CO202: Coursework 2

Autumn Term, 2020

1 *Introduction*

This assignment builds on the previous one to generate an AI that can play a game of Imperial Conquest. This coursework will be automarked: test suites are provided to check your answers.

In order to run your tests, you should run the following:

```
$ make test
```

This does not test every problem in this coursework: AIs will be assessed by making them compete against in-house versions.

The submission of your coursework should be done via LabTS using `git` commits in the usual way. The only file you need to change is `clever-ai/Submission2.lhs`. You must check that your code compiles and that the tests can be run.

During development, you might find it useful to load the source file into `GHCi`. You can do this by running

```
$ ghci -iclever-ai clever-ai/Submission2.hs
```

Make sure to run this command from the root and not inside of the `clever-ai` directory, as the `Submission2.hs` module depends on the `Lib.hs` module that is in the root directory.

from the root of the repository. This way, you can test your solutions, and ensure there are no compilation errors as you're working on your code.

2 *Game Rules*

A game of Imperial Conquest uses the datatypes described in the previous coursework. You are supplied with a server that implements all of these rules, and that will orchestrate a game between two AIs. Nevertheless, here is a brief explanation of the rules implemented in the server.

The game is run as a series of turns. Each turn is broken up into different phases which execute in order: departure, advancement, and arrival.

In the departure phase, players can order ships in a source planet they own to travel as a fleet along a wormhole to a target planet, which takes the number of turns indicated by the wormhole.

In the advancement phase any planets owned by a player increase the number of ships on the planet by the planet growth rate. Neutral planets do not produce ships, but may have a number of defending ships garrisoned there. Fleets advance by one turn through the wormhole.

The implementation of the rules can be inspected in `Main.hs`, which contains the authoritative implementation. The code that generates the maps is in `Map.hs`.

In the arrival phase, fleets who are at the end of a wormhole arrive at a planet. If more than one fleet arrives at a planet at the same time, then friendly fleets are combined. If two opposing fleets arrive at the same time they fight. The surviving fleet is the larger of the two, and contains all of its ships minus the number of ships in the smaller fleet. If the opposing fleets are the same size then there are no survivors.

The surviving ships in a fleet then invade the planet. If the target planet is owned by the same player as the invading fleet then the fleet is added to the ships garrisoned on the planet. Otherwise, the invading ships fight the garrisoned ships. If the invading fleet is larger than the garrison then it wins, and the planet is conquered by the owner of the fleet: the number of ships in the fleet minus the original garrison are the survivors and are then posted as the new garrison. Otherwise, the garrison is at least as large as the fleet, and the number of survivors is the original garrison minus the number of ships in the fleet.

The game continues until either there are no more turns to play, or one of the players is annihilated. A player is annihilated if they run out of all ships including those in a garrison or in a fleet. A player is also disqualified if they run out of time. By default each game lasts at most 400 turns, and each AI is given a total budget of 60 seconds of wall time to compute its answers for the duration of a game.

Maps are automatically generated to create an initial `GameState` that is *fair* in the sense that they are centrally symmetric. The graph that is produced ensures that every planet has at least one incoming edge and one outgoing edge. Players always start with one planet, all other planets are neutral.

3 *Initial Setup*

Imperial Conquest runs in the terminal, and boasts a fashionable text-based interface. Getting started is easy: your git repository has been set up with the game server and a simple AI. Once you have acquired the source code, the first task is to build the server, which can be done by executing the following command in the repository's root directory:

```
$ make server
```

This will create the server executable, called `Main`, in the repository root. The server takes two paths as arguments, and it will look for an `AI.hs` file in these two paths, compile them, and run them against each other on a randomly generated map.

The server finds the `AI.hs` in the `stupid-ai` directory, compiles it, then spawns it as a subprocess. The communication code between the AI and server has been provided for you. The first argument is *player 1 (red)*, and the second argument is *player 2 (green)*. To make AI writing easier, both AIs receive the state of the world as if they were *player 1*.

To try out the game, you can run the provided simple AI against itself by invoking the following command:

```
$ ./Main stupid-ai stupid-ai
```

Make sure to run the game in a large enough terminal window, otherwise the graphics will fall apart! A running instance can always be killed by pressing `Ctrl+c`.

