OVERVIEW

This practical involved building a simulator and build order optimiser for Starcraft II: WoL. The goal of the program is to reach a given game state for the Terran race as fast as possible and find optimal build orders. The presented problem involves various concepts: modelling the game, processing the goal, making decisions, and updating the game state. The program sets the goal it needs to reach by accepting one command-line argument (a String describing the goal), generates an optimised build order to accomplish that goal, and then prints the build order to terminal. If the command-line arguments are invalid, an informative error message is printed to terminal and the program terminates without generating a build order.

For each goal presented in the specification, our program produced the following time ranges for the basic deliverable:

- 6 Marines: 3:43 - 4:06
- 16 Marines: 4:59 - 5:33
- 50 Marines: 7:52 - 8:40
- 10 Hellions: 6:17 - 7:13
- 6 Marines, 4 Hellions: 5:55 - 6:55
- 8 Marines, 2 Medivacs: 6:19 - 8:31
- 8 Marines, 2 Medivacs, 2 Vikings: 7:21 - 9.06
- 16 Marines, 8 Hellions, 3 Medivacs: 8.03 - 10.30

Extensions:

In addition to the basic requirement, a number of extensions were implemented. The extension in the “delay” folder implements a solution for a more realistic timing for a player. The extension in the “intermediate” folder implements the intermediate requirement for the advanced goals outlined in the specification, also implementing supply.

For each goal presented in the advanced goals specification, our program produced the following time ranges:

- 2 Marines, 8 Hellions, 10 Siege Tanks, 2 Thors: 14:30-16:30
- 16 Marines, 8 Marauders: 7:55 - 9:22
- 16 Marines, 8 Marauders, 4 Medivacs: 9:40-12:40
- 2 Marines, 3 Hellions, 8 Tanks, 8 Vikings: 14:25 -16.14
- 16 Marines, 4 Banshees, 4 Vikings: 11:05 - 12.47

Besides the extensions already mentioned, our program implements a sophisticated search strategy that uses a number of heuristics to optimise the decision making process, rather than making decisions randomly.

DESIGN & IMPLEMENTATION

General approach

When designing and writing our program's code, the two principles we had in mind were readability and efficiency. Since modeling the program involved many different classes and changing a large number of variables when tracking the game state, keeping the code readable was a priority so that we could both understand the code we were writing and it was much more straightforward to manipulate variables along the way.

For readability, we used lambda expressions and regular expressions on multiple occasions, because it keeps the code cleaner. The code is also thoroughly commented to explain what each attribute is storing and what part of the functionality each method is performing, because the complexity of the program makes it harder to understand what action is performed without additional comments. To further improve readability, the program is refactored as much as possible and duplicate code is avoided.

To ensure the most efficient build orders, we adopted a number of heuristics to optimise the decision making process based on the requirements of the goal and the current game state. This was achieved by prioritising building valuable units, buildings, and also by limiting the building of constructions relevant only to resource gathering. This approach allowed us create a program that covers all important aspects for generating the most efficient build orders. Some of these aspects included ensuring that workers were assigned to mineral patches immediately after the game begun, ensuring that all workers were busy collecting resources at all times, and that all dependencies were taken into account before units were built.

Modelling the classes

To model the game, we created a class for every unit or building that could be built for the Terran race (for the basic requirements). Each unit class extends an abstract Unit class and each building class extends an abstract Building class. In addition, both the Unit class and Building class extend an abstract Construction class. We classified anything that could be built in the game as a construction. These classes are abstract to make it clear that they are not to be instantiated and to model the game better, make the relationships between elements of the game more clear.

Within each unit and building class, attributes are created to store their mineral cost, gas cost, build time, and dependencies. The Unit class had an additional attribute which stored the name of the building it had to be built from. All of these attributes were made static as they were the same for all instances of these classes made. Each class had a final String attribute called IDENT, which was used to as key values in HashMaps such as units and buildings inside the GameState class. Initially, we were storing the key values as instances of the unit or building in these HashMaps and ended up writing lines of code that were too lengthy and not easy to read. We also realised that it was unnecessary to store the key values as instances of these classes as we only needed to identify the construction type we were tracking and its quantity to simulate the game. Another approach we attempted was to use the Java Reflection API and use the Class names as the key values in the

HashMaps, but some operations required instances of the constructions, and the code got very complicated and cluttered. In the end, we decided that using identifiers to store information about the various constructions was the best approach for the design we created, allowing us to keep the code more efficient and readable.

For the Worker class, there are three additional final attributes called FREE, GAS and MINERALS which help to identify the three different states that workers can have in game. The attribute FREE identifies workers that are not assigned while MINERAL and GAS identify workers that are assigned to mineral patches and geysers respectively. These additional attributes were added so that the actions performed by the workers could be tracked after the game was initialised. A separate HashMap called workers was created with FREE, GAS and MINERALS stored as keys to represent workers of each type along with the number of workers of each type as the corresponding values.

