Chapter S:II

II. Basic Search Algorithms

- Systematic Search
- Graph Theory Basics
- State Space Search
- Depth-First Search
- Backtracking
- Breadth-First Search
- Uniform-Cost Search

- AND-OR Graph Basics
- Depth-First Search of AND-OR Graphs
- AND-OR Graph Search Basics
- AND-OR Graph Search

Systematic Search

Types of Problems

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

- 8-Queens Problem: positions for 8 queens on a chessboard.
- 8-Puzzle Problem: finite sequence of moves for the tiles.
- TSP: travel tour given as order of visits to the cities.

In particular, a desired solution is expected to be in S. A systematic search should consider all solution candidates, but each at most once.

Systematic Search

Types of Problems

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

- 8-Queens Problem: positions for 8 queens on a chessboard.
- 8-Puzzle Problem: finite sequence of moves for the tiles.
- TSP: travel tour given as order of visits to the cities.

In particular, a desired solution is expected to be in S. A systematic search should consider all solution candidates, but each at most once.

Distinguish two problem types:

1. Constraint satisfaction problems.

   A solution has to fulfill constraints.

2. Optimization problems.

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

Remarks:

- ❑ General Remark:
  In this and further parts, we will consider the constructive approach to search, i.e. finding a solution is modeled as problem solving.

- ❑ Constraints can characterize required properties of a solution candidate to be solution:
  - 8-Queens: board configuration without threat on any queen;
  - 8-Puzzle: move sequence leads to target board configuration;
  - TSP: travel tour forms a Hamiltonian cycle in the TSP graph.

- ❑ Although we are not interested in finding optimum solutions to constraint satisfaction problems, a solution should be found with minimum search effort. Therefore, we will apply approaches to solve optimization problems to constraint satisfaction problems as well.

- ❑ In optimization, a target function is given, a desired solution should maximize (or minimize). We will use *solution cost* as optimization criterion which is to be minimized. Another option is to use *merit*, e.g., utility, as optimization criterion which is to be maximized.
  In constraint satisfaction, cost can be used as an additional constraint for a desired solution, e.g., that it costs C maximum (or minimum) for a given bound C.

- ❑ We require a function $\star($*solution_candidate*$)$ which can test whether the constraints for a solution are fulfilled.

- ❑ Q. Is it possible to pose the 8-Queens problem as (a special case of) an optimization problem?

Systematic Search

Modeling Search as Problem Solving

Task: Find a solution within the search space of solution candidates.

Local Search: Consider solution candidates directly.

Construction: Consider the task as a problem and search for a sequence of problem solving steps.

Ingredients:

- We are dealing with (remaining / rest) problems.
- The start problem describes the search task.
- A set of operators (inspired by the search task) is available that allow to solve / simplify a problem at hand.
- We are done when the remaining problem is trivial (solved).
- We are trying to solve the start problem by providing a sequence of problem solving / simplification steps that leads to a trivial rest problem.

Systematic Search

Abstract Framework for Search as Problem Solving

Definition 1 (State Transition System STS)

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

- S, a set of states (remaining / rest problems),
- $T \subseteq S \times S$, a transition relation (problem solving / simplification steps),
- $s \in S$, a start state (start problem), and
- $F \subseteq S$, a set of final states (trivial rest problems).

Observation:

- A search task is reformulated as the task of finding, in a state transition system (S, T, s, F), a sequence of transitions s.t. a $g \in F$ is reached from s.
- (S, T) defines a directed graph, known as the *state space graph*.
- Because (S, T) is closely connected the search space of solution candidates and because (S, T) defines the search space of paths from s to states in F, we call this graph also the *search space graph* of a search problem.

Remarks:

- Transitions (s_0, s_1) are entirely local.

  A transition (s_0, s_1) can be used to effect a state change regardless of which transition led to the s_0 state or which transitions will be used to change s_1.

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

  Reachability can be shown by providing a corresponding sequence of transitions.

  Reachability corresponds to the existence of paths in (S, T).

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

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

- Transitions in an STS can be considered operation causing a state change. Often, transitions can be grouped or categorized, e.g. in case of the 8-Puzzle Problem where we have transitions that are instances of operators "moving left", "right", "up", and "down". Therefore, labeled STSs can be used to represent such operator information. We will assume that such information can also be determined from simply knowing a state and its successor state w.r.t. some transition.

Systematic Search

Modeling Problem Solving as Reachability Problem for STS

Processes, especially problem solving, can be modeled as state transition systems, whereby specific questions and related knowledge can be captured.

States are certain situations (remaining problems) of a process that can be characterized by unique descriptions.

Transitions are state changes of a process that are initiated by some rules or operations or actions.

Final States are certain states of a process, (not necessarily terminal).

Problem is to check for an initial state, whether a final state can be reached considering certain (solution) constraints.

Solution is then a suitable sequence of states (transitions).

Solution Candidates are finite sequences of transitions starting in s.

Search Space is the set of all finite sequences of transitions starting in s.

Systematic Search

Modeling Problem Solving as Reachability Problem for STS

Processes, especially problem solving, can be modeled as state transition systems, whereby specific questions and related knowledge can be captured.

States are certain situations (remaining problems) of a process that can be characterized by unique descriptions.

Transitions are state changes of a process that are initiated by some rules or operations or actions.

Final States are certain states of a process, (not necessarily terminal).

Problem is to check for an initial state, whether a final state can be reached considering certain (solution) constraints.

Solution is then a suitable sequence of states (transitions).

Solution Candidates are finite sequences of transitions starting in s.

Search Space is the set of all finite sequences of transitions starting in s.

Systematic Search

Modeling Problem Solving as Reachability Problem for STS

Processes, especially problem solving, can be modeled as state transition systems, whereby specific questions and related knowledge can be captured.

States are certain situations (remaining problems) of a process
that can be characterized by unique descriptions.

Transitions are state changes of a process
that are initiated by some rules or operations or actions.

Final States are certain states of a process, (not necessarily terminal).

Problem is to check for an initial state, whether a final state can be reached
considering certain (solution) constraints.

Solution is then a suitable sequence of states (transitions).

Solution Candidates are finite sequences of transitions starting in s.

Search Space is the set of all finite sequences of transitions starting in s.

Remarks:

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

Graph Theory Basics

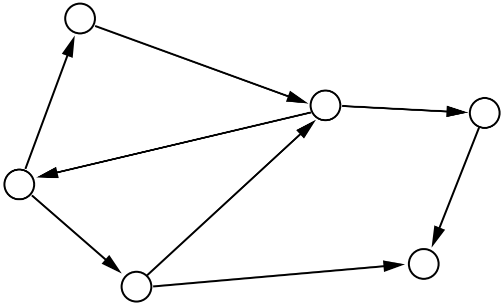
Definition 2 (Directed Graph)

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

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

For two vertices v, v' in V that are connected with an edge $e = (v, v')$, the vertex v' is called immediate / direct successor or child of v; the vertex v is called immediate / direct predecessor or parent of v'.

For an edge $e = (v, v')$, the vertices v, v' are adjacent, and the vertex-edge combination v, e and v', e are called incident, respectively.



Graph Theory Basics

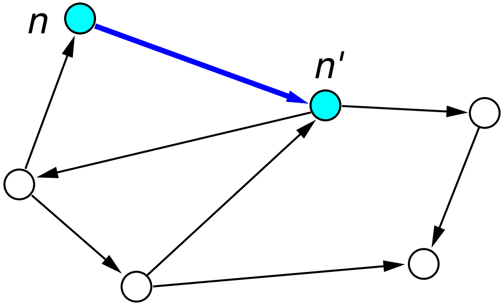
Definition 2 (Directed Graph)

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

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

For two vertices v, v' in V that are connected with an edge $e = (v, v')$, the vertex v' is called immediate / direct successor or child of v; the vertex v is called immediate / direct predecessor or parent of v'.

For an edge $e = (v, v')$, the vertices v, v' are adjacent, and the vertex-edge combination v, e and v', e are called incident, respectively.



Remarks:

- Graphs are not restricted to be finite (i.e., graphs can have an infinite set of vertices V and, in this case, an infinite set of edges E as well).

- Here, we restrict ourselves to simple graphs (i.e., we do not allow multiple edges connecting a vertex v to a successor vertex v' in a graph as it may occur in multigraphs). If multiple edges are needed, they can be simulated by introducing unique intermediate vertices in such edges.

- Further, we restrict ourselves to graphs without loops, i.e., there is no edge connecting a vertex to itself. If loops are needed, they can be simulated by introducing unique intermediate vertices so that loops become cycles of length 2.

- When dealing with undirected graphs, we assume that an undirected edge $\{v, w\}$ is an abbreviation for the two directed edges (v, w) and (w, v).

