Modelagem em multithread da interação entre colônias de formiga

Germano Andrade, João Alcindo e Tiago da Silva

14 de Abril de 2022

> This will sound to you like it's a relatively new phenomenon on your planet, but it's not. Even pelagibacter transmit information, if only to daughter cells. Ants spray pheromones, bees dance, birds sing—all of these are comparatively low-bandwidth systems for communication. But your system caused an inflection point. The graph of data flow switched from linear to exponential growth.
>
> *A Beautifully Foolish Endeavor*,
> Hank Green.

Conteúdo

1 Introdução

Neste documento, vamos descrever os aspectos que enformaram nossas implementações e caracterizaram as nossas decisões de modelagem; contemplaremos, além disso, situações tremendamente atribuladas que caracterizaram tanto a implementação serial – em uma thread – quanto a multithread deste sistema. Na seção seguinte, portanto, introduzimos os atributos gerais de nosso programa, enfatizando como os agentes – as formigas, os formigueiros, as comidas e os objetos subjacentes, como os mapas e as coordenadas – foram desenhados, e apontando, também, para a contemplação dos mecanismos que ensejam sua interação.

Na seção subsequente, vislumbramos os alicereces que culminaram na versão mulithread do programa, explicitando a utilização de variáveis de exclusão mútua, de variáveis de condição e de semáforos com o objetivo de lograr as idiossincrasias inconvenientes das programação paralela, como as condições de corrida e a inanição (*starvation*).

2 Desenho dos agentes

A simulação da interaçao entre formigueiros contempla, neste cenário, múltiplos agentes; em um momento inicial, portanto, é importante que os caracterizemos, garantindo que eles possam interagir consistentemente durante a simulação. Em nossa implementação, em particular, identificamos cada agente com uma classe; alguns gozam de múltiplas instâncias (como o agente `Ant`, formiga), e outros, não (como o agente `Map`, mapa). Explicitamente, introduzimos as classes

1. `Anthill`, formigueiro,
2. `Ant`, formiga,
3. `Food`, comida,
4. `Map`, mapa e
5. `Tile`, azulejo (ou, equivalentemente, coordenada);

elas estão, assim, descritas na Figura 1. Perceba, logo, que as formigas estão amarradas ao formigueiro e, além disso, interagem com ele, incrementando o seu armazém de alimento; em contraste, a interação entre as formigas e a comida consiste no decremento de um atributo – o volume da instância de comida; por outro lado, as formigas não modificam, objetivamente, os atributos das coordenadas do mapa – eles que, na verdade, rastreiam, em uma tabela hash (em que as chaves correspondem aos nomes dos formigueiros) de pilhas, as formigas depositadas neles em cada iteração. Esta é, aliás, a interação mais crucial da simulação: as formigas se movem pelo mapa, identificam a comida, a capturam e, em um deslocamento para o formigueiro, incrementam o seu armazenamento de alimento. Nestas condições, o movimento das formigas é delicado; ele é, desta maneira, o tópico da seção seguinte.