As its name suggests, `stupid-ai` is not a very strong contender. Its strategy is to evenly split its forces in all directions from every planet it owns, hoping for the best. If you run the command above and leave it running for a while, you will notice in the side bar that the match is even:

```
Seed: 229367113-1
Remaining turns: 684
Player 1 ships:  1631
Player 1 growth: 40
Player 2 ships:  1631
Player 2 growth: 40
```

And indeed, this game ultimately ends in a `Draw`. This is because the generated maps are *fair*, as they are all centrally symmetric. In addition to the game status, the side bar also displays the seed that was used to generate the map. The server optionally takes the seed as the third command line argument, which can be used to replay the same map:

```
$ ./Main stupid-ai stupid-ai --seed 229367113-1
```

Rendering the map in every turn does have a non-neglibile overhead, so in case you wish to run the game without rendering the screen, you can do so by calling

```
$ ./Main stupid-ai stupid-ai --headless
```

Your goal in this coursework is to implement a better AI than `stupid-ai`. A skeleton has been set up in the `clever-ai` directory. You are expected to only modify the `Submission2.hs` file.

In particular, you will be asked to implement a number of predefined strategies, and finally to come up with your own. The `Strategy` type defines the list of strategies.

```
data Strategy
  = Pacifist
  | ZergRush
  | PlanetRankRush
  | Skynet
  deriving (Enum, Bounded, Show, Read)
```

The main entry point of the AI is the `logic` function, which gets called every turn with the most recent `GameState`, as computed by the server, and returns a list of orders to be carried out by the server.

`logic` itself does not implement any AI logic, other than incrementing the counter of the `AIState` (more on this below). Instead, it dispatches to the relevant implementation of the given strategies. Your task is to implement these strategies.

```
logic :: Strategy -> GameState -> AIState -> ([Order], Log, AIState)
logic strat gs ai
  = let logic' = case strat of
          Pacifist       -> pacifist
          ZergRush       -> zergRush
          PlanetRankRush -> planetRankRush
          Skynet         -> skynet
    in logic' gs ai {turn = turn ai + 1}
```

The `AIState` type is the AI's internal state, which you are free to modify (but do not remove the existing `turn` field). Initially, it is defined as such:

```
data AIState = AIState
  { turn :: Turns
  } deriving Generic

initialState :: AIState
initialState = AIState
  { turn = 0
  }
```

As you can see, the provided `logic` function increments the current turn by one every time it gets called. This is useful for keeping track of the game's progress. If you wish to persist information (perhaps cache results) between turns, you can add more fields to the state by extending the `AIState` type, and providing an initial value for it in `initialState`.

Finally, in addition to returning a list of orders and the modified AI state, the logic function also returns a log, for debugging.

```
type Log = [String]
```

A complimentary strategy, `pacifist`, is given to you:

```
pacifist :: GameState -> AIState -> ([Order], Log, AIState)
pacifist _ ai = ([], ["Do no harm."], ai)
```

Here, the AI does not do anything, and records this fact in the log. Note that the `AIState` is returned unmodified in this case.

You can test your AI against others (or itself) by invoking

```
$ ./Main clever-ai stupid-ai --strategy1 Pacifist
```

The `--strategy1` flag allows you to specify which strategy to run your AI with. Similarly, `--strategy2` is the strategy the second AI is called with. (Here, specifiying `--strategy2` would have no effect, as `stupid-ai` does the same thing no matter what it's told).

Once the server is running, the log messages are dumped into `out/log1.txt` and `out/log2.txt` for the two players respectively. If you wish to monitor the logs in real time, you can do so by invoking

```
$ tail -F out/log1.txt
```

in a new terminal window (also from the root directory of the repository). Doing so with the given setup will show the logs (displayed for each turn):

```
------------- 0: -------------
Doing nothing

------------- 1: -------------
Doing nothing

------------- 2: -------------
Doing nothing

...
```

The logging infrastructure is there to help you debug your AI code, so feel free to put anything you wish there.

4 *Coordinating Forces*

Your first task is to implement an "all in" strategy, known as the *Zerg Rush*. The plan is to find an enemy planet, and attack it with all available forces. Once it's conquered, find another one, and repeat.

