Yves Deville

3 Solve problems by searching, including with constraints, a fundamental pillar

3.1 Why is solving problems by search important within the broader artificial intelligence (AI) domain?

Search techniques are important and efficient tools for decision-making and solving complex combinatorial problems. This chapter proposes different paradigms that can be integrated in search agents to handle various classes of problems. Classical search algorithms are used in goal-based agents that are looking for a series of actions leading to a specific goal. A classical application is route planning in a search algorithm, the environment is modeled as a state. At each step, the agent has to choose between different actions that change the current state of the environment. The set of possible actions lead to state space graph that has to be searched by the algorithm in order to find an adequate series of actions leading to the goal. Many AI applications are optimization problems looking for the best solution according to some cost function. Search algorithms can then be extended to ensure the finding of the best solution and integrate heuristics to speed-up the search.

The second searching paradigm is called constraint satisfaction problem (CSP). It generalizes classical search techniques as it proposes an expressive modeling through decision variables and constraints that must be satisfied by a solution. It also covers optimization problems by adding an objective function to be maximized or minimized. Two approaches are described for solving CSP. Constraint programming offers a propagation strategy that reduces the search space. Local search, although it cannot always find an optimal solution, is a pragmatic and well-used technique for solving complex problems. It ensures to rapidly find an approximation of the best solution. This chapter presents an application related to a configuration problem often present in product and software delivery. But CSPs can be used in many AI and operation research problems such as scheduling, timetabling, manufacturing problems, business, circuit design, configuration, etc.

Logic is an important tool in AI. First-order and other advanced logics can be used for reasoning and knowledge representation (see Chapters 4 and 5). This chapter shows how propositional logic, the simplest logic, can be used not only to model complex AI problems and CSPs but also to efficiently solve them by using SAT solvers.

Open Access. © 2024 the author(s), published by De Gruyter. This work is licensed under the Creative Commons Attribution 4.0 International License.
https://doi.org/10.1515/9783111426143-003

3.2 Search algorithms

Search strategies are important methods underlying many approaches for problem solving. Search algorithms are also the basis for many optimization and planning methods.

3.2.1 What category of problems do search algorithms solve?

Search algorithms address problems for which the environment can be observed and modeled by a set of states, with known and deterministic rules of transition from one state to another. Using these rules, we look ahead for the best sequence of actions to take in order to reach from an initial state a specific goal, either a specific end state or a state that matches some conditions. The environmental knowledge that supports such decision is explicitly represented by states, and the decision taken by the agent involves the consideration of possible steps in the future, in order to reach the defined goal. So, the proposed techniques belong therefore to the category of "goal based" agent.

The use of search techniques requires an abstract formulation of the problem and of the available steps to construct a solution. By search, we mean the process of looking for a sequence of actions that leads to a goal starting from an initial state. For an optimization problem, one looks for the best sequence of actions, that is, the sequence with the lowest cost.

Example. The 8-puzzle, also called sliding blocks, is a simple but illustrative example. Starting from an initial state (see Figure 3.1), an action consists of moving a tile adjacent to the empty cell, also called blank cell, to the empty cell. The objective is then to find the (minimal number of) actions leading to the goal state from the initial state.

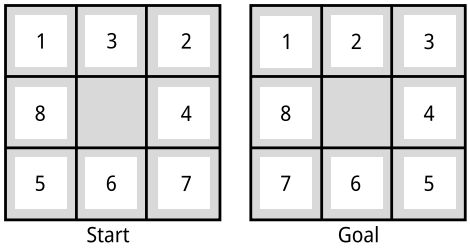


Figure 3.1: 8-Puzzle.

A state is an abstract representation of a possible stage of the problem. The initial state is a starting state for the search. The hypothesis is that the state is discrete, meaning that the number of states is finite, although usually very large.

An action transforms one state into another one by the application of an operator to the state. An operator can have preconditions on the given state to be applicable. The set of available operators depends on the given problem. The number of possible actions is an important factor on the complexity for solving the problem.

The goal can be an explicit state, but more generally, the goal is represented by a condition that must be satisfied by a state. In this case, the goal is a set of states.

In an optimization problem, a nonnegative cost is associated to each action. The cost of a sequence of actions, called a path cost, is then the sum of the cost of the actions in the sequence.

Example. A state for the 8-puzzle could be represented by a vector of 9 elements, listing the tiles, line by line, using the symbol B for the empty cell. The initial state is represented by $[1, 3, 2, 8, B, 4, 5, 6, 7]$. Of course, many other representations can be used, such as a 3×3 matrix.

An action for the 8-puzzle could be to move a (numbered) tile Up, Down, Left, or Right. Each action has a precondition that the move is possible (the target position is the empty cell). We thus have potentially $8 \times 4 = 32$ actions for a given state. A better design for the actions is to consider the move of the empty tile Up, Down, Left, or Right. We now only have 4 possible actions per state. As an example, $moveRight([1, 3, 2, 8, B, 4, 5, 6, 7]) = [1, 3, 2, 8, 4, B, 5, 6, 7]$.

The goal test of the 8-puzzle is the single state $[1, 2, 3, 8, B, 4, 7, 6, 5]$.

In the 8-puzzle, the cost of each action is 1.

3.2.2 Solving problems by search without heuristics

As illustrated by the above example, the overall approach to solve search problems requires essentially the following:

- modeling the problem as a search problem, defined by states, an initial state and a description of the possible transitions from a state to another one;
- a goal, defined by a condition which must be satisfied;
- search activities to identify the right sequence of actions, that is, choose the next action from several possibilities, in order to reach the goal.

Once a sequence of actions to reach the goal conditions is found, the agent can then execute that list of actions. The search activities essentially consist of growing a search tree directly on a "state space" graph until a solution is found. Different search strategies exist, and they can be uninformed or informed, with the notion of "heuristics."

The search for a solution, that is a sequence of actions, can be performed on a search tree derived from expanding the current state using the possible operators. This leads to tree-search algorithms that generate and traverse such a tree in order to find a state satisfying the goal.

A node in a search tree contains a state, a link to its parent node, the action applied on the state of the parent node yielding the state of the node, the depth of the node in the tree, and the path cost from the root to the node. In Figure 3.2, labels on the node correspond to different states. It clearly shows that in a search tree, the same state may appear in different nodes. Although the number of states is finite, the search tree can therefore be infinite.

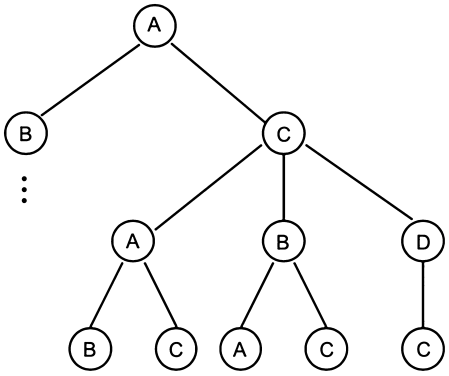


Figure 3.2: A search tree.

It is important to notice that the solution of the problem is not only the node with a state satisfying the goal, called a goal node, but the whole path from the root of the tree to a goal node defining a sequence of actions.

The parameters characterizing the complexity of a search problem are the branching factor, that is the average number of children of a node. In the 8-puzzle, the branching factor is about 1.732, assuming we do not consider the action to return to the state of the parent node. An example of a search tree for the 8-puzzle problem is shown in Figure 3.3.

3.2.2.1 A generic tree search algorithm

Figure 3.4 proposes a generic tree search algorithm. It has two parameters, the problem to be solved, and the frontier that should be empty when launching this algorithm. The frontier is an abstract data structure maintaining a set of nodes which ancestors in the search tree have all been goal-tested that have been visited.

The algorithm picks a node in the frontier and checks whether it has a state that satisfies the goal. If it is not the case, news nodes are created for each of the possible states reachable by an action from the state of the current node (the expand algorithm). These nodes are then inserted in the frontier and the search continues. At the beginning, the algorithm should start from the initial state. This is achieved by calling this tree

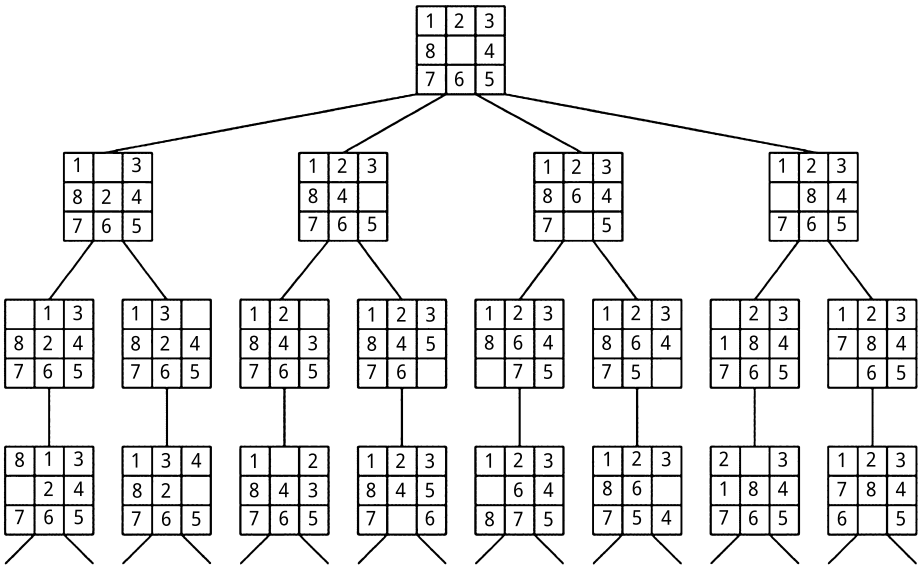


Figure 3.3: A search tree for the 8-puzzle.

```
tree_search( problem, frontier) : Node or Failure
        root := Node( problem.initial_state )
        frontier.insert( root )
        while not frontier.is_empty()
                node := frontier.pop()
                if problem.goal_test( node.state )  return node
                frontier.insert( expand( node, problem ) )
        return Failure

expand( node, problem ) : list of Nodes
        successors := empty_set
        for each operator in problem.operators such that operator.precondition( node.state )
                n := Node()
                n.state := operator.result(node.state)
                n.depth := node.depth + 1
                n. parent := node
                n.operator := operator
                n.path_cost := node.path_cost + operator.cost
                successors.add( n )
        return successors
```

Figure 3.4: Generic tree search algorithm.

search algorithm with the frontier containing only the root node with the initial state. The search will then start from the initial state.

The frontier is organized in such a way that the search algorithm chooses the next node to test and expand according to some preferred search strategy. Different organizations of the frontier lead to different versions of the algorithms. The frontier should be able to insert one or several nodes and should return the first element according to the internal organization of the frontier.

Figure 3.5 illustrates the first four iterations of the tree search algorithm on a small abstract example. The list of nodes in the frontier is successively (A), then (B, C), (C, D, E), and (D, E, F, G). In each step, the expanded node is highlighted by a marker.

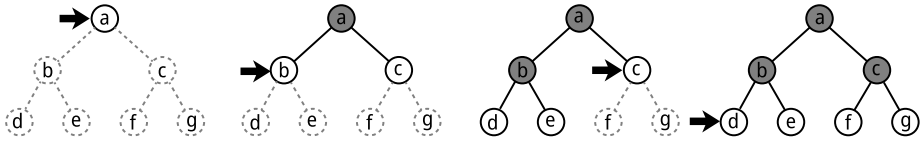


Figure 3.5: An example of execution.

The result of the tree-search algorithm is not only a node, but it is also implicitly the path from the root to the returned node, from which the sequence of actions to reach the goal from the initial state can be extracted.

The generic tree-search algorithm performed an uninformed search, also called blind search. The search is not goal oriented, nor does it exploit the path costs to guide the search to an optimal solution.

3.2.2.2 Breadth-first and depth-first search algorithms

Breadth-first search (BFS) is an instance of the generic tree search algorithm, depicted in Figure 3.6, where the frontier is organized as a FIFO queue (first in, first out). It means that after the visit of the root node, the algorithm will visit all the nodes at depth 1 before visiting the nodes at depth 2, etc. The order of the node visit is illustrated in Figure 3.7. It is assumed that the insertion of the nodes in the frontier realized by the frontier.insert() operation is done from left to right.

