Enunciado de PCD - 2023/24 - 2º Semestre
Snakes: jogo concorrente e distribuído

Luís Mota

Versão 1 *- 13 de março de 2024

Resumo

Pretende-se desenvolver como projeto de PCD um jogo distribuído, aqui designado *Snakes*, uma versão simplificada (e no plural) do popular jogo *Snake*[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, nesta versão do jogo, não ter jogadores humanos.

1 Descrição genérica do projeto

1.1 1ª parte: concorrência e coordenação

O jogo *Snakes* joga-se entre cobras controladas automaticamente. Cada cobra no tabuleiro corresponde a uma instância da classe `Snake`. O objetivo de cada cobra é capturar um prémio existente no tabuleiro, representado por um algarismo entre 1 e 9, que é o seu valor. Como consequência da captura do prémio, a cobra vai acrescentar ao seu comprimento atual o valor do prémio capturado. O crescimento da cobra ocorre gradualmente, uma posição por vez, a partir da cabeça da cobra, a cada novo movimento. As cobras são controladas automaticamente, e vão deambular pelo tabuleiro de

*__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://en.wikipedia.org/wiki/Snake_(video_game_genre)`

forma a dirigir-se pelo caminho mais curto para o prémio, evitando apenas as posições ocupadas pela própria cobra. O objetivo das cobras é ir conquistando prémios, até uma capturar o valor máximo de 9. Nessa altura, acaba o jogo e todas as cobras ficam imóveis na posição que ocupavam.

Naturalmente, no movimento, uma cobra pode tentar ocupar uma posição já ocupada por outra cobra, ou por um obstáculo (descrito mais adiante). Neste caso, ao contrário do habitual em jogos deste tipo, a cobra não morrerá e a ocupação dessa posição deve ficar em espera, até que a posição seja desocupada. A movimentação dos jogadores automáticos deve orientar-se para a posição atual do prémio, não devendo, em particular, evitar as posições com outros jogadores ou obstáculos. O movimento deve, porém, evitar células ocupadas pela própria cobra.

Existirão também obstáculos no tabuleiro, colocados aleatoriamente no início do jogo, em posições que estejam livres. Estes obstáculos serão manipulados por *threads* próprias: cada obstáculo terá uma *thread* associada, instância da classe `ObstacleMover`. Estas *threads associadas* não poderão estar todas em execução simultânea: apenas poderão estar em execução a cada momento `NUM_SIMULTANEOUS_MOVING_OBSTACLES`, p.ex. no número de três. Esta limitação à execução deve ser conseguida com auxílio de uma `ThreadPool`. A ação dos `ObstacleMovers` é muito simples: após um curto período de espera (`Obstacle.OBSTACLE_MOVE_INTERVAL`), a localização do obstáculo deve ser alterada para uma posição livre do tabuleiro, escolhida aleatoriamente. Este reposicionamento do obstáculo será repetido `Obstacle.NUM_MOVES` vezes, após o que o `ObstacleMover` terminará. Se uma cobra tentar deslocar-se para a posição onde se encontre um obstáculo, o movimento não poderá prosseguir e a cobra ficará bloqueada. Este debloqueio poderá acabar se o obstáculo se deslocar para outra localização, ou for apagado (o que será descrito mais à frente).

À medida que os `ObstacleMovers` forem terminando a sua execução, devem ser introduzidos novos elementos de jogo, designado *Killers*. Esta criação de *Killers* será coordenada com ajuda de uma barreira, a implementar, da classe `CyclicBarrier`. Imediatamente antes de um `ObstacleMover` terminas a sua execução, deve invocar o método `await` da barreira. Por cada `CyclicBarrier.NUMBER_WAITERS` vezes que for invocado o método `await`, deve ser criada e introduzida no jogo uma nova instância de *Killer*, através da designada `BarrierAction` da barreira, descrita na teórica correspondente.

