FIT2107 Assignment 2: Whitebox Testing, Unit Testing & Continuous Integration

The *Joules Up* EV Charging Calculator

Group members	
Name	**Student ID**
Ng Yu Kang	30897343
Pan Wei Hung	30883644
Lin Chen Xiang	30897300

Work Breakdown Agreement (WBA)			
Member Name	**Approx hours spent (per week)**	**Test Cases**	**Other contribution**
Ng Yu Kang	10 hours	- Requirement 1 whitebox testing cases - Solar API whitebox test cases	- Planning - Coordinating group - Implementation of req1 - Implementation of retrieving data from solar API
Pan Wei Hung	10 hours	Requirement 3 whitebox testing cases	- Helping in researching related information - Implementation of req 3 - Help in finding issues with the code
Lin Chen Xiang	7 hours	Requirement 2 whitebox testing cases Helper function test cases	- Gathered and formatted school holiday dates for app usage - Implementation of req 2

Testing Strategy

Requirement 1: Assignment 1 functionality: charging cost and time calculation

Whitebox testing

Majority of the functions implemented for time and cost calculation functions are simple functions that either have no if statements or have simple branches, so for these functions, we opted to use line coverage for those that don't have any branches and branch coverage or path coverage using cfg for those that have branches to test these functions.

The time calculation function does not have any branches, so this function can simply be tested with line coverage technique. The cost calculation functions were split into two: one that only calculates a specific period of time, and one that calculates the entire charging session.

The cost calculation for a specific period of time is used in the total session cost calculation. This is since a charging session may span different days and hours, we would need to calculate each period differently. For this function, there are only some simple branches, so we also use branch coverage here.

The total session cost calculator works by checking each one hour period or from the current time to the next hour (like 16:37 to 17:00), and checks that period if it’s in peak hour and if it’s in the holidays/weekdays or non-holiday weekend. With those in mind, it is accumulated and later sent to the cost calculation for a specific period of time to calculate different conditions' cost. In this function, there is a while loop and there are branches in the while loop. In this function, we will be testing the if statement inside the white loop using path coverage.