Here is a helper function that identifies whether a given planet is owned by the enemy:

```
enemyPlanet :: Planet -> Bool
enemyPlanet (Planet (Owned Player2) _ _) = True
enemyPlanet _                            = False
```

Problem 1: *Finding Enemies*

Implement the `findEnemyPlanet` function, which finds the first enemy planet from the `GameState` as its target.

```
findEnemyPlanet :: GameState -> Maybe PlanetId
findEnemyPlanet = undefined
```

HINT: You may want to make use of functions such as `M.filter` and M.toList from Data.Map.

Once you have set your eyes on the target, you need to plan an all out attack. The idea is simple: from all of your planets, find the shortest path to the target, then send all your available forces along these paths from your planets.

You may want to make use of code from the previous coursework. If you do, copy it directly into Submission2.lhs since this is the only code we assume you will change.

The key strength of this strategy is that unless the enemy pays extra attention to defending their planet, it is very likely that you conquer it. However, the findEnemyPlanet function might return a different planet every time the enemy conquers a new planet, but the Zerg Rush attack needs to single out a planet and keep attacking until it's conquered.

Problem 2: *Tracking Enemies*

To make sure that you don't lose focus of a planet until it's yours, extend the AIState type to hold the current target in the rushTarget field, which should initially be set to **Nothing**.
HINT: The AIState type should look like this:

```
data AIState = AIState
  { turn       :: Turns
  , rushTarget :: Maybe PlanetId
  }
```

The send function orders the specified number of ships to be sent through a given wormhole, provided that you own the source of the wormhole, and have enough ships. If no number is specified, or the number is too large, then it simply sends all available ships.

Problem 3: *Sending Ships*

Finish the implementation of the send function:

```
send :: WormholeId -> Maybe Ships -> GameState -> [Order]
send wId mShips st = undefined
 where
  Wormhole (Source src) _ _ = lookupWormhole wId st
  planet@(Planet _ totalShips _) = lookupPlanet src st
```

HINT: You should ensure that you do not send more ships than you have garrisoned on a planet.

Finally, you need to orchestrate the attack. A simple way of doing this is to order all your ships from all your planets to attack the target via the shortest path from each planet to the target.

The shortestPath function returns the shortest path from a source planet to a target planet.

```
shortestPath :: PlanetId -> PlanetId -> GameState
```

```
             -> Maybe (Path (WormholeId, Wormhole))
shortestPath src dst st
  = case filter ((== dst) . target) (shortestPaths st src) of
      [] -> Nothing
      (x : _) -> Just x
```

You might find the following functions helpful:

```
ourPlanet :: Planet -> Bool
ourPlanet (Planet (Owned Player1) _ _) = True
ourPlanet _ = False

ourPlanets :: GameState -> Planets
ourPlanets (GameState ps _ _) = M.filter ourPlanet ps

lookupWormhole :: WormholeId -> GameState -> Wormhole
lookupWormhole wId (GameState _ wormholes _)
  = wormholes M.! wId

lookupPlanet :: PlanetId -> GameState -> Planet
lookupPlanet pId (GameState planets _ _)
  = planets M.! pId
```

Problem 4: *All out attack*

Implement the `attackFromAll` function that issues ships from all of your planets to be sent towards a target along the shortest paths.

```
attackFromAll :: PlanetId -> GameState -> [Order]
attackFromAll targetId gs
  = undefined
```

HINT: If there is no direct path from a planet to the target, it should send its ships to the next planet towards the target along the shortest path.

Now you have all the required pieces for carrying out a `ZergRush` attack.

Problem 5: *Zerg Rush*

Implement the `zergRush` strategy.

```
zergRush :: GameState -> AIState
         -> ([Order], Log, AIState)
zergRush gs ai = undefined
```

HINT: First check if there is a target in `rushTarget`, and that this target is not already yours. If this is the case, then initiate an all-out attack using the `attackFromAll` function. Otherwise, pick a new target and store it in the `rushTarget` field of `AIState`.

To see your AI in action, execute the following command:

```
./Main clever-ai stupid-ai --strategy1 ZergRush
```

5 Ranking Targets

Choosing which planets to attack first is crucial aspect of any reasonable strategy. There are many different ways of deciding which planet is important, and in this part of the coursework we will be interested in the planets which are the most *connected*.

