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

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.
- Fast Median Computation the median of a list of numbers.
  Special case of the selection problem *"determine k-th element in list of numbers according to some sorting"*; algorithm by by Blum, Floyd, Pratt, Rivest, and Tarjan requires at most $6n$ comparisons.
- 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.
- Matrix Multiplication.
  Strassen algorithm requires $\mathcal{O}(n^{2.808})$ arithmetic operations compared to $\mathcal{O}(n^3)$ required by a naive implementation.

AND-OR Graph Basics

Building Blocks of Problem Reduction

- Decomposition into subproblems.

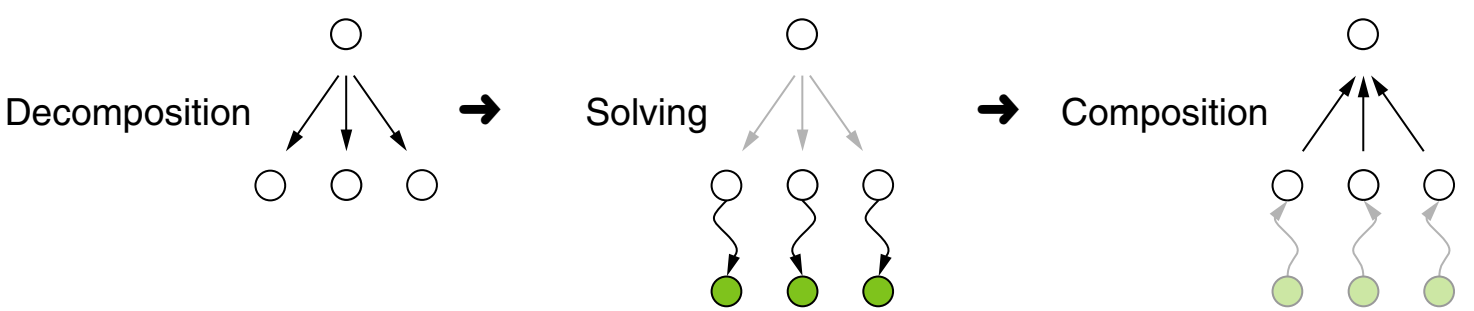
  A problem at hand is decomposed into a finite set of independently solvable subproblems.

- Solving all subproblems.

  All the subproblems are solved independently, i.e., without interactions between solution processes for different subproblems.

- Solution composition.

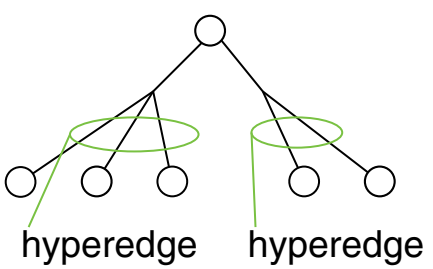
  Solutions to the subproblems are used to construct a solution of the problem at hand in a (relatively) simple way.

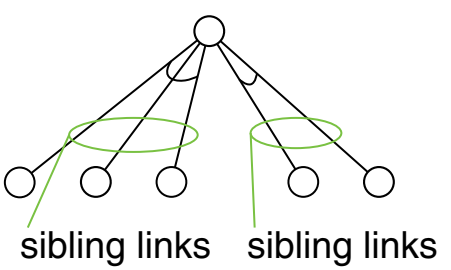


AND-OR Graph Basics

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?

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 straight forward way. The definition of hyperpaths, however, includes the property of being acyclic and minimal. [Nielsen/Andersen/Pretolani 2005]

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 10 ((Canonical) Problem-Reduction Graph, AND-OR Graph)

Let S be a state-space and $l : S \to \{\text{AND}, \text{OR}\}$ 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.

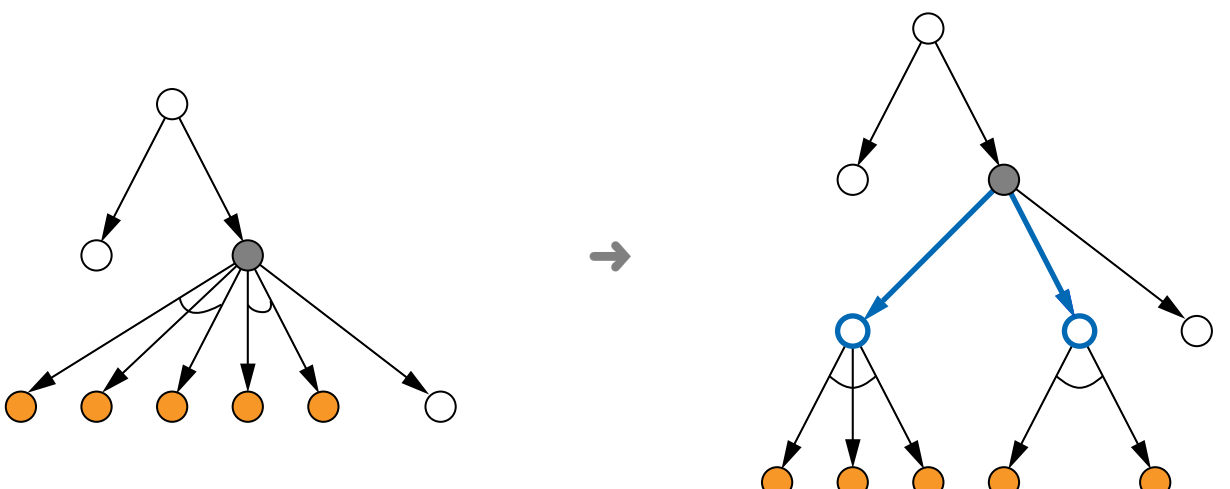
Usually, the labeling l will not be mentioned for AND-OR graphs.

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.

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, the root node may not be of OR type.
- 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.

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.

AND-OR Graph Basics

Solutions in AND-OR Graphs

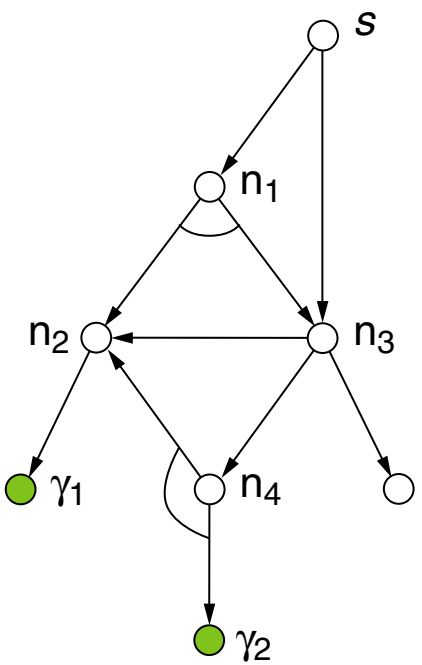
Desirable characteristics:

- 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.
  Solution should not contain superfluous parts.

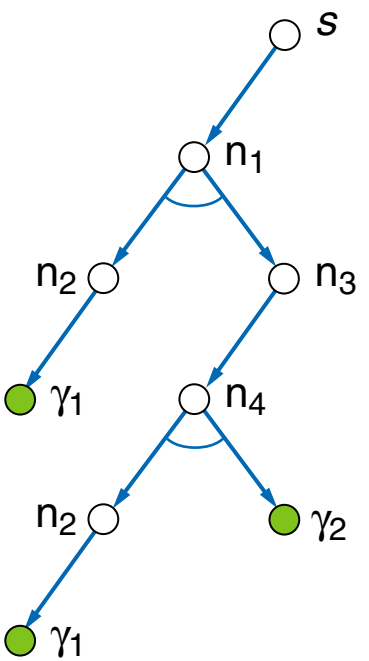
AND-OR Graph Basics

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

AND-OR Graph Basics

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

Definition 11 (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 tree H is called a *solution tree for* n *in* G iff ($\leftrightarrow$) the following conditions hold:

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

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 can 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 conflict?

AND-OR Graph Basics

Solution Trees in AND-OR Graphs (continued)

Definition 12 (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.

AND-OR Graph 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.
7. G has a function $\star(.)$ returning true if a solution base is a solution path.

Task:

- Determine in G a solution tree for s.

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.

AND-OR Graph 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.

AND-OR Graph Basics

Algorithm: Generic_AND-OR_Tree_Search

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*.

AND-OR Graph Basics [Generic_OR_Search]

Generic_AND-OR_Tree_Search(s, *successors*, $\star$)

```
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_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 tree.
        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 tree.
          push(b', OPEN);   // Store b' on OPEN waiting for extension.
        ENDDO
      ENDIF
7.  ENDLOOP
```

Remarks:

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

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

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

- 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*$(b, n, .)$ 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.

AND-OR Graph Basics

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.
- By sharing of initial parts, the maintained solution bases form a traversal tree.
- A solution bases is uniquely identified by the corresponding tip node in the traversal tree.

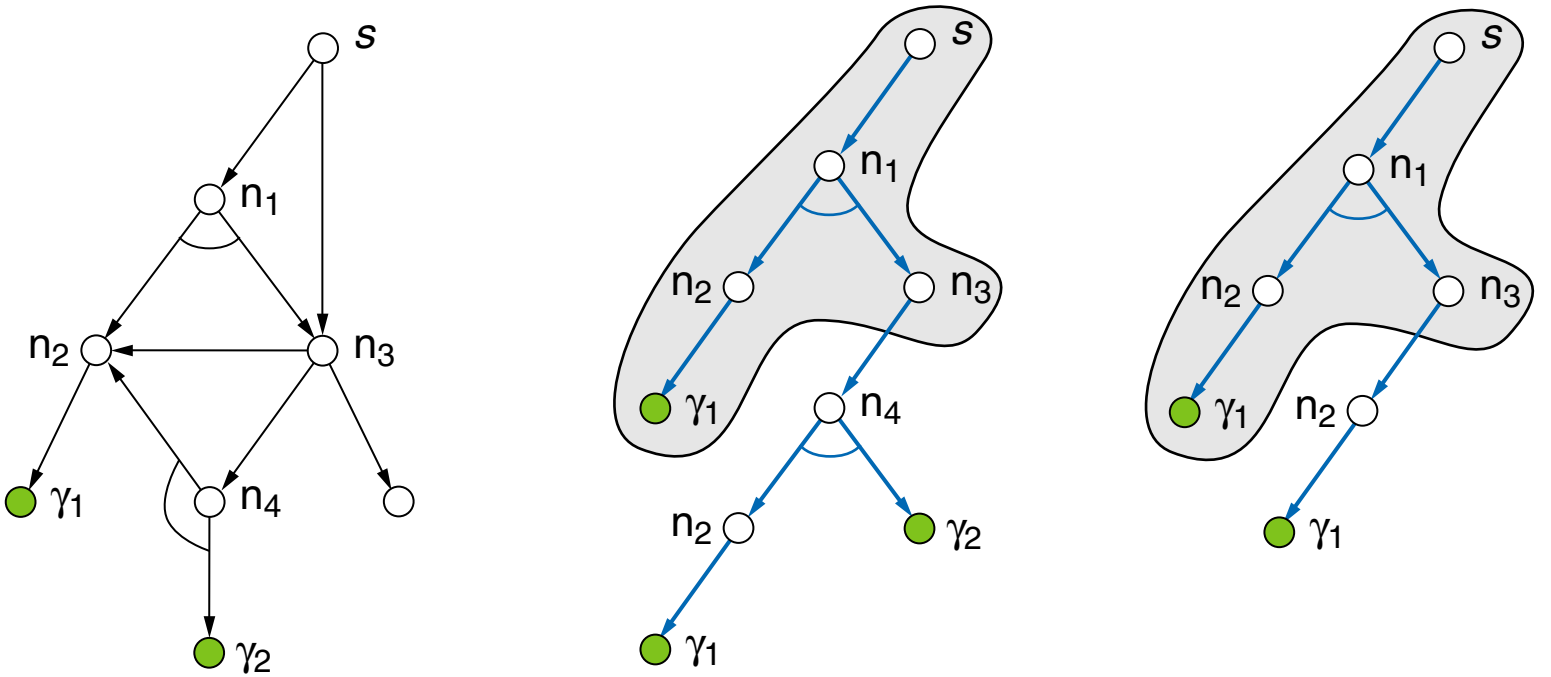
Idea: Reusing the backpointer concept of `Basic_OR_Search`
allows sharing of upper parts of solution-tree bases as well.

