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

Systematic Search

Types of Problems

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

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

In particular, a desired solution is expected to be in S.

Systematic Search

Types of Problems

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

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

In particular, a desired solution is expected to be in S.

Distinguish two problem types:

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

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

Remarks:

- Constraints can characterize required properties of a solution candidate:
  - 8-queens: board configuration without threat on any queen,
  - 8-puzzle: move sequence leads to target board configuration,
  - TSP: travel tour forms a path in the TSP graph.
- 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 *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

Generic Framework for Modeling Search Problems

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

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

Observation:

- Transitions (s_0, s_1) are entirely local.

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

Remarks:

- ❑ 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.

Systematic Search

Modeling Problem Solving as Reachability Problem for STS

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

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

Systematic Search

Modeling Problem Solving as Reachability Problem for STS

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

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

Systematic Search

Modeling Problem Solving as Reachability Problem for STS

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

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

Remarks:

- The simplest constraint for a solution is "no constraint" (i.e., any sequence of transitions from the start node to a goal node is acceptable).
- Some sources define state transition systems by states and transitions, only. The difference is whether we see s and F as part of the setting or as part of the reachability question.
- In general, the modeling of real world problems as state transition systems requires discretization of continuous transitions and simplification of dependencies.
- A state description can also be considered as 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

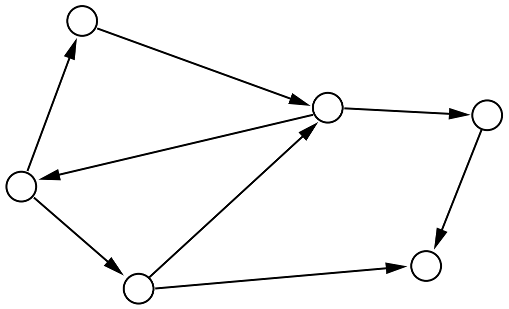
Definition 2 (Directed Graph)

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

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

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



Graph Theory Basics

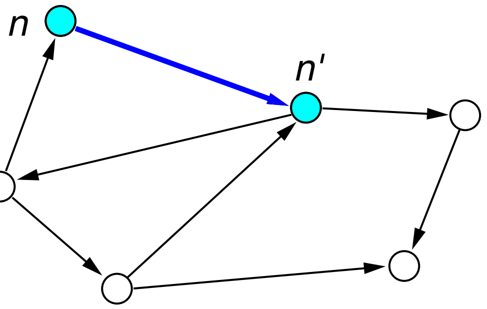
Definition 2 (Directed Graph)

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

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

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



Remarks:

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

Graph Theory Basics

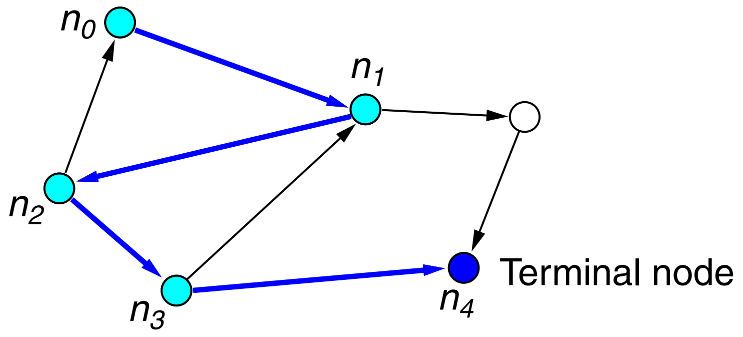
Definition 3 (Path, Node Types, Outdegree)

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

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

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

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



Graph Theory Basics

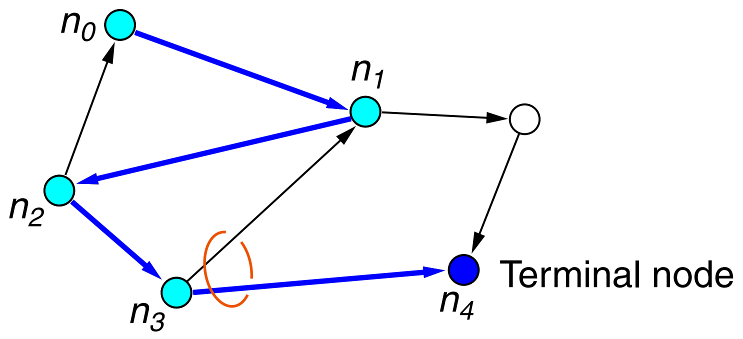
Definition 3 (Path, Node Types, Outdegree)

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

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

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

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



Remarks:

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

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

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

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

Graph Theory Basics

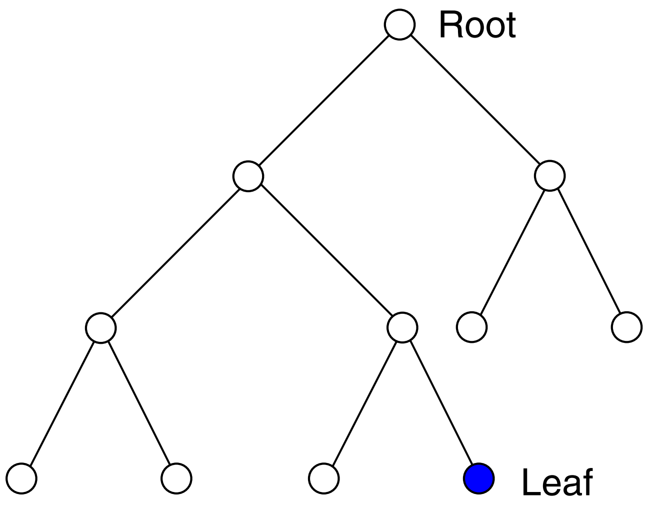
Definition 4 (Directed Tree, Uniform Tree)

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 node in V.

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

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

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



Remarks:

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

State Space Search

Modeling Problem Solving as Path-Seeking Problem

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

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

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

State Space Search

Modeling Problem Solving as Path-Seeking 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:

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

Modeling:

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

Remarks:

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

State Space Search

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

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

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

A search for a solution 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 node s in G.

- Solution Bases

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

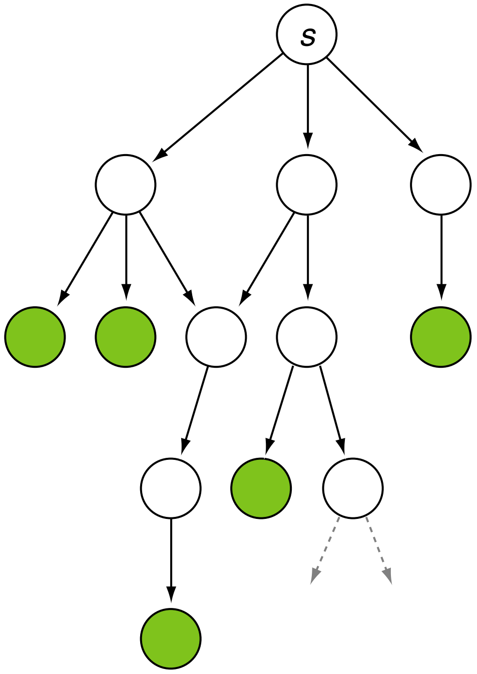
Remarks:

- Obviously, all solution paths contained in G are also solution bases.
- The tip node of a solution base represents the so-called *remaining problem*.
- When modeling problem solving as a path-seeking problem in a state-space graph G, the search space consists of all finite paths in G starting in s.
- While solution paths are often discussed with respect to the underlying state-space graph, solution bases are only considered within a finite subgraph of the state-space graph which was explored so far by an algorithm.
- If P is a solution base for a node n in G and if n' is some node in P, then the subpath P' of P starting in n' and ending in the tip node of P is a solution base for n'. This subpath P' is sometimes called the *solution base in P induced by n'*.

State Space Search

Illustration of Solution Paths and Solution Bases

State-space graph:

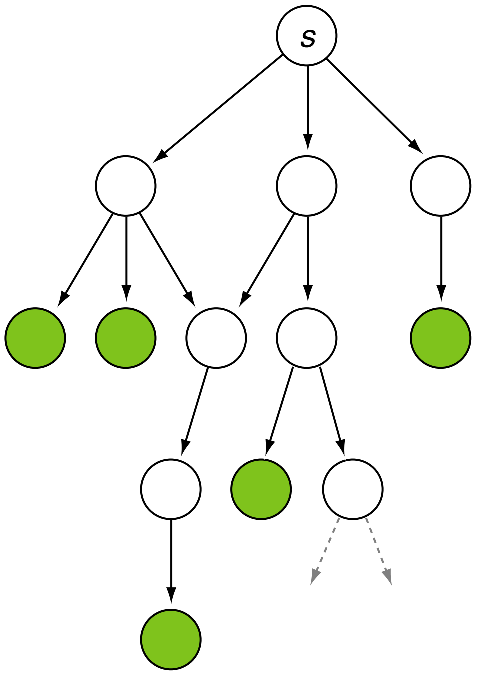


🟢 Goal node
(solved rest problem)

