

Learning Objectives

- Use CSP to optimize resource usage for job-shop scheduling
- Use CSP to implement a scheduler for the TerraBot
- Implement a dynamic scheduler that is based on agent observations

This is a group assignment consisting of three programming parts, plus a presentation. You should decide with your teammates how to divide the work on the first three parts, but everyone **is to work together** on Part 4 (the presentation).

Download and unzip the assignment code from Canvas. Place the folder in `~/Desktop/TerraBot/agents/` and link (or copy) the files `layers.py`, `behavior.py` and `ros.hardware.py` from your `ROS_HW` directory, `greenhouse_behaviors.py` and `ping_behaviors.py` from your `FSM_HW` directory and `camera_behavior.py` from your `CV_HW` directory. **Copy** your most recent versions of `greenhouse_agents.py` and `greenhouse_behaviors.py`, as you will be needing to edit those files. You will also need a library called `dill`, which is built on top of `pickle`. You can install it using `pip install dill`.

In this assignment, you will make extensive use of `ortools`. Some of the common functions you may want to use to add constraints include:

- `Add()`: used to indicate (in)equality between variables (or constants), multiplying/dividing by constants, summing variables in a list (remember that you can sum binary variables to get counts).

You can add `.OnlyEnforceIf()` to the end of an `Add` constraint to indicate that the constraint should be applied only if the argument to `OnlyEnforceIf` is `True` (or 1). Note that the argument **must be a simple Boolean variable**, for instance, it cannot be a sum or a relation. If you need to, you can get that behavior by creating an intermediate Boolean variable and setting it equal to whatever formula you need to be true.

- `AddMultiplicationEquality()`: used to constrain one variable equal to the product of other variables (`AddDivisionEquality` works similarly).
- `AddMaxEquality()`: constrain one variable to the maximum of a list of variables (`MinEquality` works similarly).
- `AddNoOverlap()`: constrain a set of **interval variables** not to overlap in time (that is, for each interval in the list, either the start of an interval is \geq the ends of all the other intervals, or the end of that interval is \leq the starts of the other intervals).

When using these constraints with `OptionalIntervals`, `ortools` does the “right” thing by enforcing these constraints only for intervals that are “present” (i.e., those that are scheduled to occur).

- `AddReservoirConstraintWithActive()`: Used to model resource usage. The function takes a list of time variables, at which some change to the resource might occur; a list of “demands”, which are the changes to the resources; a list of “actives”, which are Boolean variables that indicate whether the change actually is scheduled to take place, and a min/max range of the resource.

The idea is that for each time, if the associated active variable (same position in the list) is `True/1`, then the associated demand is added to/subtracted from the reservoir (depending on whether the demand is positive or negative). The `AddReservoirConstraintWithActive` constraint ensures that the value of the reservoir never exceeds the range limits (i.e., is never less than the minimum and never greater than the maximum). When you set up the reservoir, you don’t need to worry about the order of the lists (e.g., which time comes first) - the constraint will take care of that automatically for you. See the November 14 lecture notes for more details.

Refer to https://developers.google.com/optimization/reference/python/sat/python/cp_model for a complete list of the functions you can use to add constraints.

Part 1: Resource Utilization for Automated Factories (25 points)

In this part, you will be using `ortools` to schedule factory jobs while trying to maximize profit. A factory job produces an object (or objects) with a certain value, and one maximizes profit by using various resources in optimal ways to produce high-value objects. Specifically, a job consists of a list of tasks that must be completed in the order given. A task can be carried out using a machine and, for most tasks, there are options about which machine to use (note that some machines can be used for different tasks). The decision about which machine to use for a task is based on the availability of the machine, the resources that it uses, the time it takes the machine to complete the task, and the value of the product produced (which relates to how much money the object can be sold for). Typically, there is an inverse relationship between time to complete the task and value – the longer it takes to complete the task, the higher the quality, and the higher the value of the object produced.

