# Solving Problems by Searching

Artificial Intelligence I — CS475/505

## 1 Introduction

*Search* is an important part of our problem solving process. Practically, we search for a solution every time we try to solve a problem. Search is often needed when we do not have a step-by-step algorithm but we know what is a solution. Examples:

- *Traveling salesman problem*: Given $n$ cities, distance between every pair of two cities. A salesman needs to visit these cities, each at least one. Find for him a shortest route through the cities.
- *Knap-sack problem*: Given $n$ items, the weight and value of each item, and a knap-sack and its capacity. Find the most valuable way to pack the items into the knap-sack.
- *Navigation path*: Find a path connecting the two points on a map for a robot.
- *$n$-queen problem*: Given a $n \times n$ chess board. Find a placing of $n$ queens so that no pair of queens attack each other.
- *8-puzzle problem*: Given a $3 \times 3$ board with eight numbered tiles and a blank space. A tile adjacent to the blank space can slide into the space. The goal is to reach a specified goal state.
- *SLD derivation*: Given a goal $?g$, find a SLD derivation for $g$. (We did not discuss this example in the class, try to find out about it if you can!)

**Definition 1.1** *Search is an enumeration of a set of potential partial solutions to a problem so that they can be checked to see if they truly are solutions, or could lead to a solution.*

To carry out a search, we need:

- A definition of a potential solution.
- A method of generating the potential solutions (hopefully in a clever way).
- A way to check whether a potential solution is a solution.

**Example 1.1** *1. Search can be used to solve the n-queens problem with*

- *A* potential solution *in this problem is a $n \times n$-chess board with n-queens on it.*
- *Placing the queen one-by-one on the board is a method for generating the potential solutions (this is obviously not so clever!)*
- *If no two queens on the board attack each other, then the current potential solution is indeed a solution.*

2. *Search can be used to solve the 8-puzzle problem with*

   - *A* potential solution *in this problem is a possible configuration of the $3 \times 3$-board with 8 digits and an empty cell.*
   - *Exchanging the places of the empty cell and one of its neighbors allows us to generate potential solutions.*
   - *If the current configuration of the board satisfies the final configuration then it is a solution.*

3. *Search can be used to solve the knap-sack problem with*

   - *A* potential solution *in this problem is a subset of the available items.*
   - *Taking out one item or putting in one item allows us to generate potential solutions.*
   - *If the current subset (in the knap-sack) satisfying the desirable properties (total weitgh smaller than the allowed weight and total value maximal) then it is a solution.*

A search problem can be formally defined as follows.

**Definition 1.2** *A search problem is given by a tuple $\langle \Sigma, succ, S_0, G \rangle$ where*

- $\Sigma$ *is the set of* states*, which represents the set of* all *potential solutions.*
- $succ \subseteq \Sigma \times \Sigma$ *is a binary relation over the set of states, which denotes a relationship between the potential solutions. Intuitively, $(s, u) \in succ$ means that the potential solution $u$ can be constructed from $s$.*
- $S_0$ *is a set of states describing the set of* initial *states from which the search starts.*
- $G$ *is a set of states describing the set of* goal *states where the search can stop.*

**Note:** We can also add a cost function for the transition function between states as follows:

**Definition 1.3** *A search problem is given by a tuple $\langle \Sigma, succ, S_0, G \rangle$ where*

- $\Sigma$ *is the set of* states*, which represents the set of* all *potential solutions.*

- *succ* $\subseteq \Sigma \times \Sigma \times R^+$ *is a ternary relation over the set of states and the set of positive real values, which denotes a relationship between the potential solutions. Intuitively,* $(s, u, c) \in$ *succ means that the potential solution* $u$ *can be constructed from* $s$ *and* the cost for the transition from $s$ to $u$ is $c$.

- $S_0$ *is a set of states describing the set of* initial *states from which the search starts.*

- $G$ *is a set of states describing the set of* goal *states where the search can stop.*

As we can see from the example, it is possible that $S_0$ (resp. $G$) contains more than one states (Why is it so?). Although it is okay to view of problem solving as search problem, there is something we need to think before we can develop a program to solve (hopefully) *all* search problems:

- To develop a program that can solve problems using search, we need a way to specify the set of states in a clever way because the number of states can be extremely huge! For example, the knapsack problem with $n$ items can have $2^n$ possible states. To see how big it is, let $n = 100$; in this case, there are $2^{100}$ states which is much more than $10^{12}$–an estimation of ALL the stars in our universe!

- For the same reason, we will need to have a way to specify the relation *succ* since enumerating all pairs in *succ* will not work, for sure!

Before we get back to these issues, we will discuss a way to find solutions for a search problem.

# 2 Graph Searching

Graph is used to present general mechanism of searching. To solve a problem using search, we translate it into a graph searching problem and use the graph searching algorithms to solve it.

**Definition 2.1** *A graph consists of a set* $N$ *of* nodes *and a set* $A$ *of ordered pairs of nodes, called* arcs *(or* edges*).*

Two possible ways to represent a problem as a graph:

- *State-space graph*: each node represents a state of the world and an arc represents changing from one state to another.

- *Problem-space graph*: each node represents a problem to be solved and an arc represents alternate decomposition of the problems.

Example:

- *State-space graph*: finding path for robot – each node is a location. The state of the world is the location of the robot.
- *Problem-space graph*: SLD resolution – each node is a goal. Connection from one node to the other represents that the second one is obtained from the other through a SLD resolution. (**Note**: you might not know about SLD resolution yet; skip this one then.)

We will need the following terminologies:

Node $n_2$ is a **neighbor** (or a **successor**) of $n_1$ if there is an arc from $n_1$ to $n_2$. That is, if $\langle n_1, n_2 \rangle \in A$. An arc may be labeled.

A **path** is a sequence of nodes $\langle n_0, n_1, ..., n_k \rangle$ such that $\langle n_{i+1}, n_i \rangle \in A$.

A **cycle** is a nonempty path such that the end node is the same as the start node. A graph without cycle is called **directed acyclic graph** or DAG.

Given a set of start nodes and goal nodes, a solution is a path from a start node to a goal node.

The *forward branching factor* of a node is the number of arcs going out from the node, and he *backward branching factor* of a node is the number of arcs going into form the node.

# 3 A Generic Searching Algorithm

Given a graph, the set of start nodes, and the set of goal nodes. A path between a start node and a goal node is a solution. Searching algorithms provide us a way to find a solution.

