Teoria de Jogos e Minions

Juliana Mayumi Hosoume

4 de junho de 2019

Universidade de Brasília - Instituto de Ciências Exatas
Departamento de Ciência da Computação - CIC 116653 - Programação Concorrente
2019.1 - Turmas A - Professor Eduardo Alchieri
Prédio CIC/EST - Campus Universitário Darcy Ribeiro
Asa Norte 70919-970 Brasília, DF
ju.hosoume@gmail.com

1 Introdução

O campo da teoria de jogos está ligado com a modelagem matemática de interações racionais (Camerer et al., 2004). Há aplicações da teoria de jogos em diferentes campos, como ciências sociais, economia, biologia, ciência da computação, entre outros. Os conceitos de teoria de jogos são aplicados, por exemplo, em pesquisas sobre a evolução da cooperação.

Um dos modelos para a evolução da cooperação é baseado no Dilema do Prisioneiro (Axelrod e Dion, 1988). Nesse jogo, é apresentada uma situação em que dois prisioneiros são apresentados possibilidades. Cada prisioneiro está em uma solitária, portanto não há comunicação entre eles. Há falta de provas para condenar ambos com a pena máxima, entretanto podem condená-los com uma pena menor. Simultaneamente, é proposto uma oportunidade de delação. Dessa forma, uma opção é delatar o outro prisioneiro, enquanto a outra é cooperar com o outro prisioneiro. Assim, podem ser encontrados três cenários:

1. Ambos delatam, assim ficam 2 anos na prisão;
2. Um delata e o outro coopera permanecendo em silêncio, o que delatou fica livre, enquanto o que cooperou passa 3 anos na prisão;
3. Ambos cooperaram, assim cada um fica 1 ano na prisão.

Nessa situação, para o melhor resultado individual, o prisioneiro deve escolher não cooperar e, assim, delatar o outro. Todavia, quando ambos escolhem não cooperar, há o pior resultado geral. Interessantemente, humanos, quando apresentados à esse dilema ou em questões similares, possuem a tendência de cooperar (Fehr e Fischbacher, 2003).

Modelos computacionais para simular interações que envolvem o Dilema do Prisioneiro e outros problemas de Teoria de Jogos têm sido feitos para auxiliar no estudo

dos possíveis *payoffs* dadas estratégias distintas. Um dos primeiros torneiros realizados para encontrar uma estratégia ótima foi feito por Axelrod em 1984 e envolveu diversas estratégias feitas por diferentes pesquisadores.

Para a implementação computacional de interações, podem ser utilizadas técnicas de programação concorrente. Ainda que a utilização de diversas *threads* para serem executadas em diferentes núcleos não aumente a velocidade de execução pelo número de núcleos, a implementação correta de concorrência pode acelerar na obtenção de resultados em um problema que possibilite o uso de tal técnica (Jakimovska et al., 2012).

Uma das formas de utilizar técnicas de programação concorrente em linguagens cujo paradigma não é padrão está no uso de APIs externas. Para sistemas UNIX, uma API disponível é a de Pthreads (POSIX threads). A Pthreads segue o padrão IEEE POSIX para a linguagem C, de modo a permitir portabilidade. Pthreads fornece funções importantes para a exclusão mútua e resolução de sincronização, por exemplo. Para tanto, estão disponíveis *locks*, semáforos, variáveis de condição, entre outros (Barney).

Como forma de aplicar conceitos e técnicas de programação concorrente no contexto de teoria dos jogos, este trabalho propõe uma situação imaginária de modo a maximizar a concorrência em um problema similar ao dilema do prisioneiro. Assim, utiliza-se os conceitos de *Minions* e de seu chefe, Gru, para modelar interações e como essas interações afetam gerações de *Minions*. Na próxima seção, há uma definição mais detalhada do problema. Nas seções seguintes serão apresentadas as soluções para questões levantadas e as conclusões decorrentes.

