An Insertion-Based Linked List Variable and Regular Constraint for Classical Planning with Constraint Programming

Damien Van Meerbeeck

Linköping University, Sweden
damien.van.meerbeeck@liu.se

Introduction

Constraint Programming (CP) is a powerful paradigm for modeling and solving combinatorial problems. Thanks to its expressiveness and flexibility, we are able to model a wide range of problems with various types of constraints and data structures. The choices of variables, constraints, and search heuristics are crucial for the performance of CP solvers, and can be tailored to the specific problem at hand.

Classical Planning is the problem of finding a sequence of actions that transforms an initial state into a goal state. Modeling planning tasks in CP often represents the plan as a sequence of actions, where each variable represents the action executed at a given step. The dynamics of the task are then modeled on top of this representation by different means. For example, a compact representation can be modeled using REGULAR constraints (Pesant 2004) over a factored representation of the task, to form a compact model of the planning task (Babaki, Pesant, and Quimper 2020).

While this representation is simple and effective, it has some limitations when it comes to designing branching heuristics. An advantage of CP-based planners compared to state-of-the-art heuristic search planners is the ability to construct the plan in a non-sequential way. However, this advantage is diminished by the list representation of the plan, which requires choosing the position of an action when adding it to the plan. A single misplaced action can therefore render the resulting subproblem unsolvable, even if the action was required to reach the goal.

Branching heuristics based on element precedence rather than their position would allow for more flexible and accurate search strategies, as it would allow to insert actions in a partially constructed plan without having to decide on their exact position. Such heuristics have been successfully applied to problems such as scheduling and routing, but have yet to be explored for classical planning. These models rely on insertion-based variables, such as the insertion sequence variable (Thomas, Kameugne, and Schaus 2020; Delecluse, Schaus, and Van Hentenryck 2025) which allows for branching heuristics inserting new elements in a partially constructed sequence of ordered elements. However, to our knowledge, no existing insertion-based variable is suited for classical planning, as they do not allow for multiple occurrences of the same element to occur, and in the case of the sequence variable, represents each element individually which can lead to inefficient memory usage as each predecessor-successor relationship needs to be represented.

In this article, we propose the linked list variable, a novel insertion-based variable allowing for multiple occurrences of the same element and a compact representation of the predecessor-successor relationships. We also propose a new REGULAR constraint for this variable and show that the propagation of the variable can be used to collect information to guide the search.

Background

Automaton An *automaton* $\mathcal{A}$ is a tuple $\langle \Sigma, \mathcal{Q}, \delta, q_0, F \rangle$, where Σ is a finite *alphabet*, $\mathcal{Q}$ is a finite set of *states*, δ is the *transition function*, q_0 is the *initial state*, and F is the set of *final states*. An automaton $\mathcal{A}$ can be transformed into an *normalized* automaton $\mathcal{N}^{\mathcal{A}}_{\alpha,\beta}$ with an additional initial state with a transition α to the original initial state, and an additional unique final state with transitions β from the original final states. An example of the transformation is illustrated in Figure 2. If a word w over Σ is in the language of an automaton $\mathcal{A}$, we say that w is *accepted* by $\mathcal{A}$.

Constraint Programming A problem in CP is modelled with sets of variables $\mathcal{X}$, domains $\mathcal{D}$, and constraints $\mathcal{C}$ over the variables. Each variable $x \in \mathcal{X}$ is associated the domain $\mathcal{D}(x) \subseteq \mathcal{D}$.

Linked List Variable

The linked list variable $\mathcal{L}$ is a variable whose domain $\mathcal{D}(\mathcal{L})$ is a set of ordered sequences of elements from a given set Σ, with repetitions and a maximum length $\ell_{max} \geq 2$. The variable is represented as an ordered sequence of nodes $\overrightarrow{\mathcal{L}}$, where each node $N \in \overrightarrow{\mathcal{L}}$ is used to maintain both the structure of the list and the information for the possible values of the variable. A node N is represented as a tuple $\langle \textit{id}, \textit{prev}, \textit{next}, v, B, v^{pred}, v^{succ}, \psi \rangle$, where *id* is a unique identifier for the node, *prev* and *next* are the structural identifiers of the previous and next nodes, $v \in \Sigma$ is the value of the node, $B \subseteq \Sigma$ is a set containing the possible elements that can be inserted between the node and the next, v^{pred} and v^{succ} are variables whose domains are the possible values of the direct predecessor and successor, and ψ is the variable for the position of the node, with domain $\mathcal{D}(\psi) = \{0, \ldots, \ell_{max} - 1\}$.

