

Automated Conversion of Static to Dynamic Scheduler via Natural Language

Paul Mingzheng TANG^a, Kenji Kah Hoe LEONG^a, Nowshad SHAIK^a and Hoong Chuin LAU^{a,*}

^aSchool of Computing and Information Systems, Singapore Management University

Abstract. In this paper, we explore the potential application of Large Language Models (LLMs) that will automatically model constraints and generate code for dynamic scheduling problems given an existing static model. Static scheduling problems are modelled and coded by optimization experts. These models may be easily obsoleted as the underlying constraints may need to be fine-tuned in order to reflect changes in the scheduling rules. Furthermore, it may be necessary to turn a static model into a dynamic one in order to cope with disturbances in the environment. In this paper, we propose a Retrieval-Augmented Generation (RAG) based LLM model to automate the process of implementing constraints for Dynamic Scheduling (RAGDYS), without seeking help from an optimization modeling expert. Our framework aims to minimize technical complexities related to mathematical modelling and computational workload for end-users, thereby allowing end-users to quickly obtain a new schedule close to the original schedule with changes reflected by natural language constraint descriptions.

1 Introduction

Companies today are facing increasing pressure to apply optimization models to solve complex resource planning and scheduling problems. Traditionally, these models are developed by human expert modelers and once they have been handed over and deployed in a production system, the models will remain unchanged until the next release of the system.

In an increasing dynamic and uncertain world, the models developed may need to constantly evolve in order to remain relevant. Maintaining a team of human experts to keep the underlying optimization models updated is usually costly; many end-users organizations lament that the models expire quickly, and often they lack the expertise to keep the models updated in a production setting.

Our research is motivated by the recent advances in large language models (LLM). Our goal is to develop a Generative AI methodology that enables the deployed resource optimization models to be updated (finetuned) by end-users themselves to cope with dynamics and uncertainty. This is a significant departure from the traditional modus operandi of hefty investments in either maintaining optimization modeling experts in-house or outsourcing the software maintenance to a third-party provider. In a wider context, this work is by the ideal to have human-AI collaboration via Generative AI in the context of resource planning, scheduling and optimization. While LLMs may not be directly relied upon for generating complex planning and

scheduling models from ground zero [9], we investigate in this paper the idea of prompt engineering for revising an existing planning model, which we believe to be a less daunting task.

In this paper, we propose a LLM-based methodology that enables an existing static optimization model to be revised automatically in order to cope with dynamics and uncertainty. More precisely, we consider a well-known manpower scheduling/rostering problem, and explore how an LLM can be exploited to provide the necessary changes to the underlying constraints to cope with a dynamic variant of the underlying static scheduling problem. We assume that a static model exists and solutions for solving the model can be obtained via an efficient mathematical programming solver (such as Gurobi, Google OR-Tools etc). The interesting question we like to address is, without relying on a human modeling expert, to what extent can a planner interact with a pre-trained LLM to effectively generate the revised optimization model based on the static model?

1.1 Motivating Example

To illustrate how a static nurse scheduling problem is extended into a dynamic scheduling problem, we present the following problem setting as a motivating example.

The static problem is to generate a schedule for nurses over a planning horizon over set of shift patterns. The following constraints for the schedule are:

- Every day, each shift is assigned to a single nurse.
- No nurse works more than one shift each day.
- Each nurse is assigned to at least M number of shifts overall.

Each nurse has specified which shifts per day are preferable. The objective is to maximize the total number of preferences fulfilled.

We define the parameters below

- N : Number of nurses
- D : Number of days
- S : Number of shifts
- M : Minimum number of shifts per nurse
- $P_{n,d,s}$: A boolean matrix over (N, D, S) , indicating if nurse n prefers to be assigned day d and shift s .

We define the decision variables as:

$X_{n,d,s}$: An array of binary decision variables over (N, D, S) , indicating if nurse n is assigned to day d and shift s .

We formulate the constraints below:

- $\sum_n^N X_{n,d,s} = 1, \forall (d, s) \in D, S$

* Corresponding Author

- $\sum_s^S X_{n,d,s} \leq 1, \forall (n, d) \in N, D$
- $\sum_d^D \sum_s^S X_{n,d,s} \geq M, \forall n \in N$

The objective is to maximise:

$$\sum_n^N \sum_d^D \sum_s^S X_{n,d,s} \times P_{n,d,s} \quad (1)$$

Now suppose that the model has been formulated and implemented in code written by an optimization expert. The end-user planner runs the code and obtains an initial schedule to be used, X . Now suppose on the day before the schedule was to be used, a nurse A has to take a medical leave from day D_1 to D_2 . We term this as a *disturbance*. The planner wants to ensure that the repaired schedule is within T number of changes to X . We term T as the perturbation threshold. In order for the planner to repair the schedule due to this and other disturbances, ideally the model need to be adjusted to include the following constraints:

$$X_{A,d,s} = 0, \forall s \in S, \forall d \in [D_1, D_2] \quad (2)$$

$$\sum_n^N \sum_d^D \sum_s^S |X_{n,d,s} - X'_{n,d,s}| \leq T \quad (3)$$

Our proposed method will allow the planner to generate a revised model and the corresponding code to reflect these constraints, via a natural language description of the disturbance and perturbation. An example prompt chain for the planner could be: "Add a constraint such that nurse A is not available from day D1 to D2.". After generating the new code, the additional prompt is "Add a constraint such that the schedule generated does not change too much from the original schedule. The number of changes to the schedule should not exceed T."

1.2 Contributions

Our contributions are as follows:

- We introduce the problem arising from developing such an automated tool for commercial manpower scheduling application in service organizations (such as restaurant chains and hospitals);
- We propose a framework that enables the underlying constraint model and the resulting code to be revised over time. This involves designing a dataset comprising natural language descriptions, mathematical formulations, and code samples both before and after incorporating dynamic constraints.
- We utilize Retrieval-Augmented Generation (RAG) to generate the revised constraint formulation to solve the Minimum Perturbation Problem, whose aim is make use of an existing schedule obtained previously to generate a revised schedule automatically, such that the perturbation from the existing schedule is no more than the perturbation threshold T .

2 Background and Related Work

Before going through the details of our proposed method, we would like to introduce some main concepts and related work to contextualize our work: Constraint Programming and Optimization, Minimum Perturbation in Dynamic Scheduling, LLMs for Constraint Generation, and Retrieval-Augmented Generation.

2.1 Constraint Programming and Optimization

Constraint programming is now a relatively mature technology for solving a wide range of difficult combinatorial search problems [8], and in our context used to formulate our static and dynamic scheduling problems. Unlike Mixed-Integer Programming and Linear Programming models, Constraint programming models include functions to model more complex relations between variables, such as the `AllDifferent` function. Due to addition of these functions, which are closer to natural language as opposed to linear constraints, we opted the use constraint programming formulations and code for our LLM planning and coding agents to learn from.

2.2 Minimum Perturbation in Dynamic Scheduling

The idea of modifying an existing scheduling problem (formulated as a Constraint Satisfaction Problem (CSP)) and the initial schedule to reflect new constraints introduced by disturbances, and maintaining a "minimally changed" new schedule, is termed a *Minimum Perturbation Problem* (MPP)[3]. The work formally defines a minimum perturbation problem as a 5-tuple $(\Theta_i, \alpha_i, C_{del}, C_{add}, \delta)$.

- Θ_i is a CSP
- α_i is a solution to Θ_i
- C_{del}, C_{add} are constraint removal and addition sets
- δ is a distance function between 2 complete solutions.

The original MPP defines a CSP, rather than a constraint optimization problem with an objective function. The objective function for the MPP is then defined as a minimization of the distance function δ , which is described as $d_x = |x - c|$ where x is the new schedule variables and c is the initial schedule assignments. For our problem, we formulate the MPP constraint as a hard constraint, where $d_x \leq T$ and T is a threshold value for the maximum Hamming distance between the new and initial schedules. This minimum perturbation constraint serves 2 purposes: 1) It reduces the amount of unnecessary changes to the current schedule, which end-users are likely to want to be as stable as possible. 2) It reduces the search space considerably as we set a threshold for the maximum amount of change from the pre-computed schedule to the new schedule.

2.3 LLMs for Constraint Generation

There has been recent progress on using LLMs for combinatorial optimization, specifically on how to generate constraints for these problems based on natural language. For our work, we applied some elements from the OptiMUS framework presented in [1] such as the planning and coding agents, which breaks down the challenging problem of formulating constraints from natural language into smaller stages. Our work, however, differs in 1) the problem representation and 2) the context-specific constraint generation produced by the coding agent. To cater the the non-technical end-users of our framework, we do not expect them to formulate their problems in a structured form, and instead simply expects simple natural language to describe the new constraint to be added. To provide better context for the planning and coding agents, our scenario of dynamic scheduling already assumes an initial problem description, formulation, and code. By using these components, in addition to following including RAG examples, the planning agent is able to make use of existing parameters, variables, and constraints to generate a plan for new parameters, variables and constraints. The coding agent then makes use of the original code and the planning agent output to produce a modified code which implements the new constraint.