**Idea:** Incrementally explore paths from start nodes. Maintaining a **frontier** or **fringe** of paths from the start nodes that have been explored.

The algorithm:

```
Input: a graph,
       a set of start nodes,
       Boolean procedure goal(n) that tests if n is a goal node.

frontier := {<s> : s is a start node};

while frontier is not empty:
       select and remove path <n0, . . . , nk> from frontier;
       if goal(nk)
            return <n0, . . . , nk>;
       for every neighbor n of nk
            add <n0, . . . , nk, n> to frontier;
end while
```

# 4 Uninformed (or Blind) Search Strategies

So far, we do not pay attention to the detail of how to select the next node when expand the frontier. The algorithm does not specify how they should be implemented.

**Definition.** A *search strategy* specifies which node should be selected at each iteration and how the frontier should be expanded.

**Definition.** A *blind search strategy* is a search strategy that does not take into account where the goal is.

- **Depth-First Search**: Completing the search of one path before exploring the other. Treats the frontier as a stack.
- **Breadth-First Search**: Always takes the path with fewest arcs to expand. Treats the frontier as a queue.

**Space and Complexity of the Blind Search Strategies:** Important factors in deciding which strategy to use.

| Depth-First | Breadth-First |
|---|---|
| Might not find the solution | Guarantee to find a solution if one exists if branching factor is finite |
| Linear in size of the path being explored | Exponential time and space in size of the path being explored |
| Search is unconstrained until solution is found | Search is unconstrained by the goal |

Table 1: Comparison between DFS and BFS

It is customary to use a data structure called *node* in the implementation of the search algorithms. A node will contain the following information:

- the state
- the parent node
- the cost to reach the node

For now, we will not pay attention to the cost. For the implementation of BFS, the frontier will be a queue of nodes. For the implementation of DFS, the frontier will be a stack of nodes. In addition, a list of states that have been explored is created to avoid cycle. The general algorithms will have the following form:

```
Input: a graph,
       a set of start nodes,
       a set of goal nodes

frontier := {<s> : s is a start node};
visited  := {};

while frontier is not empty:
       select and remove path <n0, . . . , nk> from frontier;
       if (nk) has not been visited
            add nk to visited
            if goal(nk)
               return <n0, . . . , nk>;
            endif
            for every neighbor n of nk
               if n is not in visited
                  add <n0, . . . , nk, n> to frontier;
               endif
            endfor
end while
return no-solution
```

We see how the algorithms work in the next example.

**Example 4.1** *Let us consider the problem* $\langle \Sigma, succ, S_0, G \rangle$ *where*

- $\Sigma = \{1, 2, 3, 4, 5, 6, 7\}$
- $succ = \{(1,2), (3,4), (1,5), (1,1), (2,3), (4,6), (6,4), (3,5), (5,6), (4,4), (2,7)\}$
- $S_0 = \{1\}$
- $G = \{4\}$

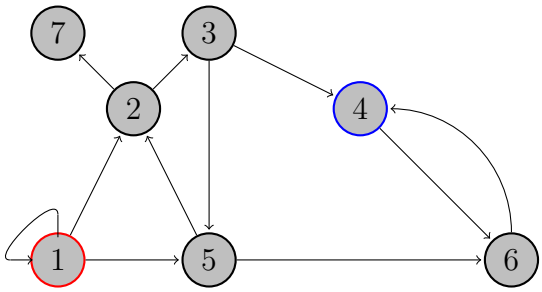



Figure 1: A search problem

*Suppose that we run the BFS (resp. DFS) to find a solution. The following computation occurs:*

- *BFS: initial node* $(1, nil)$ *(or* $\langle 1 \rangle$*) - we call this node a;*

  - *Iteration 1: node a is selected; state 1 has not been visited; it is expanded with successors (neighbors) 2 and 5. The result is the queue containing* $(2, a)$ *(or* $\langle 1, 2 \rangle$*, call this node b) and* $(5, a)$ *(or* $\langle 1, 5 \rangle$*, call this node c). The visited states contain* $\{1\}$*.*
  - *Iteration 2: node b is selected; state 2 has not been visited; it is expanded with successors (neighbors) 7 and 3. The result is the queue will contain* $(5, a)$ *(or* $\langle 1, 5 \rangle$*, node c),* $(7, b)$ *(or* $\langle 1, 2, 7 \rangle$*, node d),* $(3, b)$ *(or* $\langle 1, 2, 3 \rangle$*, node e). The visited states contain* $\{1, 2\}$*.*
  - *Iteration 3: node c is selected; state 5 has not been visited; it is expanded with successors (neighbors) 6 and 2. Since 2 has been visited, the result is the queue will contain* $(7, b)$ *(or* $\langle 1, 2, 7 \rangle$*, node d),* $(3, b)$ *(or* $\langle 1, 2, 3 \rangle$*, node e),* $(6, c)$ *(or* $\langle 1, 5, 6 \rangle$*, node f). The visited states contain* $\{1, 2, 5\}$*.*
  - *Iteration 4: node d is selected; state 7 has not been visited; it has no successor. So, the result is the queue will contain* $(3, b)$ *(or* $\langle 1, 2, 3 \rangle$*, node e),* $(6, c)$ *(or* $\langle 1, 5, 6 \rangle$*, node f). The visited states contain* $\{1, 2, 5, 7\}$*.*
  - *Iteration 5: node e is selected; state 3 has not been visited; it has successors 4 and 5. 5 has been visited. So, the result is the queue will contain* $(6, c)$ *(or* $\langle 1, 5, 6 \rangle$*, node f),* $(4, e)$ *(or* $\langle 1, 2, 3, 4 \rangle$*, node g). The visited states contain* $\{1, 2, 5, 7, 3\}$*.*
  - *Iteration 6: node f is selected; state 6 has not been visited; it has successor 4. 4 has not been visited. So, the result is the queue will contain* $(4, e)$ *(or* $\langle 1, 2, 3, 4 \rangle$*, node g),* $(4, f)$ *(or* $\langle 1, 5, 6, 4 \rangle$*, node h). The visited states contain* $\{1, 2, 5, 7, 3, 6\}$*.*
  - *Iteration 7: node g is selected; state 4 has not been visited; it is the goal. So, the result is the path* $\langle 1, 2, 3, 4 \rangle$*. The visited states contain* $\{1, 2, 5, 7, 3, 6, 4\}$*.*