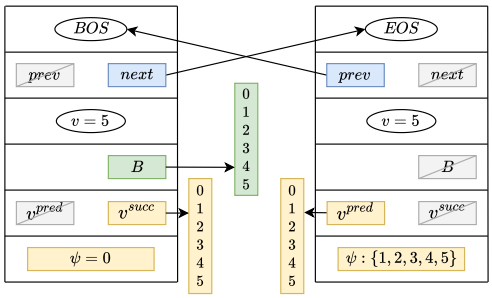

Figure 1: Linked list variable after initialization for elements $\{0, \ldots, 5\}$, a maximum length of 6, and value 5 for both *BOS* and *EOS* nodes.

The information contained is chosen to be fully reversible, meaning that a CP solver can easily backtrack to a previous state of the search tree by restoring the previous state of each node of the variable. Only the *id* and the v do not need to be reversible, since they are fixed when the node is created and do not change. For the sake of simplicity, only the possible values for insertion between a node and the next node is represented since the possible insertion before the node can be obtained from the previous node.

Nodes Relations For two consecutive nodes $N_i, N_j \in \overrightarrow{\mathcal{L}}$ such that $next_i = id_j$ and $prev_j = id_i$, the relation is noted as $N_i \rightarrow N_j$. If $N_i \rightarrow N_j$ and no elements can be inserted between the two nodes, the node N_i is the *predecessor* of N_j and we write $N_i \mapsto N_j$. On the other hand, if $N_i \rightarrow N_j$, and if at least one element must be placed between them, then N_i is *not the predecessor* of N_j and we write $N_i \rightsquigarrow N_j$.

Variable Consistency To ensure the consistency of the variable during search, we need to check that the structure of the list is maintained and that the information contained in each node is valid. Therefore, at all times, we need to ensure that for all nodes $N_i, N_j \in \overrightarrow{\mathcal{L}}$:

- If $N_i \rightarrow N_j$ then $\psi_i < \psi_j$.
- If $N_i \rightarrow N_j$ and $\psi_j - \psi_i = 1$, then $N_i \mapsto N_j$.
- $N_i \mapsto N_j$ if and only if $B_i = \emptyset$.
- If $N_i \mapsto N_j$, then the value of N_i is the only value in v^{pred} of N_j, the value of N_j is the only value in v^{succ} of N_i and the difference of the position is 1.
- If $N_i \rightarrow N_j$, and if $v_j \notin v_i^{succ}$ or $v_i \notin v_j^{pred}$, then $N_i \rightsquigarrow N_j$,
- $N_i \rightsquigarrow N_j$ if and only if $\psi_j - \psi_i > 1$.

Initialization To hold the structure of the list, the variable is initialized with two special nodes representing the beginning and end of the list, respectively *BOS* and *EOS* $\in \overrightarrow{\mathcal{L}}$, for which the value is chosen at initialization. An example of the initialization of the variable is illustrated in Figure 1. Since the node *BOS* is the start of the list, we fix its position to 0, and *prev* and v^{pred} are not defined. Similarly, since *EOS* is the end of the list, *prev*, v^{succ} and B are not defined.

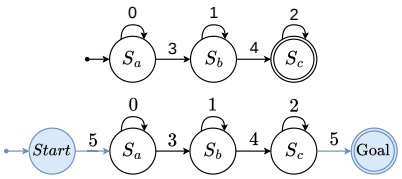

Figure 2: Normalization of an automaton. The original automaton $\mathcal{A}$ is depicted on the top, and the normalized automaton $\mathcal{N}^{\mathcal{A}}_{5,5}$ is depicted on the bottom.

Insertion To insert a new element between two consecutive nodes N_i and N_j, we first ensure that the value to be inserted is in B_i. We then instantiate a new node N_m with the inserted element as its value and update the predecessor and successor pointers to reflect the relation $N_i \rightarrow N_m \rightarrow N_j$. Then, B_m is initialized with the elements of B_i. The same is done for the sets v_m^{pred} and v_m^{succ}, which are initialized with the values of v_i^{pred} and v_j^{succ} respectively, and to which we add v_i and v_j.