2.4 Retrieval-Augmented Generation

With minimizing technical expertise and computational resource requirements for end-users as our goal, Retrieval-Augmented Generation (RAG) techniques can be applied. RAG is commonly used to introduce context-specific knowledge which the pre-trained model may not have been trained on, as shown in [6], without the need for further fine-tuning. In the case of dynamic scheduling, worker scheduling problems can vary significantly and have context specific constraints which pre-trained LLMs are unlikely to be trained on. While there are various RAG techniques, as surveyed in [4], we opted to use Naive RAG for our method which has minimal fine-tuning steps.

2.5 Challenges in Dynamic Constraint Generation

For our problem of dynamic constraint generation, it would be preferable to have a set of benchmark worker scheduling problems to use, accompanied with natural language descriptions for each problem. There are a few challenges.

- 1) There is a lack of worker scheduling benchmark datasets available publicly. Most papers define the constraints specific to their problem and leave out the values of parameters required to construct a full problem instance to run.
- 2) There is a lack of natural language descriptions for worker scheduling problem instances. When scraping natural language description from optimization papers describing a worker scheduling problem, most papers use technical terms or mathematical formulation to express what they mean, which would differ significantly for an end-user's layperson description of a dynamic constraint to be added.

3 Proposed Method

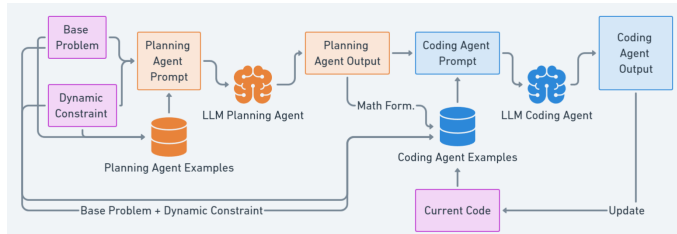


Figure 1. Overview of our approach.

We introduce our proposed framework, RAGDYS, which takes as input (see pink components in Figure 1) a base problem description, a dynamic constraint description, and the current code for the base problem. In our context, the initial base problem description and current code are provided by an optimization expert, while the dynamic constraint description is provided by a non-technical end-user. The final output to be used would be the coding agent output, which is a modified version of the current code that includes code for new parameters, variables and constraints to reflect the dynamic constraint inserted. This new code can be executed to obtain a new schedule for the end-user. Further details of our approach and implementation are presented in the following section 3.1.

3.1 Architecture and Approach Details

RAGDYS contains 3 stages. In the planning and coding stages, we use RAG methods to inform the LLM agents with the most similar example input output pair. The vector database used for our ex-

periments was implemented using ChromaDB, which by default creates document embedding vectors using all-MiniLM-L6-v2 [10], and performs cosine similarity search to find the most similar example prompts from ChromaDB database.

3.1.1 Step 1: Planning Stage

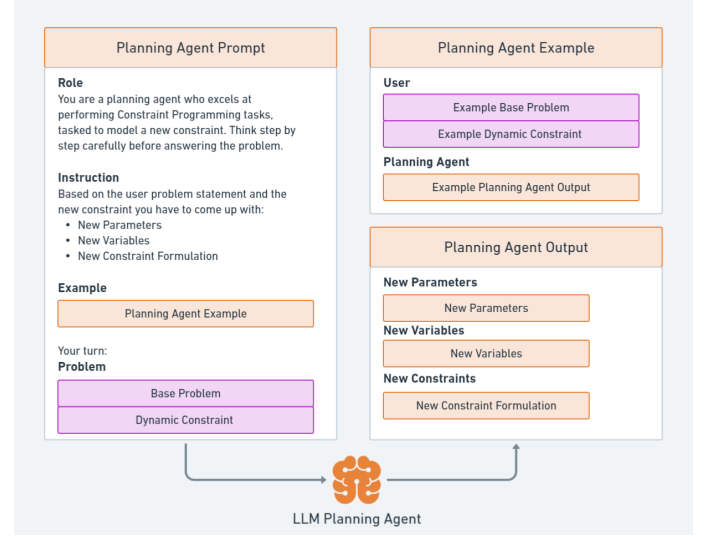


Figure 2. Planning Stage Details.

The planning stage takes as input the base problem description and dynamic constraint description. Following RAG methods, the most similar example is retrieved from a vector database of planning agent examples, which contain example user input and planning agent output pairs. These examples, along with the base problem and dynamic constraint descriptions, are inserted into the planning agent prompt template (See Appendix C for example planning agent prompt). The planning agent prompt is sent to an LLM agent (our implementation uses Gemini [5]), and the output of the LLM agent should contain 3 sections: New parameters, New variables and New constraints. See Figure 2 for more details. This output is then used in the coding stage.

3.1.2 Step 2: Coding Stage

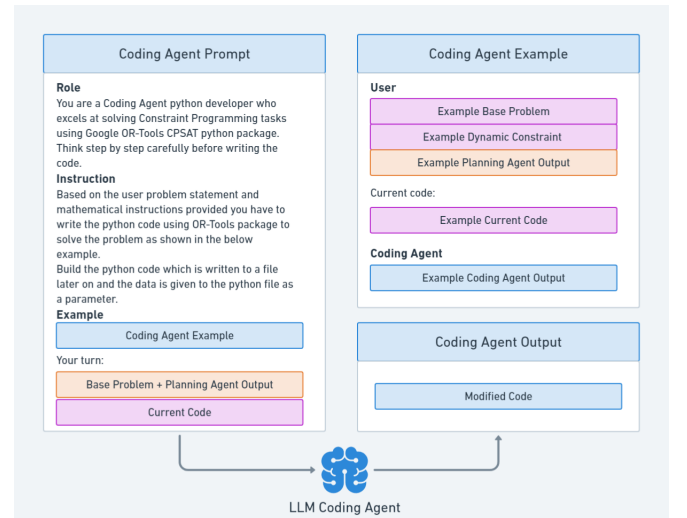


Figure 3. Coding Stage Details.

The coding stage takes as input the base problem and dynamic constraint descriptions, the planning agent output, and the current code to be modified. Similar to the planning stage, we obtain most similar example user input and coding agent output pairs. All these components are inserted into the coding agent prompt template (See Appendix D for example coding agent prompt) to produce the coding agent prompt. The LLM coding agent (implemented with Claude Haiku [2] and OpenAI GPT-4 [7]), will take this prompt as input and produce the modified version of the code as output. See Figure 3 for more details.

3.1.3 Step 3: Execution Stage

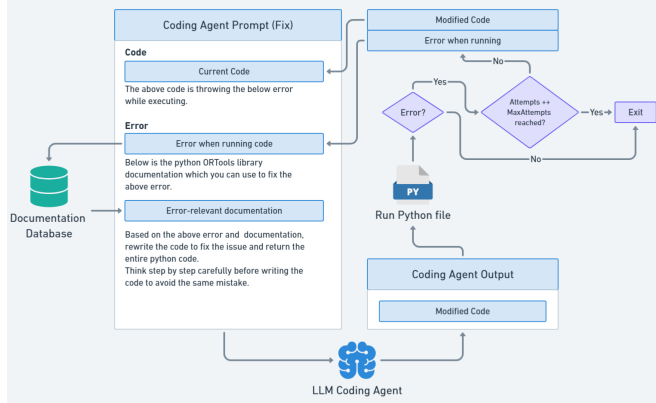


Figure 4. Code-fixing stage Details.

The execution stage is where the user is able to decide how to progress from the generated code output from the coding stage. The user may choose to either execute the modified code to produce a new schedule, or repeat the process to add another dynamic constraint. Our framework proposes to not only add the dynamic constraint associated with a disturbance, but also leveraging the existing schedule (if it exists) and minimal adjustments to obtain a new schedule efficiently through a repeated process of adding the minimum perturbation constraint using natural language.

When executing the new code, the code may produce errors and our framework will enter the code-fixing stage. In the code-fixing stage, the coding engine is called repeatedly to fix the code. The number of attempts cannot exceed the `maxAttempts` threshold. Using RAG, we retrieve the most relevant document stored in a database containing documentation on Google OR-Tools CP-SAT objects and methods. Combining the error message, relevant documentation, and current code, we prompt the coding engine to amend the current code to fix the error. See Figure 4 for overall pipeline on the code-fixing stage.

3.2 Dataset Generation

We manually prepare a data set to be inserted into the vector database and used as references for RAG when solving unseen problems. We also manually write a test set for the unseen problem instances. We multiply the total number of instances used in our database and testing set by prompting a LLM to generate paraphrased versions of the natural language descriptions, as shown in Figure 5. Due to possible mistakes generated when the LLM paraphrases the description (removes or changes the name of parameters to be used in the constraint), we opt to manually fix these mistakes.

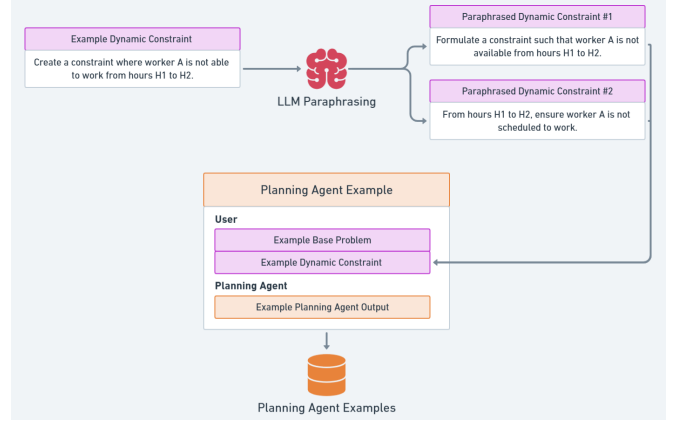


Figure 5. LLM paraphrasing for dataset generation

3.2.1 Database Set

We manually write natural language prompts with a base worker scheduling problem (In our case, we only use the Gig-Scheduling Problem (GSP)), planning agent outputs, and before and after constraint insertion codes written in Python using the Google OR-Tools CP-SAT solver library.