To this end, we will be implementing the PageRank algorithm, which was originally used by Google to rank webpages according to their importance. The measure of importance of a webpage in this setting is the number of links into it from other pages.

5.1 PageRank

The goal of the PageRank algorithm is to assign the likelihood that a given web page will be visited when links are visited randomly by a user. The web pages and links are modelled as a graph whose vertices are pages and whose edges are links between pages. There are certain important assumptions that must not be violated for PageRank to work properly: all pages link to at least one other, there is at most one link from a page to another, and no page links to itself.

Each page p is thus assigned a probability $PR(p)$ between 0 and 1 that they will be visited. The formula for a given page p_i is given by the following:

$$PR(p_i) = \frac{1-d}{n} + d \times \sum_{p_j \in S(p_i)} PR(p_j) \times \frac{1}{|T(p_j)|} \tag{1}$$

Here, the value n is the number of webpages in consideration. The value d is the damping factor which is the probability that the user moves to another page. The set of pages $S(p_i)$, called the *sources* of p_i, contains all pages p_s where there is a link from p_s to p_i. The set of pages $T(p_j)$, called the *targets* of p_j, contains all pages p_t where there is a link from p_j to p_t.

The idea is that there is a probability that the page p_i is accessed directly by some user and remains there, which is covered by the term $\frac{1-d}{n}$. Otherwise, there is a chance that the user arrives at p_i from some other page p_j with a PageRank $PR(p_j)$ that links to it. This is calculated by starting with that PageRank, and taking the probability that the user clicks on a link on that page, which is d, where it is assumed that links out of a page are visited with equal probability.

The value of Equation 1 is typically approximated by iterating the equation in turns, where each value of $PR(p_i)$ at some given iteration $t+1$ is based on all the values $PR(p_j)$ at iteration t. The initial value of $PR(p_i)$ is set to $\frac{1}{n}$, where n is the number of pages. This technique eventually converges to the true PageRank.

5.2 *Implementation*

The PageRank can be thought of as a mapping from pages to their probability, so this is implemented as a `Map` which takes a `pageId`, which is an abstract type variable, to a value between 0 and 1. The following code sets up the type `PageRanks` as the map that associates `pageId`s with their given `PageRank`. A `PageRank` is in fact just a wrapper around a `Double`, and inherits many of its properties.

A value of type `Map k v` associates keys of type `k` to values of type `v`. The interface for `Map` describes this datastructures and the complexities of its operations: `https://hackage.haskell.org/package/containers/docs/Data-Map-Lazy.html`

```
newtype PageRank = PageRank Double
  deriving (Num, Eq, Ord, Fractional)

type PageRanks pageId = Map pageId PageRank
```

Using this, given `pr :: PageRank pageId` the PageRank $PR(p_i)$ of a given page with index `i` is found in `pr M.! i`.

Initially, each page is given a probability of $\frac{1}{n}$, where n is the total number of pages. The following code achieves this:

```
initPageRanks :: (Graph g e pageId, Ord pageId)
              => g -> PageRanks pageId
initPageRanks g = M.fromList [ (p, PageRank (1 / fromIntegral n))
                             | p <- ps ]
  where ps = vertices g
        n  = length ps
```

As an example of using this function, here is a graph that contains some vertices connected as described in Figure 5.2 (edges weights do not make a difference to PageRank, but are needed for this graph definition):

```
example1 :: [(String, String, Integer)]
example1 = [("a","b",1), ("a","c",1), ("a","d",1),
            ("b","a",1), ("c","a",1), ("d","a",1), ("c","d",1)]
```

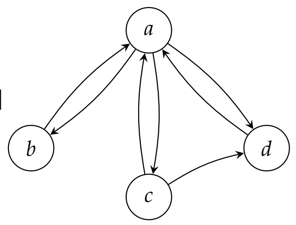

Figure 1: Graph of `example1`

The result of `initPageRanks example1` is as follows:

```
GHCi> initPageRanks example1
fromList [("a",0.250),("b",0.250),("c",0.250),("d",0.250)]
```

