L3: Risky Strategies

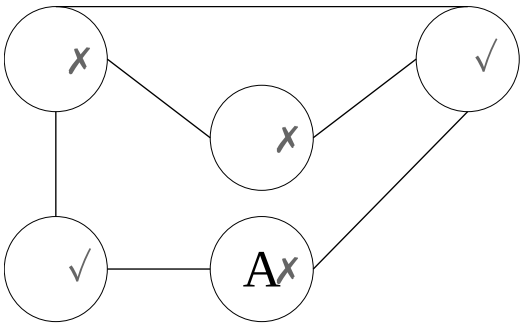
The Kingdoms of Alfagonia and Betastan have been waging very risky conflicts based on board games for centuries. The loser must wear a court jester's outfit, which is a significant blow to their reputation. The kings, however, face a problem – they have run out of games. A designer suspiciously resembling a famous bandit from the nearby forest has been recruited. The game has already been designed, and the task of implementing it rests on the best student of the best university of the seven kingdoms – You. The kings have also described their strategies and expect that they won't have to do anything, and the game will make moves for them.

The game board in *Risky Strategies* is a graph where each vertex can have an owner. Each player can at any moment capture a field on the board that neighbors at least one of their fields. This process takes `MOVE_MS` milliseconds and cannot be interrupted. After capturing a field, the player receives one point.

An identical **strategy** has formed in the kings' minds: choose a random field from the entire board and try to capture it. However, the game must follow the rules, so the board state does not change and the player does not gain a point when:

- They already own the chosen field,
- They do not own any field neighboring the chosen field.

An integral part of the strategy is also complaining – printing information why the move failed.



✓ - legal move for player 'A', X - illegal move for player 'A'

The board is represented by an adjacency list. The structure storing data about a given field is called `region_t` and is defined in the `risk.h` file along with a helper function that loads the board from disk as an array of structures. Ownership of a given field is marked with the symbol of the given player (`'A'` and in later stages `'B'`) or a minus sign `'-'`. It is possible that completing some stages will require defining additional data regarding a given field. These must be stored in a separate data structure. You can assume that in the loaded array, the neighbor indices of a given vertex are always sorted (e.g. `neighbors[0]=1, ...[1]=3, ...[2]=4`) and that vertices are numbered sequentially without gaps (from 0 to `num_regions`).

Stages

Stage 1 (5 pts.)

Load the board from the file provided in the first parameter of the program using the `load_regions` function. Set the character `'A'` as the owner character of a random field on the board and create a player thread `'A'` that will perform a **single** move according to the strategy described above. The player should print the reason why they failed to make a move. The main thread waits for the player thread to finish, then prints the board state and terminates. The board state consists of a list of lines containing information about subsequent fields. Each should contain the index in the main array, the highlighted owner character, and then the list of neighbors. E.g.:

```
0 [A] : 1;2
1 [-] : 0
2 [-] : 0;3
3 [-] : 2
```

Stage 2 (5 pts.)

Change the program's behavior so that the player makes moves one after another until they make `FRUSTRATION_LIMIT` illegal moves **in a row** – then they give up. The main thread prints the state of the entire board atomically every `SHOW_MS` milliseconds. If the player has given up, after printing the board, the main thread terminates the program.

Stage 3 (7 pts.)

The main thread places two players `'A'` and `'B'` on the board. Then it creates their threads and the players make moves simultaneously – in particular, it should be possible for two players to enter different fields at the same moment. If any player gives up, the main thread, after printing the board state, waits for both players to give up and their threads to finish, and then displays their points.

Ensure that the state of checked fields does not change during the execution of a move. Your code obviously cannot allow deadlocks to occur.

Stage 4 (5 pts.)

Add a thread handling signals. Upon receiving `SIGINT`, the program should turn a random field belonging to any player into a neutral one and subtract one point from them, and then print the board state. This process should be thread-safe. When `SIGTERM` is received, all threads except the main one are immediately terminated, information that Robin Hood wins is printed, and the program ends.

Starting code

- Makefile
- **maps/**
  - diamond.risk
  - ring.risk
  - square.risk
  - torus.risk
- risk.h
- sop-risk.c

Download as zip