In each unit class, there is a final attribute called INDEX which provides information on the decisions that the program is required to make based on the type of units that are required to achieve the specified goal. The indices stored in the INDEX attribute are used in the Decision class to ensure that decisions made are relevant to the goal, so as to avoid making random decisions that could be wasting time and resources. To keep track of the decisions made in the simulation, the Decision class also contains an ArrayList that stores all decisions made in the simulation, whichas to mak is then used for printing the build order. This is stored here rather than in the GameState class because it is more relevant to the decisions than to the state of the game.

The goal that needs to be met is parsed and stored in the Goal class as a HashMap that contains the identifiers and the number of the units that need to be built. This was the most straightforward approach, also matching the collections used for storage in the GameState class, which made it easy to check whether the goal has been met.

The GameState class is modelled to simulate the game. It keeps track of all the completed units, the completed buildings, the constructions that are in the process of being built, and the workers and what they are assigned to at any given second. The buildings that are constructing a unit are separated from the buildings that are not using different HashMaps to make it easier to keep track of the available buildings. The constructions that are in the process of being built are stored in a HashMap called constructionsBeingBuilt, which stores the identifiers of the specific constructions as keys and ArrayLists that contain the number of seconds until completion for each instance being built as integer values. This made it easier to check when a construction has finished building and updates the state of the game accordingly. The GameState class also keeps track of the available resources and the time elapsed since the start of the simulation. The reason these elements are stored as attributes inside this class is that they are important aspects of the state of the game and the design of the program only requires keeping track of the current values of these elements. The number of mineral patches and gas geysers are stored in a similar manner, because we realised that we did not need information on specific workers being assigned to specific geysers or patches. The program only looks at the number of workers assigned to either, so only the number of geysers and patches is sufficient.

The main class makes use of the other classes to simulate a game of Starcraft and generate an optimised build order. It parses the goal, initialises the game, and then keeps making decisions until

the goal is met. Then the decisions made for the build order are printed to the terminal, along with the total time taken to reach the desired game state.

Setting the goal

Before the game state is initialised, the goal is first set after the program accepts a single command line argument which is a String describing the goal for the Terran race. The String is first parsed by the method parseGoal() in the Goal class where the units and number of units specified in the String is parsed into a String and Integer and placed into a HashMap<String, Integer> called goal in the GameState class. The goal HashMap is created so that the game state can be checked against it when the simulation begins running to determine when the goal has been achieved.

Concurrently, the names of units that are required for the goal are stored as a key in another HashMap in the GameState class called possibleDecisions along with its INDEX attribute as the corresponding value so that this can be later used to point the program to build only the necessary units and dependencies to achieve the goal when the method updateState() is invoked.

When parsing the goal, it is also checked that the user input is syntactically correct, that only valid units are entered (workers are not considered valid units), and that each goal or sub-goal is entered as <value> <unit>. If not, it stops the program from starting the simulation so as to avoid errors created by an invalid goal.

Updating the game state

The state of the game is updated at every second in the simulation. One of the reasons for this is to ensure that resources are collected at every second, making it possible to start building units or constructions earlier on in the game. If we update resources by the minute instead of the second, it would significantly limit the decisions that can be made and decrease efficiency. The resources are updated before a decision is made for that second, because if more resources are available, the probability of being able to make a valuable decision increases. The constructions in the process of being built are also updated before making a decision, so that if a construction relevant to the goal has been finished, it can be used in the current second. After updating the constructions, the program also checks whether the goal has been met and terminates the decision making process if so because there is no need to make other decisions. The time is updated after a decision has been made so that next decision is made at the beginning of the second instead of the end of it and also to maintain the accuracy of when a construction had completed building.

The program is designed such that multiple decisions can be made every second, such as building more than one constructions if there are enough resources and reassigning workers. This improves efficiency by allowing multiple constructions that are relevant to the goal to start being built at the same time if enough resources are available. It also ensures that all workers are assigned to a mineral patch or a gas geyser, optimising resource collection. Resource collection is also maximised when updating the resources by assuming that the workers are distributed in the most efficient way. The specification states that the first 2 workers assigned to a mineral patch have a collection rate of 41 minerals per second, but a third worker added to the same patch would have a collection rate of 20 minerals per second. Therefer, if 3 workers being assigned to mineral patches, the program updates

the resources assuming that 2 of them are collecting from one patch and 1 of them are collecting from a different patch.

The constructionsBeingBuilt HashMap is updated from the Decision class whenever a decision to build a construction is made. When a unit is being built, the freeBuildings and busyBuildings maps are also updated (availability is checked in the Decision class). The constructionsBeingBuilt map is updated at every second by creating a new ArrayList for the remaining times that only contains the remaining build times that are not going to reach 0 at the current game time. This way, it removes the completed constructions from the map. When a building is completed, it is added to the freeBuildings map. When a unit is completed, it is added to the units map, and also the freeBuildings and busyBuildings maps are updated depending on which building the specific unit is built from. Storing the constructions in maps and only looking at how many constructions we have at the current second made this process less complicated than using objects would have.