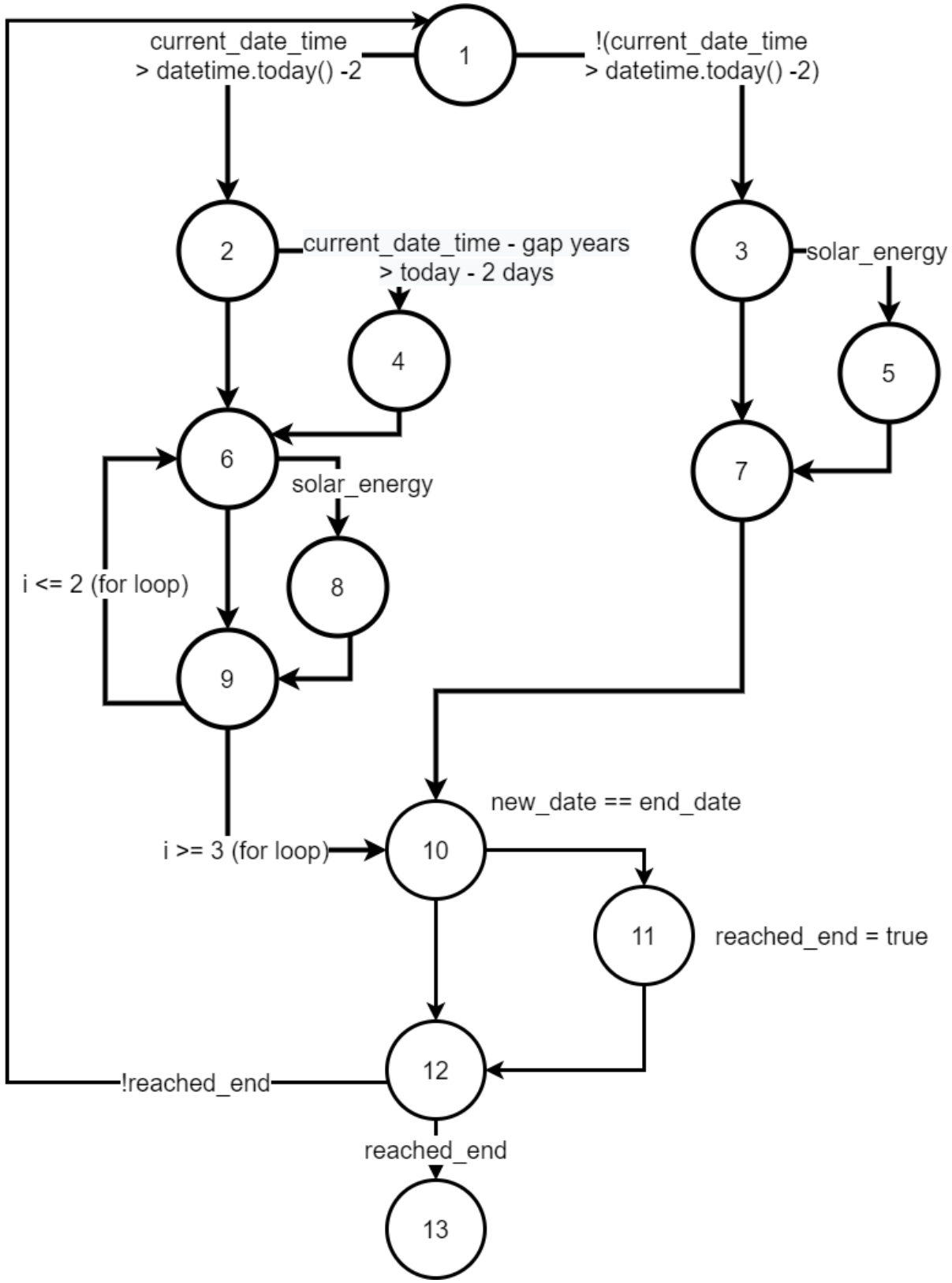
In the functions, we would have if statements to raise ValueError if we were given invalid inputs. Those will be tested with simple line coverage to make sure that the error is raised. The explanation below won’t be considering invalid inputs, but for **all functions, we would run line coverage tests using invalid input** and check if it raises appropriate errors.

We have made an assumption that regardless of the number of locations returned by the solar API, we would choose the first location to simplify implementation. Letting the user choose would have involved wasting too much time on learning how to use Flask to do so, making it too much implementation focused than testing.

Whitebox testing for total_cost_calculation

In requirement 1, the total_cost_calculation for requirement 1 is for calculation, in past and future date without considering solar energy, so in this section we would only test these.

Using path coverage via CFG, we will find out these test cases for past and future calculation without solar.



Here, we would consider:

- 1,2,6,9,10
- 1,2,4,6,9,10
- 1,3,7,10

With a test case that loops at least more than once (at least multiple hours), we can make sure that the nodes 10,11,12 are fully covered, and thus fully test requirement 1 functionality of calculating total_cost without solar energy considered.
With that in mind, this is the 4 test cases derived from the cfg. The table below will only show the input variable that changes. Other variables that doesn't change between test cases will not be shown in the table and are static, which are:

- start_state : 20
- postcode: 4000
- solar_energy: 0
- end_time (obtained by another function in which we will test later)

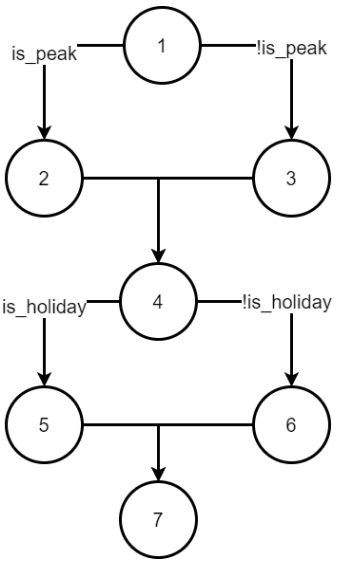
Do note that power,capacity and base price in the table does not affect the decision, but is just here so that we can reuse test cases derived before in blackbox testing, and that in the actual unittest, power and base price will be obtained via get_configuration method too, which we will also test later.
We would mock is_holiday in the cases using date_today() to ensure that the day specified is not holiday in test case 3 and yes holiday in test case 2, so we can make sure that the expected cost is correct.

