



Optimizing the Trigger Menu Compiler in ATLAS Phase-II Upgrades

Hope Elgart¹

Supervisors: Aimilianos Koulouris², Antoine Marzin²

¹Physics Department, Loyola University Chicago

²Experimental Physics Department, CERN

Abstract

Upgrades to the ATLAS trigger system are critical in handling the expected 40 million crossings per second in the Phase-II operation of the LHC. Located in a human-readable *Trigger Menu* file, there exist certain trigger requirements that facilitate the selection of interesting events from this large stream of collision data. The Trigger Menu Compiler (TMC) configures the file into one that can be interpreted by field-programmable gate arrays (FPGAs). However, due to novel hardware and a change in the implementation of trigger inputs, Phase II requires a new TMC, known as TMC2. We present an investigation on TMC2, including the alteration and refactoring of the current Python scripts.

Contents

1	Introduction	2
2	ATLAS Phase-II Trigger Menu Compiler	3
3	Script Improvements	4
3.1	Trigger Item Processing	4
3.2	Clique ID Counter	4
3.3	Redundancy Checks	5
4	The Knapsack Problem	5
4.1	Current Implementation	5
4.2	Backup Solvers	6
5	Greedy Algorithm	6
5.1	Two Sorts	7
5.2	Troubleshooting	8

1 Introduction

In the Phase-II operation of the LHC, the interactions per crossing are expected to reach roughly 200 as a result of the 40 MHz bunch crossing rate [4]. In order to efficiently handle this increase in data, numerous updates will be made to the Phase-II ATLAS trigger system, which consists of a first-level trigger and a software-based high-level trigger (HLT). The first-level trigger, or the Level-0 (L0) Trigger, is built with custom electronics and collects information from the calorimeters and the muon trigger detectors, as shown in Figure 1.

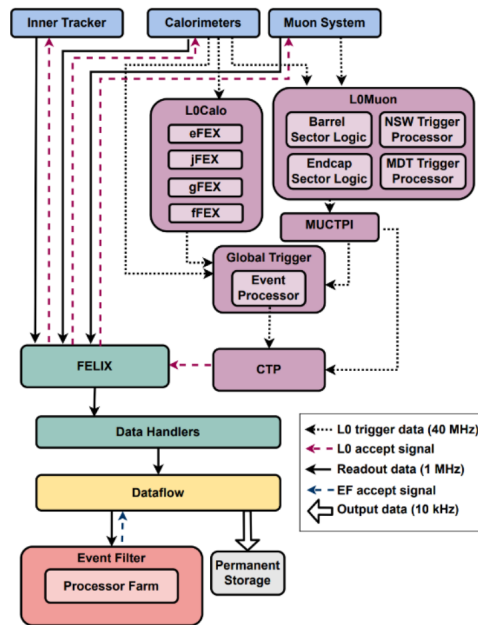


Figure 1: Schematic detailing data-flow in Phase-II upgrades of the ATLAS Trigger and Data Acquisition system [1].

A new feature of the L0 system, known as the Global Trigger, serves as a high-granularity event filter to help refine e/gamma, tau, muon, and jet selections [1]. However, the Central Trigger Processor (CTP) is what ultimately issues the final L0 decision by matching the received detector trigger inputs to any of the pre-defined combinations. Once the L0 accept signal is issued, the trigger and detector data are then transmitted through the Data Acquisition (DAQ) system at 1 MHz. However, as the data is processed by the Event Filter, which consists of software-based high-level triggers, the event rate must be reduced to 10 kHz before being transferred for permanent storage.

Trigger menus, specifically the trigger items contained within them, are a critical component in the data acquisition and selection process [7]. Trigger items are made up of selection algorithms that pick specific signals. Trigger signatures, such as Muons, B-physics, and Jets, are groups of closely correlated trigger items. Related signatures documented in

the same dataset are then further grouped into trigger streams. ATLAS possesses the following four streams: muon, electron/photon, jet/tau/ E_T^{miss} , and minimum bias. The full collection of trigger items, signatures, and streams is located in a trigger menu.

Within the trigger menu, trigger requirements form the L0 trigger logic that the CTP must consider. In the Phase II operation of the LHC, the CTP will encounter up to 1024 trigger logic input bits, each with multiplicity requirements received from the Global Trigger algorithms. 1024 trigger items are then derived from these input bits. Gathered in the trigger menu, any one of these items can trigger the first-level accept signal.