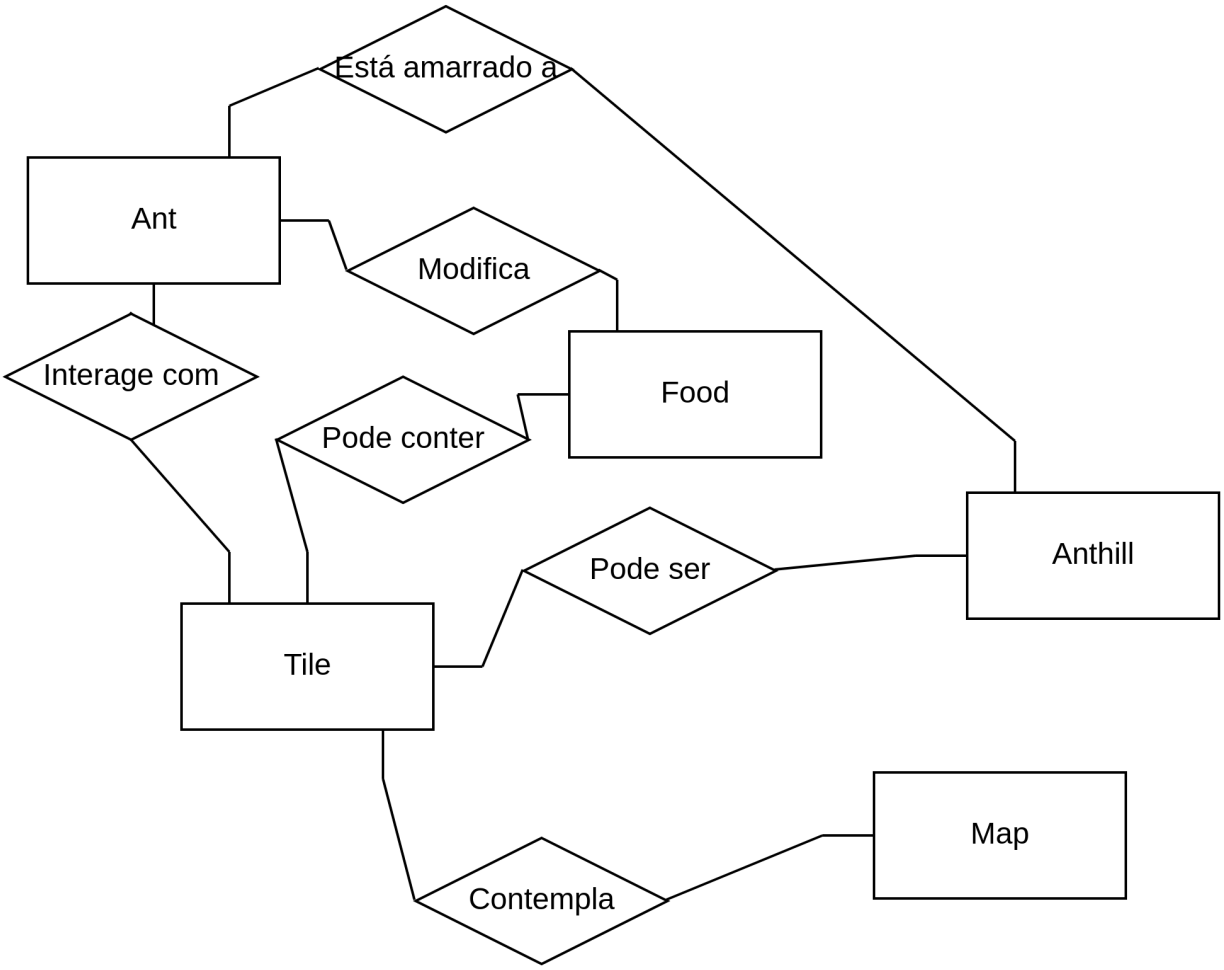

Figura 1: Modelagem e interação dos agentes na simulação.

2.1 Movimento das formigas

Em cada iteração, há incisivamente um tripleto de ações que as formigas podem executar: ou elas se movem aleatoriamente (método `moveRandomly`), ou elas se direcionam ao formigueiro (método `moveToColony`), ou elas procuram o alimento (método `moveToFood`; elas podem, também, entrar em combate; descreveremos, mais tarde, nossa aproximação para isso). Vislumbramos, nos itens seguintes, cada um desses movimentos.

1. `moveRandomly`. No movimento aleatório, identificamos as coordenadas em que a formiga está e, em seguida, capturamos seus vizinhos[1], verificando, importantemente, a sua existência (com esse objetivo, o método `getTile`, da classe `Map`, levanta uma exceção `BorderError`, sinalizando que o programa está acessando uma coordenada que transcende as bordas do mapa; o tratamento desta exceção, assim, garante a consistência da identificação das coordenadas vizinhas). Escolhemos, então, uma coordenada aleatoriamente; a probabilidade de uma instância ser escolhida é diretamente proporcional à quantidade de feromônio que ela contém[2] (a liberação de feromônios é contemplada na seção seguinte).

2. `moveToColony`. Equipadas com alimento, as formigas se direcionam ao formigueiro – elas têm a informação de sua localização. Elas executam, para isso, um movimento retilíneo, implementado como uma adaptação do algoritmo de Breseham: inicialmente, traçamos, no plano cartersiano, o segmento que amarra a coordenada atual da formiga à de seu formigueiro; na etapa subsequente, computamos a coordenada vizinha da formiga mais próxima deste segmento, e a movemos nesta direção.

3. `moveToFood`. As formigas envisionam, em seu movimento, coordenadas a uma distância caracterizada pelo atributo – do mapa; todas as formigas, em todos os formigueiros, gozam de campos de visão idênticos (contudo, poderíamos modificar a função que identifica instâncias de `Food` nas vizinhanças e distinguir a busca por comida para cada formiga – escolhemos, neste sentido, o desenho mais parcimonioso) –; portanto, elas podem verificar se há alimentos e executar um movimento informado. Neste caso, com as restrições de acesso simultâneo aos alimentos, este "movimento informado"pode, com efeito, consistir em aguardar e, possivelmente, lutar.