A task typically requires a set of tools and a set of parts to complete the task. Tools are in use during the task, but then get returned to the pool; Parts are used up by the task, but there may be jobs that can produce additional parts. There is also a **deadline** for completing the various tasks (e.g., an 8-hour shift at the factory). It is likely that not all jobs will be able to be completed within the allotted time – part of the scheduling problem is to determine which jobs to schedule and which to leave out.

We have defined a number of classes to aid in this assignment:

- **Machine:** represents a factory machine that can be used to make things. A **Machine** object has a **name** and its **energy_cost** per hour (some machines are more efficient than others).
- **Tool:** represents a pool of tools that can be used by tasks, such as some number of wrenches. A **Tool** object has a **name** and the number of tools (**num**) available in the pool.
- **Part:** represents a pool of parts that are needed to produce objects. A **Part** object has a **name**, the **quantity** of parts initially available in the pool, and the **cost** of using one of the parts.
- **Task:** represents part of the sequence needed by a job to make objects. A **Task** object has a **name**, a list of **tools** that it requires, and a list of **parts** it requires. Each tool in the list needs to be available throughout the duration of the task. Each part in the list needs to be available at the start of the task and used up by the task. If a task needs multiple tools or parts of the same type, they are repeated in the list (e.g. ['wrench', 'hammer', 'wrench'] indicates the need for two wrenches and a hammer).
- **PartsTask:** represents a task that can create new parts, which can then be used by other tasks that need that type of part. It is a subtype of **Task**. In addition to the **name**, **tools** and **parts** required for a **Task**, a **PartsTask** object specifies which **part** is produced and the **quantity** that is produced at one time.
- **TaskMachine:** represents the fact that a machine can be used to complete some task. A **TaskMachine** specifies the **Task** and **Machine** objects, the **duration** the machine takes to complete the task (given by an integral number of hours), and the **value** of the product produced.
- **Job:** Represents the tasks needed to create some object. A **Job** specifies the **name** of the job and a list of **Tasks**, all of which need to be done, in order, to produce the object. Note that the value of a job is equal to the sum of the values for all the tasks that comprise that job.

For example, one might have job J1 consisting of tasks T1 and T2. T1 needs tool Tool1 and parts Part1 and Part2, and can be done by either machine M1, taking one hour and producing a value of 200, or by M2, taking 2 hours and producing a value of 300. Task T2 needs no tools and two instances of part Part3 and can be done by either machine M2, taking 3 hours and producing a value of 500, or by M3, taking 2 hours and producing a value of 250. Thus, there are 4 choices for completing J1: 1) using M1 and M2, taking 4 hours with a value of 700; 2) using M1 and M3, taking 3 hours with a value of 450; 3) using M2 followed by M2 again, taking 5 hours with a value of 800; and 4) using M2 and M3, taking 4 hours with a value of 550. Which is best depends on what other jobs need to be scheduled and how much energy each machine uses (since electricity usage reduces profit).

We have also specified sets of variables that will likely be useful in what follows (see the function `create_job_task_variables`). Specifically:

- **starts**: is a dictionary whose key is a list of job, task and machine (use the function `self.key(Job, Task, Machine)` for consistency). The values are the integer ortool variables representing the start times for doing the task on the particular machine for that job.
- **ends**: is a dictionary whose key is a list of job, task and machine, and whose values are the integer variables representing the end times for doing the task on the particular machine for that job.
- **scheduleds**: is a dictionary whose key is a list of job, task and machine, and whose values are Boolean ortool variables indicating whether that machine was actually scheduled to do the task for that job.
- **intervals**: is a dictionary whose key is a list of job, task and machine, and whose values are `OptionalInterval` variables that encode the start, end, and duration of a task being performed by the machine. All intervals are constrained to start at, or after, time 1 and end no later than the deadline.

