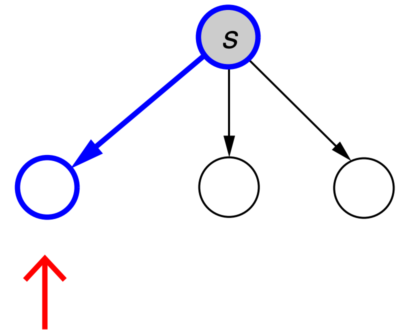
Illustration of Basic_AND-OR (continued)

Before the second run of the main loop (after expansion of s):

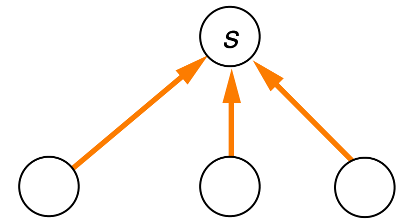
AND-OR graph G



Maintained subgraph G_e



Backpointer structure



Solved rest problem

Next solution base

OPEN node / CLOSED node

solved node / unsolvable node

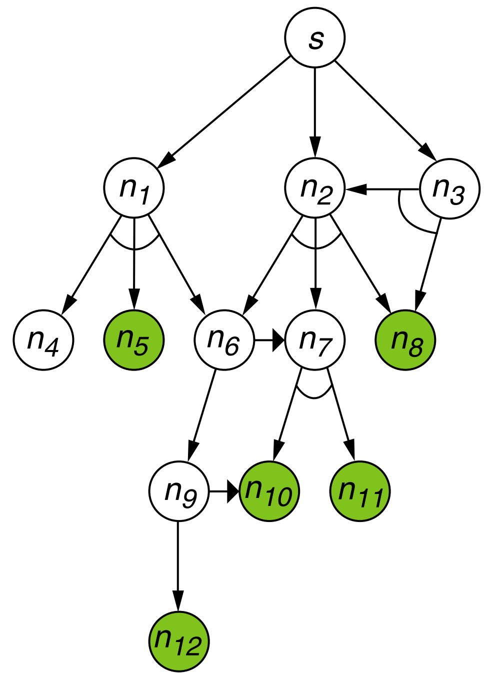
©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

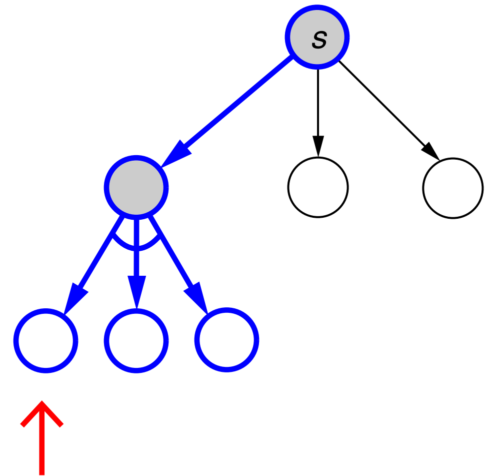
Illustration of Basic_AND-OR (continued)

Before the third run of the main loop (after expansion of n_1):

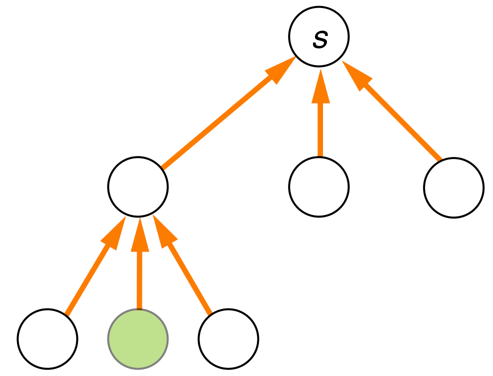
AND-OR graph G



Maintained subgraph G_e



Backpointer structure



Solved rest problem

Next solution base

OPEN node / CLOSED node

solved node / unsolvable node

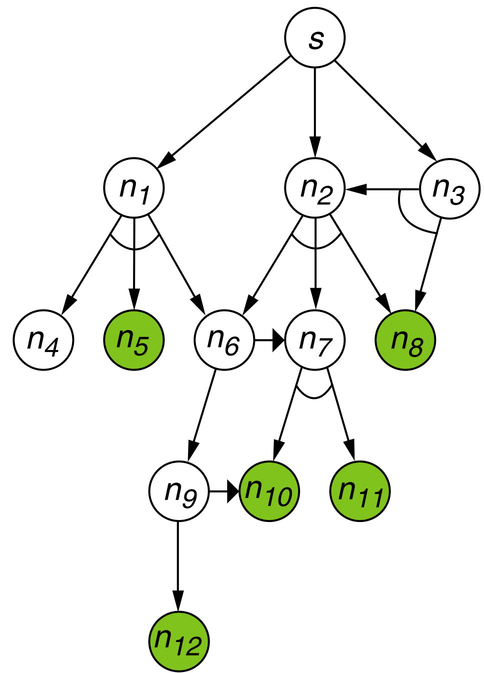
© Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