Graph Theory Basics

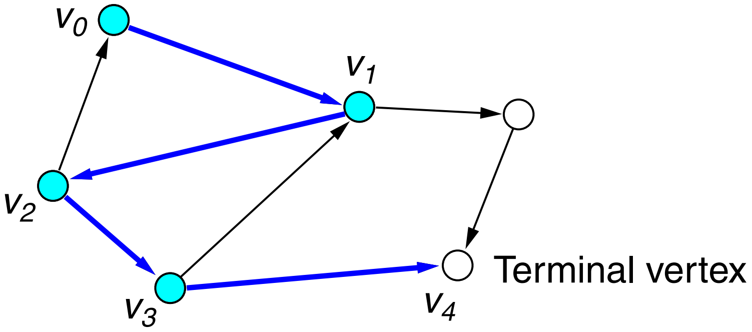
Definition 3 (Path, Vertex Types, Outdegree, Local Finiteness)

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

A sequence $P = (v_0, v_1, \dots, v_k)$ of vertices, where each v_i, $i = 1, \dots, k$, is direct successor of v_{i-1}, is called (directed) *path* of length k from vertex v_0 to the vertex v_k.

The vertices $v_1, \dots, v_k$ are called *successors* or *descendants* of the vertex v_0. The vertices $v_0, \dots, v_{k-1}$ are called *predecessors* or *ancestors* of the vertex v_k. A vertex without a successor is called *terminal*.

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



Graph Theory Basics

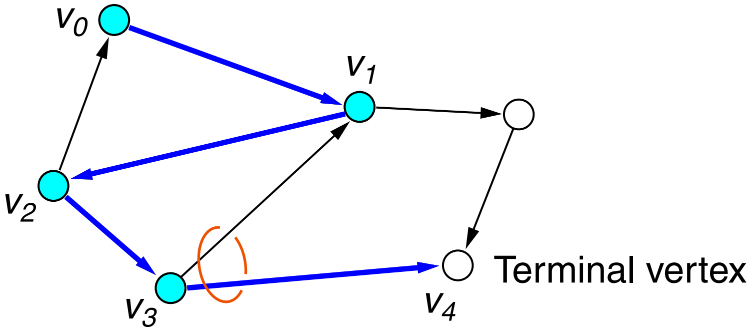
Definition 3 (Path, Vertex Types, Outdegree, Local Finiteness)

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

A sequence $P = (v_0, v_1, \ldots, v_k)$ of vertices, where each v_i, $i = 1, \ldots, k$, is direct successor of v_{i-1}, is called (directed) *path* of length k from vertex v_0 to the vertex v_k.

The vertices $v_1, \ldots, v_k$ are called *successors* or *descendants* of the vertex v_0. The vertices $v_0, \ldots, v_{k-1}$ are called *predecessors* or *ancestors* of the vertex v_k. A vertex without a successor is called *terminal*.

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



Remarks:

- If first and last vertex of a path $(v_0, v_1, \ldots, v_k)$ are of interest, we denote such a path by $P_{v_0-v_k}$.
- There is no uniform notation for graphs. Some sources define (directed) paths as special type of walks.

  A walk is a finite sequence of (directed) edges, where each two subsequent edges are incident with the same vertex. Further constraints can be made on uniqueness of vertices or edges in a walk. [Wikipedia (accessed 21-09-21)]

  As we are dealing with simple graphs (no multigraphs), the edges traversed are immediately clear from the sequence of vertices.
- As there is no uniform terminology for paths, we explicitly state that multiple occurrences of vertices are allowed in our context. So, paths can be cyclic.
- The concept of a path can be extended to cover infinite paths by defining them as an infinite sequence of vertices $(v_0, v_1, v_2 \ldots)$. An infinite path has no end vertex.
- The outdegree of a graph is the maximum of the outdegrees of its vertices.

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

Graph Theory Basics

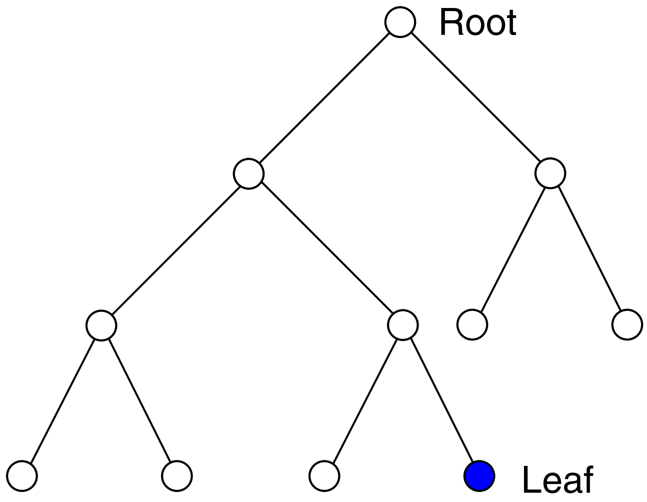
Definition 4 (Directed Tree, Uniform Tree)

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

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

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

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



Remarks:

- For trees, the indication of the edge direction usually will be omitted, since all edges are equally oriented, from the root vertex towards the leaf vertices.
- Backpointer structures established by search algorithms match trees when all edge directions are flipped.

State Space Search

Modeling Problem Solving as Path-Finding Problem

- Processes, esp. problem solving, can be modeled as state transition systems.
- A state transition system $\mathcal{T} = (S, T, s, F)$ defines a graph (S, T) of possible transitions.

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

Vertices represent states of a process.

Edges represent state transitions of a process.

Goal Vertices represent goal states (solved problems).

Problem is to decide for vertex s whether a goal vertex is reachable considering certain (solution) constraints.

Solution is then a suitable path from s to a goal vertex.

Solution Candidates are finite paths starting from s.

Search Space is the set of all finite paths starting from s.

State Space Search

Modeling Problem Solving as Path-Finding Problem (continued) [Problem-Reduction]

Definition 5 (State-Space, State-Space Graph)

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

Naming conventions:

- Vertices of in a state-space graph are called states.
- The edges in a state-space graph define alternatives how the parent problem can be simplified. The edges are called OR edges (OR links).
- States with only outgoing OR edges are called OR states.
- The state-space graph is an OR graph (i.e., a graph that contains only OR edges).

Modeling:

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

Remarks:

- A state-space graph is a variant of the more general concept of a search space graph also includes other types of problem solving steps, such as problem decompositions.

- If state transitions in a system or a process result from the application of an operator, rule, or action, the edge in the state-space graph can be labeled accordingly.

  In the 8-Puzzle problem the operators are $\{\text{up}, \text{down}, \text{left}, \text{right}\}$. Their parameters can be uniquely determined by the states to which they are applied.

State Space Search

Modeling Problem Solving as Path-Finding Problem (continued) [Problem-Reduction]

Definition 6 (Solution Path, Solution Base, State-Space Search)

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

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

Usage:

- Solution Paths

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

- Solution Bases

  Algorithms maintain a number of promising solution bases and extend them until it turns out that one of them is a solution path.

- Constraint Satisfaction Problems

  Some solution constraints are given and we search for a solution path satisfying these constraints.

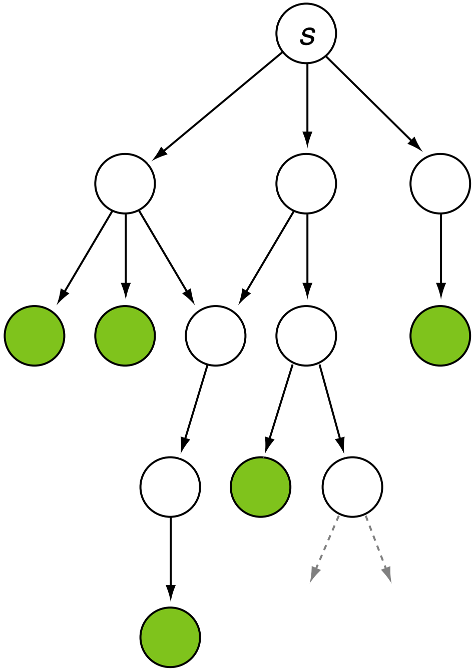
Remarks:

- Please note, that in the definition of solution paths and solution bases state s does not denote the start state. We can have solution paths and solution bases for any state in a state-space graph.
- Obviously, all solution paths contained in G are also solution bases.
- When modeling problem solving as a path-finding problem in a state-space graph G, the search space consists of all finite paths in G starting in a start state s. A solution base represents a set of candidates paths in this search space (i.e., the set of all finite paths in G that have the solution base as an initial part).
- While solution paths are often discussed with respect to the underlying state-space graph, solution bases are only considered within a finite subgraph of the state-space graph which was explored so far by an algorithm.
- If P is a solution base for a state s' in G and if s_0 is some state in P, then the subpath P_0 of P starting in s_0 and ending in the tip state of P is a solution base for s_0. This subpath P_0 is sometimes called the *solution base in P induced by s_0*.

