Enunciado de PCD - 2022/23
Agar.pcd: jogo concorrente e distribuído

Luís Mota

Versão 5 *- 21 de novembro de 2022

Resumo

Pretende-se desenvolver como projeto de PCD um jogo distribuído, aqui designado Agar.pcd, uma versão simplificada do popular jogo online Agar.io[1]. Apelando ao caráter lúdico deste tipo de jogos, pretende-se assim motivar os alunos a dedicar o esforço necessário à aprendizagem dos conteúdos da cadeira.

Para bem servir os propósitos de aprendizagem, as características do jogo foram adaptadas, por forma a serem um bom cenário de aplicação de questões habituais no domínio da concorrência. Optou-se também por ter os adversários humanos a funcionar remotamente, para impor um caráter distribuído ao projeto.

1 Descrição genérica do projeto

O jogo Agar.pcd joga-se online, entre vários jogadores ligados remotamente. Cada jogador controla um elemento do jogo, representado por uma instância da classe `Player`. O objetivo de cada jogador é entrar em confronto com os outros jogadores. Como consequência de cada confronto, um jogador morre, sendo o seu nível de energia absorvido pelo vencedor. Existem também jogadores controlados automaticamente, que vão deambular pelo tabuleiro de forma aleatória, podendo entrar em confronto com qualquer outro jogador presente no tabuleiro.

*__Nota:__ o enunciado pode ser alterado para melhorar a clareza ou adequar-se melhor às necessidades de aprendizagem. Qualquer nova versão será publicada no moodle e será feita uma notificação por email.

[1] `https://pt.wikipedia.org/wiki/Agar.io`

O objetivo dos jogadores (humanos ou automáticos) é ir conquistando pontos de energia, até atingir o máximo de 10. Nessa altura, acabam a participação no jogo e ficam imóveis, sendo representados graficamente na GUI por um 'X'.

Os confrontos ocorrem quando um jogador pretenda mover-se para a posição ocupada por outro jogador. Os confrontos entre jogadores são sempre vencidos pelo jogador com mais energia. Apenas se o nível de energia for semelhante entre os dois jogadores é que o resultado será aleatório, com vitória equiprovável para ambos os jogadores. O vencedor do confronto absorve a energia do derrotado, tomando em conta o limite máximo da energia. O derrotado deixa de participar no jogo, transformando-se num obstáculo imóvel, representado por fundo amarelo: a posição ocupada deixa de poder ser percorrida por outros jogadores ainda a participar no jogo. Para evitar conflitos de posição, quando finalizado o confronto, ambos os jogadores manter-se-ão na posição p_n que ocupavam antes.

A movimentação dos jogadores automáticos deve ser absolutamente aleatória, não devendo, nomeadamente, evitar as posições com jogadores inativos, que funcionarão como obstáculos.

O jogo acaba com sucesso quando 3 jogadores tenham atingido o nível máximo de energia.

2 Requisitos do projeto

O projeto é uma ferramenta de aprendizagem, pelo que deve servir para adquirir conhecimentos em vários tópicos do programa de PCD, que passamos a elencar e que devem ser escrupulosamente seguidos.

Todos os mecanismos de coordenação devem ser desenvolvidos pelo próprio grupo, não devendo ser usados os equivalentes disponibilizados nas bibliotecas padrão do Java.

Colocação dos jogadores Aquando do início do jogo, os jogadores participantes, em número a definir em constante no programa, devem ser colocados numa posição aleatória. Caso a posição escolhida para um jogador já esteja ocupada por outro jogador colocado antes, a colocação do segundo jogador deve ser bloqueada, esperando que a posição fique livre para este ser efetivamente colocado em jogo. Esta situação de conflito deve ser claramente identificada numa mensagem na consola, indicando a posição, qual o jogador que já está a ocupar a posição e qual o que o se encontra impedido de o fazer.