As instâncias de `Killer` são elementos passivos do jogo: quando criados são posicionados numa célula livre aleatória. Se uma cobra se dirigir a uma célula ocupada por um `Killer`, será considerada morta e deixa de se deslocar.

Os prémios, representados pela classe `Goal` serão manipulados pelas cobras, que se encarregarão de relocalizar o prémio e incrementar o seu valor

de cada vez que o prémio for capturado, o que será provocado pela invocação do método `Goal.captureGoal` pela cobra que capturar o prémio.

Dado o movimento relativamente pouco inteligente das cobras automáticas, é bastante provável que estas fiquem bloqueadas em obstáculos ou noutras cobras, sobretudo quando estes existirem em número elevado. Para libertar esses bloqueios, será implementado um mecanismo na 2ª parte do projeto.

1.2 2ª parte: Programação em rede

A 2ª parte apenas deve ser desenvolvida após a conclusão da primeira, depois da apresentação intermédia. Serão então criados intervenientes no jogo remotos, que se ligarão em qualquer momento a um jogo já existente,e servirão para resolver as situações de bloqueio do jogo. Estes clientes remotos serão independentes e poderão coexistir em qualquer número.

A lógica de funcionamento dos clientes remotos é muito simples:

1. geração repetida de coordenadas aleatórias, que serão enviadas ao servidor através de canais de texto;

2. receção da resposta do servidor através de uma uma instância da classe `ActionResult`.

Estes ligar-se-ão ao jogo através de um servidor que estará disponível num endereço e porto conhecidos.

O servidor, pelo seu turno, deve analisar as coordenadas recebidas e verificar se nelas existe algum elemento do jogo. Caso lá esteja posicionado um obstáculo ou uma cobra morta, o elemento deve ser apagado do jogo, libertando a(s) célula(s) correspondente(s). Caso a célula esteja vazia ou ocupada por uma cobra ativa ou um `Killer`, nada acontece. Conforme o caso, deverá ser enviada uma instância de `ActionResult`, que terá os campos `boolean wasSuccessful`, indicando se algum elemento do jogo foi efetivamente apagado, e `boolean gameEnded`, indicando que o jogo acabou. Neste último caso, o cliente remoto deve consequentemente terminar a sua execução.

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 disponibiliza-

dos nas bibliotecas padrão do Java. No entanto, pode ser usada a `ThreadPool` 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 na primeira coluna da esquerda, ocupando uma única célula. Caso a posição escolhida para um jogador já esteja ocupada por outro jogador colocado antes, a escolha da posição do novo jogador deve ser repetida até ser encontrada uma célula livre nessa primeira coluna.

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 ou equivalente, 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. Deve ser dada particular atenção ao movimento das cobras: estes envolvem as células de origem e destino. A solução ideal deve aplicar exclusão mútua a ambas as células, evitando, naturalmente, situações de bloqueio (*deadlock* ou equivalente).

Coordenação para a ocupação de células Uma cobra pode pretender ocupar uma célula transitoriamente ocupada por outra cobra ou por um obstáculo. Deve ser implementado um mecanismo de coordenação que coloque esta cobra em espera, aguardando que a célula destino fique livre.

Aplicação de `ThreadPool` Os obstáculos devem ser movimentados por `threads` dedicadas: cada obstáculo deve ter um `thread` associada, da classe `ObstacleMover`. A execução destas `threads` deve ser gerida por uma `ThreadPool`: em cada momento apenas devem estar em execução simultânea um número de `threads` definido em `LocalBoard.NUM_SIMULTANEOUS_MOVING_OBSTACLES`.

Coordenação para o fim do jogo Como descrito na secção anterior, o jogo termina quando uma cobra capture o prémio com o valor 9. Quando tal acontecer,o jogo termina para todos os participantes. Para implementar esta coordenação, devem ser interrompidas todas as cobras em execução: se o jogo de facto tiver terminado, a sua execução deve terminar

Introdução dos *Killers* : por cada 3 obstáculos que acabem a sua movimentação, deve ser inserido no jogo uma instância de `Killer`, conforme descrito anteriormente, com auxílio de uma *Barreira*.