2 ATLAS Phase-II Trigger Menu Compiler

TMC2 handles hundreds of trigger items within a trigger menu, where each item contains logical expressions of multiplicity requirements for the trigger signature. As shown in Figure 2, the TMC2 translates this human-readable menu into an electronically readable file by assigning the incoming trigger items to lookup tables (LUTs), or data structures storing pre-calculated results.

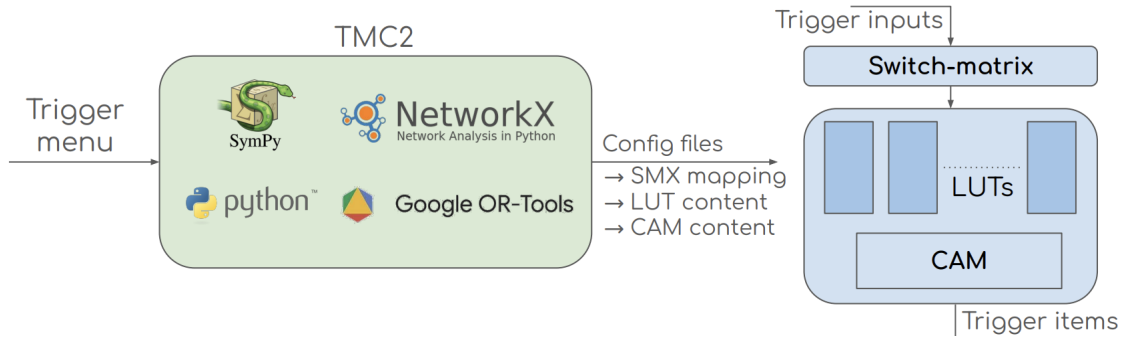


Figure 2: The flow of data and packages within the TMC2 that enable the assignment of trigger items to LUTs in the firmware trigger logic [3].

Trigger items are processed in five stages:

1. The TMC2 first uses the string representation of item descriptions within the trigger menu to form logical variables and expressions that can be interpreted by *SymPy*, or the Python library primarily used for symbolic mathematics.
2. Following simplification of these logical expressions, the logic modules provided by *SymPy* evaluate them into their simplest conjunctive normal form (CNF) [6]. Note that CNF is a conjunction, or logical AND, of clauses, where each clause is a disjunction, or a logical OR.
3. Trigger symbols with the same ORs form cliques. The network analysis, *Networkx*, enables visualization of these cliques and the connected components among the numerous trigger input symbols.
4. The *Google OR Tools* knapsack solver optimally assigns the created cliques to LUTs.
5. Lastly, configuration files are built according to the hardware formatting requirements.

3 Script Improvements

The TMC2 software consists of 16 Python files, each facilitating the compilation of a given trigger menu into a CTP file suitable for an FPGA. Due to this extensive structure, a significant amount of time was spent understanding the intricacies of each script's architecture, in addition to its implementation in the other files. The following subsections detail minor updates and investigations into the TMC2 to ensure proper familiarization and editing accuracy before optimizing the knapsack problem.

3.1 Trigger Item Processing

Upon calling the orchestration script, `tmc2.py`, the compilation begins with the processing of trigger items inside the `Item.py` file via three classes. The classes handle menu items, the logical ORs appearing in menu items, and the created menu cliques. Within `class Item`, several lines of code replace punctuation, such as brackets, dashes, and exclamation points, which would otherwise be misinterpreted by *SymPy* when parsing logical expressions. For example, an incoming trigger item assumes the form "MU8[x1] &! jJ12[x2]", where a bracket denotes the multiplicity requirement and an exclamation point is a logical NOT. Attempting to circumvent this need for a replacement function, three possible solutions were investigated.

sympy.xreplace() method: The Symbols Class converts arbitrary expressions or variables into ones that can be handled by the *SymPy* library [6]. It does so through pattern matching. However, a major issue arises with `xreplace`, as it cannot handle strings before they are converted into a *SymPy* readable expression. Consequently, the Python `replace` method would still need to be implemented before calling `xreplace`.

Python RegEx Package: Similarly, the Regular Expressions package from Python provides regular expression matching to search and replace characters appearing in strings [13]. With no advantages over the current code, this replacement method is almost identical to the one already established, thereby making it inessential to utilize one over the other.