GSP is a variant of the worker scheduling problem, where workers are not assigned a shift out of a fixed set of shifts per day, but are assigned directly to hours subject to constraints (such as specifying the minimum and maximum length of consecutive hours assigned). These worker-hour assignments must then match, as much as possible, the task demands. Tasks are defined as a job to be assigned to a worker for a single hour, and is constrained by skill requirements (only a subset of workers with the qualified skills can be assigned to the task), and no multi-tasking (a worker can only be assigned at most 1 task per hour). See Appendix A for the mathematical programming formulations for the base GSP.

3.2.2 Test Set

Similar to the Database set, we manually write natural language prompts, planning agent outputs, and code for the GSP. In addition, to test if our proposed method is able to generalize to different base problems, we also manually construct data for the standard Nurse Scheduling Problem (NSP). See Appendix B for the mathematical programming formulations for the base NSP. We use the same LLM paraphrasing method to increase the initial testing set of size 14 to size 70 (35 GSP instances, 35 NSP instances).

4 Evaluation and Results

In the following, we report experimental results. Besides performing automatic evaluation, we also conduct manual inspection to evaluate the correctness of the constraints and code. As described in 3.2, we populated the dataset using GSP problem instances, and test the performance of the planning and coding agents using 2 sets: 1) Unseen dynamic constraints for the GSP, to verify if the LLMs are able to generalise dynamic constraints with similar base problem); and 2) Unseen dynamic constraints for the NSP, to verify if the LLMs are able to generalise to different base worker scheduling problem and apply dynamic constraints to the new context.

4.1 Automatic Evaluation

Similar to OptiMUS, we can evaluate the performance of our model by executing the generated code from the coding agent and comparing the results with the expected results. There are a few possible outcomes when running the code:

- KeyError for data
- Syntax error in expressing constraints
- Infeasible solution
- Feasible solution but objective value does not match
- Solution matches expectation

We classify the outcome as a solution matching the target when one of the following are met:

- If both target and generated code reach an optimal solution, then the 2 solutions are exactly equal.
- If both target and generated code are at least feasible, then check if the computed upper and lower bounds are similar (measure the overlap between the 2 intervals). If the overlap covers half the total coverage, it is considered a match.

We should note that the results of the automatic evaluation method are not completely reliable and produce false positives (Detect a Match in expected solution, but the constraint formulated differs from what the prompt intends) and false negatives (Detect a mismatch in solution despite correct constraint formulation. This is due to the non-deterministic nature of running Google OR-Tools with multi-threading). Hence, we also conduct manual evaluation to investigate these possible outcomes, as well as look provide more in-depth explanations for the errors.

4.2 Manual Evaluation

During manual inspection, we delve deeper into the specifics behind the results displayed in the automatic evaluation. This includes identifying the reasons behind the outcomes and double-checking solutions for both false positives and false negatives.

We can identify the following:

- False Positives
- False Negatives
- Hallucinating Parameters
- Incorrect Reasoning

In the following section, we detail our experimental results.

4.3 Results and Discussion

	Outcome					Total
	KeyError	Syntax Error	Infeasible	Feasible (Not Match)	Match	
GSP	0	0	0	6	29	35
NSP	0	1	0	0	34	35
Total	0	1	0	6	63	70

Table 1. Evaluation results on test sets using OpenAI GPT4

Table 1 shows the overall results for automatic evaluation with OpenAI GPT-4 as the coding agent. In general, automatic evaluation has shown that for both seen and unseen constraints for the same and different base problems, the model attains a 90% match rate. As mentioned in the earlier section, we would need to further inspect the output code for the cases of false positives and false negatives. In the next section, we measure the false positive and negatives, as well as analyze the code outputs to obtain explanations for the errors.

	False Positives	False Negative	Parameter Hallucination	Incorrect Reasoning
GSP	6	2	0	9
NSP	4	0	3	1

Table 2. Manual Evaluation results on test sets OpenAI GPT4

4.3.1 Example False Positive

Prompt Number	Main Prompt	Paraphrased Prompt	Code	Target Code
3.1	The total number of hours worked by worker A must be at least the total number of hours worked by worker B.	Worker A must work more or equal hours than worker B.	for h in range(h): model.Add(workerHours[A, h] >= workerHours[B, h])	
3.3	Worker A should be at least the total number of hours worked by worker B.	Worker A should work at most the same hours as worker B.	for h in range(h): model.Add(workerHours[A, h] <= workerHours[B, h])	model.Add(sum(workerHours[A, h] for h in range(h)) >= sum(workerHours[B, h] for h in range(h)))
3.4	Worker A should work the same number of hours as worker B, or more.	Worker A should work the same number of hours as worker B, or more.	for h in range(h): model.Add(workerHours[A, h] >= workerHours[B, h])	

Figure 6. Table of results for false positive cases for GSP prompt 3 and the paraphrased versions.

As shown in Figure 6, there are instances where the automatic evaluation flags the result as a match, but the code generated does not formulate the correct constraint. Looking at the code and target code, we can see that the coding agent has misinterpreted the paraphrased prompt as "Worker A must have at least the same schedule as Worker B, with additional hours added on" rather than "Total number of hours worked by worker A >= Total number of hours worked by worker B". It is possible that this error is caused by poor phrasing of the paraphrased prompt.

4.3.2 Example False Negative

Prompt Number	Main Prompt	Paraphrased Prompt	Code	Target Code
7.1	The difference in the number of hours worked by any 2 workers should not exceed workerHourDiffThreshold.	Workers should not have a difference in their hours worked that exceeds the workerHourDiffThreshold.	for w in range(w): model.Add(hours_worked[w] == sum(workerHours[w, h] for h in range(h))) for w in range(w): for w2 in range(w2): if w != w2: diff = model.NewIntVar(workerHourDiffThreshold, workerHourDiffThreshold, "diff_{}_{}_gap".format(w, w2)) model.Add(diff == hours_worked[w] - hours_worked[w2]) model.Add(diff <= workerHourDiffThreshold) model.Add(diff >= -workerHourDiffThreshold)	for w in range(w): for w2 in range(w2, w): model.AddAbsEql(changedVar[w, w2], sum(workerHours[w, h] for h in range(h)) - sum(workerHours[w2, h] for h in range(h))) model.Add(changedVar[w, w2] <= workerHourDiffThreshold)
7.2	The number of hours worked by any two workers should not differ by more than workerHourDiffThreshold.	The number of hours worked by any two workers should not differ by more than workerHourDiffThreshold.	for w in range(w): for w2 in range(w2): if w != w2: model.Add(workerTotalHours[w] - workerTotalHours[w2] <= workerHourDiffThreshold) model.Add(workerTotalHours[w2] - workerTotalHours[w] <= workerHourDiffThreshold)	

Figure 7. Table of results for false negative cases for GSP prompt 7 and the paraphrased versions.

In Figure 7, we observe that the generated code is logical and is consistent with the target code, despite sometimes creating unnecessary variables or constraints. However, the automatic evaluation flagged the result as a mismatch, because the objective value obtained after running the code differs from the target code's objective value. This is a result of how ORTools handles different constraint formulations and how one formulation could be computationally more efficient, as well as the non-deterministic nature of ORTools when running on multiple threads, meaning even with the exact same code running there is a possibility that the output will differ if we do not allow the solver to run to optimality. Due to the nature of the GSP instance, we opted to run the code with multiple threads and a fixed time limit so as to obtain results in reasonable time.

4.3.3 Example Parameter Hallucination

As shown in Figure 8, we can see an example of parameter hallucination when observing prompts like prompt 11. In this NSP prompt, we opted to use numbers directly instead of variable names for the days, such as D1 and D2. Because most code is written using the variable formulation, there is a possibility that the LLM coding agent hallucinates parameters for day 1 to day 2, replacing it with D1 and D2 as parameters expected to be provided by the user.