Depth-first search (DFS) is another instance of the tree search algorithm, also depicted in Figure 3.6. In DFS, the frontier is organized as a stack or LIFO queue (last in, first out). The order of the node visit is illustrated in Figure 3.8. It is assumed that the insertion of the nodes in the frontier realized by the frontier.insert() operation is done from right to left.

```
breadth_first_search_tree_search( problem, frontier) : Node or Failure
        return tree_search( problem, FIFO_queue() )
depth_first_search_tree_search( problem, frontier) : Node or Failure
        return tree_search( problem, stack() )
uniform_cost_tree_search( problem, frontier) : Node or Failure
        return tree_search( problem, priority_queue_G() )
```

Figure 3.6: BFS, DFS and uniform cost algorithms.

The evaluation of search algorithms can be done according to different criteria. Does the algorithm find a solution if one exists, what is called completeness? Is it able to find the least cost solution, what is called optimality? What is its computation time and space complexities? Computation time and space complexities are measured in terms of the (maximum) branching factor (b) of the search tree, the depth of the least cost solution (d) and the maximum depth (m) of the search tree. The value b is assumed to be finite, but the value m can be infinite. The time complexity is measured in terms of the number of nodes visited.

BFS is complete; its time and space complexities are $O(b^{d+1})$, the number of generated nodes. It is also optimal when the cost of each possible action is 1. BFS is nice as it handles very well infinite search tree. Its space complexity is often a problem for large problem. No nodes can be deleted during the search as they may be part of the solution, that is a path from the root to a goal node.

BFS can also be seen as an instance of the generic algorithm where the frontier is organized as a priority queue ordered by the depth of the node. Uniform cost search is a standard variant of BFS, depicted in Figure 3.4, where the frontier is a priority queue ordered by the path cost of the node (denoted $g(n)$). The node with the lowest cost is thus explored first. Uniform cost is complete and optimal.

DFS is not complete as it can fall in an infinite branch of the tree when m is infinite. If m is finite, the time complexity is $O(b^m)$. Its space complexity is $O(mb)$, that is the maximum number of nodes in the frontier. One can observe that all the nodes in the frontier are all children of nodes in a single path in the tree. All the visited nodes without children in the frontier can thus be removed during the search. DFS is not optimal.

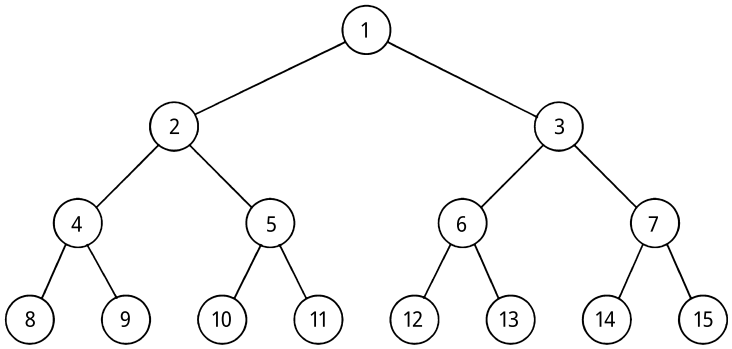


Figure 3.7: Order of node visit in BFS algorithms.

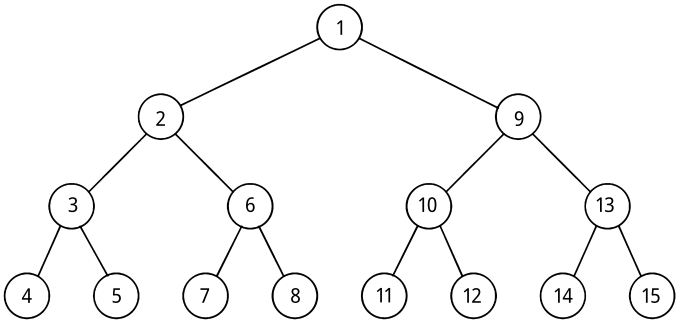


Figure 3.8: Order of node visit in DFS algorithm.

3.2.2.3 Iterative deepening

BFS has nice properties such as completeness and optimality but has a space issue. DFS has a nice space complexity but is not complete nor optimal. Iterative deepening combines the qualities of both algorithms. The idea is to apply DFS, but with a limited depth. The search tree is not expanded beyond the depth limit. Of course, DFS with a depth limit is not complete. Iterative deepening performs a succession of DFS with a increasing depth limit $(0, 1, 2, \dots)$ until a solution is found. It is complete as a solution will be found if there is one. The space complexity is $O(mb)$ as we restart a DFS at each step. Many nodes are visited multiple times, as illustrated in Figure 3.9. The root node is generated $(d + 1)$ times, the d nodes at depth 1 are generated d times, ... the b^{d-1} nodes at depth $d - 1$ are generated twice, the b^d nodes at depth d are generated once. From a complexity perspective, the time complexity is thus $O(b^d)$. Iterative deepening is thus a suitable uninformed algorithm for tree search.

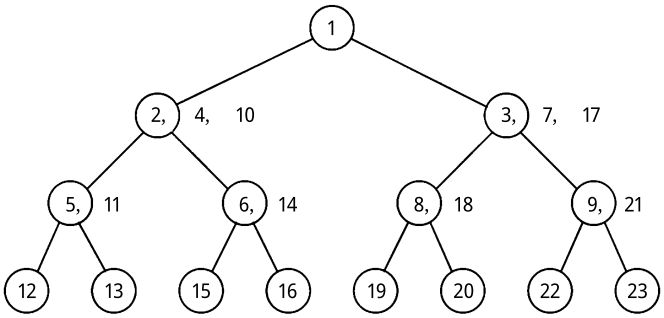


Figure 3.9: Order of node visit in iterative deepening algorithm.

3.2.3 Solving problems by search with heuristics

3.2.3.1 Evaluation and heuristics

BFS has nice properties. The idea of informed search is to exploit problem specific knowledge to find or deduce information about future states and future paths. This

information is then used to make better decisions when traversing the search tree. Technically, informed search associates an evaluation function to each node n, denoted $f(n)$. This value models the quality of the node. Node with a low $f(n)$ value will be preferred when selecting a node in the frontier.

One usually decomposes the evaluation function $f(n)$ in two components:

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the *exact* cost to reach node n from root node with the initial state, and $h(n)$ is an *estimated* cost, called the heuristic function, to reach a goal from node n. When no heuristic is used (i. e. $h(n) = 0$ for all n), the search reduces to an (uniformed) uniform cost search based on $g(n)$.

The objective of the heuristic function is to speed up the search from a node to a goal, by favoring nodes that are expected to be closer to a goal. A heuristic function $h(n)$ is called *admissible* if it never overestimates the real cost to reach a goal. The heuristic is thus optimistic. As a consequence, the value $f(n)$ never overestimates the cost of a solution through the node n. When n is a node with a goal state, $h(n) = 0$ if the heuristic is admissible.

3.2.3.2 The A* algorithm

The integration of a heuristic function leads to two different instances of the generic algorithm where the frontier is organized as a priority queue. In the greedy best first algorithm, the order is based on the heuristic function $h(n)$. In the A* algorithm, the order is based on $g(n) + f(n)$. In the greedy best first algorithm, the objective is to reach a goal node as soon as possible from the frontier. In the A* algorithm, the objective is to find the optimal solution as soon as possible. A* is thus particularly adapted to optimization problems.

Greedy best first algorithm is not complete as it can fall in an infinite branch of the tree when m is infinite. If m is finite, its time and space complexity is $O(b^m)$. It is also not optimal. But this algorithm is often better than DFS thanks to the heuristic function guiding the search.

A* is complete, with a time and space complexity if $O(b^d)$. Contrary to DFS, all the visited nodes must be kept in memory as they may be part of the solution. When the heuristics is admissible, then A* is optimal. It thus always returns an optimal solution. If the cost of the best solution is c^*, then A* will expand each node n with $f(n) < c^*$ but never expand a node n with $f(n) > c^*$. A nonadmissible heuristic can be useful to speed up the search, but A* does not longer grantees to find an optimal solution.

The idea of iterative deepening can also be applied on A*, leading to iterative deepening A* (IDA*). Instead of increasing the depth of the search from one execution of A* to the execution of A*, one increases the maximum value of the evaluation of nodes in

the frontier that are expanded. Technically speaking, at each iteration, A* only expands nodes with an evaluation less than or equal to the smallest value of the evaluation function among all the nonexpanded nodes in the previous iteration. IDA* is complete, its time complexity is exponential (but hard to express) and its space complexity is linear. It is also optimal.

3.2.3.3 Examples

Example 8-puzzle

Different heuristics could be defined for the 8-puzzle problem.

- $h_1(n)$ is the number of misplaced tiles ($h(n) = 4$ in the start state in Figure 3.1).
- $h_2(n)$ is the sum of the Manhattan distance (horizontal + vertical distance) of each tile to its position in the goal ($h(n) = 6$ in the start state in Figure 3.1).

Both heuristics can be computed efficiently and are also admissible. They never overestimate the number of actions required to reach the goal. For any given node n, heuristics $h_2(n)$ will always be greater than or equal to $h_1(n)$. Heuristic $h_2(n)$ is said to dominate $h_1(n)$ as A* will always expand less nodes with $h_2(n)$ than with $h_1(n)$.

Example route planner

Finding the shortest path to travel from one position to another is a classical problem in combinatorial optimization. It can be modeled as finding the shortest path between a source node and a destination node in a weighted graph (**single-pair shortest path problem**), where the cost of an arc(A, B) is the distance (or time) to reach A from B and vice versa. An example is illustrated in Figure 3.10. The shortest path from A to D is *A-B-F-C-D* and has a cost of 18.

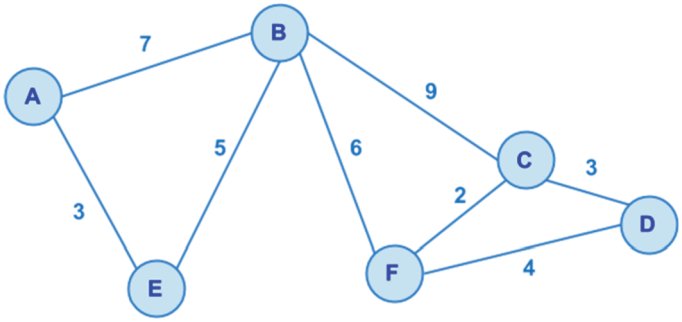


Figure 3.10: Shortest path problem.

The shortest path can be solved by the well-known Dijkstra algorithm. The complexity of this algorithm is $O((n+a)\log(n))$ using a priority queue, where n is the number of nodes and a the number of arcs in the graph. The Dijkstra algorithm actually computes the single-source shortest path problem, that is, finding the shortest path from a source node to all other nodes of the graph. For solving the single-pair shortest path problem, such a complexity can be unaffordable for very large graph as requires at worst to visit all the nodes and arcs of the graph.

The Dijsktra algorithm for solving the single-pair shortest path problem can be seen as the A* algorithm, but with a heuristic function $h(n) = 0$. By adding a heuristic, the performance of the search can be drastically improved. The standard heuristic used in travel routing problem is to use the Euclidean distance between a node of the graph and the goal node as an approximation of the cost of the path from this node of the graph to the goal node. This heuristic is admissible as it never overestimates the real cost: the Euclidian distance is the shortest distance between two points.

3.2.3.4 Designing and comparing heuristics

Designing a heuristic is not a simple task. A heuristic $h(n) = 0$ for all nodes is admissible but does not provide any information. On the other extreme, a heuristic function $h(n)$ yielding the exact distance to a goal is very informative but is computationally too costly as its evaluation requires itself a whole search. The basic idea is to relax the problem, for instance by reducing restrictions on possible actions allowing an efficient computation. In the 8 puzzle, heuristic $h_1(n)$ allows teleportation and $h_2(n)$ allows the superposition of tiles. In the shortest path problem, the heuristic removes the requirement to take roads.