SymPy Parsing Functions: The parsing functions offered by *SymPy* convert strings into *SymPy* expressions [6]. Because the function recognizes an exclamation point as a factorial instead of a NOT and a dash as a subtraction sign, they must first be replaced via the Python `replace` method, again making it redundant to utilize these parsing functions.

3.2 Clique ID Counter

Trigger menus vary in size, with some containing up to 500 trigger items. For a menu of this size, the number of ORs increases to approximately 300, while the number of cliques is around 260. Therefore, it is critical to identify all created cliques and their corresponding ORs when increasing the debug level, which provides more information on each compilation step. This was accomplished through the addition of a running counter, known as the Python `id` object, which calculates a unique integer for a given object that remains constant for the object's lifetime [13]. The object was implemented within `Item.py`, specifically in the menu cliques and ORs classes. An example of the resulting printout is shown in Figure 3.

```

===== this group: ['MU20']
CLIQUE CREATION 0: ['MU20']
===== this group: ['MU10', 'MU4']
CLIQUE CREATION 1: ['MU10', 'MU4']
===== this group: ['ALFA_A7R1L', 'jJ12', 'ALFA_B7R1L', 'ALFA3_A7L1L']
CLIQUE CREATION 2: ['ALFA_A7R1L', 'jJ12', 'ALFA_B7R1L', 'ALFA3_A7L1L']

```

Figure 3: Counter assisting in identification of cliques in the printout.

3.3 Redundancy Checks

After the simplification and CNF process, several redundancy checks ensure the avoidance of overlapping multiplicity requirements. The following checks are currently in place: the AND of two of the same items with differing multiplicities reduces to the item with the higher multiplicity, while the OR of these items only requires the smaller multiplicity. Extensive logical mixing and repetition of item descriptions was used to investigate the limits of these checks, in addition to determining if other redundancy checks are needed. For example, when investigating the statement $(A[x1] \mid B[x1]) \& (A[x2] \mid B[x1])$, it is first simplified to $(A[x1] \& A[x2]) \mid B[x1]$. However, the AND redundancy in the resulting expression is already handled by the aforementioned checks. Similar tests led to either complex group warnings or an expression that the software was already equipped to process.

4 The Knapsack Problem

The knapsack problem (KP) refers to a combinatorial optimization problem. The goal is to maximize the number of items assigned to a knapsack of fixed capacity. Each item has an associated weight and value. For the TMC2, the items are the cliques, with weights corresponding to clique sizes. The values are not significant to the constraints of this sort, thus all values are set to one. Lastly, the knapsacks are the LUTs.

4.1 Current Implementation

Google OR Tools serves as the current knapsack solver. It is an open-source software suite developed for different optimization problems [10]. The TMC2 utilizes an adaptation of the multiple integer programming (MIP) solver, which is a technique involving a linear objective function with one variable assuming either integer or binary values. The definition containing the *OR Tools* solver is named `myKnapsack`, which begins by defining the data, including the number of items and bins, bin capacity, and the weight and value of each item. Following this, the MIP solver is declared. The creation of a 0-1 variable facilitates the differentiation between unassigned and assigned variables, where one indicates that an item has been packed into a bin, and zero if not. Several constraints then ensure that each item is assigned to at most one bin and that the packed amount does not exceed bin capacity. Before the solver is called, an objective maximizes the total value of packed items. Lastly, in the TMC2 adaptation of the software, several error messages indicate if any cliques are not assigned. When this occurs, the KP solution cannot be written to the data file serving as the input for the production of a content-addressable memory (CAM) file that will be loaded onto the CTP.

4.2 Backup Solvers

No matter the complexity of an incoming trigger menu, the MIP *Google OR Tools* solver provides an exact solution, where all cliques are assigned. However, a backup solver is necessary in the event the software changes or is no longer compatible with the needs of the TMC2. The following KP solving approaches were investigated:

Neural Networks: Inspired by the human brain, neural algorithms teach computers how to process data to update parameters of the given model, trying to be solved [9]. In doing so, the computer searches for the best parameters to perform the task.

Greedy Solvers: Greedy algorithms provide a locally optimal problem solving approach [2]. The heuristic makes the best choice at each local step, thereby aiming to provide an optimal global solution.

Dynamic Programming: This method recursively breaks down the larger problem into subproblems, where the result of each subproblem is stored in a table [5]. The storing process avoids redundancies in recomputing problems as the solver progresses.