➜ For the solution tree bases maintained by `Generic_AND-OR_Tree_Search`, the resulting structure is a tree that conforms to traversal trees in OR-graph search.

Q. Can we apply OR-graph search to AND-OR graphs to build such trees?
Q. Can we use OR-graph search to detect a solution-tree that is contained?
Q. Can we use OR-graph search to detect, whether a solution tree is contained?

AND-OR Graph Basics

Example: Sharing initial subtree in solution trees for AND-OR graphs



Q. Which node instances are unified, which are kept separate when sharing initial subtrees?

Q. What do the resulting trees look like for the maintained solution-tree bases in line 3 of algorithm `Generic_AND-OR_Tree_Search`?

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).

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: Allows the currently known subgraph of the search space graph for the instantiation of a solution tree?
- To do: Propagate knowledge about solved or unsolved problems upwards from terminal nodes to the root node.
- Nodes whose labels have been propagated can be discarded.

Approach: Adapt [DFS] (or [BFS]) for the labeling.

- Labels are propagated along backpointers.

Depth-First Search of AND-OR Graphs

Solved Labeling

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

Definition 13 (Solved-Labeling Procedure)

Let G be an acyclic AND-OR graph. A node in G is labeled as “solved”, 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 nonterminal OR node is labeled as “solved” as soon as one of its successor nodes is labeled as “solved”; it is labeled as “unsolvable” as soon as all of its successor nodes are labeled as “unsolvable”.
3. A nonterminal AND node is labeled as “solved” as soon as all of its successor nodes are labeled as “solved”; it is labeled as “unsolvable” as soon as one of its successor nodes is labeled as “unsolvable”.

Note: No additional solution constraints are considered here.

Depth-First Search of AND-OR Graphs

Adaption of 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).

Replace termination condition.

- Instead of terminating with failure when OPEN is empty, terminate as soon as s is labeled:

  IF *is_labeled*(s) RETURN(*label*(s))

  (If OPEN is empty, start node s will be labeled “unsolvable”.)

Use a depth bound.

Depth-First Search of AND-OR Graphs

Adaption of DFS (continued)

Perform labeling during node expansion (**FOREACH** loop):

- Instead of terminating with success when finding a goal node, this case is handled in the same way as a dead end case, i.e., the node is labeled and the label is propagated along the backpointer path using the function *propagate_label*(n):

```
FOREACH n' IN successors(n) DO    // Expand n.
  add_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');
    cleanup_closed();
  ELSE
    push(n', OPEN);
  ENDIF
ENDDO
```

Depth-First Search of AND-OR Graphs

Adaption of DFS (continued)

Propagation of labels:

- Function *propagate_label*(n) will follow the backpointers. For the start node s the backpointer parent node is assumed to be `null`.