- *DFS: initial node* $(1, nil)$ *(or* $\langle 1 \rangle$*) - we call this node a;*

  - *Iteration 1: node a is selected; state 1 has not been visited; it is expanded with successors (neighbors) 2 and 5. The result is the stack containing* $(2, a)$ *(or* $\langle 1, 2 \rangle$*, call this node b) and* $(5, a)$ *(or* $\langle 1, 5 \rangle$*, call this node c). The visited states contain* $\{1\}$*.*
  - *Iteration 2: node c is selected (assuming that we consider the relation succ in the same order, then* $(5, a)$ *will be pushed into the stack later than* $(2, a)$*); state 5 has not been visited; it is expanded with successors (neighbors) 2 and 7. The result is the queue will contain – in the order –* $(6, c)$ *(or* $\langle 1, 5, 6 \rangle$*, node d),* $(2, c)$ *(or* $\langle 1, 5, 2 \rangle$*, node e),* $(2, a)$ *(or* $\langle 1, 2 \rangle$*, node b), The visited states contain* $\{1, 5\}$*.*

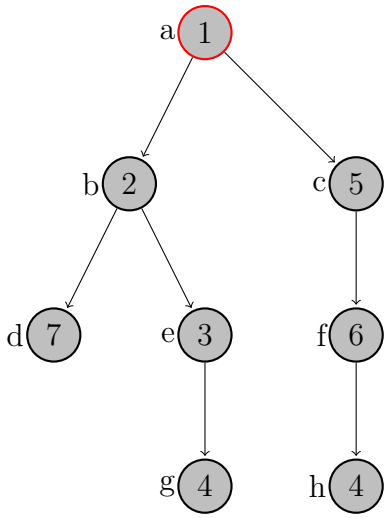



Figure 2: BFS search tree

– *Iteration 3: node d is selected; state 6 has not been visited; it is expanded with successor 4. The result is the queue will contain – in the order –* $(4, d)$ *(or* $\langle 1, 5, 6, 4\rangle$*, node f),* $(2, c)$ *(or* $\langle 1, 5, 2\rangle$*, node e),* $(2, a)$ *(or* $\langle 1, 2\rangle$*, node b), The visited states contain* $\{1, 5, 6\}$*.*

– *Iteration 4: node f is selected; state 4 has not been visited; it is the goal. Return the path* $\langle 1, 5, 6, 4\rangle$*. The visited states contain* $\{1, 5, 6, 4\}$*.*

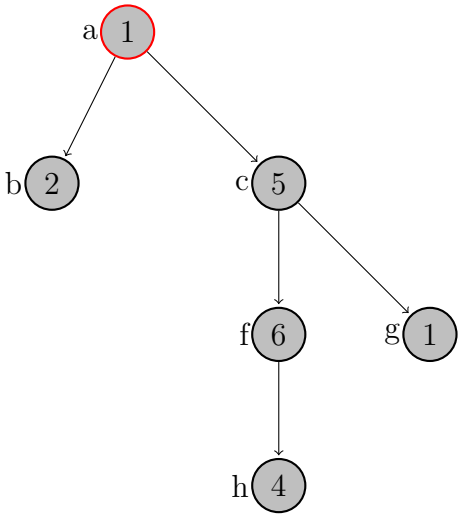



Figure 3: DFS search tree

# 5 Variations of Breadth/Depth first search

Different variations for BFS and DFS have been proposed. Each has certain advantages but also has some disadvantages.

- **Lowest-Cost-First Search** also called **Uniform-Cost Search**: Need to have a function $c(n)$ that returns the cost of reaching a node $n$. Very often, this function is defined by associating some cost to the arcs of the graph and the cost between two nodes $n$ and $n'$ is defined by the summation of all the costs of the arcs along the path. This strategy requires the expansion of the lowest cost path first. Treats the frontier as a priority queue. Lowest-cost strategy is similar to breath-first. It coincides with BFS if the cost of every arc is the same.
- **Depth-limited search**: Use DFS and limit the depth that will be explored. The advantage of this approach is that it does not require exponential space. The disadvantage of this approach is that we often do not have enough information to select a depth that can be used as the effective limit.
- **Iterative deepening DFS** also called **IDS**: Use depth-limited search for depth = 0, 1, ..., until find the solution. Guarantees completeness and minimal solution. The disadvantage was the amount of nodes that need to be recomputed.
- **Bidirectional search**: a great idea but it is difficult to implement due to the fact that real-world problems do not provide an easy way for computing of the inverse function.

# 6 Comparing between Search Strategies

| Criterion | BFS | Uniform Cost | DFS | Depth-Limited | Iterative Deepening | Bidirectional |
|---|---|---|---|---|---|---|
| Complete? | Yes$^{a}$ | Yes$^{a,b}$ | No | No | Yes$^{a}$ | Yes$^{a,d}$ |
| Time | $O(b^{d+1})$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^{d+1})$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(bl)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal | Yes$^{c}$ | Yes | No | No | Yes$^{c}$ | Yes$^{c,d}$ |

In the above table:

- $b$ is the forward branching factor.
- $d$ is the depth of the shallowest solution.
- $m$ is the maximum depth of the search tree.
- $l$ is the depth limit.
- $C^*$ is the cost of the optimal solution.

- Superscript meaning: $^a$ complete if $b$ is finite; $^b$ complete if step costs $\geq \epsilon$ for positive $\epsilon$; $^c$ optimal if step costs are all identical; $^d$ if both directions use breadth-first search.

**Remark**: note that the above discuss the complexity of the algorithms without consideration about the visited states.

# 7 Heuristic Search – Informed Search Strategies

**Idea:** Taking into account the goal information and (if available) knowledge about the goal. At any iteration, the "most promising" node – one, that probably leads to the goal – is selected to expand the frontier. Represent as an *evaluation function*, $f$, that estimates the cost of reaching a node $n$. The node with the lowest cost will be chosen.

The choice of $f$ determines the search strategy. Most best first search algorithm includes a component $h$, called a *heuristic function* from the set of nodes into non-negative real numbers, i.e., for each node $n$, $h(n) \geq 0$.

$h(n)$ is *underestimate* if it is less than or equal the actual cost of the lowest-cost path from $n$ to the goal.

