Tentativa Paralelização A*

Daniel Souza de Campos – 2018054664

Introdução a Programação Paralela – 2021/2

Sumário

1 – Introdução

No contexto de modelagem de Problemas com Estados, podemos dizer que um estado pode ser alcançado por um Agente ao ser realizada alguma Ação. Uma boa representação para essa modelagem é via o uso de Grafos. Nesses grafos, os nós representam estados e arestas representam possibilidade de se chegar a um estado a partir de outro via uma ação. As arestas podem também ter um peso associado representando o custo de se tomar a ação representada. Dessa forma, um problema a ser resolvido pode ser o de encontrar um caminho entre dois nós, um nó de origem e um de destino, para que o Agente percorra esse caminho dadas as suas necessidades.

Existem vários Algoritmos de Busca de Estados para encontrar esse caminho. Entre eles, pode-se citar o BFS, DFS, UCS, GS e o A*. A principal questão nesses algoritmos é balancear a quantidade de nós considerados para fazer parte desse caminho com a capacidade de encontrar o melhor caminho entre os nós objetivos.

Dessa forma, surge o Algoritmo A*. Ele é um algoritmo com Informação o que significa que já se conhece qual é o nó objetivo que o Agente deve conseguir chegar. A principal característica desse algoritmo é que ele faz uso do conhecimento do nó alvo para direcionar a sua pesquisa por meio de uma Heurística. A heurística é uma função utilizada ao longo do processo de busca para que o algoritmo seja eficiente ao escolher o próximo nó a considerar no caminho. Algumas heurísticas conhecidas são a Distância Manhattan e a Distância Euclidiana entre os nós.

Como o A* é um algoritmo importante, naturalmente, para aproveitar melhor a capacidade das máquinas atuais, surgiram tentativas de sua paralelização. Algumas tentativas que se pode citar são: *Parallel A* Graph Search* (Weinstock, Hollaway), *PA-Star* (Sunfield et. al) e *PNBA** (Rios, Chaimowicz).

Esse trabalho mostra sobre a tentativa de paralelização do Algoritmo A* com base no algoritmo PNBA* com Pthreads.

2 – Ferramentas para o trabalho

Para gerar os grafos de teste foi utilizada a linguagem Python e a biblioteca Networkz. Para o programa principal, foi utilizada a linguagem C++11. Para desenvolvimento do trabalho, foi utilizado o editor de códigos *Visual Studio Code*. A máquina de desenvolvimento possui o sistema operacional Windows com o *Windows Subsystem For Linux* instalado, o que permitiu o uso de Pthreads, 8GB RAM e processador Intel Core I5 de sétima geração.

Como repositório e controle de versionamento do código, foi utilizado o Github e o trabalho estará disponível em https://github.com/Pendulun/ParallelAStar.

3 – Instruções de compilação e uso do programa

Para compilar e executar o trabalho, é necessário estar em ambiente Linux para fazer uso de Pthreads. Pode ser usado o *Windows Susbsystem for Linux* também. Para realizar os testes, pode-se executar o comando:

make test ARG=<quantidadeDeNósDoGrafo>

Exemplo:

make test ARG=2000

Atualmente, o valor passado no ARG pode ser um da seguinte lista:

100, 1000, 2000, 5000, 10000, 30000, 50000, 80000, 100000.

Para executar a versão paralela (que não está funcionando corretamente) basta adicionar um valor inteiro à regra do Makefile na linha 58 ao final. Por exemplo:

```
test: ${EXEC}
	./${EXEC} ${GRAPHS}teste${ARG}.txt ${GRAPHS}nodePosGraph${ARG}.txt ${GRAPHS}nodesToGetPathGraph${ARG}.txt 4
```

Executará a versão paralela com 4 threads.

Para executar a versão sequencial, basta retirar esse último número do comando.

4 – A*

Como dito antes, o A* é um Algoritmo de Busca com informação. Isso significa que se pode aproveitar a informação das coordenadas do nó objetivo para direcionar a pesquisa do melhor caminho até ele. Isso é feito fazendo uso da heurística.