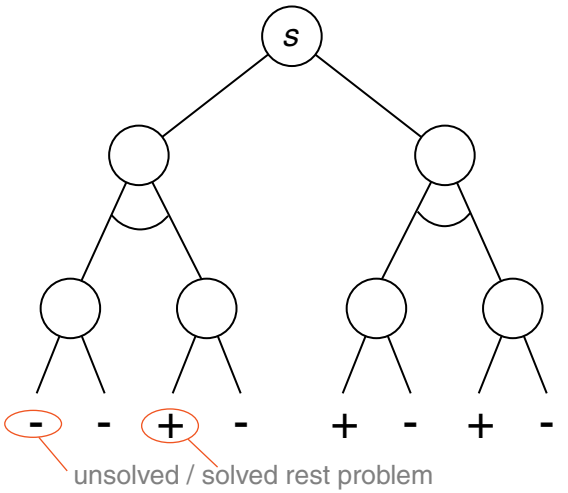
```
propagate_label(n)
  l = label(n); p = backpointer_parent(n);
  WHILE p ≠ null AND is_unlabeled(p) DO    // Process parent node p.
    IF is_OR_node(p) AND l = "solved" THEN set_label(p,"solved");
    IF is_OR_node(p) AND l = "unsolvable"
      THEN
      unlabeled_succ(p) = unlabeled_succ(p) − 1;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

Remarks:

- We assume that $\perp(n)$ returns true if n has no successors. Then, no node without successor nodes will be expanded and this case does not need to be covered.
- Since DFS is performed, only a path, i.e., a sequence of nodes, is processed in function *propagate_label*(n). However, the label of each node processed by DFS will be set at most once. Therefore, the overall effort in labeling is linear in the number of nodes (node instantiations) processed by DFS.

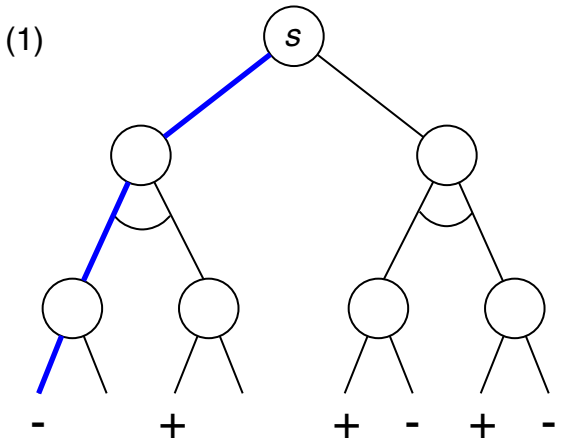
Depth-First Search of AND-OR Graphs

Example: Solved Labeling Using DFS



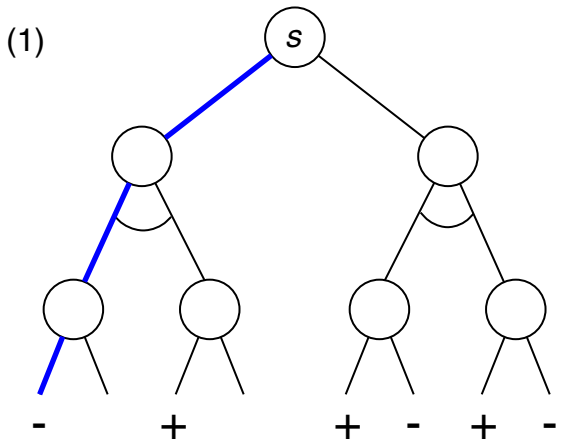
Depth-First Search of AND-OR Graphs

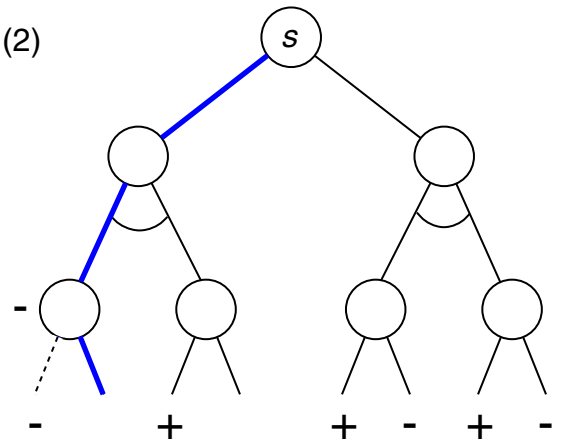
Example: Solved Labeling Using DFS (continued)



Depth-First Search of AND-OR Graphs

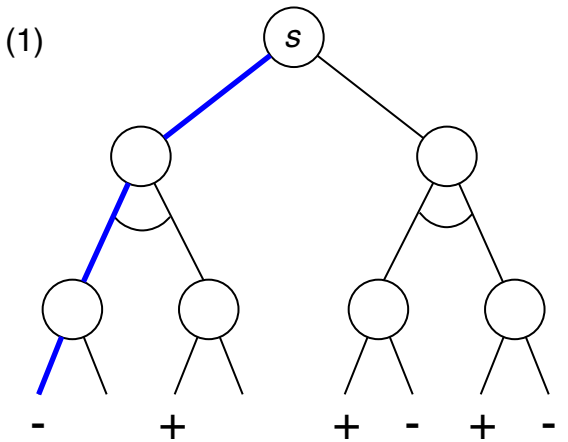
Example: Solved Labeling Using DFS (continued)

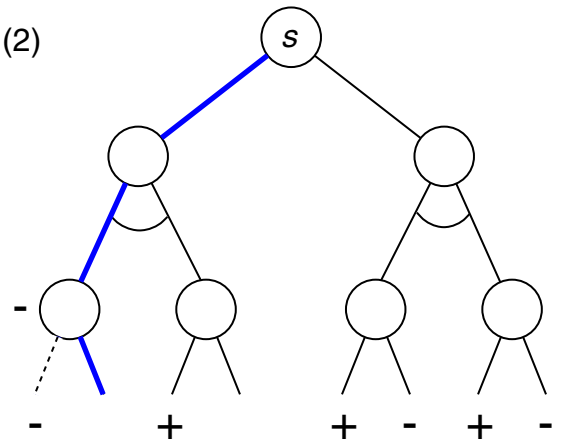


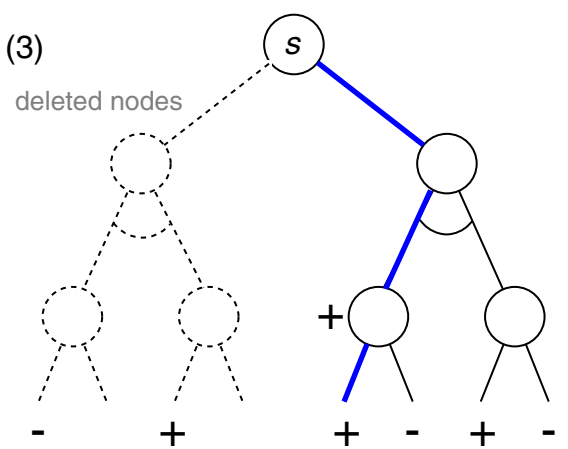


Depth-First Search of AND-OR Graphs

Example: Solved Labeling Using DFS (continued)

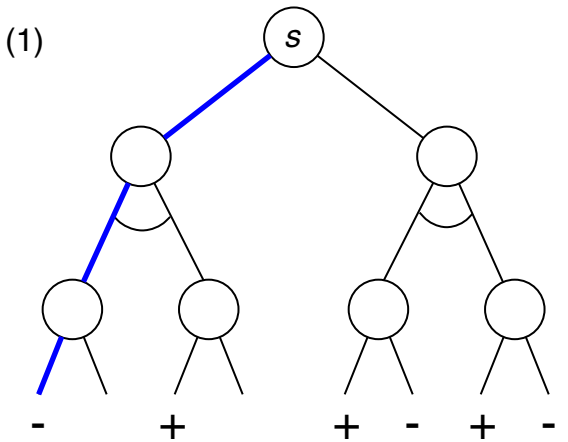


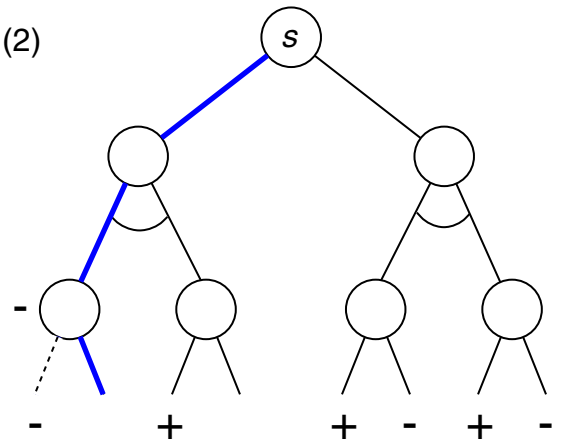


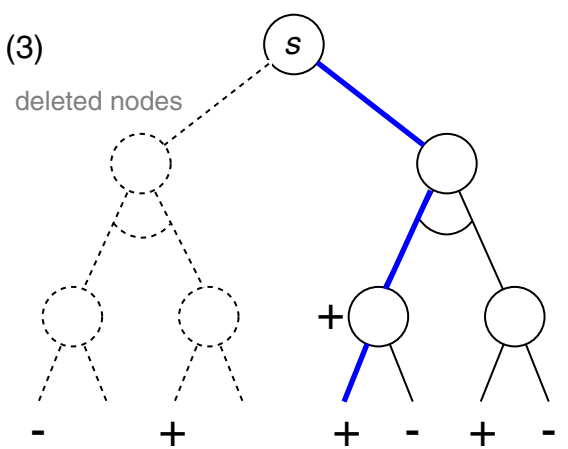


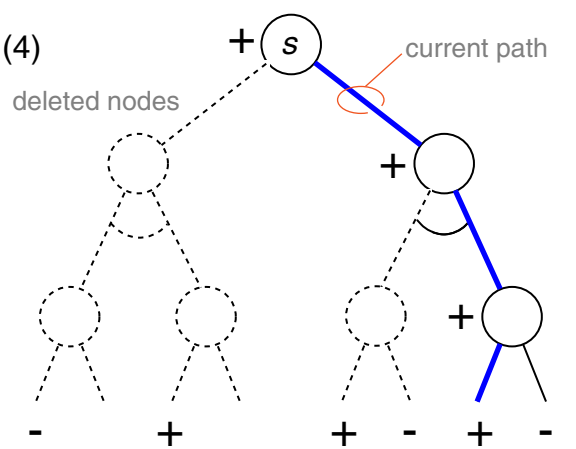
Depth-First Search of AND-OR Graphs

Example: Solved Labeling Using DFS (continued)









Remarks:

- Information “solved” resp. “unsolvable” for a node is propagated to its parent node following the backpointer. Once the final label is determined, it can propagated upwards and search of sibling nodes can be left undone (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.

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 “solvable”.

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

- Scope of the analysis by DFS search:

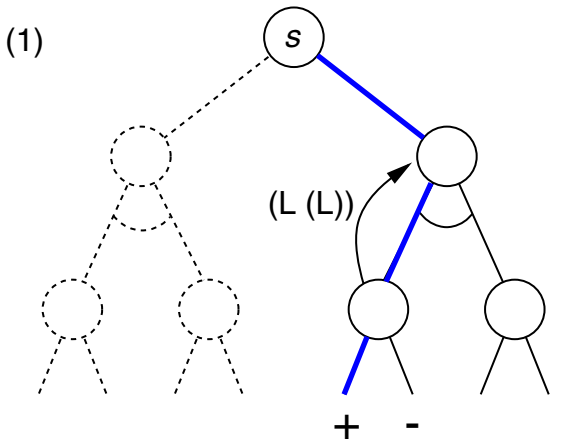
  Solved-labeling procedure:
  The label of the start node s indicates the existence of solution trees.
  ➜ OR-nodes are not fully analyzed once “solvable” is known.

  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.

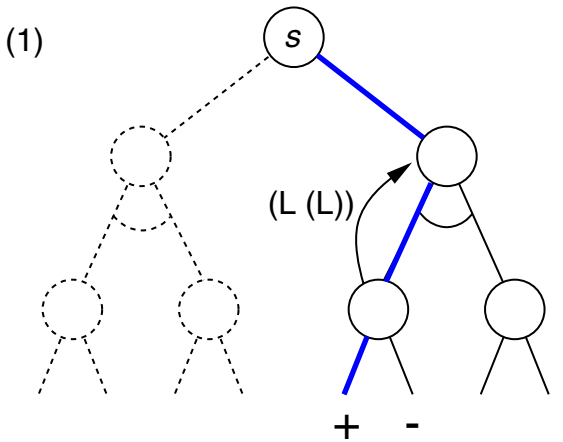
Depth-First Search of AND-OR Graphs