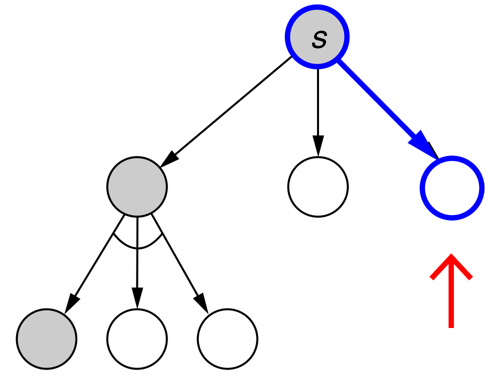
Illustration of Basic_AND-OR (continued)

Before the fourth run of the main loop (after expansion of n_4):

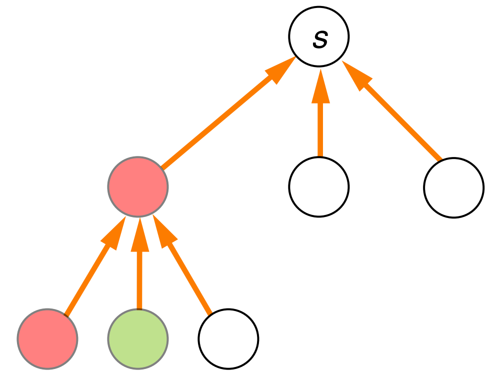
AND-OR graph G



Maintained subgraph G_e



Backpointer structure



Solved rest problem

Next solution base

OPEN node / CLOSED node

solved node / unsolvable node

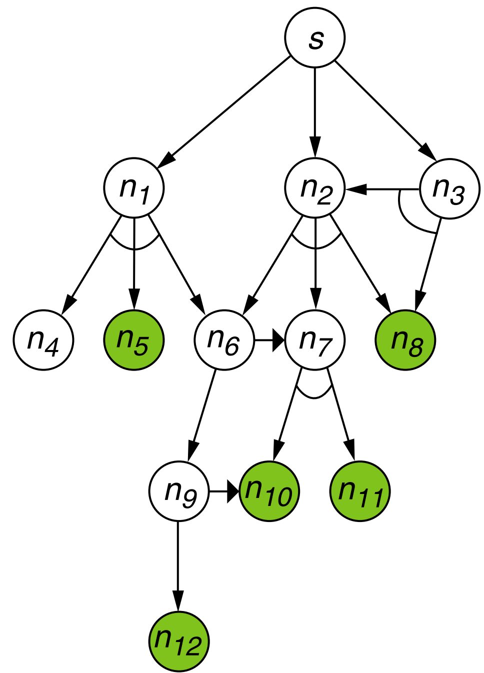
©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

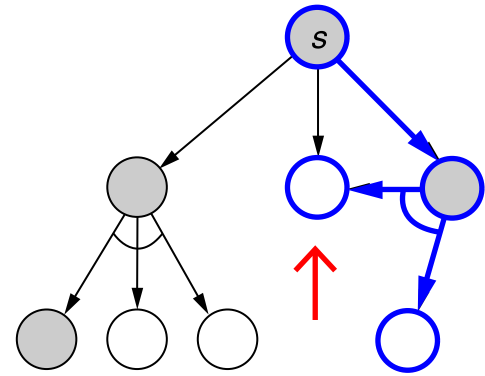
Illustration of Basic_AND-OR (continued)

Before the fifth run of the main loop (after expansion of n_3):