This shows that the `Map` that is created is the one that would associate equal weights to each of the nodes. Also note that the sum of all the weights is 1, since this is the total probability mass. In fact, this makes for a good sanity check for any PageRank result.

```
GHCi> sum (M.elems (initPageRanks example1))
1.000
```

Problem 6: *Initialise PageRank*

The function `initPageRanks` takes as its input a list of elements to construct a new PageRank map. This uses `fromList`, which takes $O(n \log n)$ time to compute for a given list of size n.
For our purposes, the `Map` can be created from the existing `Planets` value that is in the `GameState`, since it is itself a `Map`. This can be done more efficiently by using the `M.map` function since the whole `Map` need not be recreated. Implement `initPageRank'` that achieves this:

```
initPageRank' :: Map pageId a -> PageRanks pageId
initPageRank' = undefined
```

HINT: The `M.map` function is the qualified import from `Data.Map`: you will have to ignore the values in the map, either by making a custom function that ignores this parameter, or by using the `const` function.

The algorithm iterates the function `nextPageRank` through each `pageId`. The following code calculates the PageRank of p_i using what is outlined in Equation 1.

```
nextPageRank :: (Ord pageId, Edge e pageId, Graph g e pageId) =>
  g -> PageRanks pageId -> pageId -> PageRank
nextPageRank g pr i = (1 - d) / n + d * sum [ pr M.! j / t j
                                            | j <- s i ]
 where
  d   = 0.85
  n   = fromIntegral (length (vertices g))
  t j = fromIntegral (length (edgesFrom g j))
  s i = map source (edgesTo g i)
```

This equation only updates the new PageRank of a single page p_i for a given `i`. In order to update all the pages requires an update to all of the map. This can be done efficiently by using the `mapWithKey` function:

```
nextPageRanks :: Ord pageId => Graph g e pageId =>
  g -> PageRanks pageId -> PageRanks pageId
nextPageRanks g pr = M.mapWithKey (const . nextPageRank g pr) pr
```

Thus a single iteration of this function will update the weightings accordingly:

```
GHCi> nextPageRanks example1 (initPageRanks example1)
fromList [("a",0.569),("b",0.108),("c",0.108),("d",0.215)]
```

This shows that after one iteration page `"a"` is expected to be visited over half of the time. Further applications of `nextPageRanks` will redistribute weights until some convergence has happened.

The `iterate` function can be used to create an infinite list of iterations of a single function applied to an argument. This is how a list of all the `pageRanks` can be created for a given graph:

```
pageRanks :: (Ord pageId, Graph g e pageId) => g -> [PageRanks pageId]
pageRanks g = iterate (nextPageRanks g) (initPageRanks g)
```

Here are the first 5 results of `pageRanks example1`, which shows how the weights are converging towards some values.

```
GHCi> take 5 (pageRanks example1)
[ fromList [("a",0.250),("b",0.250),("c",0.250),("d",0.250)]
, fromList [("a",0.569),("b",0.108),("c",0.108),("d",0.215)]
, fromList [("a",0.358),("b",0.199),("c",0.199),("d",0.245)]
, fromList [("a",0.499),("b",0.139),("c",0.139),("d",0.223)]
, fromList [("a",0.405),("b",0.179),("c",0.179),("d",0.238)] ]
```

In fact, convergence is achieved somewhere around the 20th value:

```
GHCi> pageRanks example1 !! 20
fromList [("a",0.442),("b",0.163),("c",0.163),("d",0.232)]
```

This has a stable state where node `"a"` is clearly the most important node, followed by `"d"`, and where `"b"` and `"c"` are tied.

Finally, the PageRank algorithm can be expressed using the `pageRank` function:

```
pageRank :: (Ord pageId, Graph g e pageId) =>
  g -> PageRanks pageId
pageRank g = pageRanks g !! 200
```

This always takes the PageRank after 200 iterations, even if the function has already converged.

5.3 *Rank Convergence*

The `pageRank` function produces an infinite list of `PageRanks` and the algorithm does not check for convergence in any way: it will blindly continue to produce `PageRanks` even when no changes are made. Therefore, to pick out a particular `PageRanks` requires a lookup at some arbitrary point, as was done in the example above.

