juliacon

DisjunctiveProgramming.jl: Generalized Disjunctive Programming Models and Algorithms for JuMP

Hector D. Perez, Shivank Joshi, Ignacio E. Grossmann*

Carnegie Mellon University, Pittsburgh, PA, USA
*grossmann@cmu.edu

ABSTRACT

We present a Julia package, `DisjunctiveProgramming.jl`, that extends the functionality in `JuMP.jl` to allow modeling problems via logical propositions and disjunctive constraints. Such models can then be reformulated into Mixed-Integer Programs (MIPs) that can be solved with the various MIP solvers supported by JuMP. To do so, logical propositions are converted to Conjunctive Normal Form (CNF) and reformulated into equivalent algebraic constraints. Disjunctions are reformulated into mixed-integer constraints via the reformulation technique specified by the user (Big-M or Hull reformulations). The package supports reformulations for disjunctions containing linear, quadratic, and nonlinear constraints.

Keywords

JuMP, Mathematical Optimization, Generalized Disjunctive Programming

1. Introduction

The modeling of systems with discrete and continuous decisions is commonly done in algebraic form with mixed-integer programming (MIP) models. When the problems can be defined by purely linear constraints and a linear objective function, they are referred to as mixed-integer linear programs (MILP). When nonlinearities arrise in either the feasible space or the objective function, they are called mixed-integer nonlinear programs (MINLP).

A more systematic approach to modeling such systems is to use Generalized Disjunctive Programming (GDP) [5, 15], which generalizes the Disjunctive Programming paradigm proposed by Balas [2]. GDP enables the modeling of systems from a logic-based level of abstraction that captures the fundamental rules governing such systems via algebraic constraints and logic. This formulation is useful for expressing problems in an intuitive way that relies on their logical underpinnings without needing to introduce mixed-integer constraints. GDP models are often easier to understand as related constraints are grouped into disjuncts that describe clearly defined subsets of the feasible space. The models obtained via GDP can be reformulated into the pure algebraic form best suited for the application of interest. It is also often possible to exploit the explicit logical structure of a GDP model to provide tighter relaxations than corresponding MIP models, which may improve convergence speed and robustness for solutions via advanced solution algorithms [5].

Within the optimization community, there is a high volume of ongoing research that relies on GDP to formulate models for a variety of applications. Due to the combinatorial nature of system design problems, the GDP paradigm has been applied to the synthesis of complex processes and networks [21, 27], the planning and optimal control of energy systems [8, 19], and the modeling of chemical synthesis under uncertainty [7]. These and numerous other applications of GDP illustrate the benefit of having a robust package for GDP that removes much of the overhead associated with developing and testing GDP models. Although packages with GDP capabilities exist for `Pyomo` [6] and `GAMS` [26], having such a package available in Julia can greatly accelerate research in optimization, where packages like `JuMP.jl` [9] are gaining significant traction.

This paper provides background on the GDP paradigm, and the techniques for reformulating and solving such models. It then presents the package `DisjunctiveProgramming.jl` as an extension to `JuMP.jl` for creating models for optimization that follow the GDP modeling paradigm and can be solved using the vast list of supported solvers [10]. A case study demonstrates the use of the package for chemical process superstructure optimization.

2. Generalized Disjunctive Programming

The GDP form of modeling is an abstraction that uses both algebraic and logical constraints to capture the fundamental rules governing a system. The two main reformulation strategies to transform GDP models into their equivalent MIP models are the Big-M reformulation [23, 24] and the Hull reformulation [20], the latter of which yields tighter models at the expense of larger model sizes [14].

2.1 Model

The general notation for a GDP problem is given in Eq. (1) - (6).

$$\min f(x) \tag{1}$$
$$\text{s.t. } g(x) \leq 0 \tag{2}$$
$$\bigvee_{i \in J_k} \begin{bmatrix} Y_{ik} \\ h_{ik}(x) \leq 0 \end{bmatrix} \quad \forall k \in K \tag{3}$$
$$\Omega(Y) = true \tag{4}$$
$$Y_{ik} \in \{true, false\} \quad \forall i \in J_k, k \in K \tag{5}$$
$$x \in X \subseteq \mathbb{R}^n \tag{6}$$

Here $f(x)$ is the objective function to be minimized over the continuous variable x, $g(x)$ represents the global constraints, $h(x)$ represents the disjunct-specific constraints, and Y is the Boolean variable governing each disjunction. In this notation, there are k disjunctions with i disjuncts in each. Constraints $h_{ik}(x) \leq 0$, are applied only if the Boolean indicator variable for the respective disjunct, Y_{ik}, is denoted as being active (i.e., *true*) [5]. The set of log-

ical constraints, $\Omega(Y)$, describe the logical relationships between the selections of indicator variables. These can take the form of logical propositions or constraint programming expressions.