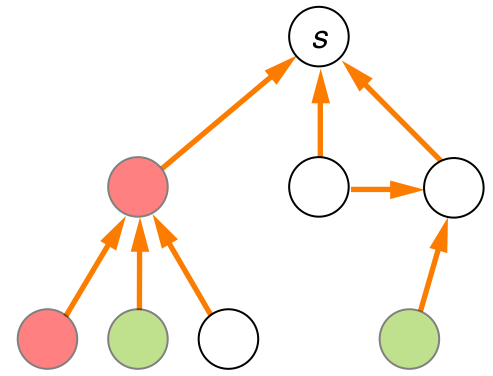
AND-OR graph G



Maintained subgraph G_e



Backpointer structure



Solved rest problem

Next solution base

OPEN node / CLOSED node

solved node / unsolvable node

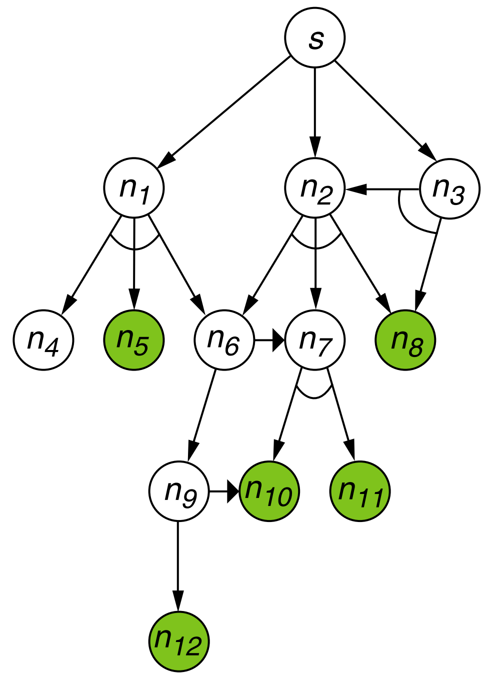
 ©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

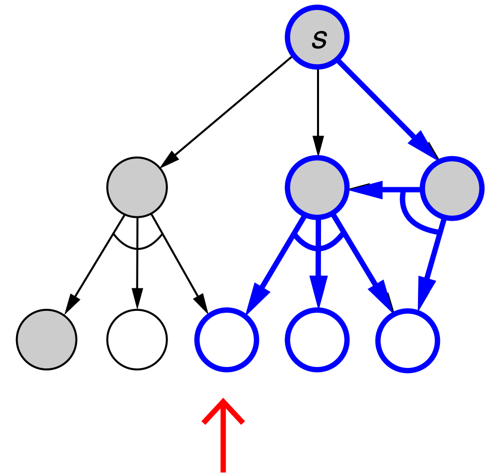
Illustration of Basic_AND-OR (continued)

Before the sixth run of the main loop (after expansion of n_2):

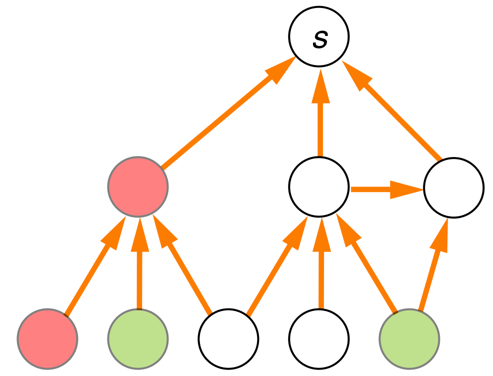
AND-OR graph G



Maintained subgraph G_e



Backpointer structure



Solved rest problem

Next solution base

OPEN node / CLOSED node

solved node / unsolvable node

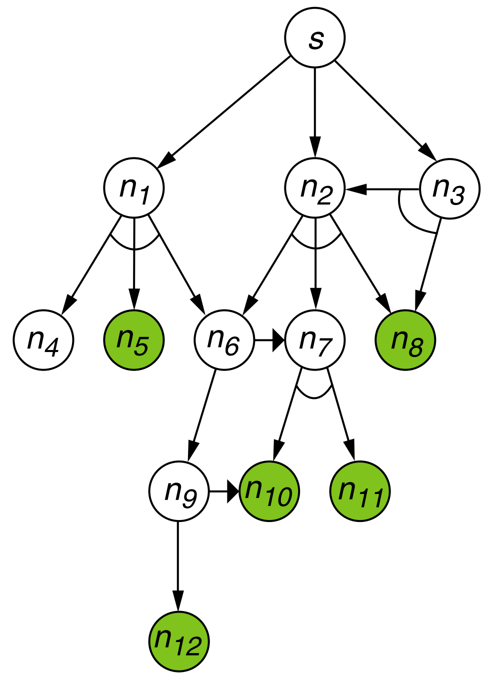
©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

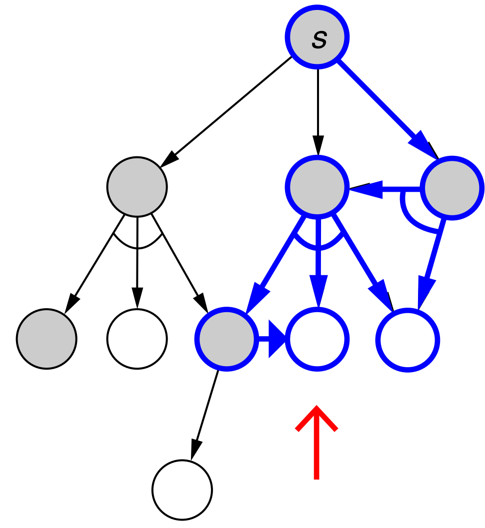
Illustration of Basic_AND-OR (continued)