The program only stores the current game state and it does not generate other possible game states because the design of the decision making process does not require that. The decision is made based on its relevance to the goal, not by comparing different possible states. The current game state is then updated according to the decision that is made within the simulation.

Making decisions

Decisions are made based on the goal passed as a single command-line argument. The units that are relevant to the goal are already stored in the possibleDecisions map (updated when parsing the goal). The map stores the indices of the units as its values, and when attempting to make a decision, it selects a random index from the values contained in the map. The selected index corresponds to a case in the switch case where the decision is made. In this way, only the relevant case statements will be accessed. For example, if the goal is “6 Marines”, the simulation will never try to build Vikings, Medivacs, Hellions, or buildings that are not useful for the Marines (useful buildings for a unit is the building that builds it or the dependencies of that building). If the sub-goal for a specific unit has been met, the unit is removed from the possibleDecisions map so that the simulation does not attempt to build that unit again. In this way, the simulation does not waste resources by building more units than needed.

When making a decision, it first tries to build the unit corresponding to the selected index, because that is the most useful action for getting closer to meeting the goal. If at the current game state there is a free building that builds the specific unit and there are enough resources, the unit is built. If a free building does not exist, it tries to build one based on the following condition: there are no buildings of that type in the process of being built or there are no busy buildings of that type. This is done to have some randomness when choosing to build a new building, because we noticed that this implementation produces more varied build orders. Therefore, the times for the simulated build orders have a wider range, but they also contain better times overall. If the dependency of the building is not met, it tries to build the dependency, and this process is repeated until a building is built or until a building does not have a dependency. The dependencies are built only once, because having one instance of the dependency is enough to construct buildings that produce units, therefore building more would be a waste of resources.

The process described above only involves the constructions that are directly relevant to the goal. After the process is performed, if a decision was not made, it calls the randomDecision() method, which makes a decision relevant to resources. Whenever this method is called, if there are free workers they are reassigned to mineral patches or gas geysers based on the goal, and then a random decision is made inside the switch statement. If only minerals are relevant to the goal (only Marines need to be built), it will try to either build a Command Center, build a Worker, or reassign a Worker from collecting gas to collecting minerals. If both minerals and gas are relevant, it can also build a Refinery or reassign a Worker from collecting minerals to collecting gas. This method is designed such that only the relevant resources are considered, therefore it does not waste time or resources on unnecessary actions.

Workers are limited to the maximum number that can be assigned to collection sites, because we can have a maximum of 3 workers collecting resources on either geysers or patches. If more Workers would be built, they would not be able to be allocated to a collection site, so that would essentially be a waste of resources. The maximum number of Refineries is limited to the number of geysers, because their only purpose is to enable gas collection, so, again, building more would be a waste of resources. Command Centers are limited to 3, because having more would not improve the rate at which Workers are built, since they are limited as well. Workers are reassigned between gas and minerals collection based on the resources at the current game time, so if there is less gas than minerals, a worker would be reassigned from a mineral patch to a gas geyser, and vice versa. This is done to further optimise resource collection.

After the decision making process is completed, the program checks whether a decision has been made, and if so, it keeps making decisions until a decision is no longer made. This is done so that all free workers are assigned to a collection site, because ideally we would not have workers not doing anything. This is especially useful in the beginning, because the simulation is initiated with 6 free workers, and they all get reassigned in the first second. It also allows multiple constructions to be built every second, which improves the overall time it takes to reach the required game state. For example, if both Marines and Vikings need to be built, at the same second it can start building a Marine and a Starport if all conditions and dependencies are met.

The decision making process was designed this way to ensure that only valuable decisions would be made for the specific goal given. However, the algorithm allows for some randomness in order to produce wider time ranges for accomplishing a specific goal. These time ranges also contain better overall times than a more rigid algorithm, therefore this more flexible approach was preferred. The time ranges tend to increase when there is a larger number of units to be achieved due to the increased permutations of decisions that can be made after the game simulation begins.

Extensions

The design of the basic program made implementing solutions relatively easy.

For the “delay” extension, only some additional conditions needed to be added to produce more accurate times for realistic players. In this implementation, the decision making process is only

allowed every 5 seconds. Multiple decisions can still be made every second. The overall time is closer to reality, but not drastically different to the times produces by the basic implementation.

For the “intermediate” extension, the classes for the constructions specified in the advanced goals were added and supply was implemented. The overall logic for making decisions for the new constructions was similar to the one used for the basic requirement, but this decision making process also took supply into consideration. The class storing the game state keeps track of the total supply that exists in the simulation, the supply already used, the maximum supply allowed, and the supply needed to achieve the goal. When a decision is made, it is also checked that enough supply is available for that decision. The randomDecision() method was modified so that a decision can be made to only improve supply (build a Command Center or a Supply Depot), further optimising the algorithm. If the goal entered exceeds the maximum supply allowed or if the maximum supply is reached during the simulation, an informative message is printed to the terminal and the program terminates without printing a build order. (Note: when running the “intermediate” program with Siege Tanks, they should be entered as “Tanks” and not “Siege Tanks”, otherwise it will be picked up as an error).