How to compare different heuristics for a given problem? For a given instance of the problem, the shape of the search tree for the two heuristics could be very different and, therefore, not comparable. Comparing the number of expanded nodes in both trees is a possibility, but these numbers are specific to instances. In order to be independent from the instances, one introduces the *effective branching factor*. The idea is to reorganize a search tree into a complete tree in order to make them comparable. The branching factor b* of the complete tree is then the effective branching factor of the initial search tree. If we have a search tree with N nodes, the effective branching factor b* is the solution of the following equation where d is the depth of the goal node:

$$N = 1 + \mathrm{b}^* + (\mathrm{b}^*)^2 + \cdots + (\mathrm{b}^*)^d$$

What is nice is that the effective branching factor is relatively independent from the depth of the solution and comparison can be made between different instances. An effective branching factor of 1 is ideal and means that at each step of the search, the right choice is made.

The effective branching factor of the 8-puzzle without heuristics is around 2.80 but reduces around 1.45 with heuristic $h_1(n)$ and around 1.30 with heuristic $h_2(n)$.

3.2.4 What are the limitations of search algorithms?

Solving by search proposes a series of algorithms that are more similar than different. They are all based on the same generic algorithm; they differ from the order of the nodes

that are visited in the frontier. For solving a specific problem, one has to complete the chosen algorithm by designing a data structure representing a state and by developing methods encoding the goal test and the possible actions (operators) on a state. It thus requires some programming skill.

From an algorithm point of view, if the search is uninformed, IDA is a good choice among the described algorithms in this chapter. When it is possible to exploit a heuristic, A^* is a nice algorithm but has space complexity issue. IDA^* is a possible way, but there exist many other approaches, based on A^*, that offer a better space complexity than A^* (recursive best-first search, simple memory-bounded A^*).

Search algorithms are well suited when there is an initial state, a (set of) goal state(s), and the expected result is a sequence of actions leading to a goal. However, for many optimization problems, the objective is to find a goal and there is no need for a sequence of actions. The cost of a solution is then a function of the state rather than a sum of the cost of actions.

In all the proposed algorithms, the same state can be visited multiple times. This can be avoided by considering graph search instead of tree search. The idea is simple. In the generic tree search algorithm, a list of visited states, called closed, is maintained during the search. The extracted node in the frontier is expanded only if its state has never been visited. The generic graph search algorithm is depicted in Figure 3.11. All the variants of the tree search can also be applied for graph search. The time complexities are similar as they describe worst cases. However, the space complexity of graph search is always exponential as all the visited states have to be recorded.

```
graph_search( problem, frontier) : Node or Failure
        closed := emp yset
        root := Node( problem.initial_state )
        frontier.insert( root )
        while not frontier.is_empty()
                node := frontier.pop()
                if problem.goal_test( node.state )  return node
                if  node.state  not in  closed
                        closed.add( node.state )
                        frontier.insert( expand( node, problem ) )
        return Failure
```

Figure 3.11: Generic graph search algorithm.

When solving a problem by a search algorithm, we need a procedural representation of the problem through a data structure and methods encoding the goal test and the possible actions. A procedural representation explicitly describes how to do or compute things. At the opposite, a declarative representation describes the characteristics and properties of the problem without explicitly proposing a computational way to update data structures not to solve the problem. Query languages to query databases like SQL, are typically declarative, for example. In order to exploit a declarative representation of a problem, one needs a generic problem solver such as constraint satisfaction or SAT solvers, developed in the next two sections. A generic problem solver is an algorithm that is able to solve any problem based on a given representation of the problem.

In an informed search, it is possible to exploit a heuristic, that is, properties of the problem to speed-up the search without changing the search algorithm. The heuristic is coded in a specific and external evaluation function. This approach has two limitations. First, it can only handle properties that can be encoded in a function evaluating the cost from a state to a goal. Second, it can only state problem dependent heuristics and no generic heuristics.

Search techniques only apply to deterministic and known environments that can be modeled by states and atomic actions. Limitations are that the modeling and the heuristics are problem specific, that the modeling might lead to an explosion of the number of states, and that although search techniques are able to perform some "look ahead" prediction through heuristics, but they don't infer nor deduce information, like "reasoning approaches" do.

The next sections will show how these limitations can be lifted and how a generic heuristic can be handled.

3.2.5 Industry example. The fastest route with search and heuristics, using Bing Maps Routes API

Yves Deville, Sanjib Dutta

3.2.5.1 Route planner

A route planner is a tool that finds the shortest path from one location to another location on an existing road network. While it sounds like a very trivial problem to solve, it has many challenges given the size of the road networks and the level of customization needed to support various user choices. For instance, preferences and constraints can be expressed to leave or arrive at a certain time, go through intermediate locations, change the cost function (time, distance, emission, financial), avoid some roads, combine different transport modes, or integrate schedules of public transportation. Route planning services are proposed by many tools and apps such as Google Maps, Route XL, Waze, MapQuest, Bing, and GraphHopper.

Unlike most of the other graph problems, a routing graph is dynamic. The cost of an arc can change dynamically based on the traffic condition at a particular time. A route, which is least cost at 12 AM in the night, may not be the best choice at 7 AM. Reversible road segments also reverse the direction of travel based on the time of the day; some turns also become illegal based on the time. Some roads segments may or may not be traveled based on the user's destination. For example, if the user's destination is a parking lot, the parking lot may be used for routing, but should not be used (even if using it leads to a better cost route) otherwise.

Performance is an important issue when planning a route from one point to some destination point. Because of the size of the road network, a simple 50-mile route may

end up touching millions of edges before finding the optimal route. One of the techniques that is widely used to solve the performance issue is using shortcuts (precomputed small routes) that reduces the number of edges significantly, but shortcuts cannot be used with dynamic graphs or where user preference is considered.

A* with a good choice of heuristics can be used to considerably reduce the number of nodes needed to be examined during the search in a dynamic graph and user preference-based filtering.

3.2.5.2 Bing map routing using A*

Bing routing offers a set of tools to get directions and route information to and from anywhere. It provides an application programming interface (API) called Bing Maps Routes API to perform different tasks, including various routing problems. Bing router uses Euclidean distance (great circle distance) to the target location for the heuristic function to determine the cost of a node that is being explored next. To calculate the time required to travel this estimated distance, various options can be used. Examples are (i) incoming road speed, (ii) a fixed road speed based on the region (max speed in that region). Issues with (i) is that highways get prioritized too much over local roads and there is a risk that many legitimate local paths may not be explored. On the other hand, (ii) does not prioritize highways over the local roads. Bing map routing exploits a refined version of (ii) using road hierarchy. A higher hierarchy road will be given a little more preference unless it is too near to the source or the destination. Assuming there are 5 levels of road hierarchies, with hierarchy 1 higher than hierarchy 2, penalties can be assigned to lower hierarchy roads gradually, such as

$$\text{hierarchyModifiers} = [1.0, 1.05, 1.2, 1.3, 1.4].$$

Given a node n, the heuristic $h(n)$, that is, the estimated (time) cost to travel from n to the destination node is computed as follows. The estimated cost is first computed by dividing the Euclidian distance to the destination by a given fixed speed. If the node n is not too near to the source or the destination, this estimated cost is then multiplied by the hierarchy modifier corresponding to the road used to lead to node n in the search. In the above example, the estimated cost of nodes reached by a road with hierarchy 3 will be multiplied by 1.2.

This routing approach is illustrated in Figure 3.12 with a partial graph showing connected roads with two road hierarchies and length of the arcs in meter.

- Green: Hierarchy 3, Speed 36 km/h
- Purple: Hierarchy 1, Speed 54 km/h

The starting node is n3 and the destination node is n14. Table 3.1 shows the Euclidean distance between each node and the destination node n14. For each hierarchy level, the

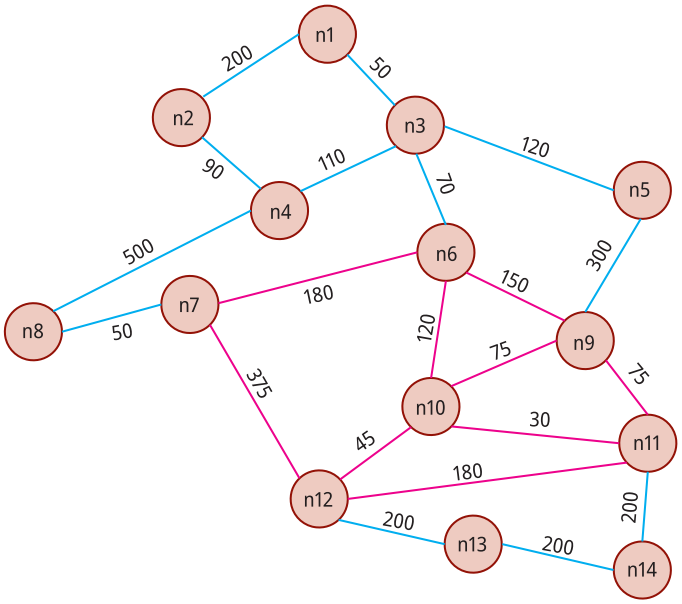


Figure 3.12: Finding a route from node n3 to n14.

Table 3.1: Table of distances and heuristics of nodes to destination n14.

	n1		n2		n3		n4		n5		n6		n7	
Dist	**780**		**780**		**350**		**300**		**250**		**250**		**280**	
Hrchy	1	3	1	3	1	3	1	3	1	3	1	3	1	3
$h(n,.)$	52	62	52	62	23	28	20	24	17	20	17	20	19	22

	n8		n9		n10		n11		n12		n13	
Dist	**300**		**220**		**220**		**200**		**220**		**200**	
Hrchy	1	3	1	3	1	3	1	3	1	3	1	3
$h(n,.)$	23	28	15	18	15	18	13	16	15	18	13	16

estimated cost is also provided. The used fixed speed here is 54 km/h. So, for node n1, we get the following values for $h(\text{n1}, i)$, expressed in seconds, where i is the hierarchy of the arc leading to node n1 in the search.

– $h(\text{n1}, 1) = 780\,\text{m}/54\,\text{km/h} * 1.0 = 52\,\text{sec}$
– $h(\text{n1}, 3) = 780\,\text{m}/54\,\text{km/h} * 1.2 = 62\,\text{sec}$

In this small example, hierarchy modification will be applied whatever the distance of the node to the source and the destination.

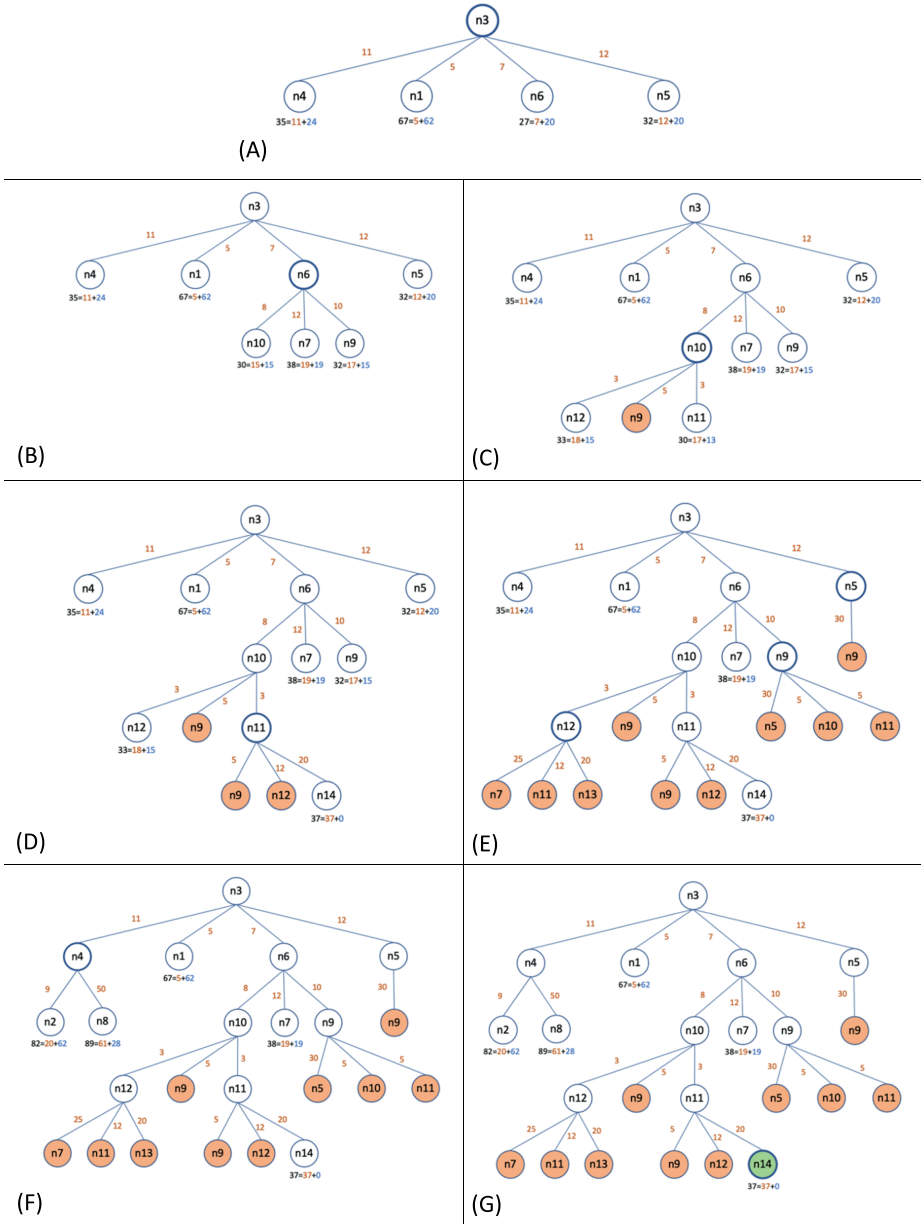


Figure 3.13: Steps in the search tree for a route from n3 to n14.