Prompt Number	Main Prompt	Paraphrased Prompt	Code	Target Code
11	Nurse K is unable to work on an afternoon (PM) slot from day 1 to day 2.	Nurse K is unable to work on an afternoon (PM) slot from day 1 to day 2.	for d in range(D1, D2 + 1): model.Add(nurseDaySlot[K, d, PM] == 0)	for d in range(1, 3): model.Add(nurseDaySlot[K, d, PM] == 0)
11.2	Nurse K cannot work during the afternoon shift (PM) from day 1 to day 2.	Nurse K cannot work during the afternoon shift (PM) from day 1 to day 2.	for d in range(D1, D2): model.Add(nurseDaySlot[K, d, PM] == 0)	
11.3	Nurse K cannot work during the afternoon shift (PM) from day 1 through day 2.	Nurse K cannot work during the afternoon shift (PM) from day 1 through day 2.	for d in range(D1, D2 + 1): model.Add(nurseDaySlot[K, d, PM] == 0)	

Figure 8. Table of results for false positive cases for NSP prompt 11 and the paraphrased versions, which exemplify parameter hallucination

4.3.4 Example Incorrect Reasoning

Prompt Number	Main Prompt	Paraphrased Prompt	Code	Target Code
5.1	Worker A must work X hours than initially scheduled. The original schedule, denoted as origSchedule[w,h], outlines the assignments for each worker during each hour.	Worker A must work X hours less compared to his original schedule. The original schedule is stored as origSchedule[w,h], containing all the worker hour assignment for each worker for each hour in the original schedule.	total_hours_A = sum(workerHours[A, h] for h in range(H)) model.Add(total_hours_A - sum(reducedSchedule[A, h] for h in range(H)) == X) model.Add(reducedSchedule[A, h] <= workerHours[A, h])	model.Add(sum(workerHours[A, h] for h in range(H)) - sum(origSchedule[A, h] for h in range(H)) - X)
5.3	Compared to the initial schedule, worker A must work at least X hours less. The original schedule, denoted as origSchedule[w,h], contains the work assignments for each worker during each hour.	Design a new work schedule for the nurses while keeping changes from the original schedule. The original schedule is stored as [prevSchedule]. The constraint you are formulating aims to ensure that the total number of changes made to the nurseDayShift assignments does not exceed the [changeThreshold].	model.Add(sum(workerHours[A, h] - origSchedule[A, h] for h in range(H)) >= X)	
13.3	Create a new working schedule for the nurses but ensure that the new schedule should not change much from the original schedule. The original schedule is stored as [prevSchedule]. The constraint you are formulating aims to ensure that the total number of changes made to the nurseDayShift assignments does not exceed the [changeThreshold].		No code inserted	for h in range(H): for d in range(D): for s in range(S): model.AddAbsEquality(changedVar[n,d,s], nurseDayShift[n,d,s] - prevSchedule[n,d,s]) model.Add(sum(changedVar[n,d,s] for n in range(N) for d in range(D) for s in range(S)) <= changeThreshold)

Figure 9. Table of results for false positive cases where we identify incorrect reasoning causing incorrect / empty constraints being generated

In Figure 9, we can see a few examples of incorrect reasoning. For prompt 13.3, the LLM coding agent did not generate any additional code on top of the base code, implying that it was simply unable to reason what the prompt wanted (Minimum perturbation constraint). For prompts 5.1, we can see a combination of hallucinated parameters (reducedHours) and incorrect reasoning (not using origSchedule parameter at all, as well as adding illogical constraints). Lastly, in prompt 5.3 we can see an almost correct constraint, however the change in number of hours from origSchedule should be computed with the expression "origSchedule[A,h] - workerHours[A,h]" rather than "workerHours[A,h] - origSchedule[A,h]".

4.4 Ablation Study

We conduct an ablation study on how a different LLM (Claude Haiku) performs against OpenAI GPT-4 when being used as the coding agent. The results of the experiments are shown in Table 3.

4.4.1 Ablation 1: Varying the LLM

	Outcome				
	KeyError	Syntax Error	Infeasible	Feasible (Not Match)	Match
GSP	5	3	1	5	21
NSP	7	1	5	0	22
Total	12	4	6	5	43

Table 3. Evaluation results on test sets using Claude Haiku

We can observe the difference in performance using GPT-4 and Claude Haiku, looking at Tables 1 and 3. As expected of the more expensive model, OpenAI GPT-4 outperforms Claude Haiku when used in the coding agent, with an automatic evaluation match rate of 90% while Claude Haiku has an automatic match rate of 60%.

Looking at the outputs of Claude Haiku in Figure 10, we can see its limitations as opposed to GPT-4. Very often, Claude Haiku

Prompt Number	Prompt	GPT-4 Code	Haiku Code	Target Code
2	The total number of hours worked by worker A and worker B must be the same.	model.Add(sum(workerHours[A, h] for h in range(H)) == sum(workerHours[B, h] for h in range(H)))	model.AddAbsEquality(sum(workerHours[A * H + h] for h in range(H)), sum(workerHours[B * H + h] for h in range(H)))	model.Add(sum(workerHours[A, h] for h in range(H)) == sum(workerHours[B, h] for h in range(H)))
6	The difference in the number of hours worked by worker A and worker B should not exceed workerHourDiffThreshold.	hour_diff = model.NewIntVar(-workerHourDiffThreshold, workerHourDiffThreshold, "hour_diff") model.Add(hour_diff == sum(workerHours[A, h] for h in range(H)) - sum(workerHours[B, h] for h in range(H))	No code added. Replaced constraints with comment: # constraints # ... (the constraints remain the same)	model.AddAbsEquality(changedVar, sum(workerHours[A, h] for h in range(H)) - sum(workerHours[B, h] for h in range(H))) model.Add(changedVar <= workerHourDiffThreshold)
14	Nurse K cannot be assigned all AM slots over 3 consecutive days.	for d in range(D - 2): model.Add(nurseDaySlot[K, d, AM] + nurseDaySlot[K, d + 1, AM] + nurseDaySlot[K, d + 2, AM] <= 2	Removed code again. Added code: for day in range(D - MaxConsecutiveAMShifts + 1): model.Add(sum(nurseDaySlot[K, d, AM] for d in range(day, day + MaxConsecutiveAMShifts)) <= MaxConsecutiveAMShifts	for d in range(D-3): model.Add(nurseDaySlot[K,d,AM] + nurseDaySlot[K,d+1,AM] + nurseDaySlot[K,d+2,AM] <= 2)

Figure 10. Table of some prompts and generated code from OpenAI GPT-4 and Claude Haiku as the coding agent.

will remove the existing constraints despite instructions in the system prompt to output the full code. In addition, it frequently hallucinates parameters such as MaxConsecutiveAMShifts and even library functions such as AddEquality.

5 Conclusion and Future Work

We propose a framework RAGDYS which automates the process of adding dynamic constraints to existing static scheduling problems, thereby minimizing the need for human optimization experts.

For future work, we plan to encompass a wider range of worker scheduling problems and explore other combinatorial optimization problems that can be formulated with minimum perturbation constraints. Routing problems are well-studied and routing-specific methods (such as AddCircuit in ORTools) in constraint programming libraries should be useful and intuitive for LLMs to make use of. Additionally, further prompt optimization can be done, specifically on the intricacies and techniques specific to the solver library. For example, to further improve the performance of the coding agent, our coding agent prompt requires additional knowledge on how the Python absolute (abs) function cannot be used when defining variables and constraints in OR-Tools. We should explore strategies for prompt optimization and apply them to enhance the current framework.

Acknowledgements

We would like to thank Nayan Rokhade, Derek Cho, and Ming Bing Ng for assistance during code implementation.

References

- [1] A. AhmadiTeshnizi, W. Gao, and M. Udell. OptiMUS: Optimization modeling using mip solvers and large language models, 2024. URL <https://openreview.net/forum?id=2FAPahXyVh>.
- [2] Anthropic. The claude 3 model family: Opus, sonnet, haiku, 2024. URL <https://paperswithcode.com/paper/the-claude-3-model-family-opus-sonnet-haiku>.
- [3] H. El Sakkout, T. Richards, and M. Wallace. Minimal perturbation in dynamic scheduling. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*. John Wiley & Sons. Citeseer, 1998.
- [4] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, and H. Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023.
- [5] Gemini. Gemini: A family of highly capable multimodal models, 2024.
- [6] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.
- [7] OpenAI. Gpt-4 technical report, 2024.
- [8] F. Rossi, P. Van Beek, and T. Walsh. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–211, 2008.

- [9] K. Valmeekam, A. Olmo, S. Sreedharan, and S. Kambhampati. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). *arXiv preprint arXiv:2206.10498*, 2022.
- [10] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. In *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, 2020.

Appendices

A Gig Scheduling Problem (GSP) Formulation

A.1 Variables

- W, H, T represents the number of workers, hours and tasks respectively.
- $x_{w,h}$ and $y_{w,t}$ is the binary worker-hour assignment and the binary worker-task assignment.
- $A_{w,h}$ and $S_{w,t}$ is the worker-hour availability and worker-task skill compatibility parameters.
- $taskHour(t)$ returns the hour of task t , and T_h is the set of tasks in hour h .
- $s_{w,h}$ and $e_{w,h}$ are the intermediate binary variables indicating the start and end of a block of consecutive working hours starting / ending at hour h for worker w .
- $BMin, BMax$, and $RMin$ are the parameters for minimum block length, maximum block length, and minimum rest length.
- u_t is the soft constraint variable indicating if task t is unassigned.

A.2 Constraint Description

- **Worker Hour Availability:** hard constraint ensuring that workers can only be assigned hours they are available in as specified in $A_{w,h}$.
- **Worker Task Skill Requirements:** hard constraint ensuring that workers can only be assigned tasks if they meet the skill requirements, specified in $workerTaskSkills[w, t]$.
- **Worker Task Hour Precedence:** hard constraint ensuring that a worker must be assigned the task's hour before being assigned the task.
- **No Multi-task:** hard constraint ensuring that a worker can only work at most 1 task per hour.
- **Minimum Block Length:** hard constraint ensuring an assigned block of hours for a worker is at least $BMin$ long.
- **Maximum Block Length:** hard constraint ensuring an assigned block of hours for a worker is at most $BMax$ long.
- **Minimum Rest:** hard constraint ensuring the length of rest time between consecutive blocks is at least $RMin$ long.
- **Unassigned Tasks:** soft constraint which counts the number of tasks that are not assigned to any worker.