TESTING

To test our solution, we ran multiple tests at different stages of the building process. The examples in the following section demonstrate the types of tests we ran and the output our final program produces. For the basic deliverable program, we tested the program with various valid inputs and checked that it printed the desired build order in the right format. For the “delay” extension, the we tested the timings the program produced and compared it to the basic deliverable timings. For the “intermediate” extension, we tested that supply was implemented and printed correctly and that the additional error messages when printed when required. Both the basic program and the extensions were tested with varied invalid inputs to ensure that error handling was working properly. All implementations were also tested to ensure that different aspects of the game state (number of units, resources, etc.) were updated properly (To do this, printing methods were used. They were commented out so as not to clutter the output printed to the terminal, but they are indicated in the program and can be uncommented to test different aspects of the functionality).

EXAMPLES

Basic

What was Tested	Command Line Arguments	Expected Output	Actual Output
Wrong number of command line arguments	`java StarcraftOptimiser`	Useful error message about program usage printed and program terminates	`Usage: java StarcraftOptimiser <goal>`
	`java StarcraftOptimiser "6 Marines" "10 Hellions"`	Useful error message about program usage printed and program terminates	`Usage: java StarcraftOptimiser <goal>`
Invalid command-line arguments	`java StarcraftOptimiser cheese`	Useful error message with information about error and program terminates	`Invalid goal entered. Error found near 'cheese'.`
	`java StarcraftOptimiser "6 Buildings, 10 Constructions"`	Useful error message with information about error and program terminates	`Invalid goal entered. Error found near 'building'.`
	`java StarcraftOptimiser "10 Workers"`	Useful error message with information about error and program terminates	`Invalid goal entered. Error found near 'worker'.`
Invalid goal input syntax	`java StarcraftOptimiser "Hellions 2"`	Useful error message with information about error and program terminates	`Invalid goal entered. Error found near 'hellions'.`

Valid goal entered with only one unit type (Marines)	`java StarcraftOptimiser "6 Marines"`	Optimal build order is printed with time stamps, along with the total time elapsed in the simulation	`worker (finish at 0:17) 0:00`
			`worker (finish at 0:37) 0:20`
			`supply depot (finish at 1:04) 0:34`
			`worker (finish at 1:02) 0:45`
			`worker (finish at 1:20) 1:03`
			`barracks (finish at 2:23) 1:18`
			`worker (finish at 1:47) 1:30`

			worker (finish at 2:04) 1:47 barracks (finish at 3:00) 1:55 worker (finish at 2:21) 2:04 barracks (finish at 3:25) 2:20 marine (finish at 2:51) 2:26 worker (finish at 2:48) 2:31 marine (finish at 3:16) 2:51 worker (finish at 3:12) 2:55 marine (finish at 3:25) 3:00 marine (finish at 3:41) 3:16 marine (finish at 3:50) 3:25 marine (finish at 3:50) 3:25 Total time: 3:50
Valid goal entered with only one unit type (Hellions)	java StarcraftOptimiser "10 Hellions"	Optimal build order is printed with time stamps, along with the total time elapsed in the simulation	worker (finish at 0:17) 0:00 worker (finish at 0:36) 0:19 supply depot (finish at 1:04) 0:34 worker (finish at 1:01) 0:44 refinery (finish at 1:27) 0:57 worker (finish at 1:23) 1:06 refinery (finish at 1:57) 1:27 worker (finish at 1:51) 1:34 barracks (finish at 2:54) 1:49 worker (finish at 2:24) 2:07 worker (finish at 2:41) 2:24 worker (finish at 3:01) 2:44 factory (finish at 3:55) 2:55 worker (finish at 3:23) 3:06 worker (finish at 3:42) 3:25 factory (finish at 4:31) 3:31 worker (finish at 3:59) 3:42 hellion (finish at 4:25) 3:55 worker (finish at 4:20) 4:03 hellion (finish at 4:55) 4:25 worker (finish at 4:42) 4:25 hellion (finish at 5:01) 4:31 factory (finish at 5:31) 4:31 hellion (finish at 5:25) 4:55 hellion (finish at 5:31) 5:01 worker (finish at 5:20) 5:03 hellion (finish at 5:55) 5:25 hellion (finish at 6:01) 5:31 hellion (finish at 6:01) 5:31 worker (finish at 5:51) 5:34 factory (finish at 6:44) 5:44 hellion (finish at 6:25) 5:55 hellion (finish at 6:31) 6:01 Total time: 6:31
Valid goal entered with only one unit type (Vikings)	java StarcraftOptimiser "1 Viking"	Optimal build order is printed with time stamps, along with the total time elapsed in the simulation	worker (finish at 0:17) 0:00 worker (finish at 0:35) 0:18 refinery (finish at 1:00) 0:30 worker (finish at 0:58) 0:41 supply depot (finish at 1:27) 0:57 worker (finish at 1:25) 1:08 refinery (finish at 1:54) 1:24 worker (finish at 1:49) 1:32 worker (finish at 2:06) 1:49 worker (finish at 2:23) 2:06 barracks (finish at 3:24) 2:19 worker (finish at 2:44) 2:27 worker (finish at 3:06) 2:49 worker (finish at 3:24) 3:07