State Space Search

Illustration of Solution Paths and Solution Bases

State-space graph:

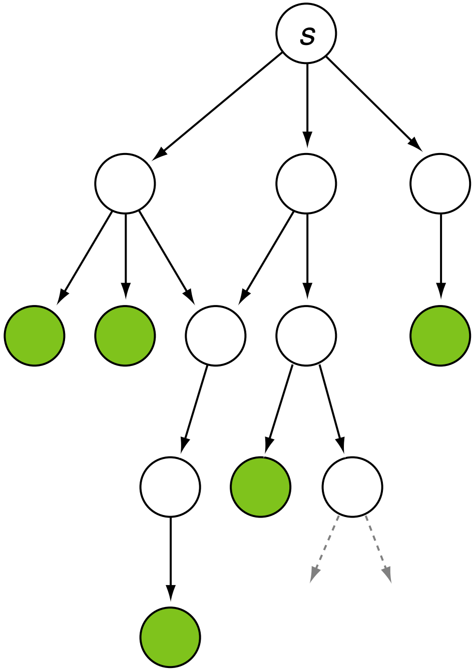


🟢 Goal state
(solved rest problem)

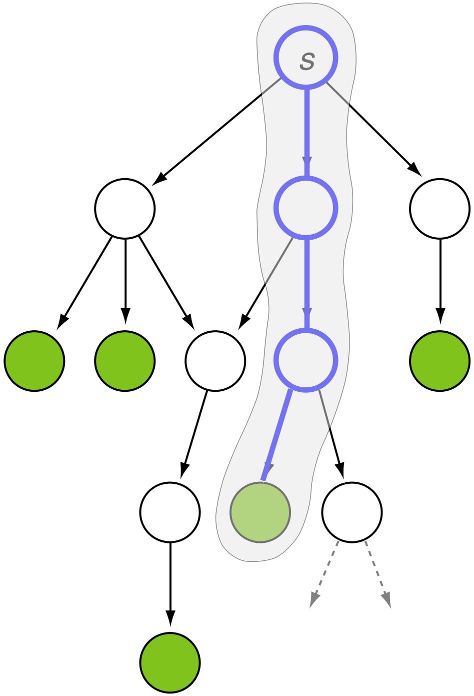
State Space Search

Illustration of Solution Paths and Solution Bases

State-space graph:



Solution path for state *s*:

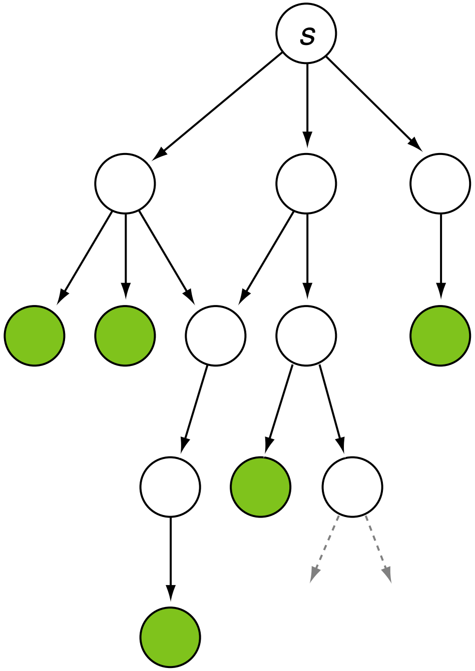


🟢 Goal state
(solved rest problem)

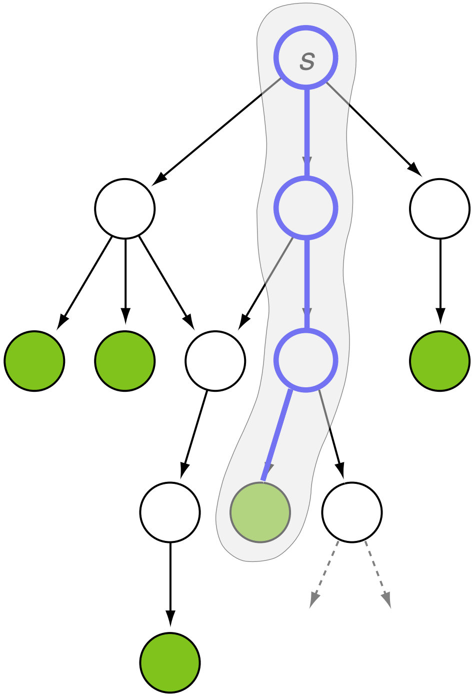
State Space Search

Illustration of Solution Paths and Solution Bases

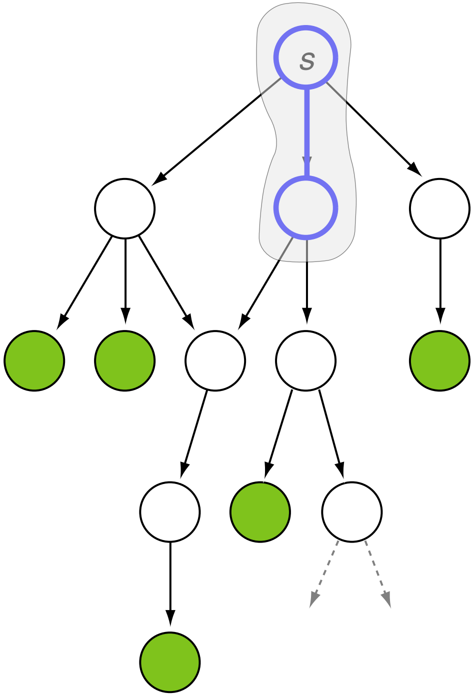
State-space graph:



Solution path for state *s*:



Solution base for *s*:



🟢 Goal state
(solved rest problem)

State Space Search

Graph Representation

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

State Space Search

Graph Representation

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

Explicit Graph (Representation)

- ❑ A graph $G = (V, E)$ is specified by listing all vertices in V and all edges in E explicitly.
- ➜ Algorithms using an explicit graph representation can handle finite graphs only.

Implicit Graph (Representation)

- ❑ A graph G is specified by listing a set S of start vertices and giving a function *successors*(v) returning the set of direct successors of vertex v in G.
- ❑ A graph G is specified by listing a set S of start vertices and giving a function *next_successor*(v) returning an unseen direct successor of vertex v in G.
- ➜ Algorithms using an implicit graph representation can handle graphs with finite set S and computable function *successors*(v) (resp. computable *next_successor*(v) and *expanded*(v)).
- ➜ We restrict ourselves to singleton sets S (i.e., we have a single start vertex s).

Remarks:

- The function *next_successor*(v) generates the direct successors of v in G one by one. In order to simplify notation and algorithms, we assume that a function *next_successor*(v) itself stores which successors have been generated so far, and returns `null` if none are left.
- An explicit representation of a graph G can be determined from its implicit representation in the following way.

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

  Then $G = (V, E)$.
- There are graphs for which no implicit representation with finite set S of start vertices exists (e.g., graphs with infinite set of vertices, but empty edge set).
- Let $G = (V, E)$ be an explicitly defined graph and let graph G' be implicitly defined by a start vertex $s \in V$ and an appropriate function *successors*(n). In general, these two graphs will not be the same. Without further notice, we will restrict ourselves to the subgraph of G induced by the set of vertices that are reachable from s.

State Space Search

Graph Algorithms and Back-pointer Structures

Let algorithm $\mathcal{A}$ process a state-space graph G with an implicit representation given by a start state s and a function *next_successor*$(.)$ respectively *successors*$(.)$.

After a finite number of steps, algorithm $\mathcal{A}$ has encountered . . .

- a finite number of states in $V(G)$ including s,
- a finite number of edges.

Information about G is stored in structures of type NODE, typically denoted by n and called nodes, containing at least

- a reference to a state in $V(G)$ that is represented,
- a reference to a parent NODE structure.
  (Applying *next_successor*$(.)$ respectively *successors*$(.)$ to the state of the parent returned the current state as successor.)

In order to simplify algorithms, we assume that *next_successor*$(.)$ respectively *successors*$(.)$ accept NODEs as arguments and return NEW NODEs.

State Space Search

Graph Algorithms and Back-pointer Structures

Let algorithm $\mathcal{A}$ process a state-space graph G with an implicit representation given by a start state s and a function *next_successor*$(.)$ respectively *successors*$(.)$.

After a finite number of steps, algorithm $\mathcal{A}$ has encountered . . .

- a finite number of states in $V(G)$ including s,
- a finite number of edges.

Information about G is stored in structures of type NODE, typically denoted by n and called nodes, containing at least

- a reference to a state in $V(G)$ that is represented,
- a reference to a parent NODE structure.
  (Applying *next_successor*$(.)$ respectively *successors*$(.)$ to the state of the parent returned the current state as successor.)

In order to simplify algorithms, we assume that *next_successor*$(.)$ respectively *successors*$(.)$ accept NODEs as arguments and return NEW NODEs.

State Space Search

Graph Algorithms and Back-pointer Structures

Let algorithm $\mathcal{A}$ process a state-space graph G with an implicit representation given by a start state s and a function *next_successor*$(.)$ respectively *successors*$(.)$.

After a finite number of steps, algorithm $\mathcal{A}$ has encountered . . .

- a finite number of states in $V(G)$ including s,
- a finite number of edges.

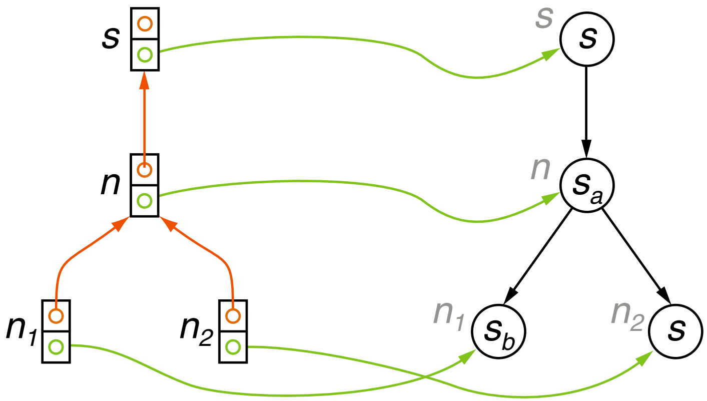
Information about G is stored in structures of type NODE, typically denoted by n and called nodes, containing at least

- a reference to a state in $V(G)$ that is represented,
- a reference to a parent NODE structure.
  (Applying *next_successor*$(.)$ respectively *successors*$(.)$ to the state of the parent returned the current state as successor.)