While exact numerical convergence is quite unlikely to happen quickly, the algorithm can converge to within some error margin extremely quickly. It would be nice if the PageRank was produced either after some arbitrary cutoff, or when convergence is achieved within some degree of accuracy.

The first step towards a PageRank that stops near convergence is to add a cutoff `k`:

```
nextPageRank' :: (Ord pageId, Edge e pageId, Graph g e pageId) =>
  g -> PageRanks pageId -> PageRank -> pageId -> PageRank -> Maybe PageRank
nextPageRank' g pr k i pri
  | abs (pri - pri') < k  = Nothing
  | otherwise             = Just pri'
 where
   pri' = nextPageRank g pr i
```

Convergence only occurs when every PageRank has converged. The following function threads through a `Bool` when it traverses the PageRanks that represents whether or not the traversal has converged so far. Initially this value is `True`, is passed along whenever the result of `nextPageRank'` indicates that nothing has changed. Otherwise, the PageRanks differ significantly and the flag is set to `False`. If all PageRanks have converged then `Nothing` is returned, indicating that nothing has changed, otherwise, the new PageRank `pr'` is returned.

```
nextPageRanks' :: Ord pageId => Graph g e pageId =>
  g -> PageRank -> PageRanks pageId -> Maybe (PageRanks pageId)
nextPageRanks' g k pr = case M.mapAccumWithKey nextPageRank'' True pr of
                          (True,  pr)  -> Nothing
                          (False, pr') -> Just pr'
  where
    nextPageRank'' converged i pri = case nextPageRank' g pr k i pri of
                            Nothing   -> (converged, pri)
                            Just pri' -> (False, pri')
```

Finally, the `pageRanks'` function iterates forever unless convergence up to a given cutoff value has been found.

```
pageRanks' :: (Ord pageId, Graph g e pageId)
  => g -> PageRank -> [PageRanks pageId]
pageRanks' g k = iterateMaybe (nextPageRanks' g k) (initPageRanks g)

iterateMaybe :: (a -> Maybe a) -> a -> [a]
iterateMaybe f x = x : maybe [] (iterateMaybe f) (f x)
```

Problem 7: *Converging PageRank*

Define `pageRank'` so that it uses `pageRanks'` to find the PageRank of all vertices in a given graph. The accuracy level `k` should be

`0.0001`, and you should let the iteration happen no more than `200` times.

```
pageRank' :: (Ord pageId, Graph g e pageId) =>
  g -> PageRanks pageId
pageRank' g = undefined
```

HINT: You should ensure that this works even when `pageRanks` returns a list whose length is shorter than `200`. You may want to make use of `take` and `last`.

5.4 *PlanetRank*

The PageRank algorithm is a good starting point for ranking the importance of a planet in a game of IMPERIAL CONQUEST. However, a few tweaks will be required.

Ranking planets in terms of their desirability is surely a topic for much debate and discussion. For simplicity, we will assume that the best planets to attack are strategically well placed with many outbound edges and that have a healthy growth rate. The PlanetRank algorithm ranks planets using this information, as well as the structure of the `GameState` graph.

The PageRank algorithm will need two modifications to accommodate the ranking of planets. First, PageRank places a high value on a web page that have many edges *going to* it, whereas PlanetRank places a high value on a planet that has many edges *coming from* it. In other words, the directions of links needs to be reversed: we will switch sources for targets, and vice versa. The second change is that, rather than contributing the PageRank back in equal proportion, a PlanetRank will contribute back in proportion to the growth rate. These changes are reflected in the following equation:

$$PR(p_i) = \frac{1-d}{n} + d \times \sum_{p_j \in T(p_i)} PR(p_j) \times \frac{G(p_i)}{\sum_{p_k \in S(p_j)} G(p_k)} \tag{2}$$

The term $G(p_k)$ is the growth rate of some planet p_k. Other than this addition the terms in this equation are simply the same as before. The value n is the number of planets in consideration. The value d is the damping factor. The set of pages $S(p_i)$, called the *sources* of p_i, contains all pages p_s where there is a link from p_s to p_i. The set of pages $T(p_j)$, called the *targets* of p_j, contains all pages p_t where there is a link from p_j to p_t.

