Systems Programming – Assignment #1
The coalition race
2023/Fall

Lecturer in charge: Marina

TAs in charge: Itai, Nir

1 Before You Start

- It is mandatory to submit all the assignments in pairs. If you can't find a partner, use the designated forum.
- Read the assignment together with your partner and make sure you understand the tasks. **Before writing any code, make sure you read the whole assignment.**
- Skeleton files will be provided on the assignment page, you must use these classes as a basis for your project and implement (at least) all the functions that are declared in them.

KEEP IN MIND

While you are free to develop your project on whatever environment you want, your project will be tested and graded ONLY on a CS LAB UNIX machine. It is your own responsibility to deliver a code that compiles, links and runs on it. Failure to do so will result in a grade 0 to your assignment.

Therefore, it is mandatory that you compile, link and run your assignment on a lab unix machine before submitting it.

We will reject, upfront, any appeal regarding this matter!!

We do not care if it runs on any other Unix/Windows machine.

Please remember, it is unpleasant for us, at least as it is for you, to fail your assignments, just do it the right way.

2 Assignment Goals

The objective of this assignment is to design an object-oriented system and gain implementation experience in C++ while using classes, standard data structures, and unique C++ properties such as the “Rule of 5”. You will learn how to handle memory in C++ and avoid memory leaks. The resulting program must be as efficient as possible.

3 Assignment Definition

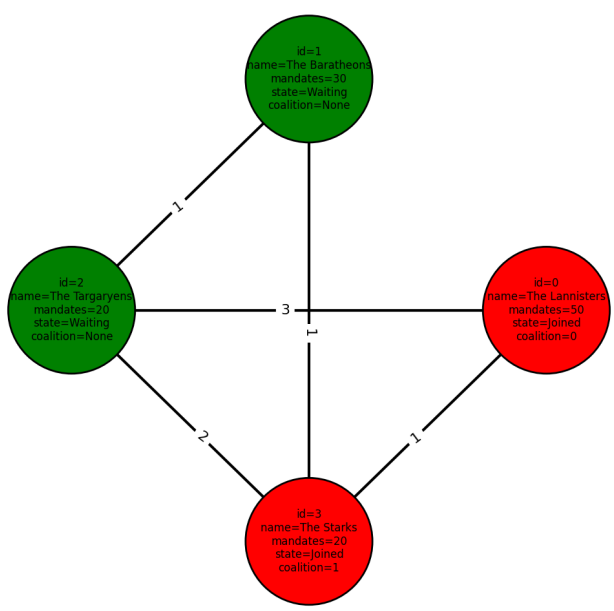
In a faraway democracy, SPLand, there is a political crisis, they can't form a government in years. The reason is complex promises about whether to cooperate with other parties.
We need you to help the politicians find a coalition!

After 5 elections, the SPLand president decided to change the method of forming a coalition by letting multiple "agents" from different parties try their best to create a 61 coalition as fast as possible.

In this assignment, you will write a C++ program that simulates the "Coalition Race" and report the first coalition formed (or failure).
The simulator should be based on a graph that contains:

- Parties as vertices
- Collaborations as edges (2 parties that agree to cooperate)
- “Similarity score” as edge weight, for every 2 connected parties

Here is an example for a graph at the beginning of the simulation. The colors represent the state of the parties (will be explained later).



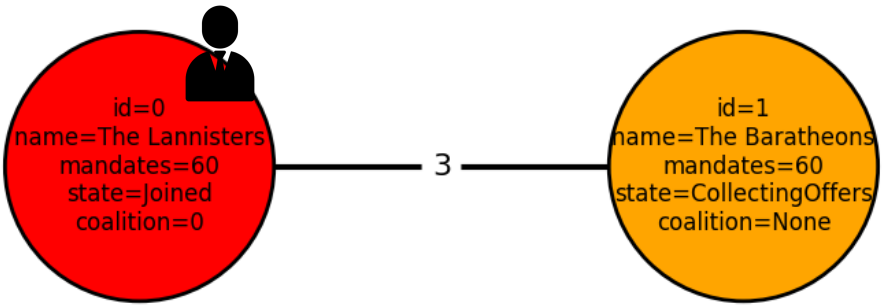
The program will receive a config file (JSON) as input, which includes the parties' details (id, name, number of mandates, join policy), the graph, and the list of agents (id, party id, selection policy). “Join policy” and “Selection policy” will be explained later.