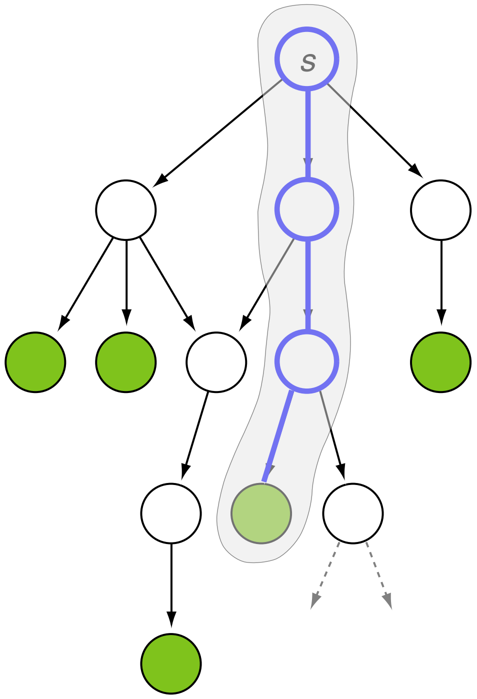
State Space Search

Illustration of Solution Paths and Solution Bases

State-space graph:



Solution path for node s:

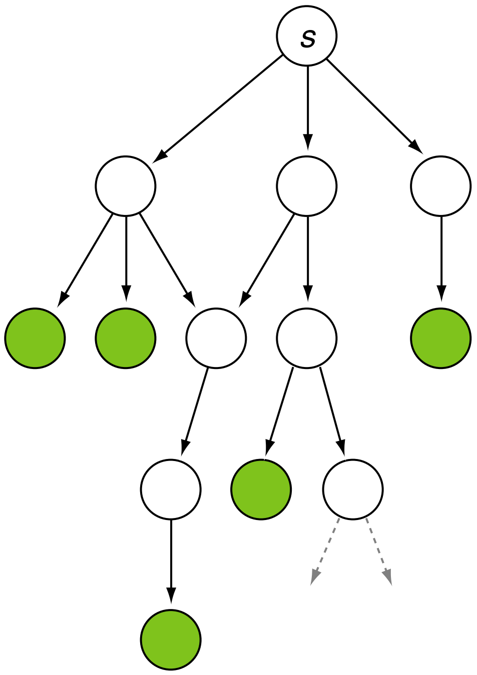


🟢 Goal node
(solved rest problem)

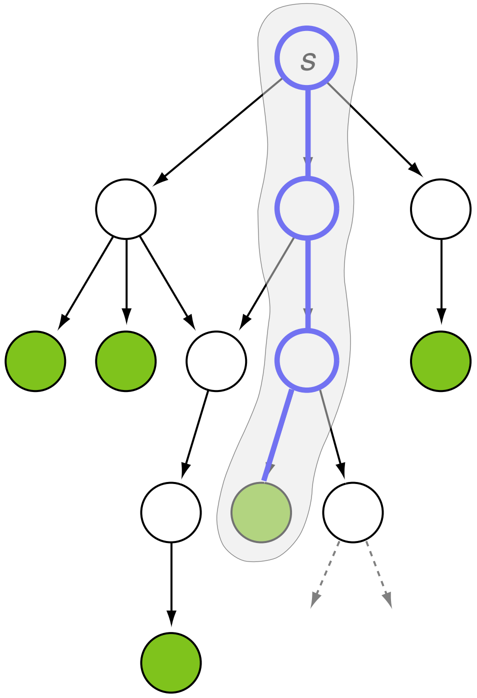
State Space Search

Illustration of Solution Paths and Solution Bases

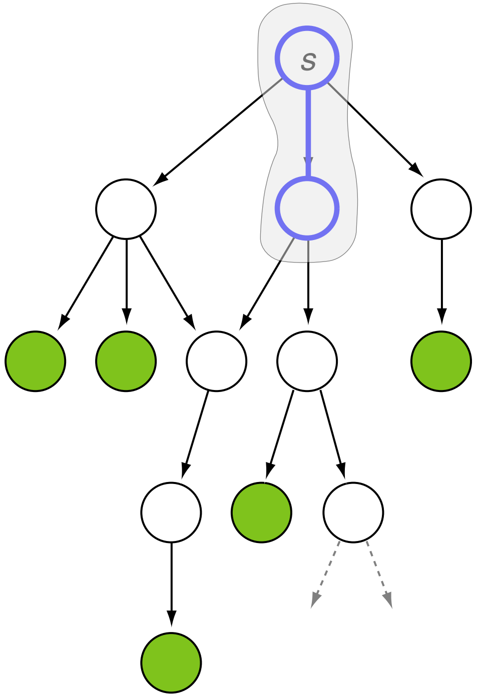
State-space graph:



Solution path for node *s*:



Solution base for *s*:



🟢 Goal node
(solved rest problem)

State Space Search

Interpretation of Solution Bases

Candidate-oriented perspective

- A solution base represents a set of solution candidates (i.e., the set of all finite paths in G that have the solution base as an initial part):
  $(s, \ldots, n)$ represents $\{(s, \ldots, n, \ldots, n') \mid (s, \ldots, n, \ldots, n')$ is path in $G\}$
- Subsets of solution candidates can be split (and discarded) until a contained solution path is easily identified.

Problem-oriented perspective

- A state represents a problem. Solution steps (transitions in the STS) lead to states that (hopefully) represent simpler problems. A solution base represents an unfinished sequence of solution steps.
- Solution candidates are constructed from scratch as a sequence of solution steps until the remaining problem is trivial (solved).

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 nodes in V and all edges in E explicitly.
- ➜ Algorithms using an explicit graph representation can handle finite graphs only.

Implicit Graph (Representation)

- ❑ A graph G is specified by listing a set S of start nodes and giving a function *successors*(n) returning the set of direct successors of node n in G.
- ❑ A graph G is specified by listing a set S of start nodes and giving a function *next_successor*(n) returning an unseen direct successor of node n in G.
- ➜ Algorithms using an implicit graph representation can handle graphs with finite set S and computable function *successors*(n) (resp. computable *next_successor*(n) and *expanded*(n)).

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

Remarks:

- The function *next_successor*(n) generates the direct successors of n in G one by one. In order to simplify notation and algorithms, we assume that a function *next_successor*(n) itself stores which successors have been generated so far.
- An explicit representation of a graph G can be determined from its implicit representation in the following way.

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

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

State Space Search

Definition 7 (Node Generation)

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

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

Definition 8 (Node Expansion)

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

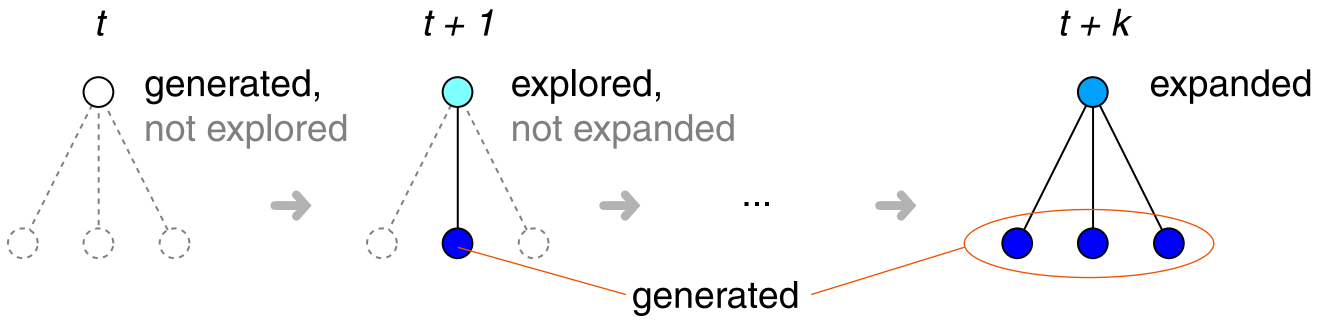
Applying the function *successors*(n) and thereby generating all direct successors of a node in one (time) step is called *node expansion*.

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

➜ Except for 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:

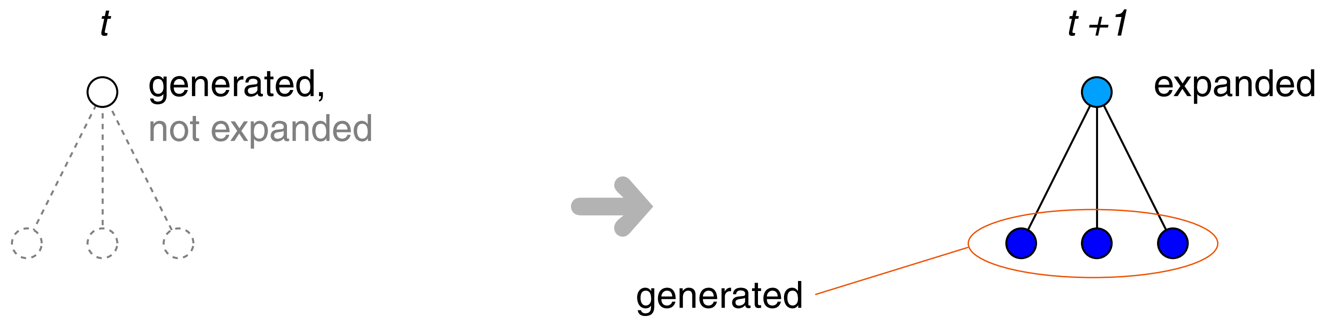
- Left: The status of the newly generated successor is called *generated*.
- Middle: The status of the parent node is called *explored*.
- Right: The status of the parent node is called *expanded* if all of its direct successors are generated.