In the original PageRank formula 1 the contribution to the PageRank of a given node p_j is $PR(p_j) \times 1/|T(p_j)|$, where $T(p_j)$ is the number of edges coming from p_j, or in other words, by default we assume that p_j evenly distributes its PageRank to the pages it connects to. For PlanetRank this is changed to $PR(p_j) \times G(p_i)/\sum_{p_k \in S(p_j)} G(p_k)$. So, planet p_i receives a contribution of the PlanetRank of p_j proportional to the growth rate of p_i among all the other planets connecting to p_j.

As an example, consider the following `GameState`:

```
example2 :: GameState
example2 = GameState planets wormholes fleets where
  planets = M.fromList
    [ (PlanetId 0, Planet (Owned Player1) (Ships 300) (Growth 7))
    , (PlanetId 1, Planet Neutral         (Ships 200) (Growth 2))
    , (PlanetId 2, Planet Neutral         (Ships 150) (Growth 3))
    , (PlanetId 3, Planet Neutral         (Ships 30)  (Growth 6))
    ]
  wormholes = M.fromList
    [ (WormholeId 0, Wormhole (Source 0) (Target 1) (Turns 1))
    , (WormholeId 1, Wormhole (Source 0) (Target 2) (Turns 1))
    , (WormholeId 2, Wormhole (Source 0) (Target 3) (Turns 1))
    , (WormholeId 3, Wormhole (Source 1) (Target 0) (Turns 1))
    , (WormholeId 4, Wormhole (Source 2) (Target 0) (Turns 1))
    , (WormholeId 5, Wormhole (Source 3) (Target 0) (Turns 1))
    , (WormholeId 6, Wormhole (Source 2) (Target 3) (Turns 1))
    ]
  fleets = []
```

Now consider how the PlanetRank of p_2 is calculated by applying Equation 2. In this instance, $T(p_2) = \{p_0, p_3\}$ since these are the planets which are the targets of p_2. Furthermore, $S(p_0) = \{p_1, p_2, p_3\}$, and $S(p_3) = \{p_0, p_2\}$ are the sets of planets that connect to p_0 and p_3 respectively. Instantiating the summations gives:

$$PR(p_2) = \frac{1-d}{n} + d \times \left(\frac{PR(p_0) \times G(p_2)}{G(p_1) + G(p_2) + G(p_3)} + \frac{PR(p_3) \times G(p_2)}{G(p_0) + G(p_2)} \right)$$

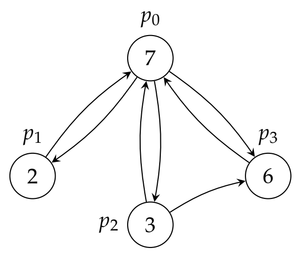

Figure 2: Graph of `example2`. Vertices display growth rate.

As with PageRank, the easiest way to calculate the PlanetRank is iteratively.

The types `PlanetRank` and `PlanetRanks` are required for the implementation of PlanetRank.

```
newtype PlanetRank = PlanetRank Double
  deriving (Num, Eq, Ord, Fractional)

type PlanetRanks = Map PlanetId PlanetRank

instance Show PlanetRank where
  show (PlanetRank p) = printf "%.4f" p
```

Initially, we assume the PlanetRank of each planet is equal, which is implemented by the following:

```
initPlanetRanks :: GameState -> PlanetRanks
initPlanetRanks g = M.fromList [ (p, PlanetRank (1 / fromIntegral n))
                               | p <- ps ]
  where ps = vertices g
        n  = length ps
```

This can be tested on `example2` in the terminal:

```
GHCi> initPlanetRanks example2
fromList [(0,0.2500),(1,0.2500),(2,0.2500),(3,0.2500)]
```

From here, the structure of the PlanetRank algorithm is essentially the same as PageRank, where the `planetRank` is given by picking out an approximation generated in the list given by `planetRanks`.

```
planetRank :: GameState -> PlanetRanks
planetRank g = planetRanks g !! 200

planetRanks :: GameState -> [PlanetRanks]
planetRanks g = iterate (nextPlanetRanks g) (initPlanetRanks g)
```

The `nextPlanetRanks` are calculated by calling `nextPlanetRank` on every planet.

```
nextPlanetRanks :: GameState -> PlanetRanks -> PlanetRanks
nextPlanetRanks g pr = M.mapWithKey (const . nextPlanetRank g pr) pr
```

Finally, `nextPlanetRank` is computed according to Equation 2.