Ligação remota dos clientes As cobras, na sua movimentação, poderão tentar mover-se para uma posição ocupada, quer por outra cobra quer por um obstáculo. Por vezes, estes bloqueios serão irresolúveis, por serem originados por um obstáculo que já não se vai mover, ou por outra cobra também ela bloqueada. Para evitar que o jogo fique bloqueado para sempre, deve ser feito um mecanismo, desencadeado por clientes remotos, que ciclicamente apague obstáculos e cobras inativas..

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. Também é dada a implementação, completa ou em esqueleto conforme os casos, de quase todas as classes necessárias ao desenvolvimento do projeto.

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 `Board.setChanged()`, que assegurará que a interface gráfica é atualizada.

Para assegurar a consistência dos dados, sugere-se fortemente que a localização dos jogadores seja mantida **preferencialmente** nas células mantidas no atributo `cells` da classe `Board`.

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

4 Fases de desenvolvimento

Nesta secção sugere-se um encadeamento de fases que permitem um desenvolvimento sustentado deste projeto. As primeiras sete fases devem idealmente ser cumpridas no início, pois vão constituir a entrega intercalar. Não é obrigatório ter todas estas fases concluídas nessa altura, mas é desejável.

1. **Colocação inicial dos jogadores:** colocação inicial dos jogadores automáticos numa célula livre da primeira coluna.

2. **Movimentação básica dos jogadores automáticos:** Os jogadores devem realizar a sua deslocação apenas a intervalos regulares. Por isso, use a classe `AutomaticSnake` 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. O intervalo entre repetições do envio deve ser de acordo com a constante `Board.PLAYER_PLAY_INTERVAL`, que está inicialmente configurada para 100ms. 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 `setChanged()`.

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 ou obstáculo, deverá ficar em espera segundo as regras enunciadas acima. Recomenda-se, como oportunidade extra de aprendizagem, a utilização de variáveis condicionais:

   (b) se o jogador atingir o prémio máximo, deve acabar a sua execução e registar este facto no estado do jogo.

4. **Final do jogo:** como descrito na secção anterior, também é necessária coordenação, através de uma interrupção, para processar o final do jogo, conforme detalhadamente descrito acima.

5. **Análise 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.

6. **Gestão dos movimentos dos obstáculos**: Aplicação da `ThreadPool` segundo indicações acima.

7. **Resolução da imobilização no movimento por aplicações remotas:** como descrito anteriormente, se um jogador automático tentar deslocar-se para cima de um 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 obstáculo poderá nunca sair desta

posição, é necessário encontrar uma forma de desbloquear o movimento suspenso. Para tal, deve ser aplicada a técnica descrita na secção 2. Para aprofundar os conhecimentos de programação distribuída, deve esta técnica de desbloqueio, nesta última fase, ser desenvolvida enquanto aplicação remota

5 Entregas

Nota: as datas propostas abaixo ainda não estão validadas pelo conselho de ano, pelo que são provisórias e poderão sofrer alterações.

Para favorecer o início atempado do desenvolvimento do projeto, existe uma discussão intermédia. Pede-se que seja feita uma apresentação das fases iniciais, referentes à versão concorrente, no turno de laboratório de a 24 de abril, onde deverá ser feita uma breve apresentação do trabalho feito. Esta apresentação é obrigatória e eliminatória, e feita preferencialmente pelo Zoom. Não é necessário nesta fase fazer qualquer entrega formal do código.

A entrega final será às 09:00 de 20.05.2024, em página a disponibilizar oportunamente no Moodle. As discussões finais, de que alguns alunos poderão eventualmente ser dispensados, decorrerão nos laboratórios de 22 de maio, e se necessário, estender-se-ão eventualmente para os dias seguintes.

O modelo da avaliação é o habitual nas cadeiras de programação: o projeto é obrigatório mas não conta diretamente para a nota, podendo 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