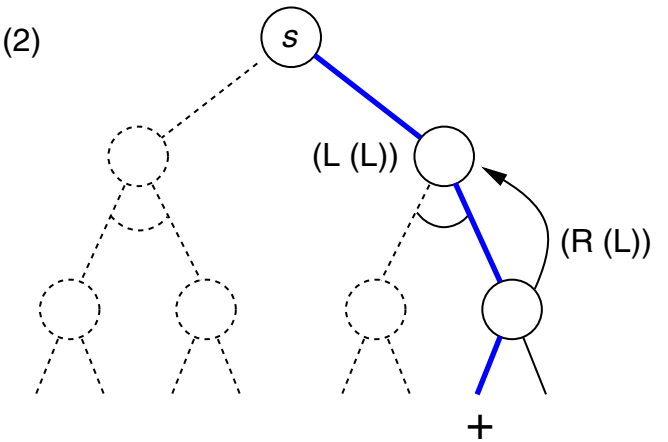
Example: *Solution Labeling* Using DFS



Depth-First Search of AND-OR Graphs

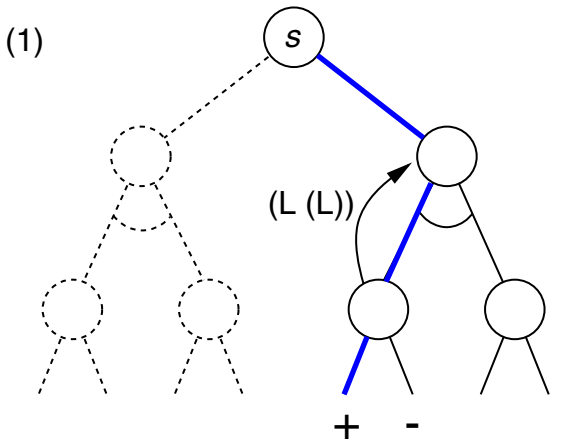
Example: *Solution Labeling* Using DFS (continued)

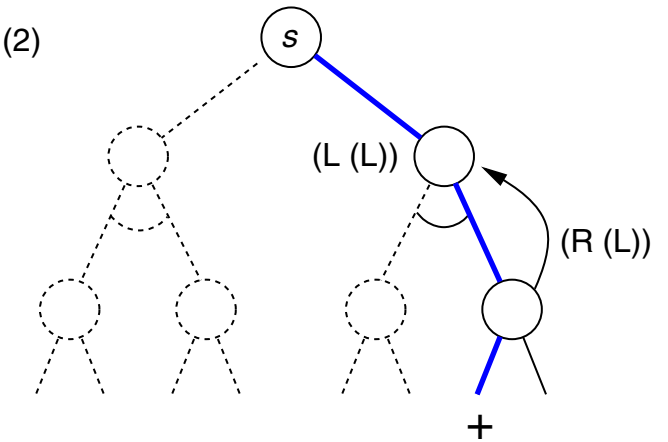


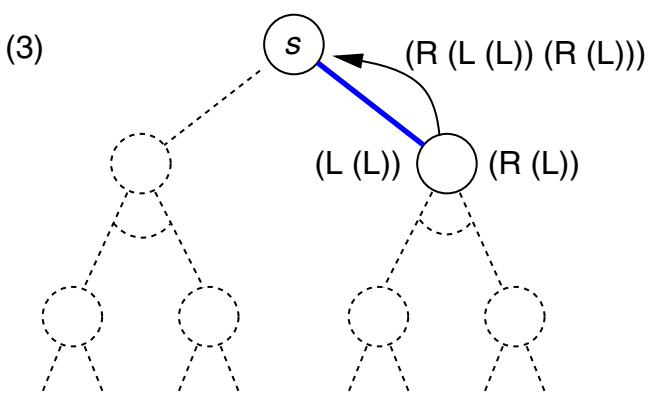


Depth-First Search of AND-OR Graphs

Example: *Solution Labeling* Using DFS (continued)

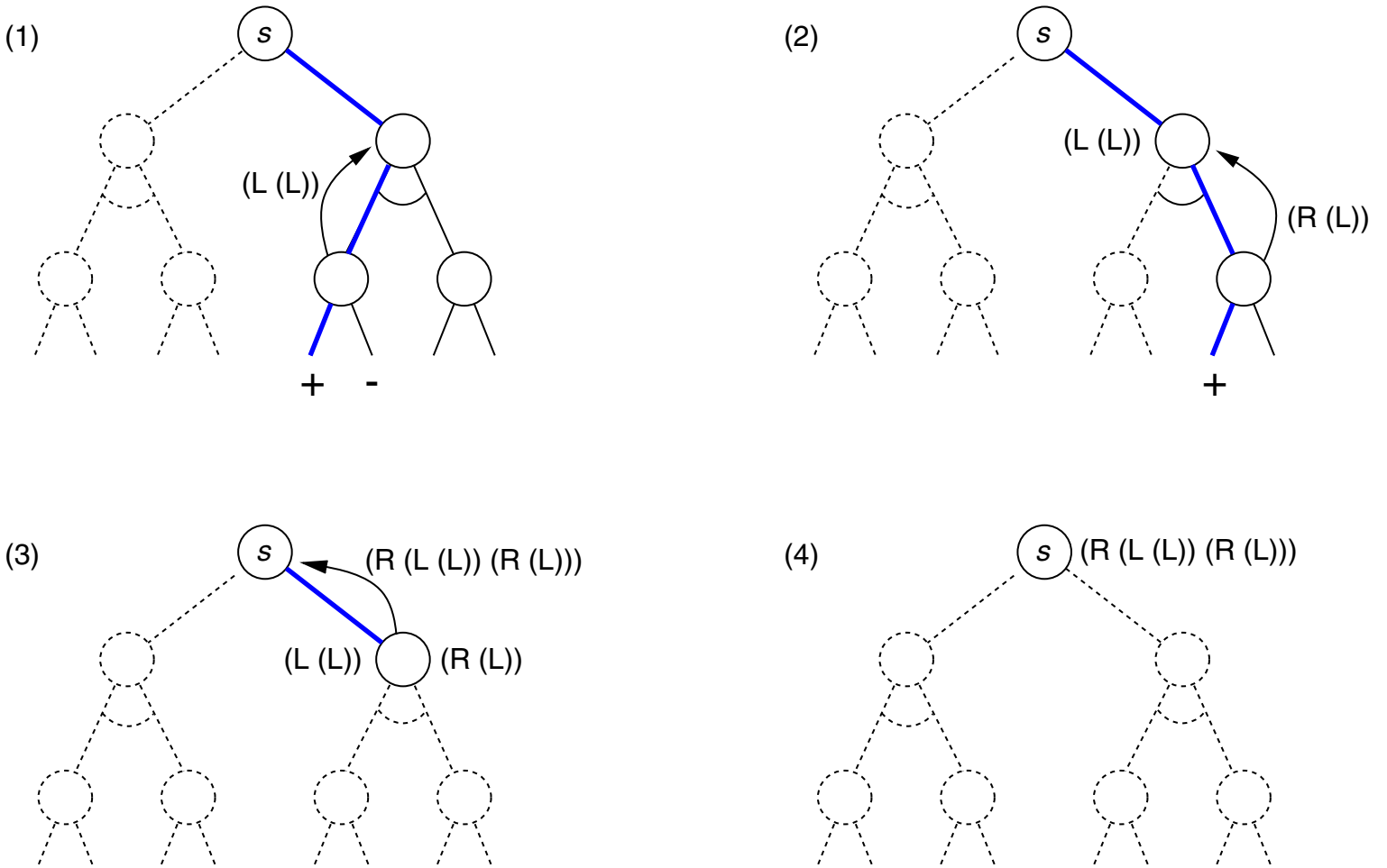






Depth-First Search of AND-OR Graphs

Example: *Solution Labeling* Using DFS (continued)



Remarks:

- If in 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, DFS could also propagate a list of all possible solution trees. In inner OR node 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.

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.

Depth-First Search of AND-OR Graphs

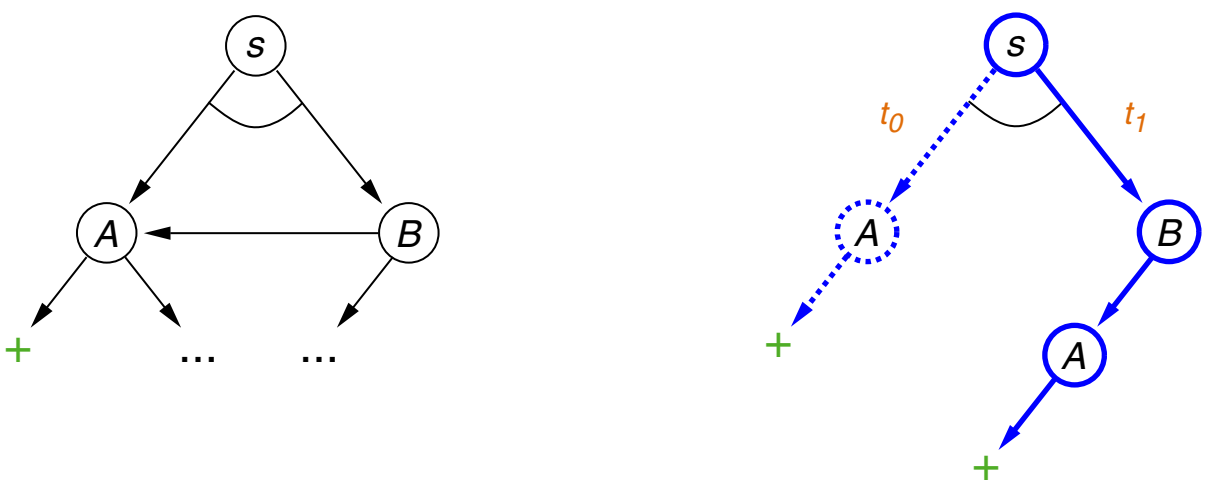
Redundant Problem Solving

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):



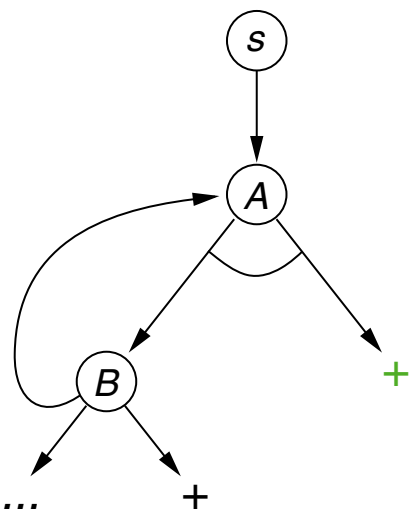
Remarks:

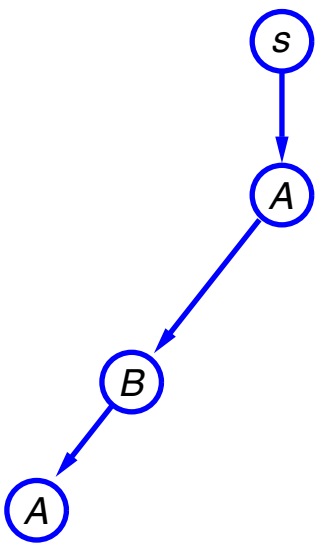
- With a complete occurrence check, i.e., an occurrence check that considers all explored nodes so far, recurring subproblems can be identified.
- A complete occurrence check entails a memory consumption similar to BFS, which renders such a check very unattractive—if not impossible. In most cases the 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 backpointer path [Pearl 84]. Recall that DFS stores the partial solution path on the CLOSED list (with exeption 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.

Depth-First Search of AND-OR Graphs

Dealing with Cycles (1)

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

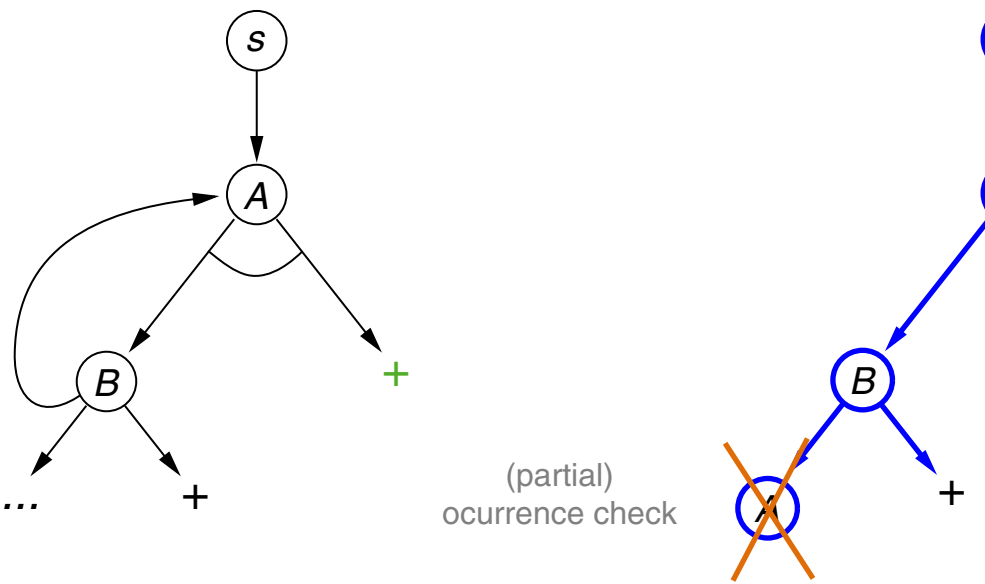




Depth-First Search of AND-OR Graphs

Dealing with Cycles (1)

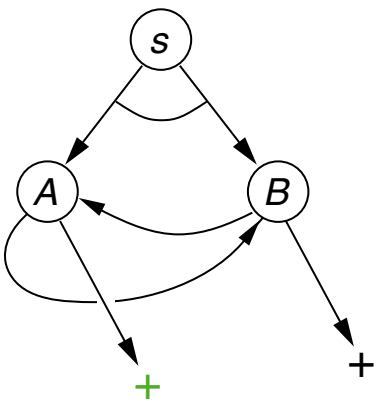
AND-OR graph (left) and its exploration (= unfolding) with DFS (right):



Depth-First Search of AND-OR Graphs

Dealing with Cycles (2)

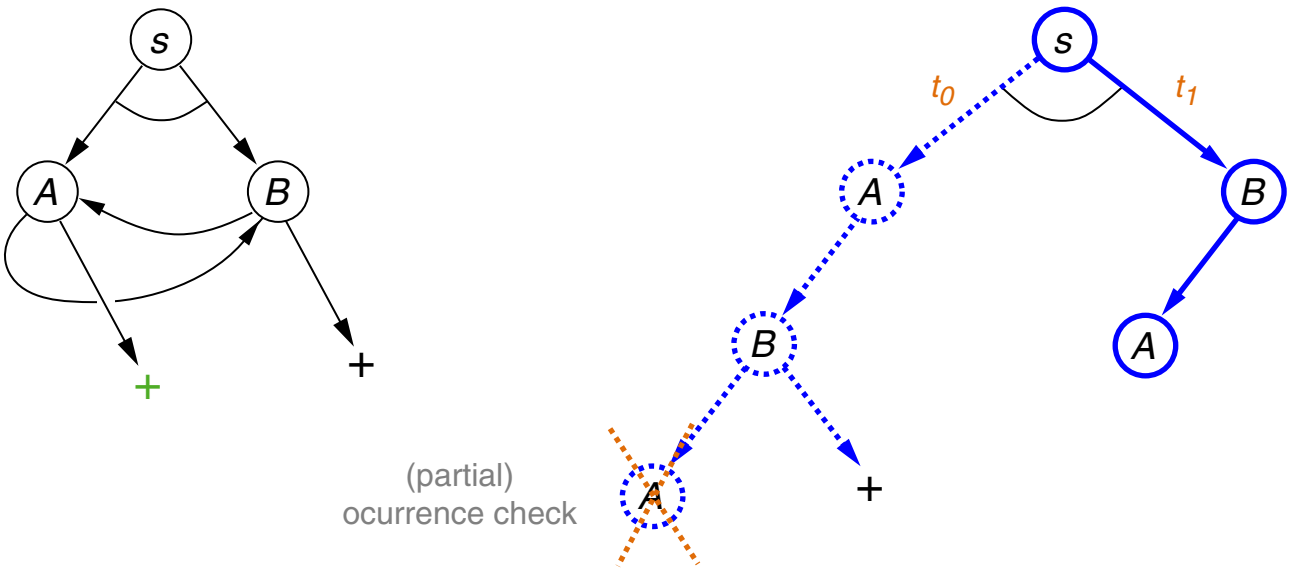
AND-OR graph (left) and its exploration (= unfolding) with DFS (right):



Depth-First Search of AND-OR Graphs

Dealing with Cycles (2)

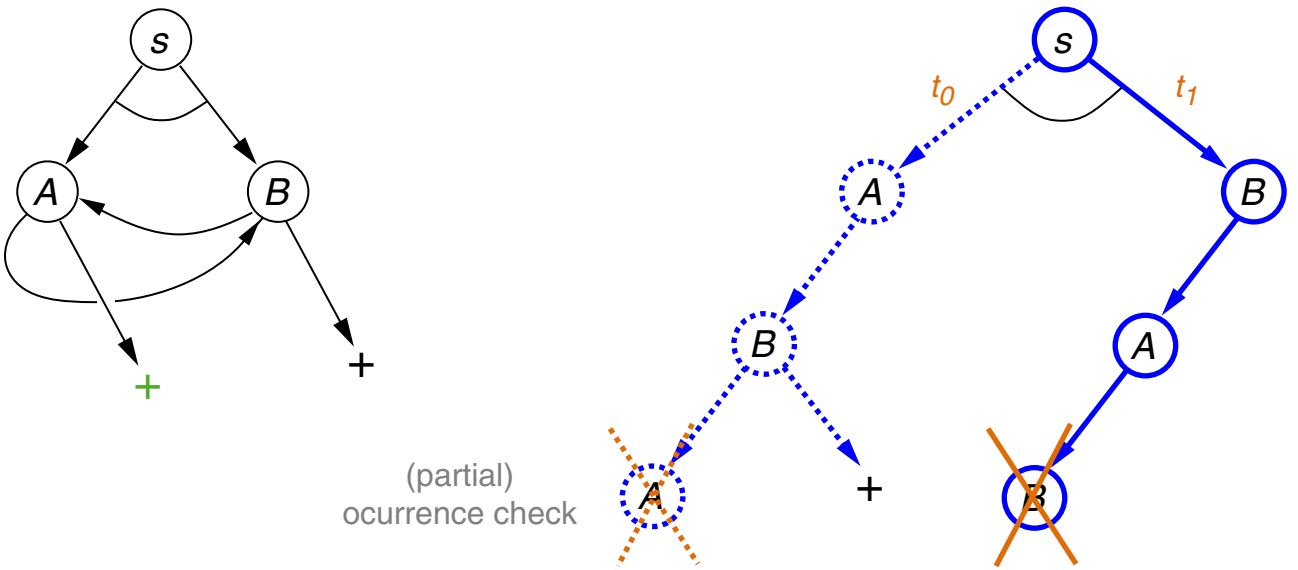
AND-OR graph (left) and its exploration (= unfolding) with DFS (right):



Depth-First Search of AND-OR Graphs

Dealing with Cycles (2)

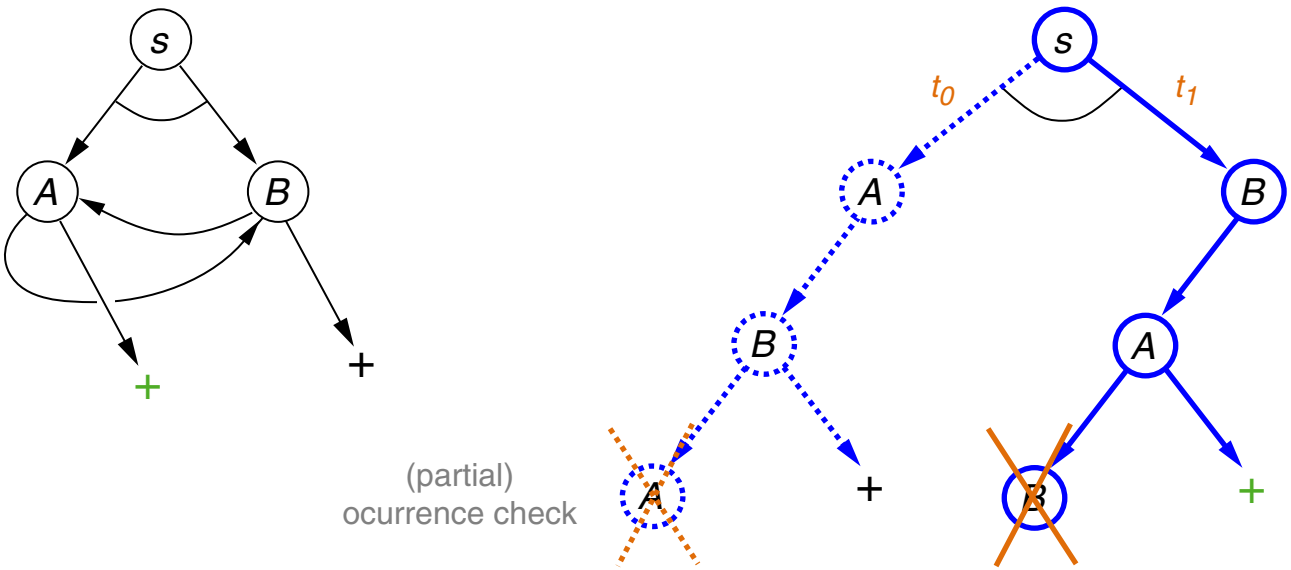
AND-OR graph (left) and its exploration (= unfolding) with DFS (right):



Depth-First Search of AND-OR Graphs

Dealing with Cycles (2)

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



Remarks:

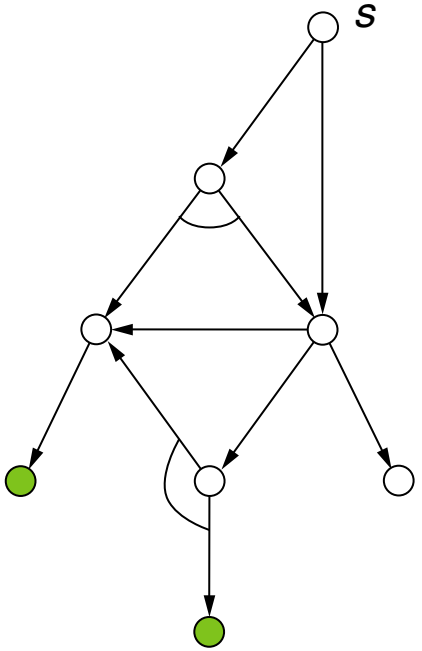
- Q. Why does the discarding of recurring nodes (partial occurrence check) not compromise the completeness of solved-labeling?
- Recapitulation. When searching an AND-OR graph with DFS, a partial occurrence check cannot address the issue of redundant problem solving in general.
- Given the previous AND-OR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

AND-OR Graph Search

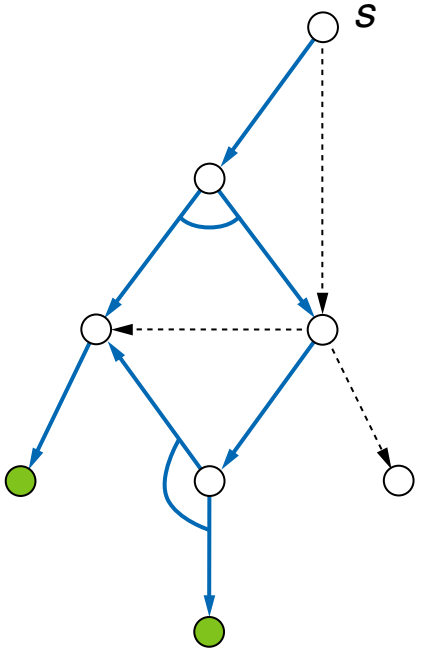
Enfolding of Solution Trees: Compact Representation as AND-OR Graph

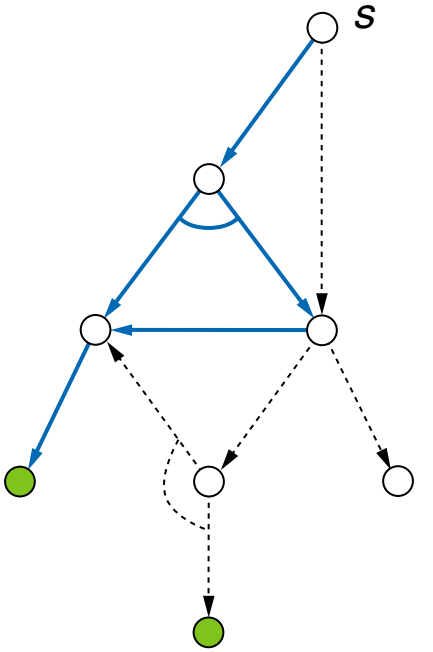
Advantage: Avoiding multiple solving of the same problem.

AND-OR graph example:



Two possible solution graphs:





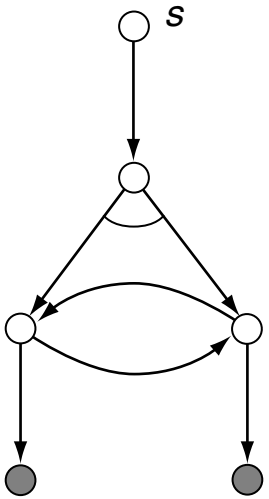
Idea: Multiple instances of nodes from the AND-OR graph G are merged.

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

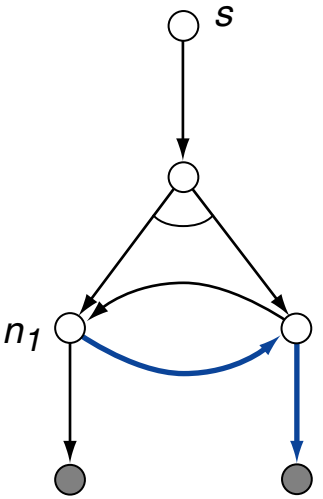
AND-OR Graph Search

Enfolding of Solution Trees

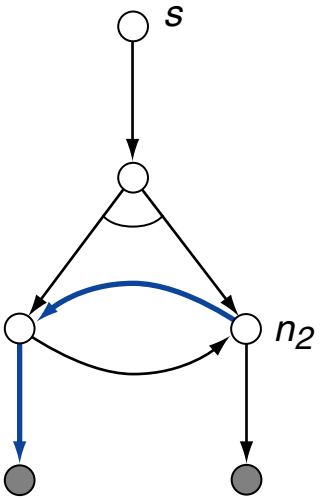
Enfolding of solution trees can result in cyclic AND-OR graphs (no unique structure):



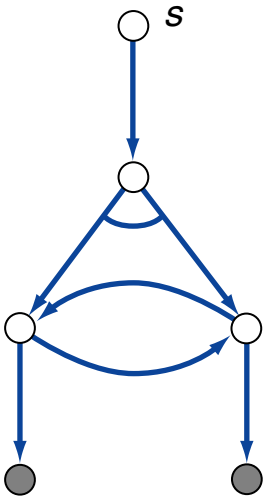
AND-OR graph



Solution graph for n_1



Solution graph for n_2

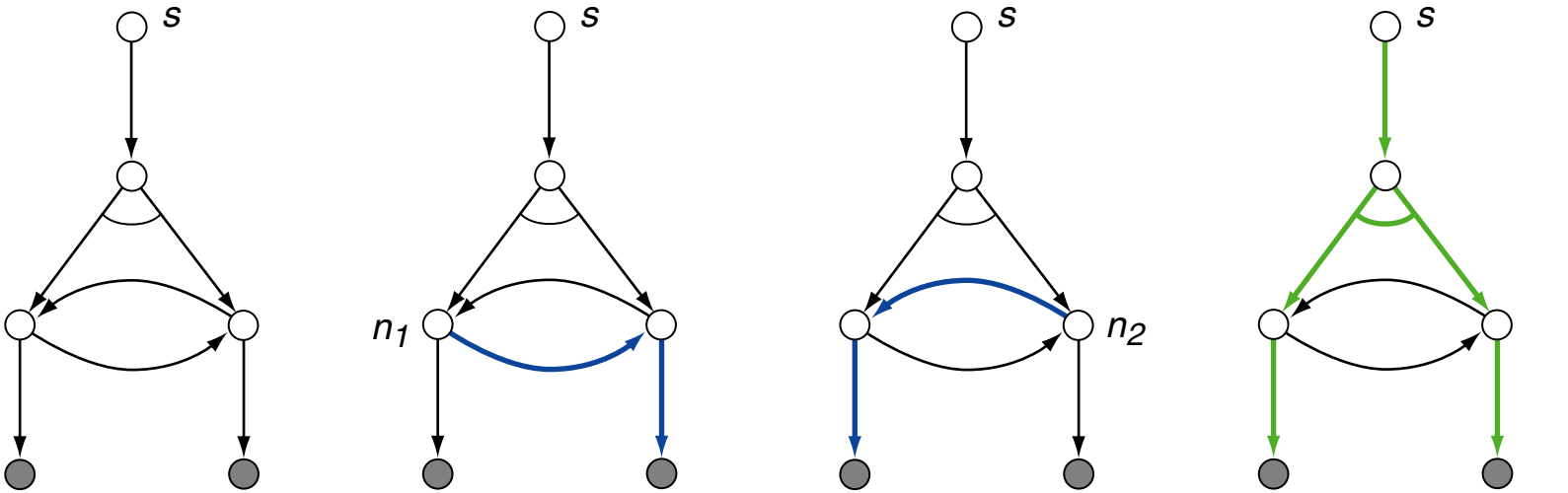


Solution graph for *s ??*

AND-OR Graph Search

Enfolding of Solution Trees

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



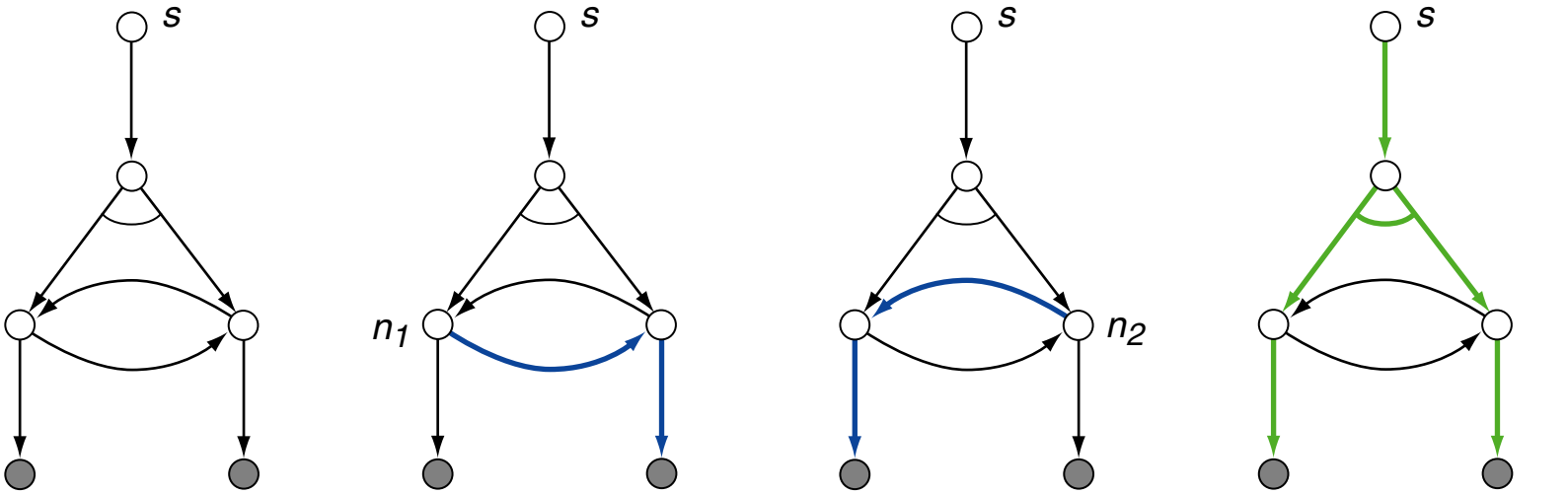
AND-OR graph | Solution graph for n_1 | Solution graph for n_2 | Acyclic solution graph for s

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

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 | Acyclic solution graph for s

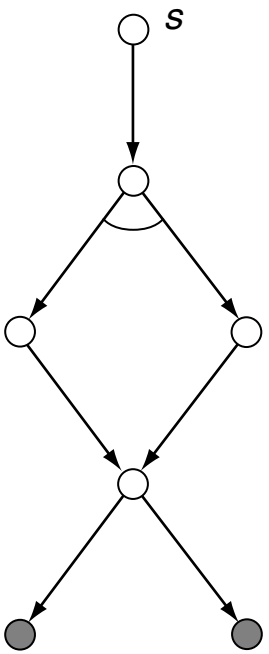
Consequence: AND-OR solution graphs are required to be acyclic.

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

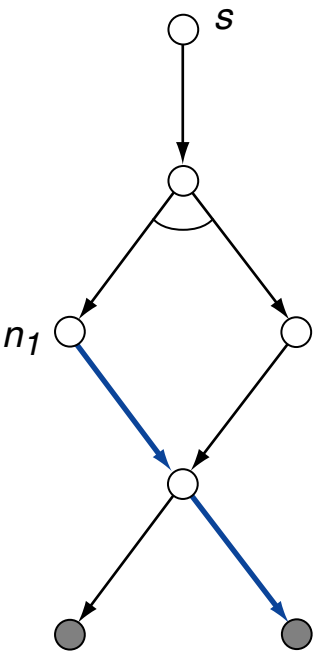
AND-OR Graph Search

Enfolding of Solution Trees (continued)

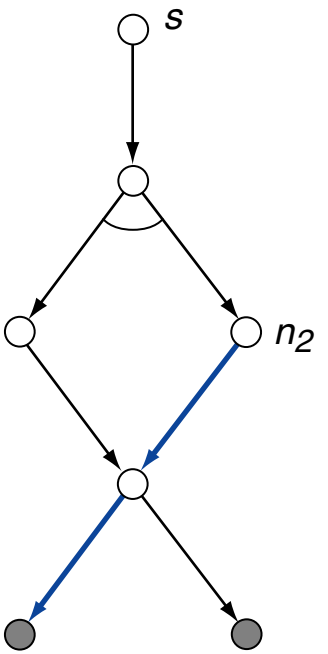
Synthesized solution graphs are not necessarily minimum:



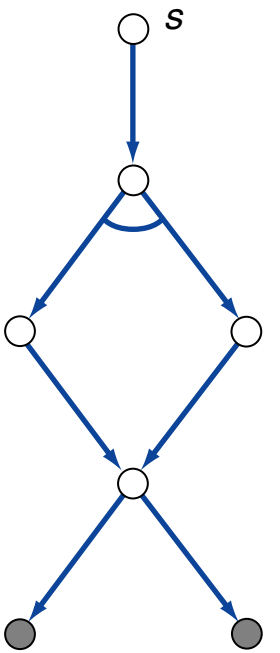
AND-OR graph



Solution graph for n_1



Solution graph for n_2

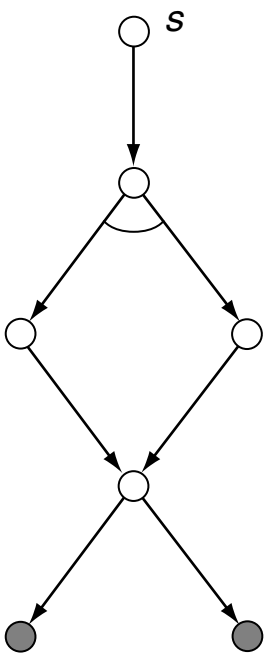


Solution graph for *s ??*

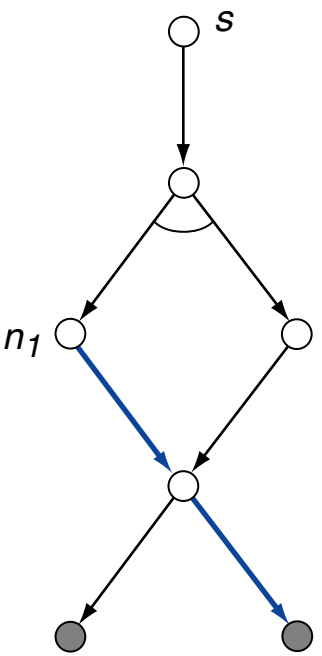
AND-OR Graph Search

Enfolding of Solution Trees (continued)

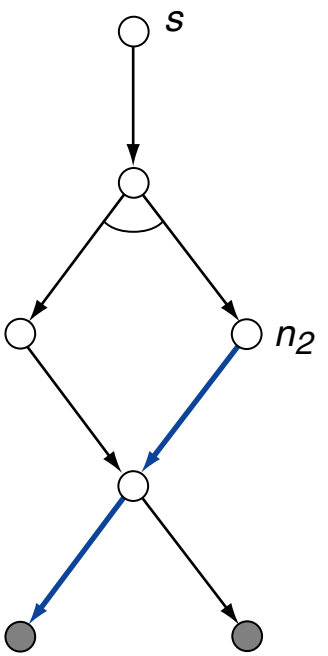
Synthesized solution graphs are not necessarily minimum:



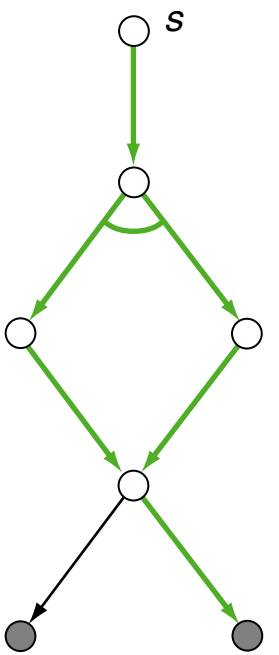