			factory (finish at 4:24) 3:24 worker (finish at 3:41) 3:24 worker (finish at 3:59) 3:42 worker (finish at 4:22) 4:05 command center (finish at 5:49) 4:09 worker (finish at 4:39) 4:22 starport (finish at 5:19) 4:29 worker (finish at 4:56) 4:39 starport (finish at 5:38) 4:48 worker (finish at 5:15) 4:58 starport (finish at 5:58) 5:08 worker (finish at 5:35) 5:18 viking (finish at 6:08) 5:26 Total time: 6:08
Valid goal entered with only one unit type (Medivacs)	`java StarcraftOptimiser "3 Medivacs"`	Optimal build order is printed with time stamps, along with the total time elapsed in the simulation	worker (finish at 0:17) 0:00 worker (finish at 0:34) 0:17 refinery (finish at 0:59) 0:29 worker (finish at 0:56) 0:39 refinery (finish at 1:23) 0:53 worker (finish at 1:20) 1:03 supply depot (finish at 1:47) 1:17 worker (finish at 1:44) 1:27 worker (finish at 2:01) 1:44 worker (finish at 2:22) 2:05 barracks (finish at 3:23) 2:18 worker (finish at 2:47) 2:30 worker (finish at 3:04) 2:47 worker (finish at 3:22) 3:05 worker (finish at 3:39) 3:22 factory (finish at 4:23) 3:23 worker (finish at 4:00) 3:43 worker (finish at 4:17) 4:00 command center (finish at 5:57) 4:17 worker (finish at 4:40) 4:23 starport (finish at 5:26) 4:36 worker (finish at 4:58) 4:41 starport (finish at 5:43) 4:53 worker (finish at 5:16) 4:59 starport (finish at 5:59) 5:09 worker (finish at 5:33) 5:16 medivac (finish at 6:08) 5:26 worker (finish at 5:52) 5:35 medivac (finish at 6:25) 5:43 command center (finish at 7:47) 6:07 medivac (finish at 7:16) 6:34 Total time: 7:16
Valid goal with more than one unit type entered	`java StarcraftOptimiser "2 Hellions, 2 Vikings, 1 Medivac"`	Optimal build order is printed with time stamps, along with the total time elapsed in the simulation	worker (finish at 0:23) 0:06 supply depot (finish at 0:55) 0:25 worker (finish at 1:02) 0:45 refinery (finish at 1:21) 0:51 worker (finish at 1:27) 1:10 refinery (finish at 1:49) 1:19 worker (finish at 1:50) 1:33 worker (finish at 2:08) 1:51 barracks (finish at 3:06) 2:01 worker (finish at 2:31) 2:14 worker (finish at 2:48) 2:31 worker (finish at 3:07) 2:50 factory (finish at 4:06) 3:06 worker (finish at 3:24) 3:07 worker (finish at 3:41) 3:24

			factory (finish at 4:31) 3:31 worker (finish at 3:58) 3:41 worker (finish at 4:15) 3:58 starport (finish at 4:57) 4:07 worker (finish at 4:32) 4:15 hellion (finish at 4:59) 4:29 worker (finish at 4:54) 4:37 hellion (finish at 5:12) 4:42 worker (finish at 5:12) 4:55 medivac (finish at 5:39) 4:57 starport (finish at 6:02) 5:12 worker (finish at 5:33) 5:16 worker (finish at 5:51) 5:34 viking (finish at 6:21) 5:39 worker (finish at 6:09) 5:52 viking (finish at 6:44) 6:02 Total time: 6:44
Valid goal with all units entered	`java StarcraftOptimiser "2 Marines, 1 Medivac, 1 Hellion, 1 Viking"`	Optimal build order is printed with time stamps, along with the total time elapsed in the simulation	worker (finish at 0:17) 0:00 worker (finish at 0:34) 0:17 refinery (finish at 1:00) 0:30 worker (finish at 0:56) 0:39 worker (finish at 1:13) 0:56 refinery (finish at 1:32) 1:02 worker (finish at 1:35) 1:18 supply depot (finish at 1:58) 1:28 worker (finish at 1:57) 1:40 worker (finish at 2:14) 1:57 worker (finish at 2:31) 2:14 barracks (finish at 3:32) 2:27 worker (finish at 2:57) 2:40 worker (finish at 3:16) 2:59 barracks (finish at 4:10) 3:05 worker (finish at 3:33) 3:16 marine (finish at 3:57) 3:32 worker (finish at 3:51) 3:34 factory (finish at 4:44) 3:44 worker (finish at 4:15) 3:58 marine (finish at 4:28) 4:03 factory (finish at 5:10) 4:10 worker (finish at 4:34) 4:17 factory (finish at 5:26) 4:26 worker (finish at 4:55) 4:38 factory (finish at 5:43) 4:43 hellion (finish at 5:21) 4:51 worker (finish at 5:14) 4:57 starport (finish at 5:59) 5:09 worker (finish at 5:32) 5:15 worker (finish at 5:49) 5:32 starport (finish at 6:32) 5:42 worker (finish at 6:06) 5:49 viking (finish at 6:46) 6:04 worker (finish at 6:30) 6:13 command center (finish at 8:05) 6:25 worker (finish at 6:47) 6:30 worker (finish at 7:05) 6:48 medivac (finish at 7:35) 6:53 Total time: 7:35