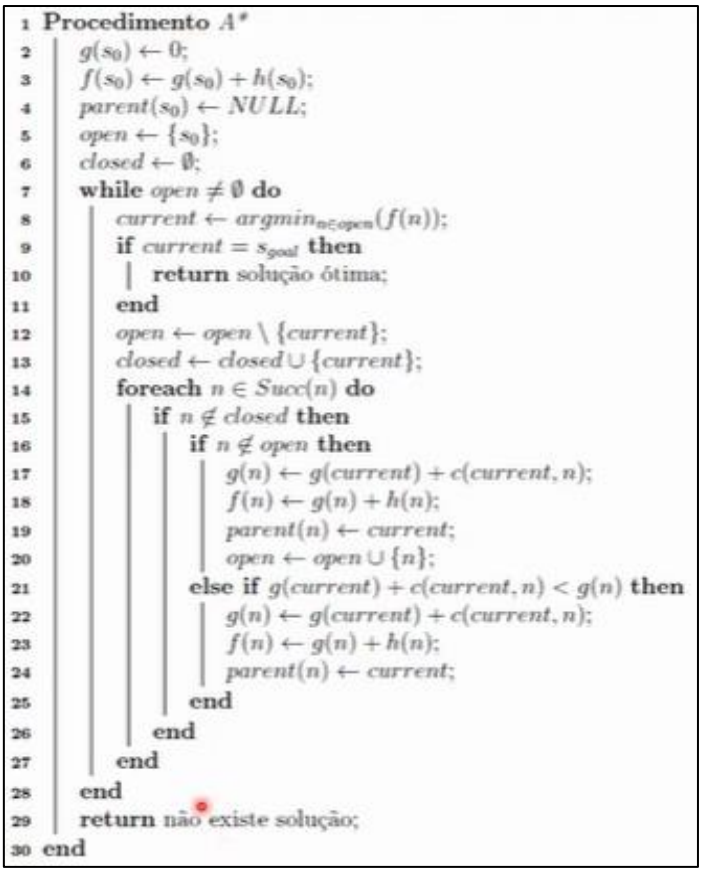
Além da heurística, a soma dos pesos das arestas para se chegar em um nó é conhecido como custo do caminho $g(x)$ de um nó x. O custo do caminho $g(x)$ junto com a heurística $h(x)$ aplicada ao nó forma o custo total do nó: $f(x) = g(x) + h(x)$.

O algoritmo A* possui duas estruturas de dados principais: O conjunto aberto e o conjunto fechado de nós. O conjunto aberto é formado pelos vizinhos dos nós que já foram explorados e, portanto, esses nós são chamados de alcançados. Esse conjunto é ordenado pelo custo total de um nó, isso é, o custo do caminho até esse nó mais o custo retornado pela heurística.

O conjunto fechado é formado pelos nós já explorados. Explorar um nó significa descobrir a quais outros nós esse nó está conectado. Além disso, caso a heurística seja consistente e admissível, ao explorar um nó, temos a certeza de que encontramos o melhor caminho até esse nó sendo explorado. Dessa forma, ao explorar o nó objetivo, já temos um caminho até ele.

O algoritmo funciona como o seguinte:

1. Adicionamos o nó de origem ao conjunto aberto levando em consideração o seu $f(x)$
2. Enquanto o conjunto aberto não estiver vazio
    a. Retira-se um nó x do conjunto aberto
    b. Se esse nó x é o nó objetivo:
        i. Retorne o caminho até x
    c. Adiciona-se esse nó x no conjunto fechado
    d. Para todo vizinho y de x:
        i. Se y não está no conjunto fechado
            1. Se y não está no conjunto aberto, calculamos $f(y)$ e adicionamos y ao conjunto aberto
            2. Se y está no conjunto aberto associado a um $f'(y)$ maior do que o $f(y)$ calculado atualmente, atualizamos o valor de custo total de y no conjunto aberto
3. Retorne vazio



```
1  Procedimento A*
2      g(s_0) ← 0;
3      f(s_0) ← g(s_0) + h(s_0);
4      parent(s_0) ← NULL;
5      open ← {s_0};
6      closed ← ∅;
7      while open ≠ ∅ do
8          current ← argmin_{n∈open}(f(n));
9          if current = s_goal then
10             return solução ótima;
11         end
12         open ← open \ {current};
13         closed ← closed ∪ {current};
14         foreach n ∈ Succ(n) do
15             if n ∉ closed then
16                 if n ∉ open then
17                     g(n) ← g(current) + c(current, n);
18                     f(n) ← g(n) + h(n);
19                     parent(n) ← current;
20                     open ← open ∪ {n};
21                 else if g(current) + c(current, n) < g(n) then
22                     g(n) ← g(current) + c(current, n);
23                     f(n) ← g(n) + h(n);
24                     parent(n) ← current;
25                 end
26             end
27         end
28     end
29     return não existe solução;
30 end
```