The following steps create your constraint model one step at a time. At each step, you can test your model using `python autograder.py -p 1 -s <n>`, where `n` is the step number. All constraints up to, and including, that step will be added to test your solution. You can also use the `-v` flag to generate more verbose output and the `-g` flag to visualize the schedule (although for some of the steps, where the constraints are not fully developed, visualization may not work correctly).

Step 1: Task Constraints (2 points)

The first step is to implement `create_task_constraints` to encode the constraint that, for each job, each task must be achieved by only one machine. Recall that the definition of tasks include a list of `TaskMachine` objects that indicate what machines can be used to achieve that task.

Test using `python autograder.py -p 1 -s 1`.

Step 2: Machine Constraints (2 points)

The next step is to implement `create_machine_constraints` to encode the constraint that each machine can handle only one task at a time. The implementation will be a bit different from the class scheduling exercise you did in class since here we are using the interval model, rather than the binary model, but the lecture notes should help.

Test using `python autograder.py -p 1 -s 2`, which will also include the task constraints from Step 1.

Step 3: Task Ordering Constraints (3 points)

This constraint is a bit more involved. You need to implement `create_task_ordering_constraints` to encode that all the tasks of a given job are done in sequence – for instance, whichever machine is used for the first task in a job, it must end before the machine chosen for the second task in that job begins. Recall that the `scheduleds` variables indicate whether a machine has actually been scheduled to achieve a task for a job.

Test using `python autograder.py -p 1 -s 3`

Step 4: Task Completion Constraints (3 points)

This constraint is also a bit more involved. You need to implement `create_task_completion_constraints` to encode that if any job is started it must be finished by the deadline. That is, either all tasks in a job are scheduled (more accurately, if any machines are scheduled to achieve the tasks), or none are.

Test using `python autograder.py -p 1 -s 4`

Step 5: Tool Constraints (5 points)

Now we get to resource usage. For this step, implement `create_tool_constraints` to encode how tools are used by the factory. That is, at the start of a scheduled task the tool is removed from the pool and at the end it is returned. You need to use `ortools` to ensure that the number of tools in concurrent use is never greater than the pool size. See the lecture notes from November 14 if you need help on this part.

Test using `python autograder.py -p 1 -s 5`

Step 6: Part Constraints (5 points)

Continuing with resource usage, implement `create_part_constraints` to encode how parts are used by the factory. That is, if a scheduled task needs a part, it is removed from the pool of parts at the **start** of the task (and is not returned at the end – it is consumed). In addition, if a scheduled `PartsTask` task creates a part, the quantity of that part produced is added to the pool at the **end** of the `PartsTask` task (You can determine if a task is a `PartsTask` with the function `task.isPartsTask()`). You need to use `ortools` to ensure that the number of parts in never falls below zero. **As a extra constraint, assume that the number of parts in the pool can never be greater than the number at the start** – this encodes that there is some storage limit for parts. See the lecture notes from November 14 if you need help on this part.

Test using `python autograder.py -p 1 -s 6`

Step 7: Maximizing Profit (5 points)

Up until now, the optimization criterion was to maximize total value (implemented for you in `add_values`). For this step, you will maximize net profit, which is defined as the **value** of the products minus the **cost** of producing them. There are two costs that need to be taken into account:

- Each machine uses energy to produce the parts (some machines are more efficient than others). One part of the costs is the total cost of the energy used by the machine over the duration that it is working (note that the `energy_cost` attribute of a machine is the energy usage **per hour**).
- Tasks may need to use parts in the manufacturing process (the `Task` object specifies the list of parts it needs to use). Those parts have a cost, specified by `part.cost` that must be accounted for in the cost function.

Fill in `add_costs` function to set the variable `self.cost` to the total cost of production (the sum of the two costs above). The function `add_optimization` will then define an objective function that maximizes net profit (you don't need to alter `add_optimization`).

Test using `python brader.py -p 1 -s 7`.