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: An algorithm has to store fewer nodes.

State Space Search

Node Expansion as a Basic Step



Node status:

- Left: The status of the newly generated successor is called *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 time step.

Advantage: *successors*(n) is easier to implement.
Disadvantage: An algorithm 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) need to be marked.
- Knowledge about operator application can be maintained by appending a label to the encoding of a state.

State Space Search

Explicit Parts of State-Space Subgraphs

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

- What A could know about G after t node expansions:
  Explored (sub)graph H_t of G after t node expansions.

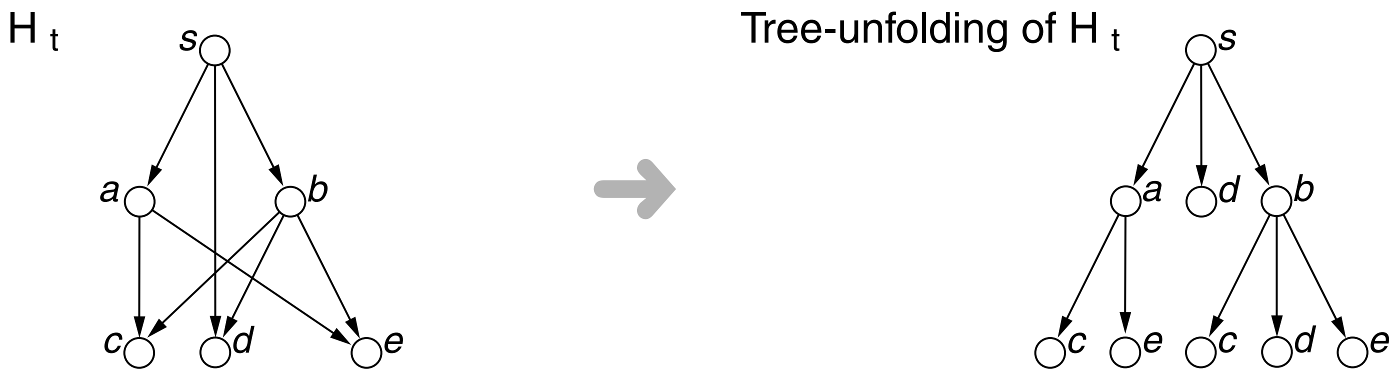
  Inductive Definition: $H_0 = (\{s\}, \{\})$

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

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

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

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



State Space Search

Requirements for an Algorithmization of State-Space Search

Properties of Search Space Graphs

1. G is a state-space graph (directed OR graph).
2. G is implicitly defined by
   (a) a single start node s and
   (b) a function *successors*(n) or *next_successor*(n) returning successors of a node.

   Computing successors always returns new clones of nodes.
3. G is locally finite.
4. A set Γ of goal nodes for G is given; in general Γ will not be a singleton set.
5. G has a function $\star(.)$ returning true if a solution base is a solution path.

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

Remarks (continued) :

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

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 a successor node and store it as a new candidate.
   (d) Determine whether a solution graph resp. solution path 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.

State Space Search [Basic_OR_Search]

Algorithm: Generic_OR_Search

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

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

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

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

Remarks:

- Generic_OR_Search maintains and manipulates solution bases which are stored as sequences of nodes.
- The last node of a solution base represents a remaining problem which still is to be solved. So, a solution base represents some open problem.

  Therefore, the solution base storage is called OPEN list.
- For an OPEN list some appropriate data structure can be used.
- Function *append*(b, n') returns a representation of the extended solution base. For that purpose, b might be copied. Then, each solution base is represented separately, although they often share initial subsequences.
- Information on the explored part of the search space graph is stored within the solution bases. Nodes are contained in solution bases and edges are stored as pairs of nodes (occurring in solution bases).
  In fact, after t node expansions, the solution bases store parts of the explicit part H_t of the search base graph by storing all maximal paths in H_t, that still have to be considered (i.e., all paths from s to a terminal node in H_t).
- For algorithm Generic_OR_Search it is not important that function *successors*(n) returns new clones for each successor node of an expanded node. There is no need for a unique node representing a state as paths will be stored as sequences and not as sets.

