AI-COPILOT FOR BUSINESS OPTIMISATION: A FRAMEWORK AND A CASE STUDY IN PRODUCTION SCHEDULING

Pivithuru Thejan Amarasinghe

Research Center for Data Analytics and Cognition
La Trobe University
Australia
p.amarasinghe@latrobe.edu.au

Yuan Sun

Research Center for Data Analytics and Cognition
La Trobe University
Australia
yuan.sun@latrobe.edu.au

Su Nguyen

College of Business and Law RMIT University Australia su.nguyen@rmit.edu.au

Damminda Alahakoon

Research Center for Data Analytics and Cognition
La Trobe University
Australia
d.alahakoon@latrobe.edu.au

October 20, 2023

ABSTRACT

Business optimisation refers to the process of finding and implementing efficient and cost-effective means of operation to bring a competitive advantage for businesses. Synthesizing problem formulations is an integral part of business optimisation, which relies on human expertise to construct problem formulations using optimisation languages. Interestingly, with advancements in Large Language Models (LLMs), the human expertise needed in problem formulation can be minimized. However, developing an LLM for problem formulation is challenging, due to training data, token limitations, and lack of appropriate performance metrics. For the requirement of training data, recent attention has been directed towards fine-tuning pre-trained LLMs for downstream tasks rather than training an LLM from scratch for a specific task. In this paper, we adopt an LLM fine-tuning approach and propose an AI-Copilot for business optimisation problem formulation. For token limitations, we introduce modularization and prompt engineering techniques to synthesize complex problem formulations as modules that fit into the token limits of LLMs. Additionally, we design performance evaluation metrics that are better suited for assessing the accuracy and quality of problem formulations. The experiment results demonstrate that with this approach we can synthesize complex and large problem formulations for a typical business optimisation problem in production scheduling.

Keywords Copilot, Large Language Model (LLM), Artificial Intelligence (AI), Business optimisation, Problem Formulation, Production Scheduling

1 Introduction

Business optimisation is an important process to help businesses gain competitive advantages by reducing operational costs, improving customer satisfaction, and mitigating risks. Advances in digital technologies, such as Internet-of-Things and cloud technologies, have enabled new business models with complex operations. Optimising key business decisions (operational, tactical, and strategic) in complex and dynamic systems is challenging and requires the involvement of different stakeholders. Handling business rules and various practical constraints is also not a trivial task. Although modern optimisation technologies have offered businesses different ways to formulate and solve their problems, successfully adopting these technologies still requires significant domain knowledge and optimisation expertise.

While solving business optimisation problems, businesses and optimisation experts engage at different stages. Usually, businesses commence the process by providing a problem description to an optimisation expert. Subsequently, the optimisation expert formulates the problem description into a mathematical model [Antoniou and Lu, 2007] and translates the mathematical model to an executable problem formulation to solve using a solver [Boyd and Vandenberghe, 2004]. Later, the optimisation expert interprets the results and suggests the best actions for the business. As the final step, a software engineer integrates the mathematical models developed by the optimisation expert into the business's systems and applications. Although solving optimisation problems can be handled efficiently by many advanced solvers such as Gurobi Optimization, LLC [2023], Google [2023], Cplex [2009], and meta-heuristics, transforming a problem description to an executable and accurate problem formulation is time-consuming and requires expert knowledge. Poor problem formulations can lead to infeasible solutions (e.g., failure to address constraints and optimise the objective of interest) and significantly slow down solving process.

LLMs have become increasingly popular due to their broad applications. Initiated by transformer [Vaswani et al., 2017] for machine translation, LLMs have quickly adopted within different software and business functions such as analysing business data [Cheng et al., 2023], creating marketing content [Rivas and Zhao, 2023], generating code for visualisations [OpenAI, 2023], supporting programmers with auto-completion [Nguyen and Nadi, 2022], and working as optimisers for simple continuous and discrete optimisation problems [Yang et al., 2023]. With respect to supporting technical users, Salesforce uses code-generating LLMs in its development teams [Le et al., 2022]. Meanwhile, GitHub Copilot [Nguyen and Nadi, 2022] enables code suggestions and code completions for programmers to improve their coding efficiency. Furthermore, Amazon CodeWhisperer [Yetiştiren et al., 2023] helps developers to code efficiently as well as write code related to AWS resources. Meanwhile, ServiceNow has worked with the open-source community to introduce StarCoder [Li et al., 2023] as a free AI code generator. Going beyond supporting technical users, LLMs support non-technical users to implement technical tasks. For instance, code-generating LLMs now enable non-technical users to generate a simple website or create a simple query to retrieve data from a database, without technical support. In fact, the motivation behind this research is to leverage code-generating LLMs to support non-expert users to successfully carry out business optimisations without having to consult experts to significantly reduce traditionally required effort.

Given the nature of problem formulation as a language-to-language translation, code-generating LLMs can be a powerful tool to transform problem descriptions into problem formulations. Furthermore, recent considerable attention towards using LLMs to automate code generation tasks, paves the way to fine-tuning a pre-trained code-generating LLM for problem formulation. Additionally, the introduction of unlabelled data to train LLMs for code generation [Chen et al., 2021], has eliminated most of the limitations in early-stage code-generating LLMs that were trained using labelled datasets [Mastropaolo et al., 2021]. Recently, fine-tuning pre-trained models for downstream tasks has enabled training an LLM for a specific set of tasks with just a hundred or two hundred data points [Solaiman and Dennison, 2021].

However, in general, LLM-based applications in complex decision-making scenarios are still limited. Since existing code-generating LLMs are trained on generic programming problems, problem formulation is a non trivial task for those code-generating LLMs due to complex constraints, different optimisation requirements, and the need of selecting the most suitable optimisation technique. Additionally, due to token limitation, code-generating LLMs cannot generate

large problem formulations, and large computational and memory requirements of some of the code-generating LLMs limit their practical use. Also, the existing performance evaluation metrics in code-generating LLMs are not suitable for problem formulation since the result as well as the optimisation technique need to be considered.

Although machine translation LLMs have been recently fine-tuned for auto-formulation of optimisation models, such models are restricted to mathematical modeling with linear programming [Ramamonjison et al., 2022] or conceptual models that are still in the experimental stage [Tsouros et al., 2023]. Moreover, datasets used in these LLMs contain comparatively smaller problem formulations with limited constraints and variables. As a result, applications of such LLMs are significantly limited in practice since real-world problems often have large numbers of variables and constraints while part of the variables are integers.

Accordingly, we introduce our AI-Copilot as a step towards automating problem formulations for complex real-world optimisation problems. To do so, we select production scheduling as a case study as it has been comprehensively researched in the past and contains complex constraints and different optimisation objectives [Xiong et al., 2022]. We fine-tune a code-generating LLM, which uses limited memory, and computational resources, using a data set created by us that comprises 100 pairs of problem descriptions and formulations. As a result, we minimize the requirement of large training data using this fine-tuning step without training an LLM from scratch for problem formulation. In addition, we apply modularization and prompt engineering techniques on our AI-Copilot to cater to token limitations when formulating complex problem formulations. Furthermore, we use loss and execution based performance evaluation metrics to assess the accuracy and quality of problem formulations compared to existing evaluation metrics.