Regular Constraint for the Linked List Variable

The REGULAR constraint for the linked list variable is a global constraint that ensures that the sequence of elements represented by the linked list variable is accepted by a given automaton. Contrary to the propagation of the REGULAR constraint for a sequence of integer variables (Pesant 2004), for which any automaton $\mathcal{A}$ can be used, the linked list variable requires the automaton to be normalized as $\mathcal{N}^{\mathcal{A}}_{\alpha,\beta}$, where α and β are the values of the *BOS* and *EOS*, respectively. We therefore define the REGULAR constraint for a linked list variable $\mathcal{L}$ as REGULAR$(\mathcal{L}, \mathcal{N}^{\mathcal{A}}_{\alpha,\beta})$ and this constraint holds if the sequence of elements represented by $\mathcal{L}$ is accepted by the automaton $\mathcal{N}^{\mathcal{A}}_{\alpha,\beta}$.

Propagation

Since the linked list variable is used to represent a partially ordered sequence of elements, to accurately filter the domains of the variable, the propagation algorithm needs to take into account all possible subsequences of elements that can be placed between two consecutive nodes. To do so efficiently, during its execution, the algorithm keeps track of and combines a number of properties for all sequences of the same length reaching the same state of the automaton. These properties are the *possible* and *required* elements to be placed between two consecutive nodes, the *starting* elements that can start the subsequences, the *ending* elements that can end the subsequences, and a possible *path* to reach the next node. Additionally, with the path property, we can also keep track of the *distance* to the next node.

The propagation is performed in three main steps. First, starting from the beginning of the list, a forward phase computes for each consecutive pair of nodes, the properties of every possible subsequences of elements that can be placed between the two nodes grouped by distance. Then, we perform a backward phase, starting from the end of the list,

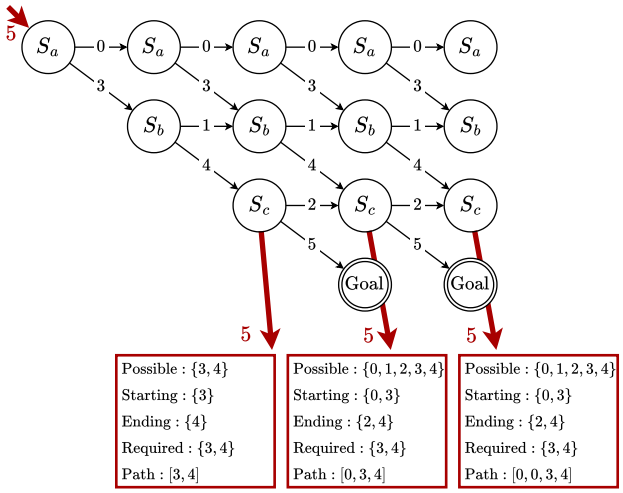

Figure 3: An iteration of the forward phase of the linked list variable of Figure 1 for the normalized automaton of Figure 2. The unfolding of the automaton starts from the leading state S_a of the node *BOS* and ends when the last position of the next node *EOS* can be applied. The properties represented at the bottom of the figure are the combined properties of all subsequences of elements reaching the state where the value of the next node can be applied.

to remove the subsequences that are no longer consistent if their extension does not lead to be accepted by the automaton. Finally, we perform a domain pruning step to remove elements from the domains of the nodes in the list that are not consistent with any accepting sequence of elements in the automaton.

Forward Phase To perform the forward phase, we start from the beginning of the linked list and for each pair of consecutive nodes, we compute the properties of the sequences of elements that can be placed between the two nodes. To do so, we execute a time-wise unfolding of the automaton from each leading state of the first node, and for each applicable transition, we update the properties of the subsequences of the resulting state. We execute this process from the position of the first node, until the next node can be reached to ensure that each possible sequence of elements between the two nodes is taken into account. While doing so, each time the element of the next node can be applied, we keep track of the properties of the leading state. The result of this process is a set of properties for each state of the automaton where the next node can be executed. Once the analysis is complete for a pair of consecutive nodes, the reached states will be the starting states for the next pair of consecutive nodes.