Problem 8: *PlanetRank*

Implement `nextPlanetRank` by filling in the definitions of `targets` and `growths` according to Equation 2.

```
nextPlanetRank :: GameState -> PlanetRanks
               -> PlanetId -> PlanetRank
nextPlanetRank g@(GameState planets _ _) pr i =
 (1 - d) / n + d * sum [ pr M.! j * growth i / growths j
                       | j <- targets i ]
 where
  d   = 0.85
  n   = fromIntegral (length planets)

  growth :: PlanetId -> PlanetRank
  growth i  = (\(Planet _ _ g) -> fromIntegral g)
                                  (planets M.! i)
  targets :: PlanetId -> [PlanetId]
  targets i = undefined
```

```
    growths :: PlanetId -> PlanetRank
    growths j = undefined
```

Hint: Recall that a `GameState` is a `Graph`, so you can make use of graph and edge operations such as `edgesFrom`, `edgesTo`, `source`, and `target`. Also note that `PlanetRank` behaves like a `Double`, so the function `sum :: [PlanetRank] -> PlanetRank` will work as expected to sum some values.

Having implemented `nextPlanetRank` it should be possible to verify one step of the PlanetRank algorithm as follows:

```
GHCi> nextPlanetRanks example2 (initPlanetRanks example2)
fromList [(0,0.6112),(1,0.0761),(2,0.1592),(3,0.1534)]
```

As a sanity check, you may also want to verify that your `PlanetRank` always sums to `1`, which can be verified with the following function:

```
checkPlanetRanks :: PlanetRanks -> PlanetRank
checkPlanetRanks = sum . M.elems
```

Executing this on an example should produce the following (though you may find that due to precision some values are only very close to `1.0`):

```
GHCi> checkPlanetRanks (planetRanks example2 !! 100)
1.0000
```

6 *Combining Strategies*

Now the final task is to combine the strategy of Zerg Rush with that of Planet Rank. Recall that the Zerg Rush strategy was based on finding a single target (owned by the enemy), and attacking it until it is conquered. While this is an effective strategy, its main shortcoming is that the choice of planet is somewhat random. A much better approach would be to use the result of the PlanetRank algorithm to find the *most important* planet that is not owned by you, and conquer it.

Problem 9

Implement the `planetRankRush` algorithm, that targets the planet not owned by you (so either owned by `Player2` or `Neutral`), and attacks it until it is conquered. Once done, move on to the next highest ranking planet, and repeat until you win (or get obliterated by an even more sophisticated AI).

```
planetRankRush :: GameState -> AIState
```

```
                -> ([Order], Log, AIState)
planetRankRush _ _ = undefined
```

HINT: This function is algorithmically simple, but requires a few different parts to be in place. Firstly, since computing PlanetRank is quite expensive, you should think about caching it in the `AIState` (just like `rushTarget`) and avoid recomputation after the first round. Secondly, use the `attackFromAll` function from before to carry out the attack against the planet with the highest PlanetRank. At this point you might have better ideas about how to move your forces around, but save them for the `skynet` strategy!

7 *Skynet*

Problem 10: *Skynet*

Create your own AI and strategies to do whatever you like by implementing the `skynet` function:

```
skynet :: GameState -> AIState
       -> ([Order], Log, AIState)
skynet _ _ = undefined
```

There is no specification for what your AI should do. In order to succeed at this task, your AI needs to be able to beat the `planetRank` algorithm.
HINT: One idea would be to use the Knapsack algorithm to pick out nearby neutral planets before there is any conflict. You could also think about adjusting the Planet Rank algorithm so that it accounts for the travel distances between the planets. Also, none of the strategies above took into account the enemy's fleets!
Your submissions will be competing against each other in a tournament, and the winner will be announced at the end, winning eternal fame and glory! You can have friendly matches against each other by dropping your AI directory into next to one another and invoking them with the same server (renaming at least one from `clever-ai` to something else in this case to avoid name collisions), but we will run all of them against each other at the end.