A.3 Constraint Formulation

$$x_{w,h} \leq A_{w,h}, \forall w \in W, h \in H \quad (4)$$

$$y_{w,t} \leq S_{w,t}, \forall w \in W, t \in T \quad (5)$$

$$y_{w,t} \leq x_{w,taskHour(t)}, \forall w \in W, t \in T \quad (6)$$

$$\sum_t^{T_h} y_{w,t} \leq 1, \forall w \in W, \forall h \in H \quad (7)$$

Intermediate constraints formulating the start and end of blocks.

$$s_{w,h} \geq x_{w,h} - x_{w,h-1}, \forall w \in W, \forall h \in H \quad (8)$$

$$s_{w,0} = x_{w,0}, \forall w \in W \quad (9)$$

$$e_{w,h} \geq x_{w,h} - x_{w,h+1}, \forall w \in W, \forall h \in H \quad (10)$$

$$e_{w,H-1} = x_{w,H-1}, \forall w \in W \quad (11)$$

Minimum block length constraint

$$\sum_{h'=h}^{h+BMin-1} x_{w,h'} \geq s_{w,h} \times BMin \quad (12)$$

Maximum block length constraint

$$\sum_{h'=h}^{h+BMax} 1 - x_{w,h'} \geq s_{w,h} \quad (13)$$

Minimum Rest constraint

$$e_{w,h} \leq x_{w,h'}, \forall w \in W, \forall h \in H, \forall h' \in [h, h + RMin] \quad (14)$$

Unassigned Task constraint

$$\sum_w^W y_{w,t} \geq 1 - u_t, \forall t \in T \quad (15)$$

A.4 Objective

Minimize the following:

$$\sum_{w,h}^{W,H} x_{w,h} + \alpha \sum_t^T u_t \quad (16)$$

$$\alpha = 1000$$

B Nurse Scheduling Problem (NSP) Formulation

B.1 Variables

- N, D, S, T represents the number of nurses, days and shifts and slots respectively.
- There are 3 slots, morning AM, afternoon PM, and night ND.
- $x_{n,d,s}$ and $y_{n,d,t}$ is the binary nurse-day-shift assignment and the binary nurse-day-slot assignment.
- $A_{n,d,s}$ is the nurse availability parameter for each day and shift.
- $shiftSlot(s)$ returns the slot number for shift s .
- $shiftHours(s)$ returns the number of hours for shift s .
- $Y_{d,t}$ is the demand for the nurses required on day d slot t .
- $R_{n,d}$ is the specified rest days for each nurse for each day.
- $MinHours, MaxHours, MaxWorkDays$ represent the minimum hours worked per nurse, maximum hours worked per nurse, and the maximum days worked per nurse.
- $surplus$ and $shortfall$ are intermediate integer variables counting the over or under supply of nurses on day d slot t .

B.2 Constraint Description

- **Nurse Availability:** a nurse can only be assigned a day and shift if she is available.
- **At most 1 shift per day:** a nurse can at most be assigned 1 shift per day.
- **Nurse Shift Slot:** a nurse assigned to a shift indicates the nurse is assigned the shift's slot.
- **Demand constraint:** the number of nurses assigned to day d and slot t equals $Y_{d,t} + surplus_{d,t} - shortfall_{d,t}$
- **Minimum hours:** minimum hours assigned to nurse is $MinHours$.
- **Maximum hours:** maximum hours assigned to nurse is $MaxHours$.

- **Maximum days:** maximum days assigned to a nurse is $MaxDays$
- **No ND-AM pattern:** a nurse cannot work a night slot followed by a morning slot.

B.3 Constraint Formulation

$$x_{n,d,s} \leq A_{n,d,s}, \forall n, d, s \in N, D, S \quad (17)$$

$$\sum_s^S x_{n,d,s} \leq 1, \forall n, d \in N, D \quad (18)$$

$$x_{n,d,s} \leq y_{n,d,shiftSlot(s)}, \forall n, d, s \in N, D, S \quad (19)$$

$$\sum_{d,s}^{D,S} x_{n,d,s} \times shiftHours(s) \geq MinHours, \forall n \in N \quad (20)$$

$$\sum_{d,s}^{D,S} x_{n,d,s} \times shiftHours(s) \leq MaxHours, \forall n \in N \quad (21)$$

$$\sum_{d,s}^{D,S} x_{n,d,s} \leq MaxDays, \forall n \in N \quad (22)$$

Demand Constraint

$$\sum_n^N y_{n,d,t} = Y_{d,t} + surplus_{d,t} - shortfall_{d,t} \quad (23)$$

No ND-AM Pattern

$$y_{n,d,ND} + y_{n,d+1,AM} \leq 1, \forall n, d \in N, D \quad (24)$$

B.4 Objective

Minimize the following:

$$\sum_{d,t}^{D,T} surplus_{d,t} + shortfall_{d,t} \quad (25)$$

C Planning Agent Prompt Example

Role

You are a planning agent who excels at performing Constraint Programming tasks, tasked to model a new c
Think step by step carefully before answering the problem.

Instruction

Based on the user problem statement and the new constraint you have to come up with:

- New Parameters
- New Variables
- New Constraint Formulation

Example

User

Problem:

You are tasked to solve a worker scheduling problem.

In this problem, there are {W} workers, {H} hours, and {T} tasks.

Each worker can be assigned to each hour.

Each task has a fixed hour, and can be assigned to each worker. {taskHour[t]} specified which hour task

The main task is to assign workers to hours and tasks to workers, subject to various constraints.

The objective is to minimize the following:

total_hours_assigned + {UNASSIGNED_PENALTY} * total_number_of_unassigned_tasks

Parameters:

W: Number of workers

H: Number of hours

T: Number of tasks

BMin: Minimum block length

BMax: Maximum block length

RMin: Minimum rest length

availableHours[w,h]: Indicates if worker w is available to be assigned hour h (1 if available, otherwis

workerTaskSkills[w,t]: Indicates if worker w is skilled enough to be assigned task t (1 if sufficiently

taskHour[t]: Returns the hour of the task

UNASSIGNED_PENALTY: Penalty per unassigned task in the objective function

Decision Variables:

workerHours[w,h]: Indicates if worker w is assigned hour h

workerTasks[w,t]: Indicates if worker w is assigned task t

startBlock[w,h]: Indicates if worker w is assigned a consecutive block of hours starting at hour h

endBlock[w,h]: Indicates if worker w is assigned a consecutive block of hours ending at hour h

unassignedTasks[t]: Indicates if task t is not assigned all workers

Constraints:

1) Available hours constraint. If a worker is assigned an hour, he must be available for that hour.

workerHours[w,h] <= availableHours[w,h] for each w for each h

2) Worker task skill constraint. If a worker is assigned a task, he must be skilled enough for that tas

workerTask[w,t] <= workerTaskSkills[w,t] for each w for each t

3) Task hour assignment constraint. If a worker is assigned a task, he must be assigned the task's hour

workerTask[w,t] <= workerHours[w,taskHour[t]] for each w for each t

4) At most 1 task assigned for each worker for each hour

sum(workerTask[w,t] for each t if taskHour[t]=h) <= 1 for each w for each h

5) Start block constraint. If a worker w starts a block of consecutive worker-hour assignments at hour

workerHours[w,h] + (1-workerHours[w,h-1]) - 1 <= startBlock[w,h] for each w for each h

workerHours[w,0] <= startBlock[w,0] for each w

6) End block constraint. If a worker w ends a block of consecutive worker-hour assignments at hour h, e

workerHours[w,h] + (1-workerHours[w,h+1]) - 1 <= endBlock[w,h] for each w for each h

workerHours[w,H-1] <= endBlock[w,H-1] for each w

7) Minimum block length constraint. If a worker starts a block at hour h, the worker must be assigned t

startBlock[w,h] <= workerHours[w,x] for each w for each h for each x in range(h,h+BMin)

8) Maximum block length constraint. If a worker starts a block at hour h, the worker must end within {B

startBlock[w,h] <= sum(endBlock[w,x] for x in range(h,h+BMax)) for each w for each h

9) Minimum Rest constraint. The minimum length of rest (block of consecutive worker-hour = 0) between t and t+1 is $\text{endBlock}[w,h] \leq 1 - \text{workerHours}[w,x]$ for each w for each h for each x in range(h+1,h+RMin+1)
10) Unassigned Task constraint. If a task is not assigned to any worker, $\text{unassignedTasks}[t] = 1$.

Given the above context, provide the mathematical formulation for the new constraint mentioned below.:

New Constraint:

11) Formulate a constraint such that worker A does not work from hours H1 to H2.

You are free to add parameters and decision variables.

New Parameters:

A: Worker affected from hours H1 to H2

H1: Start of hours where worker A does not work

H2: End of hours where worker A does not work

New Variables:

None

New Constraints:

$\text{workerHours}[A,h] = 0$ for each h in range(H1,H2)

Your turn

Problem

You are tasked to solve a nurse rostering problem.

In this problem, there are {N} nurses, {D} days, {S} shifts and {T} slots.

Each worker can be a shift for each day.

Each shift takes a certain number of hours, specified in {shiftHours}.

Each shift fills either a Morning(AM) (Slot 0), Afternoon(PM) (Slot 1), or Night(ND) (Slot 2) slot.