**Example:** For the robot delivery, the straight-line distance between the node and the goal is a good heuristic function, which is underestimate. In the class, we use the example of finding a path connecting two cities.

**Example:** For the SLD search graph, the number of atoms in the query is a heuristic function.

- **Greedy Best-First Search**: Always select the element that appears to be the closest to the goal, i.e., lowest $h(n)$. Frontier is treated as priority queue.
- **Heuristic Depth-First Search**: Like depth-first, but use $h(n)$ in deciding what branch of the search tree to explore. (this is not discussed in the book)
- **A* Search**: Selecting the next node based on the actual cost and the estimate cost. If the actual cost to the node $n$ is $c(n)$ and the estimate cost from $n$ to the goal is $h(n)$, the value $f(n) = c(n) + h(n)$ will be used in selecting the node to expand the frontier. This method is implemented by a priority queue based on $f(n)$.

| Best-First | Depth-First | A* |
|---|---|---|
| Might not find the solution | Might not find the solution | Guarantee to find an optimal solution if one exists and branching factor is finite for admissible heuristic |
| Exponential time and space in size of the path being explored | Linear in size of the path being explored | Exponential time and space in size of the path being explored |

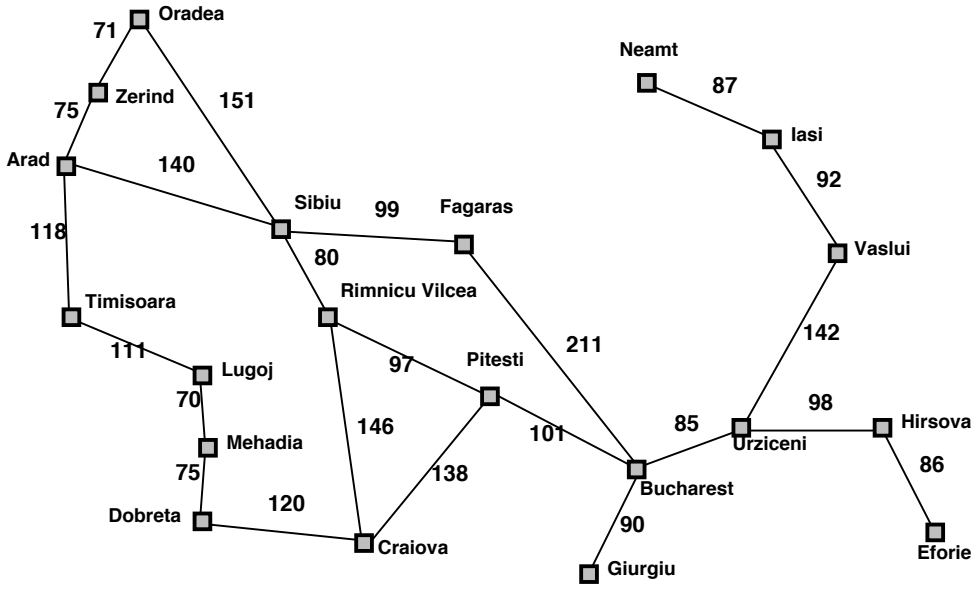
## Conditions for optimality: admissibility and consistency

**Admissible heuristic:** $h(n)$ never overestimates the cost to reach the goal.

**Consistent heuristic:** for every $(n, n') \in succ$, $h(n) \leq c(n, n') + h(n')$ (triangle inequality)

Admissible guarantees that $A^*$ is complete and optimal even if visited states are not recorded; Consistency guarantees that $A^*$ is complete and optimal if visited states are recorded.

**Example 7.1** *Consider the driving to Bucharest problem from the book. The map of Romania and table containing straight-line distance between cities of Romania and Bucharest are given in Figure 4. We wish to go to Bucharest from Arad.*




| Straight–line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 178 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 98 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

Figure 4: Map of Romania with Straight Line Distance to Bucharest

*The straight line distance can be used as a heuristic function for this problem. The heuristic function is admissible?*

## Memory-bounded heuristic search

**Iterative deepening A***: Instead of using the depth as the bound, use $f(n)$. Initially, $f(s)$ is used ($s$ is the start node with minimal $h$-value). When the search fails unnaturally, the next bound is the minimal $f$-value that exceeded the previous bound.

**Recursive best-first search** (RBFS) keeps track of the best $f$-value of alternate path. Use this value to backtrack.

# 8 Refinements to Search Strategies

**Idea:** Deals with cycles in the graph

- **Cycle checking**: Before inserting new paths into the frontier, check for their occurrence in the path. If the path selected to expand is $\langle n_0, \ldots, n_k \rangle$ and $m$ is a neighbor of $n_k$ we add to the frontier the path $\langle n_0, \ldots, n_k, m \rangle$ if $m$ does not occur in $\langle n_0, \ldots, n_k \rangle$. This is

  - easy to implement in depth-first (one extra bit); set when visits; reset when backtracks;
  - need more time in exponential space strategies.

  Observe that this eliminates only the cycle within a path. The next technique is the one that we discuss in the class.

- **Multi-path Pruning**: Before inserting new paths into the frontier, check for the occurrence of new neighbor in the frontier. Need to be done carefully if shortest/lowest cost path need to be found. In A*, *monotone restriction* is sufficient to guarantee that the shortest path to a node is the first path found to the node.

  **monotone restriction**: $|h(n') - h(n)| \leq d(n', n)$ where $d(n', n)$ is the actual cost from $n'$ to $n$.

  Subsumes cycle checking. Preferred in strategies where the visited nodes are explicitely stored (breath-first); not preferred in depth-first search since the requirement of space required.

- **Iterative deepening**: Instead of storing the frontier, recompute it. Use depth-first to explore paths of 1, 2, 3, ... arcs until solution is found. When the search fails *unnaturally*, i.e., the depth bound is reached; in that case, restart with the new depth bound.

  - Linear space in size of the path being explored.
  - Little overhead in recomputing the frontier.

- **Direction of Search**: forward (from start to goal), backward (from goal to start), bidirectional (both directions until meet). The main problem in bidirectional search is to ensure that the frontiers will meet (e.g. breath-first in one direction and depth-first in the other).

  - **Island-driven Search**: Limit the places where backward and forward search will meet (designated *islands* on the graph). Allows a decomposition of the problem in group of smaller problems. To find a path between $s$ and $g$, identify the set of islands $i_0, \ldots, i_k$ and then find the path from $s$ to $i_0$, from $i_j$ to $i_{j+1}$, and finally from $i_k$ to $g$.