Test case	start_date (date obj)	start_time (time obj)	base_price (float)	power (float)	capacity (float)	Notes
1	date(2021,8,24)	time(8)	5	2	50	BTC1
2	date_today+1year	time(0)	12.5	11	50	BTC8
3	date_today()	time(8)	10	7.2	50	BTC5

(BTC = black box test case, for self-reference)
With these test cases, we will be able to exercise 100% path coverage for this function (At least for calculation without solar energy).. We have one that loops only once (test case 2), and others loop more than once.

White box testing for cost calculation

For this function, we will also use path coverage here. There are no while loops, but to make sure we test all of the possible behaviours of this function, we need to make sure to test all combinations of inputs that cause branches. In this case, is_peak and is_holiday are the conditions that branches, so we would need 4 test cases here (2^2), or we can look below at the cfg for this below. Since it's similar to the test cases derived for the total cost calculation, we will reuse the test cases here too.



Static variables

- Initial_state: 20
- capacity: 50
- Final_state: 40

Test case	is_holiday	is_peak	base_price (float)	capacity (float)	Notes
1	true	true	5	50	BTC1
2	true	false	12.5	50	BTC8
3	false	true	10	50	BTC5
4	false	false	5	50	BTC2

This should help us achieve 100% path coverage (which is 100% line coverage) and test the expected behaviour of this function, by covering all possible branches of the code.

Whitebox testing: time_calculation

For this function, a simple line coverage test can be performed, since it does not have any branches at all, so that one valid test cases would be able to cover the entire function.

Test case:

initial_state	final_state	capacity	power
20	40	50	2

Whitebox testing: get_configuration

Same as before, it has no branches, so all it really requires is just line coverage. Using input of 1, we can achieve 100% line coverage with just one test case to go through all parts of the function.

Whitebox testing: is_holiday

In is_holiday we need to check whether the date given is a public holiday or is a weekday. This translates to two different conditions which are connected by ORs. For this, we would use MC/DC (why not), since there are 2^2 = 4 cases if we do path coverage.

For public holidays, we will assume that the holiday will be observed on the day itself, and never moved. School holidays are read from files, and if the date isn't in there, we would assume it is not a school holiday (aka we can't predict future school holidays, and would not estimate it using past holidays)

Test case	Weekday (a)	Public holiday (b)	(a || b || c)
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

Weekday cases: {2,4}
Public holiday cases: {3,4}
Combined: {2,3,4}
With these, we will be able to achieve 100% MC/DC coverage, by inputting dates and states that matches the truth value of test case 2, 3 and 4, we would be able to cover all branches of the function and be confident in its correctness.

Whitebox testing: is peak

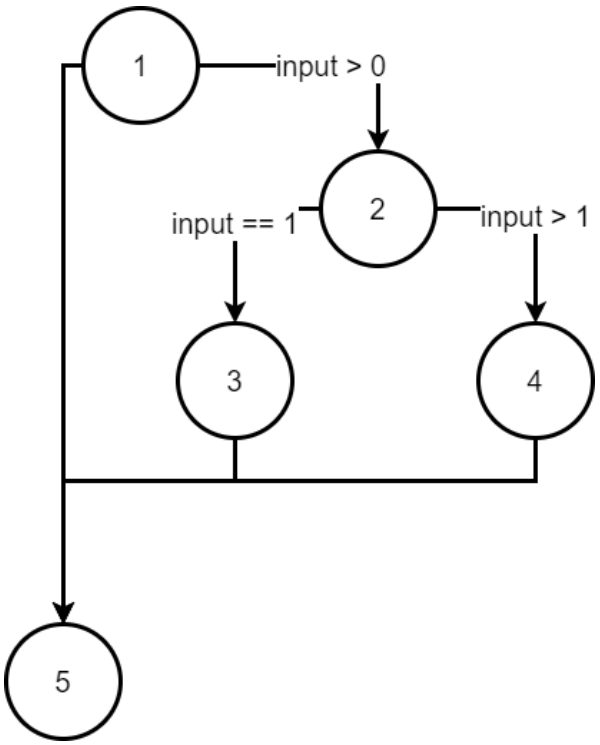
When you look at the function initially, it should be covered using line coverage. However, when you really look at it, a given input for this function has a condition inside: input_time >= 6.00am and input time <= 6.00pm. Using path coverage and having two variables in the condition, there would only be 4 in theory, but in reality only having 3 cases since a time can't both not be >= 6.00am and not <= 6.00pm.