Before the seventh run of the main loop (after expansion of n_6):

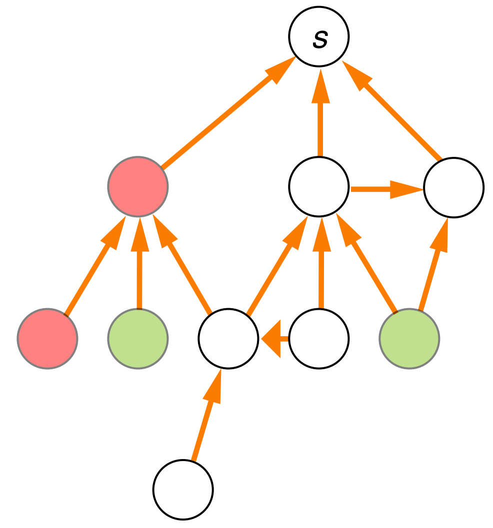
AND-OR graph G



Maintained subgraph G_e



Backpointer structure



Next solution base

OPEN node / CLOSED node

solved node / unsolvable node

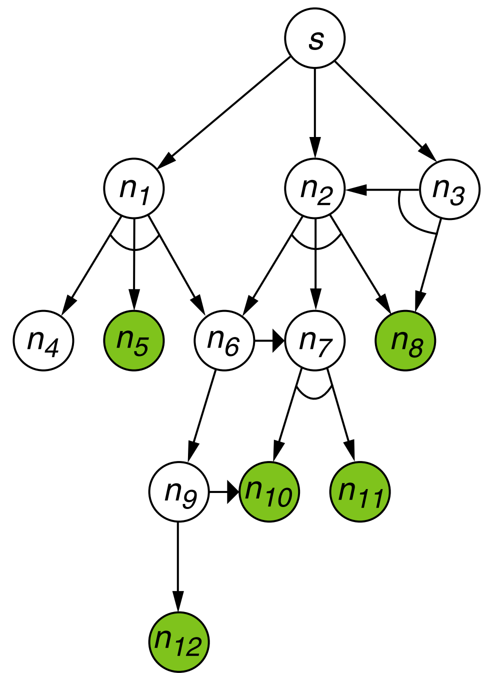
©Stein/Lettmann/Hagen 1998–2021

AND-OR Graph Search

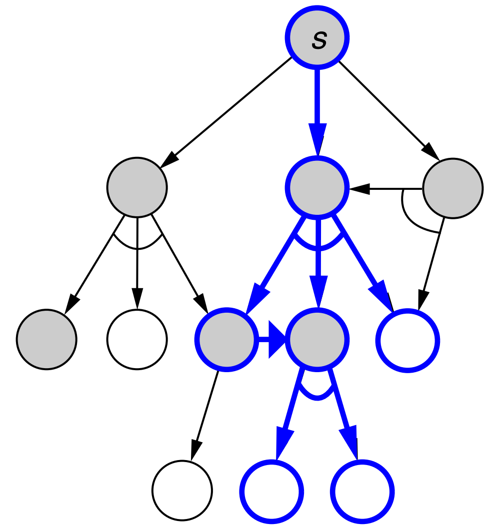
Illustration of Basic_AND-OR (continued)

During the seventh run of the main loop: expansion of n_7.

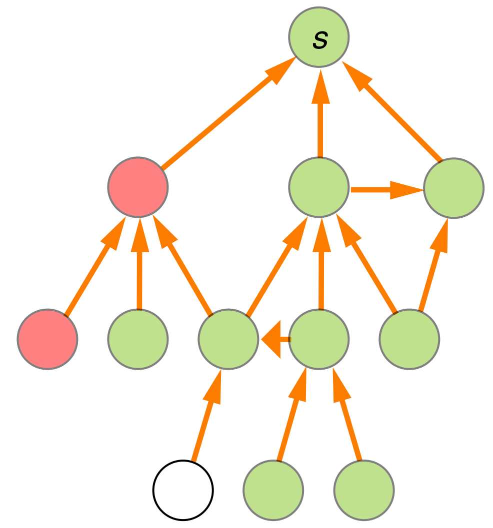
AND-OR graph G



Maintained subgraph G_e



Backpointer structure



Solved rest problem

Contained solution graph

OPEN node / CLOSED node

solved node / unsolvable node

s is labelled "solved", a solution graph was found.

©Stein/Lettmann/Hagen 1998–2021