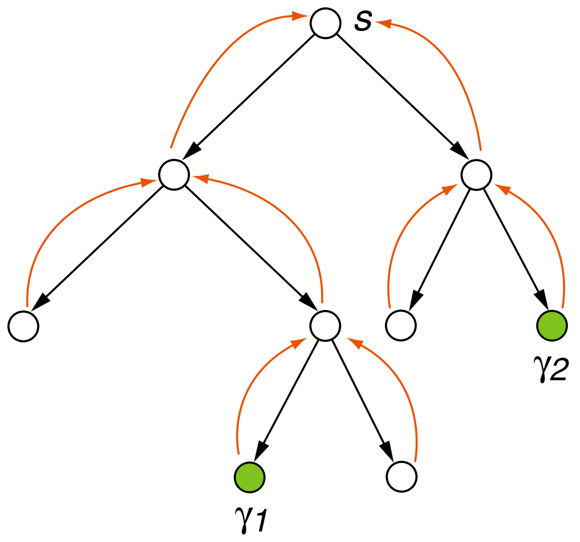
State Space Search

Efficient Storage of Solution Bases

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

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

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



➜ A solution base is represented by a backpointer path.

Remarks:

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

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

State Space Search

Efficient Storage of Solution Bases (continued)

- Algorithms that only store or discard nodes but change backpointers in a very restrictive way once they are set, store a tree-like graph any point in time.
- If the edges (backpointers) were reversed, it would be a tree unfolding of some part of the explored part of G.

State Space Search

Efficient Storage of Solution Bases (continued)

- Algorithms that only store or discard nodes but change backpointers in a very restrictive way once they are set, store a tree-like graph any point in time.
- If the edges (backpointers) were reversed, it would be a tree unfolding of some part of the explored part of G.

Definition 9 (Traversal Tree, Backpointer Path)

Let A be an algorithm working on G using its implicit definition. Let A store with each generated node instance a backpointer to its parent node (i.e., the node instance that was expanded). Hereby, a graph G_A is defined with node instances from G and edges reverse to the backpointers.

The graph G_A maintained by A is a tree rooted in s, we call G_A a *traversal tree*.

For a path $(n, \ldots, s)$ defined by the backpointers of the nodes, the reversed sequence of nodes $(s, \ldots, n)$ defines a s path in G_A, the so-called *backpointer path* for n, denoted as PP_{s-n}.

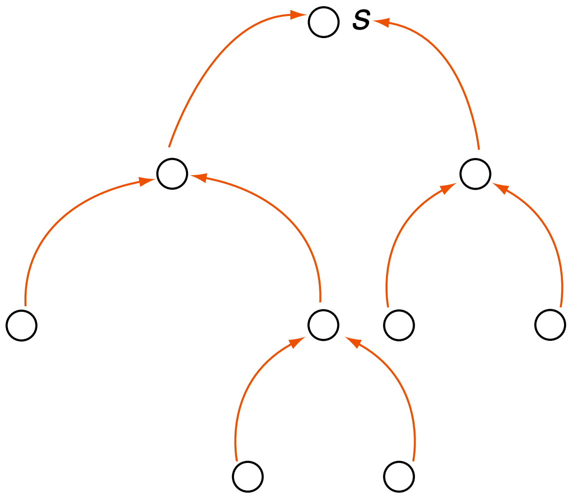
Remarks:

- A traversal tree is defined by explored nodes along with the edges inverse to the current setting of backpointers.
- Pointer-paths are those paths in G that have been found so far by algorithm A.
- Traversal trees grow with each node expansion. To be precise, we have to consider traversal tress at specific points in time (e.g., at the beginning of the main loop of an algorithm).
- In general, traversal trees are no subgraphs of the search space graph G, since G_A can contain multiple instances of nodes in G. If an algorithm performs an occurrence check and discards multiple instances in some way, the traversal tree will represent cycle-free paths in G starting in s.