Neural networks are capable of providing the exact solution that the TMC2 requires; however, the result of a solver, such as *Google OR Tools*, is still needed for training the neural network. Additionally, neural algorithms typically act as a black-box, meaning it is difficult to discern how a solution was calculated. Many dynamic programming examples were studied, but it was difficult to adapt the matrices used to assign items within these simplistic examples to the TMC2. Thus, due to time constraints and feasibility of incorporating the solver into the TMC2 architecture, greedy algorithms were determined as the best sort to implement.

5 Greedy Algorithm

Greedy algorithms quickly seek an optimal local choice, without considering what might be best for the problem as a whole [2]. For the KP, greedy sorts, which are responsible for item assignment, must mirror this urgency while also checking that the solution produced obeys any given constraints. LUT input and output requirements serve as the constraints for the TMC2, where any solver implemented must consider the bin capacities, or the size of the LUT inputs, and the ORs the cliques create.

In the TMC2, the greedy algorithm is contained in one definition, known as `greedy_knapsack`. One can easily toggle between `myKnapsack` and `greedy_knapsack` via the addition of the parameter “-k” after calling `tmc2.py`. Similar to *Google OR Tools*, the greedy definition begins with defining the data. It then lists the constraints and the assignment found in Listing 1.

```
1 assigned = [False] * num_items
```

Listing 1: The Python `False` keyword creates a set of booleans indicating unassigned items via the use of the numerical value zero.

`assigned` tracks the cliques allocated to a LUT during the sorting process in order to prevent the same cliques from being placed into multiple LUTs. Because the TMC2 requires

an exact solution, Listing 1 is also a vital component when troubleshooting, as it identifies unassigned cliques.

Next, a sort facilitates the selection process. The sort informs the solver what items to place into a given knapsack according to this defined heuristic. Because there exists a multitude of sorts that could be considered for any given KP, such as by value, weight, or a value-to-weight ratio, careful consideration was given to the constraints of the TMC2. Additionally, the sort directly impacts the KP solution, meaning different orderings had to be investigated to determine which one could produce an exact solution for any given menu.

5.1 Two Sorts

In the initial stages of designing a greedy algorithm, two types of sorts, both of which are optimal for problems involving items of equal value, were implemented. The first sort, known as a density sort, is shown in Listing 2. Due to the Python `reverse` method, the largest density will be selected first.

```
1 item_order = sorted(range(num_items),
2   key=lambda i: (1/ (weights[i] + lut_output[i])), reverse=True)
```

Listing 2: A density sort of the clique sizes and the LUT outputs

Listing 2 strives to efficiently fill the knapsacks according to the given constraints. The second sort tested is shown in Listing 3 and aims to maximize the amount packed in each bin. Listing 3 is a sort by size that ultimately favors smaller cliques due to the `reverse` method assigning them first.

```
1 item_order = sorted(range(num_items),
2   key=lambda i: (weights[i] + lut_output[i]), reverse=True)
```

Listing 3: Greedy sort by size.

The two sorts were tested against *Google OR Tools* using the Linux `time` command, which measures the execution time for three metrics: real, user, and system [11]. The real time pertains to the actual time elapsed from the moment the script is called to the end of its execution. User time is the amount of time that the CPU needs to perform the user-mode instructions within the process. Note, user mode is a restricted environment where application code and tasks are handled. Lastly, system time is evaluated on behalf of the process as it considers the period of time for the CPU to execute system-level instructions. Table 1 displays each of these times for the greedy sorts and *Google OR Tools*.

Menu Complexity			Size Sort [s]			Density [s]			<i>Google OR Tools</i> [s]		
Items	ORs	Cliques	Real	User	Sys	Real	User	Sys	Real	User	Sys
2	6	3	4.554	2.095	0.781	4.457	2.067	0.816	3.077	2.057	0.733
242	139	77	5.890	9.640	1.256	5.239	9.223	1.263	5.552	9.766	1.339
464	350	271	11.201	15.775	1.604	8.595	16.175	1.377	11.222	17.742	1.579

Table 1: Linux time tests for menus of varying complexity.