Igualmente, a energia inicial e_n^i de cada jogador i deve ser aleatoriamente determinada entre 1 e 3. Para compensar esta vantagem, haverá uma li-

mitação em sentido contrário: cada jogador apenas se poderá deslocar uma vez em cada e_n^i ciclos do jogo: assim, por exemplo, um jogador t com $e_t^i = 1$ poderá deslocar-se em todos os ciclos, mas outro jogador q com $e_q^i = 3$ apenas se deslocará uma vez em cada 3. Estas limitações iniciais mantêm-se mesmo quando a energia subir após confrontos com outros jogadores: a limitação da movimentação apenas depende da energia inicial, e não da energia momentânea.

Sincronização de acesso às secções críticas Como sempre necessário em situações de acesso concorrente a informação partilhada, devem ser identificadas as secções críticas e devidamente protegidas de interferências, usando o mecanismo de sincronização, que assegurará a exclusão mútua de acessos por diferentes *threads*. A sincronização deve, naturalmente, ser tão localizada quanto possível, para permitir o máximo de concorrência.

Implementação dos confrontos É necessário um jogador pretender mover-se para a célula ocupada por outro para poder existir um confronto, havendo como consequência alterações na energia e no estado dos jogadores. A localização dos jogadores permanece inalterada, independentemente de quem saia vitorioso do confronto.

Coordenação para o fim do jogo Como descrito na secção anterior, o jogo termina quando houver 3 jogadores com o nível máximo de energia. Quando tal acontecer,o jogo termina para todos os participantes ainda ativos. Para implementar esta coordenação deve ser usado uma ferramenta de coordenação temporal, conforme abordado no laboratório da terceira semana sobre coordenação. Existe nesse laboratório uma chamada de atenção para encaminhar esta questão.

Resolução de imobilização Os jogadores automáticos, na sua movimentação automática, poderão tentar mover-se para uma posição ocupada por um jogador inativo, que se comporta como obstáculo. Estas movimentações não devem ser ser possíveis, pelo que o jogador ficará bloqueado. Para evitar que este movimento fique bloqueado para sempre, deve ser feito um mecanismo, com ajuda de *thread* autónoma, que, após 2 segundos, interrompa o jogador e o seu movimento bloqueado, permitindo consequentemente ao jogador automático voltar a escolher um novo movimento que, possivelmente, já não bloqueará.

O mesmo raciocínio deve ser aplicado à colocação inicial: embora pouco provável, pode acontecer que um jogador fique à espera de ocupar a célula

onde se encontra outro jogador que morra logo no início do jogo. Neste caso, o jogador em espera nunca seria colocado no jogo.

Ligação remota dos jogadores No sentido de aplicar os conceitos de programação distribuída, devem os jogadores humanos ser implementados como aplicações remotas, que se ligam a um servidor existente, associado ao jogo. Cada jogador remoto deve ser lançado e controlado por uma aplicação diferente e autónoma, que se pode juntar ao jogo a qualquer momento.

A estratégia de comunicação a seguir consiste em o servidor enviar ciclicamente a todos os clientes remotos o estado do jogo. Do outro lado, o cliente remoto apenas enviaria a direção escolhida pelo utilizador quando este de facto premir as teclas correspondentes. Não haverá, assim, nenhuma relação clara entre a comunicação nos dois sentidos. Por isso, o envio e receção da informação deve ser considerado independente e, consequentemente, deixado à responsabilidade de *threads* autónomas.

A comunicação entre o servidor e estes clientes deve, no sentido servidor-cliente usar canais de objetos, e, no sentido oposto, canais simples de texto.

3 Detalhes de implementação

Para poder focar o desenvolvimento nas matérias relevantes de PCD, a interface gráfica já é fornecida, com todas as capacidades para representar o jogo em todos os seus momentos.

Sempre que, durante o jogo, se proceda a alguma alteração, como p.ex. a energia ou posição de um jogador, deve ser invocado o método `Game.notifyChange()`, que assegurará que a interface gráfica é atualizada.