Figure 3.13 depicted the search tree in the different steps of the A* algorithm on the above graph to find an (optimal) route from n3 to n14. The frontier is initially the node n3.

- (A) Node n3 is extracted from the frontier. It is expanded and its successors nodes (n4, n1, n6, and n5) are added in the search tree and inserted in the frontier. Each node in the frontier is associated with the value of the evaluation function $f(n) = g(n) + h(n, i)$ where $g(n)$ is the *exact* cost to reach node n from root node n3, and $h(n, i)$ is an *estimated* cost described by the above heuristic function. For node n6, the values are computed as follows:
  - $g(\text{n6}) = 70\,\text{m}/36\,\text{km/h} = 7\,\text{sec}$.
  - $h(\text{n6}, 3) = 20$ as the hierarchy of the arc n3-n4 is 3
  - $f(\text{n6}) = 27 = 7 + 20$
- (B) Node n6 is extracted from the frontier as it has the smallest evaluation. It is expanded and its successor nodes (n10, n7, n9) are added in the search tree and inserted in the frontier. For node n6, the values are computed as follows:
  - $g(\text{n10}) = 120\,\text{m}/54\,\text{km/h} = 8\,\text{sec}$.
  - $h(\text{n10}, 1) = 15$ as the hierarchy of the arc n6-n10 is 1
  - $f(\text{n10}) = 27 = 7 + 8 + 15$
- (C) Node n10 is extracted from the frontier and expanded. The successor node n9 with an evaluation of $35 = 20 + 15$ is, however, not added in the frontier as this node is already in the frontier with a smaller evaluation (32). This is a basic optimization of the presented A* algorithm.
- (D) Node n11 is extracted from the frontier and expanded. The successor nodes n9 and n12 are not added in the frontier as they are already in the frontier with a smaller evaluation. Node n14 is added with a cost $37 = 37 + 0$. This is a goal node, but the search continues as other nodes in the frontier with a smaller evaluation could perhaps lead to n14 with cost smaller than 37.
- (E) Nodes n5, n9, and n12 are successively extracted from the frontier. None of their successor nodes are, however, added in the frontier as they are already in the frontier with a smaller evaluation.
- (F) Node n4 is extracted from the frontier and replaced by its successors, nodes n2 and n8.
- (G) Node n14 is extracted from the frontier. It is recognized as a goal node and the search returns the path n3-n6-n10-n11-n14 with a cost of 37.

3.2.5.3 Key challenges

On large graphs, the A* algorithm explores significantly a smaller number of edges than other algorithms such as Dijkstra. It, however, has a number of caveats.

A* is not suited for routing problems involving constraints such as combining different transport or integration of public transport schedules.

The above routing algorithm uses a fixed speed to determine the estimated remaining cost. If this speed is an upper bound of the maximum speed, the heuristic (without hierarchy) is admissible and A* always returns the best solution. When using a fixed

speed lower than the maximum speed or introducing hierarchy modifiers, the heuristic is not always admissible as the estimated cost could be greater than the real cost. However, these cases are very rare and generally can be mitigated by continuing the search for some more time to check if better results exist.

3.3 Constraint satisfaction problems

When a state can be represented with a set of variables, each with a set of possible values, we move from an "atomic" representation to a richer "factored" representation that can take advantage of the structure of states and their constraints. This eliminates large portions of the search space and reduces the possible set of values by identifying variable/value combinations that violate the constraints. The SEARCH approach described in the previous section can then be combined with some form of INFERENCE based on the CONSTRAINTS. So, CSP solving systems can be more efficient than pure "state space" searchers as they take advantage of the structure of states.

3.3.1 What category of problems does constraint satisfaction solve?

A constraint satisfaction problem (CSP) is a problem that can be described by a set of (decision) variables, usually over a finite domain, and a set of constraints over the variables. More specifically, constraints specify the legal values and combination of values assigned to the variables: they can restrict the value of a single variable (un-ary), pairs of values (bin-ary), or more generally t-uples of values. Global constraints like "AllDifferent" can also be specified.

A solution of the problem is an assignment of values to the variables satisfying all the constraints.

A constraint optimization problem (COP) is a CSP with an additional objective function relating the variables. The goal is then to find a solution maximizing (or minimizing) the objective function.

The techniques for solving CSP and COP depend on the computation domain of the variables. Most approaches are handling finite domains, that is, each variable of the CSP has a finite domain. A particular case of finite domain is the domain $\{0,1\}$, leading to Boolean CSP. Extensions of finite domains are finite sets and graphs, that is, where the value of a variable is a finite set (from a given initial finite set), or a graph (subgraph of an initial given graph). All these finite computation domains can be used to model combinatorial problems. Another class of problems are continuous CSP where the value of a variable is a real and has an infinite domain.

The description of a CSP or COP is a declarative representation of the problem. There are no data structures nor any algorithm in the description. CSP and COP cover a large class of problems such as vehicle routing, scheduling, warehouse location, planning, knapsack, configuration problems, and graph coloring.

3.3.1.1 Examples

Graph coloring

Given a graph and a set of colors, color the vertices of the graph so that two adjacent vertices are colored with different colors, as illustrated in Figure 3.14. This classical graph coloring CSP is known to be NP-complete, that is, a problem for which it is easy to check whether a given assignment of colors to the vertices is a solution, but finding a solution is very hard.

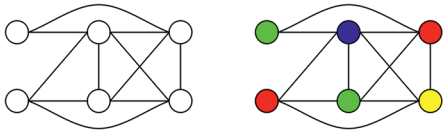


Figure 3.14: A solution of a graph coloring problem.

More formally, given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, and a set of colors C, the (decision) variables are the vector colour$[v]$, with $v \in V$. The domain of each variable color$[v]$ is the set of colors C. For each edge $(v, w) \in E$, we have the constraint color$[v] \neq$ color$[w]$.

The corresponding graph coloring COP is to find the minimum number of colors necessary to color the vertices of the graph so that two adjacent vertices are colored with different colors.

Graph coloring can be used to model various practical and industrial problems such as scheduling problems, pattern matching, or course schedule.

Cryptarithmetic puzzle

A classical cryptarithmetic puzzle is the SEND + MORE = MONEY problem, where each letter has to be replaced by a different digit so that this addition is correct, as illustrated in Figure 3.15. The unique solution is 9567 + 1085 = 10652. A possible model for this puzzle is to use the set of variables $\{\mathrm{S}, \mathrm{E}, \mathrm{N}, \mathrm{D}, \mathrm{M}, \mathrm{O}, \mathrm{R}, \mathrm{Y}\}$ with domain $\{0, \ldots, 9\}$, and the additional four carry variables $\{C_1, C_2, C_3, C_4\}$ with domain $\{0, 1\}$. A constraint

$$
\begin{array}{rcccc}
 & C_1 & C_2 & C_3 & C_4 \\
 & S & E & N & D \\
+ & M & O & R & E \\
\hline
M & O & N & E & Y
\end{array}
$$

Figure 3.15: Cryptoarithmetic problem.

allDiff(S, E, N, D, M, O, R, Y) states that all the variables must have a different value. The constraints for the additions are then modeled as

$$
\begin{aligned}
& S \neq 0; \quad M \neq 0 \\
& D + E = 10C_1 + Y \\
& C_1 + N + R = 10C_2 + E \\
& C_2 + E + O = 10C_3 + N \\
& C_3 + S + M = 10C_4 + O \\
& C_4 = M
\end{aligned}
$$

Sudoku

Sudoku is a classical puzzle played on a 9×9 grid, divided in 9 squares made up of 3×3 cells. Each cell must be filled with a digit 1–9 such that each row, column and square, each having 9 cells, must have different digits. Some of the cells are given a specific value. An example is given in Figure 3.16, where the small digits are the possible values for the cells.

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

Figure 3.16: Sudoku.

The model of a sudoku has one variable per cell, that is, 81 variables, each with a domain $\{1, \ldots, 9\}$. There is an allDiff constraint for the variables of each row, of each column, and each square. The allDiff(X) constraints all the variables in X to be different. Then there are constraints imposing the specific value for the given cells of the instance of the sudoku.

3.3.1.2 Solving paradigms

There exist many approaches to solve CSP and COP. Classifying the different paradigms is helpful to their understanding and comparison. Two classification criteria will be used. The first one differentiates *perturbative* versus *constructive* approaches, while the second one distinguishes systematic versus incomplete search.

In a *perturbative* approach, a candidate solution is a complete solution (a value is assigned to each variable). New solutions are generated by modifications or perturbations of an existing one. The search is performed on a succession of candidate solutions. This is illustrated on graph coloring in Figure 3.17 where at each step, the color of one of the vertex is modified.

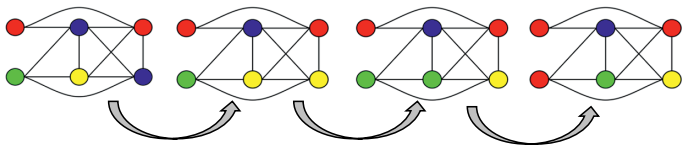


Figure 3.17: Graph coloring: perturbative approach.

In a *constructive* approach, a candidate solution is a partial solution (a value is assigned to some of the variables) and partial solutions are progressively extended. This is illustrated on graph coloring in Figure 3.18 where at each step, one new vertex is colored.

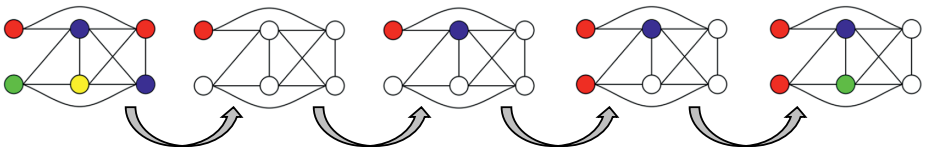


Figure 3.18: Graph coloring: Constructive approach.

In a *systematic* approach, the whole search space is systematically explored. As a consequence, the approach is complete; it finds an exact solution for a CSP and find a best solution for a COP. If no solution is found, this is a proof of nonexistence of solutions.

Of course, a systematic approach can be computationally expensive for large instances of problems.

In an *incomplete* approach, the exploration of the search space is based on a neighborhood function. The approach is incomplete in the sense that it may not find an exact solution of the CSP, but only an approximation of a solution where some of the constraints might be violated. For a COP, it will find a good solution that is not proved to be the optimal solution. An incomplete approach is computationally less expensive and usually allows us to find a reasonable (approximated) solution for large instances of problems.

Different paradigms for solving CSP and COP can be classified according to the above two criteria, as shown in Table 3.2. In this chapter, two paradigms will be developed: local search and constraint programming.

Table 3.2: Solving paradigms for CSP and COP.