- **Searching in a Hierarchy of Abstractions**: Find solution at different level of abstraction. Details are added to the solution in refinement steps.

- **Dynamic Programming**: construct the perfect heuristic function that allows depth-first heuristic to find a solution without backtracking. The heuristic function represents the *exact costs* of a minimal path from each node to the goal. This will allow us to specify *which arcs to take* in each step, which is called a **policy**. Define

$$dist(n) = \begin{cases} 0 & if \ \ is\_goal(n) \\ min_{\langle n,m\rangle \in A}(|\langle n,m\rangle| + dist(m)) & otherwise \end{cases}$$

  where $dist(n)$ is the actual cost to the goal from $n$.

  $dist(n)$ can be computed backward from the goal to each node. It can then be stored and used in selecting the next node to visit.

  - $dist(n)$ depends on the goal;
  - $dist(n)$ can be pre-computed only when the graph is finite;
  - when $dist(n)$ is available, only linear space/time is needed to reach the goal;

# 9 Local Search Algorithms

Most of the previous algorithms need to store the complete search tree in the memory and return the path from the start node to the goal node as solution. In several problems, it is not very *important* to know *how to get to the goal*. For example,

- In the map coloring problem, we are interested in figuring out whether there is a way to color the map. It is not so important to know in which order the states/countries are colored;

- In the $n \times n$ queens problem, we are interested in the final configuration of the board, in which we know the location of each queen. It does not matter whether the queen on the first column is placed on the board in the last step or in the first step.

- There are several practical problems with this property: floor-planning, job-shop scheduling, etc.

For problems with the above characterization, local search algorithms can be useful. The key idea of local search algorithms is

- it keeps only one complete-state representation of the problem in the memory;

- it considers only *neighbors of the current state* in deciding which will be the next state (In this sense, it does not complete the *expand* phase of the previously studied algorithms);

- it uses an evaluation function to evaluate possible successors; the decision for which state should be consider next is made based using this function. This function is often referred to as *heuristic function*.

We discuss the above using the two examples:

- **Complete-state representation**:

  For the map coloring example, a local search algorithm will store the incomplete colored map (perhaps as a set of pairs of the form $(S, C)$ where $S$ is a state a $C$ is a color or *nil*; the set must contain – for each state – one pair).

  For the $n$ queens problem, a possible complete state representation consists of the location of the $n$-queens.

- **Neighbor**:

  For the map coloring example, a neighbor of a state can be any of the possible coloring which has one more state with a new color provided that this state is not colored before and shares border with some already colored states.

  For the $n$ queens problem, a neighbor of a state is another configuration in which one of the queen moves to a new location (in the same row).

- **Evaluation function**:

  For the map coloring example, the evaluation function can be one that returns the number of states which have not been colored and share some border with some already colored states.

  For the $n$ queens problem, the evaluation function can be one that returns the number of pairs of attacking queens.

There are a number of local search algorithms:

- **Hill-climbing algorithm** (also called **greedy local search**): the basic idea of this algorithm is to consider the *best* successor among all possible neighbors. The algorithm begins with the generation of a random initial state. It then goes into a loop which executes (a) evaluate the neighbors; (b) select a neighbor for the next move (if the goal has not been reached yet). For example, if the current state $S$ has the value $h(S) = 3$, then the next state $S'$ has to have the value $h(S') > 3$. Please see the book for the example on the $n$ queens problem.

  This algorithm is *incomplete* for different reasons: it can (i) stuck at a local maxima; (ii) stuck at a ridge; or (iii) stuck at a shoulder.

- **Random-restart hill climbing**: try to avoid the local maxima by randomly restart the search. This is done as follows: (a) timeout the algorithm after a certain limit; (b) regenerate the initial state. This method guarantees that the solution will be found if exists with the probability 1. This algorithm avoids the possible of the algorithm getting stucked at a local maxima.

- **Random sideways moves**: in this algorithm, not only the best neighbor but also a neighbor with the same value as the current state will be considered. This helps to avoid shoulders.

- **Simulated annealing**: this algorithm tries to avoid the local maxima and the shoulders by accepting some bad moves, with different probability in the long run.

**NOTE':** All algorithms can be found in the book or on the slide.

# 10 Searching with NonDeterministic Actions

Our study of search algorithms so far focuses on the classical search problems. We discuss the algorithms at an abstract level and do not pay attention to their implementation. We assume that the set of states $\Sigma$ anf the *succ* relation are given. This assumption is good for the understanding of the search algorithms but is, in many situations, impractical. For example, in the 8-puzzle problem, we have 9! (36288) states; in the 8-queens problem, we have around $60^8$ states. The *succ* relation would have a larger number of elements than the number of elements in the set of states. This is a big issue for using the search algorithms as it is represented so far (just think that you would need to create the data file for the implementation of the search algorithm in the first programming assignment for the 8-puzzle problems!). How can we address these problems? A first idea is to encode the states by number, so the set of states can be specified by one number, the size of $\Sigma$ (so the states are $1, \ldots, |\Sigma|$). This only solves the representation problem for the set of states but not for the *succ* relation. One way to solve this problem for the *succ* relation is that we replace *succ* with a *set of actions* that can be used to define *succ*. So, we will be representing a *problem* using

$$\langle A, S_0, G \rangle$$

where

- We now do not have an explicit definition for the set of states $\Sigma$ which is now implicitly given by $S_0$ and the execution of sequences of actions in $S_0$.

- $A$ is a set of actions that can be used to generate states.

- $S_0$ is a set of states describing the set of *initial* states from which the search starts.

- $G$ is a set of states describing the set of *goal* states where the search can stop.

For each action $a \in A$, the *execution* of the action $a$ in a state $s$ will result in zero, one, or many states. For example, the set of actions for the 8-puzzle problem is $\{left, right, down, up\}$. Given any pair of a state of the 8-puzzle problem and an action, we can easily determine the next state. An illustration of this is given in Figure 5.

We can represent the *succ* relation as follows: $(s, s')$ iff there exists $a \in A$ such that $s'$ is the result of the execution of $a$ in $s$. Mathematically, we can write

$$succ \subseteq \Sigma \times A \times \Sigma$$

You might wonder whether this is helpful to us. In many applications, it will, since we can write a procedure to compute the *succ* relation instead of enumerating all pairs in *succ*.