Delay

What was Tested	Command Line Arguments	Expected Output	Actual Output

Valid goal with more than one unit type entered. Decisions made only every 5 seconds	`java StarcraftOptimiser "6 marins, 4 hellions"`	Optimal build order is printed with time stamps, along with the total time elapsed in the simulation	`worker (finish at 0:27) 0:10`
			`supply depot (finish at 1:10) 0:40`
			`worker (finish at 1:12) 0:55`
			`barracks (finish at 2:25) 1:20`
			`worker (finish at 1:57) 1:40`
			`refinery (finish at 2:20) 1:50`
			`barracks (finish at 3:15) 2:10`
			`marine (finish at 2:50) 2:25`
			`worker (finish at 2:42) 2:25`
			`marine (finish at 3:25) 3:00`
			`refinery (finish at 3:30) 3:00`
			`marine (finish at 3:40) 3:15`
			`barracks (finish at 4:25) 3:20`
			`marine (finish at 4:20) 3:55`
			`marine (finish at 4:25) 4:00`
			`factory (finish at 5:10) 4:10`
			`marine (finish at 4:45) 4:20`
			`worker (finish at 4:52) 4:35`
			`worker (finish at 5:17) 5:00`
			`hellion (finish at 5:40) 5:10`
			`worker (finish at 5:37) 5:20`
			`hellion (finish at 6:10) 5:40`
			`hellion (finish at 6:40) 6:10`
			`worker (finish at 6:27) 6:10`
			`hellion (finish at 7:10) 6:40`
			
			`Total time: 7:10`

Intermediate

What was Tested	Command Line Arguments	Expected Output	Actual Output
Supply exceeded when setting the goal.	`Java IntermediateStarcraftOptimiser "80 banshees"`	Useful error message with information about error and program terminates	`Maximum supply exceeded. Cannot meet goal.`

Valid goal entered with only one unit type (Banshees)	`Java IntermediateStarcraftOptimiser "10 banshees"`	Optimal build order is printed with supply and time stamps, along with the total time elapsed in the simulation	`(supply)`
			`0 worker (finish at 0:17) 0:00`
			`1 refinery (finish at 0:49) 0:19`
			`1 worker (finish at 0:47) 0:30`
			`2 refinery (finish at 1:17) 0:47`
			`2 worker (finish at 1:16) 0:59`
			`3 supply depot (finish at 1:46) 1:16`
			`3 worker (finish at 1:50) 1:33`
			`4 worker (finish at 2:07) 1:50`
			`5 worker (finish at 2:26) 2:09`
			`6 supply depot (finish at 2:47) 2:17`