AND-OR graph



Solution graph for n_1



Solution graph for n_2

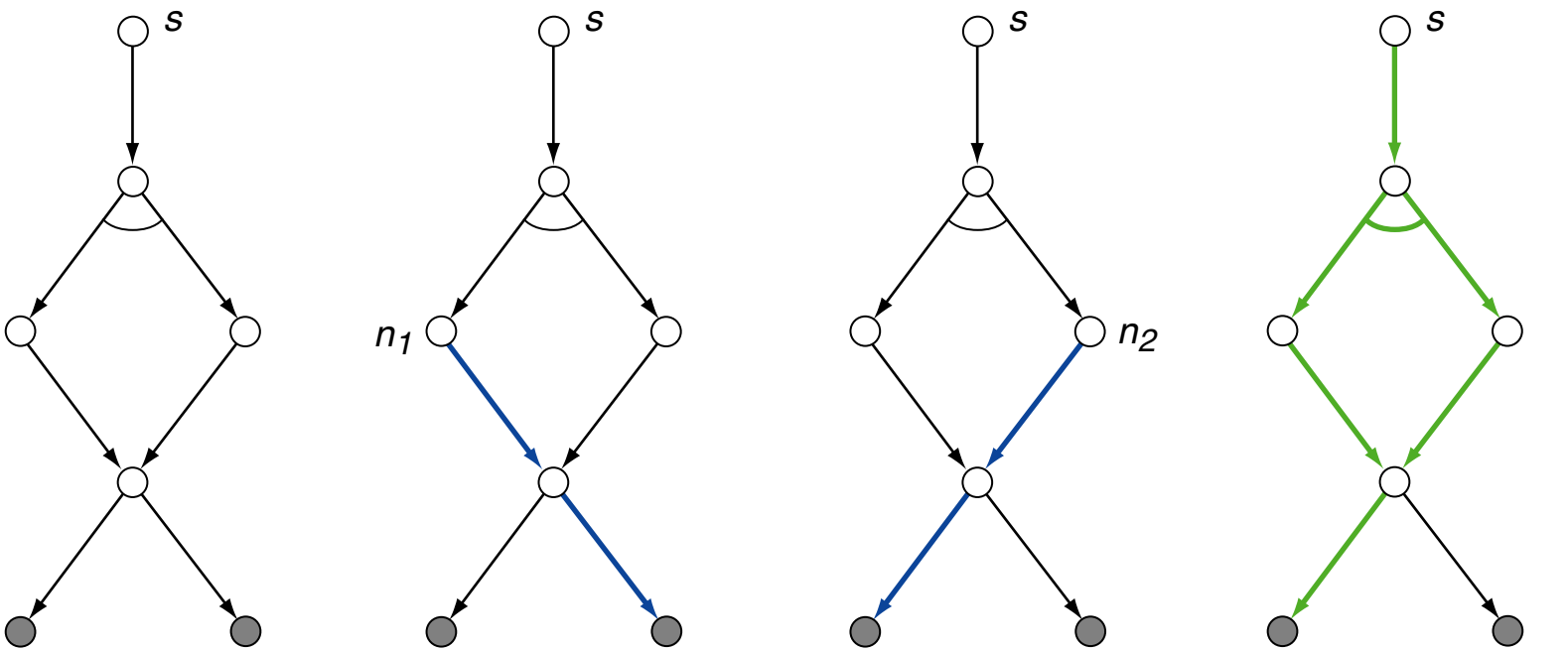


Minimum solution graph for s

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.

AND-OR Graph Search

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

Definition 14 (Solution Graph 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 subgraph H of G 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.
3. If H contains an inner OR node n, then H contains exactly one link to a successor node of n in G and this successor node.
4. If H contains an inner AND node n, then H contains all links to its successor nodes of n in G and all these successor nodes.
5. The leaf nodes in H are goal nodes in G.
6. H satisfies the additional solution constraints.
7. H is minimal: it contains no additional nodes or edges.

The acyclicity condition on H can be omitted if search is restricted to acyclic AND-OR graphs G.

Remarks:

- If H is a solution graph 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 graph for a node 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. E.g., a solution graph for some AND node is given by combining solution graphs of its successors.
- A solution graph is compact in the sense that it does not contain multiple instances of identical rest problems.
- 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.
- Because of condition (1) in the above definition, searching for a solution graph s in an AND-OR graph corresponds to searching for a cycle-free solution path in an OR graph.

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 look like follows:

  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).

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

AND-OR Graph Search

Solution Graphs in AND-OR Graphs (continued)

Definition 15 (Solution Base in an AND-OR Graph)

Let G be an AND-OR graph, and let n be a node in G. A *solution base H for n in G* is defined in the same way as a solution graph 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 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 graph G.

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

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.

AND-OR Graph Search

Algorithm: Generic_AND-OR_Search

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 tree.

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

AND-OR Graph Search [Generic_AND_OR_Tree_Search] [Generic_OR_Search]

Generic_AND-OR_Search(s, *successors*, $\star$)