Each agent belongs to a party. In every step, the agent offers an adjacent party to join. When a party has decided to join a coalition, it should **clone the agent who made the offer to the newly joined party** (so the joined party can now help the coalition).
To clarify:

- Coalition is a set of connected parties (in SPLand, connectivity is enough).
- In each party, there is a single agent if the party belongs to a coalition (state **Joined**).
- The total number of agents in the simulation equals the number of parties in the graph belonging to some coalition (state **Joined**).

<u>Example</u>

Suppose an agent from party#0 offered party#1 to join its coalition.
If party#1 accepts the offer, it adds itself to the coalition and adds a clone of the agent to the vector of agents (as well as updating the id and party id).



3.1 Classes

To solve this assignment, you will use the OOP approach. We will provide you skeleton files with some classes. Some of the declared methods will be implemented for you, and some are left for you to implement. You are free to add more classes, members, and methods to the existing classes but you **must not** change the given signatures.

Parser – This is a class with static methods only, that is responsible for parsing from JSON to object of class ***Simulation*** and the other way around. We supply the full implementation of the class, you should not change it, but you might want to look at it.

Simulation - This class is responsible for managing the simulation resources and making simulation steps (described in The program flow). This object is created by ***Parser*** using the initial values from the JSON configuration file. The Simulation object contains

- ***Graph***
- Vector of ***Agents***

Graph – This class is represented by an adjacency matrix (edges) and ***parties*** vector (vertices). Since the graph is undirected, you may assume that the matrix is symmetric. Note that an edge exists if and only if the weight is greater than 0.

Party – A vertex in the graph. Each party has an id, name, number of mandates, join policy and party state. The states are:

1. **Waiting** - the party is not a part of any coalition yet, and also didn't get offers yet.
2. **CollectingOffers** - after receiving the first offer, the party changes its state and starts collecting offers from ***Agents*** for joining coalitions.
   The party has a "cooldown" - a timer of exactly 3 iterations, until moving to the next state by joining some coalition. Note, if a party got the first offer at the end of iteration i, it will join some coalition at the beginning of iteration $i + 3$.
   Note - the iteration is counted even if the party did not get an offer in that iteration.
3. **Joined** - after 3 iterations, the party uses *JoinPolicy::join(...)* method for deciding which offer to accept. After joining a coalition, the party's state won't change anymore and it cannot join any other coalitions.

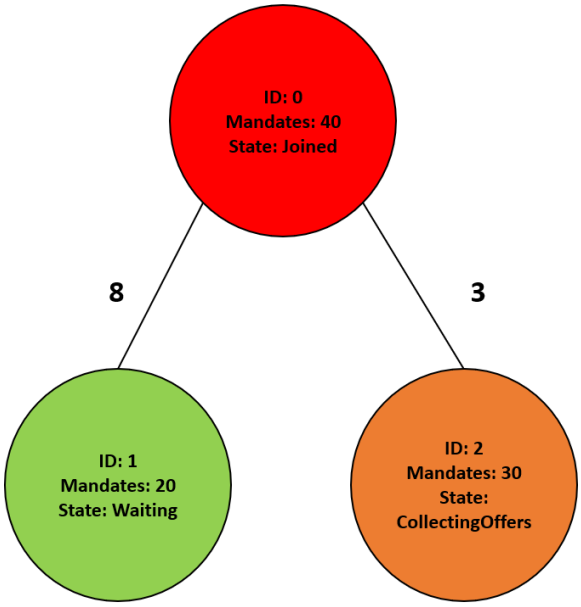
Agent – This class has agentId, partyId (each agent associated with a single party), and selection policy. In each step, an agent is trying to select a neighboring party (in the graph) using *SelectionPolicy::select(...)* and offers the party to join its coalition.

SelectionPolicy - an abstract class that represents a strategy algorithm for selecting the next party to offer. It has the abstract method *SelectionPolicy::select(...)* that will be implemented in the derived classes:

- MandatesSelectionPolicy - Selects the neighboring party with the most mandates.
- EdgeWeightSelectionPolicy - Selects the neighboring party with the highest edge weight.