Para assegurar a consistência dos dados, sugere-se fortemente que a localização dos jogadores seja mantida **em exclusivo** nas células mantidas no atributo `board` da classe `Game`.

A temporização dos movimentos dos jogadores não deve ser centralizada: todos deverão respeitar um tempo de espera fixo, definido na constante `Game.REFRESH_INTERVAL`. Assim, após efetuar o seu movimento, cada jogador deve adormecer por este período de tempo.

Para permitir a ligação de jogadores remotos, o início dos movimentos do jogo apenas deve ocorrer cerca de 10 segundos depois de este ser lançado: as threads dos jogadores tentarão colocar imediatamente o jogador a posição inicial, mas apenas iniciarão a sua movimentação normal após decorrido este tempo. Sugere-se que se coloque um `sleep` no método `run` logo após a colocação na posição inicial.

4 Fases de desenvolvimento

Nesta secção sugere-se um encadeamento de fases que permitem um desenvolvimento sustentado deste projeto. As primeiras duas fases devem imperiosamente ser cumpridas no início, pois vão constituir a entrega intercalar.

1. **Colocação inicial dos jogadores:** tratar da colocação inicial dos jogadores automáticos, segundo as indicações anteriores. Recomenda-se, como oportunidade extra de aprendizagem, a utilização de variáveis condicionais.

   Inicialmente, pode ser ignorada a possibilidade de um jogador estar à espera de uma posição ocupada por um jogador que tenha já morrido, mas esses casos devem ser considerados nas versões subsequentes. A estratégia para resolver esse bloqueio deve desejavelmente ser diferente da que usar na fase 4 e deve ser feita uma descrição da estratégia escolhida, em relatório autónomo a incluir na entrega final.

2. **Movimentação básica dos jogadores automáticos:** Os jogadores devem realizar a sua deslocação apenas a intervalos regulares. Por isso, crie una classe para os jogadores automáticos, que devem poder funcionar enquanto processo ligeiro (*Thread*). A sua execução deve conter uma repetição regular do envio para o jogo do pedido de movimentação numa direção aleatória. O intervalo entre repetições do envio deve ser de acordo com a constante `Game.REFRESH_INTERVAL`, que está inicialmente configurada para 400ms. Nesta fase não considere obstáculos nem outras limitações de movimentos. Sempre que for feito um movimento,deve ser desencadeada a atualização da interface gráfica, invocando o método `notifyChange`.

3. **Movimentação completa dos jogadores:** desenvolva a fase anterior, considerando as possíveis consequências dos movimentos:

   (a) se o movimento for para cima de outro jogador, será disputado um confronto segundo as regras enunciadas acima;

   (b) se o movimento for para cima de um jogador morto, o movimento deve ficar bloqueado;

   (c) se o jogador atingir o nível máximo de energia, deve acabar a sua execução e registar este facto no estado do jogo.

4. **Resolução da imobilização no movimento:** como descrito anteriormente, se um jogador automático tentar deslocar-se para cima de

um jogador inativo, este fará o papel de obstáculo, pelo que o movimento não será possível. Neste caso, o jogador deve bloquear, usando as habituais operações de `wait` ou `await`, conforme estejam a usar cadeados implícitos ou explícitos. Como o jogador inativo nunca sairá desta posição, é necessário encontrar uma forma de desbloquear o movimento suspenso. Para tal, deve ser aplicada a técnica descrita na secção 2.

5. **Final do jogo:** como descrito na secção anterior, também é necessária coordenação, através de uma barreira ou objeto de coordenação similar, para detetar o final do jogo, conforme detalhadamente descrito acima. O mecanismo de coordenação utilizado deve ser implementado propositadamente para este projeto, não devendo ser usada o equivalente existente nas bibliotecas padrão do Java.