Step 8: Testing Everything Together (0 points)

When you are all done, run `python autograder.py -p 1`, and it will run all the tests for this part. In particular, it will test whether constraints you added in later steps somehow messed up earlier solutions. It will also test a larger range of problems – some of which will take a significant amount of time to run.

Part 2: Creating a Greenhouse Scheduler (25 points total)

For this part, you will modify `greenhouse_binary.py` in the areas delineated by `BEGIN/END STUDENT CODE`.

For this part we will be creating a program that will schedule each of your behaviors. These 4 steps will guide you through a way to write the scheduler but you are free to delete all our code in `greenhouse_binary.py` and create your own if you wish. We have implemented a visualization function for you that creates your schedules into a graph as well as the initialization function and your constraint variables. `self.all_jobs[behavior, time]` is a Boolean constraint of whether a behavior is on at the time block `time`. `self.minutes_per_chunk` represent how many minutes is in a block of time and `self.horizon` represents how many chunks of time there are in

a day. `self.print` simply is to tell your code to print out the schedule it generated and `self.name` is the name of the file you will be saving to. Finally, `self.behaviors.info` is a dictionary of all the behaviors. For each behavior, it stores a three tuple of:

0. the least cumulative amount of time in minutes that the behavior needs to be run
1. the spacing between the behaviors in minutes (min, max) such that there is at a maximum 1 instance of the behavior within any min time block and at least 1 behavior in any max time block. Note that a block is a sliding window. This is only used for some behaviors ["LowerTemp", "LowerHumid", "LowerSMoist", "Camera", "RaiseTemp", "RaiseSMoist"] and not for any other
2. the maximum amount of time the behavior should run at night between [20,24) U [0,8)

The auto-grader has 5 test cases, the first two are sanity-checks to help you test your code whereas the final 3 are for testing the correctness of your code. The first test checks that given a valid set of requirements a schedule is created. The second test, run only part 4 when all schedules are active, checks that given an impossible set of requirements, no schedule is outputted. The third test checks the correctness of the output of the first test and the fourth and fifth test check more possible set of constraints for their correctness. Note that this means if you use -g to visualize the schedules, only 3 will be shown.

Step 1: Implementing Duration Constraints (5 points)

For this part you will implement the duration constraint of each behavior. This should be done in the function `createDurationConstraints`

For this, you should loop through each of the behaviors in `behavior_info` and make sure that the sum of all the times of that behavior run exceeds the time required by `behavior_info`. Make sure to convert the absolute time that `behavior_info` gives the minimum time to run in into the number of time blocks that it needs to be on by.

You can test this with `python autograder.py -p 2 -s 1`

Step 2: Mutually Exclusive Constraints (5 points)

For this part you will implement the mutually exclusive constraint of each behavior. This should be done in the function `createMutuallyExclusiveConstraints`

For this, you should loop through each time from 0 till time horizon and make sure that non of the mutually exclusive behaviors are on at the same time. A behavior is mutually exclusive with another behavior if it shares the same actuator, if one behavior wants the actuator on and the other off, or if it is the Raise and Lower for the same sensor. A full list is provided as a comment in the function. Hint: A simple way to do this is to simply include all the pairwise mutually exclusive constraints.

You can test everything so far with `python autograder.py -p 2 -s 2`

Step 3: Night Constraints (5 points)

For this part you will implement the night constraint of each behavior. This should be done in the function `createNightConstraints`

For this, you should loop through each of the behaviors in `behavior_info` and make sure that the sum of all the times of that behavior run during the night is less than or equal to the time allowed for that behavior by `behavior_info`. Make sure to convert the absolute time that `behavior_info` gives the maximum time to run in the night into the number of time blocks that it can be on

You can test everything so far with `python autograder.py -p 2 -s 3`

Step 4: Spacing Constraints (10 points)

For this final step you will implement the spacing constraint of each behavior. This should be done in the function `createSpacingConstraints`

For this, you should loop through each of the behaviors in

`["LowerTemp","LowerHumid","LowerSMoist","Camera","RaiseTemp","RaiseSMoist"]` and make sure that only up to one block of the behavior is scheduled the minimum time window and at least one instance of the behavior is scheduled in the maximum time block is on. We recommend doing so by implementing a sliding window of size minimum time block and maximum time block to check. Furthermore, in this part, the camera behavior should only be run during the day so only create time blocks for the camera during the day (not 20-24 and 0-8) (We don't want to space any camera photos out during the night). Make sure to convert the absolute time that `behavior_info` gives the minimum time to run in into the number of time blocks.