Figura 1: Algoritmo A (Norvig, Russel)*

O algoritmo A*, apesar de eficiente na procura, ainda sofre com alguns problemas. Um desses problemas é o seu grande uso de memória já que ele armazena todos os nós pelos quais ele passou na procura do melhor caminho.

5 – PNBA*

Para auxiliar na tentativa de paralelização do algoritmo A*, foram consultados na bibliografia alguns trabalhos já existentes. Entre os algoritmos encontrados, pode-se destacar o PNBA* (Rios, Chaimowicz) pois ele se demonstrou passível de melhorias como os próprios autores concluem em seu trabalho.

O PNBA* é uma versão paralela de um outro algoritmo conhecido como NBA*. O NBA* tenta melhorar a velocidade da pesquisa ao fazer com que duas pesquisas aconteçam ao mesmo tempo, mas não paralelamente, ou seja, ele ainda é um algoritmo sequencial. A primeira pesquisa ocorre partindo do nó de origem com objetivo de chegar no nó alvo. A segunda pesquisa ocorre partindo do nó alvo com objetivo de chegar no nó inicial. Dessa forma, o algoritmo deve fazer com que ambas pesquisas concordem em um nó intermediário para formar o caminho completo.

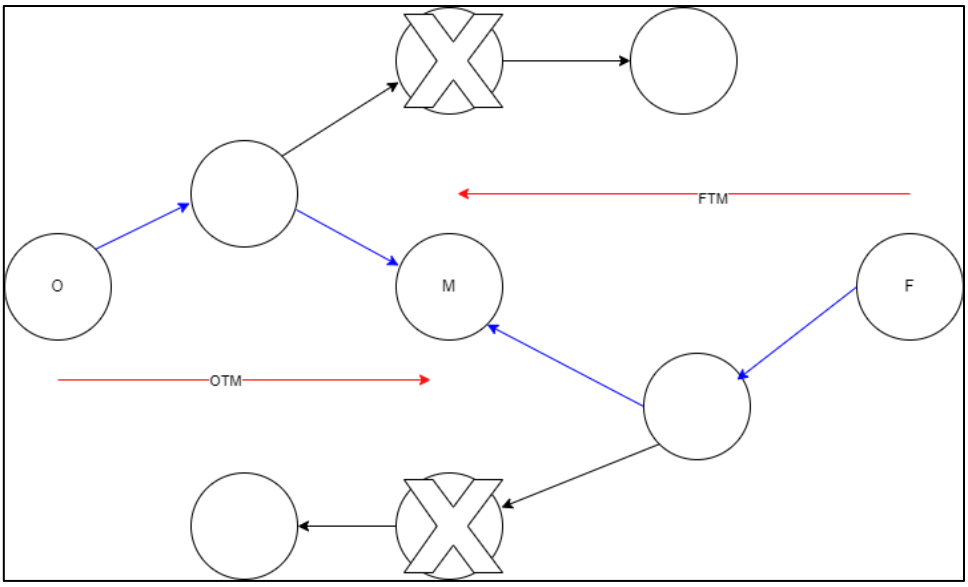


Figura 2: Funcionamento do NBA e PNBA**

Como dito antes, o NBA* não é paralelo e, portanto, o PNBA* veio para ser a versão sua paralela ao empregar uma *thread* em cada lado da pesquisa.