2 Definição do Problema

Os *minions* serão as entidades que interagem, de acordo com uma estratégia definida. Assim, cada *minion* é independente e pode procurar sua ocupação. Neste trabalho, são apresentadas apenas duas estratégias, a dos *minions* brincalhões e a dos *minions* malvados. A estratégia dos *minions* brincalhões é a estratégia da cooperatividade, enquanto os *minions* malvados são os que não cooperam. Gru é o chefe e pode, quando quiser, chamar uma quantidade aleatória de *minions* para executar uma tarefa. Essa tarefa tem prioridade máxima e deve ser cumprida pelos *minions* que estiverem disponíveis. Os *minions* que não estiverem na tarefa, devem continuar tentando realizar interações.

- **Minions**: um conjunto de *minions* é feito por geração. Cada *minion* possui uma estratégia distinta e definida pela geração. Eles podem escolher cooperar ou desertar com frequência determinada pela estratégia. Cada minion é uma *thread* distinta. Durante a geração eles podem interagir ou realizar uma tarefa proposta pelo Gru. A tarefa do Gru tem a prioridade máxima. Se o *minion* foi alocado para uma tarefa do Gru, deve esperar até realizar outra interação. *Minions* não reservados pelo Gru podem continuar as interações. *Minion* verifica se Gru está chamando e se a cota de *minions* ainda não foi alcançada. Se *minions* ainda são necessários, próximos *minions* que estiverem

disponíveis devem imediatamente atender ao chamado do Gru. *Minions* verificam se o outro *minion* que ele escolheu para interação está ocupado e também se ele mesmo não está em uma outra interação. Caso contrário, *minion* fica aguardando e pode interagir com outros que queiram.

- **Interações**: cada interação reserva dois *minions* e, então, o *score* da interação é calculado e salvo no *score* total da geração. Cada *minion* só pode participar de uma interação por vez.

- **Gru**: em tempos aleatórios convoca um grande número de *minions* para realizar uma tarefa. Enquanto os *minions* estão realizando tarefas, não podem participar de interações, ademais as tarefas do Gru possuem prioridade. A tarefa só pode ser iniciada se o número de *minions* necessários for alcançado. Ainda que a tarefa não tenha sido iniciada, os *minions* alocados para a tarefa devem ficar esperando.

- **Gerações**: uma *thread* será responsável por calcular cada uma das gerações. Ela somente será finalizada quando o número de interações definido for alcançado. A quantidade de *minions* com cada estratégia na geração será definida pelo seu *score* na geração anterior.

3 Descrição do Algoritmo Desenvolvido para a Solução do Problema

O problema fundamental para a questão proposta está na sincronização dos *minions* e maximização da concorrência entre as interações. Desse modo, cada *minion* possui uma *flag* para indicar se está ocupado ou não ($minions[id].occupied$). Além disso, são utilizadas variáveis condicionais para fazer com que os *minions* aguardem caso eles mesmos estejam ocupados ou o *minion* que se quer interagir esteja ocupado ($minions[id].mcond$). É utilizado um *lock* ($lock\_minions$) para evitar que mais de um *minion* altere seu estado ou do outro ao mesmo tempo, dessa forma criando uma exclusão de uma região crítica. O *lock* também é utilizado para que os sinais sejam devidamente recebidos pela condição de espera.

Nota-se também, no código a seguir, que os *minions* esperam em uma condicional quando recrutados pelo Gru (sinalizado por $gru\_recruiting$). Por meio do contador de *minions* ($to\_recruit$), é sinalizada se meta do Gru já foi alcançada e assim há um *post* para que Gru continue a executar. Essa parte do problema será discutida com mais detalhes quando explorada a *thread* do Gru.

```
while (minions[id].occupied ||
       minions[partner].occupied ||
       (gru_recruiting && (to_recruit > 0))) {
  if (minions[id].occupied) {
    cout << "Minion " << id << " is already occupied." << endl;
    pthread_cond_wait(&minions[id].mcond, &lock_minions);
  } else if (gru_recruiting && (to_recruit > 0)) {
    minions[id].occupied = true;
    --to_recruit;
    minions_recruited.push_back(id);
```

```
11      cout << "!!! Minion " << id << " RECRUITED!" << endl;
12      if (to_recruit <= 0) {
13        cout << "!!! Minion " << id << " REC FIN!" << endl;
14        sem_post(&gru_recruited);
15      }
16      pthread_cond_wait(&minions[id].mcond, &lock_minions);
17    } else if (minions[partner].occupied) {
18      cout << "Minion " << id << " is waiting partner " << partner << "."
        << endl;
19      pthread_cond_wait(&minions[partner].mcond, &lock_minions);
20    }
21  }
```