You can test everything with `python autograder.py -p 2 -s 4`

Part 3: Dynamic Scheduler (30 points)

For this part, your team needs to come up with an approach to adapt the schedule to current conditions. This can take any number of forms – we offer two suggested approaches below. If you want to take it in another direction, please talk to us about your plan before Monday November 21, so that we can approve of what you intend to do.

Ideas:

Plants have different needs depending on their stage of growth, especially how much light and moisture they need/can tolerate. Do some research to determine what radishes and lettuce need when they first germinate, start growing, and get to the point where they are starting to mature (after about two weeks). Use vision to determine what stage your plants are at and have the schedule change the frequency that the lighting and watering behaviors are enabled. Also use the insolation prediction that you developed in Assignment 6 to monitor and maintain the appropriate amount of light during the day. Based on these values, you can adjust the next day's schedule such that the daily light and moisture average out appropriately.

You will notice that some behaviors rarely need to be enabled (e.g., lowering temperature rarely has to run, especially in the evening) and some behaviors may need to be run more frequently. Implement monitors to determine when behaviors are actually needed (e.g., they turn actuators on) and when they should be needed (e.g., some sensor limit is reached when the associated behavior is not running). Use this data, say averaged over a day, to change the scheduled frequency of behaviors. You don't have to do this for all the behaviors – pick two that you think would most benefit from this rescheduling.

In all cases, you will need to:

Create a Monitor to observe and compute the values you need. Implement functions as necessary in the `greenhouse_binary_ref` or `greenhouse_binary` so that as information is updated, it can be used by the scheduler. Call `Greenhouse_Binary` to create and output a schedule based on modified constraints based on given conditions. You may find that you have to change the parameters in the example "main" schedule for some behaviors to allow you to optimize the behaviors you care about in this assignment. Call the Planner functions `setTestingSchedule` and `switch_to_test_sched` to set and load a testing file and run from that (rather than having it just read the txt file).

For this part, we have provided a reference solution of Part 2 for your usage but you may also use your own Part 2 solution as well. The reference solution file is `greenhouse_binary_ref` and you can use the constructor and `solve_solution`. You may also wish to read the initial part of Part 2 to see how to initialize your variables for the scheduler.

Part 4: Presentation (20 points)

For this part you will **work as a group** to create a presentation to present to the class the dynamic scheduler you have created and why you have chosen to do so and what benefits your scheduler has.

Run your agent in simulation using the standard `greenhouse_schedule.txt` for 2 weeks. Then run your agent using your dynamic schedules.

Slide 1 and 2: Describe your goal in dynamic scheduling (idea 1, 2, or something else), and your implementation of it. What did your monitor do? What are you trying to dynamically optimize? How did you change the `Greenhouse_Binary` scheduler to capture this goal?

Slide 3 and 4: Capture 2-3 schedules and visualize them using the function `plot_binary.py` in `visualize_solution.py` (look at `autograder.py` for an example of how to use that function). Identify the constraints that made them different from main, and visually how are they different from main?

Slides 5 and 6: Compare and contrast the plant growth based on these two techniques. Capture your estimated plant health or other quantities important for your scheduling. If possible, capture some visualizations of your plants in both cases to compare as well.

Submission

Submit the following files to Canvas: `greenhouse_binary.py`, `job_scheduling.py`, `README`, and any files that you created/edited for Part 3.

When added to a directory with the rest of the agent files, your code should run without error.