There is a demand requirement for each day for each slot, where we want to assign the nurses their shift.

The main task is to assign nurses a shift for each day (or no shift if the nurse is not working for that day).

The objective is to minimize the following:

surplus + shortfall

surplus is the total number of nurses assigned to a slot more than the specified demand for that day and

shortfall is the total number of nurses assigned to a slot less than the specified demand for that day

Parameters:

N: Number of nurses

D: Number of days

S: Number of shifts

T: Number of slots

availableShifts[n,d,s]: Indicates if nurse n is available to be assigned day d and slot s (1 if available, 0 otherwise)

shiftSlot[s] returns the slot number for shift s. (Eg. 7am to 11am is a Morning shift(AM), so it has a slot number of 0)

shiftHours[s] = Integer value for the number of hours in shift s. (Eg. 7am to 11am shift has a value of 4)

demandSlot[d,t] = Integer value of the demand required specified per day d per slot t

restDays[n,d]: Indicates if nurse n is resting (Not available) on day d.

MinHours = data["MinHours"]

MaxHours = data["MaxHours"]

MaxWorkingDays = data["MaxWorkingDays"]

Decision Variables:

nurseDayShift[n,d,s]: Indicates if nurse n is assigned day d and shift s

nurseDaySlot[n,d,t]: Indicates if nurse n is assigned a shift in day d slot t

surplus[d,t] = Integer value of the surplus nurses assigned on day d slot t. (If X more nurses are assigned than demand, surplus is X)

shortfall[d,t] = Integer value of the shortfall nurses assigned on day d slot t. (If X less nurses are assigned than demand, shortfall is X)

Constraints:

1) Available shift constraint. If nurse n is assigned day d and shift s, she must be available on day d and shift s

$\text{nurseDayShift}[n,d,s] \leq \text{availableShifts}[n,d,s]$ for each n for each d for each s

2) Max 1 shift per day constraint. At most 1 shift per day constraint

$\sum(\text{nurseDayShift}[n,d,s] \text{ for each } s) \leq 1$, for each n for each d

3) Shift to slot assignment constraint. If a nurse n is assigned day d shift s , then $nurseDaySlot[n,d,s]$
 $nurseDaySlot[n,d,t] \leq \sum(nurseDayShift[n,d,s] \text{ for each } s \text{ if } shiftSlot[s] == t)$

4) Demand Requirement Constraint. Number of nurses assigned for each day d for each slot t == demandSlot[d,t]
 $\sum(nurseDaySlot[n,d,t] \text{ for each } n) == demandSlot[d,t] + surplus[d,t] - shortfall[d,t]$

5) Minimum Hours Constraint. Each nurse must work at at least {MinHours}
 $\sum(shiftHours[s] * nurseDayShift[n,d,s] \text{ for each } d \text{ for each } s) \geq MinHours, \text{ for each } n$

6) Maximum Hours Constraint: Each nurse must work at most {MaxHours}
 $\sum(shiftHours[s] * nurseDayShift[n,d,s] \text{ for each } d \text{ for each } s) \leq MaxHours, \text{ for each } n$

7) Rest Days Constraint. If a nurse n is assigned any shift on day d , she must not be on a rest.
 $nurseDayShift[n,d,s] \leq 1 - restDays[n,d] \text{ for each } n \text{ for each } d \text{ for each } s$

8) Maximum number of working days per week constraint. Each nurse must work at most (MaxWorkingDays) per week
 $\sum(nurseDayShift[n,w*7 + d,s] \text{ for } d \text{ in range}(7) \text{ for each } s) \leq MaxWorkingDays, \text{ for each } n \text{ for each week } w$

9) No consecutive Night (ND) shift - Morning (AM) shift assignment for any nurse constraint.
 $nurseDaySlot[n,d,ND] + nurseDaySlot[n,d+1,AM] \leq 1, \text{ for each } n \text{ for each } d$

New Constraint

Nurse K cannot work on days $D1$.

D Coding Agent Prompt Example

Role

You are a Coding Agent python developer who excels at solving Constraint Programming tasks using Google OR-Tools package to solve the problem as shown in the below example. Think step by step carefully before writing the code.

Instruction

Based on the user problem statement and mathematical instructions provided you have to write the python code using OR-Tools package to solve the problem as shown in the below example.

Build the python code which is written to a file later on and the data is given to the python file as a

Note

Note that your response should only contain python code without any explanation or run command is needed

Example

User

Problem:

You are tasked to solve a worker scheduling problem.

In this problem, there are {W} workers, {H} hours, and {T} tasks.

Each worker can be assigned to each hour.

Each task has a fixed hour, and can be assigned to each worker. {taskHour[t]} specified which hour task

The main task is to assign workers to hours and tasks to workers, subject to various constraints.

The objective is to minimize the following:

total_hours_assigned + {UNASSIGNED_PENALTY} * total_number_of_unassigned_tasks

Parameters:

W: Number of workers

H: Number of hours

T: Number of tasks

BMin: Minimum block length

BMax: Maximum block length

RMin: Minimum rest length

availableHours[w,h]: Indicates if worker w is available to be assigned hour h (1 if available, otherwise 0)

workerTaskSkills[w,t]: Indicates if worker w is skilled enough to be assigned task t (1 if sufficiently skilled, otherwise 0)

taskHour[t]: Returns the hour of the task

UNASSIGNED_PENALTY: Penalty per unassigned task in the objective function

Decision Variables:

workerHours[w,h]: Indicates if worker w is assigned hour h

workerTasks[w,t]: Indicates if worker w is assigned task t

startBlock[w,h]: Indicates if worker w is assigned a consecutive block of hours starting at hour h

endBlock[w,h]: Indicates if worker w is assigned a consecutive block of hours ending at hour h

unassignedTasks[t]: Indicates if task t is not assigned all workers

Constraints:

1) Available hours constraint. If a worker is assigned an hour, he must be available for that hour.

workerHours[w,h] <= availableHours[w,h] for each w for each h

2) Worker task skill constraint. If a worker is assigned a task, he must be skilled enough for that task.

workerTask[w,t] <= workerTaskSkills[w,t] for each w for each t

3) Task hour assignment constraint. If a worker is assigned a task, he must be assigned the task's hour.

workerTask[w,t] <= workerHours[w,taskHour[t]] for each w for each t

4) At most 1 task assigned for each worker for each hour

sum(workerTask[w,t] for each t if taskHour[t]=h) <= 1 for each w for each h

5) Start block constraint. If a worker w starts a block of consecutive worker-hour assignments at hour h

workerHours[w,h] + (1-workerHours[w,h-1]) - 1 <= startBlock[w,h] for each w for each h

workerHours[w,0] <= startBlock[w,0] for each w

6) End block constraint. If a worker w ends a block of consecutive worker-hour assignments at hour h, e

workerHours[w,h] + (1-workerHours[w,h+1]) - 1 <= endBlock[w,h] for each w for each h

workerHours[w,H-1] <= endBlock[w,H-1] for each w

7) Minimum block length constraint. If a worker starts a block at hour h, the worker must be assigned t

startBlock[w,h] <= workerHours[w,x] for each w for each h for each x in range(h,h+BMin)

8) Maximum block length constraint. If a worker starts a block at hour h, the worker must end within {B

startBlock[w,h] <= sum(endBlock[w,x] for x in range(h,h+BMax)) for each w for each h

9) Minimum Rest constraint. The minimum length of rest (block of consecutive worker-hour = 0) between t

endBlock[w,h] <= 1-workerHours[w,x] for each w for each h for each x in range(h+1,h+RMin+1)

10) Unassigned Task constraint. If a task is not assigned to any worker, unassignedTasks[t] = 1.

Given the above context, provide the mathematical formulation for the new constraint mentioned below:

New Constraint:

11) Formulate a constraint such that worker A does not work from hours H1 to H2.

You are free to add parameters and decision variables.

New Parameters:

A: Worker affected from hours H1 to H2

H1: Start of hours where worker A does not work

H2: End of hours where worker A does not work

New Variables:

None

New Constraints:

workerHours[A,h] = 0 for each h in range(H1,H2)

Current Code

```
```python
from ortools.sat.python import cp_model
import pandas as pd
import sys
import os
sys.path.append(os.getcwd())
from readdata import read_input_json

Read data
current_dir = os.getcwd()
Parameters
input_data_file = os.path.join(current_dir, "gsp_input.json")
data = read_input_json(input_data_file)
W = data["W"]
H = data["H"]
T = data["T"]
availableHours = data["availableHours"]
taskHour = data["taskHour"]
workerTaskSkills = data["workerTaskSkills"]
BMin = data["BMin"]
BMax = data["BMax"]
RMin = data["RMin"]

Start model object
model = cp_model.CpModel()

Create Variables
workerHours = model.NewBoolVarSeries("workerHours", pd.Index([(w,h) for w in range(W) for h in range(H)]))
startBlock = model.NewBoolVarSeries("startBlock", pd.Index([(w,h) for w in range(W) for h in range(H)]))
endBlock = model.NewBoolVarSeries("endBlock", pd.Index([(w,h) for w in range(W) for h in range(H)]))
taskAssigned = model.NewBoolVarSeries("taskAssigned", pd.Index([(w,t) for w in range(W) for t in range(T)]))
unassignedTask = model.NewBoolVarSeries("unassignedTask", pd.Index([t for t in range(T)]))