Na situação em que os *minions* podem realizar a interação, o *minion* requerente da interação altera as flags (*minions*[*id*].*occupied*) e demanda o cálculo pelo *score* da interação. Ademais, incrementa *interactions* que é utilizada para marcar quando uma geração chegou ao fim. Com isso, ele libera o *lock* e entra no tempo de *sleep* que corresponde ao tempo de interação.

```
1     ++interactions;
2     minions[id].occupied = true;
3     minions[partner].occupied = true;
4     sleep_time = minions[id].strategy.interaction_duration;
5     cout << "Minion " << id << " is interacting with " << partner <<
      endl;
6     cout << "Num interactions = " << interactions << endl;
7     generation.addScore(minions[id].strategy, minions[partner].strategy
      );
8     pthread_mutex_unlock(&lock_minions);
9
10    sleep(sleep_time);
```

Ao encerrar a interação, o *minion* requerente da interação libera ambos os *minions* envolvidos. Em seguida, sinaliza para todos os que estiverem esperando que eles já podem acordar para tentar uma nova interação(*broadcast* em *minions*[*id*].*mcond*).

```
1     pthread_mutex_lock(&lock_minions);
2     minions[id].occupied = false;
3     minions[partner].occupied = false;
4     pthread_cond_broadcast(&minions[id].mcond);
5     pthread_cond_broadcast(&minions[partner].mcond);
6     cout << "Minion " << id << " is done with " << partner << endl;
7     pthread_mutex_unlock(&lock_minions);
8     sleep(5);
```

A geração é encarregada de sincronizar e sinalizar aos *minions* quando devem iniciar as interações (*broadcast* em *gen_cond*). Para tanto, ela utiliza contadores para verificar se o número máximo de interações foi alcançado. Por conseguinte, a fim de manter o incremente e a verificação desse contador, é usado um *lock* (*lock_minions*). Para a sincronização dos *minions* e do *Gru* é utilizada sinalização de variável de condição (*gru_cond*).

```
1 void Generations::startGeneration() {
2   vector<int> vec {0, 0};
3   ++num_gen;
4   scores.push_back(vec);
```

```
  pthread_mutex_lock(&lock_minions);
  gen_calc = true;
  gru_generation = true;
  interactions = 0;
  done_minions = 0;
  pthread_cond_broadcast(&gen_cond);
  pthread_cond_signal(&gru_cond);
  pthread_mutex_unlock(&lock_minions);
}
```

Quando o último *minion* requerente de interação verifica que o número de interações máximo da geração foi atingido, ele acorda a *thread* responsável pelo cálculo das gerações e sinaliza que o *Gru* deve parar de recrutar. Em seguida, espera o sinal de uma nova geração por uma variável de condição.

```
    if (interactions >= NUM_INTERACTIONS) {
      gru_generation = false;
      ++done_minions;
      cout << "DONE!" << endl;
      if (done_minions == NUM_MINIONS) {
        pthread_cond_signal(&minion_done);
      }
      pthread_cond_wait(&gen_cond, &lock_minions);
      pthread_mutex_unlock(&lock_minions);
      continue;
    }
```