Em cada iteração da simulação, assim, cada formiga executa um movimento condicional ao seu estado atual: se ela estiver equipada com comida, ela, incondicionalmente ao seu ambiente, ao formigueiro; se não, ela escaneia o ambiente que a entorna e verifica se há

[1] Que não contêm alimento; as formigas não interagem com estes azulejos – elas modificam as instâncias da classe `Food` depositadas neles.

[2] Encetamos, para isso, gerando uma variável aleatória uniformemente distribuída no intervalo unitário da reta real; instanciamos, também, uma `array` com as somas acumuladas das quantidades de feromônios (mais um; por exemplo, um objeto com um feromônio teria peso igual a dois, e um com dois, igual a três) em cada coordenada, `cumSum`. Aplicamos, então, uma divisão de cada elemento de `cumSum` pela quantidade total de fermônios nas vizinhanças, e verificamos, assim, o intervalo correspondente à variável aleatória uniforme, que equivale à nossa escolha aleatória.

alimentos – se houver, ela se direciona a eles e, se não, ela se movimenta aleatoriamente. Em algumas circunstâncias, ela escolhe lutar; postergamos a descrição desta ação, contudo, para as seções seguintes. Na seção seguinte, caracterizamos o movimento de formigas com comida e a introdução de feromônios.

2.2 Liberação de feromônios

Quando capturam alimentos, as formigas se direcionam impetuosamente ao seu formigueiro; em cada movimento, contudo, elas liberam feromônios (`releasePheromone`; este é um dos métodos da classe `Ant`) com o objetivo de sinalizar a outras formigas (possivelmente, de outros formigueiros) que existe um caminho, nas proximidades, para uma instância de alimento. Estes feromônios, `struct Pheromone` com o atributo `lifetime`, devem ser extraídos do mapa a cada intervalo de iterações; assim, cada instância de `Tile` contém uma lista, `pheromones`, de feromônios liberados pelas formigas, que é, em cada iteração, percorrida, ensejando a captura de feromônios mortos – a intensidade de feromônios nesta coordenada, a propósito, é igual ao tamanho desta lista, que, logo, também é modificada em cada iteração.

2.3 Combate entre formigas

Em cada iteração, as formigas que não estão direcionando alimentos ao formigueiro podem escolher combater as adversárias; há, neste sentido, um par de cenários em que a luta é executada. Em um deles, a formiga vislumbra, a uma distância (na métrica L_1) igual a uma unidade, um alimento com volume positivo e, neste caso, seu objetivo é o defender – neste caso, ela luta deterministicamente, se houver adversários em sua coordenada. Em outro, ela não contempla alimentos em suas vizinhanças e, portanto, ela joga uma moeda justa (utilizamos, neste caso, o algoritmo de amostragem utilizado no movimento aleatório com pesos de feromônios) e executa uma escolha entre um movimento aleatório e uma luta – se, outra vez, houver adversários. Mais precisamente, ela escolhe guerrear, não lutar, conforme descrevemos nas sentenças seguintes.

Quando, desta maneira, uma formiga decide lutar, ela executa um ataque coordenado a todas as formigas adversárias em sua coordenada; a probabilidade de que ela vença é, neste caso, proporcional à quantidade de aliadas em sua coordenada. Apesar do ponto culminante desta luta, uma das formigas é *gravemente ferida* e rotulada com o atributo `isDead`; na iteração seguinte, quando percorrermos a lista de feromônios, também capturaremos as formigas mortas e as extrairíremos da simulação. Enfatizamos, aliás, que, mesmo que a rotulação das formigas seja executada em threads distintas, a sua extração é executada na thread principal; isso porque, como elas estão contempladas em um atributo da classe `Map`, `allAnts`, precisaríamos garantir que cada thread modificasse este atributo consitentemente, o que, no entanto, é essencialmente equivalente à execução por uma thread (é bastante difícil particionar o tratamento de uma lista entre threads).

2.4 Sumário dos agentes

A movimentação das formigas, a liberação de feromônios e os aspectos bélicos foram, logo, particularmente inconvenientes, na medida em que eles culminavam na caracterização de ações bastante entrelaçadas – as formigas modificam as comidas e, também, modificam a si mesmas; a dinamicidade introduzida pelo surgimento da morte foi, também, aflitivo, porque precisávamos, neste caso, ser mais cautelosos na escolha da formiga seguinte a executar o movimento e interagir com o mapa. A propósito, a introdução de aspectos estocásticos na simulação injetou oscilações no sistema que, em alguns cenários, foram inadequadas.