If the maximum mandates/edge weight is not unique, take the first party (with the lower id).

<u>Example:</u>



In this example, the agent (associated with party#0) will select party#1 when using EdgeWeightSelectionPolicy, or party#2 when using MandatesSelectionPolicy.
Note that a party can receive offers in both **Waiting** and **CollectingOffers** states (as mentioned already, if a party is **Waiting**, it will change its state to **CollectingOffers** and trigger the cooldown timer).

JoinPolicy - an abstract class that represents a strategy algorithm for choosing which offer to accept and which coalition to join. It has *JoinPolicy::join(...)* abstract method that will be implemented in the derived classes:

- MandatesJoinPolicy - Selects the coalition with the largest number of mandates. Note that this refers to the number of mandates in the iteration when the offer is **accepted**. If the number of mandates is not unique, join the coalition that offered first.
- LastOfferJoinPolicy - Selects the coalition that made the last offer.

<u>Example:</u>
Suppose agent#1 offered party#1 to join its coalition in **iteration 0**, and agent#2 offered party#1 to join its coalition in **iteration 1**.
In **iteration 3**, party#1 needs to choose which coalition to join. In this iteration, the coalition of agent#1 has 45 mandates and the coalition of agent#2 has 40 mandates.
If party#1 is using MandatesJoinPolicy (configured in JSON file), it will join agent#1 coalition, and if the party is using LastOfferJoinPolicy, it will join agent#2 coalition.

You can (but don't have to) add more classes if you find it helpful.

3.2 The program flow

We recommend following the main.cpp implementation we supplied while reading this section.

The program receives the config file's path as the first command line argument. Once the program starts, it parses the config file and creates a “Simulation” object.
The format of the file is JSON, which is a very common format for object representation.

The main function checks every iteration if the termination conditions are satisfied (described in Termination conditions). Until they are satisfied, the main function calls the *Simulation::step()* method and saves the state of the simulation as JSON. At the end of the simulation, the results list will be written to an output file (this is how we can grade your solution).
All the file-handling methods are implemented for you in the supplied code.

The flow of *Simulation::step()* is applying *Party::step(Simulation& s)* for each party (from graph) and **then** applying *Agent::step(Simulation& s)* for each agent (keep the original order).

The flow of *Party::step(Simulation& s)* is to check if the status is **CollectingOffers** and update the timer if so. Once the timer of 3 iterations is done - join some coalition (using the JoinPolicy). When joining, the party’s state should be changed to **Joined** and the agent must be cloned.

The flow of Agent::step(*Simulation& s)* is to select a party using SelectionPolicy::select(...) and then offer the party to join the coalition (in which the agent is a member).
The selection of the party is **from the set of parties that satisfy the conditions**:

1. The party is a neighbor of the party to which the agent belongs.
2. The party is in **Waiting** or in **CollectingOffers** state.
3. The party **did not** already receive an offer from **any agent from the coalition to which the agent belongs.** That is, if an agent from a coalition offers a party to join, other agents from the same coalition should not select the party (this might be a waste of an offer).

   When there are no parties to select from, the agent is idle.

3.3 Configuration file format

The configuration file is given in a JSON format, and it contains a dictionary (hash map) with the following entries:

- parties - list of objects with **name**, **mandates**, **join_policy**.
- graph - a 2D list that represents the adjacency matrix.
- agents - list of objects with **party_id**, **selection_policy**.

The **agentId** and the **partyId** members are based on the index of the object in the list. When you clone an agent, make sure to change those fields in the cloned object.

You may assume that the config file is valid, i.e. it is a JSON file that contains the mentioned dictionary, the adjacency matrix is a symmetric $N \times N$ matrix (when N is the number of parties), and the number of mandates sums to 120.

Parsing of JSON is easy in C++ using Niels Lohman's JSON for Modern C++ - an open-source project. If you want, you can learn how to use this package, and see examples here.

3.4 Termination Conditions

The program terminates when one of the coalitions reaches at least 61 mandates **or** when all the parties have already joined a coalition (all parties are in state **Joined**).
You may assume that any given input leads to one of these conditions.