In contrast to existing machine translation LLM based auto-formulation models such as Ramamonjison et al. [2022], our method performs text-to-code translation and formulates constraint programming problems. Moreover, our AI-Copilot can formulate complex problem formulations compared to existing machine translation LLM based auto-formulation models [Ramamonjison et al., 2022, Tsouros et al., 2023]. Therefore the contributions of this paper toward automating problem formulations could be highlighted as:

- Constructing an open-source dataset with production scheduling to fine-tune a code-generating LLM for problem formulation.
- Fine-tuning a code-generating LLM for problem formulation that consumes limited computing resources.
- Developing a modularization and prompt engineering technique to manage large problem formulations.
- Designing suitable performance metrics for assessing the accuracy and quality of problem formulation.

2 Literature Review

2.1 Business Optimisation and Optimisation Technologies

Business optimisation has been described as the "Philosophy of Continuous Improvement" [Singh and Singh, 2009], where businesses attempt to make their operation as perfect as possible. Generally, business optimisation covers all processes and efforts to improve productivity, efficiency, performance, etc. However, this research considers business optimisation from a computational and mathematical perspective where one tries to minimize or maximize an important characteristic of a process by an appropriate choice of decisions [Kallrath and Wilson, 1997]. For example, problem descriptions are formulated as mathematical models and solved using solvers to provide suggestions for business decisions. Traditionally, combinatorial optimisation is a class of optimisation, that is used for mathematical and computational requirements of business optimisation [Yu, 2013]. In spite of the benefits from business optimisation to businesses, challenges such as intensive computational and human expertise requirements and inadequate support structure for business optimisation may exist. While past studies have predominantly focused on algorithmic improvements for business optimisation, our proposed AI-Copilot will focus on improving support structure and reducing intensive human expertise requirements.

In general combinatorial optimisation is known to be the process of finding the minimum or maximum of an objective function that has a large discrete domain. Such a process is needed in real-world scenarios such as vehicle routing problem to select the optimal set of routes to serve a given set of customers [Toth and Vigo, 2002], bin-packing problem for multiprocessor scheduling [Coffman Jr et al., 1978], cut waste reduction problem in furniture manufacturing [Kłosowski et al., 2018], production scheduling problem [Pochet and Wolsey, 2006], and many more. The solution space for such a scenario is too broad for a pure brute-force approach. Therefore, algorithmic techniques such as dynamic programming, branch and bound, random-restart hill climbing, simulated annealing, genetic algorithms, and tabu search are developed. From a computer science perspective, the above algorithmic techniques reduce solution space or accelerate solution search using mathematical methods.

Recently considerable attention has been directed towards applying machine learning techniques for combinatorial optimisation rather than traditional mathematical improvements to combinatorial optimisation algorithms. Although the latest research focuses on improving combinatorial optimisation techniques using machine learning, our AI-Copilot focuses on generating problem formulations related to combinatorial optimisation.

2.2 Problem Formulation and Solvers

An optimisation problem can be formulated by selecting one or more optimisation variables, an objective function, and constraints. Such optimisation problems can be categorised into unconstrained optimisations, simple bound constraints, and constrained optimisations. While the name suggests that unconstrained optimisations have no constraints, simple bound constraints have boundaries for design parameters, but no constraints on the solution. However, constrained optimisation is the most complex category of optimisation problems, where the solution must satisfy a set of linear or nonlinear constraints and bounds to design parameters.

Unquestionably solvers embed powerful algorithms to solve problem formulations. Nevertheless, solvers are different from each other, due to the facts such as computational efficiency and solution strategies. Despite Gurobi Optimization, LLC [2023] being the state-of-art commercial solver for mathematical programming which solves a wide range of problems including linear programming and mixed integer programming, SCIP [Bestuzheva et al., 2021] is the fastest non-commercial solver currently available for mixed integer programming and mixed integer non-linear programming. In addition, Gurobi Optimization, LLC [2023] is more competitive over SCIP in solving complex problem formulations [Avella et al., 2023].

More importantly, optimisation languages like MiniZinc [Nethercote et al., 2007], GAMS [Soroudi, 2017], and AMPL [Fourer et al., 1990] supply problem formulation specific syntax to users to represent a mathematical model related to a problem description. But due to differences between optimisation languages over each other, users require significant training to master an optimisation language. Such differences mainly happen in solving capability, licencing approach, expressiveness of syntax, and documentation. Furthermore, due to transforming a problem description from a native language to an optimisation language is time consuming, optimisation languages limit the application of business optimisation in a wide range of problems. Yet our proposed AI-Copilot will go beyond existing technologies to bridge this gap and reduce the requirement of mastering optimisation languages.

2.3 Large Language Models and Code Generation

Considerable attention has been recently directed towards LLMs, since LLMs can perform a wide range of tasks such as writing academic literature [Lund et al., 2023], question answering [Wang et al., 2019], language translation [OpenAI, 2023], code generation [Le et al., 2022], among others. Zan et al. [2022] report a comprehensive study on twenty-seven such code-generating LLMs. Unquestionably transformer-based machine translation [Vaswani et al., 2017] has paved the way for code generation using LLMs, and the first code-generating LLMs were trained using labelled datasets [Mastropaolo et al., 2021]. These models had limitations since such techniques had practicality issues of requiring labelled data to even fine-tune an LLM for code generation. However, promising results could be seen with

the introduction of unlabelled data to train LLMs for code generation by Chen et al. [2021]. Since Chen et al. [2021] is a code-generating LLM fine-tuned using a large corpus of GitHub Python code, it can cover a broader spectrum.

Meanwhile Le et al. [2022] have improved the quality of generated code by considering the results of test execution with reinforcement learning techniques. Accordingly, they use an actor-critic network to supply rewards for generated code and use these rewards to further improve its code generation capability. Furthermore, Zhang et al. [2023] introduce a model agnostic planning process based on Markov's decision process, as planning capabilities for code-generating LLMs can generate more accurate code for problem descriptions by considering future scenarios. They use the beam search for the evaluation process since the tree search-based planning process developed by them is inspired by the Monte-Carlo tree search. As such techniques can cause computational efficiency-related discrepancies that come with the Monte-Carlo tree search algorithm, they suggest caching beam-search used in the code-generating LLM.

Benchmarks and metrics play a key role in finding progressive improvements in the code-generating capabilities of LLMs. APPS [Hendrycks et al., 2021] benchmark includes 10,000 programming problems and their solutions in Python. A significant feature of this benchmark is that it has problem descriptions closer to the natural language. Furthermore, it includes simple problems as well as complex algorithms. Meanwhile, the HumanEval [Chen et al., 2021] benchmark includes 164 hand-written programming problems, with a focus on measuring the functional correctness of generated code. Each programming problem of HumanEval has a function signature, docstring, body, and several unit tests. Nonetheless, the major difference between APPS and HumanEval is that problems included in HumanEval are new and do not contain solutions in GitHub. Since for a sizeable part of APPS benchmark problems, there are solutions in GitHub, for code-generating LLMs trained using GitHub data, HumanEval is more precise compared to APPS. Conversely, the MBPP [Austin et al., 2021] benchmark includes 974 Python programming problems, which suit entry-level programmers. Meanwhile, some specialised metric named pass@k has been proposed by Kulal et al. [2019] to evaluate generated code, where for a particular problem description k number of solution codes are generated. Generated codes are run against a test case related to a programming problem and the programming problem is considered solved if at least one solution code out of k can pass the test case.

