140011146 140000346 CS1006

Overview:

In this practical we had to make a build order optimiser for the video game Starcraft 2: Heart of the Swarm. The goal is to be able to enter any amount of units into the program and it should find the faster build order in which those number of units will be met. The extensions we have implemented are: Supply, chronoboost, warp gate, upgrades, extra bases/resource depletion, special units and GUI.

Design:

Search tree algorithm:

To find the most optimised build order, we used a combination of a random tree walk and a heuristic approach. Every time step, we find all the possible actions while applying a heuristic filter, so as to not have many useless results that don't reach the goal. This means that we can quickly find a basic build order. The depth of our tree is determined by the fastest build completion time found, which means, as faster builds are found, the search time will improve considerably, since the depth of the tree will be shallower.

getPossibleOperations() gets all the possible actions that could occur at that time in the game. It runs through a loop of all the units in the game and adds operations of those units such as “build” unit or “chronoboost” unit as long as it satisfies the heuristic conditions we applied. When we create a new **TimeState**, one of the possible operations is chosen randomly. This is the random part of our tree. That operation is then executed in the simulation in **executeOperation()**.

To avoid a **StackOverflowError**, which happened early on in development, we decided to implement a **next()** method, which returns a the new **TimeState** object, as opposed to creating it in the constructor of the current object. We hoped that the latter would be sufficient for java to garbage collect our old **TimeState** (since there were no references to it), but this proved to not be the case.

Data:

To represent all the data of the many units and their resource costs, we use a class called **UnitData** which stores strings as the names of the units and doubles as their resource costs. To make sure everything is consistent, the spelling of the names matter as we also use the same string names for values such as dependencies and where the unit is built from. Supply cost is an integer and not a double because we don't have decimals in supply for the game. The other resources are doubles as we calculate our mining income as a double.

The **Datasheet** class is where all the unit names and costs go. It also holds most of the constants in the game, such as the resource mining rate and maximum supply. This was very good for us to easily refer to dependencies and resource costs of the various units as we could find the dependency of a Stalker by going **Datasheet.getDependency(“Stalker”).**

As for data collections, we mainly used **Hashmaps** as a way to store the name of a unit and the number of those units. We extended **HashMap<String,Integer>** in a class called **UnitNumbers**,

for convenient access. For example, to deal with the difference between the number of units in the goal and number of units in the game we simply compared the two **Hashmaps** and if all the units in the goal have the same number as in our simulation, then the goal is complete. When we first started, we would decrease the number of units in the goal whenever a new one was made and check when all the units in the goal are at 0. The issue with this was whether the goal should be complete when the unit is queued or when it is completed.

Supply:

For supply, we have it as one of the resources in the **Datasheet** and **UnitData**. If the supply is equal to the supply cap then units that cost supply cannot be made. This check happens when the unit is queued to be built as it would in the real game instead of when the unit is actually made. This prevents new units to be added to the build queue when the game is supply blocked. Being supply blocked is bad and makes a build less optimised because there will be a period of time when being supply blocked where no new units can be made but resources are still being gathered. This can lead to situations where we have enough resources to complete the goal but we are not using the time efficiently.

To increase the supply cap, pylons or nexi must be constructed. We have a few methods to try and predict when to build another pylon to increase the cap. We also make sure not to build too many pylons as it would be a waste of resources. For example we check the total supply of all the units required in the goal and if we don't need to go past that number then we don't build extra pylons. There is a hard supply cap in the game of 200 so we cannot go past this supply limit even if we build more pylons.

Chronoboost:

Chronoboost decreases the time taken to build a unit by 50% while it is being chronoboosted. For us, we made it so if a unit was being chronoboosted, whenever it's build time increments, instead of increasing by 1, it increases by 1.5 as it is being built 50% faster. We use a simple boolean to check if a unit is being chronoboosted while it is being constructed. In the build order, we show which building we are chronoboosting. Buildings and special units like Archons and Interceptors cannot be chronoboosted as they are not built from a building. So we use a simple to check to see if the unit (or upgrade) is built from a building before allowing it to be chronoboosted.

Warp Gate:

Warp Gate is one of the upgrades that can be researched that can make Gateways become Warpgates. As Warpgates, instead of building a unit, it warps in a unit and has a cooldown depending on what unit was warped in. The cooldown can be Chronoboosted to cooldown faster. For this we had to use a new structure to store the cooldowns of the Gateway units in **Datasheet** called **warpgateCooldowns** and use a different build called **WarpgateBuild**. As Warpgate greatly increases the speed at which units can be made, we had to make new checks on all the heuristics depending if we are using Warpgate or not. We also have a check to see if we should research Warpgate even if it's not in the goal as it decreases our build time significantly.