Main Constraints - start
Constraint 1: Available hours constraint
for w in range(W):
 for h in range(H):
 model.Add(workerHours[w,h] <= availableHours[w,h])

Constraint 2: Worker task skill constraint
for w in range(W):
 for t in range(T):
 model.Add(taskAssigned[w,t] <= workerTaskSkills[w,t])

Constraint 3: Task hour assignment constraint
for w in range(W):
 for t in range(T):
 model.Add(taskAssigned[w,t] <= workerHours[w,taskHour[t]])
```

```

Constraint 4: At most 1 task assigned for each worker for each hour
for w in range(W):
 for h in range(H):
 model.Add(sum(taskAssigned[w,t] for t in range(T) if taskHour[t] == h) <= 1)

Constraint 5: startBlock[w,h] = 1 indicates the start of a consecutive assignment of workerHours for
for w in range(W):
 for h in range(1, H):
 model.Add(workerHours[w,h-1].Not() + workerHours[w,h] - 1 <= startBlock[w,h])

 model.Add(workerHours[w,0] <= startBlock[w,0])

Constraint 6: endBlock[w,h] = 1 indicates the end of a consecutive assignment of workerHours for wor
for w in range(W):
 for h in range(0,H-1):
 model.Add(workerHours[w,h] + workerHours[w,h+1].Not() - 1 <= endBlock[w,h])

 model.Add(workerHours[w,H-1] <= endBlock[w,H-1])

Constraint 7: Min Block Length constraint
for w in range(W):
 for h in range(H-BMin):
 for x in range(h, h+BMin):
 model.Add(startBlock[w,h] <= workerHours[w,x])

 for h in range(H-BMin, H):
 model.Add(startBlock[w,h] == 0)

Constraint 8: Max Block Length Constraint
for w in range(W):
 for h in range(H-BMax-1):
 model.Add(startBlock[w,h] <= sum(workerHours[w,x].Not() for x in range(h+1,h+BMax+1)))

Constraint 9: Minimum Rest constraint
for w in range(W):
 for h in range(H):
 for x in range(h+1, min(H, h+RMin+1)):
 model.Add(endBlock[w,h] <= workerHours[w,x].Not())

Constraint 10: Unassigned task constraint
for t in range(T):
 model.Add(sum(taskAssigned[w,t] for w in range(W)) >= 1 - unassignedTask[t])
Main Constraints - end

Dynamic Constraints

Objective
total_hours_assigned = sum(workerHours[w,h] for w in range(W) for h in range(H))
total_unassigned_tasks = sum(unassignedTask[t] for t in range(T))
UNASSIGNED_PENALTY = 1000
objective = total_hours_assigned + UNASSIGNED_PENALTY * total_unassigned_tasks
model.Minimize(objective)

Run Solver
solver = cp_model.CpSolver()
solver.parameters.linearization_level = 0
solver.parameters.max_time_in_seconds = 180
solver.parameters.optimize_with_core = True
solver.parameters.num_search_workers = 1
solver.parameters.log_search_progress = True
status = solver.Solve(model)

```



```
'''
```

```
Coding Agent
```

```
```python
```

```
from ortools.sat.python import cp_model
```

```
import pandas as pd
```

```
from readdata import read_input_json
```

```
# Global Parameters
```

```
input_data_file = "simple_input.json"
```

```
data = read_input_json(input_data_file)
```

```
W = data["W"]
```

```
H = data["H"]
```

```
T = data["T"]
```

```
availableHours = data["availableHours"]
```

```
taskHour = data["taskHour"]
```

```
workerTaskSkills = data["workerTaskSkills"]
```

```
BMin = data["BMin"]
```

```
BMax = data["BMax"]
```

```
RMin = data["RMin"]
```

```
A = data["A"]
```

```
H1 = data["H1"]
```

```
H2 = data["H2"]
```

```
# Global Parameters - end
```

```
# Start model object
```

```
model = cp_model.CpModel()
```

```
# Global Variables
```

```
workerHours = model.NewBoolVarSeries("workerHours", pd.Index([(w,h) for w in range(W) for h in range(H)]))
```

```
startBlock = model.NewBoolVarSeries("startBlock", pd.Index([(w,h) for w in range(W) for h in range(H)]))
```

```
endBlock = model.NewBoolVarSeries("endBlock", pd.Index([(w,h) for w in range(W) for h in range(H)]))
```

```
taskAssigned = model.NewBoolVarSeries("taskAssigned", pd.Index([(w,t) for w in range(W) for t in range(T)]))
```

```
unassignedTask = model.NewBoolVarSeries("unassignedTask", pd.Index([t for t in range(T)]))
```

```
None
```

```
# Global Variables - end
```

```
# Main Constraints - start
```

```
## Constraint: 1
```

```
"""
```

```
Desc: Skill prerequisite constraint: If a task is assigned to a worker, the worker-task skills must be co
```

```
Type: Main
```

```
Parameters: None
```

```
Variables: None
```

```
"""
```

```
### Code:
```

```
for w in range(W):
```

```
    for t in range(T):
```

```
        model.Add(taskAssigned[w,t] <= workerTaskSkills[w,t])
```

```
## Constraint: 2
```

```
"""
```

```
Desc: Time prerequisite constraint: If a task is assigned to a worker, the worker must be assigned to t
```

```
Type: Main
```

```
Parameters: None
```

```
Variables: None
```

```
"""
```

```
### Code:
```

```
for w in range(W):
```

```
    for t in range(T):
```

```
        model.Add(taskAssigned[w,t] <= workerHours[w,taskHour[t]])
```

```
## Constraint: 3
```

```
"""
```

```
Desc: No multi-tasking constraint: A worker can only be assigned at most 1 task per hour. A worker can
```

```

Type: Main
Parameters: None
Variables: None
"""
### Code:
for w in range(W):
    for h in range(H):
        model.Add(sum(taskAssigned[w,t] for t in range(T) if taskHour[t] == h) <= 1)

## Constraint: 4
"""
Desc: Time availability constraint: If a worker is assigned an hour, the worker must be available for t
Type: Main
Parameters: None
Variables: None
"""
### Code:
for w in range(W):
    for h in range(H):
        model.Add(workerHours[w,h] <= availableHours[w,h])

## Constraint: 5
"""
Desc: startBlock[w,h] = 1 indicates the start of a consecutive assignment of workerHours for worker w s
Type: Main
Parameters: None
Variables: None
"""
### Code:
for w in range(W):
    for h in range(1, H):
        model.Add(workerHours[w,h-1].Not() + workerHours[w,h] - 1 <= startBlock[w,h])

        model.Add(workerHours[w,0] <= startBlock[w,0])

## Constraint: 6
"""
Desc: endBlock[w,h] = 1 indicates the end of a consecutive assignment of workerHours for worker w endin
Type: Main
Parameters: None
Variables: None
"""
### Code:
for w in range(W):
    for h in range(1, H):
        model.Add(workerHours[w,h-1] + workerHours[w,h].Not() - 1 <= endBlock[w,h-1])

        model.Add(workerHours[w,H-1] <= endBlock[w,H-1])

## Constraint: 7
"""
Desc: If a task is not assigned to a single worker (if taskAssigned[w,t] = 0 for all w), then unassignedTask[t]
Type: Main
Parameters: None
Variables: None
"""
### Code:
for t in range(T):
    model.Add(sum(taskAssigned[w,t] for w in range(W)) >= 1 - unassignedTask[t])

## Constraint: 8
"""
Desc: Min Block Length constraint: Minimum length of consecutive hours assigned to a worker is {BMin}.

```

```

Type: Main
Parameters: None
Variables: None
"""
### Code:
for w in range(W):
    for h in range(H-BMin):
        for x in range(h, h+BMin):
            model.Add(startBlock[w,h] <= workerHours[w,x])

    for h in range(H-BMin, H):
        model.Add(startBlock[w,h] == 0)

## Constraint: 9
"""
Desc: Max Block Length Constraint: Maximum length of consecutive hours assigned to a worker is {BMax}.
Type: Main
Parameters: None
Variables: None
"""
### Code:
for w in range(W):
    for h in range(H-BMax-1):
        model.Add(startBlock[w,h] <= sum(workerHours[w,x].Not() for x in range(h+1,h+BMax+1)))

## Constraint: 10
"""
Desc: Minimum Rest constraint: Minimum length of consecutive non-working hours after hour h for worker
Type: Main
Parameters: None
Variables: None
"""
### Code:
for w in range(W):
    for h in range(H):
        for x in range(h+1, min(H, h+RMin+1)):
            model.Add(endBlock[w,h] <= workerHours[w,x].Not())

## Constraint: 11

### Code:

for h in range(H1, H2+1):
    model.Add(workerHours[A,h] == 0)

# Objective
total_hours_assigned = sum(workerHours[w,h] for w in range(W) for h in range(H))
total_unassigned_tasks = sum(unassignedTask[t] for t in range(T))
UNASSIGNED_PENALTY = 1000
objective = total_hours_assigned + UNASSIGNED_PENALTY * total_unassigned_tasks
model.Minimize(objective)

# Run Solver
solver = cp_model.CpSolver()
solver.parameters.linearization_level = 0
solver.parameters.max_time_in_seconds = 60 * 3
solver.parameters.optimize_with_core = True
solver.parameters.num_search_workers = 1
solver.parameters.log_search_progress = True
status = solver.Solve(model)
'''

```

Your turn:

Problem:

You are tasked to solve a nurse rostering problem.