The existing problem formulation research uses NLP techniques [Ramamonjison et al., 2022] in contrast to the codegenerating LLM based problem formulation automation approach introduced by our AI-Copilot. In addition, recent problem formulation research is based on the NL4Opt [Ramamonjison et al., 2023] dataset and it contains simpler problem descriptions and linear programming problem formulations related to them. Furthermore, according to the statistics, the NL4Opt dataset contains on average 2.08 variables and 2.83 constraints per problem formulation. Recently, Tsouros et al. [2023] introduced a conceptual framework for text to code transformation for problem formulation using prompt engineering techniques with generic LLMs. However, experiment results of the framework have not been released as yet.

2.4 Machine Learning in Combinatorial Optimisation

Recent attempts have been made by machine learning and operational research communities to leverage machine learning for combinatorial optimisation. Modern combinatorial optimisation algorithms use handcrafted heuristics to make decisions that are computationally expensive or mathematically complex [Bengio et al., 2021]. Since patterns for efficient heuristics can be discovered by observing data from the optimisation process (e.g., search historial, optimal solutions), machine learning is naturally a good candidate to enhance decision making in optimisation methods. Therefore, some past studies have applied reinforcement learning to make low-level optimisation decisions based on the dynamic states of an optimisation process. Furthermore, approximations can be learned through imitation learning because of demonstrations done by an expert on how a model should behave.

Recently, Baltean-Lugojan et al. [2018] introduce a neural network-based model to estimate the objective of a time-consuming semidefinite programming problem to decide the most promising cutting planes. With regard to branching policies in branch-and-bound trees of mixed integer linear programming, Gasse et al. [2019] introduce a neural network

to learn strong branching [Cook et al., 2011] approach. For container pre-marshalling problems, Hottung et al. [2020] introduce convolutional neural networks for learning branching policy and estimating the value of partial solutions. Going further beyond, for the traveling salesman problem, Khalil et al. [2017] leverage graph neural networks to learn selection criteria for the next node.

2.5 Summary

Recently, shifting boundaries using AI to enhance workplaces [Jarrahi et al., 2023] has become a popular topic. In fact, it focuses on introducing virtual assistants to help employees in an organization. In contrast to other domains, the business optimisation domain lacks such tools to ease the burden on the people involved. Going further, the application of chatbots builds a more convenient workplace for employees [Wang et al., 2023]. Even though LLMs can be treated as perfect candidates for supporting such requirements, the application of code-generating LLMs for problem formulation introduces several challenges such as training data, token limitations, evaluation metrics, and expensive computational resources required for LLM execution. Therefore research in this area is timely and critical.

3 Case Study

Job Shop Scheduling (JSS) is one class of combinatorial optimisation problems that is common in manufacturing [Pinedo, 2005]. Due to its practical constraints and complex nature, JSS is one of the most popular optimisation problems investigated by researchers in operational research and computer science. Furthermore, JSS is treated as one of the renowned NP-hard problems in literature. The goal of JSS is to schedule a number of jobs over a number of machines, and each job consists of a set of operations that need to be executed in a given order on the allocated machine. In addition, the machine is allowed to process one operation at a time, and different objectives such as makespan and weighted tardiness can be minimized. It should be noted that methods such as integer programming [Ku and Beck, 2016], metaheuristics [Kreipl, 2000], and constraint programming [Beck et al., 2011, Watson and Beck, 2008] can be used to solve JSS.

For the static JSS problem instance, the shop, such as, the working or manufacturing environment includes a set of M machines and N jobs that need to be scheduled. Each job j has its own pre-determined route through a sequence of machines to follow and its own processing time at each machine it visits. The following notation is used to define the mathematical model for the JSS [Nguyen et al., 2021].

Parameters:

- $J = \{1, ..., j, ..., N\}$: the set of all jobs
- n_i : the number of operations of job j
- $route_j = (m_{j1}, ..., m_{jn_j})$: the sequence of machines that job j will visit, where m_{ji} is the machine that processes the i^{th} operation of job j
- $time_j = (p_{j1}, ..., p_{jn_j})$: the processing times of all operations of job j, where p_{ij} is the processing time of the i^{th} operation of job j
- r_i : the release time of job j
- d_i : the due date of job j
- w_i : the weight of job j

Variables:

- s_{ii} : the starting time of the i^{th} operation of job j
- e_{ji} : the ending time of the i^{th} operation of job j

- C_j : the completion time of job j
- T_j : the tardiness of job j calculated by $T_j = \max(C_j d_j, 0)$

The constraint programming formulation for the JSS is defined as follows.

$$\forall j \in J : s_{i1} > r_i \tag{1}$$

$$\forall j \in J, i \in \{1, ..., n_j\} : e_{ji} = s_{ji} + p_{ji} \tag{2}$$

$$\forall j \in J : C_j = e_{jn_j} \tag{3}$$

$$\forall j \in J : T_j = \max(C_j - d_j, 0) \tag{4}$$

Where (1): starting time of the first operation of the job should be greater than the release time of the job, (2): ending time of an operation equals to the sum of starting time and processing time of an operation, (3): completion time of a job equals to the ending time of the last operation of the job, (4): tardiness of a job equals to the difference between the job completion time and the due date of the job if it is positive or zero otherwise.

To ensure no overlap between operations (or disjunctive constraints) on the same machine:

$$\forall j, k \in J, u \in \{1, ..., n_j\}, v \in \{1, ..., n_k\},$$

$$m \in route_j, o \in route_k : m_{ju} = o_{kv} \Rightarrow s_{ju} \ge e_{kv} \lor s_{kv} \ge e_{ju} \quad (5)$$

That is if operations u and v from different jobs are to execute on the same machine $m_{ju} = o_{kv}$, the start time of one of these jobs must be greater than the end time of the other job.

There are a number of precedence constraints between the operations of a job:

$$\forall j \in J, i \in \{1, ..., n_j - 1\} : s_{j,i+1} \ge e_{ji} \tag{6}$$

The objective functions are defined as follows:

• Makespan: Defined variable C_{\max} which represents the latest completion time of any job. The objective is to minimise C_{\max} subject also to constrain (8):

$$\min C_{\max} \tag{7}$$

$$\forall j \in J : C_{\text{max}} \ge e_{jn_j} \tag{8}$$

• Maximum tardiness: Defined variable T_{max} which represents the maximum tardiness of any job. The objective is to minimise T_{max} subject to constrain (10):

$$\min T_{\max} \tag{9}$$

$$\forall j \in J : T_{max} \ge T_j \tag{10}$$

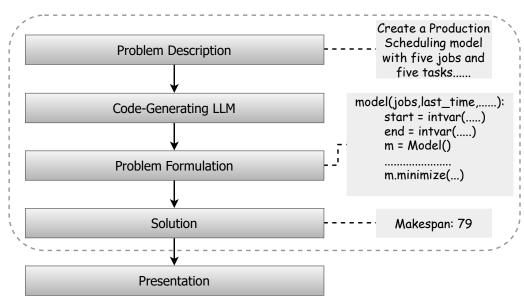
• Total Weighted Tardiness (TWT): The objective is to minimise cumulative tardiness' across all jobs:

$$\min \sum_{j \in J} w_j T_j \tag{11}$$

4 Proposed Method

4.1 Overview

Figure 1: Solution overview



Our approach is conceptually represented in Figure 1, which comprises of five main components:

- Problem Description which aims to capture the business optimisation scenario, that is going to be formulated.
- Code-Generating LLM which synthesizes a problem formulation from a problem description
- Problem Formulation which is the generated problem formulation for a given problem description, that can be solved using a solver.
- Solution which is the final result obtained after solving a problem formulation using the solver.
- Presentation which interprets a solution and suggests the best actions for a business optimisation scenario.