	Systematic	**Incomplete**
Perturbative	Simplex	Local search
	Systematic local search	Evolutionary algorithms
Constructive	Dynamic programming	Greedy
	Branch and bound	Ant colony optimization
	Constraint programming	GRASP
	DPLL SAT solver	

3.3.2 Solving optimization problems by local search

Local search algorithms address problems for which only the solution state matters, not the sequence nor the path cost to reach it. In addition to finding a goal state, they are used to solve optimization problems, for which the objective is to find the values that minimize a cost function.

Local search covers a family of methods sharing common properties. The approach is incomplete and perturbative. It is based on iteratively improving the solution by finding a better one in the neighborhood of the current solution. When it stops, there is no proof the given solution is the best one.

3.3.2.1 From CSP to COP

In a CSP, one has to find a solution, that is, an assignment of the variables satisfying all the constraints. There is no given objective function to optimize. As local search is an optimization process, the CSP should first be transformed into a COP by introducing a cost function. The set of constraints is split in two sets: the hard constraints and the

soft constraints. The set of hard constraints that can be emptycontains the constraints that must be satisfied by all the solutions considered by the local search. For each soft constraint, a cost or violation function measures how a given solution violates the constraint. The cost of a solution is then computed as the sum of the violation cost of all the soft constraints. The resulting objective function of this COP, usually called cost function in local search, is then used in the search for the best solution. When the cost of a given solution reaches 0, then it is a solution of the original CSP. If the search stops with a no zero cost solution, then the resulting solution is only an approximation as it still violates some of the soft constraints.

When the initial problem is a COP, the cost function used for the local search should combine the objective function of the COP and the violation function of the soft constraints, if any. The cost function usually has then the form,

$$\text{cost}(x) = \alpha.\text{obj}(x) + \beta.\text{cost_soft}(x),$$

where x are the decision variables, $\text{obj}(x)$ is the objective function of the COP, and $\text{cost_soft}(x)$ is the cost of the violation of the soft constraints and α, β are real values. Such a cost function combines optimality and feasibility. Usually, the β value is much higher than α to favor feasible solutions.

The iterative improvement of a solution needs to start from some initial solution. Such an initial solution is usually chosen randomly but respecting the hard constraints of the problem.

Graph partitioning

Given a graph with an even of vertices, divide the vertices in two equal-size sets of vertices such that the number of edges that go from one set to the other set is minimized as illustrated in Figure 3.19. More formally, given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, find a partition (V_1, V_2) of V such that $\#V_1 = \#V_2$ (balance constraint). The objective is to minimize $\text{obj}(V_1, V_2) = \#\{(v, w) \in E | v \in V_1 \wedge w \in V_2\}$. This problem is known to be NP-hard.

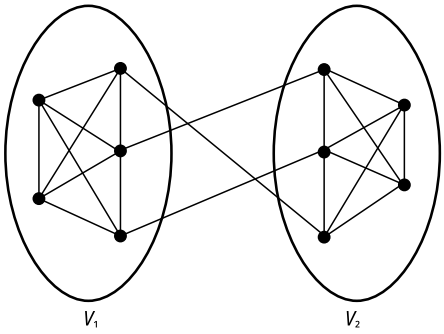


Figure 3.19: Graph partitioning: Example of a solution. The cost of this solution is 3.

3.3.2.2 Neighborhood

The neighborhood of a solution p is a set of solutions that are similar or close to p. The idea of proximity is usually measured in terms of computation. A solution in the neighborhood can thus easily be computed from the solution p. A neighborhood can be defined by one or more operators over a solution and yielding a set of solutions. Such operators must be computationally simple.

An important property of a neighborhood is called *connectivity*, stating that from each solution, there is a path to an optimal solution. This path is a succession of solutions p_1, p_2, etc. where each solution p_{i+1} is in the neighborhood of p_i. Of course, large neighborhoods lead to shorter paths to an optimal solution, but a large neighborhood needs more time to be explored to choose the next solution.

If the initial problem has constraints on the solution, should these constraints be respected by each solution in the neighborhood? If these constraints are seen as hard constraints, then the operators generating the neighborhood must be designed to fulfil the constraints. If these constraints are seen as soft constraints, then the neighborhood are larger as it may contain solutions violating these constraints, but the cost function would integrate the violation of these soft constraints.

Graph partitioning

In the graph-partitioning problem, if the balance constraint is hard, then the neighborhood of a (balanced) solution must only contain balanced solutions. A simple neighborhood could be defined by swapping two vertices in V_1 and V_2. If the balanced constraint is considered as a soft constraint, then a possible neighborhood would be to move a vertex from one set to the other set of vertices. The cost function should then integrate the cost of the soft constraint. In the following cost function for graph partitioning, the soft balance constraint measures the (square of the) distance of the solution to a balanced solution:

$$\text{cost}(V_1, V_2) = \alpha.\#\{(v, w) \in E | v \in V_1 \wedge w \in V_2\} + \beta.(\#V_1 - n/2)^2$$

These two neighborhoods are illustrated in Figure 3.20. The choice between a hard or soft constraint for a constraint of the problem is a design decision and is highly problem dependent. Choosing a soft constraint may reduce the length of the path to an optimal solution may drastically enlarge the search space.

3.3.2.3 Heuristics and metaheuristics

In local search, a heuristic defines how to choose the next solution in the neighborhood of the current solution. A heuristic uses local information, that is, the current solution

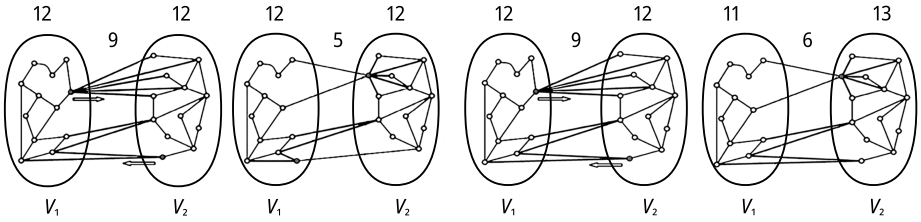


Figure 3.20: Graph partitioning: Solution in two definitions of neighborhoods.

and its neighborhood. A first property of a heuristic is whether it always improves the cost of the current solution or is it possible to have a degradation. A second property relates to the search of the neighborhood; is it an exploration or is it a random choice? The rationale behind improvement is to focus the search toward high-quality solution, what is called *intensification*. The rationale behind randomness and degradation is to direct the search toward other regions of the search space, what is called *diversification*. Different strategies to balance intensification with diversification have led to different types of heuristics. Some classical heuristics are presented hereafter.

Hill climbing explores the neighborhood and always improves the cost of the current solution. *Best-improvement* is hill climbing where the whole neighborhood is searched for the best solution while *first-improvement* stops as soon as a better solution is found. If no solution in the neighborhood improves the current solution, the local search is stopped to the current (local) optimum. Best-improvement is more costly because of the complete search of the neighborhood.

Random walk selects the next solution randomly in the neighborhood; the cost can therefore be degraded. *Random improvement* randomly selects a candidate solution; it is, however, accepted only if it improved the current solution. Otherwise, the current solution is unchanged. The *Metropolis* heuristic randomly selects a candidate solution. If it improves the current solution, the candidate solution is accepted as the new current solution. If it degrades the current solution, it is accepted with a small probability, usually depending on the severity of the degradation: a large degradation has a much smaller probability to be accepted.

While a heuristic focuses on choosing the next solution, a metaheuristic guides the search process toward a global optimum, hence trying to escape from local optimum. This usually collects information during the local search. Many metaheuristics have been defined for local search.

Simulated annealing is an analogy with metallurgy where heating causes molecules to move more freely to unusual location. During cooling, the movements are more restricted. In this metaheuristic, a temperature variable is reduced during the local search. This temperature is used in the Metropolis heuristic applied at every step; with high temperatures, the probability to accept the candidate solution is higher than with a low temperature. In the beginning of the local search degradation will be more often ac-

cepted, favoring diversification while at the end of the local search degradation will be less accepted, favoring intensification.

The idea of the *tabu search* metaheuristic is to select a neighbor that has not yet been visited. As it is difficult to keep track of all the visited solutions, tabu search maintains an abstraction of the last visited solutions. Such an abstraction can be based on the chosen operators or some properties of the solution. An evolving tabu list states criteria to reject candidate solutions. If the candidate solution is in the tabu list, then it is rejected. The abstraction of tabu list does not limit the rejection to already visited solutions. Tabu search often maintains the best solution found so far as the chosen heuristic may degrade the current solution. Tabu search is then often combined with an *aspiration* meta-heuristics overriding the tabu rejection if the candidate solution improves the best solution found so far.

In a *variable neighborhood metaheuristic*, a sequence of neighborhoods (of increasing size) is used during the local search. Guided *local search* or *dynamic local search* metaheuristics use a sequence of cost functions to escape from local optimum. *Adaptive local search* provides a more general framework where heuristics and metaheuristics are dynamically adapted during the search.

3.3.2.4 Termination

Local search is incomplete. It is impossible to know if the current solution is the best solution nor how far its cost is from the optimum. But whenever the search is stopped, a solution is provided. Termination is often defined through a time limit.

Some metaheuristics have their own termination criteria. In simulated annealing, for instance, termination occurs when the temperature reaches zero, according to a temperature schedule.

In order to favor diversification, it is interesting to do multiple local searches from different random initial solutions. That is what *Random restart* is doing, returning then the best solution. *Iterated local search* favor intensification by iterating the local search from a random initial solution close to the solution found in the previous iteration.

3.3.3 Solving problems by constraint programming

3.3.3.1 CSP as a search problem

A CSP can be solved using a search algorithm, such as depth-first search. The initial state is an empty assignment. The successor function assigns a value to some unassigned variable provided no constraint is violated. The goal test is to have all the variables assigned.

Constraint programming (CP) offers a general framework for solving CSP and COP. It can be seen as a general-purpose search algorithm, independent from the problem.

It can be applied to any CSP and COP. CP also provides generic heuristics. In practice, a CP language allows to state a model for a CSP or COP and proposes mechanisms to guide and enhance the search for a solution. Examples of CP languages are Choco, IBM cp optimizer, Gecode, OR-tools, Oscar, and JaCoP. SAS/OR and Cali Xpress also integrate CP modules.

The CP computation model is based on two steps. The *propagation* step reduces the domains of the (decision) variables while the *search* step decomposes the problem into simpler subproblems. These two steps are iterated until a solution is found or until the problem is proved to have no solution.

3.3.3.2 Propagation of the constraints

When the domain of a variable is modified, the objective of propagation is to use this information to (quickly) reduce the domain of the other variables. Propagation is made by applying a consistency algorithm that considers the constraints locally instead of considering all the constraints of the problem. The propagation aims at achieving some level of consistency. Different levels can be considered, each level offering more pruning than the previous level. Of course, a higher level of consistency requires more computation.

Any consistency algorithm must respect two properties. First, it never removes a value of a variable that is part of a solution of the CSP. This ensures that all the solutions will be found. Second, if the consistency algorithm is applied on a CSP where all the variables are assigned, then the consistency algorithm reduces to a satisfiability test. This ensures that when all the variables are assigned, the resulting assignment is a solution of the CSP.

Forward checking

When the search procedure assigns a value to a variable, forward checking only considered constraints involving this variable, but where all but one variable (say Y) is assigned to a value. Forward checking then removes all the values from the domain of Y that violate the constraint. It is clear that the removed value cannot be part of the constructed solution. Forward checking is illustrated on graph coloring in Figure 3.21.

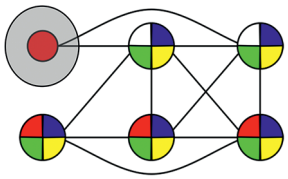


Figure 3.21: Forward checking in graph coloring. The value “red” is removed from the two neighbors to respect the constraints.

Domain consistency

Domain consistency, also called arc consistency, is easier to explain on a binary constraints $c(X, Y)$. Even if none of the variables are assigned, it is possible to remove values from the domain of X and Y. Domain consistency removes any value from the domain of X for which there is no value in the domain of Y that satisfies the constraint, and similarly for the domain of Y. It is clear that all the removed values cannot be part of the constructed solution. When all these impossible values have been removed, the constraint is said to be domain consistent. More precisely, a binary constraint $c(X, Y)$ is domain consistent when for each value a in the domain of X, there exists a value b in the domain of Y such that $c(a, b)$ holds. Domain consistency is illustrated in Figure 3.22, on the binary constraint $X \geq Y + 2$.