About half of the 100 sample menus located in the TMC2 for testing were evaluated using the two separate sorts. Table 1 is indicative of the range of menus tested, as it contains the Linux tests for a simple, medium, and complex menu. According to the `time` command, both sorts performed relatively the same as *Google OR Tools* for the first two menus. However, for the more complex menu, a shorter time was recorded for the real and user time of both the density and size sort, with an approximately 26% difference between the real time of the density sort and *OR Tools*. The greedy sorts may be performing faster, but this is solely due to the fact that they are unable to provide an exact solution. *Google OR Tools* is more methodical and does not finish running until all cliques have been assigned. As a result, more time is needed for more complex menus containing 400 to 500 items. Inversely, the density and size sorts were unable to write the solution into the output data file produced by `tmc2.py` as there was a strong presence of many residual cliques. The unassigned cliques are shown in Table 2.

Menu Complexity			Unassigned Cliques		
Items	ORs	Cliques	Size Sort	Density	<i>Google OR Tools</i>
2	6	3	0	0	0
242	139	77	65	3	0
464	350	271	266	1	0

Table 2: Unassigned cliques for the menus utilized in Table 1.

5.2 Troubleshooting

With no exact solution, an investigation into new sorts or series of sorts was conducted. The first attempted solution was the creation of a Python `for` loop that allowed for two instances of item sorting as shown in Listing 4. According to Table 2, the density sort was the closest in assigning all cliques, thus it was chosen as the first sort. At the end of the density item assignment, the `assigned` items still associated with a `False` Python keyword, were moved into the second sort by size. Initially, there was some trouble in passing the unassigned items into the next loop. Following some troubleshooting and the addition of a print statement at the start of the second sort that indicated the solver continued to the next iteration, items were successfully passed from one sort and into the next. Listing 4 worked for 45 menus, but failed for a menu containing 512 items, 379 ORs, and 274 items. With this menu, 2 unassigned cliques remained.

```

1 for i in item_order_density:
2     for b in range(num_bins):
3         if (bin_input_capacity[b] + weights[i] <= lut_sizes_in[b]):
4             if (bin_output_capacity[b] + lut_output[i] <=
lut_sizes_out[b]):
5                 bin_capacity[b].append(i)
6                 bin_input_capacity[b] += weights[i]
7                 bin_output_capacity[b] += lut_output[i]

```



```

8         assigned[i] = True #if item doesn't fit into a bin
    then it will remain false
9         break
10    if assigned[i] == False:
11        print("could not assign all from 1st sort")
12        for i in item_order_size:
13            for b in range(num_bins):
14                if (bin_input_capacity[b] + weights[i] <=
15                    lut_sizes_in[b]):
16                    if (bin_output_capacity[b] + lut_output[i] <=
17                        lut_sizes_out[b]):
18                        bin_capacity[b].append(i)
19                        bin_input_capacity[b] += weights[i]
20                        bin_output_capacity[b] += lut_output[i]
21                        assigned[i] = True
22                        break

```

Listing 4: For loop facilitating the sorting of unassigned items produced from the initial density ordering.

After analyzing the script for any errors, the order of the sorts was changed such that the density sort now followed the sort by size. Two cliques continued to remain unassigned with this change. New sort parameters were inspected, with the value-to-weight ratio found in Listing 5 serving as the most successful sort implemented during these trials.

```

1 item_order_ratio = sorted(range(num_items), key=lambda i:(values[i]/
    weights[i]), reverse=True)

```

Listing 5: Final ratio implemented in the greedy solver. By the `reverse` method the items will be sorted by decreasing value of the ratio.

Listing 5 was adapted from a fractional knapsack script that encourages the subdivision of items [8]. Similar to Listing 4, the ratio on its own was capable of handling the first 45 menus, but broke down for the 512-item menu. The ratio was then integrated into the two-sort solver. A reordering of the parameters, such that items were processed by size, density, and then value-to-weight ratio, yielded an exact solution for all the menus in the TMC2. Linux `time` tests comparing the performance of the greedy and *Google OR Tools* solvers for a range of menus is shown in Table 3.

Menu Complexity			Greedy [s]			<i>Google OR Tools</i> [s]		
Items	ORs	Cliques	Real	User	Sys	Real	User	Sys
71	47	35	2.693	1.893	0.825	3.303	2.006	0.381
318	191	117	16.001	17.923	31.99	34.229	1.575	1.777
512	377	289	5.891	8.558	1.125	15.744	17.696	1.323

Table 3: Linux time tests for new greedy sorting method evaluating one of the simplest and one of the most complex example menus.

From Table 3, it was discerned that the three consecutive sorts of the Greedy Solver consistently performed faster than the *Google OR Tools* implementation, with a 91% dif-

ference between the two real times of the 512 item menu. Overall, the Linux `time` tests demonstrated that the greedy solver’s recorded real times were faster than the *Google OR Tools* MIP for 83 of the 100 menus tested. The greedy algorithm’s user time was shorter for 80 menus and for 75 menus of the system time results.