The proposed AI-Copilot is designed to facilitate the first four components of the conceptual framework. Apart from this if we need to study the mathematical properties of a problem description, a mathematical formulation can also be generated by using a similar approach. So an optimisation expert can verify the mathematical formulation. Although verification from an optimisation expert will have a human dependency, our AI-Copilot has reduced human effort through automation. The remaining sections of this article will focus on how our AI-Copilot is developed based on our conceptual model.

4.2 Pre-Trained Model

The pre-trained model acts as the base model for developing code-generating LLM for problem formulation. In fact, using a pre-trained model reduces training time and training resource requirements. More importantly, a suitable pre-trained model should have excellent code-generating capabilities with minimum resource requirements. Accordingly, we use CodeRL as the pre-trained model due to relatively fewer resource requirements, availability as a free model, and excellent code-generating capabilities.

The underlying unified encoder-decoder architecture of CodeRL sums up to a size of 60M~770M parameters [Le et al., 2022]. Moreover, CodeRL is an advanced version of CodeT5 [Wang et al., 2021] trained on GitHub data, which holds

48 layers, 16 attention heads, and 1024 hidden states. Though CodeRL is significantly smaller compared to Codex [Chen et al., 2021], and GPT-3 [Brown et al., 2020] introduced by GPT, CodeRL has been able to perform well [Le et al., 2022]. The reason for such performance from CodeRL is that it considers the test case execution status of a generated code as a fine-tuning approach with an actor-critic reinforcement learning technique. Since CodeRL performs the actor role, a separate critic network assesses solutions generated by CodeRL, with test cases related to problem descriptions, and supplies a critic score for reinforcement learning.

Despite CodeRL showing promising results with code generation, CodeRL is not capable of problem formulation and it is the same for GPT-4 [OpenAI, 2023] (Figure 2). Furthermore, facts such as the GitHub-based dataset of CodeRL does not hold a substantial amount of problem formulation-related samples, and the six hundred token limit makes CodeRL incompetent with problem formulation. Moreover, such generic code-generating LLMs might provide problem formulations to problem descriptions, but we are not satisfied with the generated problem formulations, because generic code-generating LLMs are not specifically trained on problem formulation. In addition, by using in-context learning available in generic code-generating LLMs, users might be able to generate problem formulations, but due to the complexity of problem descriptions and effort that needs to be put in by users, generic code-generating LLMs may not be ideal for problem formulation. Such data and token limitations may be common for code-generating LLMs, but for problem formulation, those limitations should be managed. In the following sections, we will show how our AI-Copilot is capable of synthesizing complex problem formulations by fine-tuning CodeRL with prompt engineering techniques.

Figure 2: Problem Formulation with Generic Code-Generating LLMs: For the same problem description, Figure 2a shows the problem formulation generated by CodeRL, and Figure 2b shows the solution after solving the problem formulation generated by GPT-4

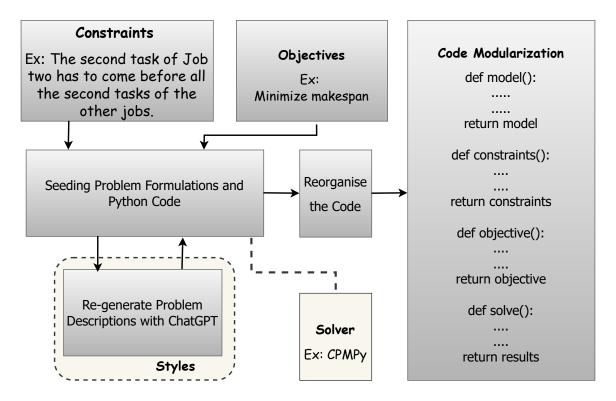
```
#!/usr/bin/env python
import random
import math
                                                       KeyError
                                                                                      Traceback (most
def makespan(x):
                                                       recent call last)
  return math.sqrt(x)
                                                       Input In [3], in <cell line: 35>()
                                                                for job in range(num_jobs):
def main():
                                                          37
                                                                  if iob != 1:
  iobs = [random.randint(1.5) for i in range(5)]
                                                       ---> 38
                                                                      model += x[1, routes[1][1], t] <= x[job,
  machines = [random.randint(1,5) for i in
                                                       routes[job][1], t + durations[1, routes[1][1]]]
                                                          40 # Step 3: Solution
range(5)]
                                                          41 model.solve()
  for i in range(5):
     for j in range(5):
                                                       KeyError: (0, 1, 20)
        for k in range(5):
           for I in range(5):
                for m in range(5):
                      for n in range(5):
```

(a) CodeRL does not create a complete problem formulation due to token limitation

(b) GPT-4 does not properly formulate the constraint of "Second task of Job two has to come before all the second tasks of other jobs"

4.3 Dataset Development

Figure 3: Dataset Development: First we create different problem descriptions with their problem formulations based on different objectives and constraints. Syntax in problem formulations depends on the solver and programming language. Next, we regenerate problem descriptions with different styles using ChatGPT to add the individuality factor. Finally, we reorganise and break down the code into modules. So we can use them when fine-tuning the code-generating LLM.



Since it is rare to find publicly available problem formulation data, we manually create a hundred production scheduling problem descriptions with their problem formulations (Figure 3). To create problem formulations, we have selected CPMpy [Guns, 2019] as the solver and Python as the programming language. Furthermore, we use different styles to add the individuality factor of different business stakeholders into problem descriptions. Therefore, we use ChatGPT [OpenAI, 2023] with prompt engineering to transfer problem descriptions into different styles [Reif et al., 2021]. Due to the fact that problem formulations related to business optimisations are lengthier compared to normal programming code and written in optimisation languages supported by solvers, our dataset contribute towards filling the gap that exists in datasets for business optimisation problem formulation. Additionally, the methodology of our AI-Copilot can be used by businesses to develop their own AI-Copilot using their data. Such applications will allow businesses to dynamically adapt optimisations based on business environment changes.

The problem descriptions in our dataset have less than six hundred tokens, and respective problem formulations have tokens in the range of 1200 to 1800. Additionally, our dataset holds different scenarios related to production scheduling to capture different requirements that are encountered in production scheduling (Table 1). However, large problem formulations included in our dataset exceed the token limits of the code-generating LLMs. To address this, we use prompt engineering techniques that will generate problem formulations as modules that fit into the token limits of code-generating LLMs. In contrast to existing datasets such as NL4Opt, our dataset focuses on constraint programming

¹Dataset has been made publicly available on GitHub at: https://github.com/pivithuruthejanamarasinghe/AI-Copilot-Data.

problem formulations that involve a significantly larger number of constraints and variables. A sample problem description can be,

Job shop scheduling model with 5 jobs and 5 machines. All jobs have random routes and their operations have random durations. The objective function is makespan. Maximum duration is 20. After solving the problem, solutions will be printed and visualised. Note: The second task of Job two has to come before all the second tasks of other jobs.