Test	(a) Input_time >= time(6)	(b) input time <= time(18)	(a && b)
1	T	T	T
2	T	F	F
3	F	T	F

With these, we will be able to achieve 100% path coverage, and thus 100% coverage of the function, and we would be able to test out all branches of the function.

Whitebox testing: get_charing_time_str

In this function, there are three if statements inside. All three of the if statements are the same, just different formatting to add to the return string. Thus, for this function, we will use path coverage using cfg for the if statement, three times, for hours, minutes and seconds.



Note that the input here means either hours, minutes or seconds, as an input to the if statement. Recall that all three if statements in the function are similar, so we have generalised it and will use this three times. As you can see from the cfg, there are only 3 paths in total, so three cases. Taking the three cases and doing it 3 times, we have 9 test cases in total. But since there are overlaps in the test cases, in total we would only need 7 cases. (All hours, minutes and seconds would have a 0 time case, so we can merge it into one case)

Test case	charge_hours
1	0
2	1
3	2
4	(1/60) (1 minute)
5	(2/60) (2 minutes)
6	(1/60/60) (1 second)
7	(2/60/60) (2 seconds)

With these inputs, we will exercise each path of all three if statements at least once, which would give us 100% path coverage and thus line coverage.

Blackbox testing

All of the test cases that were devised using blackbox testing via combinatorial testing from assignment 1 were turned into unittest test cases. This allowed us to make sure that the function we implement meets the functional requirements.

As stated back in assignment 1, the combinatorial testing test cases were generated using jenny, and in the categories we have back then, we only considered valid inputs for the combination. Invalid inputs were tested in a separate test suite which used equivalence testing on each of the inputs.

Solar Energy API Access

All methods regarding the access of the solar API were tested with line coverage. This is because all of them are just simple functions that don't branch out due to if statements. In get_weather_data, there are if statements, but they are if statements that help catch erroneous inputs, so we simply use line coverage technique to devise cases each that goes into the error catching statements.

Majority of the test cases devised for solar energy API access have had either requests.get or get_weather_data mocked, in order to make the test run faster. In the case of get_weather_data getting mocked, for each method, we mock the required return value for those functions we are calling, and see if the functions we call retrieve and manipulate the data into the return value that are correct. For mocking requests.get, this is done so that we can quickly test responses that are produced from bad inputs.

An example test case would be the test case to test get_weather_data using line coverage. First, the requests.get is mocked to return the expected location data, then the second request is also mocked to return the expected weather data. From it, we see if the function will correctly return the data as expected and not do anything extra to it. This is one of the possible branches since there are if statements to catch errors.

Yet another example would be the test case for get_cloud_cover. We mock the return data from get_weather_data, then we test if get_cloud_cover extracts the cloud cover data from the weather data correctly, by checking the expected output against actual output. This test case should be able to fully cover the entire function since there are no branches in the function.

Whitebox testing: Calculator_Form validation methods

All of the validation methods were tested using line coverage technique. This is because we would only need to give inputs to make sure that the invalid inputs are properly caught and errors are raised. Inputs to the validation functions were mocked to have a data attribute so that it can be retrieved and checked by the functions. Postcode validation function had requests.get mocked to facilitate faster testing, mocking the response appropriately according to input we test, and see if with such response, an error is raised.

An example would be testing the capacity validation method. We check if it catches errors like data being None, empty string (“”), not numerical and negative. These errors are each a test case itself, and this allows us to exercise all branches of the battery capacity validation method, thus achieving 100% coverage for the function.

Requirement 2: Assignment 2 functionality: charging cost and time calculation with the addition of solar energy generation, with the date from 1st July 2008 up to the current date-2 only.