O PNBA* funciona um pouco diferente do que o A*. Ele possui um conjunto aberto M que contém todos os nós que não sejam o de início e de fim. Todos os nós e o custo do melhor caminho L começam com o valor infinito. Os nós dentro do conjunto M tem uma categoria associada informando se eles já foram alcançados ou não.

Além disso, como existem duas buscas, o custo do caminho até um nó, o menor custo total presente no conjunto aberto e o valor da heurística de um nó para cada Pesquisa i podem ser chamados de, respectivamente, g_i, F_i e h_i.

Algorithm 1 PNBA*, Parallel New Bidirectional A*

```
1:  while ¬finished do
2:    x ← open_1.pop()
3:    if x ∈ M then
4:      if (f_1(x) < L) ∧ (g_1(x) + F_2 − h_2(x) < L) then
5:        for all edges (x, y) of the graph being explored do
6:          if (y ∈ M) ∧ (g_1(y) > g_1(x) + d_1(x, y)) then
7:            g_1(y) ← g_1(x) + d_1(x, y)
8:            f_1(y) ← g_1(y) + h_1(y)
9:            if y ∈ open_1 then
10:             open_1.remove(y)
11:           open_1.insert(y)
12:           if g_1(y) + g_2(y) < L then
13:             lock
14:             if g_1(y) + g_2(y) < L then
15:               L ← g_1(y) + g_2(y)
16:             unlock
17:   M ← M − {x}
18:   if open_1.size() > 0 then
19:     F_1 ← f_1(open_1.peek())
20:   else
21:     finished ← true
```

Se x ainda não foi explorado
Se é um ramo promissor
Para todo vizinho
Se ele não foi explorado e encontramos um caminho melhor
Insere/Atualiza o nó na openList
Se o caminho OTM + FTM melhor do que o encontrado até agora, atualiza o melhor caminho
Coloca o nó explorado na closedList
Se não existir mais nós na openList termina execução

Figura 3: PNBA (Rios, Chaimowicz) com anotações*

A Figura 3 mostra o PNBA* por completo. Essa é a versão do ponto de vista da Pesquisa 1. A Pesquisa 2 funciona igualmente apenas trocando 1 por 2 e vice-versa no código.

Pode-se perceber que, na linha 4, existe um mecanismo de poda de pesquisa. Ele confere se o nó retirado do conjunto aberto possui um custo total associado dentro da pesquisa f_i menor do que o melhor caminho encontrado até o momento. Isso serve para desconsiderar um ramo de caminho como ilustrado na Figura 2 pelos nós marcados com um X. Outra condição, ainda na linha 4, é a de se, no melhor caso, o nó x atual forma um caminho com tamanho menor do que o melhor caminho encontrado atualmente. Se o nó não satisfizer essas condições, nenhum nó será alcançado a partir dele e ele será marcado como explorado.

Se o nó satisfizer as condições, alcançamos os seus vizinhos. Para cada vizinho y, ainda é checado se, considerando esse nó y como o nó do meio do caminho, ele forma um caminho de tamanho menor do que o menor caminho já encontrado. Se sim, ele vira o novo nó do meio.

Ao final, diferentemente do algoritmo A*, o PNBA* termina a execução apenas quando não existem mais nós no conjunto aberto a serem explorados. Nesse ponto, um nó intermediário poderá ter sido encontrado pelas duas pesquisas ou não.

6 – Tentativa melhoria PNBA*

Como dito antes, esse trabalho se propôs a se basear no PNBA* para adicionar ainda mais *threads* nos processos de pesquisa do que apenas uma *thread* por pesquisa, algo que os próprios autores concluem em seu trabalho como sendo uma evolução esperada e natural para o algoritmo.