			6 barracks (finish at 3:38) 2:33 6 supply depot (finish at 3:19) 2:49 6 worker (finish at 3:11) 2:54 7 worker (finish at 3:34) 3:17 8 supply depot (finish at 3:52) 3:22 8 worker (finish at 3:53) 3:36 9 factory (finish at 4:45) 3:45 9 worker (finish at 4:10) 3:53 10 supply depot (finish at 4:39) 4:09 10 worker (finish at 4:33) 4:16 11 worker (finish at 4:55) 4:38 12 supply depot (finish at 5:12) 4:42 12 starport (finish at 5:46) 4:56 12 worker (finish at 5:18) 5:01 13 supply depot (finish at 5:46) 5:16 13 worker (finish at 5:35) 5:18 14 starport (finish at 6:21) 5:31 14 worker (finish at 5:56) 5:39 15 banshee (finish at 6:50) 5:50 18 worker (finish at 6:15) 5:58 19 supply depot (finish at 6:34) 6:04 19 worker (finish at 6:35) 6:18 20 banshee (finish at 7:21) 6:21 23 starport (finish at 7:23) 6:33 23 supply depot (finish at 7:12) 6:42 23 worker (finish at 7:07) 6:50 24 banshee (finish at 7:58) 6:58 27 supply depot (finish at 7:43) 7:13 27 banshee (finish at 8:21) 7:21 30 worker (finish at 7:46) 7:29 31 supply depot (finish at 8:13) 7:43 31 banshee (finish at 8:49) 7:49 34 worker (finish at 8:15) 7:58 35 supply depot (finish at 8:44) 8:14 35 worker (finish at 8:43) 8:26 36 command center (finish at 10:11) 8:31 36 banshee (finish at 9:42) 8:42 39 worker (finish at 9:07) 8:50 40 supply depot (finish at 9:27) 8:57 40 banshee (finish at 10:16) 9:16 43 supply depot (finish at 9:58) 9:28 43 worker (finish at 10:00) 9:43 44 banshee (finish at 10:49) 9:49 47 supply depot (finish at 10:32) 10:02 47 command center (finish at 11:52) 10:12 47 banshee (finish at 11:17) 10:17 50 worker (finish at 10:37) 10:20 51 worker (finish at 10:42) 10:25 52 supply depot (finish at 11:06) 10:36 52 banshee (finish at 11:43) 10:43 Total time: 11:43
Valid goal entered with only one unit type (Thors)	`Java IntermediateStarcraftOptimiser "1 thor"`	Optimal build order is printed with supply and time stamps, along with the total time elapsed in the simulation	(supply) 0 worker (finish at 0:17) 0:00 1 refinery (finish at 0:52) 0:22 1 refinery (finish at 1:08) 0:38 1 worker (finish at 1:06) 0:49 2 worker (finish at 1:23) 1:06 3 supply depot (finish at 1:46) 1:16 3 worker (finish at 1:42) 1:25 4 worker (finish at 2:09) 1:52 5 supply depot (finish at 2:23) 1:53 5 worker (finish at 2:26) 2:09 6 barracks (finish at 3:28) 2:23 6 worker (finish at 2:49) 2:32 7 worker (finish at 3:07) 2:50 8 supply depot (finish at 3:34) 3:04 8 worker (finish at 3:32) 3:15 9 worker (finish at 3:49) 3:32

			10 supply depot (finish at 4:05) 3:35 10 factory (finish at 4:53) 3:53 10 worker (finish at 4:19) 4:02 11 supply depot (finish at 4:44) 4:14 11 worker (finish at 4:38) 4:21 12 factory (finish at 5:40) 4:40 12 worker (finish at 5:02) 4:45 13 armory (finish at 6:04) 4:59 13 supply depot (finish at 5:44) 5:14 13 worker (finish at 5:31) 5:14 14 worker (finish at 5:49) 5:32 15 supply depot (finish at 6:14) 5:44 15 worker (finish at 6:07) 5:50 16 thor (finish at 7:04) 6:04 Total time: 7:04
Valid goal entered with only one unit type (Tanks)	Java IntermediateStarcraftOptimiser "5 tanks"	Optimal build order is printed with supply and time stamps, along with the total time elapsed in the simulation	(supply) 0 worker (finish at 0:17) 0:00 1 worker (finish at 0:34) 0:17 2 refinery (finish at 1:00) 0:30 2 worker (finish at 0:56) 0:39 3 refinery (finish at 1:24) 0:54 3 worker (finish at 1:18) 1:01 4 supply depot (finish at 1:49) 1:19 4 worker (finish at 1:44) 1:27 5 worker (finish at 2:03) 1:46 6 supply depot (finish at 2:28) 1:58 6 worker (finish at 2:23) 2:06 7 worker (finish at 2:43) 2:26 8 worker (finish at 3:00) 2:43 9 supply depot (finish at 3:18) 2:48 9 worker (finish at 3:19) 3:02 10 worker (finish at 3:36) 3:19 11 supply depot (finish at 3:50) 3:20 11 barracks (finish at 4:42) 3:37 11 worker (finish at 4:01) 3:44 12 supply depot (finish at 4:24) 3:54 12 worker (finish at 4:19) 4:02 13 worker (finish at 4:36) 4:19 14 supply depot (finish at 4:56) 4:26 14 worker (finish at 4:57) 4:40 15 factory (finish at 5:42) 4:42 15 factory (finish at 5:53) 4:53 15 worker (finish at 5:14) 4:57 16 supply depot (finish at 5:40) 5:10 16 worker (finish at 5:32) 5:15 17 factory (finish at 6:25) 5:25 17 worker (finish at 5:53) 5:36 18 supply depot (finish at 6:11) 5:41 18 tank (finish at 6:36) 5:51 21 worker (finish at 6:13) 5:56 22 tank (finish at 6:51) 6:06 25 supply depot (finish at 6:44) 6:14 25 tank (finish at 7:10) 6:25 28 tank (finish at 7:21) 6:36 31 supply depot (finish at 7:18) 6:48 31 worker (finish at 7:09) 6:52 32 tank (finish at 7:47) 7:02 Total time: 7:47
Valid goal entered with only one unit type (Marauders)	Java IntermediateStarcraftOptimiser "3 marauders"	Optimal build order is printed with supply and time stamps, along with the	(supply) 0 worker (finish at 0:17) 0:00 1 worker (finish at 0:34) 0:17 2 supply depot (finish at 1:04) 0:34 2 worker (finish at 1:00) 0:43 3 refinery (finish at 1:34) 1:04 3 worker (finish at 1:32) 1:15 4 refinery (finish at 1:54) 1:24