In the case of a linear objective function and constraint set, the GDP model can be written as Eq. (7) - (11).

$$\min c^T x \tag{7}$$
$$\text{s.t. } Ax \leq b \tag{8}$$
$$\bigvee_{i \in J_k} \begin{bmatrix} Y_{ik} \\ B_{ik}x \leq d_{ik} \end{bmatrix}, \quad \forall k \in K \tag{9}$$
$$\Omega(Y) = true \tag{10}$$
$$Y_{ik} \in \{true, false\} \quad \forall i \in J_k, k \in K \tag{11}$$

2.2 Linear GDP reformulation example

The simplest example of a linear GDP system is given in (12) - (15), where Y_i is a Boolean indicator variable that enforces the constraints in the disjunct ($Ax \leq b$ or $Cx \leq d$) when $true$.

$$\begin{bmatrix} Y_1 \\ Ax \leq b \end{bmatrix} \vee \begin{bmatrix} Y_2 \\ Cx \leq d \end{bmatrix} \tag{12}$$

$$0 \leq x \leq U \tag{13}$$

$$Y_1 \veebar Y_2 \tag{14}$$

$$Y_1, Y_2 \in \{true, false\} \tag{15}$$

For visualization purposes and without loss of generality, the simple linear example is used to illustrate the Big-M and Hull reformulations. Figure 1 illustrates the feasible space of a simple linear GDP with one disjunction and two continuous variables, x_1 and x_2. The rectangle on the left is described by the constraints in the left disjunct, $Ax \leq b$. The rectangle on the right is defined by the constraints in the right disjunct, $Cx \leq d$. The non-overlapping nature of the two regions is supported by the exclusive-OR relationship in Eq. (14).

2.2.1 Big-M Reformulation. The Big-M reformulation for this problem is given by (13), (16) - (19), where M is a sufficiently large scalar that makes the particular constraint redundant when its indicator variable is not selected (i.e., $y_i = 0$). Note that the Boolean variables, Y_i, are replaced by binary variables, y_i. When the integrality constraint in Eq. (19) is relaxed to $0 \leq x_1, x_2 \leq 1$, the resulting feasible region can be visualized by projecting the relaxed model onto the x_1, x_2 plane. This results in the region encapsulated by the dashed line in Figure 2. It should be noted that the relaxed feasible region is not as tight as possible around the original feasible solution space. The choice of the large M value determines the tightness of this relaxation, and the minimal value of M for the optimal relaxation can be found through interval arithmetic when the model is linear. For nonlinear models, the tightest M can be obtained by solving the maximization problem $\{\max h_{ik}(x) : x \in X\}$. An alternate method for tight Big-M relaxations is given in [24].