In this problem, there are {N} nurses, {D} days, {S} shifts and {T} slots.
Each worker can be a shift for each day.
Each shift takes a certain number of hours, specified in {shiftHours}.
Each shift fills either a Morning(AM) (Slot 0), Afternoon(PM) (Slot 1), or Night(ND) (Slot 2) slot.
There is a demand requirement for each day for each slot, where we want to assign the nurses their shift.
The main task is to assign nurses a shift for each day (or no shift if the nurse is not working for that day).
The objective is to minimize the following:
surplus + shortfall
surplus is the total number of nurses assigned to a slot more than the specified demand for that day and
shortfall is the total number of nurses assigned to a slot less than the specified demand for that day

Parameters:

N: Number of nurses
D: Number of days
S: Number of shifts
T: Number of slots
availableShifts[n,d,s]: Indicates if nurse n is available to be assigned day d and slot s (1 if available, 0 otherwise)
shiftSlot[s] returns the slot number for shift s. (Eg. 7am to 11am is a Morning shift(AM), so it has a value of 0)
shiftHours[s] = Integer value for the number of hours in shift s. (Eg. 7am to 11am shift has a value of 4)
demandSlot[d,t] = Integer value of the demand required specified per day d per slot t
restDays[n,d]: Indicates if nurse n is resting (Not available) on day d.
MinHours = data["MinHours"]
MaxHours = data["MaxHours"]
MaxWorkingDays = data["MaxWorkingDays"]

Decision Variables:

nurseDayShift[n,d,s]: Indicates if nurse n is assigned day d and shift s
nurseDaySlot[n,d,t]: Indicates if nurse n is assigned a shift in day d slot t
surplus[d,t] = Integer value of the surplus nurses assigned on day d slot t. (If X more nurses are assigned than demand, surplus is X)
shortfall[d,t] = Integer value of the shortfall nurses assigned on day d slot t. (If X less nurses are assigned than demand, shortfall is X)

Constraints:

- 1) Available shift constraint. If nurse n is assigned day d and shift s, she must be available on day d
 $nurseDayShift[n,d,s] \leq availableShifts[n,d,s]$ for each n for each d for each s
- 2) Max 1 shift per day constraint. At most 1 shift per day constraint
 $sum(nurseDayShift[n,d,s] \text{ for each } s) \leq 1$, for each n for each d
- 3) Shift to slot assignment constraint. If a nurse n is assigned day d shift s, then nurseDaySlot[n,d,t] must be 1 if shiftSlot[s] == t
 $nurseDaySlot[n,d,t] \leq sum(nurseDayShift[n,d,s] \text{ for each } s \text{ if } shiftSlot[s] == t)$
- 4) Demand Requirement Constraint. Number of nurses assigned for each day d for each slot t == demandSlot[d,t] + surplus[d,t] - shortfall[d,t]
 $sum(nurseDaySlot[n,d,t] \text{ for each } n) == demandSlot[d,t] + surplus[d,t] - shortfall[d,t]$
- 5) Minimum Hours Constraint. Each nurse must work at least {MinHours} hours
 $sum(shiftHours[s] * nurseDayShift[n,d,s] \text{ for each } d \text{ for each } s) \geq MinHours$, for each n
- 6) Maximum Hours Constraint: Each nurse must work at most {MaxHours} hours
 $sum(shiftHours[s] * nurseDayShift[n,d,s] \text{ for each } d \text{ for each } s) \leq MaxHours$, for each n
- 7) Rest Days Constraint. If a nurse n is assigned any shift on day d, she must not be on a rest day.
 $nurseDayShift[n,d,s] \leq 1 - restDays[n,d]$ for each n for each d for each s
- 8) Maximum number of working days per week constraint. Each nurse must work at most (MaxWorkingDays) per week
 $sum(nurseDayShift[n,w*7 + d,s] \text{ for } d \text{ in range}(7) \text{ for each } s) \leq MaxWorkingDays$, for each n for each week w
- 9) No consecutive Night(ND) shift - Morning (AM) shift assignment for any nurse constraint.
 $nurseDaySlot[n,d,ND] + nurseDaySlot[n,d+1,AM] \leq 1$, for each n for each d

New Constraint

Nurse K is not available to work on day D1.

Parameters

K: Nurse affected on day D1.
D1: Nurse K cannot work on this day.

Variables

None

Constraints

$nurseDayShift[K,D1,s] = 0$ for each s

Current code:

```
```python
from ortools.sat.python import cp_model
```

```

import pandas as pd
import numpy as np
import sys
import os
sys.path.append(os.getcwd())
from readdata import read_input_json

Read data
current_dir = os.getcwd()
input_file = os.path.join(current_dir, "nsp_input.json")
data = read_input_json(input_file)

Parameters
N = data["N"] # Num nurses
D = data["D"] # Num days
S = data["S"] # Num Shifts. A shift can be 7am to 11am, 9am to 1pm, 8pm to 7am, etc.
T = data["T"] # Num Slots: AM = 0, PM = 1, ND = 2 (Morning, Afternoon Night). A shift is classified under
AM,PM,ND = 0,1,2
availableShifts = data["availableShifts"] # availableShifts[n,d,s] returns a boolean if nurse n is available
shiftSlot = data["shiftSlot"] # shiftSlot[s] returns the slot number for shift s. (7am to 11am is a Morning slot)
shiftHours = data["shiftHours"] # shiftHours[s] returns the number of hours in shift s. (7am to 11am is 4 hours)
demandSlot = data["demandSlot"] # Demand is specified per day per slot in demandSlot[d,t]
restDays = data["restDays"] # restDays[n,d]=1 implies that availableShifts[n,d,s] = 0 for all s
MinHours = data["MinHours"]
MaxHours = data["MaxHours"]
MaxWorkingDays = data["MaxWorkingDays"]

Start model object
model = cp_model.CpModel()

Variables
nurseDayShift = model.NewBoolVarSeries("nurseDayShift", pd.Index([(n,d,s) for n in range(N) for d in range(D) for s in range(S)]))
nurseDaySlot = model.NewBoolVarSeries("nurseDaySlot", pd.Index([(n,d,t) for n in range(N) for d in range(D) for t in range(T)]))
surplus = model.NewIntVarSeries("surplus", pd.Index([(d,t) for d in range(D) for t in range(T)]), 0, 3)
shortfall = model.NewIntVarSeries("shortfall", pd.Index([(d,t) for d in range(D) for t in range(T)]), 0, 3)

Constraints
Available shift constraint
for n in range(N):
 for d in range(D):
 for s in range(S):
 model.Add(nurseDayShift[n,d,s] <= availableShifts[n,d,s])

At most 1 shift per day constraint
for n in range(N):
 for d in range(D):
 model.Add(sum(nurseDayShift[n,d,s] for s in range(S)) <= 1)

Nurse Shift Slot assignment Constraint
for n in range(N):
 for d in range(D):
 for t in range(T):
 model.Add(nurseDaySlot[n,d,t] <= sum(nurseDayShift[n,d,s] for s in range(S) if shiftSlot[s]==t))

Demand Constraint: Number of nurses assigned for each day d for each slot t == demandSlot[d,t] + surplus[d,t] - shortfall[d,t]
for d in range(D):
 for t in range(T):
 model.Add(sum(nurseDaySlot[n,d,t] for n in range(N)) == demandSlot[d,t] + surplus[d,t] - shortfall[d,t])

Min Hours Constraint
for n in range(N):
 model.Add(sum(nurseDayShift[n,d,s] * shiftHours[s] for d in range(D) for s in range(S)) >= MinHours[n])

Max Hours Constraint
for n in range(N):
 model.Add(sum(nurseDayShift[n,d,s] * shiftHours[s] for d in range(D) for s in range(S)) <= MaxHours[n])

```

```

 model.Add(sum(nurseDayShift[n,d,s] * shiftHours[s] for d in range(D) for s in range(S)) <= MaxHours

Rest Day Constraint
for n in range(N):
 for d in range(D):
 for s in range(S):
 model.Add(nurseDayShift[n,d,s] <= 1- restDays[n,d])

MaxWorkingDays = 5 per week
NumWeeks = D // 7
for n in range(N):
 for w in range(NumWeeks):
 model.Add(sum(nurseDayShift[n,w*7 + d,s] for d in range(7) for s in range(S)) <= MaxWorkingDays

No Night-Morning Shift pattern Constraint
If a nurseDaySlot[n,d,ND] = 1, then nurseDaySlot[n,d+1,AM] = 0
for n in range(N):
 for d in range(D-1):
 model.Add(nurseDaySlot[n,d,ND] <= 1 - nurseDaySlot[n,d+1,AM])

New Constraints

Objective
demand_mismatch = sum(surplus[d,t] + shortfall[d,t] for d in range(D) for t in range(T))
model.Minimize(demand_mismatch)

Run Solver
solver = cp_model.CpSolver()
solver.parameters.linearization_level = 0
solver.parameters.max_time_in_seconds = 60 * 2
solver.parameters.optimize_with_core = True
solver.parameters.num_search_workers = 16
solver.parameters.log_search_progress = True
status = solver.Solve(model)
'''

```