An example of the process of the forward phase for a pair of consecutive nodes is illustrated in Figure 3.

Backward Phase Once the forward phase is finished, we have for each pair of consecutive nodes, for each state of the automaton where the first node leads and for each state reachable by the next node, the properties of the subsequences of values that can be placed between the two nodes.

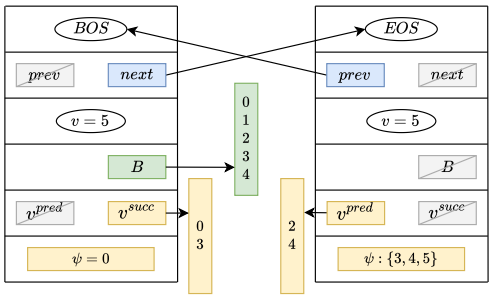

Figure 4: Linked list of Figure 1 after the propagation of the regular constraint for the normalized automaton of Figure 2.

However, some of these sequences may be inconsistent because they may ultimately not lead to an accepting state of the automaton discovered during the forward phase of a following pair of consecutive nodes. To remove these inconsistent properties, we perform a backward phase starting from the end of the linked list and for each pair of consecutive nodes, we check for each property if the resulting state was reachable from a previous iteration of the backward phase. If not, we remove the properties, else we keep it and add the state to the set of reached states for the next iteration of the backward phase.

Domain Pruning The last step of the propagation is to prune the domains of variables by removing from the nodes of the linked list, the values that are not consistent with the combined resulting properties. As such, for consecutive nodes $N_i \rightarrow N_j$, we prune B_i with the *possible* property, v_i^{succ} with the *starting* property, v_j^{pred} with the *ending* property and the position of the nodes with the *distance* of the paths. The *required* property can also be used to prune the domains of linked list variables by inserting the required values between the nodes. An example of the result of the propagation is illustrated in Figure 4.

Branching Information During the propagation, we kept track of the path property for each pair of consecutive nodes. This information can be directly used by a branching heuristic to potentially guide the search towards a solution by following the path of values that can lead to an accepting sequence in the automaton. However, the path represents a single sequence of values, while there may be multiple sequences of values that can lead to an accepting sequence in the automaton. Therefore, it is desirable to have a tie-breaking strategy to select the path to follow that will best guide the search towards a solution.

Implementation

We implemented the linked list variable and the associated REGULAR constraint in the MaxiCP solver (Schaus et al. 2026). For the search, we also implemented a custom branching heuristic that uses the *path* property from the propagation of the REGULAR constraint. During the propagation, we use a tie-breaking strategy preferring paths with the largest number of self-looping actions (i.e. the path with the least number of state-changing actions) to guide the

	solved	optimal
MiniCPBP	60	33
CP-SAT	141	69
Linked List	150	63

Table 1: Comparison of the performance of the model using the linked list variable to the two models using a sequence of integer variables for the two different solvers, MiniCPBP and CP-SAT. Over the 150 tasks of the Miconic domain, the first column reports the number of tasks solved, while the second column reports the number of tasks for which the optimal solution was found and proved optimal.

search towards a solution by inserting the actions that are more likely to be required in the plan. To determine which insertion to perform, we iterate over each constraint and if a state-changing action can be inserted immediately after or immediately before an existing action, we insert it. Otherwise, we insert a self-looping action. If all automata accept the current word represented by the linked list variable, we have reached a solution.

Experimental Comparison

To validate and evaluate our model with the linked list variable and the REGULAR constraint to solve planning tasks, we compared it to a model using an array of integer variables to represent the plan with two different solvers: MiniCPBP (Pesant 2019) with a marginal-augmented branching heuristic, and a state-of-the-art solver CP-SAT (Perron and Didier 2025) in default mode.

The evaluation is performed on the Miconic domain of the International Planning Competition (IPC), for which the goal is to find a sequence of actions that transports passengers from their initial floor to their destination floor using an elevator. This domain is particularly interesting for the evaluation of our model since it exhibits a high degree of action precedence and interactions between passengers, which makes it a good candidate to demonstrate the strength of the linked list variable in representing partially ordered sequences of actions. The automata for the tasks are constructed from the FDR representation obtained with Scorpion (Seipp, Keller, and Helmert 2020) on which the actions moving the elevator up and down are reduced to the same label if they lead to the same floor. The automata for the array model is further modified to allow trailing NOOP actions, and for the linked list model we normalize all automata. Both models are defined similarly, with a variable for the plan and a REGULAR constraint for each automaton.