$$Ax \leq b + M \cdot (1 - y_1) \tag{16}$$

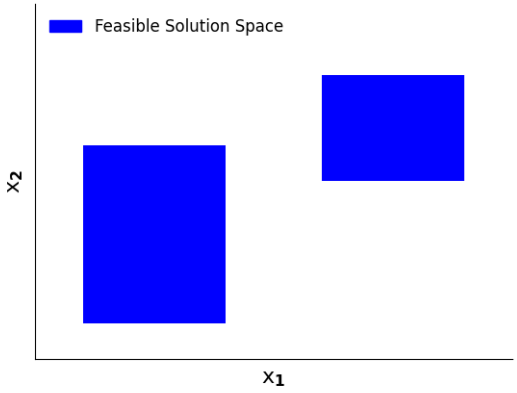

Fig. 1. Feasible solution space for example disjunction

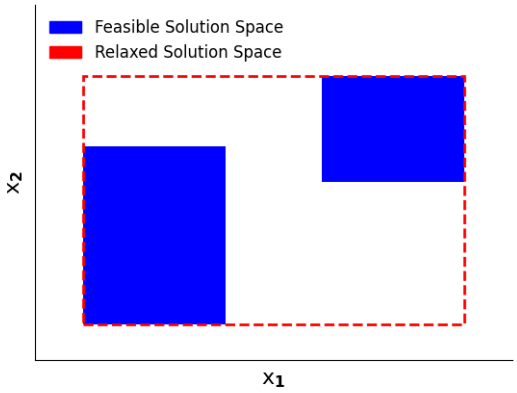

Fig. 2. Relaxed solution space using Big-M Reformulation

$$Cx \leq d + M \cdot (1 - y_2) \tag{17}$$

$$y_1 + y_2 = 1 \tag{18}$$

$$y_1, y_2 \in \{0, 1\} \tag{19}$$

2.2.2 Hull Reformulation. The Hull reformulation is given by (13), (18) - (23), which requires lifting the model to a higher-dimensional space. When projected to the original space, the continuous relaxation of the model is tighter than its Big-M equivalent [15]. The reformulation relaxation can be visualized by the region encapsulated by the dashed line in Figure 3. Note that this reformulation provides a tighter relaxation than the Big-M reformulation in Figure 2. Also note that describing the geometry of this relaxation is more complex than the Big-M relaxation, which is made possible by the increased number of constraints and variables in the model.

$$Ax_1 \leq by_1 \tag{20}$$

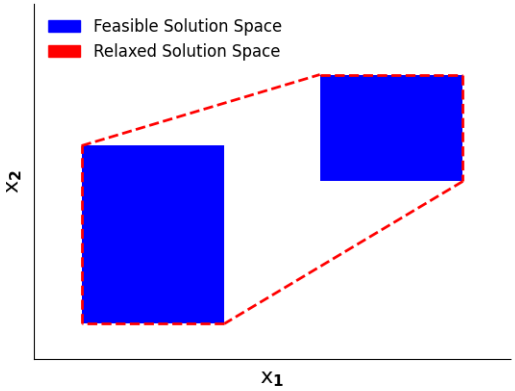

Fig. 3. Relaxed solution space using Hull Reformulation

$$Cx_2 \leq dy_2 \tag{21}$$

$$x = x_1 + x_2 \tag{22}$$

$$0 \leq x_i \leq Uy_i \quad \forall i \in \{1, 2\} \tag{23}$$

2.3 Logic constraint reformulation

2.3.1 Propositional Logic. The logic propositions within the set of decision selection relationships, Ω, must be converted into conjunctive normal form (CNF) to enable reformulating a GDP model as a MIP model. This means that each clause within the set of propositions must be formulated into a conjunction of disjunctions. This process can be accomplished by following the simplifying rules of propositional logic (De Morgan's laws). For boolean variables A, B, and C the following rules are used for converting to CNF (in the order given below),

$$\begin{aligned} A \leftrightarrow B &\text{ is replaced by } (A \rightarrow B) \wedge (B \rightarrow A) \\ A \rightarrow B &\text{ is replaced by } \neg A \vee B \\ \neg(A \vee B) &\text{ is replaced by } \neg A \wedge \neg B \\ \neg(A \wedge B) &\text{ is replaced by } \neg A \vee \neg B \\ (A \wedge B) \vee C &\text{ is replaced by } (A \vee C) \wedge (B \vee C) \end{aligned}$$

Once the logic propositions are converted to CNF, each clause can be converted into an algebraic constraint with the following equivalence,

$$\bigvee_{i \in I} Y_i \text{ becomes } \sum_{i \in I} y_i \geq 1$$

Alternate approaches exist for converting propositional logic statements into CNF, which involve preserving clause satisfiability rather than clause equivalence. These approaches prevent exponential size increase in clauses and yield logically consistent results [18].

2.3.2 Constraint Programming. The constraint-programming selection requirements for constraints within Ω are reformulated as follows,

$$\begin{aligned} \text{exactly}(n, Y) &\text{ becomes } n = \sum_i Y_i \\ \text{atleast}(n, Y) &\text{ becomes } n \leq \sum_i Y_i \\ \text{atmost}(n, Y) &\text{ becomes } n \geq \sum_i Y_i \end{aligned}$$

Exclusive-OR constraints as the one given in Eq. (14) become constraints of the form $exactly(1, Y)$.