Dessa forma, foi pensado para que a paralelização ocorresse de forma simples com mais de uma *thread* consumindo do conjunto aberto de uma pesquisa i por vez.

```
main():
   Adiciona O na openList da P1
   Adiciona F na openList da P2
   PARA N/2 threads:
      Pesquisa(O)
   PARA N/2 threads:
      Pesquisa(F)

Pesquisa():
   ENQUANTO(true):
      SE openList não vazio E não terminar:
         Retira um nó x da openList
         Vê se é um ramo que não deve ser cortado
         PARA TODO vizinho y de x:
            SE não foi explorado:
               SE não está na openList:
                  Adiciona y na openList
               SENÃO-SE está na openList com um caminho pior:
                  Atualiza y na openList
               SE y foi atualizado/inserido na openList:
                  Atualiza L e M se possível
         Adiciona x na closedList
      SENÃO:
         SE é para terminar:
            break
         Barreira esperando adicionarem mais nós ou todos as threads chegarem e definir para
acabar
```

*Figura 4: Esboço da melhoria do PNBA**

O algoritmo proposto na Figura 4 tem uma estrutura mais parecida com o A* original do que com o PNBA*, entretanto, ele também faz uso das condições de poda e também tenta encontrar um nó no meio do caminho.

A principal questão desse algoritmo é a quantidade de sincronizações que deveriam ser realizadas pelas *threads* de uma mesma pesquisa. Enquanto no PNBA* cada *thread* possuía seu próprio conjunto aberto e fechado, nessa versão, as *threads* deveriam sincronizar o seu acesso à essas estruturas de dados. Outros pontos de sincronização seriam no acesso ao nó do meio escolhido por cada Pesquisa e o acesso a nós que já estão sendo explorados ou alcançados por outras *threads* que são conhecidos como nós ocupados.

Dessa forma, ao perceber os diversos pontos de sincronização necessários, foi decidido utilizar Pthreads. Na implementação do algoritmo foram utilizados muitos pontos de sincronização, locks e barreiras. Para tentar diminuir o *overhead* ao adquirir um lock, foram utilizados *read-write locks* para permitir que várias *threads* acessassem pontos onde apenas a leitura de uma estrutura fosse de interesse.

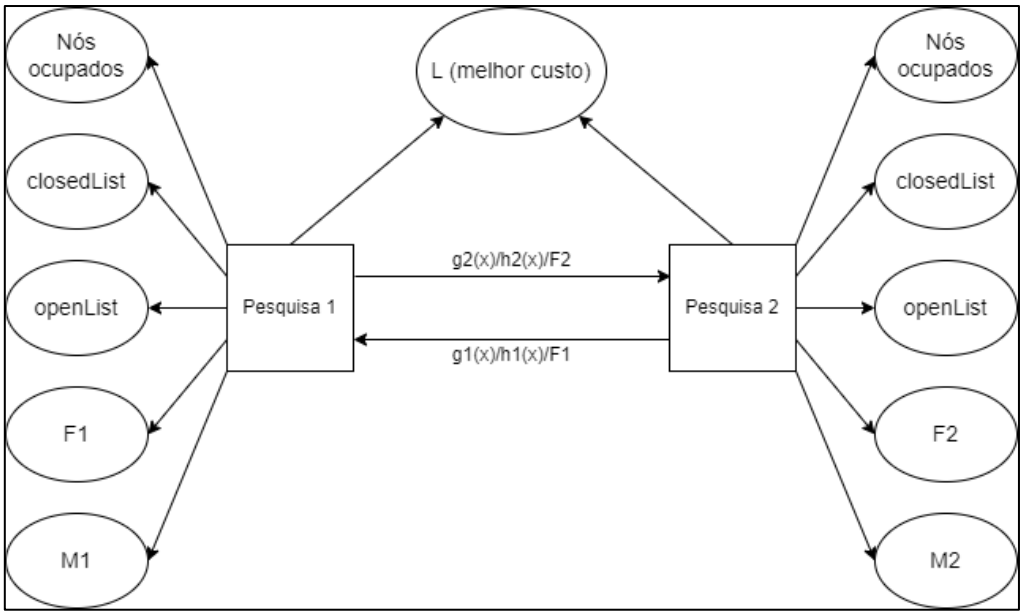


Figura 5: Dependências entre as Pesquisas e entre as threads de uma mesma Pesquisa

Entretanto, de acordo com a dificuldade de se fazer corretamente a sincronização no código, a implementação da versão melhorada do PNBA* não foi completada. Dessa forma, os testes de comparação com os tempos de execução da versão sequencial do A* não puderam ser realizados.