140011146 140000346 CS1006

Upgrades:

Upgrades are similar to units but they don't take up supply and you can only have one of a single upgrade. Instead of checking and making sure the user can only enter one upgrade, we did it in the GUI. We only allow users to tick or un-tick whether they want an upgrade or not. This prevents users from trying to get two or more of a single upgrade without having to do the checks in our program.

Extra bases/resource depletion:

Every nexus is built at a base with a limited number of resources. Although a single goal is not likely to use all the resources from one base, it could happen. Another base also increases the mineral and gas income as more probes can be mining at the same time on more resource patches.

For resource depletion, it is simply done when resources are added from the income. When we add resources to the bank, we also take it away from the resources in the base. The **totalMinerals** and **totalGas** are the amounts of resources that belong to the base while **minerals** and **gas** are the resources we currently have in the bank. We also added a maximum number of extra bases that can be taken as a static final in the **Datasheet**, this is to limit the program so it doesn't build hundreds of bases and basically get an infinite number of resources. This is also realistic with the game as it is unusual for a player to take over a dozen bases in a single game.

Special units:

Archons:

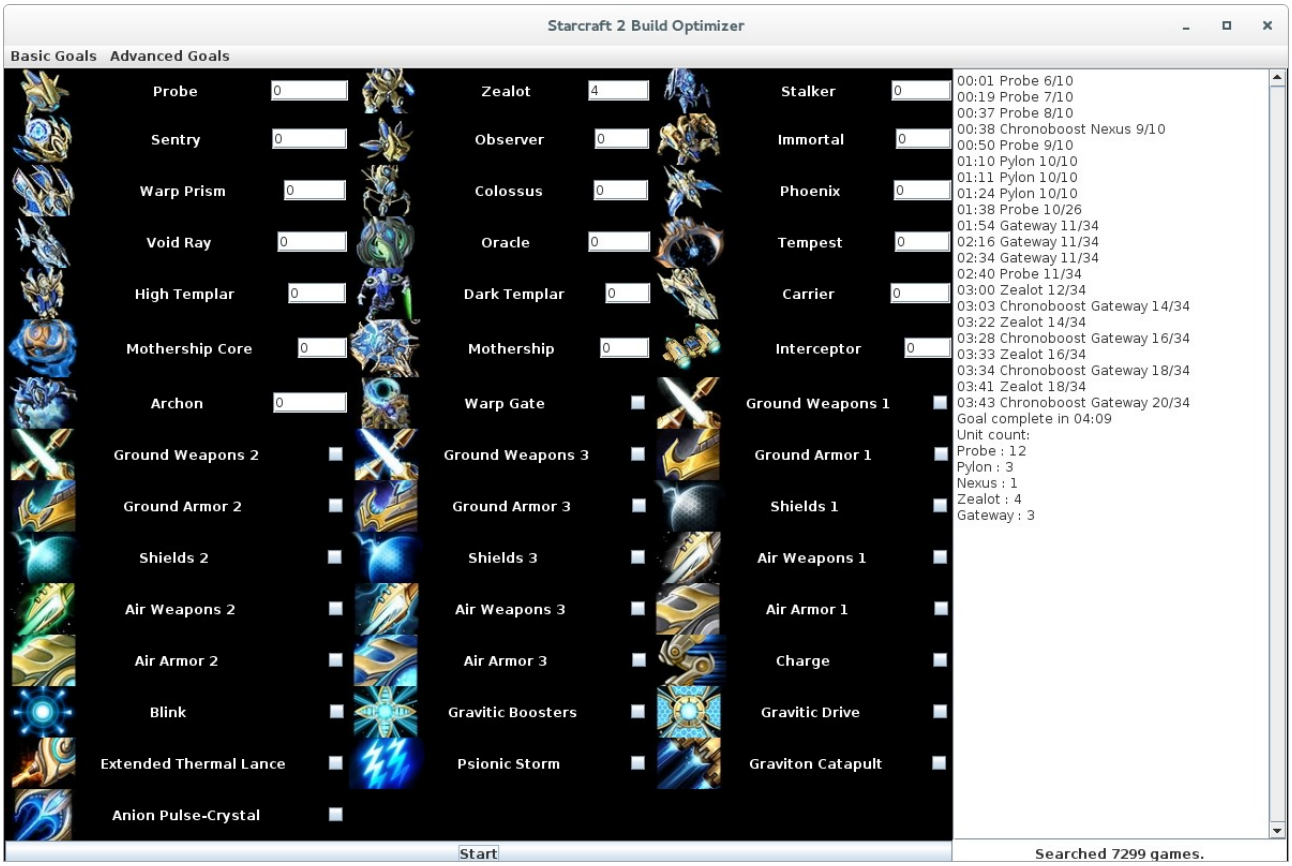
Archons were very complex to implement as they are not built from buildings but are made by the merging for two Templars. It becomes even more complicated as they can be made by merging two dark/high Templars or one of each. When it comes to the optimiser, we have to decide which merging is more efficient. If the goal was just Archons, it is cheaper resource-wise to merge them out of two high Templars. But there are other considerations such as whether the goal contained either of the two types of Templar.