2.4 Solution Techniques

2.4.1 Disjunctive branch and bound. The disjunctive branch and bound method closely mirrors the standard branch and bound approach for the solution of mixed-integer programming problems [14]. A search tree is initialized by solving the continuous relaxation of the Big-M or Hull reformulation of the original GDP to obtain a lower bound on the optimum. Branching is then done on the disjunction with an indicator binary variable closest to 1. Two nodes are created at this point: one where the respective indicator Boolean variable is fixed to $true$ (the disjunct is enforced) and another where it is fixed to $false$ (the disjunct is removed from the disjunction). Each node is reformulated and solved to obtain a candidate lower bound. If the solution to a node results in a feasible solution that satisfies all integrality constraints, the solution is an upper bound on the optimum. Any non-integral solutions that exceed an upper bound are pruned from the search tree. The process is repeated until the lower and upper bounds are within the desired tolerance.

2.4.2 Logic-based outer approximation. Logic-based outer approximation is another algorithm which mirrors a standard technique for solving mixed-integer nonlinear programming problems [11]. This approach starts by identifying a set of reduced Non-Linear Programming (NLP) sub-problems obtained by fixing Boolean variables in the different disjunctions such that each disjunct is selected at least once across the set of sub-problems (set covering step). Each sub-problem is solved to obtain an upper bound and a feasible point, about which the objective and constraints of the original GDP are linearized, and solve the resulting problem (via direct reformulation to MILP or via disjunctive branch and bound) to find a lower bound. If the lower and upper bound solutions have not converged, the Boolean variables from the previous solution are fixed and the resulting NLP is solved to find a potentially tighter upper bound solution. The procedure is repeated until convergence is obtained.

2.4.3 Hybrid cutting planes. The cutting planes method is an algorithm for tightening the relaxed solution space of a problem reformulated with Big-M before solving it by adding additional constraints which remove parts of the relaxed space that are disjoint from the actual feasible solution space. These "cuts" to the relaxed solution space are derived from the tighter, hull relaxation of the problem. This algorithm provides a middle-ground for the tradeoff between the complexity and corresponding computational expense of the Hull reformulation with the less tight Big-M reformulation. [25].

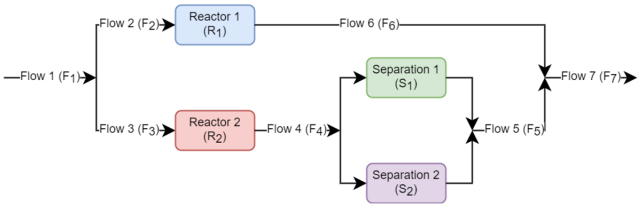

Fig. 4. Illustrative superstructure optimization problem

3. DisjunctiveProgramming.jl

The following section describes the features of the `DisjunctiveProgramming.jl` package and illustrates its syntax with an example from the chemical processing industry for superstructure optimization. The use of nested disjunctions is also shown.

3.1 Features

`DisjunctiveProgramming.jl` allows for defining JuMP models with disjunctions that are directly reformulated via Big-M or Hull methods via the `@disjunction` macro or `add_disjunction!` function. Big-M values can be specified either for the entire disjunction, for each disjunct, or for each constraint in each disjunct. Alternately, if the constraints are linear, the code can use the variable bounds to perform interval arithmetic on each constraint to determine the tightest possible M value to use [1]. For nonlinear GDP constraints, the epsilon approximation formulation for the perspective function in the Hull reformulation is used [12]. Users can specify an epsilon tolerance value to use. Perspective functions are generated by relying on manipulation of symbolic expressions via `Symbolics.jl` [13].

Logical propositions can be added to JuMP models using expressions involving the disjunction indicator variables and the standard Boolean operators ($\Leftrightarrow, \Rightarrow, \vee, \wedge,$ and $\neg$) in an `@proposition` macro. These are automatically converted into CNF and added as integer algebraic constraints to the model. The constraint programming constraints can also be added using the `choose!` function. The expressions are also automatically reformulated into integer algebraic constraints.

Nesting of disjunctions is also supported.