A *thread* do Gru, assim como a *thread* de gerações, é uma *thread* única de seu tipo, não obstante de funcionamento síncrono com as *threads* dos *minions*. O Gru só pode recrutar quando os *minions* estão ativos, por conseguinte aguarda a indicação da *thread* de gerações (*gru_generation*). Novamente, esse processo é implementado com a utilização de uma variável de condição (espera na condição *gru_cond*). Gru sinaliza para os *minions* que está recrutrando por meio da *flag gru_recruiting*. O último *minion* a ser recrutado indica para o Gru, por meio de um semáforo (*gru_recruited*) que ele pode continuar a execução da tarefa. Os *minions* recrutados, ficam em uma lista (*minions_recruited*) o que permite que sejam acordados pelo Gru para que possam continuar as interações. Pode-se perceber que o Gru não altera os *minions* que não foram recrutados, no entanto esses *minions* podem estar esperando por interação de algum *minion* que foi recrutado.

```
void *fgru(void *identifier) {
  int id = *((int *) identifier);
  free((int *)identifier);
  int min_minions = NUM_MINIONS/5;
  int max_minions = NUM_MINIONS/2;

  while(1) {
    sleep(GRU_WAIT);
    to_recruit = limited_rand(min_minions, max_minions);

    pthread_mutex_lock(&lock_minions);
    while (!gru_generation) {
      pthread_cond_wait(&gru_cond, &lock_minions);
    }
```

```
15      cout << "!!! Gru is recruiting " << to_recruit << " minions!" <<
        endl;
16      gru_recruiting = true;
17      pthread_mutex_unlock(&lock_minions);
18
19      sem_wait(&gru_recruited);
20      pthread_mutex_lock(&lock_minions);
21      gru_recruiting = false;
22      cout << "!!! Recruiting number reached!" << endl;
23      pthread_mutex_unlock(&lock_minions);
24
25      sleep(limited_rand(5, 7));
26
27      pthread_mutex_lock(&lock_minions);
28      cout << "!!! Releasing Minions!" << endl;
29      for (auto minion_indx : minions_recruited) {
30      cout << "!!! Released Minion : " << minion_indx << endl;
31        minions[minion_indx].occupied = false;
32        pthread_cond_broadcast(&minions[minion_indx].mcond);
33      }
34      minions_recruited.clear();
35      pthread_mutex_unlock(&lock_minions);
36    }
37
38    pthread_exit(0);
39  }
```

4 Conclusão

O problema proposto permitiu o exercício de diversas funções relacionadas à programação concorrente e ao Pthreads. Para sincronização e criação de exclusão foram utilizados desde contadores, *locks*, até mesmo variáveis de condição a fim de que as *threads* não ficassem em uma espera ocupada. A resolução do problema é similar à resolução do problema dos filósofos. Como proposto, foi possível calcular os *payoffs* e perceber que a estratégia de não cooperação dominava a estratégia que coopera, ainda que o *payoff* final da população fosse menor.

Pode-se notar que modelar cada interação como *thread* independente maximizaria a velocidade dos resultados e diminuiria a complexidade do programa, todavia não exercitaria tanto a resolução de problemas de concorrência. Como proposta de melhorias, as saídas poderiam estar melhor organizadas. Além disso, cada *minion* poderia possuir memória das jogadas anteriores, abrindo mais possibilidades para estratégias.

Referências

Robert Axelrod e Douglas Dion. The Further Evolution of Cooperation. *Science*, 242(4884):1385–1390, Dezembro 1988. ISSN 0036-8075, 1095-9203. doi: 10.1126/science.242.4884.1385. Disponível em <`https://science.sciencemag.org/content/242/4884/1385`>. Acesso em 2019-06-04.

Blaise Barney. POSIX Threads Programming. Disponível em <https://computing.llnl.gov/tutorials/pthreads/>. Acesso em 2019-06-04.

Colin F. Camerer, Teck-Hua Ho, e Juin Kuan Chong. Behavioural Game Theory: Thinking, Learning and Teaching. In Steffen Huck, (Ed.), *Advances in Understanding Strategic Behaviour: Game Theory, Experiments and Bounded Rationality*, pages 120–180. Palgrave Macmillan UK, London, 2004. ISBN 978-0-230-52337-1. doi: 10.1057/9780230523371_8. Disponível em <https://doi.org/10.1057/9780230523371_8>.

Ernst Fehr e Urs Fischbacher. The nature of human altruism. *Nature*, 425(6960): 785, Outubro 2003. ISSN 1476-4687. doi: 10.1038/nature02043. Disponível em <https://www.nature.com/articles/nature02043>. Acesso em 2019-06-04.

D. Jakimovska, G. Jakimovski, A. Tentov, e D. Bojchev. Performance estimation of parallel processing techniques on various platforms. In *2012 20th Telecommunications Forum (FOR)*, pages 1409–1412, Novembro 2012. doi: 10.1109/FOR.2012.6419482.