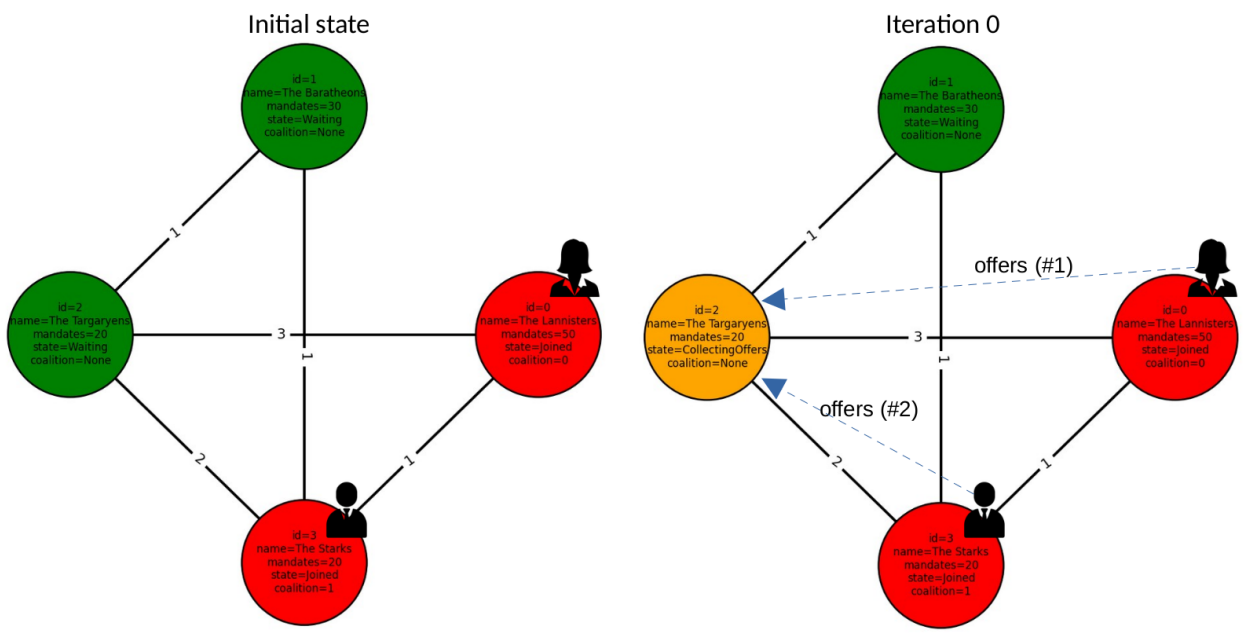
3.5 Output

The program creates a “.out” file in JSON format (which is a bit different from the input JSON). The parser is doing this part for you, but you will have to implement the methods marked with “TODO” in the skeleton files.