Entretanto, como já havia me preparado para os testes, acabei gerando os grafos necessários utilizando a biblioteca Networkz do Python. Os nós dos grafos criados estavam associados a coordenadas (x, y), de acordo com a necessidade das heurísticas distância Manhattan e distância euclidiana, e as arestas possuíam pesos. Segue um exemplo de um grafo gerado:

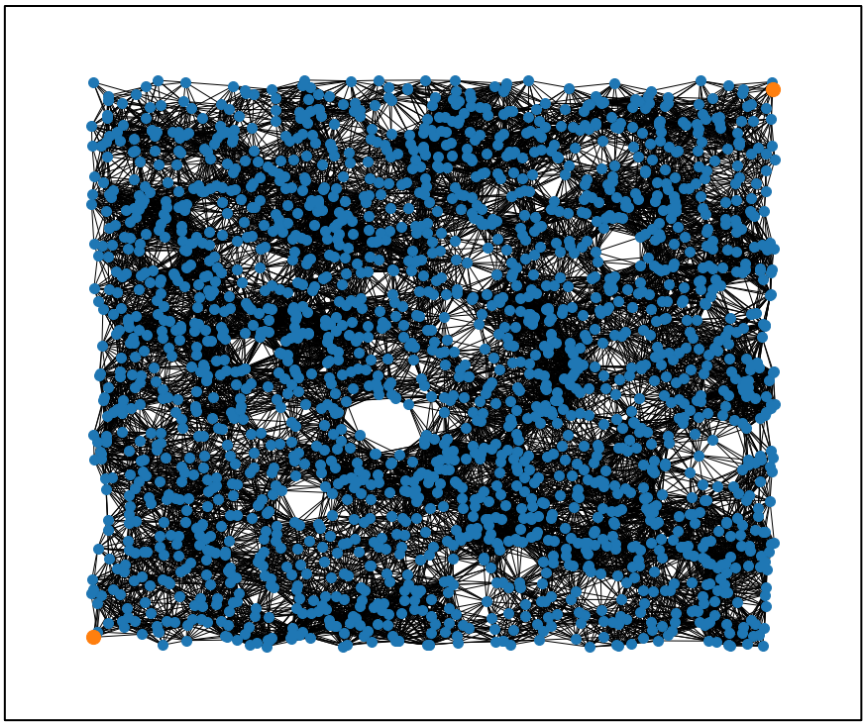


Figura 6: Grafo com 2 mil nós. O nó objetivo e o nó inicial estão de laranja.

7 - Conclusão

O objetivo do trabalho era implementar uma melhoria ao algoritmo PNBA* ao fazer com que cada uma das duas pesquisas fosse realizada com mais de uma *thread* por vez. Infelizmente, esse objetivo não foi alcançado tendo em vista que não foi possível completar a implementação do algoritmo paralelo devido à dificuldade em tratar os pontos de sincronização de modo a garantir o correto funcionamento da execução.

De qualquer forma, o trabalho foi importante para tentar por em prática o que foi aprendido ao longo da matéria. Entre essas coisas, destacam-se: Análise de dependência de dados de um algoritmo e análise de ferramentas para paralelização como Pthreads e OpenMP. Para concluir, o trabalho mostrou que a paralelização de um algoritmo pode não ser simples e que repensar a sua estrutura pode ser necessário para que menos sessões críticas apareçam e influenciem na qualidade do trabalho.

8 - Bibliografia

Parallel A Graph Search*; Weinstock, Ariana; Hollaway, Rachel

PA-Star: *A disk-assisted parallel A-Star strategy with locality-sensitive hash for multiple sequence alignment*; Sundfield, Daniel; Razzolini, Caina; Teodoro, George; Boukerche, Azzedine; de Melo, Alba; 2016

PNBA*: *A Parallel Bidirectional Heuristic Search Algorithm*; Rios, Luis; Chaimowicz, Luiz;

Artificial Intelligence A Modern Approach, 4 Ed, Norvig, Peter; Russel, Stuart

Networkz; Disponível em: https://networkx.org/documentation/stable/index.html