6. **Implementação dos jogadores humanos como aplicações remotas:** para aprofundar os conhecimentos de programação distribuída, deve, nesta última fase, ser desenvolvida uma versão remota dos jogadores, que serão controlados por um humano. Estes ligar-se-ão ao jogo através de um servidor que estará disponível num endereço e porto conhecidos. Note-se que nada impede que estes jogadores remotos coexistam no jogo com os jogadores automáticos referidos no ponto 3.

   Em termos de comunicação, devem ser respeitadas as seguintes indicações:

   - O estado do jogo deve ser enviado regularmente a todos os jogadores remotos ligados: sugere-se que se use o valor em `Game.REFRESH_INTERVAL`. Este envio deve ser feito independentemente do funcionamento do jogo, por processo ligeiro autónomo.
   - Para facilitar o desempenho destes jogadores, sugere-se que seja, inicializados com o valor 5 para a energia.
   - Para evitar sobrecarregar os canais de comunicação, sugere-se que seja enviada apenas a informação estritamente necessária: pode ser criada uma classe apenas com este propósito.
   - Os jogadores humanos não bloquearão se forem encaminhados para uma célula ocupada: ignorarão apenas o movimento.
   - Os clientes remotos devem receber 6 argumentos de execução: endereço e porto da aplicação principal, teclas para movimentar nas quatro direções. Assim poderão ser configurados dois jogadores remotos diferentes a usar o mesmo teclado.

Para ajudar o processamento do pressionamento de teclas, a classe `BoardJComponent` implementa `KeyListener` e já tem dois métodos para lidar com esta capacidade: `Direction getLastPressedDirection()` e `void clearLastPressedDirection()`. Desta forma é possível consultar que tecla foi premida recentemente e reiniciar essa informação. Sugere-se a utilização desta classe `BoardJComponent` no cliente remoto.

7. **Deteção de possíveis situações de bloqueio e conflito** Analise a sua resolução para o projeto e tente identificar possíveis situações de bloqueio, como p.ex. *deadlock*, *livelock* ou *starvation*. Caso identifique riscos de ocorrência destas situações, tente aplicar medidas que as impeçam. Deve ser feita uma descrição das medidas tomadas, em relatório autónomo a incluir na entrega final.

5 Entregas

Para favorecer o início atempado do desenvolvimento do projeto, existe uma entrega intermédia. Pede-se que seja feita uma apresentação das duas fases iniciais no turno de laboratório da semana de 14 a 18 de novembro, onde deverá ser feita uma breve apresentação do trabalho feito. Esta entrega é obrigatória e eliminatória, e feita presencialmente.

A entrega final será às 09:00 de 12.12.2022, em página no moodle a disponibilizar oportunamente. As discussões finais, de que alguns alunos poderão eventualmente ser dispensados, decorrerão nos laboratórios de 13, 14 e 15 de dezembro e, se necessário, estender-se-ão eventualmente para 19 e 20.12.2022.

O modelo da avaliação é o habitual nas cadeiras de programação: o projeto é obrigatório mas não conta diretamente para a nota: pode apenas limitar a nota final, conforme critérios apresentados na primeira aula e constante da FUC.

6 Avaliação

Todas as fases descritas na secção 4 devem ser implementadas, pois o projeto é primordialmente uma ferramenta de aprendizagem e essas fases correspondem a partes importantes da matéria a aprender e a ser avaliada.

Caso haja lacunas menores nesta implementação, os alunos poderão mesmo assim ter aprovação com uma valoração qualitativa mais baixa.

7 Histórico de versões

1 Versão original

2 Enumeração e descrição das fases de desenvolvimento

3 Alteração da arquitetura aconselhada para a comunicação remota

4 Correção de gralhas e acrescento de uma última fase de desenvolvimento, relativamente a situações de bloqueio.

5 Esclarecimento de passagens e acrescento de descrição do procedimento para deteção das teclas