Advanced Programming Techniques
ExAdvPt
Winter Term 2019/20

Project Part 2: Forward Simulation

Forward Simulation

Objective

The goal of the second project part is to write a forward simulator for SC2 build lists. Your program should determine if a given build list is valid, and if it is, simulate when which unit or building is built.

Your team should implement the forward simulation for all three SC2 races. Each team member is responsible for implementing one race. Implementing special abilities is optional but helps a lot in the third project phase when optimizing build lists.

On StudOn you can find the tech-tree information for all three races in CSV format. So you do not have to extract the information manually from the tech-tree diagrams.

You will also find another CSV file with additional information, so you do not have to extract the information manually from these PDFs.

There is an interactive validation tool, where you should check your output before submitting your solution:

`https://www10.cs.fau.de/advptSC2/`

Building and running your program

- submit your code in a file called `project.zip`
- after unpacking your code (`unzip project.zip`) there should be two bash script files: `build.sh` and `forwardSim.sh`
- calling `./build.sh` should build your program
- calling `./forwardSim.sh <race> <buildlistFile>` should run your program, simulate the given buildlist and print a detailed log to stdout as described below. Possible values for race are: *terran*, *protoss*, and *zerg*. Example buildlist files are available on StudOn.

Simulator output

Your forward simulator should print detailed status information in JSON format to stdout. On StudOn you can find links to sample output for a buildlist of each race.

The output always contains information about the simulated race and if the builtlist is valid. Here is an example of an invalid Terran buildlist:

```
{ "game"             : "Terr",
  "buildlistValid" : "0"
}
```

Valid buildlists additionally have a *messages* property as well as an optional *initialUnits* property.

```
{ "game"             : "Terr",
  "buildlistValid" : "1",
  "messages"         : <listOfMessages, as specified below>,
  "initialUnits"   : { "scv"            : [ "0", "1", "2", "3", "4", "5", "6", "7",
                                            "8", "9", "10", "11" ],
                       "command_center": [ "6" ]
                     }
}
```

The *initialUnits* property serves the purpose to uniquely identify each starting unit by an ID. These IDs can be used later on to specify the target of special abilities. If you do not trigger any special abilities in the forward simulation you can omit the *initialUnits* property and all following properties marked in blue. In your simulator these IDs can be of any type, as long as the printed string representation in the log is unique for each created unit/building.

Message Format

The *messages* property of above JSON object is a list of message objects. A message object has to be added only for times where "something happens" e.g. when at least one event (as defined below) occurs. The message object contains the current time, the resources and worker distribution **at the end** of the timestep as well as a list of events.

```
{ "time"    : <current simulation time in seconds (integer)>,
  "status" : { "resources" :
                   { "minerals"   : <minerals, rounded down to next integer>,
                     "vespene"    : <vespene,  rounded down to next integer>,
                     "supply"     : <total supply available>,
                     "supply-used": <supply used by all your units>          },
               "workers"   :
                  { "minerals"   : <number of workers collecting minerals>,
                    "vespene"    : <number of workers collecting vespene>   }
             }
  "events" : <list of events, for event specification see below>,
}
```

Event format

Events occur either when the creation of a unit/building is started or finished, or when a special ability is triggered.

Build-start event:

```
{ "type"      : "build-start",
  "name"      : <name of unit or building>,
  "producerID": <id of producer>
}
```

The *build-start* event countains the type of unit/building that is being created in the *name* property e.g. barracks, scv. The *producerID* property specifies the unit/building instance that creates this new unit/building. For this property a unit/building instance (i.e. an ID) is required, not the name of the unit type.

Build-end event:

```
{" type"          : "build-end",
   "name"         : <name of unit or building>,
   "producerID"   : <name of producer>,
   "producedIDs" : [ <ids of produced units/buildings> ]
}
```

The *name* and *producerID* have the same meaning as in the *build-start* event. For the same build process these properties have the same value as the corresponding *build-start* message.

In the *producedIDs* list, the IDs of the newly created units/buildings are specified. The *producedIDs* property is a list, since one build process can produce multiple units e.g. in case of *zerglings*.

Special ability events:

Modelling special abilities is optional. However if you model them, you have to add a special ability event:

Terran:

```
{ "type"        : "special",
  "name"        : "mule",
  "triggeredBy" : <id of building triggering the mule>
}
```

Zerg:

```
{ "type"          : "special",
  "name"          : "injectlarvae",
  "triggeredBy"   : <id of queen>,
  "targetBuilding": <id of larvae producing building>
}
```

Protoss:

```
{ "type"          : "special",
  "name"          : "chronoboost",
  "triggeredBy"   : <id of nexus triggering the chrono boost>,
  "targetBuilding": <id of building>
}
```

When to specify IDs

All properties marked in blue are optional. You have to use them only in case to want to refer to a specific unit when activating a special ability. For example in case you want to activate “injectlarvae”, the specified ID of the Hatchery has to be known either by having it listed before in *initialUnits* or in *producedIDs* of a *build-end* message. Similarly for the Terran MULE ability, the ID of the base building has to be known beforehand.

If you want to use Protoss Chronoboost, the IDs of all buildings have to be known.

Start configuration

- 50 minerals, 0 vespene
- 12 workers (SCV/Probe/Drone)
- 1 base building (Command Center/Nexus/Hatchery)
- additionally for Zergs: 1 overlord and 3 larvae inside the Hatchery

Timestep Order

Within a single timestep the ordering of sub-tasks should be as follows:

1. Update resources according to last worker distribution. Between the start of the simulation and the first message it is assumed that all 12 workers collect minerals.
2. Check if any builds are finished (generate *build-end* events)
3. Start new build or start any special abilities
4. Redistribute workers
5. Report resource and worker status at this point (*status* part of message)

Please stick to this order to ensure comparable log outputs. Note that the resource count and worker distribution refers to the state after builds and special abilities have been started.

Fulfill the following restrictions when implementing your forward simulator:

- In each second either a single build can be started or a special ability can be triggered. However, there can be multiple events in one second, e.g. a *build-start* and one or more *build-end* events.
- There can be at most 2 vespene buildings per base
- buildlists taking longer than 1000 seconds should be reported as invalid
- for this task we only model the terran "MULE" special ability, the "Extra Supplies" ability must not be used in this task. Build lists that could only be built with extra supplies have to be reported as invalid.

Zerg Details:

- initially the hatchery already holds 3 larvae

- without the special ability, zerg hatcharies/layers/hives can hold up to 3 larvas
- in each timestep, if the number of larvae in one of the hatcheries is less than three, (unless the timer is running already) a timer is started so that after 11s a new larva spawns in this hatchery.
- building a zergling pair is modeled by a single *build-start* and *build-end* e.g.

```
{ "type": "build-end", "name": "zergling",
  "producedIDs": [ "11", "12"], "producerID": "3",}
```

- while a building is morphing into a new building it still exists in the sense that dependent units/buildings can be built
- when a unit is morphing, the supply difference is accounted at the begin of the morph operation