In order to simplify algorithms, we assume that *next_successor*$(.)$ respectively *successors*$(.)$ accept NODEs as arguments and return NEW NODEs.

State Space Search

Illustration of Back-pointer Usage



Back-pointer Structure

Corresponding Tree of Paths

Left:

(s, s_a, s_b)

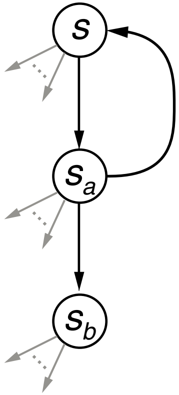
(s, n, n_1)

Right:

(s, s_a, s)

(s, n, n_2)

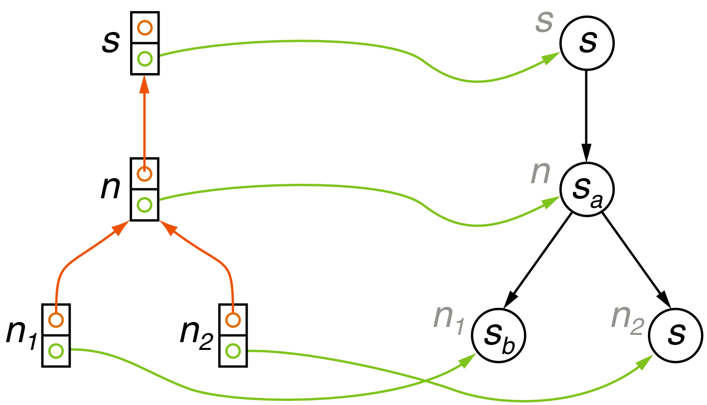
Paths in G (State Sequences)



Paths in G

State Space Search

Illustration of Back-pointer Usage



Back-pointer Structure

Corresponding Tree of Paths

Left:

(s, s_a, s_b)

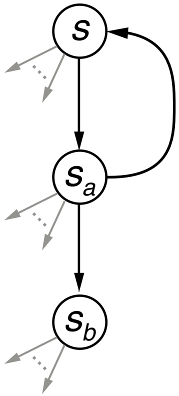
(s, n, n_1)

Right:

(s, s_a, s)

(s, n, n_2)

Paths in G (State Sequences)



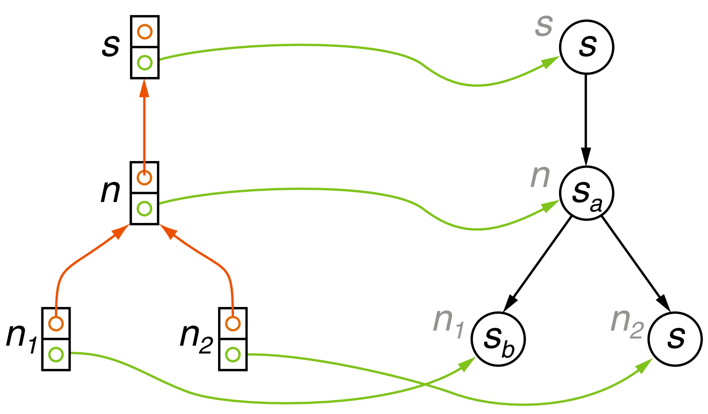
Paths in G

Paths in G

- Graph visualization appropriate for (finite) graphs.
- Emphasizing states and edges of paths does not allow to detect start and end of a path or to distinguish paths.
- Illustrating paths in a depiction of a graph requires additional information apart from states and edges.

State Space Search

Illustration of Back-pointer Usage



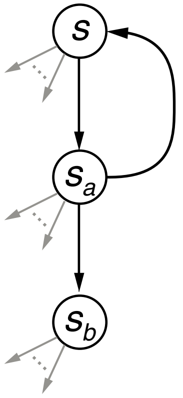
Back-pointer Structure

Corresponding Tree of Paths

Left:
(s, s_a, s_b)
(s, n, n_1)

Right:
(s, s_a, s)
(s, n, n_2)

Paths in G
(State Sequences)



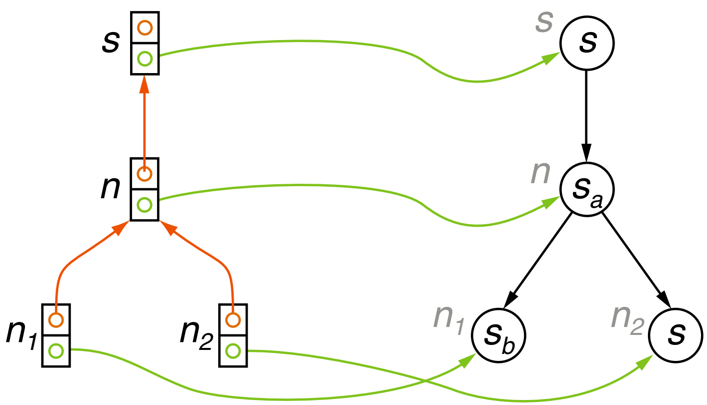
Paths in G

Paths in G (node sequences)

- ❑ Given the sequence of nodes, a path in G is uniquely described.
- ❑ States may occur multiple times in a path.
  Talking about specific positions in a path requires additional information: order of occurrences (last occurrence of s), order of states (third state in the path), naming a position (start and end of a path).
- ❑ Paths sharing an initial part cannot be detected easily.

State Space Search

Illustration of Back-pointer Usage



Back-pointer Structure

Corresponding Tree of Paths

Left:

(s, s_a, s_b)

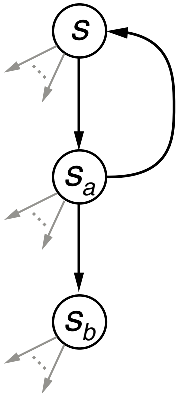
(s, n, n_1)

Right:

(s, s_a, s)

(s, n, n_2)

Paths in G (State Sequences)



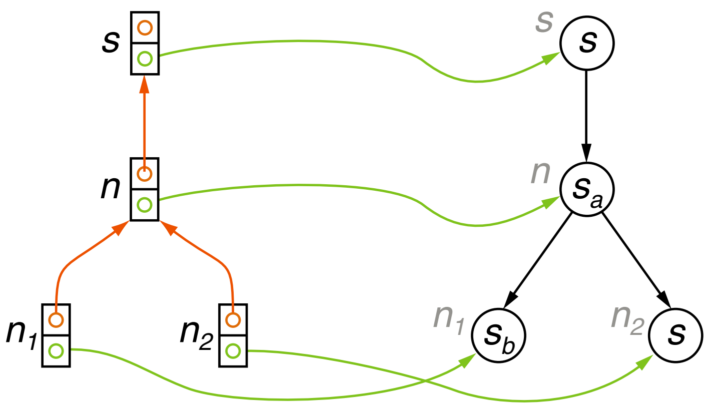
Paths in G

Tree of Paths in G

- Appropriate visualization for a (finite) set of (finte) paths with common starting state.
- Paths sharing an initial part can be detected easily.
- States may occur multiple times in paths.
  Talking about specific positions in a path requires additional information: additional naming of vertices apart from state information.
- Constructible with start state and *next_successor*$(.)$ respectively *successors*$(.)$.
- Efficiently storing paths sharing an initial part is not obvious when using edge directions.

State Space Search

Illustration of Back-pointer Usage



Back-pointer Structure

Corresponding Tree of Paths

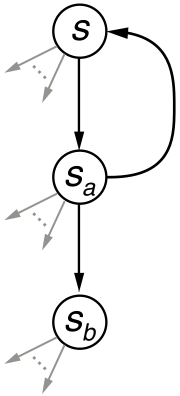
Left:
(s, s_a, s_b)

(s, n, n_1)

Right:
(s, s_a, s)

(s, n, n_2)

Paths in G
(State Sequences)



Paths in G

Back-pointer Structures for G

- Constructible with start node and *next_successor*$(.)$ respectively *successors*$(.)$.
- A name assigned to a node denotes a unique structure.
- Efficiently storing paths sharing an initial part is trivial when using opposite edge directions.
- Names for nodes can be used to specify vertices in a tree of paths.
- Names for nodes can be used to specify paths as sequence of states.
- Polymorphism: Node n is data structure, vertex in a tree of paths, path, and state.

Remarks:

- From the implementation point of view, each call of *next_successor*(n) respectively *successors*(n) with a node n results in a (list of) NEW node structures, even if the same state and/or parent is referenced as in some other node instance.

- To reduce the polymorphism, we will talk about "a path represented by a node n" or about "a node n referencing some state".

- In a slight abuse of notation, a node representing the start state s in a state-space graph is also denoted by s and a node representing a goal state γ is also denoted by γ and called a *goal node*.