3.2 Example

To illustrate the syntax in `DisjunctiveProgramming.jl` (Version 0.3.2), consider the simple superstructure optimization problem for the chemical process given in Figure 4. In this problem a chemical plant with two candidate reactor technologies (R_1 and R_2) must be designed. If the second reactor technology is chosen, a separation system must also be installed, for which two separation technologies (S_1 and S_2) are available. The GDP model seeks to maximize the product flow (F_7), while discounting for reactor (C_R) and separator (C_S) installation costs as given in (24), subject to the nested disjunction in (25) and the global mass balances in (26) - (27). The system variables are the flows on each stream i (F_i) and the installation costs, with their respective bounds given in (28) - (30). The fixed cost and process yield parameters are given by γ and β, respectively.

$$\max F_7 - C_R - C_S \tag{24}$$

$$\begin{bmatrix} Y_{R1} \\ F_6 = \beta_{R1} F_2 \\ F_3 = 0 \\ F_4 = 0 \\ F_5 = 0 \\ C_R = \gamma_{R1} \\ CS = 0 \end{bmatrix} \vee \begin{bmatrix} Y_{R2} \\ F_2 = 0 \\ F_6 = 0 \\ F_4 = \beta_{R2} F_3 \\ C_R = \gamma_{R2} \\ \begin{bmatrix} Y_{S1} \\ F5 = \beta_{S1} F_4 \\ C_S = \gamma_{S1} \end{bmatrix} \vee \begin{bmatrix} Y_{S2} \\ F5 = \beta_{S2} F_4 \\ C_S = \gamma_{S2} \end{bmatrix} \end{bmatrix} \tag{25}$$

$$F_1 = F_2 + F_3 \tag{26}$$

$$F_7 = F_5 + F_6 \tag{27}$$

$$0 \leq F_i \leq 10 \quad \forall i \in \{1, ..., 7\} \tag{28}$$

$$0 \leq C_S \leq C_S^{max} \tag{29}$$

$$C_R^{min} \leq C_R \leq C_R^{max} \tag{30}$$

The above system can be modeled and reformulated via the Big-M reformulation using `DisjunctiveProgramming.jl`. The resulting JuMP model is then optimized using the HiGHS open-source MILP solver [17] as shown below.

(1) Create the JuMP model and define the model variables and global constraints (mass balances).

```
using DisjunctiveProgramming, JuMP, HiGHS

# create model
m = JuMP.Model(HiGHS.Optimizer)
# add variables to model
@variable(m, 0 <= F[i = 1:7] <= 10)
@variable(m, 0 <= CS <= CSmax)
@variable(m, CRmin <= CR <= CRmax)

# add constraints to model
@constraints(m,
    begin
        F[1] == F[2] + F[3]
        F[7] == F[5] + F[6]
    end
)
```

(2) Define the inner (nested) disjunction for the separation technologies in the superstructure using the `@disjunction` macro.

```
# define inner disjunction
@disjunction(m,
    begin
        F[5] == β[:S1]*F[4]
        CS == γ[:S1]
    end,
    begin
        F[5] == β[:S2]*F[4]
        CS == γ[:S2]
    end,
    reformulation = :big_m, # reformulation type
    name = :YS # symbol for indicator variable
)
```

(3) Define constraints in the outer disjunctions.

```
# define constraints in left disjunct
R1_con = @constraints(m,
    begin
        F[6] == β[:R1]*F[2]
        [i = 3:5], F[i] == 0
        CR == γ[:R1]
        CS == 0
    end
)

# define constraints in right disjunct
R2_con = @constraints(m,
    begin
        F[6] == β[:R2]*F[3]
        CR == γ[:R2]
    end
)
```

(4) Build the main disjunction using the constraint blocks defined in (3) and the `add_disjunction!` function. Note that the reformulated constraints for the nested disjunction are stored in the `.ext` dictionary of the model under the name of the disjunction (`:YS` in this case).

```
# add nested disjunction to model
add_disjunction!(m,
    R1_con,
    # general constraints in R2 disjunction
    (
        R2_con,
        m.ext[:YS] #reformulated inner disj.
    ),
    reformulation = :big_m, # reformulation type
    name = :YR # symbol for indicator variable
)
```