Whitebox Testing Main

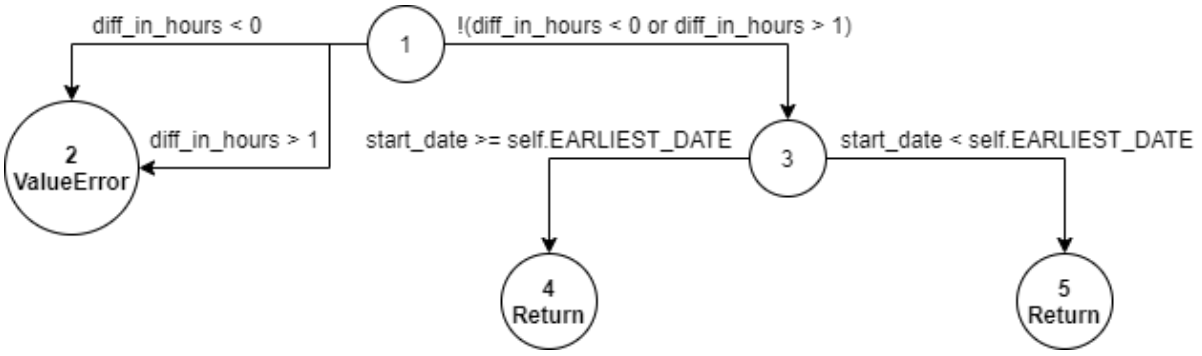
The function for requirement 2 has some conditions which are rather straightforward. Since it has paths that lead to errors and others that return different values, we decided to use **path coverage for all of requirement 2 functions,** calculate_solar_energy_past_to_currentday_minus_two. The preconditions for this function are as follows:

- Starting date must be during or after 1st July 2008
- Ending date must be 2 days before current date of input (pre-processed)
- Starting date and end date must be the same (pre-processed)
- Starting time and end time must have at most an hour interval, inclusive

Since the preconditions of start date and end date being the same is already processed from other functions, we do not have to specifically check for that. Instead, we can just find the difference between the start date and end date and check whether the difference in hours is between 0 and 1 inclusive, which is the valid interval.

After validating the start and end dates, we go through the paths which either does the full calculation if the date is during or after 1st July 2008, or 0 if the date is before 1st July 2008.

Below is the path coverage CFG for the function calculate_solar_energy_past_to_currentday_minus_two.



From the paths, we will consider:

- 1,3,4 (valid start and end, date>=01/07/2008)
- 1,3,5 (valid start and end, date<01/07/2008)
- 1,2 (diff_in_hours < 0)
- 1,2 (diff_in_hours > 1)

From these tests the postcode will be the same for consistency. Static variable postcode will be "4000". However, we will also add one extra test case which is from Example 1 of the assignment specifications, which the postcode provided is "6001". It seems that we have a duplicate test case of path 1,2. This is done as the condition to branch node 2 has multiple

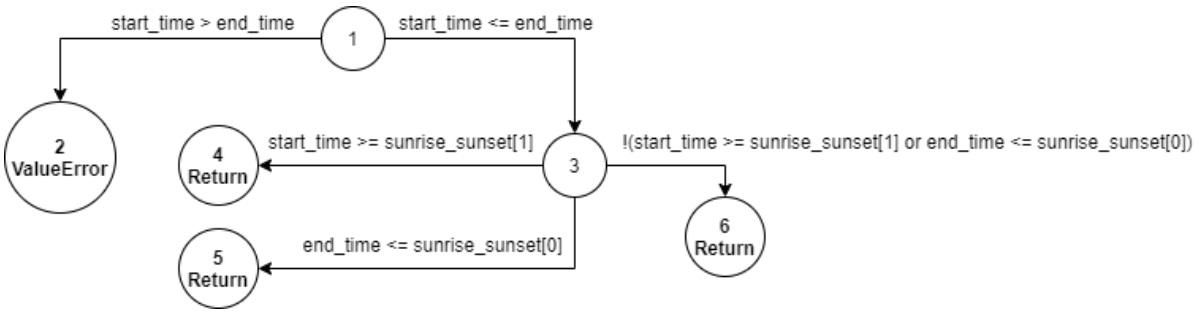
true statements, which we would want to ensure situations that reach node 2 are valid. Below is a table which shows all inputs for each test case from above.