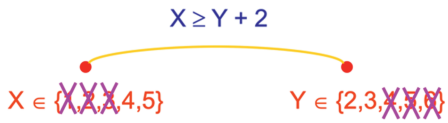


Figure 3.22: Domaon consistency for $X \geq Y + 2$.

Bound consistency

Bound consistency is a weaker form of domain consistency where the domain of a variable is approximated by an interval [min Domain(X), max Domain(X)]. Only the bounds of the domains are verified to be part of a solution. Values are thus never removed in the middle of the domain; only the bounds are moved. The pruning is weaker but for some constraints, such as arithmetic constraints, this pruning can be achieved very efficiently.

Fixpoint computation

Propagation requires iterating the process until no more value can be removed. This process is known as a fixpoint computation. If we consider the propagation achieved by domain consistency on the constraint $X \geq Y + 2$ in Figure 3.22, the domain of variable X (and also Y) has been modified. All the other constraints involving X should then be (re)considered because the removal of value $\{1, 2, 3\}$ in the domain of X could imply the removal of values in the domain of another variable. When no more value can be removed, the fixpoint is reached and the CSP is domain consistent, that is, all its constraints are domain consistent, assuming domain consistency is the chosen level of consistency. A consistency algorithm is an algorithm achieving some consistency for CSP.

Global constraint

A global constraint such as allDiff($X_1, \ldots, X_n$), is a constraint with a variable number of arguments. This is illustrated in Figure 3.23, on one of the 9 squares of a sudoku. The left part is a given state where 2 variables (X_4 and X_9) are assigned, and the others have a nonsingleton domain. The constraint on this square is that these variables must have a different value. This can be modeled by a set of thirty-six $X_i \neq X_j$ constraints. In that case, the left state of Figure 3.23 is domain consistent as any of these inequalities is domain consistent. However, when these constraints are modeled by an allDiff($X_1, \ldots, X_9$) constraint, the left state of Figure 3.23 is not domain consistent. For instance, it is not possible to find an assignment of

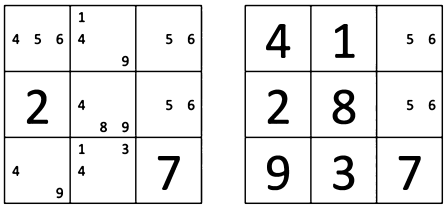


Figure 3.23: The allDiff global constraint on a sudoku square.

these nine variables with $X_1 = 5$ satisfying this allDiff constraint because the values 5 and 6 must be taken by the variables X_3 and X_6. Similarly, the values 4 and 9 must be pruned from the domain of X_5 because the value 8 occurs only in the domain of this variable. The right part of the figure is the corresponding domain consistent state. The achieved pruning is very effective.

In the case of allDiff, the constraint can be expressed as a set of simpler constraints, but as a global constraint, it achieves a much better pruning. Of course, the computational complexity is higher. Other global constraints cannot be expressed with simpler constraints, and thus increases the expressivity of the constraint language. Each global constraint requires a dedicated algorithm to achieve the expected consistency level. For the allDiff constraint, the algorithm achieving domain consistency is based on matching theory in bipartite graphs.

3.3.3.3 Search

The search implicitly develops a search tree. Each node of the search tree contains a CSP that only differs from the initial CSP by its domain of values. The root contains the initial CSP. Search and consistency checks are interleaved. At a given node, the search algorithm first applies a consistency algorithm. If the domain of a variable is empty, then the CSP of this node has no solution and is a failed node. If all the variables are assigned, this assignment is a solution of the CSP and the node is a solution node. In any other case, the search algorithm chooses a variable with a nonsingleton domain and creates children nodes for the different values of the chosen variable. The CSP in a child node is a copy of the CSP of the parent node, but with one new variable assigned. The search tree is finite and is usually traversed using a depth-first strategy. An example of (part of) a search tree for the graph coloring CSP is depicted in Figure 3.24. The root is the initial CSP where each of the six variables has four colors in its domain. In a child node, the chosen variable and its chosen value (its color) are highligthed. In the child node of the root, the top left variable has been chosen and set to the red color. The domain consistency algorithm is then applied and removed the red color from its two neigbhors variables. At the left bottom of the tree, two nodes are failed nodes as they contain a CSP with variables with an empty domain. The bottom right node is a solution node as all variables are assigned.

The basic search algorithm illustrates that a CP search is made of two components: the definition of the search tree and the use of an exploration strategy for traversing the

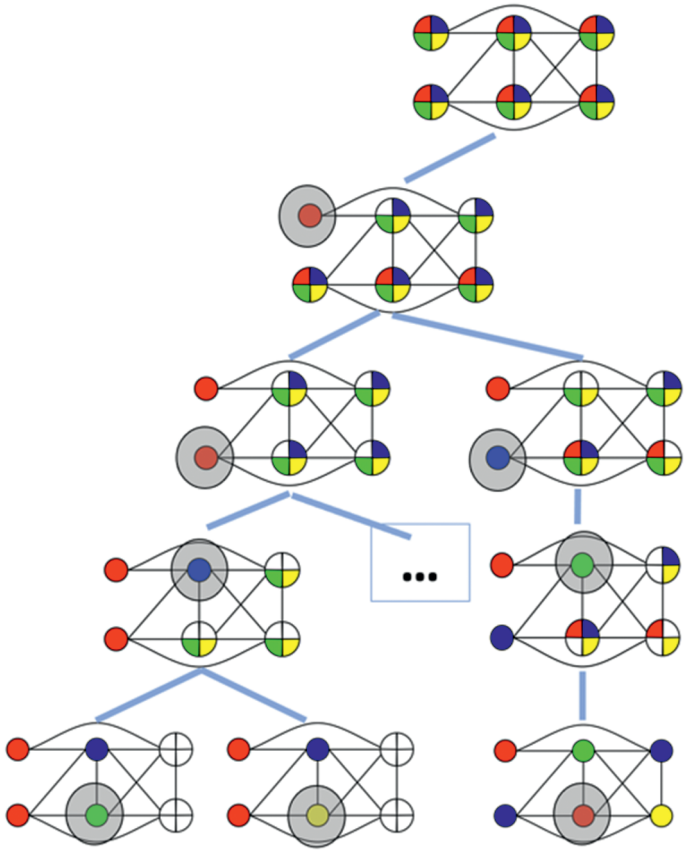


Figure 3.24: Search tree for the graph coloring CSP.

search tree. The different exploration strategies defined for searching problems could be used in this context. DFS is the standard approach, although best-first search can be applied using some measure of quality of the nodes in the fringe.

The definition of the search tree is induced by the branching strategy, that is, how are generated the children nodes of a given node. In the standard branching strategy, called labeling, a variable is chosen and there is a child for each of the value in the domain of that variable. There exists various heuristics guiding the choice of the variable to label. The *domain* heuristic chooses the variable with the smallest domain as such a choice minimizes the size of the search tree. The *degree* heuristic chooses the variable involved in the largest number of constraints. Such a choice will quickly drive the search to a fail node if the CSP has no solution.

Optimization problems
For COP, that is, CSP with an objective function $f(x)$ defined over the variables of the CSP, a branch-and-bound approach can be used to find an optimal solution. As soon as a solution (with value f* for the objective function) is found, this solution is stored as the current best solution and an additional constraint $f(x) < f^*$ is added to the CSP. With this new constraint, the only solution improving the value of the objective function will be found, thus pruning the search space from nonoptimal solution. When a better solution (with value f^{**}) is found, then it replaces the best current solution and the additional constraint is replaced by the constraint $f(x) < f^{**}$, and so, for each better solution found. When the search is completed, that is, it is not possible to find yet another better solution, then the current best solution is known to be the best solution of the COP.

Large neighborhood search
CP is a systematic approach. It is thus complete but requires exploring the whole search space what can be very costly, especially for COP where proof of optimality is not always necessary. Large neighborhood search (LNS) is a local search metaheuristic that can be applied to many frameworks, including CP. This approach is incomplete but orients the search in promising areas of the search space. LNS iteratively applies two steps: relaxation and reconstruction. Relaxation relaxes parts of the best solution found so far, that is, some random variables are chosen to become uninstatiated. The relaxed CSP is simpler than the original one and contains parts of the best solution, and thus looks promising. The reconstruction attempts to solve the relaxed CSP obtained. If the reconstruction finds a better solution, a new iteration is done. If the reconstructions fail to find a better solution in a fixed amount of time, then a new iteration begins from former solution with a different relaxation.

3.3.4 What are the limitations of constraint satisfaction problems?

CSP and COP offer a declaration description of problems. They can then be solved by different techniques. A systematic approach ensures to find a solution if one exists or to prove there is no solution. It is, however, computationally more expensive. An incomplete approach is computationally less expensive but does not ensure the finding of a solution.

Thanks to its perturbative and incomplete approach, local search always provides an (approximated) solution, whatever the chosen computation time. For many problems, an approximate solution is often sufficient, and certainly better than having no solution at all. Local search usually does not provide an estimation of the distance from the cost solution to the optimal solution. Local search requires designing and programing the neighborhood and the cost function for the soft constraints. Solving a CSP by local search usually requires some significant tuning of the parameters of the chosen heuristic and metaheurisfic. Local search covers a large variety of methods, each of them having its advantages on different classes of problems. All local search methods are based on randomness, a key feature in incomplete approaches.

Constraint programming (CP) is constructive and perturbative, hence always provides a solution if one exists. CP only requires the description of the CSP or COP and does not require any programming. CP also proposes generic heuristics to speed up the

search. Solving practical CSP with CP could however require some tuning in the modeling of constraints and in the search procedure.

Local search and constraint programming are only two of the various methods for solving CSP and COP but illustrate various interesting features and concepts.

3.3.5 Industry example: containers configuration using CSP at Maersk Container Industry A/S

Yves Deville, Karen Veldeman, Dennis Conrad

3.3.5.1 Product configuration

The need to configure products to meet special and custom requirements is becoming increasingly important for companies in different industries. A product configuration tool is used to guide a user to configure a product or process, while considering different constraints. Although product configuration tools are used in different industries, most manufacturers and automotive producers rely on them to support them in creating the right products or processes. They are often employed as design tools that allow manufacturers to constrain product lines or even end customers to codesign their own products. Once the product specifications and constraints are modeled by a subject matter expert into a product configurator, this configurator guides an employee or a customer to express his demand by proposing favorable combinations of attributes and ruling out impossible ones. It thereby removes the need for the expert to be in the specific product configuration process, while ensuring compliance with the specifications. For many manufacturers, relying on product configurators helps them to lower their distribution costs, accelerate their production, react faster to customer inquiries, and reduce overproduction of general products.

Product configurators exist in many shapes and forms. Depending on the reasoning techniques used, they are usually classified as rule-based or constraint-based systems. Rule-based configurators are used in the earliest systems and derive solutions in a forward-chaining manner, by evaluating all rules and eliminating rules that are not applicable to the configuration of the product. This approach anticipates all possible configurations by specifying the configuration using restrictive “if... then...” statements. By using constraint-based systems, companies can build product models composed of modular entities.

Use case

Maersk Container Industry A/S (MCI) manufactures containers. At MCI, a product configurator is used to translate customer specifications into a container configuration. For example, a customer can choose the type of condensers to control the temperature within



Figure 3.25: Example of a MCI container with built-in cooling system.

the container, the type of battery pack or the type of panels outside the container. An example of a MCI container is illustrated in Figure 3.25. Based on the specification of these attributes, the configured container can serve as input to generate the quote that will be shared with the customer, as well as a bill-of-materials (BOM) of raw materials and components that will be shared with the Maersk production department. The MCI product configurator is developed in Microsoft Dynamics 365 Supply Chain Management, an enterprise resource planning (ERP) system, featuring a constraint-based product configuration tool based on Z3, a solver and theorem prover, open sourced by Microsoft Software Research in 2015.