Observe that the set of actions can be something more complicated. For driving from Arad to Bucharest, the set of actions would contain the action $drive(a, b)$ for every pair of different cities $a$ and $b$ on the map (assuming that the roads are bidiretional). So, there are 380 actions since there are 20 cities! We will go back to this issue later in the course of the class.

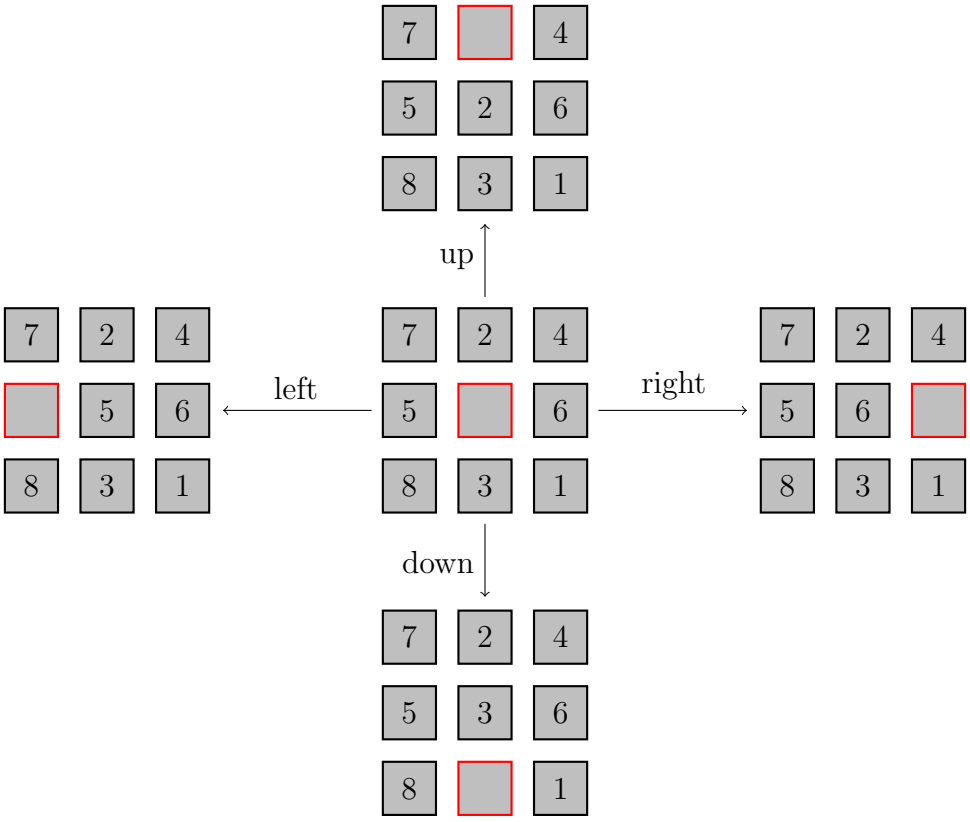



Figure 5: A state in the 8-puzzle problem with four actions

One good thing about representing a search problem as $\langle A, S_0, G \rangle$ is that, this is close to the design of an intelligent agent. $A$ is the set of actions that the agent can do. The implicitly defined set of states $\Sigma$ is the set of the states of the environment in which the

agent is situated in. A search problem then represents a *planning problem* for the agent, who is in a state $S_0$ and wants to be in one of the states in $G$. The algorithms (BFS, DFS, IDS, A$^*$, IDA$^*$, local search, etc.) allow the agents to compute the sequence of actions that achieves the goal if $S_0$ is a single state and for each pair of an action $a$ and a state $s$, the execution of $a$ in $s$ results in zero or one state.

We say that an action $a$ is *deterministic* if for each state $s$, the execution of $a$ in $s$ results in zero or one state. A problem is called *deterministic* ( if every action (of the problem) is deterministic.

We will now discuss about searching in problems with nondeterministic actions. An example is the vacuum world (Figure 6). The actions of these problem are *left, right, suck*. The actions behave as follows:

- *left*: the robot moves from right room to left room
- *right*: the robot moves from left room to right room
- *suck*:
  - when applied to a room with dirt, it cleans the room but sometimes clean the other room too
  - when applied to a clean room with dirt, it sometimes deposits dirt on the carpet

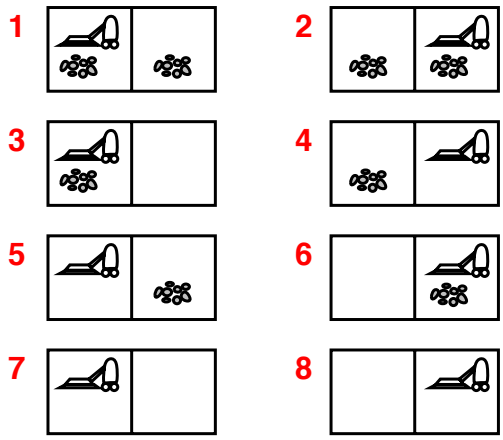



Figure 6: The vacuum world - States

The *succ* relation in this case is as follows:

- $(k, right, k+1) \in succ$ for $k = 1, 3, 5, 7$
- $(k-1, left, k) \in succ$ for $k = 2, 4, 6, 8$
- $(1, suck, 7) \in succ$ and $(1, suck, 5) \in succ$

- $(2, suck, 8) \in succ$ and $(2, suck, 4) \in succ$
- $(3, suck, 7) \in succ$
- $(4, suck, 2) \in succ$ and $(4, suck, 4) \in succ$
- $(5, suck, 1) \in succ$ and $(5, suck, 5) \in succ$
- $(6, suck, 8) \in succ$
- $(7, suck, 7) \in succ$ and $(7, suck, 5) \in succ$
- $(8, suck, 8) \in succ$ and $(8, suck, 4) \in succ$

Suppose that the robot is in the state 1, what should the robot do in order to clean the rooms? Can we use, say BFS, to find the course of actions for the robot to accomplish the job?

**AND-OR search trees**: To deal with non-deterministic actions, we create an AND-OR search tree.

An AND-OR search tree for a search problem $\langle \Sigma, A, S_0, G \rangle$ where $S_0$ is a singleton (i.e., $S_0 = \{s_0\}$) is constructed as follows.

- the root of the tree is the node $\langle s_0 \rangle$ of the type OR-node;
- for every action $a \in A$ and a leaf $\langle s_0, a_0, s_1, \ldots, a_{k-1}, s_k \rangle$ such that there exists some $s'$, $(s_k, a, s')$ belongs to *succ* then
  - if the action is deterministic, i.e., there exists only one $s'$ such that $(s_k, a, s')$ belongs to *succ*, then the node $\langle s_0, a_0, s_1, \ldots, a_{k-1}, s_k \rangle$ has a child as an OR-node which has a child node $\langle s_0, a_0, s_1, \ldots, a_{k-1}, s_k, a, s' \rangle$;
  - if the action is nondeterministic, i.e., there exists more than one $s'$ such that $(s_k, a, s')$ belongs to *succ*, then the node $\langle s_0, a_0, s_1, \ldots, a_{k-1}, s_k \rangle$ has a child as an AND-node whose children are $\langle s_0, a_0, s_1, \ldots, a_{k-1}, s_k, a, s' \rangle$ for every s' such that $(s_k, a, s') \in succ$.

An example of an AND-OR search tree for the vacuum world is given in Figure 7. The AND-OR nodes are circles and the nodes $\langle s_0, a_0, s_1, \ldots, a_{k-1}, s_k, a, s' \rangle$ are represented by the path from the root to the leaves.

**Solution**: What is a solution in an AND-OR search tree? In the search trees constructed for other algorithms, the solution is a path from the root to a leaf satisfying the goal. In an AND-OR search tree, the solution is a sub-tree satisfies the following conditions:

- has a goal node at every leaf
- specifies one action at each of its OR nodes

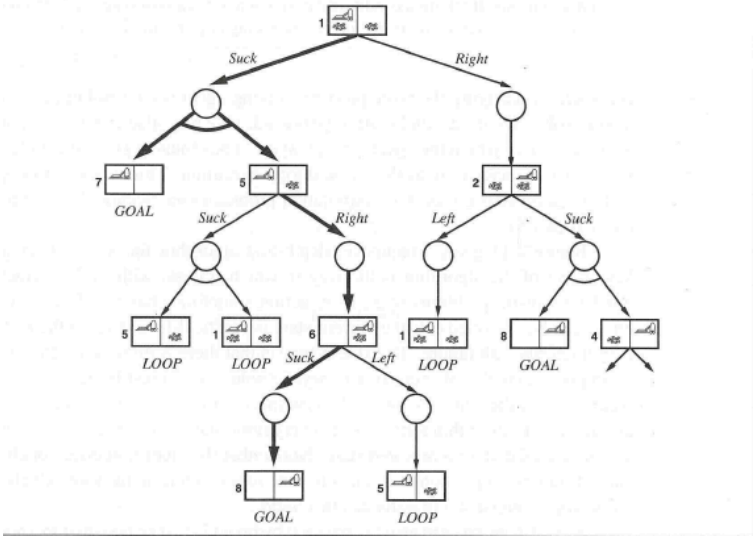



Figure 7: Part of the AND-OR search tree for the robot to clean the rooms

- includes every outcome branch at each of its AND nodes

The solution is then constructed from the sub-tree as follows: start with the empty plan [ ] and traversing with the current node as the root,

- if the child of the current node is an OR node then append the action leading to the OR node to the current plan and continue with the child of the OR node
- if the child of the current node is an AND node then append the action leading to the AND node to the current plan, compute the solutions from all children of the AND node to the goal, and append the CASE plan

```
CASE
   state=state_of_1st_child: plan_of_1st_child; break;
   .....
   state=state_of_nth_child: plan_of_nth_child; break;
ENDCASE
```

where $n$ is the number of children for the AND node.

**Implementation**: To implement the search, we can extend the previous developed algorithms. We could use the frontier to maintain a set of unexplored nodes as before. The nodes are now of two types: AND-nodes and OR-nodes. We would need to modify our representation (data structure) though.

The frontier will need to contain elements that are sets of nodes. In this note, I will write a node $n$ in the frontier as a set of elements of the form $\langle s_0, a_0, s_1, \ldots, a_{k-1}, s_k \rangle$. By $L(n)$ we

denote the set $\{s_k \mid \langle s_0, a_0, s_1, \ldots, a_{k-1}, s_k\rangle \in n\}$, i.e., $L(n)$ is the set of states in $n$. When we are programming this algorithm, we should use a structure similar to the one defined in search node class. The set of visited states should also contain sets of states.

```
search_non_deterministic(A, s0, G)
Input:  a problem with nondeterministic actions (A, s0, G)
           (assume that the function succ is implemented
            and we can compute s' such that
            (s, a, s') belongs to succ for every
            pair of s, a)

frontier := {{<s>} : s is a start node};
visited  := {};

while frontier is not empty:
       select and remove node N from frontier;
       if (L(N)) has not been visited
            add L(N) to visited
            if the goal is satisfied for every element of n
               return solution(n);
            endif
            for each <s0,a0,...,a(k-1),sk> in N
                 such that sk does not satisfy the goal
                 for each action a from A
                      such that the execution of a in sk results in some states
                      create a new node N' by replace <s0,a0,...,a(k-1),sk> with
                          the set <s0,a0,...,a(k-1),sk,a,s'>
                      add N' to the frontier
end while
return no-solution
```

The solution extraction problem is as follows:

```
solution(N)
Input: a node consisting of several search node
       of the form <s0,a0,....,a[n-1]sn>

if N is empty or contains a member of the form <s>
   for some state s
   return []
if every element in N contains the same first state
   and the first action is a then
      return [a; solution(N')]
```

```
         where N' is obtained from N by removing
         the first state and the first action
         from every element of N
else divide N into groups G1, .... , Gm
   such that the first state of members in Gi is the same
   return
      case
         state = s1: solution(G1); break;
         ....
         state = sm: solution(Gm); break;
      endcase
```

**Illustration**: Consider the vacuum world. We will use BFS. Assume that the initial state is 1 and the goal is either 7 or 8.

**Initialization:** $frontier = [\{\langle 1\rangle\}]$ and $visited = \emptyset$.

**Step 1:** Select $N = \{\langle 1\rangle\}$ for expansion. $L(N) = \{1\}$ so $visited = \{\{1\}\}$. Goal is not satisfied in 1. So, expansion continues.

The execution of *right* in 1 results in 2, and *suck* in 1 results in 7 or 5, so $frontier = [\{\langle 1, right, 2\rangle\}, \{\langle 1, suck, 5\rangle, \langle 1, suck, 7\rangle\}]$.

**Step 2:** Select $N = \{\langle 1, right, 2\rangle\}$ for expansion. $L(N) = \{2\}$ so $visited = \{\{1\}, \{2\}\}$. Goal is not satisfied in 2. So, expansion continues.

The execution of $left$ in 2 results in 1, and *suck* in 2 results in 4 or 8, so

$$frontier = \begin{bmatrix} \{\langle 1, suck, 5\rangle, \langle 1, suck, 7\rangle\}, \\ \{\langle 1, right, 2, left, 1\rangle\}, \\ \{\langle 1, right, 2, suck, 4\rangle, \langle 1, right, 2, suck, 8\rangle\} \end{bmatrix}$$

**Step 3:** Select $N = \{\langle 1, suck, 5\rangle, \langle 1, suck, 7\rangle\}$ for expansion. $L(N) = \{5, 7\}$ so $visited = \{\{1\}, \{2\}, \{5, 7\}\}$. Goal is not satisfied in 5. So, expansion continues.

7 satisfies the goal, so we consider $\langle 1, suck, 5\rangle$. The execution of *right* in 5 results in 6, and *suck* in 5 results in 1 or 5, so

$$frontier = \begin{bmatrix} \{\langle 1, right, 2, left, 1\rangle\}, \\ \{\langle 1, right, 2, suck, 4\rangle, \langle 1, right, 2, suck, 8\rangle\}, \\ \{\langle 1, suck, 5, right, 6\rangle, \langle 1, suck, 7\rangle\}, \\ \{\langle 1, suck, 5, suck5\rangle, \langle 1, suck, 5, suck, 1\rangle, \langle 1, suck, 7\rangle\} \end{bmatrix}$$

**Step 4:** Select $N = \{\langle 1, right, 2, left, 1\rangle\}$ for expansion. $L(N) = \{1\}$ already visited. So, not expanded.

$$frontier = \begin{bmatrix} \{\langle 1, right, 2, suck, 4\rangle, \langle 1, right, 2, suck, 8\rangle\}, \\ \{\langle 1, suck, 5, right, 6\rangle, \langle 1, suck, 7\rangle\}, \\ \{\langle 1, suck, 5, suck5\rangle, \langle 1, suck, 5, suck, 1\rangle, \langle 1, suck, 7\rangle\} \end{bmatrix}$$

**Step 5:** Select $N = \{\langle 1, right, 2, suck, 4\rangle, \langle 1, right, 2, suck, 8\rangle\}$ for expansion. $L(N) = \{4, 8\}$, so $visited = \{\{1\}, \{2\}, \{5, 7\}, \{4, 8\}\}$. Goal is not satisfied in 4. So, expansion continues.

8 satisfies the goal, so we consider $\langle 1, right, 2, suck, 4\rangle$ The execution of $left$ in 4 results in 3, and $suck$ in 4 results in 2 or 4, so

$$frontier = \left[\begin{array}{l} \{\langle 1, suck, 5, right, 6\rangle, \langle 1, suck, 7\rangle\}, \\ \{\langle 1, suck, 5, suck5\rangle, \langle 1, suck, 5, suck, 1\rangle, \langle 1, suck, 7\rangle\} \\ \{\langle 1, right, 2, suck, 4, left, 3\rangle, \langle 1, right, 2, suck, 8\rangle\} \\ \{\langle 1, right, 2, suck, 4, suck, 4\rangle, \langle 1, right, 2, suck, 4, suck, 2\rangle, \langle 1, right, 2, suck, 8\rangle\} \end{array}\right]$$

**Step 6:** Select $N = \{\langle 1, suck, 5, right, 6\rangle, \langle 1, suck, 7\rangle\}$ for expansion. $L(N) = \{6, 7\}$, so $visited = \{\{1\}, \{2\}, \{5, 7\}, \{4, 8\}, \{6, 7\}\}$. Goal is not satisfied in 6. So, expansion continues.

7 satisfies the goal, so we consider $\langle 1, suck, 5, right, 6\rangle$ The execution of $left$ in 6 results in 5, and $suck$ in 6 results in 8, so

$$frontier = \left[\begin{array}{l} \{\langle 1, suck, 5, suck, 5\rangle, \langle 1, suck, 5, suck, 1\rangle, \langle 1, suck, 7\rangle\} \\ \{\langle 1, right, 2, suck, 4, left, 3\rangle, \langle 1, right, 2, suck, 8\rangle\} \\ \{\langle 1, right, 2, suck, 4, suck, 4\rangle, \langle 1, right, 2, suck, 4, suck, 2\rangle, \langle 1, right, 2, suck, 8\rangle\} \\ \{\langle 1, suck, 5, right, 6, suck, 8\rangle, \langle 1, suck, 7\rangle\}, \\ \{\langle 1, suck, 5, right, 6, right, 5\rangle, \langle 1, suck, 7\rangle\} \end{array}\right]$$

**Step 7, 8, 9:** select the next three elements in the queue and expand it, similar to the above steps.

**Step 10:** Select $N = \{\langle 1, suck, 5, right, 6, suck, 8\rangle, \langle 1, suck, 7\rangle\}$ for expansion. $L(N) = \{7, 8\}$. Both satisfy goal. So, return $solution(N)$.

**How does** $solution(N)$ **work?** Let $N = \{\langle 1, suck, 5, right, 6, suck, 8\rangle, \langle 1, suck, 7\rangle\}$.

Because the first state for two members of $N$ is 1, we have

$solution(N) = [suck; solution(N')]$ where

$N' = \{\langle 5, right, 6, suck, 8\rangle, \langle 7\rangle\}$ which is **obtained from** $N$ by removing the first state and the first action from all members of $N$.

Because the first state of the two members of $N'$ is different ($5 \neq 7$), we divide $N'$ into two groups $G_1 = \{\langle 5, right, 6, suck, 8\rangle\}$ and $G_2 = \{\langle 7\rangle\}$. We then have:

$solution(N') = case\ \ state = 5 : solution(G_1); break; state = 7 : solution(G_2); break; endcase$ where $solution(G_1) = [right; suck]$ and $solution(G_2) = []$.

So, we get

$solution(N) = suck; [case\ state = 5 : right; suck; break; state = 7 : []; break; endcase]$