Table 1: Problem Formulation Scenarios

Scenario Type	Example Scenario
Task completion precedence	The second task of Job two has to come before all the second tasks of the other jobs.
Introduction of release times	The release time of a job is a random value from 0 to 50. The jobs cannot start before their release time.
Minimize makespan	The objective function is makespan.
Minimize maximum tardiness	The due dates are calculated based on the total processing time of each job multiplied by a due date allowance of 1.3. The objective function is maximum tardiness.
Minimize weighted tardiness	The due dates are calculated based on the total processing time of each job multiplied by a due date allowance of 1.3 . The release time of a job is a random value from 0 to 50 . Jobs cannot start before their release times. Each job has a weight following a random distribution in which 20% will have a weight of $1,60\%$ will have a weight of 2 , and 20% will have a weight of 4 . The objective function is total weighted tardiness.
Minimize total flow time	The objective function is total flow time (completion time – release time).
Minimize total weighted flow time	The objective function is the total weighted flowtime.

Due to the fact that there are scenarios where problem formulations must generate random dummy data to make them solvable via a solver, particular statements have been added to problem descriptions to keep the consistency of generated random dummy data. In addition, the dataset holds scenarios to capture metric conversion while generating problem formulations (Table 2). Finally, to get consistent outputs from problem formulations, the dataset has configured a

Table 2: Data Consistency Statements

Statement Type	Example Scenario
Maximum duration	The maximum duration is 20.
Release time range	The release time of a job is a random value from 0 to 50.
Metric types	Jobs two and four will have task durations in minutes. The other jobs will have task durations in seconds.

random seed (random.seed(1)) for all random generations.

4.4 Fine-Tuning

In order to fine-tune the pre-trained model CodeRL for problem formulation, we use a trainer [Hugging Face, 2023], which follows a loss-based approach. Even though the loss-based approach preserves qualitative aspects of generated

problem formulations, we compare the final solution with the actual solution to measure accuracy. Training configurations are available in Table 3, and we fine-tune the code-generating LLM on these configurations based on parameters available in Table 4. Furthermore, we pick batch size and epoch count as primary parameters for hyper-parameter tuning since they affect the learning frequency of the code-generating LLM. We will show in the next section that this parameter setting has already generated promising results, and therefore we will not further tune parameters such as learning rate.

Table 3: Training Configurations

Training Configuration	Value	
GPU type	NVIDIA Tesla V100 SXM2 32 GB	
Pre-trained model	Salesforce/codet5-large-ntp-py	
Tokenizer	Salesforce/codet5-large-ntp-py	
Learning rate	5e - 05	
Gradient checkpointing	True	
Evaluation strategy	steps	
Evaluation steps	10	
Logging steps	10	
Do Evaluation	True	

With the intention of fulfilling scalability aspects such as token limits of code-generating LLMs, we modularize problem formulations using instructions. As instructions (Figure 4), we use nine prompts that allow the code-generating LLM to create problem formulations part by part. In the end, we combine all problem formulation modules to create a final problem formulation for a particular problem description. In the modularization process, we amend the first dataset by attaching instructions as suffixes for each problem description. Correspondingly each problem description becomes nine different problem descriptions, and all together we produce nine hundred problem descriptions. Furthermore, we modularize each problem formulation into nine different sub problem formulations to align with the instructions. As an outcome of the modularization process, the original hundred problem formulations get increased to nine hundred sub problem formulations. Below is a sample modularized problem description, and some of its modularized problem formulation can be seen in Figure 4.

Create a job shop scheduling model with 6 jobs and 6 machines. All jobs have random routes and their operations have random durations. The due dates are calculated based on the total processing time of each job multiplied by a due date allowance of 1.3. The release time of a job is a random value from 0 to 50. Jobs cannot start before their release times. Each job has a weight following a random distribution in which 20% will have a weight of 1,60% will have a weight of 2, and 20% will have a weight of 4. The objective function is the total weighted flowtime. Maximum duration is 20. After

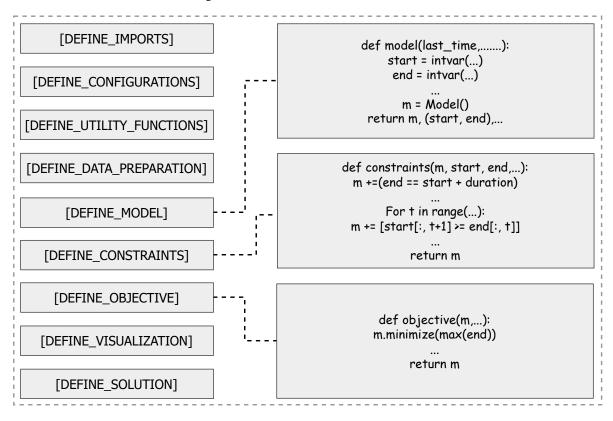


Figure 4: Instructions-based code modules

solving the problem, solutions will be printed and visualised. Note: Jobs two and four will have task durations in minutes. Other jobs will have task durations in seconds.[DEFINE_CONSTRAINTS]

4.5 Performance Metrics

We use training loss, training time, and problem formulation execution status to evaluate the performance of our AI-Copilot. The training loss is calculated by the trainer, by comparing a generated output with a target output for a particular problem description using cross-entropy loss:

$$l(x,y) = \frac{\sum_{n=1}^{N} l_n}{N},\tag{12}$$

$$l_n = -\log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} .1\{y_n \neq ignore_index\},$$

$$(13)$$

where x: logits from the code-generating LLM for a given problem description (generated problem formulation), y: target ids (target problem formulation), $ignore_index$: -100, C: number of classes, N: mini-batch dimension.

Since the cross-entropy loss is inadequate to ensure the executability and correctness of generated problem formulations, we solve problem formulations using a solver and compare the final solution with the actual solution. It should be noted that, for problem formulation executions, there are three possible outcomes which are, getting correct output, getting incorrect output, and failure due to syntax errors. Accordingly, we introduce success rate, failure rate, and exception rate to evaluate the performance of code-generating LLMs for generating problem formulations. In fact, a combination of such metrics allows us to pick the best code-generating LLM that suits problem formulation.

5 Experiments

5.1 Overview

We randomly allocate 70% of the dataset as training data, 10% as validation data, and 20% as test data. While the training and validation data are used in the fine-tuning process to avoid any overfitting and underfitting scenarios, the test data is used to evaluate the performance of the fine-tuned code-generating LLM. The parameters and metrics used in the training, validation, and testing stages are available in Tables 4 and 5.

Table 4: Parameter Definitions

Parameter	Definition			
batch size	number of data points for a batch			
epoch	number of training iterations			
	Table 5: Metric Definitions			
Metric	Definition			
loss	generated by comparing a target problem formulation and a generated problem formulation			
time	number of seconds to complete training			
success	rate of problem formulation successfully solved and giving correct output			
failure	rate of problem formulation successfully solved and giving incorrect output			
exception	rate of invalid problem formulations			

5.2 Training and Testing Performance

The training results are available in Table 6. While reviewing the results, we identify some key observations. One such observation is that, in general, loss values are low for all batch sizes. But failure and exception rates are significantly high for lower epoch counts. On the other hand, we observe a significant improvement in success rates with the increase in epoch count. Furthermore, we see that for lower epoch counts, due to minor errors in generated problem formulations, the execution of problem formulations fails. Expectedly, we identify a slight increase in loss and a significant increase in the failure and exception rates with the increase in batch size. However, when increasing the epoch count, success rates are getting improved for larger batch sizes. Based on the aforementioned observations we infer that the batch size and epoch count are key contributors in obtaining better problem formulation capabilities. While observing testing results (Table 7), we observe the same patterns as that in the training results. For instance, in a similar manner to the training results, the success rate of the testing results increases with the epoch count, and the failure and exception rates

reduce with the epoch count. From such similar training and testing observations, we infer that the fine-tuning process has not led the code-generating LLM to an overfit scenario.

Table 6: Training results of metrics based on batch size and epoch count

batch size	epoch	loss	time(sec)	success	failure	exception
1	1	0.0056	1083.78	0.16	0.23	0.61
1	2	0.0068	2105.51	0.00	0.00	1.00
1	4	0.0038	4216.54	0.00	0.01	0.99
1	8	0.0008	8561.05	0.96	0.00	0.04
2	1	0.0363	675.19	0.00	0.00	1.00
2	2	0.0027	1256.11	0.44	0.20	0.36
2	4	0.0014	2554.93	1.00	0.00	0.00
2	8	0.0008	5153.64	1.00	0.00	0.00
4	1	1.4474	429.63	0.00	0.00	1.00
4	2	0.0043	849.15	0.24	0.00	0.76
4	4	0.0017	1733.71	0.97	0.00	0.03
4	8	0.0010	3446.56	0.97	0.00	0.03

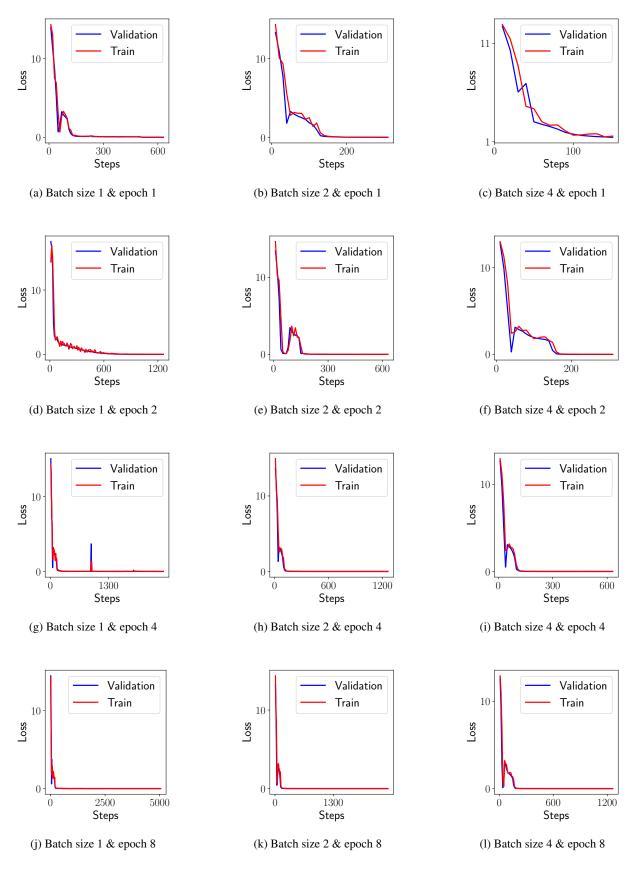
Table 7: Testing results of metrics based on batch size and epoch count

batch size	epoch	success	failure	exception
1	1	0.25	0.30	0.45
1	2	0.00	0.00	1.00
1	4	0.00	0.00	1.00
1	8	0.95	0.00	0.05
2	1	0.00	0.00	1.00
2	2	0.45	0.15	0.40
2	4	1.00	0.00	0.00
2	8	1.00	0.00	0.00
4	1	0.00	0.00	1.00
4	2	0.15	0.00	0.85
4	4	0.90	0.00	0.10
4	8	0.95	0.00	0.05

5.3 Convergence

We utilize validation error in each parameter setting to investigate the loss convergence of the code-generating LLM. While we observe in all parameter settings that the validation and training curves overlap at some point (Figure 5), for batch size one, overlapping happens at the early stages. But, such overlapping behaviour we do not observe in batch sizes two and four. In fact, the learning frequency of the code-generating LLM reduces with the increase in batch size, and batch size may cause the initial gap in training and validation curves. Unquestionably, with the increase of the epoch count even for the larger batch sizes, we see overlapping training and validation curves after having adequate learning iterations. Going further beyond, if we try to relate how success rate maps with the overlapping training and validation curves, we observe that perfect overlapping training and validation curves mean higher success rates. As we generate problem formulations as modules for a particular problem description, and combine them at the end, even minor mistakes can cause incorrect results and exceptions. Since perfectly overlapping training and validation curves mean perfect training, perfect training makes the code-generating LLM avoid minor mistakes.

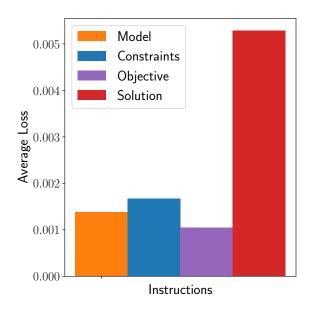
Figure 5: The behavior of the training and validation loss for different settings.

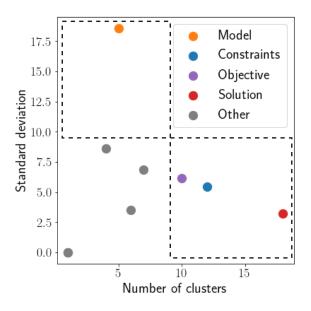


5.4 Loss Analysis

While simpler problem formulation modules have a negligible loss, problem formulation modules with more complexities such as defining model, constraints, objective function, and solution have contributed more towards the loss (Figure 6a). Additionally, to analyse loss distribution in different problem formulation modules, we use GSOM [Alahakoon et al., 2000] on target problem formulations for each problem formulation module. By doing so, we try to identify problem formulation modules that have more variety compared to other problem formulation modules. The problem formulation modules that define solution, constraints, and objective function have a higher number of GSOM clusters (Figure 6b). On the other hand, the problem formulation module model defining has few GSOM clusters but a large deviation in the number of related problem formulations per cluster (Figure 6b). In contrast, simpler problem formulation modules have few clusters and minor deviations in the number of related problem formulations per cluster. Interestingly, the above observation aligns with the loss distribution shown in Figure 6a which indicates that defining constraints, objective function, model, and solution are the most challenging problem formulation modules for the code-generating LLM.

Figure 6: The Loss Analysis: As shown in Figure 6a, there are four major contributors that determine the final status of a problem formulation. As shown in Figure 6b while other instructions are restricted to one region, these four instructions span two other regions in the diagram. This is due to the complexity of problem formulations for those instructions.





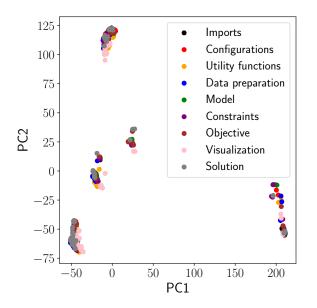
(a) The behavior of training loss based on instructions: The diagram only shows losses related to four instructions because other losses are significantly smaller.

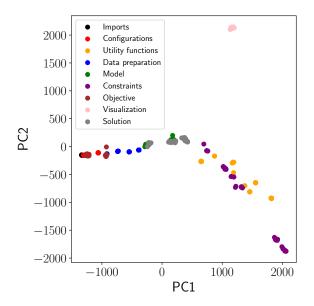
(b) Target problem formulation module wise GSOM clusters: The x-axis shows the number of clusters. The y-axis shows the standard deviation of the number of data points per cluster.

Additionally, we utilize Principal Component Analysis (PCA) to investigate how the code-generating LLM responds to different problem descriptions. Accordingly, we apply PCA on vector embeddings of the code-generating LLM as shown in Figure 7. Since the code-generating LLM is an encoder-decoder transformer model [Vaswani et al., 2017], encoder embeddings demonstrate how the code-generating LLM captures different problem descriptions and decoder embeddings demonstrate how the code-generating LLM generates problem formulations. Interestingly, we do not observe instruction-wise clusters for the encoder embeddings as shown in Figure 7a. In fact, instruction-wise problem descriptions differ from each other by a suffix attached to the original problem descriptions. Since PCA brings similar characters in problem descriptions and problem formulations together in the 2D space, we infer that PCA should provide overlapping clusters for instruction-wise problem descriptions. Furthermore, as shown in Figure

7b, we observe some clusters for the decoder embeddings. For instance, as imports and visualizations require similar problem formulation modules to be generated across all the problem descriptions, we see single clusters. But problem formulation modules for solution, constraints, and utility functions have multiple clusters or isolated points since they cover different scenarios based on problem descriptions. Furthermore, we see overlapping clusters for different problem formulation modules since they share common variables.

Figure 7: The vector embeddings of the fine-tuned code-generating LLM: The embeddings shown in Figure 7a depend on problem descriptions whereas the embeddings shown in Figure 7b depend on generated problem formulations





(a) PCA for embeddings of the encoder

(b) PCA for embeddings of the decoder

5.5 Examples

A sample of the generated problem formulations is shown in Figure 8a. Note that due to space limitations, we have only presented part of the generated code, and the complete code is available in our GitHub repository ². Figure 8b shows the output of the generated problem formulation once it is executed. While reviewing generated problem formulations and outputs, we observe that the code-generating LLM has been able to model complex scenarios mentioned in problem descriptions and generate correct problem formulations.

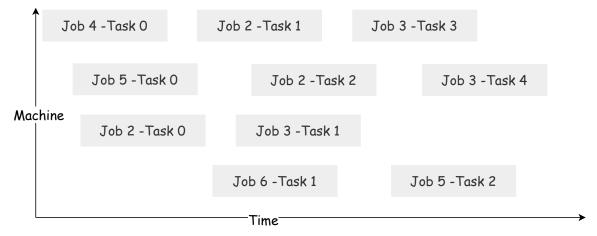
Create a job shop scheduling model with 6 jobs and 5 machines. All jobs have random routes and their operations have random durations. The objective function is makespan. Maximum duration is 20. After solving the problem, solutions will be printed and visualised. Note: The first task related to each job should be completed before the completion of any job.

²The generated problem formulations have been made publicly available on GitHub at: https://github.com/pivithuruthejanamarasinghe/AI-Copilot-Artifacts.

Figure 8: Generated problem formulations

```
# model
def model(last_time, task_to_machine_mapping):
  number_of_jobs, number_of_tasks = task_to_machine_mapping.shape # jobs are rows, tasks are
columns
  # Decision variables
  start = intvar(0, last_time, shape=(number_of_jobs, number_of_tasks), name="start") # Start
time of each task
  end = intvar(0, last_time, shape=(number_of_jobs, number_of_tasks), name="end") # End time of
each task
  # model
  m = Model()
  return m, (start, end)
# constraints
def constraints(m, start, end, durations, task_to_machine_mapping):
  number of jobs, number of tasks = task to machine mapping.shape
  m += (end == start + durations) # The end of every task is the sum of its start and duration
  # No overlap on the same machine
  for j in range(number_of_jobs):
     for t in range(number_of_tasks):
       m += [(end[j, t] <= start[j2, t2]) | (end[j2, t2] <= start[j, t])
            for j2 in range(number_of_jobs) for t2 in range(number_of_tasks)
            if task_to_machine_mapping[j, t] == task_to_machine_mapping[j2, t2] and j!= j2]
  # Precedences: do tasks in order for all jobs
  for t in range(number_of_tasks - 1):
     m += [start[:, t + 1] >= end[:, t]]
  # The first task related to each job should be completed before the completion of any job
  for i in range(number_of_jobs):
     m += (end[:, 0] <= end[i, 4])
  return m
# Objective
def objective(m, end):
  m.minimize(max(end))
  return m
```

(a) Part of a generated problem formulation. Variable last_time means the latest completion time by summing up all processing times of all jobs. Variable task_to_machine_mapping contains for all the jobs, which operation should run on which machine.



6 Conclusion

We introduce an AI-Copilot based on a code-generating LLM based problem formulation framework for business optimisation using a case study in production scheduling. The proposed AI-Copilot can generate large and complex problem formulations and requires only a small training dataset. Additionally, new performance evaluation metrics are proposed to evaluate the quality of generated problem formulations. The prompt engineering-based problem formulation modularization technique introduced in our AI-Copilot can overcome token limitation issues in code-generating LLMs for problem formulation. Although the case study is based on artificial production data and hypothetical problem descriptions, the framework introduced in our AI-Copilot is general and can be used in practical scenarios where data is available. Although the proposed AI-Copilot in this research is limited to production scheduling problem formulation, the general framework can be adapted to other types of business optimisation problems.

As further improvements, we will focus on developing the remaining components of the framework while supporting multiple problem formulation types such as routing, assignment, etc. Generating problem formulations for specific optimisation technologies different from constraint programming is also an interesting topic to explore. In the next step, we will focus on mixed-integer programming, column generation, and lazy constraints problem formulations in combinatorial optimisation that cover a broad range of business optimisation case studies. Given the importance of business decisions produced by optimisation, it is important to incorporate multiple explanation techniques and responsible AI principles to help users validate the correctness of the generated problem formulation and prevent negative consequences from the proposed AI-Copilot. use cases.

References

Damminda Alahakoon, Saman K Halgamuge, and Bala Srinivasan. Dynamic self-organizing maps with controlled growth for knowledge discovery. *IEEE Transactions on neural networks*, 11(3):601–614, 2000.

Andreas Antoniou and Wu Sheng Lu. Practical Optimization. Springer, 2007.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles & Sutton. Program synthesis with large language models. arXiv preprint arXiv:2108.07732, 2021.

Pasquale Avella, Alice Calamita, and Laura Palagi. Off-the-shelf solvers for mixed-integer conic programming: insights from a computational study on congested capacitated facility location instances. *arXiv* preprint arXiv:2303.04216, 2023.

Radu Baltean-Lugojan, Pierre Bonami, Ruth Misener, and Andrea Tramontani. Selecting cutting planes for quadratic semidefinite outer-approximation via trained neural networks. *URL: http://www. optimization-online. org/DB HTML/2018/11/6943. html*, 2018.

J Christopher Beck, T K Feng, and Jean Paul Watson. Combining constraint programming and local search for job-shop scheduling. *INFORMS Journal on Computing*, 23(1):1–14, 2011.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

Ksenia Bestuzheva, Mathieu Besançon, Wei Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob & Witzig. The SCIP optimization suite 8.0. arXiv preprint arXiv:2112.08872, 2021.

Stephen P Boyd and Lieven Vandenberghe. Convex optimization. Cambridge university press, 2004.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario & Amodei. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech & Zaremba. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

Liying Cheng, Xingxuan Li, and Lidong Bing. Is GPT-4 a Good Data Analyst? *arXiv preprint arXiv:2305.15038*, 2023.

Edward G Coffman Jr, Michael R Garey, and David S Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

- William J Cook, David L Applegate, Robert E Bixby, and Vasek Chvatal. *The traveling salesman problem: a computational study*. Princeton university press, 2011.
- IBM ILOG Cplex. V12. 1: User's Manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.
- Robert Fourer, David M Gay, and Brian W Kernighan. AMPL: A mathematical programming language. *Management Science*, 36(5):519–554, 1990.
- Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32, 2019.
- Google. CP-SAT Solver, 2023. URL https://developers.google.com/optimization/cp/cp_solver.
- Tias Guns. Increasing modeling language convenience with a universal n-dimensional array, cppy as python-embedded example. In *Proceedings of the 18th workshop on Constraint Modelling and Reformulation, Held with CP*, volume 19, 2019.
- Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob & Steinhardt. Measuring coding challenge competence with apps. *arXiv* preprint arXiv:2105.09938, 2021.
- André Hottung, Shunji Tanaka, and Kevin Tierney. Deep learning assisted heuristic tree search for the container pre-marshalling problem. *Computers & Operations Research*, 113:104781, 2020.
- Hugging Face. Trainer, 2023. URL https://huggingface.co/docs/transformers/v4.33.2/en/main_classes/trainer#transformers.Trainer.
- Mohammad Hossein Jarrahi, Christoph Lutz, Karen Boyd, Carsten Oesterlund, and Matthew Willis. Artificial intelligence in the work context. *Journal of the Association for Information Science and Technology*, 74(3):303–310, 2023.
- Josef Kallrath and John M Wilson. Business optimisation using mathematical programming. Springer, 1997.
- Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.
- Grzegorz Kłosowski, Edward Kozłowski, and Arkadiusz Gola. Integer linear programming in optimization of waste after cutting in the furniture manufacturing. In *Intelligent Systems in Production Engineering and Maintenance–ISPEM 2017: Proceedings of the First International Conference on Intelligent Systems in Production Engineering and Maintenance ISPEM 2017 1*, pages 260–270. Springer, 2018.
- Stephan Kreipl. A large step random walk for minimizing total weighted tardiness in a job shop. *Journal of Scheduling*, 3(3):125–138, 2000.
- Wen Yang Ku and J Christopher Beck. Mixed integer programming models for job shop scheduling: A computational analysis. *Computers & Operations Research*, 73:165–173, 2016.
- Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.
- Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo

Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm & de Vries. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Brady D Lund, Ting Wang, Nishith Reddy Mannuru, Bing Nie, Somipam Shimray, and Ziang Wang. ChatGPT and a new academic reality: Artificial intelligence-written research papers and the ethics of the large language models in scholarly publishing. *Journal of the Association for Information Science and Technology*, 74(5):570–581, 2023.

Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 336–347. IEEE, 2021.

Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

Nhan Nguyen and Sarah Nadi. An empirical evaluation of Github copilot's code suggestions. In *Proceedings of the* 19th International Conference on Mining Software Repositories, pages 1–5, 2022.

Su Nguyen, Dhananjay Thiruvady, Mengjie Zhang, and Kay Chen Tan. A genetic programming approach for evolving variable selectors in constraint programming. *IEEE Transactions on Evolutionary Computation*, 25(3):492–507, 2021.

OpenAI. GPT-4 Technical Report, 2023.

Michael Pinedo. Planning and scheduling in manufacturing and services. Springer, 2005.

Yves Pochet and Laurence A Wolsey. Production planning by mixed integer programming, volume 149. Springer, 2006.

Rindranirina Ramamonjison, Haley Li, Timothy T Yu, Shiqi He, Vishnu Rengan, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong & Zhang. Augmenting operations research with auto-formulation of optimization models from problem descriptions. *arXiv* preprint *arXiv*:2209.15565, 2022.

Rindranirina Ramamonjison, Timothy T Yu, Raymond Li, Haley Li, Giuseppe Carenini, Bissan Ghaddar, Shiqi He, Mahdi Mostajabdaveh, Amin Banitalebi-Dehkordi, Zirui Zhou, and Yong & Zhang. NL4Opt Competition: Formulating Optimization Problems Based on Their Natural Language Descriptions. *arXiv preprint arXiv:2303.08233*, 2023.

Emily Reif, Daphne Ippolito, Ann Yuan, Andy Coenen, Chris Callison-Burch, and Jason Wei. A recipe for arbitrary text style transfer with large language models. *arXiv preprint arXiv:2109.03910*, 2021.

Pablo Rivas and Liang Zhao. Marketing with chatgpt: Navigating the ethical terrain of gpt-based chatbot technology. *AI*, 4(2):375–384, 2023.

Jagdeep Singh and Harwinder Singh. Kaizen philosophy: a review of literature. *IUP journal of operations management*, 8(2):51, 2009.

Irene Solaiman and Christy Dennison. Process for adapting language models to society (palms) with values-targeted datasets. *Advances in Neural Information Processing Systems*, 34:5861–5873, 2021.

Alireza Soroudi. Power system optimization modeling in GAMS, volume 78. Springer, 2017.

Paolo Toth and Daniele Vigo. The vehicle routing problem. SIAM, 2002.

Dimos Tsouros, Hélène Verhaeghe, Serdar Kadıoğlu, and Tias Guns. Holy Grail 2.0: From Natural Language to Constraint Models. *arXiv preprint arXiv:2308.01589*, 2023.

- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia & Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- Xuequn Wang, Xiaolin Lin, and Bin Shao. Artificial intelligence changes the way we work: A close look at innovating with chatbots. *Journal of the Association for Information Science and Technology*, 74(3):339–353, 2023.
- Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv* preprint arXiv:2109.00859, 2021.
- Zhiguo Wang, Patrick Ng, Xiaofei Ma, Ramesh Nallapati, and Bing Xiang. Multi-passage bert: A globally normalized bert model for open-domain question answering. *arXiv preprint arXiv:1908.08167*, 2019.
- Jean Paul Watson and J Christopher Beck. A hybrid constraint programming/local search approach to the job-shop scheduling problem. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 263–277. Springer, 2008.
- Hegen Xiong, Shuangyuan Shi, Danni Ren, and Jinjin Hu. A survey of job shop scheduling problem: The types and models. *Computers & Operations Research*, 142:105731, 2022.
- Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.
- Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on Github Copilot, Amazon CodeWhisperer, and ChatGPT. *arXiv preprint arXiv:2304.10778*, 2023.
- Gang Yu. Industrial applications of combinatorial optimization, volume 16. Springer Science & Business Media, 2013.
- Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian Guang & Lou. When Neural Model Meets NL2Code: A Survey. *arXiv preprint arXiv:2212.09420*, 2022.
- Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B Tenenbaum, and Chuang Gan. Planning with large language models for code generation. *arXiv* preprint arXiv:2303.05510, 2023.