```
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
```

Remarks:

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

  Generic_AND-OR_Search 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.

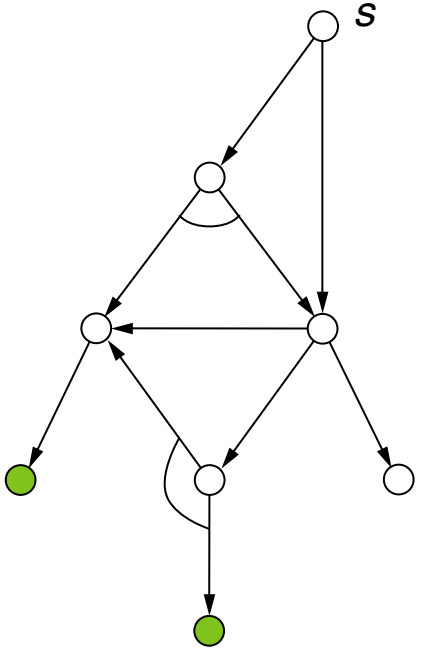
- To be precise, function *add*$(b, n, .)$ should include a check whether the resulting solution bases are still acyclic. Cyclic solution bases have to be discarded. (In order to avoid a test for cycles, it is often assumed that the search space graph G is acyclic.)

- Solution bases handled in Generic_AND-OR_Search 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.

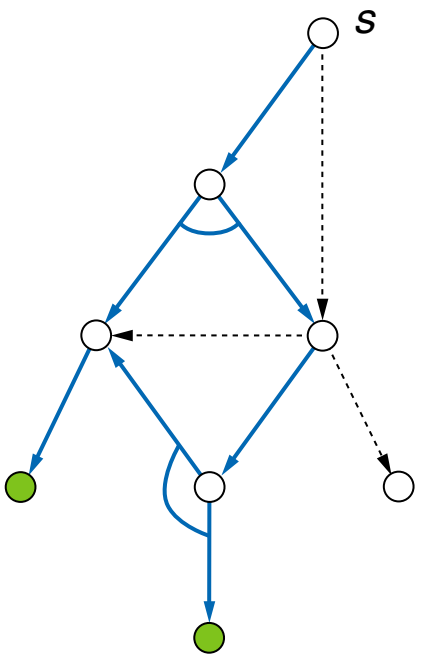
AND-OR Graph Search

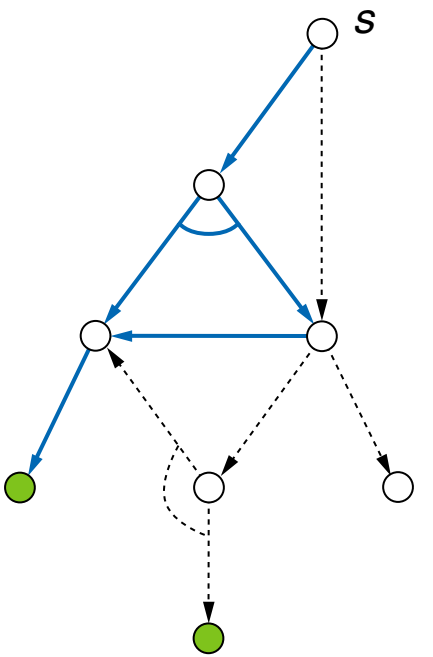
Efficient Storage of Solution Graphs

AND-OR graph example:



Two possible solution graphs:





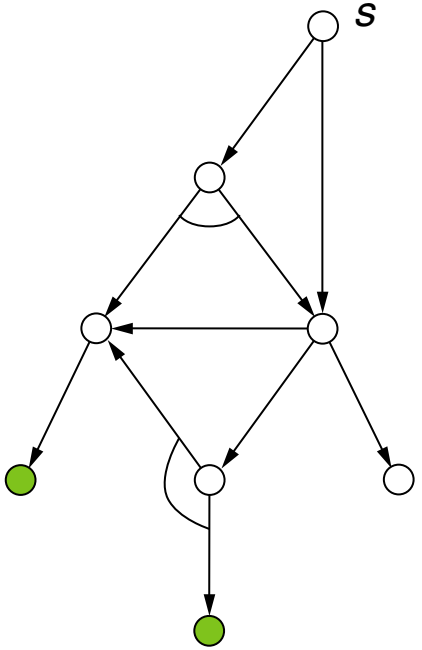
Solution bases can share multiple and completely different parts.
Node expansion can happen at any terminal node in a solution base. There is no partial order on solution bases.

Q. Is there a benefit of using backpointers?

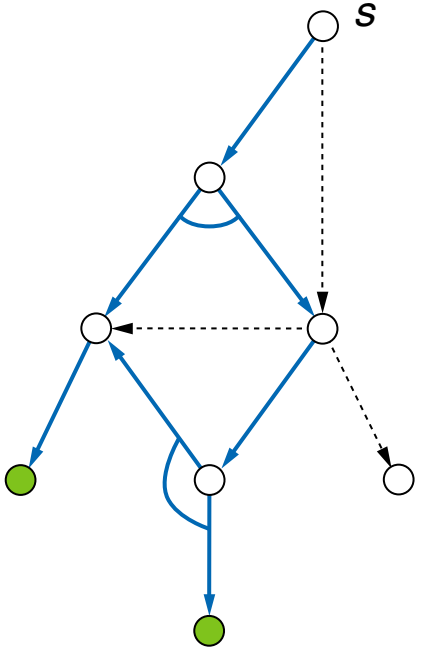
AND-OR Graph Search

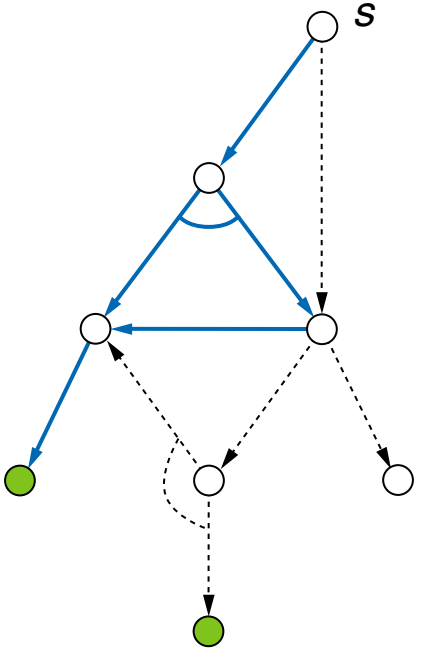
Solution Graphs and Backpointers: Sharing Initial Parts

AND-OR graph example:



Two possible solution graphs:

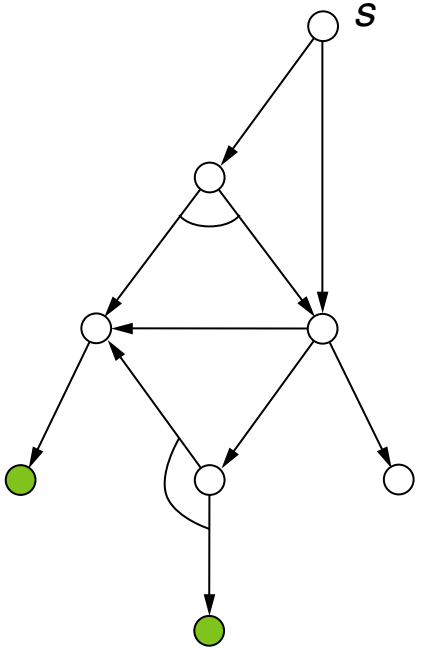




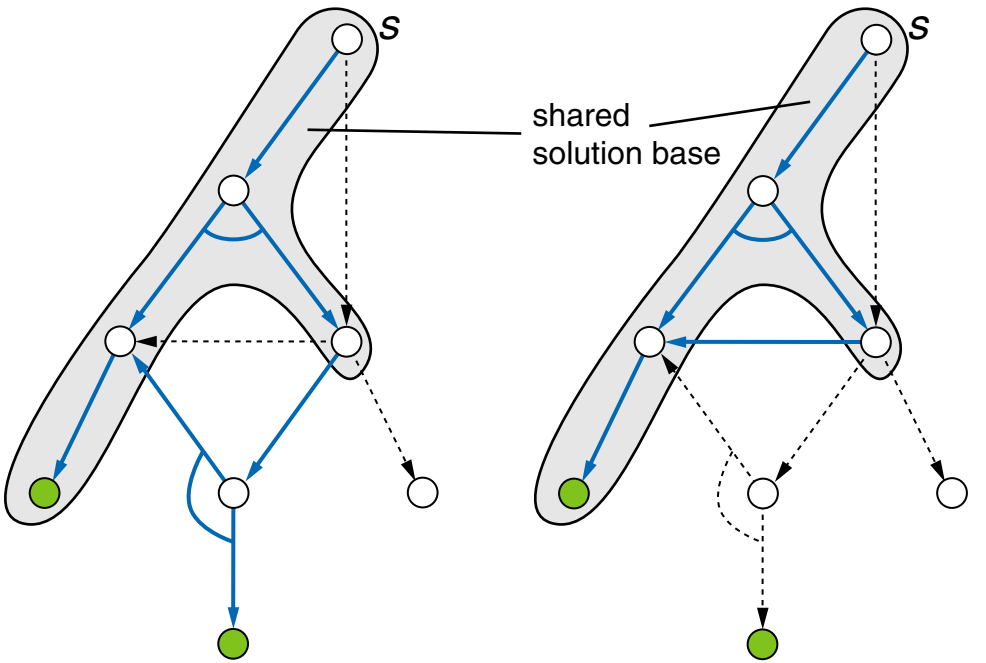
AND-OR Graph Search

Solution Graphs and Backpointers: Sharing Initial Parts

AND-OR graph example:



Two possible solution graphs:



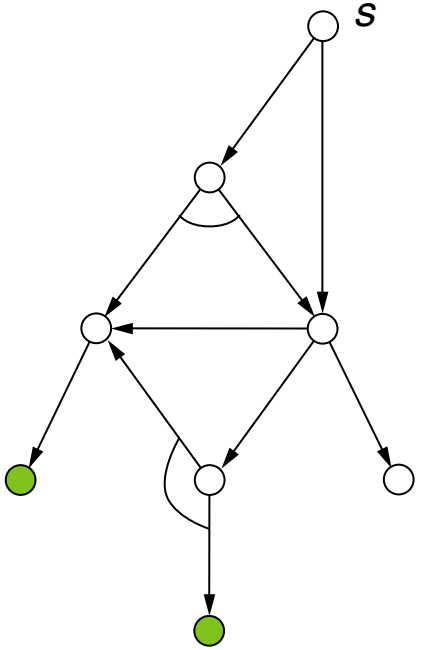
Backpointers can be used to store solution bases efficiently if a solution base is included completely.

Problem. Which solution bases are shared depends on the order of node expansions.

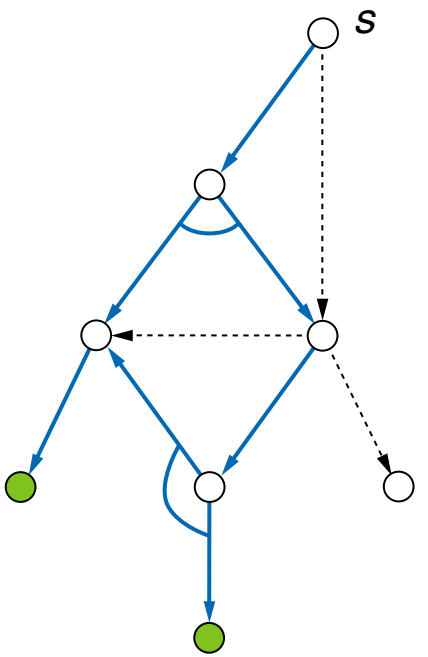
AND-OR Graph Search

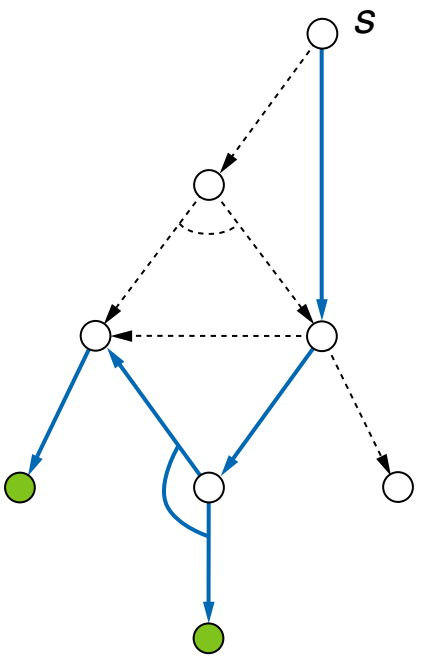
Solution Graphs and Backpointers: Sharing Solution Graphs

AND-OR graph example:



Two possible solution graphs:

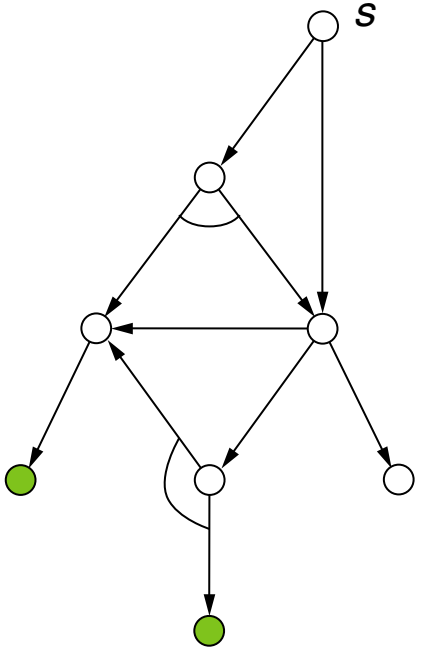




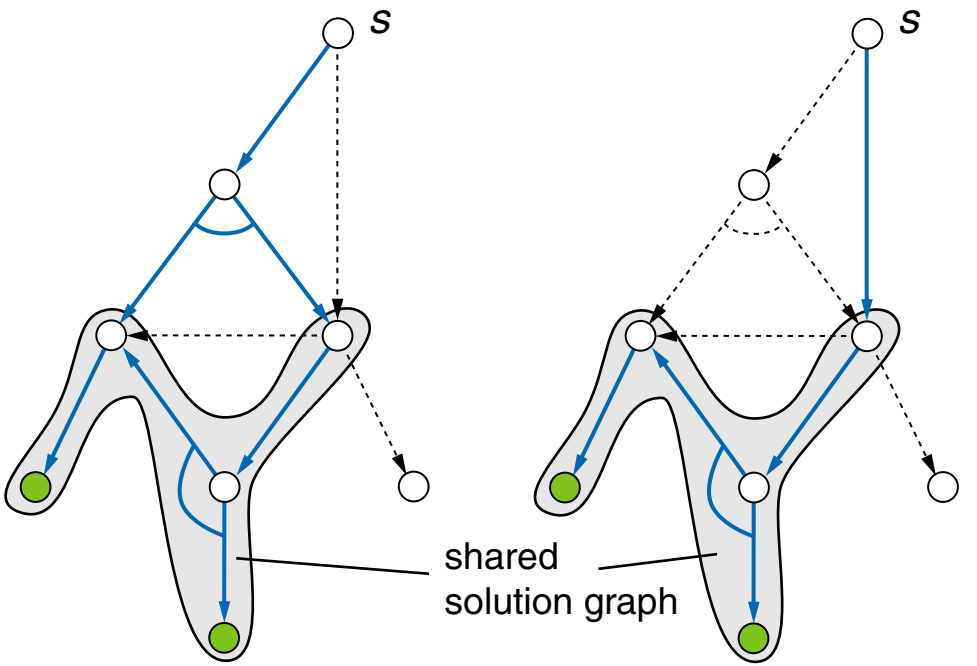
AND-OR Graph Search

Solution Graphs and Backpointers: Sharing Solution Graphs

AND-OR graph example:



Two possible solution graphs:



Solution graphs for subproblems occur in different solution graphs.

Problem. For sharing of solution graphs more than one backpointer per node is needed.

AND-OR Graph Search

Efficient Storage of Solution Bases

Idea: Since each solution base is a subgraph of G, these subgraphs can easily be combined and for a finite subgraph G_e of G, the explored part of G.
G_e is stored explicitly, i.e. edges from nodes to direct successors are stored.

Advantage:

- Only a single instance is stored for each node of G in G_e.
- Extensions by node expansions become available to all solution bases that are currently contained in G_e.
- DFS can be applied to G_e, e.g. to determine solution graphs that are contained.

AND-OR Graph Search

Efficient Storage of Solution Bases

Idea: Since each solution base is a subgraph of G, these subgraphs can easily be combined and for a finite subgraph G_e of G, the explored part of G.
G_e is stored explicitly, i.e. edges from nodes to direct successors are stored.

Advantage:

- Only a single instance is stored for each node of G in G_e.
- Extensions by node expansions become available to all solution bases that are currently contained in G_e.
- DFS can be applied to G_e, e.g. to determine solution graphs that are contained.

Idea: Backpointers connect each node with all of its expanded parents in G_e.
combined and for a finite subgraph G_e of G, the explored part of G.
G_e is stored explicitly, i.e. edges from nodes to direct successors are stored.

Advantage:

- Sharing of solution subgraphs is possible.
- Information can be propagated in any possible way in direction to the start node s.

AND-OR Graph Search

Foundation of AND-OR Graph Search

General Structure.

- Algorithms maintain the explicitly explored part G_e of the AND-OR graph G.
- 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 backpointers 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.

AND-OR Graph Search

Algorithm: Basic_AND-OR_Search

Input: s. Start node representing the initial problem.
successors(n). Returns the successors of node n.
is_goal_node(n). Predicate that is *True* if n is a goal node.

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

AND-OR Graph Search [Basic_OR_Search]