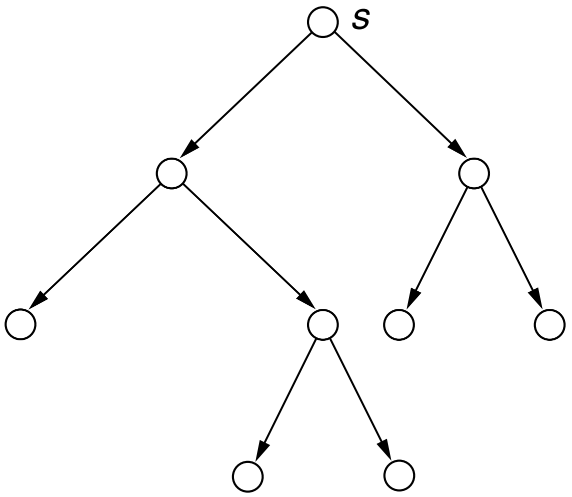
State Space Search

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:
nodes from G + backpointers

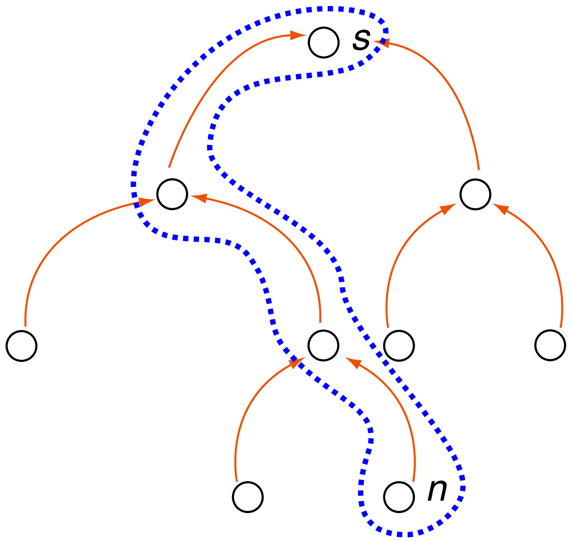


Traversal Tree

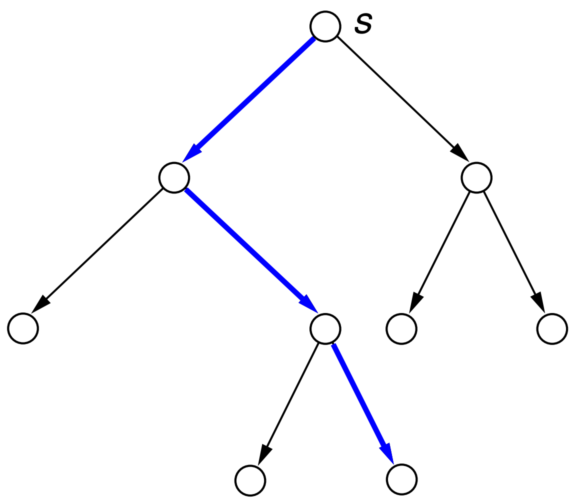
State Space Search

Efficient Storage of Solution Bases (continued)

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



Path defined by node *n*



Backpointer Path

State Space Search

Schema for Search Algorithms

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

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

Usage:

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

Remarks:

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

  During graph search, only the finite explored (sub)graph is accessible at a point in time. An explored subgraph is also denoted by G, except in cases where the underlying search space graph is addressed as well.
- An explored subgraph G is rooted at node s, the root node of the underlying search space graph. G is connected, more precisely, all nodes in G can be reached from s.
- In a narrower sense, 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 of a search space graph G can be divided into the following four sets:

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

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

State Space Search

Searching a Search Space Graph

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

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

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

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

State Space Search

Searching a Search Space Graph

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

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

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

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

Remarks:

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

State Space Search

Algorithm: Basic_OR_Search (Compare [Generic_OR_Search] [DFS] [BFS])

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

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

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

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

Remarks:

- Algorithm Basic_OR_Search takes a graph-oriented perspective:

  Basic_OR_Search maintains and manipulates nodes and backpointers.

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

- The node list CLOSED is kept for garbage collection purposes. Since nodes are shared in backpointer paths, a node can 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 backpointers of the nodes. A backpointer represents an edge with opposite direction.

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

- If storing multiple instances of nodes 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.

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.