3.3.5.2 Modeling product configuration as a CSP

The need to configure products to meet special and custom requirements is becoming increasingly important for companies in different industries. A product configuration tool is used to guide a user to configure a product or process, while considering different constraints. The definition of the underlying product configuration model has been designed by product experts. They first defined product attributes (the variables of the CSP) and their domain of possible values. For example, the Maersk container attributes integrate the type of condensers to control the temperature within the container, the

Table 3.3: Some variables and their domain in the MCI product configurator.

Variables (attributes)	Domain
Atmosphere control (AC)	{Manual ventilation (MV), Automatic ventilation (AV), Automatic ventilation (AV+), Automatic ventilation (AV+O2)}
Fresh air module (FAM)	{Manual ventilation (MV), Automatic ventilation (AV)}
Sensor module (SM)	{No sensors (No), CO2 sensors (CO2) CO2 and O2 sensors (O2)}

type of battery pack, the type of fresh air exchange module or sensor module, as illustrated in Table 3.3.

After defining the product configuration variables and their domains, the experts need to define how these attributes can be combined into one product of service by defining constraints. These constraints can make sure that a user can only configure containers that can be sold and produced. For example, not all types of fresh air exchange modules and sensor modules can be combined into one single container. These constraints are expressed in the form of a table expressing the valid combinations between one or more attributes. Each column of the table is assigned to a variable of the CSP. Each row defines a valid combination of the values for the variables. Other constraints can also be specified in the CSP. For example, the MCI configurator uses a table constraint illustrated in Table 3.4 to express that some combinations of fresh air exchange modules and sensor modules cannot be combined into one single container.

Table 3.4: A table constraint in the MCI product configurator.

Atmosphere control (AC)	Fresh air module (FAM)	Sensor module (SM)
Manual ventilation (MV)	Manual ventilation (MV)	No sensors (No)
Automatic ventilation (AV)	Automatic ventilation (AV)	No sensors (No)
Automatic ventilation (AV+)	Automatic ventilation (AV+)	CO2 sensors (CO2)
Automatic ventilation (AV+O2)	Automatic ventilation (AV+O2)	CO2 and O2 sensors (O2)

The constraint in this example ensures that a user of the product configurator cannot assemble a container with automatic ventilation (AV+) but without CO2 sensor. Once the user has chosen to configure a container with automatic ventilation (AV+), the user will only be able to select the automatic ventilation and CO2 sensor option. The configurator leverages the constraint solver to determine the feasibility of each value to help the user in selecting feasible combinations of values. If the feasible domain of an at-

tribute is reduced to a single value, this value is automatically filled in to make sure all configured containers can be sold and produced.

3.3.5.3 Solving product configuration

After modeling the product configuration as a CSP with its variables, domains and constraints, a solving paradigm has to be chosen to solve the different product configurations for the end users. After processing the input provided by the end use, the solver needs to share which product configurations match all specifications of the product model and its constraints.

The MCI product configurator integrates a constraint programming solver. The illustrated table constraint allows propagating the information on values on variables to reduce the domain of other variables. Propagation here achieves domain consistency. This is illustrated in Figure 3.26, where on the left, the three variables are symbolically related by the table constraint. On the right, we assume that the value MV is chosen for the variable AC and the propagation of the table constraint reduces the domains of the two other variables. Since there is only one value left in the domain of the variables, a solution is found $\{AC = MN, FAM = MV, SE = No\}$.

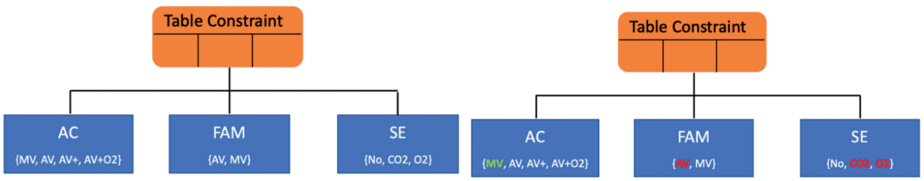


Figure 3.26: Example of propagation using the table constraint in the MCI product configurator.

3.4 SAT solvers

3.4.1 What category of problems does SAT solvers solve?

The Boolean satisfiability problem (SAT) is the problem of deciding whether a logical formula of Boolean variables is satisfiable, that is, whether a specific assignment of true and false values to those variables leads to that formula evaluating to true. It relies on propositional logic.

SAT solvers are typically helpful in problems where there are many options and dependencies that can be encoded as Boolean formulas: correctness of hardware design, automatically test, and detect connection defects, which may cause circuit failure in integrated circuits, software verification, planning and resource allocation problems

in product configurations problem to ensure that product instances and configuration options satisfy the component dependencies and customer's restrictions, etc.

When a constraint satisfaction problem is defined over Boolean variables and the constraints are expressed as propositional formulas, the CSP can then be solved using a SAT solver, that is, a solver aiming at finding a Boolean assignment of the variables, satisfying the constraints.

Graph coloring

The graph coloring problem has been described as a CSP. The objective is to color the vertices of the graph so that two adjacent vertices are colored with different colors. This problem is NP-complete. This problem can also be modeled as a Boolean CSP.

Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, and a set of colors C, the Boolean variables are the array of color$[v, c]$, with $v \in V$ and $c \in C$. An element color$[v, c]$ is true if the color of vertex v is c; it is false otherwise. For each edge $(v, w) \in E$, we have the constraint $\neg(\text{color}[v, c] \wedge \text{color}[w, c])$ preventing adjacent vertices to have the same color. For each $v \in V$, the disjunction of color$[v, c]$ over all the colors $c \in C$ ensures to have at least one color for each vertex. For each $v \in V$, the conjunction of $\neg(\text{color}[v, \text{c1}] \wedge \text{color}[v, \text{c2}])$ for each different pair of colors (c1,c2) ensures that each vertex has at most one color.

Finding a Boolean assignment to the Boolean variables that satisfies these formulas yields a solution of the problem. Such an assignment can be found by using a SAT solver.

In this section, we thus solve any problem that can be modeled as finding a Boolean assignment to variables satisfying some propositional formula.

3.4.2 Solving problems with SAT solvers

3.4.2.1 Propositional logic

Propositional logic is a very simple logic centered around propositions, that is statements that can be either true or false, such as "*Task T1 must be performed before task T2*." Logic is described by two components: the syntax defines what are the well-formed sentences or formulas in the logic, and the semantics determines the meaning of sentences.

The syntax of propositional logic

A propositional formula is formed by using the following symbols: the logical constants True and False, propositional variables (represented by string of letters), logical connectives ($\neg$ for negation, $\wedge$ for conjunction, $\vee$ for disjunction, $\Rightarrow$ for implication and $\Leftrightarrow$ for equivalence), and parentheses.

A logical constant and a propositional variable are propositional formulas. If p and q are propositional formulas, then $(\neg p)$, $(p \wedge q)$, $(p \vee q)$, $(p \Rightarrow q)$ and $(p \Leftrightarrow q)$ are propositional formulas.

The semantics of propositional logic

The semantics define the meaning of a propositional formula. More precisely, the semantics of a formula assigns a truth value to each formula in the context of some world where some facts are *true* and others are *false*. A possible world, called *interpretation*, for a given formula is an assignment that assigns a truth value (*true* or *false*) to each of the propositional variables of the formula. For a given interpretation I, it is then possible to decide the truth value of a formula following the following rules:

- The truth value of the logical constants True and False are *true* and *false*, respectively.
- The truth value of a propositional variable A is the truth value of A in the interpretation.
- The truth value of a formula of the form $(\neg p)$, $(p \wedge q)$, $(p \vee q)$, $(p \Rightarrow q)$, or $(p \Leftrightarrow q)$ depends on the truth value of the formulas p and q, following the classical table.

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
true	*true*	*false*	*true*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*	*false*
false	*true*	*true*	*false*	*true*	*true*	*false*
false	*false*	*true*	*false*	*false*	*true*	*true*

Example. Given an interpretation with the assignment ($A = true$, $B = false$, $C = false$), the truth value of the formula $(A \vee B) \wedge (B \Rightarrow \neg C)$ in this interpretation is *true* as $(true \vee false) \wedge (false \Rightarrow (\neg false))$ reduces to $(true \wedge (false \Rightarrow true))$ and to $(true \wedge true)$, which yields *true*.

Given a formula, there are interpretations where the formula is true and interpretations where the formula is false. An interpretation where the formula is true is called a *model* of the formula. The meaning of the formula, that is, its semantics is then defined by the models of the formula. A propositional formula is *satisfiable* if it has at least a model; otherwise, it is *unsatisfiable* (no model). A formula is *valid* (also called a tautology) when it is true in all interpretations. We obviously have that a formula p is valid if and only if $\neg p$ is unsatisfiable. Two formulas with the same propositional variables are equivalent if they have the same models. A (sub)formula can always be substituted by an equivalent (sub)formula without changing its semantics. Classical examples of equivalent formulas are presented in Figure 3.27.

$\neg(\neg p)$	is equivalent to	p	(double negation)
$p \wedge q$	is equivalent to	$q \wedge p$	(commutativity of $\wedge$)
$p \vee q$	is equivalent to	$q \vee p$	(commutativity of $\vee$)
$p \Leftrightarrow q$	is equivalent to	$(p \Rightarrow q) \wedge (p \Rightarrow q)$	($\Leftrightarrow$ elimination)
$p \Rightarrow q$	is equivalent to	$\neg p \vee q$	($\Rightarrow$ elimination)
$p \Rightarrow q$	is equivalent to	$\neg q \Rightarrow \neg p$	(contraposition)
$\neg(p \vee q)$	is equivalent to	$\neg p \wedge \neg q$	(de Morgan)
$\neg(p \wedge q)$	is equivalent to	$\neg p \vee \neg q$	(de Morgan)
$p \vee (q \wedge r)$	is equivalent to	$(p \vee q) \wedge (p \vee r)$	(distributivity)
$p \wedge (q \vee r)$	is equivalent to	$(p \wedge q) \vee (p \wedge r)$	(distributivity)

Figure 3.27: Examples of equivalent propositional formulas.

The satisfaction problem

The satisfaction problem aims at deciding whether a given formula is satisfiable. The set of satisfiable propositional formulas has been formalized as SAT. Deciding whether a formula belongs or not to SAT is an NP-complete problem, hence requiring an exponential complexity (assuming P $\neq$ NP). If the formula has n propositional variables, the number of interpretations is 2^n. Deciding if a formula is satisfiable can be done by enumerating all the possible interpretations of the formula and evaluating the truth value of the formula in this interpretation. The formula is then satisfied as soon as the truth value of the formula is *true* in one of the interpretations. This naïve approach is only practical when the number of variables is small.

Example. The formula $(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee B) \wedge (\neg A \vee \neg B \vee \neg C) \wedge (\neg A \vee C)$ is unsatisfiable. For each of its 8 possible interpretations, this formula evaluates to false.

3.4.2.2 SAT solvers

A SAT solver takes a propositional formula as input and either returns a model of the formula (i. e., a assignment of its variables), or output that the formula is unsatisfiable. Most SAT solvers restrict their input to specific propositional formulas called conjunctive normal forms (CNF).

Conjunctive normal form (CNF)

A *literal* is either a propositional variable (A) or its negation ($\neg A$). A *clause* is a disjunction of literals. A formula is in CNF when it is a conjunction of clauses. All propositional formulas can be transformed into an equivalent that is CNF.

Example. The formula $(A \vee B) \Leftrightarrow (\neg C \wedge D)$ can be rewritten as the following CNF formula $(\neg A \vee \neg C) \wedge (\neg A \vee D) \wedge (\neg B \vee \neg C) \wedge (\neg B \vee D) \wedge (A \vee B \vee C \vee \neg D)$. The size of the resulting CNF is, however, exponentially large. For instance, a disjunction of n subformulas, each being a conjunction of m variables leads to a CNF formulas of size $O(n.m^n)$. Tseytin transformation avoids this exponential explosion by introducing new variables

for subformulas. For instance, the formula $(A \wedge B) \vee (C \wedge D)$ is first transformed as the conjunction of the following three formulas X1 $\Leftrightarrow (A \wedge B)$, X2 $\Leftrightarrow (C \wedge D)$, and X1 $\vee$ X2. These formulas are then transformed into CNF.