```
Basic_AND-OR_Search(s, successors, ⋆)

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.    H = choose_solution_base(s, G_e);    // Choose solution base for s in G_e.
      n = choose(OPEN, H);    // Choose OPEN 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.
          insert(n', OPEN); add_node(n', G_e);
        ENDIF
        add_backpointer(n', n);
        IF ( is_goal_node(n') OR is_labeled(n') )    // Is n' solvable?
        THEN
          propagate_label(n');
          IF is_solved(s) THEN RETURN(compute_solution_graph(G_e));
        ENDIF
      ENDDO
6.  ENDLOOP
```

Remarks:

- Algorithm Basic_AND-OR_Search takes a graph-oriented perspective:
- A solution base H that needs expansion can be computed from G_e if there are OPEN nodes available.
- A single OPEN node usually does not represent a solution base (if nonterminal AND node is contained in the backpointer path). Here, OPEN can be seen as the search frontier, the border line between the explored and the unexplored part 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 in DFS], not only a path, but a directed acyclic graph is defined by the backpointers starting in a node n.
- Even if the propagation of information "solvable" guarantees that s is solved, the found solution graph does not have to be an extension of H. Therefore, function *choose_solution_graph*$(G_e))$ searches for a solution graph in G_e, e.g. using DFS.
- Again, no additional solution constraints are considered here.

Remarks (continued):

- Testing whether are 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 backpoiters and simultaneously to avoid cycles. This is again a justification for assuming that the search space graph G is acyclic.