Test Case	Start Datetime	End Datetime	Postcode
1	datetime(2008, 7, 1, 12, 30)	datetime(2008, 7, 1, 13)	"4000"
2	datetime(2008, 6, 30)	datetime(2008, 6, 30, 1)	"4000"
3	datetime(2008, 7, 2)	datetime(2008, 7, 1)	"4000"
4	datetime(2008, 7, 1, 12, 1)	datetime(2008, 7, 1, 13, 5)	"4000"
Extra from Example	datetime(2020, 12, 25, 8)	datetime(2020, 12, 25, 9)	"6001"

From the above cases we should have tested each path at least once, which will achieve 100% line coverage.

Whitebox Testing Helper Function

Additionally we also test another function get_solar_energy_duration which is only used by the requirement 2 function. This function also has its precondition, where the start time must be less than or equal to the end time. This function also has rather simple branches, which we can just make use of path coverage to cover all lines. Below is the path coverage CFG for function get_solar_energy_duration:



From these paths we will consider:

- 1, 2 (start > end)
- 1,3,4 (valid start and end, start during or after sunset)
- 1,3,5 (valid start and end, end before or during sunrise)
- 1,3,6 (valid start and end, start before sunset and end after sunrise)

These tests only take the time into account, hence we can have constant variables for date and postcode. For these tests, the constant variable are:

- Date: date(2021, 9, 10)
- Postcode: "4000"

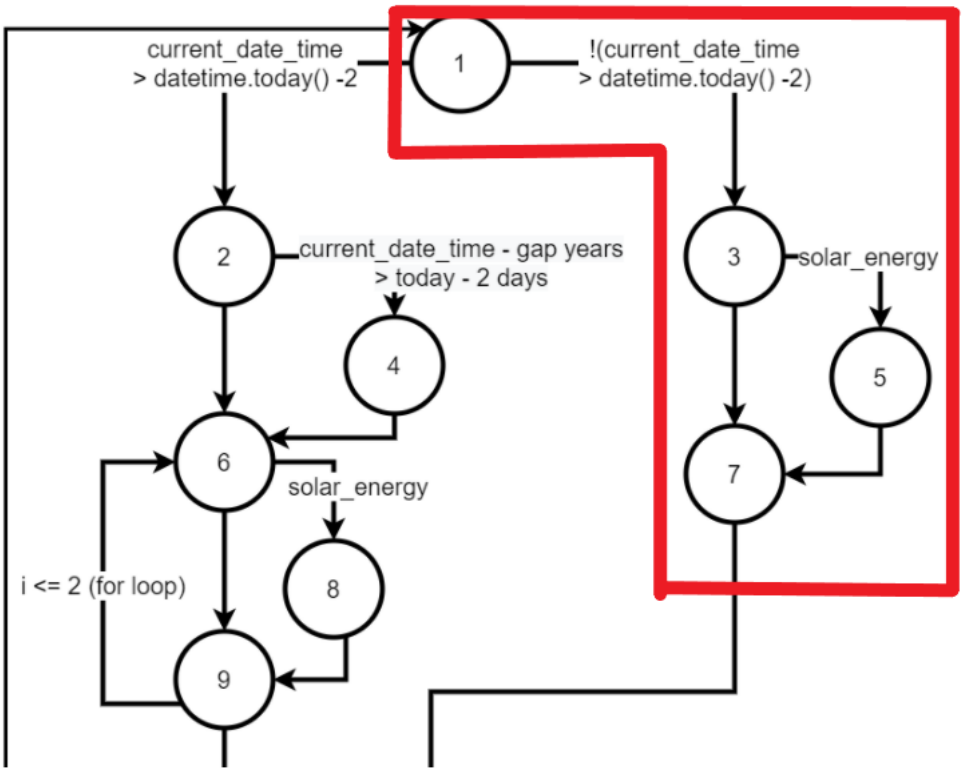
Below is a table that shows the inputs for each of the test cases above, excluding date and postcode.

Test Case	Start time	End time
1	time(13)	time(12)
2	time(22)	time(23)
3	time(3)	time(4)
4	time(12, 30)	time(13)

From the above cases we should have gone through all possible paths in this function at least once, hence achieving 100% line coverage.