All experiments are run on Intel Xeon Gold 6130 processors with a time limit of 30 minutes and a memory limit of 8 GiB per run. For each task, the search explores plan lengths ranging from the known optimal length to 1.5 times that value, this allows for an evaluation of the performance of the models in finding solutions, rather than solely focusing on optimality. The search seeks to minimize the plan length found and is performed using Limited Discrepancy Search (LDS) (Harvey and Ginsberg 1995), except for the model using CP-SAT for which we use the default search strategy of the solver.

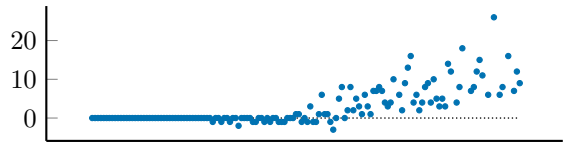

Figure 5: Difference of the best plan length found by the linked list model and the sequence model with CP-SAT for the tasks of the Miconic domain ordered by increasing difficulty.

The results on Table 1 show that compared to the model using an array representation, the model using the linked list variable is able to find a solution for all tasks of the domain. While the model using CP-SAT is able to find the most optimal solutions, the linked list model comes close and still outperforms the model using MiniCPBP. Figure 5 illustrates another strength of the linked list model, which is its ability to find good quality solutions for complex tasks, even if it may not always be able to prove their optimality within the given time limit. This suggests that the strength of the linked list model and the chosen branching strategy lies in its ability to synchronize the different precedence constraints of the actions imposed by the tasks and the interactions between passengers.

Future Works

We plan to optimize the REGULAR constraint propagation through position-wise bound consistency, expanding states only when they propagate new information in the forward phase. We will also explore more effective branching heuristics and search strategies for planning tasks. Another potential direction to consider is the use of Large Neighborhood Search (LNS) which was proven effective for insertion-based sequence variables in routing.

Conclusion

We introduced the linked list variable, a novel insertion-based variable representing an ordered sequence of elements with repetitions and showed that it can be used in the context of classical planning. We also proposed a REGULAR constraint for this variable and demonstrated that the propagation of the constraint can, in addition to pruning the domains of the variable, provide information to guide the search towards solutions. The early results suggest the potential of the linked list variable and the associated REGULAR constraint, opening promising directions for future work.

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Computations were performed at NSC Tetralith provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) funded by the Swedish Research Council through grant agreement no. 2022-06725.

References

Babaki, B.; Pesant, G.; and Quimper, C. 2020. Solving Classical AI Planning Problems Using Planning-Independent CP Modeling and Search. In Harabor, D.; and Vallati, M., eds., *Proceedings of the 13th Annual Symposium on Combinatorial Search (SoCS 2020)*, 2–10. AAAI Press.

Delecluse, A.; Schaus, P.; and Van Hentenryck, P. 2025. Sequence Variables: A Constraint Programming Computational Domain for Routing and Sequencing. *arXiv preprint arXiv:2510.09373*.

Harvey, W. D.; and Ginsberg, M. L. 1995. Limited Discrepancy Search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, 607–615. Morgan Kaufmann.

Perron, L.; and Didier, F. 2025. CP-SAT. https://developers.google.com/optimization/cp/cp_solver/. Version v9.12, Google.

Pesant, G. 2004. A Regular Language Membership Constraint for Finite Sequences of Variables. In Wallace, M., ed., *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *Lecture Notes in Computer Science*, 482–495. Springer-Verlag.

Pesant, G. 2019. From Support Propagation to Belief Propagation in Constraint Programming. *Journal of Artificial Intelligence Research*, 66: 123–150.

Schaus, P.; Derval, G.; Delecluse, A.; Michel, L.; and Hentenryck, P. V. 2026. MaxiCP: A Not So Mini Constraint Programming Solver. http://www.maxicp.org/.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *Journal of Artificial Intelligence Research*, 67: 129–167.

Thomas, C.; Kameugne, R.; and Schaus, P. 2020. Insertion sequence variables for hybrid routing and scheduling problems. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 457–474. Springer.