- When talking about back-pointer structures, we usually consider all structures that theoretically can be constructed using some start node and *next_successor*(n) respectively *successors*(n) for nodes.
  A statement like
  "For all paths in G starting in s we have ..."
  then becomes
  "For all paths in G given by a ode n we have ..."
  or even
  "For all nodes n in G we have ..."

State Space Search

Definition 7 (Node Generation, Node Expansion)

Let G be an implicitly defined state-space graph. Let $\mathcal{A}$ be an algorithm working on G, using node structures and given the start node s and a function *next_successor*(n) respectively *successors*(n).

Applying the function *next_successor*(n) in $\mathcal{A}$ and thereby generating a new node for a successor state is called *node generation*. The new node is called *generated*, the parent is (being) *explored* when the first successor state is determined and *expanded* when *next_successor*(n) returned `null`.

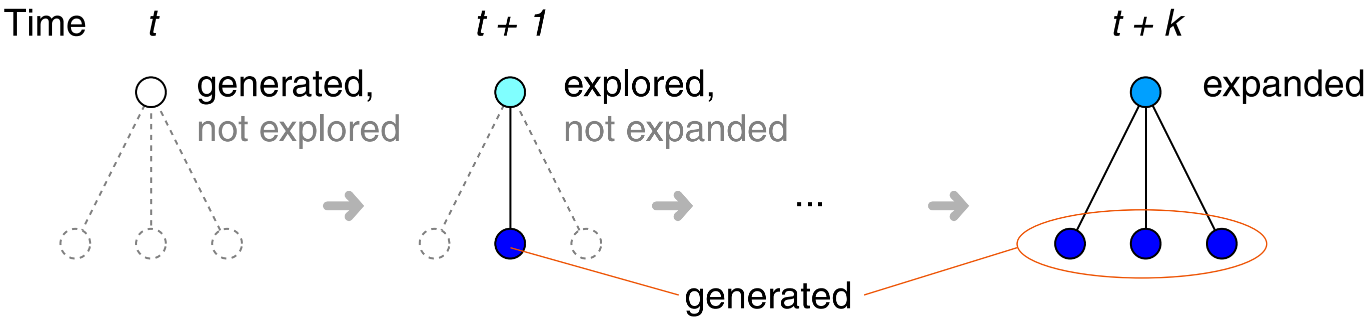
Applying the function *successors*(n) in $\mathcal{A}$ and thereby generating new nodes for all direct successor states of a node in one (time) step is called *node expansion*. The new nodes are called *generated*, the parent is *expanded*.

- Node expansion can be seen as the iterated use of node generation without interruption until all successors have been generated.

➜ Except for the algorithm Backtracking (BT), all considered algorithms will use node expansion as a basic step.

State Space Search

Node Generation as a Basic Step



Node status:

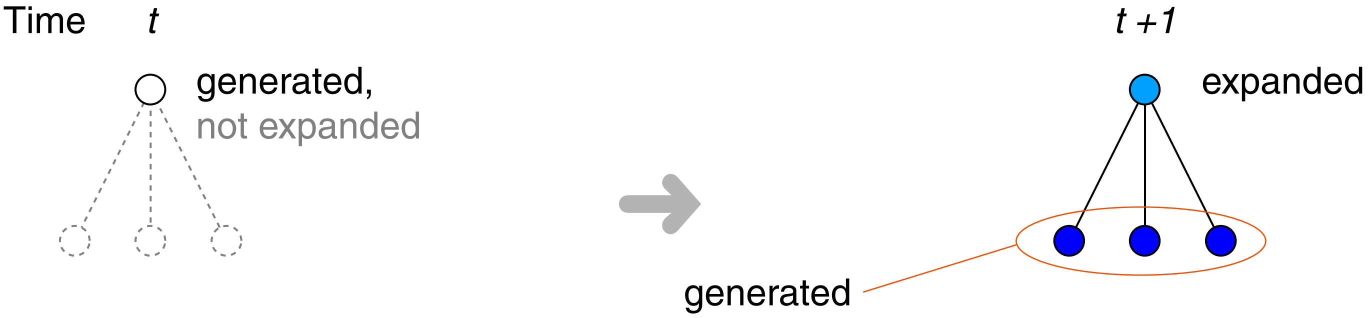
- Left: Initially, the status of the parent node has to be *generated*.
- Middle: The status of the parent node is called *explored*.
- Right: The status of the parent node is called *expanded* if all of its direct successors are generated.

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

Disadvantage: *next_successor*(n) has to keep track of generated successors.
Advantage: The algorithm itself has to store fewer nodes.

State Space Search

Node Expansion as a Basic Step



Node status:

- Left: Initially, the status of the parent node has to be *generated*.

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

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

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

Remarks:

- Using node generation as a basic step requires additional accounting: Due to the step-wise expansion, already applied operators (transitions) and thus already considered states and generated nodes must be known. Such information could be also provided, for example, by function *next_successor*(s) and stored in the generated node.

State Space Search

Explicit Parts of State-Space Subgraphs

Let $\mathcal{A}$ be an algorithm working on a state-space G, given node s and *successors*(n).

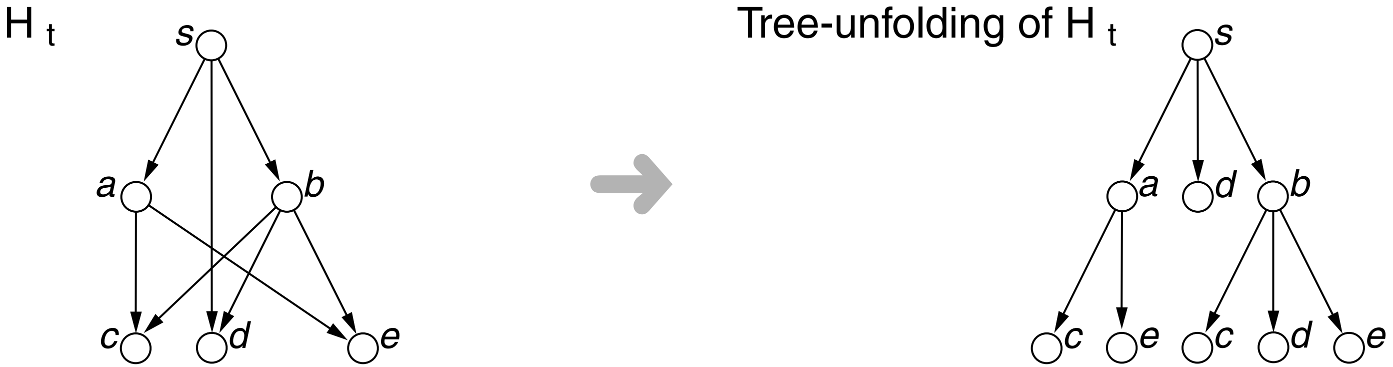
- What $\mathcal{A}$ could know about the structure of G after t node expansions:
  Explored (sub)graph H_t of G after t node expansions.
  Inductive Definition: $H_0 = (\{s\}, \{\})$

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

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

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

- What $\mathcal{A}$ could know if A does / does not care for equal states referenced in node structures:



We call H_t the explored part of G known by $\mathcal{A}$ after t node expansions.

State Space Search

Requirements for an Algorithmization of State-Space Search

Prop$_0(G)$ Required Properties of a Search Space Graph G

1. G is a state-space graph (directed OR graph).
2. G is implicitly defined.
   (a) G has a single start state s and
   (b) G has a computable function *successors*$(.)$ and *next_successor*$(.)$ returning successor states of the state given as argument.

   For a node n, the functions *successors*(n) and *next_successor*(n) respectively, always return new nodes referring to the successor states of the state referenced in node n.
3. G is locally finite.
4. A set Γ of goal states in G is given; in general Γ will not be a singleton set. Goal states are terminal states (states without successors) in G.
5. G has a computable function $\star(.)$ returning true if a given state is a goal state.

Task

❑ Determine a solution path for s in G.

Remarks:

- ❑ The terms search space, search space graph, and search graph will be used synonymously.
- ❑ Minimum requirement for a solution path is that the terminal node of a solution base is a goal node. If a solution path must satisfy given constraints, a corresponding computable function has to be available to check the constraints for a solution path.
- ❑ The property of a graph of being locally finite does not imply an upper bound for a node's degree, i.e., for the size of the node set that can be generated.
- ❑ Two nodes in an OR graph can be connected by two (oppositely) directed links, which then may be viewed as a single undirected edge. Undirected edges represent a kind of "invertible" operator.
- ❑ The functions *successors*(n) resp. *next_successor*(n) return new node instances for each successor state of an expanded node. An algorithm that does not care whether two nodes represent the same state will, therefore, consider an unfolding of the state-space graph to a tree with root s.

Remarks (continued) :

- Questions:

  1. Which of the introduced search problems give rise to undirected edges?
  2. How can multiple start states be handled?
  3. Is it possible to combine multiple goal states into a single goal state?
  4. How is it possible to guarantee that goal states are terminal (states without successors)?
  5. For which kind of problems may the search space graph not be locally finite?
  6. How can we deal with problems whose search space graph is not locally finite?
  7. What will happen if none of the goal states in Γ can be reached?

State Space Search

Generic Schema for 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) Expand the only unexpanded node in the solution base.
   (c) Extend the solution base by one successor node at a time and save it as a new candidate.
   (d) Determine whether a solution path has been 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.

State Space Search

Algorithm: Generic-OR (Compare Basic-OR.)

Input: s. Start state (initial problem) in G.
successors(s_0). Returns the successor states of s_0 in G.
$\star(s_0, \ldots, s_k)$. Predicate that is *True* if $(s_0, \ldots, s_k)$ is a solution path for s_0 in G.
constraints$(s_0, \ldots, s_k)$. Predicate that is *True* if $(s_0, \ldots, s_k)$ satisfies solution constraints.

Output: A solution base b representing a solution path or the symbol *Fail*.

```
Generic-OR($s$, successors, $\star$, constraints)

  0.  IF $\star((s))$ THEN IF constraints$((s))$ THEN RETURN($(s)$);   // Check $(s)$.
  1.  add($(s)$, OPEN);   // Store sol. base $(s)$ on OPEN to wait for expansion.
  2.  LOOP
  3.    IF (OPEN $==\emptyset$) THEN RETURN(Fail);
  4.    $b$ = choose(OPEN);   // Choose a solution base $b$ from OPEN.
        remove($b$, OPEN);   // Remove $b$ from OPEN.
  5.    $s_0$ = last_state($b$);   // Determine last state in $b$.
        FOREACH $s'$ IN successors($s_0$) DO   // Expand $s_0$.
          $b'$ = append(copy($b$), $s'$);   // Extend solution base $b$.
          IF $\star(b')$ THEN   // Check whether base $b'$ is a solution path.
            IF constraints($b'$) THEN RETURN($b'$);   // Check sol. constraints.
          add($b'$, OPEN);   // Store $b'$ on OPEN to wait for extension.
        ENDDO
  6.  ENDLOOP
```

Remarks:

- Algorithm `Generic-OR` takes a state-oriented point of view. We directly work with states and paths that are lists of states. All following algorithms will take a node-oriented point of view.
- The input parameters (here and in all following algorithms) are problem-specific. Non-problem-specific functions such as *add*$(.)$, *depth*$(.)$, *expand*$(.)$, *pop*$(.)$, *push*$(.)$, *top*$(.)$, etc. are used directly ($\approx$ "axiomatically") in the pseudocode.
- In constraint satisfaction problems, a solution path is constrained in that certain conditions must be satisfied. A predicate *constraints*$(.)$ can provide this check.

Remarks:

- Generic-OR search maintains and manipulates solution bases which are stored as sequences of states.
- The last state of a solution base represents a remaining problem which still is to be solved. So, a solution base represents some open problem. Therefore, the solution base storage is called OPEN list.
- For an OPEN list some appropriate data structure can be used.
- Function *append*$($*copy*$(b), s')$ returns a representation of the extended solution base. For simplicity we assume that b is copied when calling *append*$($*copy*$(b), s')$. Then, each solution base resulting from an expansion step is represented separately, although they share initial subsequences.
- Information on the explored part of the search space graph is stored within the solution bases. States are contained in solution bases and edges are given by pairs of neighboring states (occurring in solution bases).

  In fact, after t expansion steps, the solution bases in OPEN store parts of the explored part H_t of the search space graph G by storing maximal paths in H_t, that still have to be considered (i.e., paths from s to a terminal node in H_t).
- For algorithm Generic-OR it is not important that function *successors*$(.)$ returns unique structures for each successor state of an expanded state. There is no need for a unique state representation as paths information is stored explicitly by separate sequences (lists).

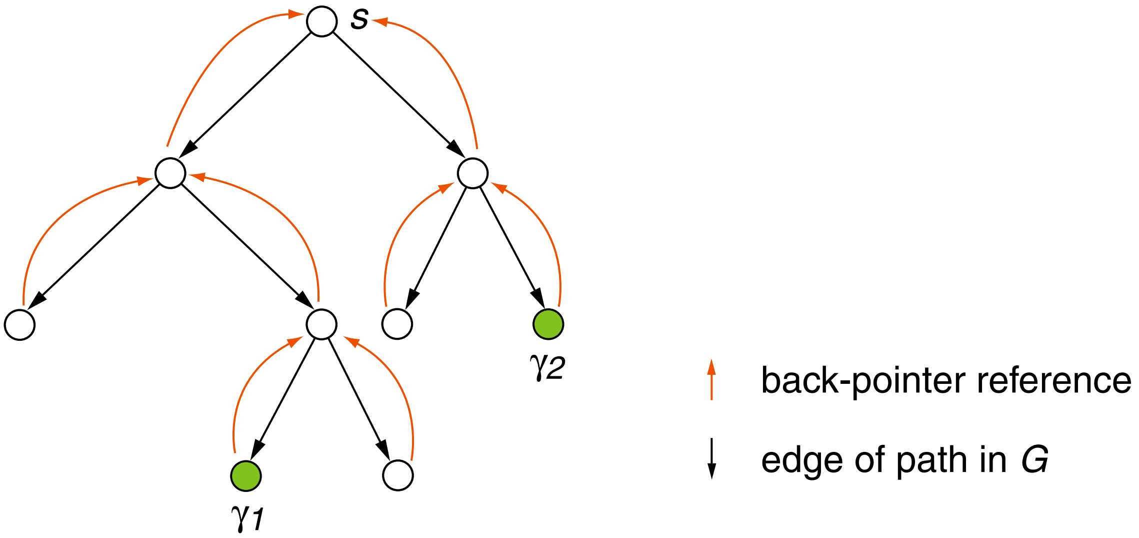
State Space Search

Efficient Storage of Solution Bases

```
5.    FOREACH $s'$ IN $successors(s_0)$ DO    // Expand $s_0$.
        $b' = append(copy(b), s')$;    // Extend solution base $b$.
        ...
```

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

➜ Back-pointer: a reference at each newly generated node, directting to its parent.



➜ A solution base is represented by a node defining a back-pointer path.

Remarks:

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

  If an algorithm wants to store more than one back-pointer per node, multiple solution bases can be represented by a node.
- Within an algorithm, we assume that a node is discarded if the node was removed from the algorithm's data structures and if the node is not accessible from some other node in the data structures. (Garbage Collection)

  Nevertheless, we sometimes mention removing a node explicitly.

State Space Search

Efficient Storage of Solution Bases (continued)

- Algorithms that only store or discard nodes but change back-pointers in a very restrictive way once they are set, store a tree-like graph at any point in time.
- The back-pointer structure represents a tree unfolding of some part of the explored part of G.

State Space Search

Efficient Storage of Solution Bases (continued)

- ❑ Algorithms that only store or discard nodes but change back-pointers in a very restrictive way once they are set, store a tree-like graph at any point in time.
- ❑ The back-pointer structure represents a tree unfolding of some part of the explored part of G.

Definition 8 (Traversal Tree, Backpointer Path)

Let $\mathcal{A}$ be an algorithm working on an implicitly defined state-space graph G.
Let $\mathcal{A}$ store with each generated node instance a back-pointer to its parent node (i.e., the node instance that was expanded). Hereby, a graph G_t is defined at each point t in time with state instances from G and edges reverse to the back-pointers.

The graph G_t maintained by $\mathcal{A}$ is a tree rooted in s, we call G_t a *traversal tree*.

For a node n, the reversed sequence of back-pointer-reachable nodes $(s, \ldots, n)$ defines a path in G_t, the so-called *back-pointer path* for n, denoted as *PP*$_{s-n,t}$.

Path *PP*$_{s-n,t}$ corresponds to a path in G (instances of the same states in G_t unified).

G_t is the tree unfolding of (a part of) the explored part H_t of G.

Remarks:

- Traversal trees and back-pointer structures change over time: traversal trees grow with each node instance that is integrated into the back-pointer structure. But usually traversal trees are considered at fixed points in time. Therefore, if the context allows, we omit the index t for the time (in between node expansions: after the last expansion and corresponding processing was finished and before the next expansion starts).

- In general, traversal trees are no subgraphs of the search space graph G, since G_t – being a tree of paths in G – can contain multiple instances of states in G. If an algorithm performs [path discarding], i.e., if the algorithm discards node instances representing a state that is already referenced by some other node in the back-pointer structure, then the traversal tree G_t will represent cycle-free paths in G starting in s and therefore G_t is a subgraph of G.

- A traversal tree is defined by the nodes stored so far and the edges inverse to the current back-pointer setting.

- Pointer-paths are those paths in G that have been found so far by algorithm $\mathcal{A}$.

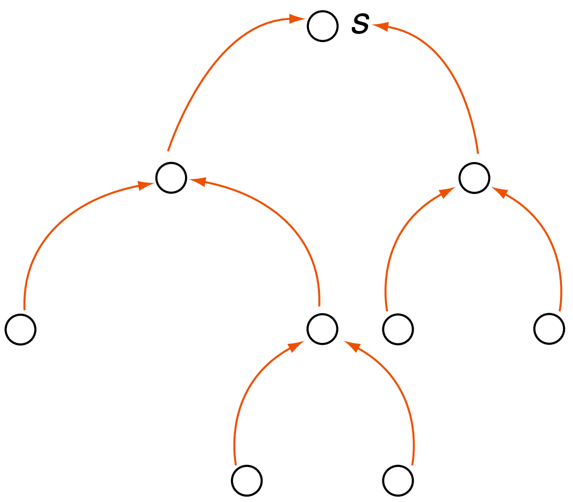
- Algorithms may remove parts of a back-pointer structure, if a part of G was completely searched without finding a solution.

  Therefore, the back-pointer-structure maintained by an algorithm at some point in time t might not correspond to the explored part H_t of G.

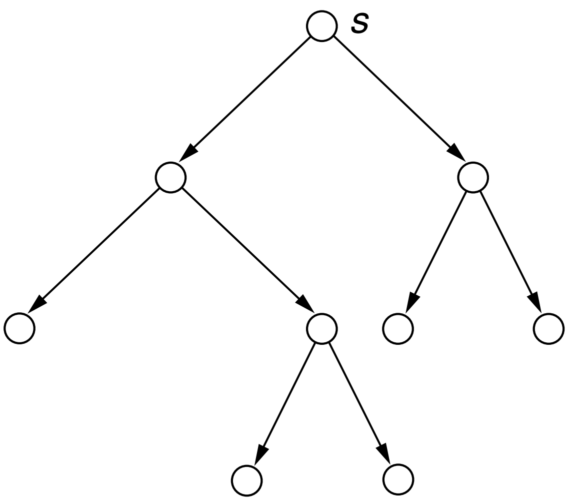
State Space Search

Efficient Storage of Solution Bases (continued)

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



Information maintained by algorithm *A*:
back-pointer structure
= nodes for states in *G* + back-pointers



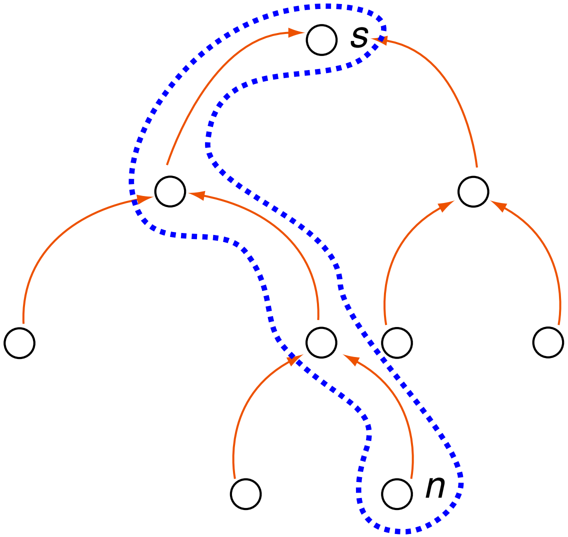
Traversal Tree

State Space Search

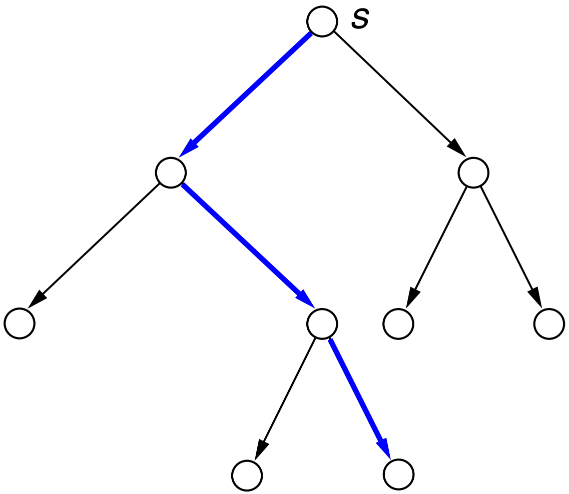
Efficient Storage of Solution Bases (continued)

- A back-pointer path is a solution base in G which can be uniquely identified by a node n in the back-pointer structure.

- Using the back-pointer concept, algorithm seem to use single nodes instead of solution bases. Even returning a solution can be done by returning the goal node found (node representing a foal state).



Path defined by node *n*



Back-pointer Path

State Space Search

Schema for Search Algorithms

... from a graph-oriented perspective inspired by the back-pointer structure
... using either node expansion or node generation as basic step:

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

Usage:

- Search algorithms maintain sets of nodes.
- Initially, only the start node s is available.
- Search algorithms can make use of graph nodes stored in two lists OPEN and CLOSED and of graph edges stored in form of back-pointers.
- Solution bases currently maintained are defined by nodes in OPEN.

Remarks:

- ❑ This schema also works for AND/OR-graph search. Instead of searching for solution paths we will then consider solution graphs in step 2(c).
- ❑ The search space graph, which is defined by the states and operators of a problem domain, is called *underlying (search space) graph*. An underlying graph is usually denoted by G. The underlying graph is not completely accessible to search algorithms.
- ❑ Graph search starts from a single node s and uses node expansion or node generation as a basic step.

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

State Space Search

Searching a Search Space Graph

At any time of a search, the nodes (for states) of a search space graph G can be divided into the following four sets according to their status:

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

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

State Space Search

Searching a Search Space Graph

At any time of a search, the nodes (for states) of a search space graph G can be divided into the following four sets according to their status:

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

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

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

State Space Search

Searching a Search Space Graph

At any time of a search, the nodes (for states) of a search space graph G can be divided into the following four sets according to their status:

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

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

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

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

Remarks:

- The classification of nodes can be seen as a classification of states in G. However, we must distinguish in which context a state was considered. Therefore, the successors of a state can be considered more than once.
- The situation characterized by Property 1 (“not generated”) is indistinguishable from situations where a node had already been generated and was discarded later on.
- The node sets that are induced by the Properties 2, 3, and 4, the generated nodes, the explored nodes, and the expanded nodes, form a subset-hierarchy: only generated nodes can be explored, only explored nodes can be expanded.

State Space Search

Algorithm: Basic-OR (Compare Generic-OR, Basic-DFS, Basic-BFS.)

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

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

```
Basic-OR($s$, successors, $\star$, constraints)   // A variant of Generic-OR.

  0.  $s$.parent = null;
      IF $\star(s)$ THEN IF constraints$(s)$ THEN RETURN$(s)$;  // Check path repr. by $s$.
  1.  add$(s$, OPEN$)$;   // Store $s$ on OPEN to wait for expansion.
  2.  LOOP
  3.    IF (OPEN $== \emptyset$) THEN RETURN(Fail);
  4.    $n$ = choose(OPEN);   // Choose a solution base represented by $n$.
        remove$(n$, OPEN$)$; add$(n$, CLOSED$)$;   // CLOSED for garbage collection.
  5.    FOREACH $n'$ IN successors$(n)$ DO   // Expand $n$.
          $n'$.parent = $n$;   // Extend solution base represented by $n$.
          IF $\star(n')$ THEN   // Check whether $n'$ represents a solution path.
            IF constraints$(n')$ THEN RETURN$(n')$;   // Check sol. constraints.
          add$(n'$, OPEN$)$;   // Store $n'$ on OPEN to wait for expansion.
        ENDDO
  6.  ENDLOOP
```

Remarks:

- Algorithm Basic-OR takes a graph-oriented perspective:

  Basic-OR maintains a back-pointer structure by processing nodes and back-pointers.

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

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

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

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

- If storing instances of nodes with the same state reference should be avoided, no two paths leading to the same node can be allowed in G, especially no cycles. In this case G is a tree with root s.

- The start node s is usually not a goal node. If this is known in advance, the second line of the algorithm can be omitted:

  ```
      IF $\star(s)$ THEN IF constraints($s$) THEN RETURN($s$);
  ```

  In future, this line will be omitted if there is not enough space. Additionally, the check for constraints will sometimes not be mentioned explicitly.

State Space Search

Important Properties of Search Algorithms

Definition 9 (Termination, Soundness, Completeness)

Let $\mathcal{A}$ be an algorithm searching a state-space graph G for a solution path for a given state s.

1. $\mathcal{A}$ terminates if $\mathcal{A}$ returns a result *Fail* or a solution within a finite number of steps.
2. $\mathcal{A}$ is sound if – in case $\mathcal{A}$ terminates and returns a path – then this path is in fact a solution path for s in G.
3. (For constraint satisfaction problems) $\mathcal{A}$ is complete if $\mathcal{A}$ terminates returning a solution (satisfying the constraints) if a solution (satisfying the constraints) exists.

➜ The algorithms presented here are sound by construction:
A solution path was checked before returning it.

Remarks:

- Usually, soundness and completeness are used in the context of algorithms $\mathcal{A}$ for decision problems:

  $\mathcal{A}$ is sound if – in case $\mathcal{A}$ returns "yes" – the true answer is "yes";
  $\mathcal{A}$ is complete if – in case "yes" is the true answer – $\mathcal{A}$ returns "yes".

State Space Search

Lemma 10 (Termination of Basic-OR for Finite Graphs without Cycles)

Basic-OR terminates for finite graphs G without cycles that have the *Prop*$_0(G)$ properties.

Proof (sketch)

1. Since G has *Prop*$_0(G)$ and G is finite and cycle-free, the number of finite paths starting in s is finite.
2. At each point in time (whenever Basic-OR is in step 2) before Basic-OR terminates, the nodes in OPEN and CLOSED represent pairwise different finite paths in G starting in s.
3. In each run through the main loop, one node is moved from OPEN to CLOSED.
4. Since the number of paths G starting in s is finite, Basic-OR must terminate.

Lemma 11 (Completeness of Basic-OR for Finite Graphs without Cycles)

Basic-OR is complete for finite graphs G without cycles that have the *Prop*$_0(G)$ properties.

Proof (sketch)

1. Since G has *Prop*$_0(G)$ and G is finite and cycle-free, Basic-OR terminates. (Lemma 10)
2. Let G contain a solution path $P_{s-\gamma}$ for s.
3. At any point in time before Basic-OR terminates, there is an OPEN node in $P_{s-\gamma}$ (not necessarily representing an initial part of $P_{s-\gamma}$).
4. As long as there is such an OPEN node available, Basic-OR will not terminate with *Fail*.
5. Latest when an OPEN node representing γ has to be selected for expansion, Basis-OR terminates returning a solution.

State Space Search

Searching a Search Space Graph (continued)

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

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

1. Blind or uninformed.

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

2. Informed, guided, or directed.

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

State Space Search

Uninformed Systematic Search

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

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

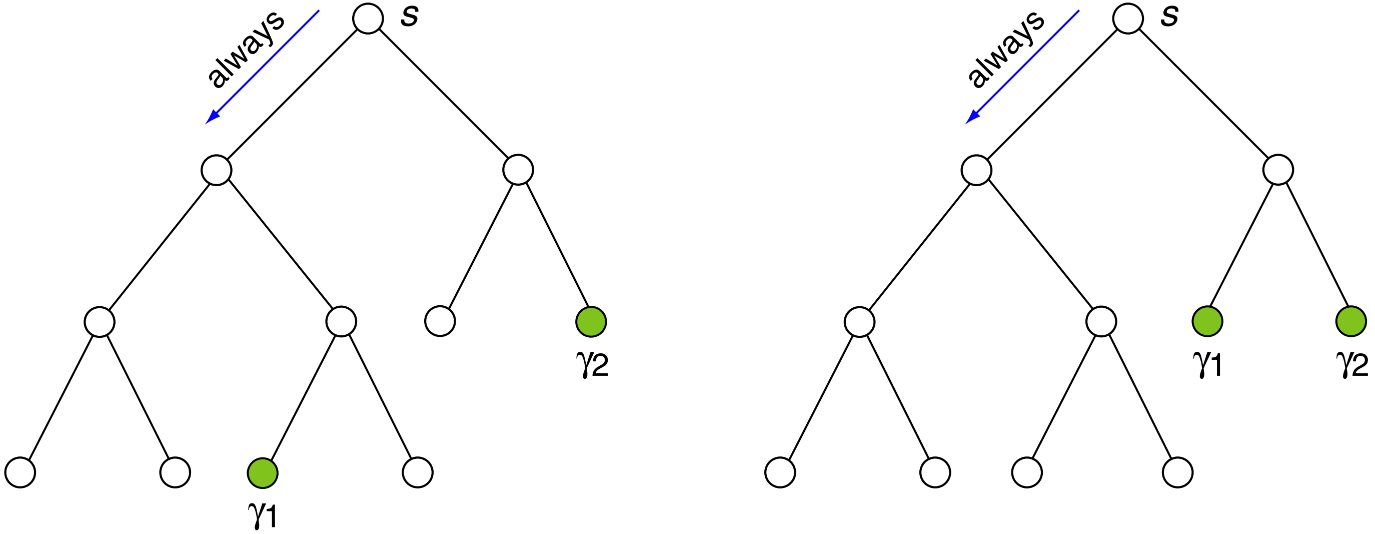
State Space Search

Uninformed Systematic Search

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

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

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



Remarks:

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

State Space Search

Basic-OR Search for Optimization

Setting:

- ❑ The search space graph contains several solution paths.
- ❑ A cost function assigns cost values to solution paths.

Task:

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

Approach:

- ❑ Pursue some node selection strategy.
- ❑ Store solution path with cheapest cost known so far.

Early Pruning:

- ❑ Prerequisites
  - The cost function assigns cost values to solution bases, not only solution paths.
  - The cost of a solution base is an optimistic estimate of the cheapest solution cost that can be achieved by completing the solution base.

➜ Continue search only until the costs of solution bases exceed the currently optimum cost.

State Space Search

```
Basic-Opt-OR($s$, $successors$, $\star$, $cost$)    // Basic-OR for optimization problems.
  1.  $c_{opt} = \infty$;  $s.parent = null$;  $add(s, \text{OPEN})$;    // Assuming $s$ is no goal node.
  2.  LOOP
  3.    IF ($\text{OPEN} == \emptyset$)    // No further solution bases to consider.
        THEN
          IF ($c_{opt} < \infty$) THEN RETURN($n_{opt}$);    // Solution path found.
          RETURN($Fail$);    // No solution path found.
        ENDIF
  4.    $n = choose(\text{OPEN})$;    $remove(n, \text{OPEN})$;  $add(n, \text{CLOSED})$;
  5.    FOREACH $n'$ IN $successors(n)$ DO    // Expand $n$.
          $n'.parent = n$;    // Extend solution base represented by $n$.
          IF $\star(n')$    // Solution base $n'$ is evaluated.
          THEN
            IF ($cost(n') < c_{opt}$)
            THEN
              $c_{opt} = cost(n')$;  $n_{opt} = n'$;    // $n'$ better solution.
            ENDIF
            $add(n', \text{CLOSED})$;    // Goal nodes are terminal nodes.
          ELSE
            $add(n', \text{OPEN})$;
          ENDIF;
        ENDDO
  6.  ENDLOOP
```

Remarks:

- If we have no means to estimate cost values for solution bases or if the cost estimation is not optimistic, the nesting of the conditional statements in the **FOREACH** loop has to be changed. Then we have no early pruning. The algorithm stops only if the set of solution bases in OPEN is exhausted.

- Depending on the problem, the Basic-Opt-OR search may not terminate, e.g., for infinite graphs with no optimum solution path or for graphs with an infinite number of solution bases with costs lower than the optimum solution cost.

- The optimistic cost estimation is crucial for the correctness of Basic-Opt-OR search: If the cheapest solution cost that can be achieved by completing the solution base is overestimated we might miss an optimum cost solution path.

State Space Search

Optimization Problem Example

Problem Setting and Task:

- Given a matrix of non-negative numbers.
- Determine a column with minimum column sum in the matrix.

State Space Search

Optimization Problem Example

Problem Setting and Task:

- Given a matrix of non-negative numbers.
- Determine a column with minimum column sum in the matrix.

Modeling as State Space Search Problem:

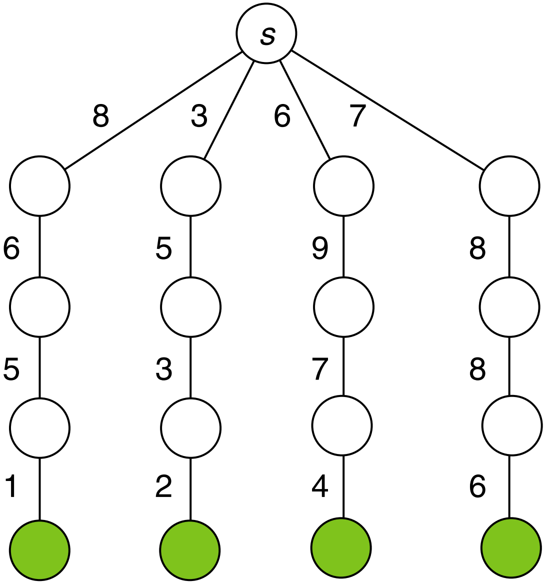
- Cells of the matrix correspond to nodes (i.e., states) in a state space graph.
- Cells connect by an edge to a cell in the next row within the same column.
- Cells in the last row are goal nodes.
- A start node s connects to the nodes for cells in the first row.
- The numbers of the matrix cells are modeled as weights of the incoming edges.
- Cost of a solution base / solution path is sum of the values of the matrix cells traversed, i.e., sum of the edge cost values in the back-pointer path.

State Space Search

Optimization Problem Example (continued)

Modeling as State Space Search Problem

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

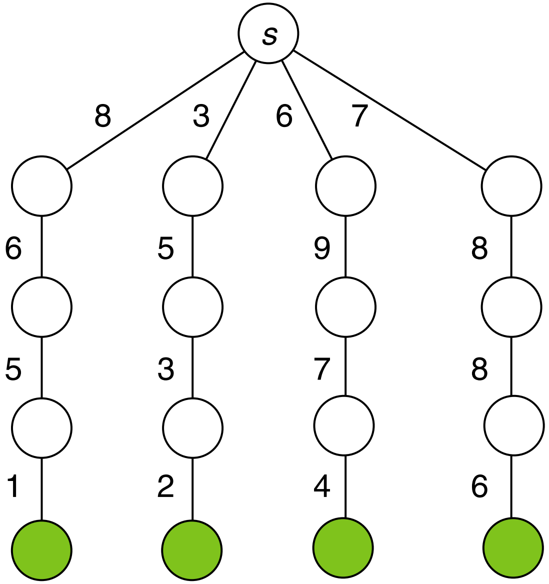


State Space Search

Optimization Problem Example (continued)

Modeling as State Space Search Problem

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



Q What heuristic can be used to assign cost values to solution bases?

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