		total time elapsed in the simulation	4 worker (finish at 1:49) 1:32 5 worker (finish at 2:06) 1:49 6 barracks (finish at 3:06) 2:01 6 worker (finish at 2:29) 2:12 7 supply depot (finish at 2:58) 2:28 7 worker (finish at 2:53) 2:36 8 barracks (finish at 3:57) 2:52 8 worker (finish at 3:15) 2:58 9 marauder (finish at 3:38) 3:08 11 worker (finish at 3:34) 3:17 12 supply depot (finish at 3:54) 3:24 12 marauder (finish at 4:08) 3:38 14 worker (finish at 4:01) 3:44 15 marauder (finish at 4:27) 3:57 Total time: 4:27
Valid goal with more than one unit type entered	Java IntermediateStarcraftOptimiser "1 marauder, 3 marines, 1 hellion, 1 banshee"	Optimal build order is printed with supply and time stamps, along with the total time elapsed in the simulation	(supply) 0 worker (finish at 0:17) 0:00 1 worker (finish at 0:34) 0:17 2 refinery (finish at 0:59) 0:29 2 worker (finish at 1:00) 0:43 3 refinery (finish at 1:24) 0:54 3 worker (finish at 1:19) 1:02 4 supply depot (finish at 1:49) 1:19 4 worker (finish at 1:44) 1:27 5 worker (finish at 2:01) 1:44 6 supply depot (finish at 2:22) 1:52 6 worker (finish at 2:18) 2:01 7 barracks (finish at 3:21) 2:16 7 supply depot (finish at 3:01) 2:31 7 worker (finish at 2:58) 2:41 8 barracks (finish at 4:02) 2:57 8 worker (finish at 3:20) 3:03 9 factory (finish at 4:21) 3:21 9 worker (finish at 3:44) 3:27 10 marine (finish at 3:59) 3:34 11 supply depot (finish at 4:18) 3:48 11 marauder (finish at 4:29) 3:59 13 marine (finish at 4:27) 4:02 14 worker (finish at 4:25) 4:08 15 supply depot (finish at 4:48) 4:18 15 marine (finish at 4:52) 4:27 16 worker (finish at 4:46) 4:29 17 hellion (finish at 5:07) 4:37 19 supply depot (finish at 5:20) 4:50 19 worker (finish at 5:13) 4:56 20 starport (finish at 6:03) 5:13 20 worker (finish at 5:37) 5:20 21 supply depot (finish at 5:58) 5:28 21 worker (finish at 5:54) 5:37 22 starport (finish at 6:37) 5:47 22 worker (finish at 6:12) 5:55 23 supply depot (finish at 6:33) 6:03 23 banshee (finish at 7:15) 6:15 Total time: 7:15
Supply exceeded when building units	Java IntermediateStarcraftOptimiser "90 marauders"	Useful error message with information about error and program terminates	Maximum supply reached. Goal was not met.

EVALUATION

Overall, our solution proves to be efficient in producing an optimal build order by only running one simulation and is able to gracefully handle various error cases that might occur. The build orders generated by the program are not perfect in terms of the time taken to reach a given game state, but are still optimal to a large extent because the build orders are within a reasonable range of consistent timings. In addition, the program implements a more sophisticated search strategy consisting of multiple algorithms that take into account maximising resources and only making necessary decisions within the simulation. Hence, it is a successful and logical solution for the presented task.

INDIVIDUAL CONTRIBUTION

180000520:

In the beginning, my partner and I pair-programmed together to model all of the classes and create a very basic decision making method that made random decisions. After this was completed, we worked individually on implementing different parts of the program. I implemented the functionality to update the state of the game. To build the more sophisticated decision-making algorithm that is implemented in the final solution, we brainstormed and pair-programmed. We also pair-programmed to implement the extensions.

CONCLUSION

In conclusion, this practical has taught us how to model and stimulate real-life scenarios, more specifically build orders in a real-time strategy game. We also picked up new concepts from this practical, such as lambda expressions and Java Reflection. This practical has enabled us to learn more about states and designing decision-making algorithms as well. We thought that the hardest parts of this practical were modelling the game state and figuring out how to track the game state and time elapsed in the game as the different decisions were made. In addition, we also found writing algorithms to point the program to making logical decisions to achieve the goal to be a big challenge and spent a substantial amount of time on that.

Given more time, we would have tried to think of more algorithms to reduce the randomness of decisions that were made when the game was simulated, which could also make the timings produced shorter and the range of timings narrower.