Whitebox Testing Total Cost Calculation

Lastly, we make use of this function into the main cost calculation function, total_cost_calculation. Since we are only testing the part where this function makes use of the function calculate_solar_energy_past_to_currentday_minus_two (req 2 function), we will only cover the path that has this function in it. Below is the path coverage CFG from whitebox testing for total_cost_calculation, but we will only be covering the highlighted part.



From these paths we will consider:

- 1,3,5,7 (accounts for solar energy for past dates)
- 1,3,7 (no solar energy)

For these two cases all variables will be the same except for solar_energy where the first will be True and the second will be False, since we just want to make sure the cost difference

between accounting for solar energy and not is actually correct. Hence the constant variables are as follows:

- Config = 8
- Start time = time(8)
- Start date = date(2020, 12, 25)
- Battery capacity = 50
- Initial_charge = 20
- Final_charge = 80
- Postcode = "6001"

Since configuration and start time and date are the same, the end time calculated will also be the same. Below is the table for the solar_energy inputs for the above test cases.

Test Case	solar_energy
1	True
2	False

With this, we cover all possible paths of total_cost_calculation for this requirement and hence reach 100% line coverage.

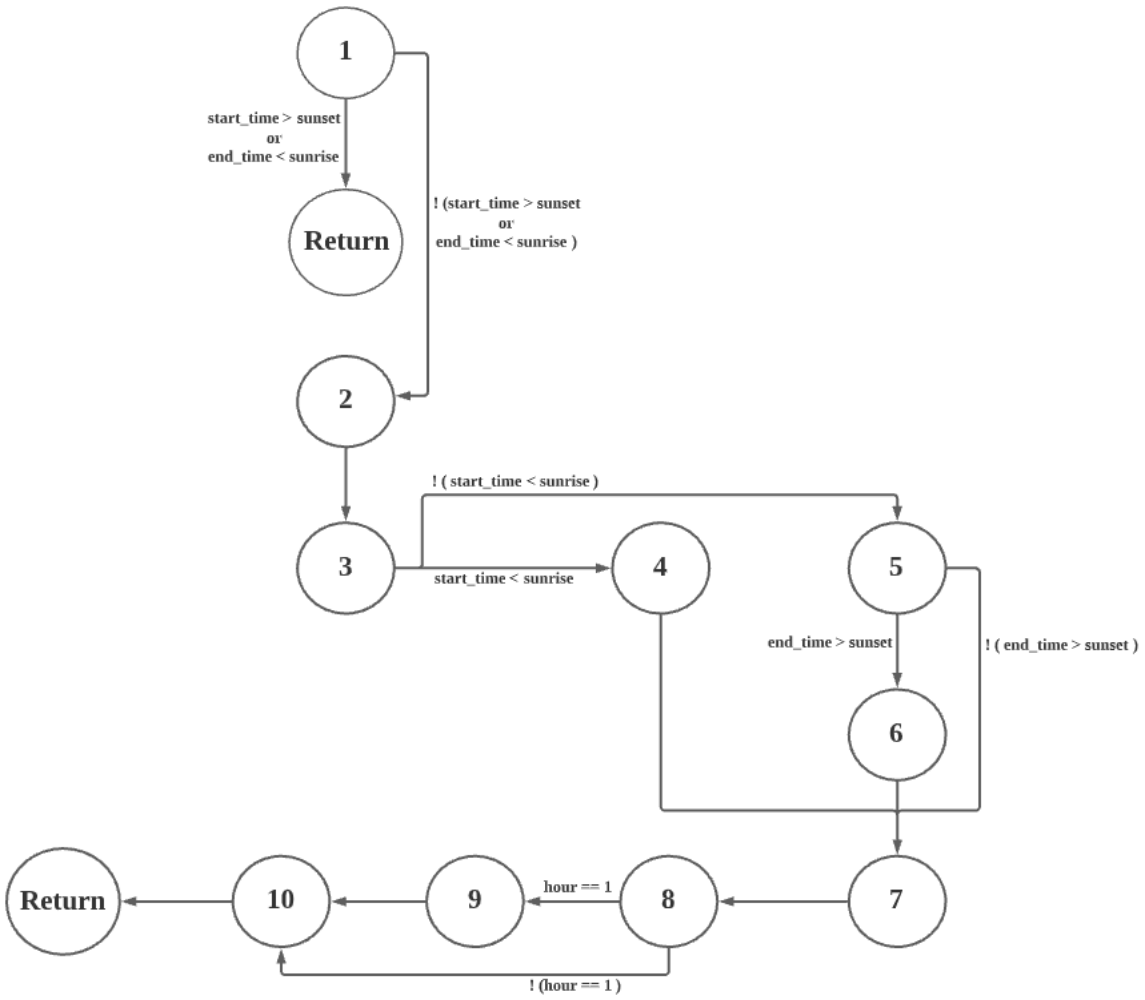
Requirement 3: Assignment 2 functionality: charging cost and time calculation with the addition of solar energy generation, with the date extending to the future and hourly weather conditions.

