Computação Escalável
Relatório A1

INTEGRANTES:

Caio Lins
João Pedro Donasolo
João Vinicius
Tomás Ferranti

<u>AVISO:</u> Os detalhes de instalação e execução se encontram no arquivo "README.md". Este relatório foca nos detalhes da modelagem e da implementação da simulação em C++.

Modelagem Geral

Como o escopo do que fazer é relativamente livre para a simulação em geral, seguem abaixo as diferenças com relação ao enunciado original:

1. Os formigueiros possuem parâmetros locais de número de formigas, mas usam alguns globais que são compartilhados entre todas as formigas da simulação, como campo de visão e duração do feromônio.
2. Cada formigueiro possui um número fixo de formigas geradas em sua posição. Essas formigas entregam comida apenas para seu formigueiro de nascença.
3. Há apenas um tipo de feromônio: o deixado por formigas após encontrarem comida. Com isso, após adquirir a comida, estas seguem um caminho determinístico de volta ao seu formigueiro.
4. O campo de visão da formiga é um retângulo de dimensões $2r \times r$ para os lados e frente, seguindo a orientação de para onde ela está olhando. O parâmetro r é configurável. Ela consegue enxergar feromônios e comidas disponíveis.
5. Formigas movem de forma aleatória até encontrarem um rastro ou uma fonte de comida disponível. Este deslocamento ocorre apenas para espaços vizinhos de onde ela está, andando apenas um de cada vez. Ela não anda na diagonal.
6. O ataque entre formigas de diferentes formigueiros não foi implementado.

Principais Decisões do Projeto

As principais decisões do projeto como consequências da modelagem citada acima foram as seguintes:

1. O projeto foi desenvolvido seguindo os princípios da OOP.
2. A simulação vai ser realizada seguindo o processo de uma iteração de cada vez, onde cada entidade pode realizar apenas uma ação a cada iteração.
3. Teremos que atualizar três tipos de objetos ao longo de uma nova iteração do nosso mundo. Estes são todos os feromônios, todas as formigas e todas as fontes de comida, nesta respectiva ordem, devido a interação entre eles.

As especificações detalhadas do código estão presentes na próxima seção.

Detalhes de Implementação

A simulação está inteiramente contida na classe Mundo (**World**). Esta contém todos os elementos da simulação e serve de interface principal do programa e a ela são atribuídas três responsabilidades:

1. **Setup**: Lê os parâmetros da simulação do arquivo *config.json*, inicializa e posiciona as entidades da simulação no mapa.
2. **Update**: Cria a quantidade de threads especificado e chama o método **.update()** de cada entidade.
3. **Draw:** Imprime o estado atual da simulação na tela.

Todas as demais classes são filhas da classe **Entity**, e, dentre essas, todas as que modificam ativamente o estado da simulação possuem o método **update**. São, nominalmente, Formigas (**Ants**), Feromônios (**Pheromones**) e Fontes de Comida (**Food**).

Threads e Problemas de Concorrência

Para manter o sincronismo ao longo da simulação, foi definido que o processamento se daria em partes: Primeiro feromônios, em seguida formigas e, por fim, fontes de comida. Para cada uma dessas três etapas, as threads são criadas, executadas, e finalizadas.

Para evitar concorrência, a classe **FlowController** determina, de forma segura, qual será a próxima entidade a ser processada para cada thread. Para o caso de a entidade sendo processada não alterar o ambiente além de si mesma, não há maiores problemas. Caso isso ocorra, como por exemplo quando um feromônio deve ser excluído (acesso concorrente à quantidade de feromônios no tile em questão), foram aplicados mutexes.

Concorrência na fonte de comida

Quando uma formiga (controlada por uma thread) entra em um Tile em que há uma fonte de comida com comida disponível, ela verifica se há lugar para ela coletar alimento. Ela faz isso por meio de um **try_acquire()** (executado na função **hasSeat**) em um semáforo membro da fonte de comida. Caso o número de threads tentando coletar alimento seja maior do que o número de bastões disponíveis (que é o valor máximo do semáforo), a formiga não consegue o **acquire()** e fica de guarda.

Caso a formiga consiga o **acquire()**, significa que há espaço para ela coletar alimento, ou seja, há bastões disponíveis. Em seguida, ela tenta encontrar um “lugar

para se sentar”, como se a fonte de comida fosse um restaurante. Ela itera sobre um array de mutexes, membro da fonte de comida, que simboliza os assentos existentes, até conseguir dar lock em algum deles. Ela, então, armazena o índice do lugar onde conseguiu se sentar, pois ele é relevante para o problema a seguir.

Para comer, a formiga necessita de obter o controle de dois bastões, sendo que, se ela se sentou no lugar de índice **i**, ela só têm acesso aos bastões localizados nas posições **i-1** e **i+1** (módulo o número de assentos) do array de bastões (representados por mutexes). Além disso, ela deve pegar um bastão de cada vez. Felizmente, esse é exatamente o problema dos filósofos, cuja solução já conhecemos! Nós implementamos a solução apresentada em sala de aula, mantendo os nomes das funções para facilitar sua identificação.

Tendo conseguido acesso aos bastões, a formiga então come, o que significa decrementar o contador de comidas disponíveis na fonte (o qual é protegido por um mutex). Por fim, ela abre mão dos recursos que estava consumindo, ou seja: os bastões, o lugar em que estava sentada e o semáforo que permite a entrada de mais formigas na fonte.