(5) Add the selection logical constraints using the `choose!` function. The first constraint enforces that only one reactor is selected. The second constraint enforces that the separation system be defined only if the second reactor (R_2) is selected. This constraint is equivalent to $Y_{R_2} \Leftrightarrow Y_{S_1} \veebar Y_{S_2}$.

```
choose!(1, YR[1], YR[2]; mode = :exactly)
choose!(YR[2], YS[1], YS[2]; mode = :exactly)
```

(6) Add the objective function and optimize.

```
@objective(m, Max, F[7] - CS - CR)
optimize!(m)
```

4. Future Work

The next steps for the `DisjunctivePrograming.jl` package rely on extending `JuMP.jl` further to allow creating GDP models that are not necessarily reformulated at model creation. Such models will allow using the different GDP solution strategies, such as direct reformulation to MI(N)LP, disjunctive branch and bound, logic-based outer approximation, hybrid cutting planes, and basic steps. The updated package will leverage existing JuMP extension infrastructure and make it possible to define indexing notation for disjunctions, a new Boolean variable type, and a new disjunciton constraint type. These improvements are expected to make the package more usable, flexible, and performant for advanced applications of GDP in JuMP.

5. Related Work

The popular Python package `Pyomo` [4, 16] is widely used for optimization development and includes an extension for generalized disjunctive programming [6]. `GAMS` [3] is a widely used optimization modeling language with support for GDP under the `GAMS EMP` solver that uses `LogMIP` [26]. Research is also being conducted to integrate modern process simulation technology, such as `Aspen`, within the GDP paradigm [22].

6. Conclusion

`DisjunctiveProgramming.jl` is an extension to `JuMP` for creating models for optimization that are formulated according to the generalized disjunctive programming paradigm. The package provides several options for reformulations including the Big-M and Hull relaxations. This package can be used to model problems, reformulate them, and optimize them using existing mathematical programming infrastructure in `JuMP`. This can be useful for industrial and academic applications of GDP, such as superstructure optimization. The capabilities of this package allow for this modeling paradigm to be exploited using `Julia`'s efficient dynamically-typed systems for rapid development, building, and testing of optimization models.

7. References

[1] Ashish Agarwal, Sooraj Bhat, Alexander Gray, and Ignacio E Grossmann. Automating mathematical program transformations. In *International Symposium on Practical Aspects of Declarative Languages*, pages 134–148. Springer, 2010.

[2] Egon Balas. *Disjunctive programming*. Springer, 2018.

[3] Michael R. Bussieck and Alex Meeraus. *General Algebraic Modeling System (GAMS)*, pages 137–157. Springer US, Boston, MA, 2004. doi:10.1007/978-1-4613-0215-5_8.

[4] Michael L. Bynum, Gabriel A. Hackebeil, William E. Hart, Carl D. Laird, Bethany L. Nicholson, John D. Siirola, Jean-Paul Watson, and David L. Woodruff. *Pyomo–optimization modeling in python*, volume 67. Springer Science & Business Media, third edition, 2021.

[5] Qi Chen and Ignacio Grossmann. Modern modeling paradigms using generalized disjunctive programming. *Processes*, 7(11), 2019. doi:10.3390/pr7110839.

[6] Qi Chen, Emma S Johnson, David E Bernal, Romeo Valentin, Sunjeev Kale, Johnny Bates, John D Siirola, and Ignacio E Grossmann. Pyomo. gdp: an ecosystem for logic based modeling and optimization development. *Optimization and Engineering*, 23(1):607–642, 2022.

[7] Ying Chen, Yixin Ye, Zhihong Yuan, Ignacio E. Grossmann, and Bingzhen Chen. Integrating stochastic programming and reliability in the optimal synthesis of chemical processes. *Computers & Chemical Engineering*, 157:107616, 2022. doi:https://doi.org/10.1016/j.compchemeng.2021.107616.

[8] Seolhee Cho and Ignacio E. Grossmann. An optimization model for expansion planning of reliable power generation systems. In Ludovic Montastruc and Stephane Negny, editors, *32nd European Symposium on Computer Aided Process Engineering*, volume 51 of *Computer Aided*

Chemical Engineering, pages 841–846. Elsevier, 2022. doi:https://doi.org/10.1016/B978-0-323-95879-0.50141-7.

[9] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi:10.1137/15m1020575. https://doi.org/10.1137/15M1020575.

[10] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017. doi:10.1137/15M1020575.