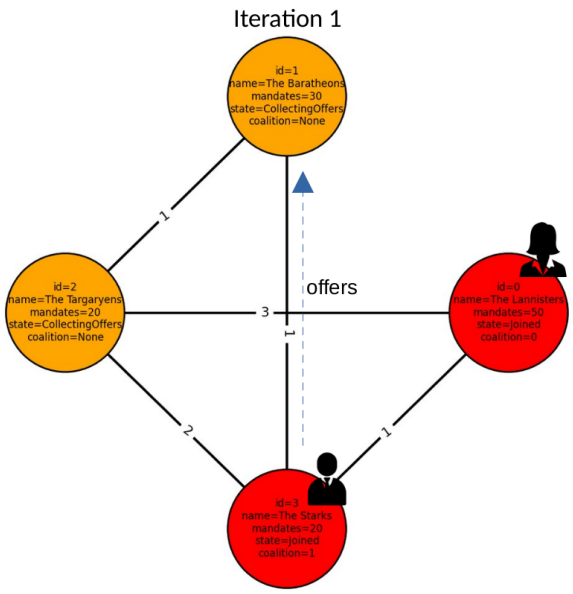
3.6 Full Example

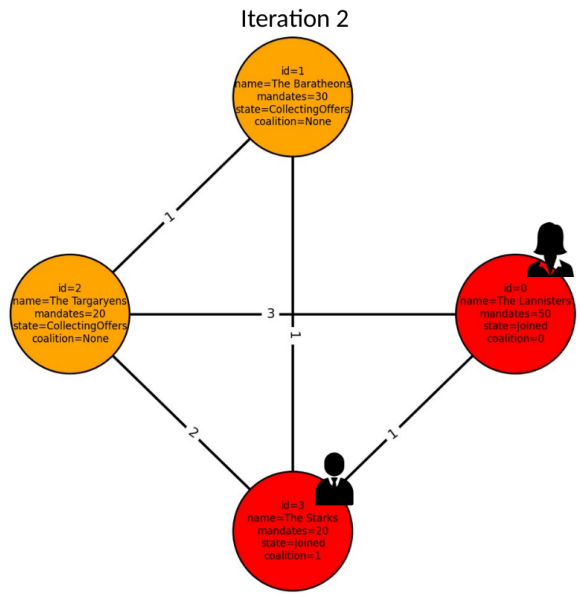
Now we will follow a full simulation example based on the "01.json" config file we supplied to you. Take a look at it.

- At the beginning of the simulation, it initialized with 4 parties, 2 agents, and 2 coalitions ({party#0}, {party#3}).
- At Iteration#0:
  - No parties in **CollectingOffers** state - pass.
  - Agent#0 selects from the set {party#2} using EdgeSelectionPolicy and offers party#2.
  - Agent#1 selects from the set {party#1, party#2} using EdgeSelectionPolicy and offers party#2.

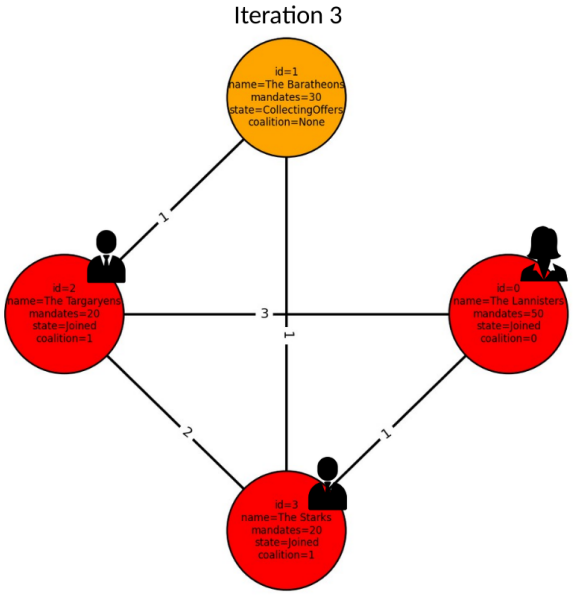


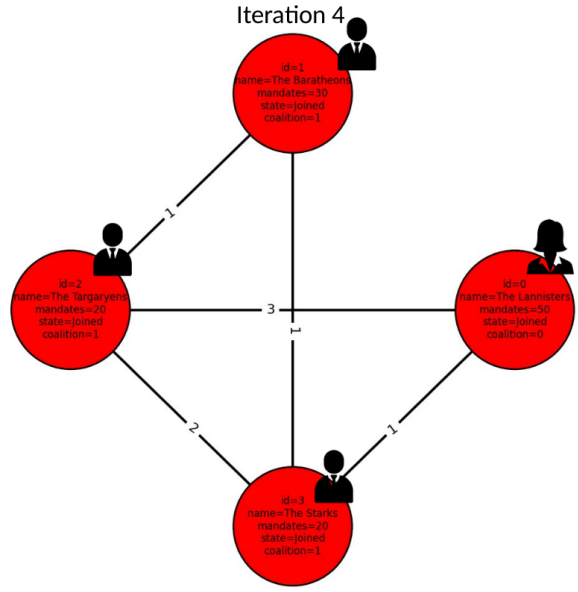
- At Iteration#1:
  - Party#2 in **CollectingOffers** state but the timer is not done - pass.
  - Agent#0 has no parties to select from - pass.
  - Agent#1 selects from the set {party#1} using EdgeSelectionPolicy and offers party#1.
- At Iteration#2:
  - Party#1 and Party#2 in **CollectingOffers** state but both timers are not done - pass.
  - Agents have no parties to select from - pass.