Basic backtracking algorithm

The basic algorithm for a SAT solver explores a search tree where each node contains a partial assignment of the variables, and the leaves are all the possible interpretations. The traversal is performed by a backtracking algorithm. It chooses a variable, assigns it a truth value and recursively checks if the resulting formula is satisfiable. If it is unsatisfiable, the other truth value is chosen. If one of the truth values leads to satisfiability, then the algorithm returns the model, otherwise it returns unsatisfiability. The basic case of the recursive algorithm is when one of the clauses evaluated to *false* under the current assignment, or when all the clauses evaluated to *true* under the current assignment. Specific data structures are used to simplify these tests. This approach is complete as it explores the whole search space. It is also constructive; it starts from an empty partial assignment and extends it until an assignment satisfying all the clauses is found.

DPLL: adding unit propagation and pure literal elimination

When a clause contains a single unassigned variable, the value of this variable can be fixed to satisfy this clause. The other truth value should not be explored. This is called *unit propagation*. If the occurrences of a variable in the different clauses are always positive (A), one could assign the value *true* to this variable; the value *false* should not be considered in the search. A similar reasoning can be made when the occurrences are always negative ($\neg A$). This is called *pure literal elimination*.

Both unit propagation and pure literal elimination can be applied iteratively at each step of the recursive algorithm, reducing the exponential growth of the search tree. The integration of unit propagation and pure literal elimination into a backtracking algorithm leads to the DPLL algorithm (Davis–Putman–Logemann–Loveland), which is the basis of many efficient SAT solvers.

CDCL: introducing clause learning

The traversal of the search tree with DPLL is depth first. Backtracking is chronological as the search goes back to the parent node in the search tree. However, when a partial assignment is found to lead to unsatisfiability, no information is deduced from the origin of this unsatisfiability. Conflict-driven clause learning (CDCL) improves DPLL in a significant way by introducing a nonchronological backtracking when the search reaches a dead end. A precise description is out of the scope of this chapter. The effect of CDCL is to build a new clause that is implied by the original clauses and blocks the current branch together with as many branches that share the choices leading to unsatisfiability. This is achieved by building an implication graph on the partial variable assignment and introducing a new clause with the negation of the assignments that lead to the conflict. Backtracking can then be made directly to the variable involved in the new clause that was first assigned in the search tree.

SAT solving through local search

Both DPLL and CDCL are complete and constructive approaches to the satisfiability problem. Although these methods are quite efficient, SAT is NP-complete and there will

always be a complexity barrier for large instances. There are methods of SAT solving based on local search, that is, using an incomplete and perturbative approach. Starting from a (random) initial assignment to the propositional variables, at each step the value of one variable is flipped. The cost function is the number of unsatisfied clauses, and a solution is found when the cost reaches zero. This optimization problem is called max-SAT as it aims at optimizing the number of satisfied clauses, what is also interesting for unsatisfiable formulas. There exist many algorithms that solve MaxSAT through a local search. For instance, GSAT and WalkSAT methods are based on a MinConflict CSP local search, that is, choosing a variable, which flip minimizes the cost function. WalkSAT also introduces some perturbation to enhance diversification and avoid local minima.

3.4.2.3 Modeling problems

Problems that can be modeled and described as a SAT problem using propositional formulas. They can be solved with SAT solvers. The resulting solution, a specific assignment of Boolean values to the variables, can then be expressed as a solution to the initial problem. SAT solvers can therefore be helpful in a variety of problems where options and dependencies can be encoded as propositional formulas, as with the following example illustrating a planning challenge aiming at having a drink. A state of the world is here modeled by five Boolean status, represented by five propositional variables: "Is there a clean glass" (*CG*), "Is there a bottle of beer" (*BB*), "Is there a beer in a glass" (*BG*), "Am I thirsty" (*T*), and "Am I happy" (*H*). Initially, there is a clean glass (*CG*), a bottle of beer (*BB*), and the person is thirsty (*T*). There is no beer in a glass ($\neg BG$) and the person is not happy ($\neg H$). There are two actions. The action MakeDrink() requires *CG* and *BB* as precondition, and *BG*, $\neg CG$, $\neg BB$ as the effect. The action Drink() requires *T* and *BG* as a precondition, and *H*, $\neg T$, $\neg BG$ as the effect. The objective is to reach a state where the person is happy. What is a suitable combination of actions to reach the goal state?

This planning problem is now modeled as a SAT problem. To simplify the presentation, it is assumed that the goal can be reached by a succession of two actions. There will then be three considered states: the initial state, the state after the first action, and the state after the second action, which will be the final state. Each propositional variable describing a state will be subscripted by 0, 1, or 2 to model these three states. For each level of action, the two propositional variables MakeDrink_i and Drink_i state depending on whether the corresponding action is performed at level *i*.

The formula *PLAN* for this planning problem has then the following form:

$$\mathit{InitialState} \wedge \mathit{DescriptionAction}_1 \wedge \mathit{DescriptionAction}_2 \wedge \mathit{Goal}$$

where *InitialState* is the formula $CG_0 \wedge BB_0 \wedge T_0 \wedge \neg BG_0 \wedge \neg H_0$, *Goal* is the formula H_2, and $\mathit{DescriptionAction}_i$ are formulas modeling which action is performed as the first and second actions in the plan.

The $\mathit{DescriptionAction}_i$ should model the following elements:

– The precondition of the action must be satisfied by the preceding state:

$$(\text{MakeDrink}_i \Rightarrow CG_i \wedge BB_i) \wedge (\text{Drink}_i \Rightarrow T_i \wedge BG_i) \quad (i = 1, 2)$$

– The effect of the action must be satisfied by the next state:

$$(\text{MakeDrink}_i \Rightarrow BG_{i+1} \wedge \neg CG_{i+1} \wedge \neg BB_{i+1}) \wedge (\text{Drink}_i \Rightarrow H_{i+1} \wedge \neg T_{i+1} \wedge \neg BG_{i+1}) \quad (i = 1, 2)$$

– After an action, the state propositional variable not involved in the effect of the action are unchanged after this action:

$$(\text{MakeDrink}_i \Rightarrow (H_i \Leftrightarrow H_{i-1}) \wedge (T_i \Leftrightarrow T_{i-1})) \wedge (\text{Drink}_i \Rightarrow (CG_i \Leftrightarrow CG_{i-1}) \wedge (BB_i \Leftrightarrow BB_{i-1})) \quad (i = 1, 2)$$

– At least one action must be performed at each level

$$\text{MakeDrink}_i \vee \text{Drink}_i \quad (i = 1, 2)$$

– At most one action can be performed at each level

$$\neg(\text{MakeDrink}_i \wedge \text{Drink}_i) \quad (i = 1, 2)$$

Finding a model for the formula *PLAN* ensures to have a plan that starts from the initial state and reaches the goal states after two actions. To find a model, the formula *PLAN* can be transformed into a CNF form and a SAT solver algorithm can then be used. The following variable assignment is a model of the formula *PLAN*:

$$CG_0, BB_0, \neg BG_0, T_0, \neg H_0, \quad \text{MakeDrink}_1, \neg\text{Drink}_1, \quad \neg CG_1, \neg BB_1, BG_1, T_1, \neg H_1,$$
$$\neg\text{MakeDrink}_2, \text{Drink}_2, \quad \neg CG_2, \neg BB_2, \neg BG_2, \neg T_2, H_2$$

The conclusion is that the first action is MakeDrink and the second action is Drink.

For a real problem, the propositional formula can be automatically generated from a formal description of the planning problem, such as in planning domain definition language (PDLL), a family of languages that allow to define a planning problem using logic. The propositional formula can also be enriched by additional clauses expressing possible mutual exclusions between actions and/or literals that can be algorithmically derived.

3.4.3 What are the limitations of SAT solvers?

SAT solvers are used to solve Boolean CSPs or any problem that can be modeled as a satisfaction problem in propositional logic. One of the advantages of propositional logic is its simplicity. It is also compositional, meaning that different formulas can be composed without affecting the meaning of each of them. Although SAT is NP-complete, modern SAT solvers are quite efficient, even for large problems. The main limitation of propositional logic is its inability to represent objects and relations between objects. It also lacks quantifiers allowing stating conjunctions or disjunctions over different objects. In the planning problem, it is impossible to have a formula about all actions in a given level. As a consequence, a propositional model of a real problem may involve tens of thousands of variables and hundreds of thousands of clauses. Despite these limitations, SAT solvers are competitive for solving complex problems.

SAT solvers have been extended to satisfiability modulo theories (SMT) solvers, such as z3. Such solvers do not only solve SAT, but also formulas over richer logics, such as arithmetic or uninterpreted functions. SAT solvers have also been combined with constraint programming solvers, leading to CP-SAT that can be used in operational research

applications. The CDCL technique is a basis for efficient SMT solvers as well as for CP-SAT.

3.4.4 Industry example: containers configuration using SAT, at Maersk Container Industry A/S

Yves Deville, Karen Veldeman, Nikolaj Bjorner

3.4.4.1 Modeling product configuration as a SAT problem

The industrial example on product configurators in Section 3.3.5 is further developed. The product configuration model has been described as a CSP using variables with a finite domain, and a set of constraints defining the possible configurations of containers. The MCI product configurator also integrates a SAT solver. The SAT solver relies on encoding all constraints using Boolean variables only in the form of clauses. On the surface, this may seem very low level and potentially losing structure, but this representation allows for efficient propagation using clauses in an efficient SAT solver.

An encoding of the sensor module attribute (SM) and its domain into SAT can be achieved using three Booleans SM.No, SM.CO2, and SM.O2, where the variable SM.x is true when the value of SM is set to x. To ensure that in an assignment of the Boolean variables, the SM attribute has a value, and the following clause is added to the CSP: SM.No $\vee$ SM.CO2 $\vee$ SM.O2. Other formulas ensure that this attribute is not assigned more than one value: $\neg$(SM.No $\wedge$ SM.CO2) $\wedge$ $\neg$(SM.No $\wedge$ SM.O2) $\wedge\neg$(SM.CO2 $\wedge$ SM.O2). The resulting clauses are then also added: $\neg$SM.No $\vee$ $\neg$SM.CO2, $\neg$SM.No $\vee$ $\neg$SM.O, $\neg$SM.CO2 $\vee$ $\neg$SM.O2. Similar clauses should also be added for the other attributes.

The table constraint illustrated in Table 3.4 expresses that some combinations of fresh air exchange modules and sensor modules cannot be combined into one single container. This table constraint should be transformed into clauses. A first idea would be to encode each line of the table as a conjunction (line 1 as AC.MV $\wedge$ FAM.MV $\wedge$ SM.No), and relate these formulas with disjunction. The size of the resulting formula when transformed in CNF is, however, exponentially large. This exponential explosion can be avoided by using Tseytin transformation that introduces a new Boolean variable to each row of the table (T1, T2, T3, and T4). A set of formulas relates these new variables with the attribute variables in the table constraint. For the first row, the formula T1 $\Rightarrow$ (AC.MV $\wedge$ FAM.MV $\wedge$ SM.No) relates T1 and the values of the attributes in this line. The resulting clauses are then AC.MV $\vee$ $\neg$T1, FAM.MV $\vee$ $\neg$T1, SM.No $\vee$ $\neg$T1. Similar clauses are added for the other rows. Tseytin transformation traditionally should also add formulas such as T1 $\Leftarrow$ (AC.MV $\wedge$ FAM.MV $\wedge$ SM.No) for the first row. It is, however, better to reexpress these formulas by other equivalent formulas. For each value of each attribute in the table, a formula relates the value to the corresponding row in the table. For the value

number of the attribute SM, the formula is SM.No $\Rightarrow$ (T1 $\vee$ T2), leading to the clause $\neg$SM.No $\vee$ T1 $\vee$ T2.

The proposed encoding is very efficient. When applying DPLL on the above SAT encoding of the problem, thanks to unit propagation, it actually achieves a pruning equivalent to domain consistency. For instance, as soon as the variable AC.MV is set to true, the variable T1 has to be set to true by the clause $\neg$AC.MV $\vee$ T1. Then the variables FAM.MV and SM.No have to be set to true by the clauses FAM.MV $\vee$ $\neg$T1 and SM.No $\vee$ $\neg$T1.