Tivemos, aliás, confrontos tétricos com a própria linguagem de programação, `C++`; procedimentos elementares, como o cômputo da distância entre uma coordenada e um segmento, exigiram tremenda introspecção. A implementação de algoritmos multithread, contudo, conformou o epítome de nossas atribulações; nós a descreveremos nas seções seguintes.

3 Impressão do mapa no console

Neste ínterim, contudo, descrevemos os procedimentos para imprimir o mapa, e suas características, no console; veja, para isso, a Figura 2. Antecipamos, aliás, que introduzimos uma variável condicional que controla as threads responsáveis pelos movimentos das formigas quando todas elas tiverem atuado; em seguida, instanciamos outro conjunto de threads para atualizar os atributos do mapa (por exemplo, extrair os feromônios, repor os alimentos ou identificar formigas mortas em combate) – estes procedimentos estão consolidados no método `prepareNextIter`, da classe `Map`. Assim, no estágio seguinte à atualização do mapa, nós o imprimimos, como na Figura 2: inicialmente, informamos, para cada colônia, a quantidade de alimentos depositada e a quantidade de formigas vivas; inserimos, então, as coordenadas de cada mapa, com as informações da intensidade de feromônios e da quantidade de formigas. Em maior detalhe, distinguimos a impressão de cada coordenada em um tripleto de categorias,

1. *andarilhos*, `|a, p|`, que contempla a quantidade de formigas, `a`, e de feromônios, `p`, em cada iteração,

2. *formigueuros*, `|AnthillName, FoodStorage, Ants|`, em que apontamos, sequencialmente, para o nome do formigueiro, a quantidade de alimento armazenado e a quantidade de formigas e

3. *comidas*, `|Food, Volume|`, no qual inserimos, adjacente ao nome `Food`, a quantidade, em unidades de volume, de alimento disponível.

Estes foram, portanto, os mecanismos que ensejaram a atualização e a impressão do mapa no console; verificaremos, mais tarde, alguns detalhes da implementação multithread.

Figura 2: Mapa em que a simulação ocorre.

4 Implementação multithread

Importantemente, encetamos com uma implementação em uma thread; o objetivo disso é garantir que, em circunstâncias com múltiplas threads, os erros sejam culminantes da paralelização, e não idiossincráticos do desenho da simulação. Vamos, desta maneira, descrever, laconicamente, a introdução de múltiplas threads no programa. Escrutinaremos, contudo, a captura de alimento, que, neste caso, consolida inconveniências amarradas fortemente à programação em paralelo.

Neste sentido, a etapa inicial do programa consiste na inicialização do mapa; precisamos instanciar as coordenadas, os formigueiros e as formigas. Apesar de podermos introduzir múltiplas threads para particionar estas tarefas, escolhemos a decisão mais parcimoniosa e executamos estes procedimentos na thread principal, porque, como esta etapa é executada uma vez, no início da simulação, rotulamos sua paralelização como inapropriada. Em sequência, precisamos executar os procedimentos da simulação – movimento das formigas e atualização do ambiente – por uma quantidade de iterações parametrizável; contemplamos, nas linhas seguintes, esta descrição.

Especificamente, a instância do mapa contempla um contador, `currAnts`, e uma lista de formigas, `allAnts`. Em cada iteração do jogo, cada thread executa o método `computeNextAnt`, do classe `Map`, e captura a formiga que executará o movimento em sua thread; nes método, com acesso controlado por um mutex, `mapMutex`, e por uma variável de condição, `cv` o contador `currAnts` é incrementado. Quando, neste cenário, `currAnts` for maior do que, ou igual ao, tamanho da lista `allAnts`, estas threads são interrompidas por `cv`. Assim, temos a informação de que o mapa precisa ser atualizado; executamos, portanto, o método `prepareNextIter`.