Further analysis of LUT utilisation metrics enabled supplementary optimization comparison testing of the two solvers. Table 4 provides the number and percentage of LUT inputs and outputs used by each solver. For all of the 100 menus evaluated, the LUT utilisation metrics recorded for both the greedy and *Google OR Tools* solvers were exactly the same.

Menu Complexity			Solution LUT Inputs		Solution LUT Outputs	
Items	ORs	Cliques	Greedy	<i>Google OR Tools</i>	Greedy	<i>Google OR Tools</i>
71	47	35	79 (15%)	79 (15%)	43 (8%)	43 (8%)
318	191	117	253 (49%)	253 (49%)	187 (35%)	187 (35%)
512	377	289	497 (97%)	497 (97%)	374 (70%)	374 (70%)

Table 4: LUT input and output utilisation metrics for the same menus tested in Table 3.

A final measurement was conducted using the `test_solution.py` script of the TMC2. `test_solution.py` loads the switch matrix, LUT, and (CAM) configuration files, all produced from the solution of `tmc.py`, and checks if the expected trigger items fire according to these solution files. The script ran for a 242-item menu. Although all items passed for both solvers, the passing test for the greedy solver’s solution took 363.30 seconds, while *Google OR Tools* needed 404.66 seconds to process and test all items. Furthermore, the greedy solver’s solution evaluated all possible input bit combinations that could make a trigger item fire at a rate of 143 combinations per second, and *Google OR Tools* assumed a rate of 129 combinations per second.

6 Conclusion

Alterations of TMC2, specifically in its LUT assignment step, were ultimately successful in enhancing overall optimization of the compiler. Metrics, such as Linux `time` tests, LUT utilisation, and `test_solution.py`, used to compare the two solvers, all abide by a common optimality proof for greedy algorithms known as “Greedy Stays Ahead” [12]. The argument states that according to some metric, the greedy algorithm always outperforms or performs the same as an optimal solver, which is taken to be *Google OR Tools*. In all implemented measurements, the three consecutive greedy sorts satisfied “Greedy Stays Ahead”, thereby proving the greedy solver’s ability to optimally create a LUT solution for the TMC2.

References

- [1] The ATLAS Collaboration. Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System. Technical report, CERN, Geneva, 2017.
- [2] Andrea García. Greedy algorithms: a review and open problems, 2024.
- [3] Aimilianos Koulouris. Phase-2 Upgrade of the ATLAS L1 Central Trigger. *ATLAS Collaboration*, 2024.
- [4] Aimilianos Koulouris, Konstantinos Axiotis, Filiberto Bonini, Stefan Ludwig Haas, Anna Malgorzata Kulinska, Luca Marsella, Antoine Marzin, Thilo Pauly, Ondrej Penc, Vladimir Ryjov, Lorenzo Sanfilippo, Rosa Simoniello, Ralf Spiwoks, Paschalis Vichoudis, and Thorsten Wengler. Phase-II Upgrade of the ATLAS L1 Central Trigger. Technical report, CERN, Geneva, 2024.
- [5] Peter N. Loxley and Ka-Wai Cheung. A dynamic programming algorithm for finding an optimal sequence of informative measurements. *Entropy*, 25(2):251, January 2023.
- [6] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.
- [7] Y Nakahama. Designing the ATLAS trigger menu for high luminosities. *Computing in High Energy and Nuclear Physics 2012*, 2012.
- [8] J. Noga and V. Sarbua. An online partially fractional knapsack problem. In *8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)*, pages 5 pp.–, 2005.
- [9] Hazem A. A. Nomer, Khalid Abdulaziz Alnowibet, Ashraf Elsayed, and Ali Wagdy Mohamed. Neural knapsack: A neural network based solver for the knapsack problem. *IEEE Access*, 8:224200–224210, 2020.
- [10] Laurent Perron and Frédéric Didier. Cp-sat.
- [11] Linus Torvalds. *Linux User's Manual*. Linux Documentation Project, 4.3 edition, July 2013.
- [12] Dieter van Melkebeek. From dynamic programs to greedy algorithms, 2025.
- [13] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Style guide for Python code. PEP 8, Centrum voor Wiskunde en Informatica (CWI), 2001.