- At Iteration#3:
  - Party#1 in **CollectingOffers** state but timer not done - pass.
  - Party#2 in **CollectingOffers** state and timer is done - joining using LastOfferJoinPolicy to coalition#1 and clone agent#1.
  - Agents have no parties to select from - pass
- At Iteration#4:
  - Party#1 in **CollectingOffers** state and timer is done - joining using MandatesJoinPolicy to coalition#1 and clone agent#1.
  - Agents have no parties to select from - pass.





3.7 General instructions

- All classes with dynamic resources (and only them) **must implement the rule of five.**
- You are **NOT ALLOWED** to modify any of the supplied functions' signatures (the declaration). We will use these functions to test your code, attempting to change their declaration might cause a compilation error and a significant reduction in your grade. You are allowed to add additional header and cpp files as you see fit.
- You **must not** add any global variables to the program.
- Besides correctness, we will test your **code quality** and **memory management principles**. That means, for example, proper use of objects, pointers, references, and stack/heap allocation, as well as performance, clean and modular code.
- Make sure to read updates and clarifications on the assignment page.
- Ask questions regarding the assignment in the forum only.

4 Provided files

The following files will be provided for you on moodle:

- Headers: *Agent.h, Graph.h, JoinPolicy.h, json.hpp* (used for parsing the JSON config file)*, Parser.h, Party.h, SelectionPolicy.h, Simulation.h*
- Sources: *Agent.cpp, Graph.cpp, main.cpp, Parser.cpp, Party.cpp, Simulation.cpp*
- Basic makefile - you should extend it
- visualization.py (will be explained in section 5)
- Tests: 3 JSON files and their expected outputs (will be explained in section 6)

5 Visualization

To help you debug your simulation, we wrote for you a python script that can generate graph visualizations from the output file.

To run it, first install some packages by running:

```
sudo apt update
sudo apt-get install python3-dev python3-setuptools
sudo apt-get install libtiff5-dev libjpeg8-dev libopenjp2-7-dev zlib1g-dev \
    libfreetype6-dev liblcms2-dev libwebp-dev tcl8.6-dev tk8.6-dev python3-tk \
    libharfbuzz-dev libfribidi-dev libxcb1-dev
sudo apt-get install python3-pip
pip3 install networkx
pip3 install matplotlib
```

and then:

```
python3 visualization.py <path to output file>
```

6 Tests

We provided you 3 JSON files for different scenarios of the program, and their expected output. You can compare the expected output with your actual output using the Linux *diff* command:

```
diff -s <expected output file> <your output file>
```

This is the important part - you should have the exact same output as expected.

We encourage you to write additional tests yourselves to further examine the behavior of your program.

7 Submission

- Your submission should be in a single zip file called ”student1ID_student2ID.zip”. The files in the zip should be set in the following structure:
  - src/
  - include/

- bin/
- makefile

src/ directory includes all .cpp files that are used in the assignment.
include/ directory includes the header (*.h or *.hpp) files used in the assignment.
bin/ directory should be empty, no submitting of binary files. It will be used to place the compiled file when checking your work.

- The makefile should compile the cpp files into the bin/ folder and create an executable named ”cRace” and place it also in the bin/ folder.
- Your submission will be built (compile+link) by running the following commands: "make".
- Your submission will be tested by running your program with different scenarios.
- Your submission must compile **without warnings or errors** on the department computers.
- We will test your program using **Valgrind** in order to ensure no memory leaks have occurred. We will use the following valgrind command:
  valgrind --leak-check=full --show-reachable=yes [program-name] [program parameters]
  The expected valgrind output is:
  All heap blocks were freed -- no leaks are possible
  ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
- Compiler commands must include the following flags:
  -g -Wall -Weffc++ -std=c++11 -Iinclude
- After you submit your zip to Moodle, re-download the file that you have just submitted, extract the files and check that it compiles on the university labs. Failure to properly compile or run on the departments’ computers will result in a zero grade for the assignment.

GOOD LUCK AND ENJOY!