[control strategy]

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

1. Blind or uninformed.

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

2. Informed, guided, or directed.

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

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)

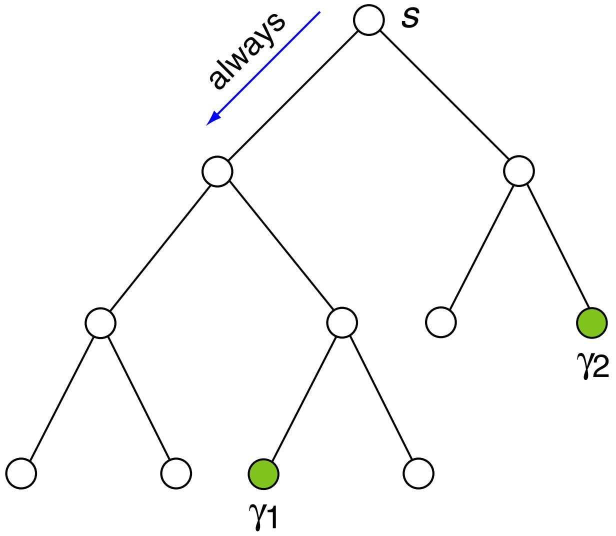
State Space Search

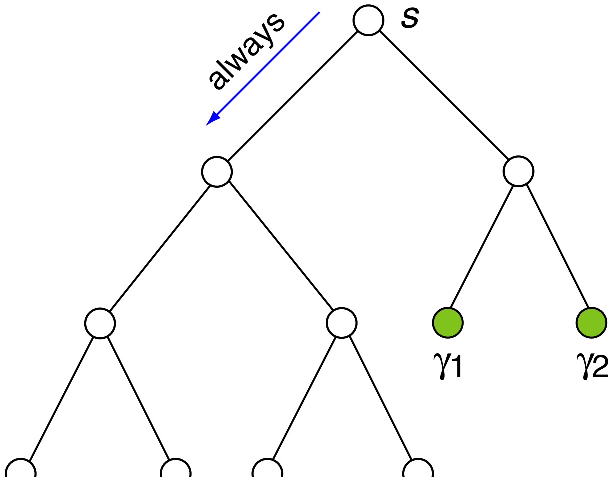
Uninformed Systematic Search

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 from 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): real-world problems cannot be tackled with uninformed search.

State Space Search

Basic_OR_Search for Optimization

Setting and Task:

- The search space graph contains several solution paths.
- A cost function assigns cost values to solution paths and solution bases.
- 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.
- Continue search only until the costs of solution bases exceed the currently optimum cost. Keyword: *Early Pruning*

Prerequisite:

- The cost of a solution base is an optimistic estimate of the cheapest solution cost that can be achieved by completing the solution base.

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:

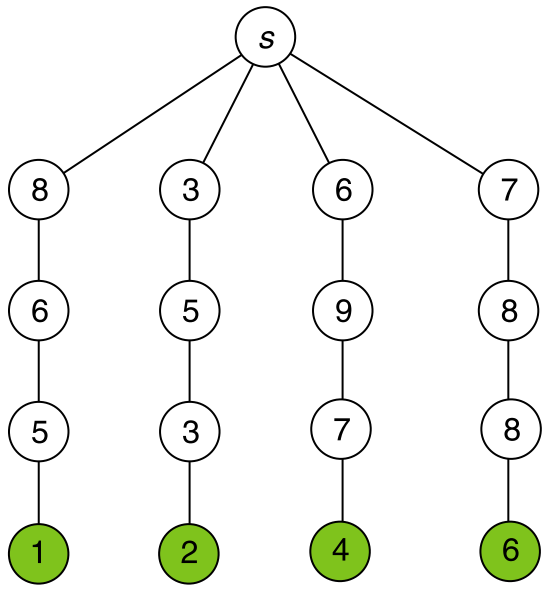
- Cells of the matrix correspond to nodes (i.e. states) in a state space graph.
- Cells connect by an edge to a cell in the next row within the same column.
- Cells in the last row are goal nodes.
- A start node s connects to the nodes for cells in the first row.
- Cost of a solution base / solution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

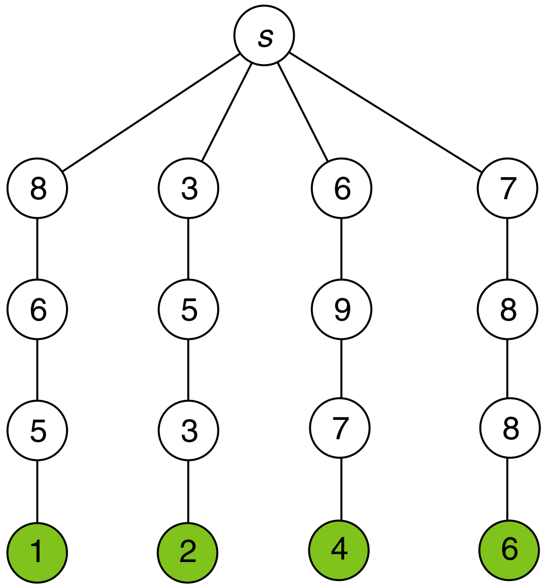


State Space Search

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



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