Neste caso, inserimos, neste método, o algoritmo canônico de processamento de dados em paralelos: particionamos o cojuntos de itens a serem processados – as coordenadas – uniformemente e, em seguida, processamos, em cada thread, cada elemento desta partição. Em maior detalhe, a instância `map` da classe `Map` goza, entre seus atributos, de uma lista de coordenadas dispostas na simulação; em sua atualização, precisamos atualizar, individualmente, cada coordenada. Portanto, suponha que haja a objetos para atualizarmos (escrevemos a, neste caso, para a área do mapa) e que precisamos distribuir esta execução em N threads; nosso procedimento, enfaticamente, processa, em cada thread, uma quantidade aproximadamente igual a $\lceil a/N \rceil$ de instâncias – e, em cada instância, há, com efeito, um par de procedimentos que precisamos executar. Por um lado, precisamos identificar os feromônios inativos – isto é, os hormônios que transcenderam seu tempo de vida – e assassinar as formigas gravemente feridas em combate (equivalentemente, rotuladas com o atributo `isDead`); por outro lado, as coordenadas com alimentos têm de, possivelmente, ser regeneradas. Mais tarde, combinamos (`join`) as threads para garantir que elas executaram suas tarefas e liberamos o espaço de memória que elas ocupavam; o contador `currAnts`, desta maneira, é atualizado e notificamos as threads originais, subjacentes ao movimento das formigas, de que elas devem continuar seu processamento[3]. Factualmente, esta etapa culmina na inicialização

[3]Perceba, logo, que, se escrevermos N para a quantidade de threads introduzidas no programam inici-

da simulação com uma configuração de mapa distinta da inicial; e, a propósito, executamos este algoritmo por uma quantidade igual ao tempo (parametrizável) da simulação[4].

4.1 Captura de alimentos

Crucialmente, precisamos, também, implementar uma interação consistente entre as formigas e os alimentos; com as restrições impostas, esta etapa foi, na verdade, tremendamente angustiante. Existe, mais especificamente, um par de condições que ensejam que uma formiga interaja com um alimento; heuristicamente, precisamos escolher $N \in \mathbb{N}$, (1) inserir N assentos nas coordenadas em que os alimentos estão e (2) N bastões – dispostas em uma mesa circular, cada formiga precisa levantar, para capturar comida, um par de bastões, à sua esquerda e à sua direita, conformando, desta maneira, o *problema dos filósofos*. Nesta seção, portanto, descrevemos, em maior detalhe, nossa aproximação a este cenário.

Em um momento inicial, precisamos controlar a quantidade de formigas que, em atitude simultânea, se sentam à mesa (equivalente, neste caso, a uma array, `seats`, atributo de `Food`) amarrada à coordenada do alimento; introduzimos, para isso, outro contador, `currAnt`, que aponta para a formiga mais recente que cativou um assento, e, quando este contador for maior do que, ou igual à, quantidade de assentos, garantimos que as formigas subsequentes não interagirão com a comida. Munidos, logo, destes assentos (na descrição do problema dos filósofos, cada filósofo gozava de um atributo – com fome, comendo ou pensando –; neste caso, os atributos estão acorrentados aos assentos e, assim, são elementos do conjunto $\{\texttt{LIVRE}, \texttt{COMENDO}, \texttt{COM FOME}\}$), aplicamos a solução de Dijkstra ao problema dos filósofos, introduzindo, para isso, um conjunto de semáforos para garantir a consistência do acesso aos bastões (e, portanto, contornar as condições de corrida) e o prosseguimento da execução (circundando, também, a inanição).

Este é, neste sentido, o algoritmo que utilizamos para coletar, consistentemente, a comida de uma fonte; esta etapa emanou, aliás, sentimentos dicotômicos. Em uma direção, precisamos de bastante circunspecção em sua implementação; neste momento, a reunião de agentes – formigas, comidas, mapa e coordenadas – pleteia tremenda prudência. Em outra, Dijkstra, no século XX, havia nos apresentado uma implementação plausível; precisávamos, portanto, a acoplar ao sistema que escrevemos. Sensivelmente, a execução aparenta ser adequada.

5 Conclusões

Neste documento, portanto, descrevemos as decisões, escolhas e, importantemente, que nos direcionaram à implementação de um sistema caracterizado pela interação entre colônias de formigas com um ambiente. Escrevemos, com efeito, a respeito dos agentes e de como eles

almente, existem, em algumas circunstâncias, $2N$ threads ativas, apesar de, simultaneamente, N estiverem ocupadas.

[4]Escolhemos, em nossa implementação, computar o tempo de simulação como uma quantidade de iteração; poderíamos, correspondentemente, executar o laço de atualizações da configuração do mapa por uma medida (real) de tempo.

interagem, consolidando, assim, um sistema emergente; contemplamos, em seguida, procedimentos que culminaram a implementação de algoritmos multithread para enformar uma simulação paralela. Enfaticamente, distinguimos as etapas paralelas das iterativas; apontamos, além disso, para seções paralelizáveis, apesar de terem sido executadas sequencialmente (como, por exemplo, a inicialização do mapa). Neste sentido, disponibilizamos as implementações; as instruções de execução, em particular, estão acopladas no arquivo `README.md`, escrito no diretório principal.