Whitebox testing

Function created to calculate the solar energy generation based on a few conditions. Besides, the calculation of solar energy generation was based on the pre-condition that the difference between start time and end time will either be a whole hour or partial hour provided from the total cost calculation function. Aside from this precondition, the date of start time and end time will have a precondition that it will be a reference date to the past instead of a future date. In addition, the input of start time is 15:30 and end time is 16:30 will not be accepted, the end time should have a maximum of 16:00 in this case. In conclusion, the suitable testing strategy for function calculating solar energy generation will be path coverage to cover all the possible paths of the function.

Besides, to calculate the charging cost with the addition of solar energy generation, we will reuse the total cost calculation function. Since the total cost calculation function was tested in the previous Requirement 1, we will only test the total cost calculation with the date extending to the future and there is solar energy generated to cover the path in the total cost calculation function.

A control flow graph for function calculate_solar_energy_future was created for clearer visualisation.



As the input of this function will only be a whole or partial hour, thus there will only be a few conditions to check. All possible paths for this function:

- `1-Return`
- `1-2-3-4-7-8-10-Return`
- `1-2-3-5-6-7-8-10-Return`
- `1-2-3-5-7-8-9-10-Return`

The explanation for each path are as follows:

`1-Return`

Explanation: This path is when the start time and end time is not within daylight length, thus it will return directly as there will be no solar energy generated

`1-2-3-4-7-8-10-Return`

Explanation: This path is when the start time hour is the same as sunrise hour but the start time minute is smaller than the sunrise minute, it will update the value of start time. Then, path 8-9-10 will not be possible as path 8-9-10 indicates the start time to end time is a whole hour,

which won’t happen as if the start time minute is smaller than sunrise minute it won’t be a whole hour.

`1-2-3-5-6-7-8-10-Return`
Explanation: This path is when the end time hour is the same as sunset hour but the end time minute is larger than the sunset minute, it will update the value of end time. Then, path 8-9-10 will not be possible as path 8-9-10 indicates the start time to end time is a whole hour, which won’t happen as if the end time minute is smaller than sunset minute it won’t be a whole hour.

`1-2-3-5-7-8-9-10-Return`
Explanation: This path is when the start time and end time has a whole hour gap which is 60 minutes. The value of du will be updated to 1.

For test cases of total cost calculation, we have covered all possible paths of total_cost_calculation. Generated test cases are as follow:
1)
Config set to 3. Start at 2022-2-22,17:30 to 2022-2-22, 18:15, with the postcode set to 7250, solar_energy set to True and the expected output will be $0.22. This test case was generated based on the assignment specification.

2)
Config set to 3. Start at 2022-2-22,17:30 to 2022-2-22, 18:15, with the postcode set to 7250, solar_energy set to False and the expected output will be $0.48. This test case was generated to cover the branch when it extends to a future date and solar energy generation was not taken into account, by default, this is not possible based on requirement 3, but leaving this path enable more flexibility for future extension.

3)
Config set to 3. Start at 2021-12-12,17:30 to 2021-12-12, 18:15, with the postcode set to 7250, solar_energy set to True and the expected output will be $0.16. This test case was generated manually to test out the reference date selection of the function to be 2020 instead of 2021 where 2021-12-12 was actually a future date. By testing this function, we can know that the reference date calculation was working perfectly even when the year of the future date was same as the current year.