[11] Ignacio E. Grossmann. *Logic-based outer approximationLogic-Based Outer Approximation*, pages 1928–1931. Springer US, Boston, MA, 2009. doi:10.1007/978-0-387-74759-0_348.

[12] Kevin C. Furman, Nicolas W. Sawaya, and Ignacio E. Grossmann. A computationally useful algebraic representation of nonlinear disjunctive convex sets using the perspective function. *Computational Optimization and Applications*, 76(2):589–614, 2020. doi:10.1007/s10589-020-00176-0.

[13] Shashi Gowda, Yingbo Ma, Alessandro Cheli, Maja Gwóźdź, Viral B. Shah, Alan Edelman, and Christopher Rackauckas. High-performance symbolic-numerics via multiple dispatch. *ACM Commun. Comput. Algebra*, 55(3):92–96, jan 2022. doi:10.1145/3511528.3511535.

[14] Ignacio E. Grossmann and Sangbum Lee. Generalized convex disjunctive programming: Nonlinear convex hull relaxation. *Computational Optimization and Applications*, 26(1):83–100, 2003. doi:10.1023/a:1025154322278.

[15] Ignacio E. Grossmann and Francisco Trespalacios. Systematic modeling of discrete-continuous optimization models through generalized disjunctive programming. *AIChE Journal*, 59(9):3276–3295, 2013. doi:https://doi.org/10.1002/aic.14088. https://aiche.onlinelibrary.wiley.com/doi/pdf/10.1002/aic.14088.

[16] William E Hart, Jean-Paul Watson, and David L Woodruff. Pyomo: modeling and solving mathematical programs in python. *Mathematical Programming Computation*, 3(3):219–260, 2011.

[17] Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

[18] Paul Jackson and Daniel Sheridan. Clause form conversions for boolean circuits. *Theory and Applications of Satisfiability Testing*, page 183–198, 2005. doi:10.1007/11527695_15.

[19] Donghun Kim. Generalized disjunctive programming-based, mixed integer linear mpc formulation for optimal operation of a district energy system for pv self-consumption and grid decarbonization: Field implementation. *International High Performance Buildings Conference*, 2022.

[20] Sangbum Lee and Ignacio E. Grossmann. New algorithms for nonlinear generalized disjunctive programming. *Computers & Chemical Engineering*, 24(9):2125–2141, 2000. doi:https://doi.org/10.1016/S0098-1354(00)00581-0.

[21] Fahad Matovu, Shuhaimi Mahadzir, Rasel Ahmed, and Nor Erniza Mohammad Rozali. Synthesis and optimization of multilevel refrigeration systems using generalized disjunctive programming. *Computers & Chemical Engineering*, 163:107856, 2022. doi:https://doi.org/10.1016/j.compchemeng.2022.107856.

[22] Miguel A. Navarro-Amorós, Rubén Ruiz-Femenia, and José A. Caballero. Integration of modular process simulators under the generalized disjunctive programming framework for the structural flowsheet optimization. *Computers & Chemical Engineering*, 67:13–25, 2014. doi:https://doi.org/10.1016/j.compchemeng.2014.03.014.

[23] George L. Nemhauser. *Integer and combinatorial optimization*. John Wiley and Sons, 1999.

[24] Francisco Trespalacios and Ignacio E. Grossmann. Improved big-m reformulation for generalized disjunctive programs. *Computers & Chemical Engineering*, 76:98–103, 2015. doi:https://doi.org/10.1016/j.compchemeng.2015.02.013.

[25] Francisco Trespalacios and Ignacio E. Grossmann. Cutting plane algorithm for convex generalized disjunctive programs. *INFORMS Journal on Computing*, 28(2):209–222, 2016. doi:10.1287/ijoc.2015.0669.

[26] Aldo Vecchietti and Ignacio E Grossmann. Logmip: a disjunctive 0–1 non-linear optimizer for process system models. *Computers & chemical engineering*, 23(4-5):555–565, 1999.

[27] Wenjin Zhou, Kashif Iqbal, Xiaogang Sun, Dinghui Gan, Chun Deng, José María Ponce-Ortega, and Chunmao Chen. Disjunctive programming model for the synthesis of property-based water supply network with multiple resources. *Chemical Engineering Research and Design*, 187:69–83, 2022. doi:https://doi.org/10.1016/j.cherd.2022.08.027.