Interceptors:

Interceptors were also challenging as four interceptors come with one Carrier and each Carrier can only build a maximum of eight Interceptors. But since the Interceptors are built directly form the Carriers it wasn't too hard to make Carriers come with four Interceptors when they are built and have Interceptors simply be added to the buildQueue of Carriers.

GUI:

For the GUI, we used GridBagLayout, as it was able to be customized to represent a grid of units, in addition to an output. We extended JPanel to easily make a panel to store the picture of the unit, its name and a JComponent which would be the input. We extended this class to make a different panel for unit and upgrades, since upgrades behave differently, as they can only be reaseached once.



In addition, we accessed the Swing worker thread directly from our Game thread, to ensure the redrawing of the GUI (when the counter is increased, which happens very frequently) is executed on the Swing thread, instead of inside our game thread, which could cause runtime exceptions.

Testing:

Iteration:

In many instances of our program ,we had to iterate through an array while removing elements. We found that this lead to various run-time errors, which lead us to use an Iterator on the list while simultaneously checking if the next element existed, which proved slightly slower, but allowed us to execute removals without queuing them for deletion.

Threading:

Before we implemented the GUI, we weren't concerned with threading, but later came across problems with redrawing the GUI. Therefore, we created our own GameThread class, which handles everything related to the simulation, and implemented some methods to safely end the thread when the user clicks the stop button. This was necessary since our implementation of the simulation runs ad infinitum, even when it finds the most optimal solution, since it has not way of knowing whether or not there is a better one without continuing to search.

Timer:

When we got to a finished stage in our program, games with goals that contained lots of units

completed much slower than games with few units in the goal. To solve this problem and understand why our program was so slow, we implemented a **Timer** class that could check how faster our methods executed. At first we thought it was our heuristics since we loop through all the units regardless of whether they are in the goal. With the timer we found out that the issue wasn't to do with our methods but that with a large goal, we create many more **TimeStates** than with a small goal and that was what was slowing our program down.

```
SCWindow [Java Application] /usr/local/jdk1.8.0_31/bin/java (2 Apr 2015 14:54:08)
'Playing game' ended in 42 milliseconds.
'Playing game' starting...
'Playing game' ended in 43 milliseconds.
'Playing game' starting...
'Playing game' ended in 45 milliseconds.
'Playing game' starting...
'Playing game' ended in 41 milliseconds.
'Playing game' starting...
'Playing game' ended in 42 milliseconds.
'Playing game' starting...
'Playing game' ended in 45 milliseconds.
'Playing game' starting...
'Playing game' ended in 44 milliseconds.
'Playing game' starting...
'Playing game' ended in 46 milliseconds.
```

Our goals:

- 12 Zealots, 8 Stalkers, 4 Sentries, 3 Colossus, 1 Observer, 2 High Templar, 3 Archons, Ground Weapons 2, Ground Armor 2, Blink , Charge, Extended Thermal Lance, Psionic Storm.
- 20 Stalkers, Blink, Ground Weapons 1, Ground Armor 1
- 5 Phoenix, 12 Void Rays, 3 Carriers, Air Weapons 2, Air Armor 2

Evaluation:

One problem with our program is that large goals take a long time for a solution to be found. From testing we know that this is not a problem with the way we approach the solution, but simply the **TimeStates** are both too deep and too wide, making it take more time. Once a single solution is found the time to complete new games speeds up as we stop going down a path if the time exceeds our solution's best time.

Another imperfection of our simulation is that our chronoboosting doesn't carry over to the next unit building when it completes construction. A part of this is because we made the decision not to implement unit queueing, since it is considered an inefficient use of resources, especially in professional play. Another reason is that our Operation class might be too limited, in that it contains only basic information, like a verb and a noun, instead of a more complex data structure, like a reference to a specific object. On the other hand, our simple implementation means that the simulation can be easily extensible, and new features and races can be added quickly, since the adding of operations and the executing of them are independant of each other, with only one needing to check for conditions.

140011146 140000346 CS1006

Conclusion:

Our program uses a combination of random tree walk and heuristics. The heuristics prevents the tree from becoming too big and also helps speed up the time it takes the tree walk to find the goal. Although our program runs slowly if the goal contains lots of units, once it finds a correct path it starts to speed up. The program will also loop infinitely until stopped to keep trying to find a better solution. It will only consider solutions that take less time than the current best solution and so we can stop simulating games once they hit that time cap. We are able to find and output the build orders for the basic and advanced goals as well as some of our own. We also completed many extensions highlighted in the design section above.

Sample output: