Uma proposta de uma variante otimizada do algoritmo A* para sistemas multi-núcleo

Carlos Pires[1], Paulo Shirley[2]

[1] Universidade Aberta de Portugal, Lisboa, Portugal
c.alexandre.pires@gmail.com
[2] Universidade Aberta de Portugal, Lisboa, Portugal
paulo.shirley@uab.pt

Resumo

Este artigo propõe uma variante otimizada do algoritmo A* para melhorar o desempenho em sistemas multi-núcleo. A abordagem proposta envolve a utilização de filas prioritárias locais (min-heaps) em cada tarefa ou núcleo, permitindo o processamento em paralelo. A comunicação entre as tarefas é realizada por meio de um buffer compartilhado do tipo produtor/consumidor, permitindo a troca de informações sobre os nós sucessores. Um protótipo é descrito, envolvendo a implementação das estruturas de dados, a lógica das tarefas, a comunicação entre as tarefas e a avaliação do desempenho em sistemas multi-núcleo. Os resultados preliminares mostram um ganho de desempenho em comparação com a versão sequencial do algoritmo A*.

Palavras-chave: algoritmo A*, otimização, sistemas multi-núcleo, paralelismo, filas prioritárias, comunicação entre tarefas.

Title: Proposal for an optimized variant of A* Algorithm for multi-core systems

Abstract: This paper proposes an optimized variant of the A* algorithm to improve performance in multi-core systems. The proposed approach involves the use of local priority queues (min-heaps) in each task or core, enabling parallel processing. Communication between tasks is facilitated through a producer/consumer buffer, allowing for the exchange of information regarding successor nodes. A prototype is described, covering the implementation of data structures, task logic, inter-task communication, and performance evaluation in multi-core systems. Preliminary results demonstrate a performance gain compared to the sequential version of the A* algorithm.

Keywords: A* algorithm, optimization, multi-core systems, parallel processing, priority queues, inter-task communication.

1. Introdução

A procura de caminhos em grafos é um problema fundamental em muitas aplicações como robótica, jogos de computador e sistemas de navegação [Russell 2010]. O algoritmo A* é amplamente utilizado para encontrar o caminho mais curto em grafos ponderados, combinando a procura em largura com uma heurística para estimar o custo restante até ao objetivo [Hart 1968, Russell 2010]. No entanto, em sistemas multi-núcleo, a eficiência do algoritmo A* pode ser comprometida devido à natureza sequencial da sua implementação.

Nos sistemas multi-núcleo modernos, onde vários núcleos de processamento podem ser utilizados simultaneamente, é essencial explorar o paralelismo para melhorar o desempenho. Este artigo tenta mostrar que a procura do caminho de menor custo com o algoritmo A* é paralelizável, uma vez que a exploração de nós adjacentes pode ser realizada de forma independente.

2. Contexto

O algoritmo A* utiliza uma heurística para guiar a procura, dando prioridade aos nós que parecem mais promissores para atingir o objetivo. Essa heurística, no entanto, introduz um desafio para a paralelização do algoritmo, uma vez que a heurística pode exigir acesso concorrente às informações dos nós abertos, além disso, a implementação sequencial do algoritmo A* geralmente mantém uma única lista de nós abertos, o que pode levar a contenção e consequente redução de desempenho em sistemas multi-núcleo, onde a concorrência é necessária para explorar diferentes partes do espaço de procura.

Há, portanto, uma necessidade de se desenvolver uma variante do algoritmo A* que possa tirar proveito do paralelismo oferecido pelos sistemas multi-núcleo, permitindo que várias tarefas processem de forma independente os nós e explorem o espaço de busca de maneira mais eficiente.

Neste artigo, propomos uma variante otimizada do algoritmo A* que utiliza filas prioritárias locais em cada tarefa para melhorar o desempenho em sistemas multi-núcleo. Além disso, abstraímos o algoritmo em relação à implementação do problema, separando as estruturas de dados para estados e nós, e introduzimos um mecanismo eficiente de comunicação entre as tarefas usando um buffer partilhado do tipo produtor/consumidor. A implementação do protótipo e a avaliação de desempenho são detalhadas nas seções subsequentes

3. Trabalho relacionado

No contexto dos trabalhos relacionados, destaca-se o algoritmo Hash Distributed A* (HDA*), uma extensão do algoritmo A* para ambientes de processamento paralelo [Kishimoto 2009, Weinstock 2016]. O HDA* utiliza uma função de hash para atribuir estados a processos individuais, permitindo a deteção de estados duplicados de forma eficiente e localizada, reduzindo a sobrecarga de comunicação. Este algoritmo representa uma contribuição significativa para a pesquisa de caminhos otimizados em ambientes

paralelos. O algoritmo proposto representa uma melhoria pois introduz abstrações do problema e da implementação do algoritmo.

4. Algoritmo A*

O algoritmo pode ser considerado como uma combinação do algoritmo de procura em largura (BFS) e do algoritmo de procura em profundidade (DFS) onde se utiliza uma heurística para direcionar a busca para as regiões mais promissoras do espaço de procura. Durante a procura, o A* mantém uma lista de nós abertos ou nós que ainda precisam ser explorados e seleciona o nó com menor custo total, custo este que é calculado da seguinte forma [Russel 2010]:

$$f(nó) = g(nó) + h(nó)$$

onde temos:

$g(nó)$ → Custo do caminho conhecido até o nó atual, calculado através da soma dos custos dos nós anteriores que fazem parte do caminho até ao nó atual, sendo utilizado para determinar qual o caminho mais curto durante a procura.

$h(nó)$ → Função heurística para estimar o custo do caminho restante até ao objetivo, sendo importante ter alguns cuidados ao escolher e implementar a heurística, esta deve ser admissível, ou seja, não deve sobrestimar a distância para o objetivo, deve ser escalável e relevante para o problema em questão. Ao levar em consideração esses cuidados, pode-se garantir que o algoritmo A* seja eficiente e eficaz em encontrar o caminho mais curto para o objetivo.

5. Proposta de otimização

A solução proposta visa melhorar o desempenho do algoritmo A* em sistemas multi-núcleo, utilizando filas prioritárias locais e comunicação entre as tarefas para processamento paralelo (fig.1). Para o desenvolvimento de um protótipo foi implementado um programa multitarefa em linguagem C recorrendo à biblioteca de tarefas *POSIX threads* [GNU Project] e tendo-se em conta os seguintes aspetos:

Divisão de tarefas: O espaço de busca é dividido entre as tarefas para garantir o processamento paralelo. Cada tarefa é responsável por uma parte do espaço de busca. A divisão do espaço é feita através da atribuição de um estado a uma tarefa através do *hash* [Drozdek 2013] dos dados do estado.

Fila prioritária local: Cada tarefa possui uma fila prioritária local, implementada como uma min-heap [Drozdek 2013], para armazenar os nós abertos que pertencem à sua parte do espaço de busca. A fila prioritária local é acedida exclusivamente pela própria tarefa.

Expansão local de nós: Cada tarefa retira o nó com menor custo da sua fila prioritária local, verifica se o estado a que o nó se refere é uma solução (através da chamada ao método *goal*), e caso este não seja, executa a expansão dos estados sucessores (através de uma chamada ao método *expand*). Os estados gerados e a referência do nó que os gerou são então enviados para a respetiva tarefa, baseadas no *hash* dos dados estado.

Comunicação entre tarefas: Cada tarefa recebe através do buffer do tipo produtor/consumidor uma mensagem que contêm um estado gerado e o nó pai que o

gerou, este método permite que os múltiplos estados gerados sejam processados por outras tarefas em paralelo.

Atualização das filas prioritárias: Cada tarefa ao receber os estados sucessores de outras tarefas, verifica se existe um nó para esse estado na *hashtable* [Drozdek 2013] e avalia a distância *g*, heurística *h* e custo total *f*. O nó é então inserido na fila local, caso não esteja presente, ou o custo na fila prioritária é atualizado se o novo caminho tiver um custo menor.

Convergência: O processo de expansão, comunicação e atualização das filas prioritárias continua até que o objetivo seja encontrado (1ª solução), ou até que todas as tarefas tenham nós existentes nas suas filas locais com custo estimado maior que a solução já encontrada (procura exaustiva), ou que não haja mais nós a serem processados.

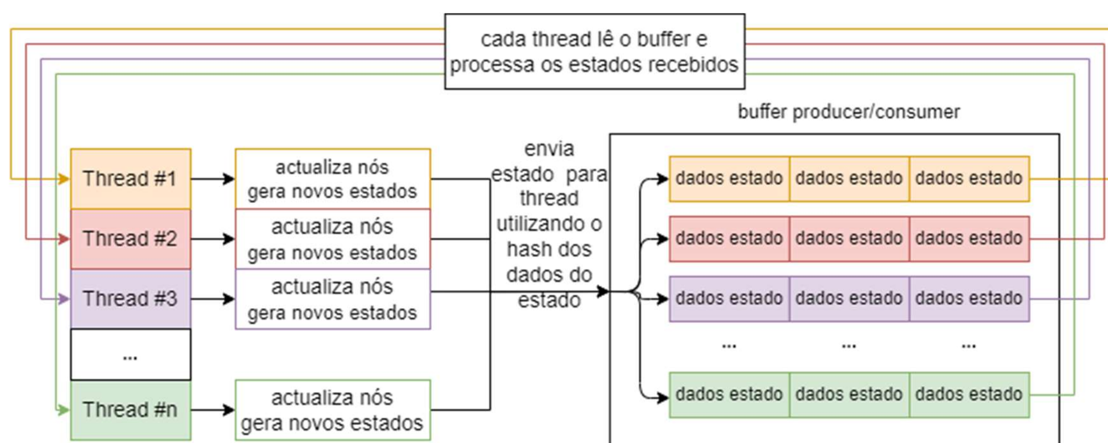

Figura 1. Visão geral do uso de um buffer produtor/consumidor no algoritmo paralelo

6. Detalhes da implementação

6.1 Estruturas de dados

Neste trabalho procedemos à implementação das duas versões do algoritmo, sequencial e paralelo, e de forma a podermos garantir uma comparação real de ambos os algoritmos, estes, partilham as mesmas estruturas de dados (fig.2).

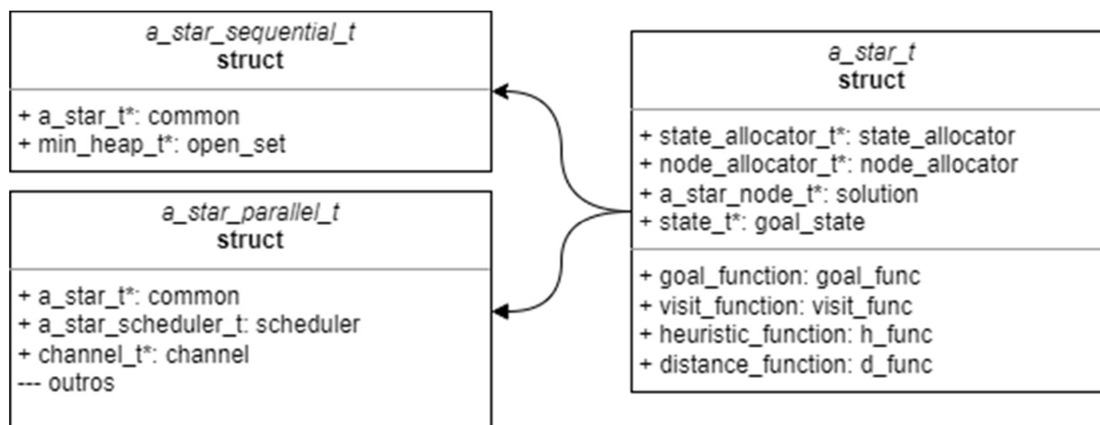

Figura 2. Estruturas de dados das versões sequencial e paralela

A implementação dos problemas também foi abstraída do algoritmo, separando-se assim as estruturas de dados utilizadas para gerir os estados do problema e os nós do algoritmo A* (fig.3).

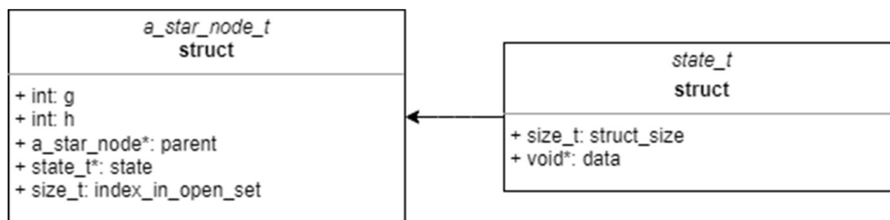

Figura 3. Estruturas de dados para nó e contentor de estado

O funcionamento é simples, um nó (*a_star_node_t*) aponta para um contentor de um estado (*state_t*), que por sua vez aponta para o respetivo estado do problema (fig.4).

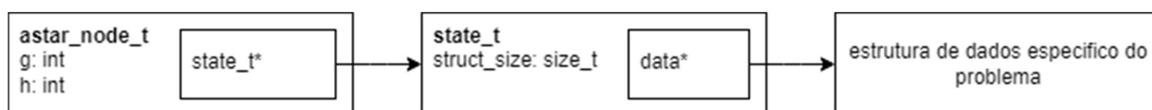

Figura 4. Ligação entre nó e o estado de um problema

Para ambas as estruturas de dados, *a_star_node_t* e *state_t*, foi implementado um alocador de memória (fig.5) e uma *hashtable* para garantir a indexação dos dados e evitar a geração de duplicados, *node_allocator_t* e *state_allocator_t*.

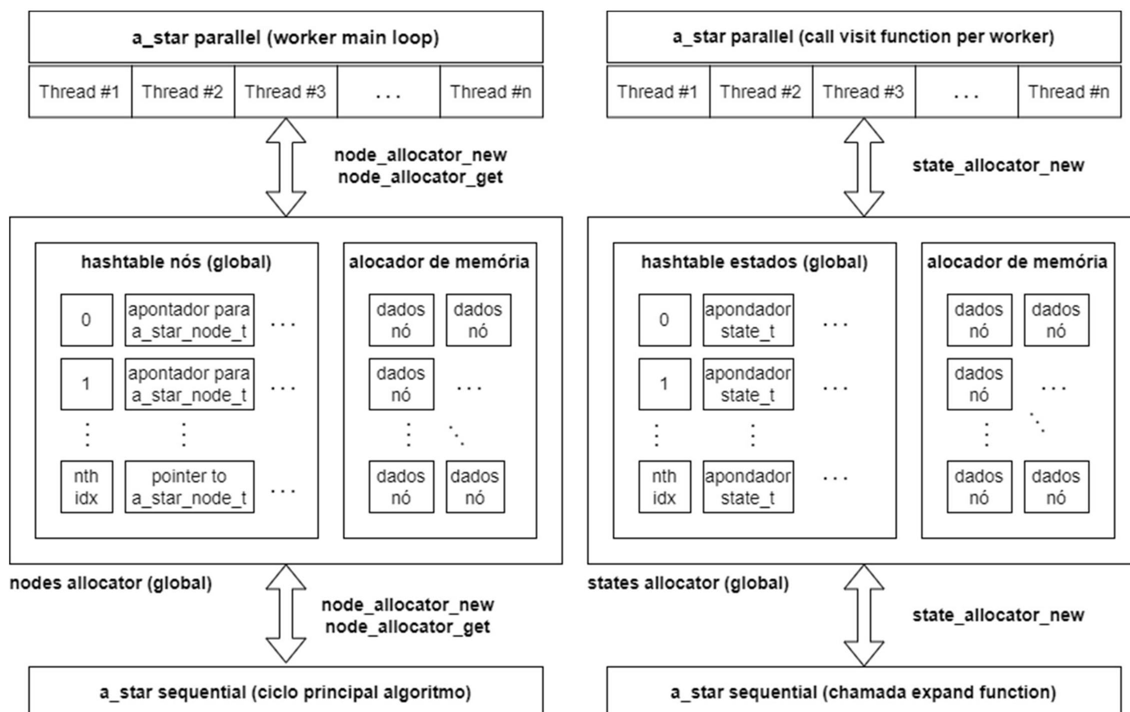

Figura 5. Comparação de acesso aos alocadores de nós e estados (sequencial e paralelo)

Ao garantirmos a unicidade de nós e estados podemos assumir que estamos a estabelecer uma bijeção entre o conjunto (A) de todos os nós necessários para a exploração do problema e conjunto (B) de todos os estados do espaço do problema (fig. 6), ou seja, a garantir que para cada estado de um problema existe um e um só nó e que cada nó se refere a um e apenas um estado.

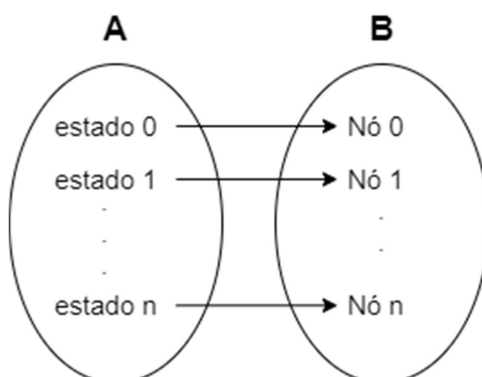

Figura 6. Bijeção entre o conjunto de estados do problema e o conjunto de nós do algoritmo

A bijeção está garantida pela forma como a *hashtable* está implementada, esta permite que seja redefinido o método de comparação entre dois elementos na *hashtable* e o método de *hashing* (fig.7).

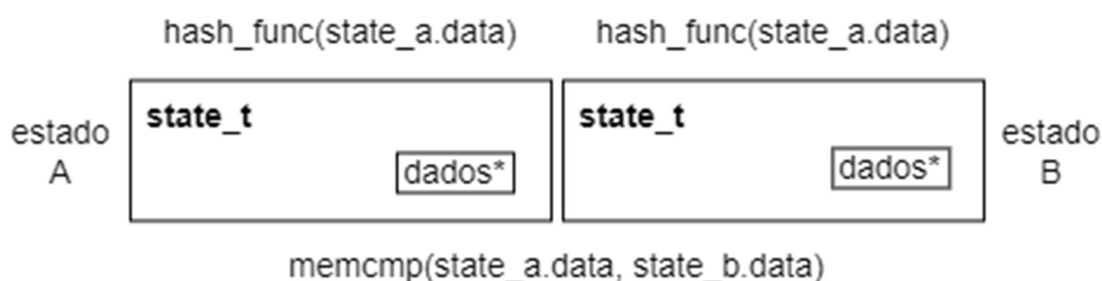

Figura 7. Método de comparação e *hashing* utilizado na *hashtable* do gestor de estados

Na *hashtable* referente aos estados, fazemos uma comparação entre duas zonas de memória (*memcmp*) apontadas pelo *pointer* existente (data) e para o *hashing* utilizamos os dados do estado.

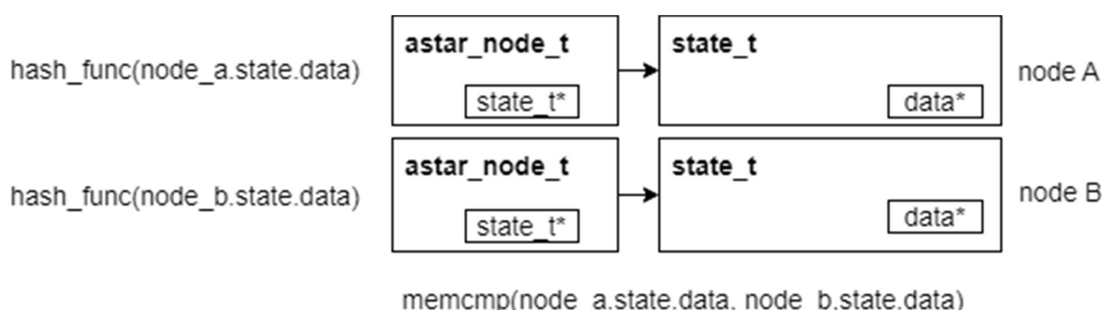

Figura 8. Método de comparação e *hashing* utilizado na *hashtable* do gestor de nós

No caso da *hashtable* dos nós, como cada nó aponta para um estado, o método de comparação é redefinido para que compare o conteúdo de memória dos estados e não dos nós, garantindo assim que existe apenas um nó para um estado e o mesmo para o *hashing* (fig.8).

Ao estabelecermos esta abstração e ao garantir a bijeção entre estados e nós, torna possível que o algoritmo se foque apenas na procura pelo espaço de soluções, e que a implementação do problema se foque na lógica dos estados e na heurística (fig.9).

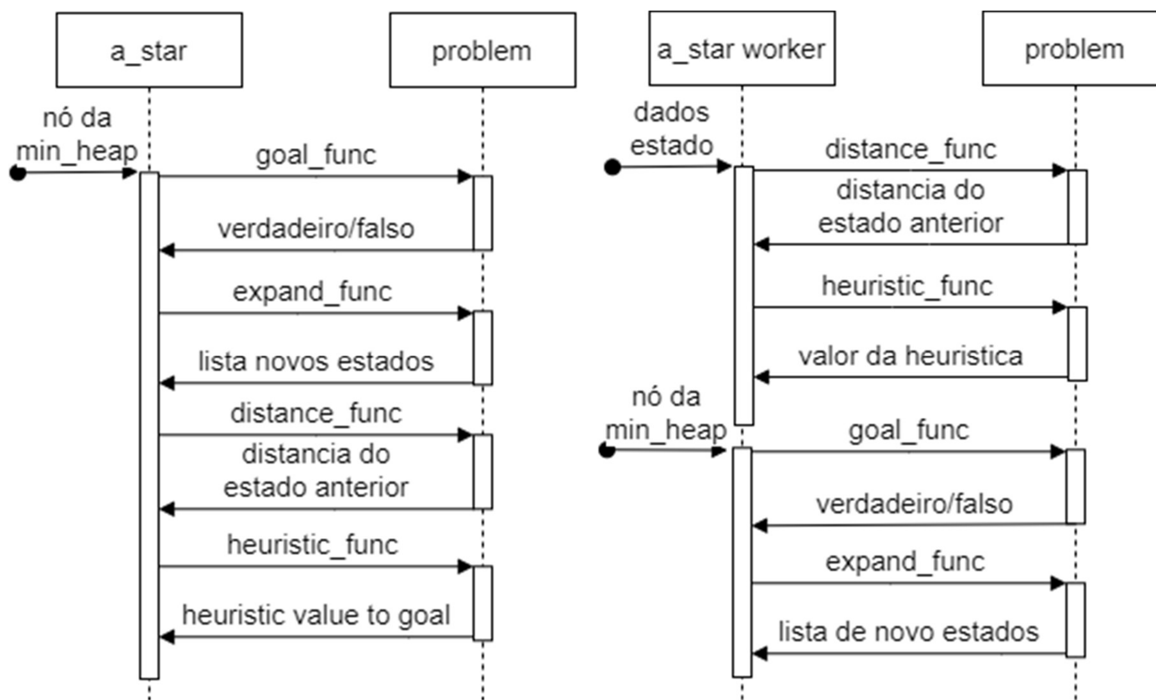

Figura 9. Diagrama de sequência das chamadas entre o algoritmo e um problema (sequencial e paralelo)

6.2. Abstração do problema

6.2.1. Interface

Para a implementação de cada problema é necessário implementar um conjunto de 4 métodos com assinaturas próprias e necessárias para que o algoritmo funcione corretamente.

```
// Cálculo da heurística do estado t para o estado n
typedef int (*heuristic_function)(const state_t*, const state_t*);
// Gerar estados sucessores do estado t
typedef void (*expand_function)(state_t*, state_allocator_t*, linked_list_t*);
// Verificar se o estado t é um objetivo do problema
typedef bool (*goal_function)(const state_t*, const state_t*);
// Distancia de um estado t para o seu vizinho n
typedef int (*distance_function)(const state_t*, const state_t*);
```

Código 1. Protótipos dos métodos necessários para implementação de um problema.

Os protótipos dos métodos *heuristic_function*, *goal_function* e *distance_function* são triviais, apenas recebem como argumentos estados, para que possam proceder à execução da lógica especifica do problema. O método *expand_function* já tem um protótipo mais complexo, não recebe apenas o estado "pai" para gerar os respetivos sucessores de acordo com a lógica do problema, recebe também o alocador de estados, para que um problema ao gerar um estado sucessor permita ao alocador geri-lo e indexá-lo de forma a garantir a sua unicidade.

6.2.2. Método *expand_function* em detalhe

Como referimos anteriormente para garantir que não existem estados duplicados, o alocador de estados é utilizado como argumento da função *expand_function*, assim, o problema pode se abstrair da gestão de um estado gerado. Um pseudocódigo para demonstrar como pode ficar a função *expand_function* na implementação de um problema genérico.

```
function expandFunction (estado, alocadorDeEstados, estadosGerados):
        estadoProblema = estado.dados
        for estadoPossivel in estadoProblema:
                // para cada estado possível é o gestor de estados que se encarrega
                // alocar/indexar o novo estado, caso um estado igual exista
                // retorna o estado já existente
                estadoGerado = alocadorDeEstados.novo(estadoPossivel)
                estadoGerados.adiciona(estadoGerado)
```

Pseudocódigo 1. Funcionamento de um método *expand* em que se garante a unicidade de estados.

6.3. Comunicação entre tarefas

Para comunicação entre tarefas de estados a processar, é utilizado um buffer do tipo produtor/consumidor. De forma que o algoritmo possa manter o espaço dos nós do grafo consistente é obrigatório que um estado tenha sempre um nó pai quando é processado, apenas assim o algoritmo A* pode funcionar de forma eficaz.

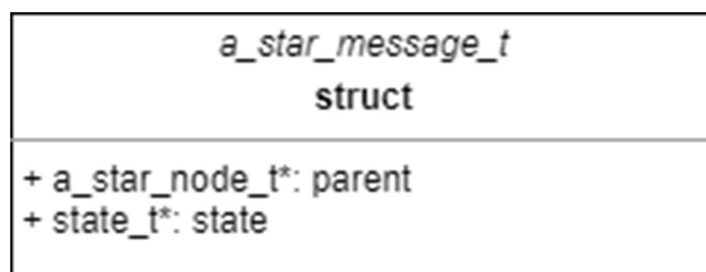

Figura 10. Estrutura da mensagem trocada entre tarefas

Uma tarefa após receber uma mensagem, ou seja, um estado a processar (fig.10), pode então executar a devida atualização do nó referente ao estado recebido (fig.11).

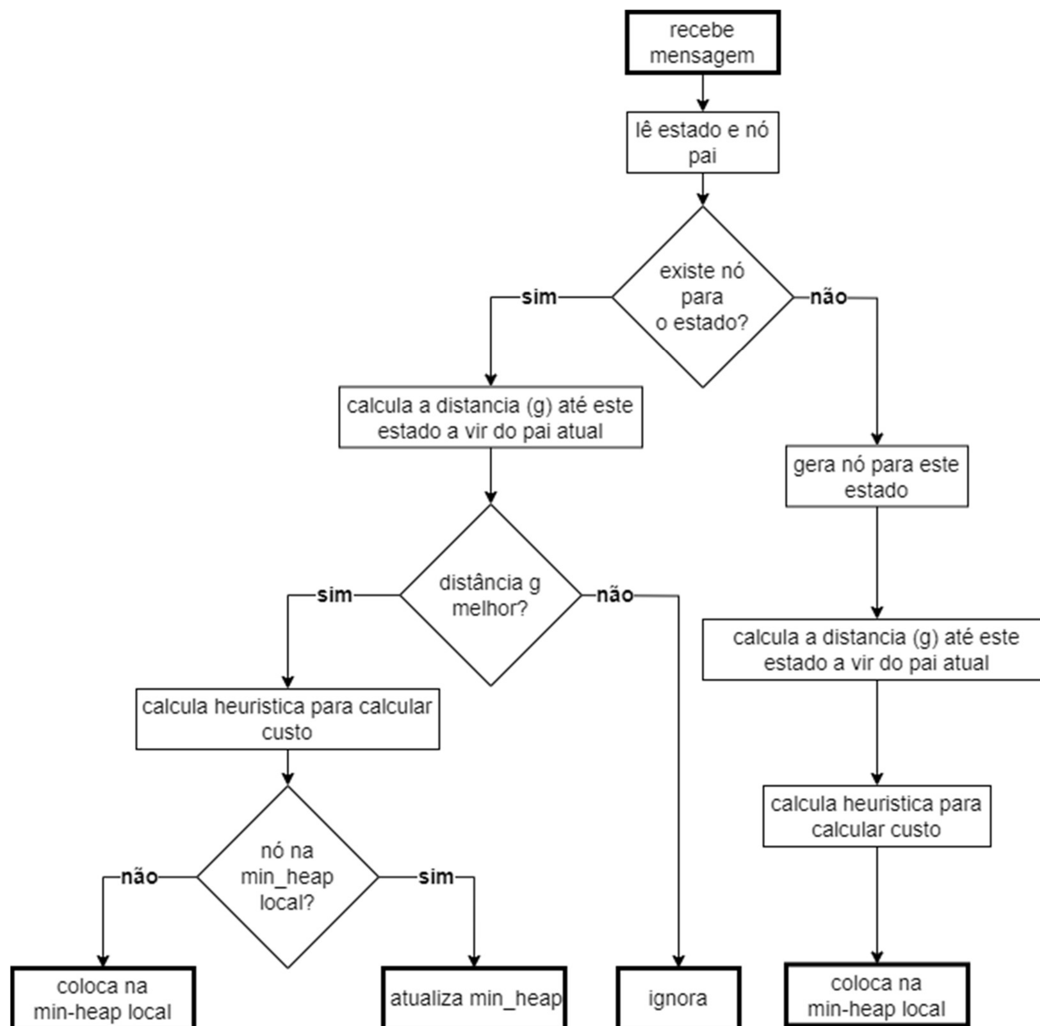

Figura 11. Fluxograma sobre processamento de um estado recebido por mensagem

6.4. Implementação do Algoritmo A*

6.4.1. Pseudocódigo da implementação sequencial

Como referimos anteriormente, ambas as formas do algoritmo partilham todas as estruturas de dados, e partilham a mesma arquitetura na abstração do código do problema, assim a versão sequencial do algoritmo é a seguinte:

```
function AStar (estadoInicial, objetivo):
        nosAbertos=[]
        nóInicial = novo_nó(estadoInicial)
        nóInicial.g = 0
        nóInicial.h = heuristica(estadoInicial,objetivo)
        nóInicial.f = nóInicial.g + nóInicial.h
        nosAbertos.adiciona(nóInicial)
        while nosAbertos is not empty:
                noAtual = retira_nó(nosAbertos) // com menor f
```

```
                // chamada ao problema para verificar se o estado
                // a que o nó aponta é a solução do problema
                if é_solução(noAtual.estado, objetivo):
                        soluçãoEncontrada = nóAtual
                        break
                // chamada ao problema para gerar os estados sucessores
                expand(noAtual.estado, alocador, sucessores)
                for sucessor in noAtual.sucessors:
                        // existe um nó referente a este estado sucessor?
                        nó = lê_na_hashtable_global(sucessor)
                        if nó == NULL:
                                nó = novo_nó(sucessor)
                                nó.parent = noAtual
                                nó.g = noPai.g + dist(noAtual.estado, sucessor)
                                nó.h = heuristica(sucessor, objectivo)
                                nó.f = no.g + no.h
                                nosAbertos.adiciona(nó)
                        else:
                                g_tentativa = noAtual.g + dist(noAtual.estado, sucessor)
                                if g_tentativa >= no.g:
                                        continue
                                nó.parent = nóPai
                                nó.g = g_tentativa
                                nó.h = heuristica(estado, objectivo)
                                nó.f = no.g + no.h
                                nosAbertos.atualiza(nó)
        return soluçãoEncontrada
```

Pseudocódigo 2. Algoritmo A* na sua forma sequencial

6.4.2. Pseudocódigo da implementação paralela

Apresentamos agora as partes mais importantes do pseudocódigo da versão paralela do algoritmo A*, é de notar que na implementação real existe uma pequena variação onde podemos forçar o algoritmo a não sair na primeira solução e esgotar a procura.

```
function AStar(estadoInicial, objectivo, nTrabalhadores):
        mensagem.pai = NULL
        mensagem.estado = estadoInicial
        envia_pelo_buffer(mensagem)
        inicia_trabalhadores(nTrabalhadores,objetivo)
        em_execução = true
        while em_execução:
                ociosos = numero_trabalhadores_ociosos()
                em_execução = ociosos < nTrabalhadores and not soluçãoEncontrada
        return soluçãoEncontrada
```

Pseudocódigo 3. Rotina principal do algoritmo A* paralelo

```
function numero_trabalhadores_ociosos_():
        ocisosos = 0
        for trabalhador in trabalhadores:
                if trabalhador.nosAbertos.tamanho > 0:
                        continue
                if mensagens_no_buffer(trabalhador):
                        continue
                ociosos++
        return ociosos
```

Pseudocódigo 4. Rotina de verificação se um trabalhador se encontra ocioso

```
function AStar_worker():
        nosAbertos=[]
        while em_execução:
                // Retiramos todas as mensagens existentes para este
                // trabalhador do buffer
                while mensagens_no_buffer(this):
                        // retiramos a mensagem e fazemos "unpack" dos dados
                        mensagem = obtem_mensagem_buffer(this)
                        nóPai = mensagem.pai
                        sucessor = mensagem.estado
                        if nóPai == NULL:
                                // estado inicial, pois não tem estado pai
                                // associamos um nó ao estado e adicionamos à
                                // min-heap local deste trabalhador
                                nóInicial = novo_nó(estadoInicial)
                                nóInicial.g = 0
                                nóInicial.h = heuristica(estadoInicial,objetivo)
                                nóInicial.f = nóInicial.g + nóInicial.h
                                nosAbertos.adiciona(nóInicial)
                                // saímos pois não existem mais mensagens
                                Break
                        // neste segmento executamos o código do algoritmo A*
                        // referente ao processamento de um estado sucessor e à
                        // respetiva atualização na min-heap local
                        // existe um nó referente a este estado sucessor?
                        nó = lê_na_hashtable_global(sucessor)
                        if nó == NULL:
                                nó = novo_nó(sucessor)
                                nó.parent = noPai
                                nó.g = noPai.g + dist(noPai.estado, sucessor)
                                nó.h = heuristica(sucessor, objectivo)
                                nó.f = nó.g + nó.h
                                nosAbertos.adiciona(nó)
                        else:
                                g_tentativa = noPai.g + dist(noPai.estado, sucessor)
                                if g_tentativa >= no.g:
                                        continue
```

```
                        nó.parent = nóPai
                        nó.g = g_tentativa
                        nó.h = heuristica(sucessor, objectivo)
                        nó.f = nó.g + nó.h
                        // neste segmento existe uma pequena diferença do
                        // algoritmo sequencial já que no caso paralelo um nó
                        // pode já ter sido retirado préviamente da min_heap
                        // local e nesse caso deve ser reintroduzido
                        if nó not in nosAbertos:
                                nosAbertos.adiciona(nó)
                        else:
                                nosAbertos.atualiza(nó)
        // o segmento que se segue é semelhante ao algoritmo sequencial
        // a diferença é que após os estados sucessores serem gerados eles
        // são imediatamente enviados para a tarefa respetiva
        if nosAbertos.tamanho > 0:
                noAtual = retira_nó(nosAbertos) // com menor f

                // chamada ao problema para verificar se o estado
                // a que o nó aponta é a solução do problema
                if é_solução(noAtual.estado, objetivo):
                        soluçãoEncontrada = nóAtual
                        break
                // chamada ao problema para gerar os estados sucessores
                expande(noAtual.estado, alocador, sucessores)
                for sucessor in noAtual.sucessors:
                        mensagem.pai = noAtual
                        mensagem.estado = sucessor
                        envia_pelo_buffer(mensagem)
```

Pseudocódigo 5. Rotina de um trabalhador, algoritmo A* adaptado

6.4.3. Comparação das versões (sequencial vs paralelo)

Conforme se pode observar no ponto anterior cada trabalhador executa o algoritmo A* adaptado, na generalidade o algoritmo é bastante próximo ao sequencial, apenas difere na ordem a que se procede ao cálculo de cada estado sucessor e no processo do nó com menor custo.

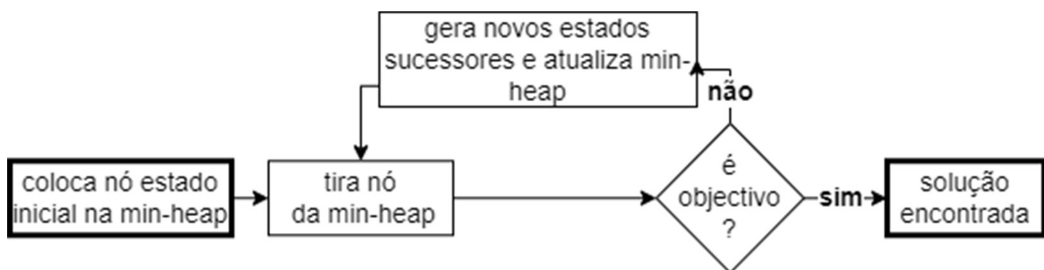

Figura 12. Versão sequencial do algoritmo

Na versão sequencial (fig.12) retiramos o nó da fila prioritária, verificamos se contêm o estado com a solução do problema, caso contrário, fazemos uma chamada ao problema para gerar os estados sucessores e proceder á atualização se necessário dos respetivos nós associado a cada estado gerado.

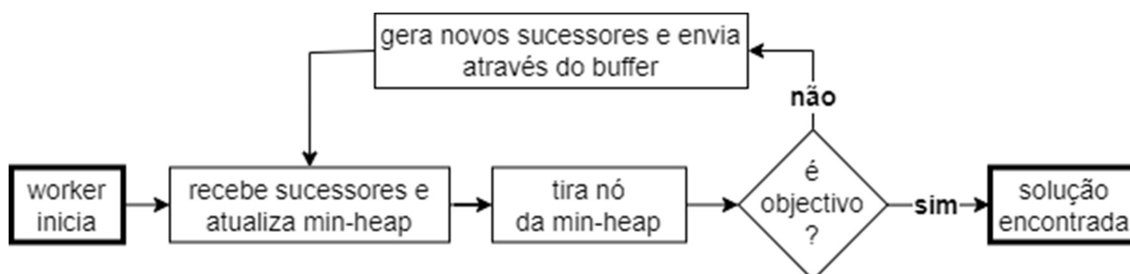

Figura 13. Versão paralela do algoritmo

Na versão paralela (fig.13), primeiro processamos os sucessores e só depois retiramos o nó da fila prioritária local, verificamos se contêm o estado com a solução do problema, caso contrário, fazemos uma chamada ao problema para gerar os estados sucessores enviando-os imediatamente pelo buffer para o trabalhador correspondente (baseado no *hash* dos dados do estado), para que este possa então proceder á atualização se necessário dos respetivos nós associado a cada estado gerado. É importante realçar que ao utilizarmos o *hash* para atribuir um estado a uma tarefa, estamos a garantir que um estado irá ser sempre tratado pela mesma tarefa, e isto confere estabilidade à procura.

7. Medições de desempenho

7.1. Problemas apresentados

Para a medição de desempenho entre as duas versões do algoritmo foram implementados três problemas distintos. O primeiro problema é um problema com uma heurística bastante conhecida e eficaz, o puzzle 8. O segundo problema foi escolhido de forma que a heurística não fosse a melhor e onde o espaço de procura fosse relativamente maior, o *numberlink*. O terceiro problema escolhido foi resolução de labirintos. Acredita-se que estes problemas irão permitir obter uma medição comparativa para os ganhos ou perdas introduzidas pela paralelização do algoritmo.

7.1.2. Problema Puzzle 8

O problema do puzzle 8 é um quebra-cabeças em que o objetivo é rearranjar os números em uma grelha 3x3 para alcançar uma configuração final. O A* é bastante eficaz na resolução do puzzle 8, encontrando o caminho mais curto para a solução final. A heurística utilizada, a distância de Manhattan, estima o custo restante para atingir a solução.

7.1.2. Problema *NumberLink*

O problema *Numberlink* passa por encontrar caminhos entre pares de números em um tabuleiro sem cruzamentos. Como heurística foi utilizada uma conjugação entre a distância de Manhattan e o número de pares ligados.

7.1.3. Problema Labirinto

O problema Labirinto passa por encontrar o caminho possível num labirinto NxN. Como heurística foi utilizada a distância Euclidiana entre a posição atual e a saída do labirinto.

7.2. Interpretação dos resultados

Para facilitar a interpretação das tabelas apresentadas deve-se ter em conta as seguintes definições:

- Estados gerados – quantidade de estados sucessores diferentes gerados durante a resolução de um problema.
- Estados explorados – quantidade de estados que foram expandidos pelo algoritmo.
- Heap – tamanho máximo que a heap teve durante a execução do algoritmo, nos algoritmos paralelos este valor é a media aritmética.
- Nós novos – número de nós criados e adicionados ao grafo, este valor deve ter o mesmo valor que o número de estados gerados.
- Nós reinseridos – nós que foram reinseridos na heap por ter sido encontrado um caminho melhor.
- Caminhos piores – caminhos que foram excluídos por haver outro caminho com menor custo.
- Caminhos melhores – caminhos que foram explorados (expandidos).
- Procura exaustiva – apenas termina a procura quando os nós existentes na fila prioritária local têm um custo superior à solução encontrada até ao momento.
- 1ª solução – termina a procura ao encontrar a primeira solução.
- *Speedup S=Ts/Tp* (Ganho de Velocidade) – Métrica para medir o desempenho de sistemas paralelos. É igual ao quociente do tempo *Ts* de execução da versão sequencial pelo tempo *Tp* da versão paralela. Tem como máximo teórico o número de processadores (núcleos) utilizados na execução do programa.

Nas tabelas, o campo *Threads* representa o número de tarefas do programa, sendo cada tarefa executada num processador (núcleo) diferente, ou seja, com uma relação tarefa**:** núcleo de 1**:**1.

8. Conclusões e trabalho futuro

8.1. Resultados principais

Os resultados mostram que o algoritmo A*, pela sua natureza, é bastante eficaz em direcionar a procura da solução no espaço de procura. Quando a heurística é ótima não se obtêm ganhos significativos de desempenho (*Speedup*) com a paralelização do algoritmo (caso do Puzzle 8). Os ganhos são visíveis quando a heurística não é boa e/ou espaço de procura é tão extenso que obriga à exploração de um número bem significativo de estados (*numberlink* e labirinto), ou quando a heurística é boa, contudo o problema não tem solução (puzzle 8 – impossível 1 e impossível 2), nesse caso o algoritmo paralelo é mais eficaz a explorar todo o espaço de procura.

Outro ponto a salientar é que todas as versões do algoritmo encontraram sempre a solução com o mesmo custo, isto indica que a versão paralela com procura exaustiva, ou seja, que apenas termina quando os nós existentes na fila prioritária local têm um custo superior à solução encontrada até ao momento, parece não ser necessária.

8.2. Limitações

Conclui-se que a paralelização do algoritmo A* só faz sentido quando temos problemas em que a heurística não é boa e/ou em que o espaço de procura é vasto. O que a implementação proposta mostra, é que tal paralelização é possível, e o algoritmo adaptado para sistemas multi-núcleo, conceptualmente, não é bastante diferente da sua versão sequencial.

8.2. Desenvolvimento futuro e melhorias

Uma nota final importante é que na nossa implementação foram utilizados métodos de sincronização de *threads* baseados em semáforos, isto pode estar na origem de alguma perda de desempenho, é possível que no caso em que se usem estruturas de dados onde a sincronização é gerida por operações atómicas [Sundell 2003], se obtenham resultados mais promissores.

A implementação apresentada pode ser encontrada no github no seguinte endereço https://github.com/alexyz79/1600024-projecto-informatica.

Referências

Drozdek, Adam. (2013). Data Structures and Algorithms in C++, Fourth Edition. Cengage Learning

GNU Project. (n.d.). pthreads: POSIX threads programming. Man pages. Retrieved from http://man7.org/linux/man-pages/man7/pthreads.7.html

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.

A. Kishimoto, A. S. Fukunaga, A. Botea, et al., "Scalable, parallel best-first search for optimal sequential planning.," in ICAPS, 2009

Russell, S. J., & Norvig, P. (2010). Artificial intelligence: a modern approach. Pearson Education.

Sundell, H., and Tsigas, P. 2003. Fast and Lock-free Concurrent Priority Queues For Multi-thread Systems. In Parallel and Distributed Processing Symposium, 2003. Proceedings. International, 11–pp. IEEE.

Ariana Weinstock & Rachel Holladay (2016). Parallel A* Graph Search, CMU 15-418/618 Parallel Computer Architecture and Programming final project.

Anexo
Resultados Computacionais

1. Tabelas - Problema Puzzle 8

Tabela 1.1. Puzzle 8: Medições versão sequencial do algoritmo.

Problema	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Tempo Execução
Fácil 1	sim	18	192	117	77	192	0	122	1	0,000136
Fácil 2	sim	20	442	268	176	442	0	274	2	0,000215
Fácil 3	sim	20	1034	646	390	1034	0	690	8	0,00042
Difícil 1	sim	31	12127	7920	4230	12127	20	8688	255	0,006205
Difícil 2	sim	31	13944	9123	4829	13944	6	9990	276	0,00756
Impossível 1	não	0	181394	182782	19597	181394	1387	293679	11868	1,423808
Impossível 2	não	0	181401	182676	19530	181401	1274	293614	11700	1,409324

Tabela 1.2. Puzzle 8: Medições versão paralela do algoritmo – procura exaustiva.

Problema	Threads	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Speed-up
Fácil 1	2	sim	18	395	249	150	395	0	252	1	0,489
Fácil 1	4	sim	18	430	273	167	430	0	273	3	0,564
Fácil 1	6	sim	18	482	301	191	482	0	303	2	0,496
Fácil 2	2	sim	20	624	385	242	624	0	400	4	0,566
Fácil 2	4	sim	20	704	440	273	704	0	454	6	0,632
Fácil 2	6	sim	20	919	587	343	919	0	610	10	0,513
Fácil 3	2	sim	20	1016	662	356	1016	0	670	9	0,741
Fácil 3	4	sim	20	1076	712	381	1076	1	715	11	0,727
Fácil 3	6	sim	20	1145	739	421	1145	1	757	11	0,845
Difícil 1	2	sim	31	20215	13501	6716	20215	50	15031	577	0,478
Difícil 1	4	sim	31	23198	15651	7599	23198	68	17433	714	0,626
Difícil 1	6	sim	31	20230	13665	6624	20230	70	15007	608	0,952
Difícil 2	2	sim	31	19008	12616	6395	19008	33	13991	498	0,646
Difícil 2	4	sim	31	24355	16512	7892	24355	77	18456	787	0,707
Difícil 2	6	sim	31	18011	12055	5981	18011	55	13118	493	1,353
Impossível 1	2	não	0	181397	192679	23575	181397	11281	304395	27642	1,325
Impossível 1	4	não	0	181384	196762	23617	181384	15377	311099	31382	2,432
Impossível 1	6	não	0	181376	190892	22763	181376	9514	302406	24655	3,764
Impossível 2	2	não	0	181395	192482	23141	181395	11086	304162	27378	1,293
Impossível 2	4	não	0	181386	197151	23396	181386	15764	311527	31964	2,383
Impossível 2	6	não	0	181377	190038	22390	181377	8659	301348	23532	3,739

Tabela 1.3. Puzzle 8: Medições versão paralela do algoritmo – primeira solução.

Problema	Threads	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Speed-up
Fácil 1	2	sim	18	352	218	138	352	0	223	1	0,544
Fácil 1	4	sim	18	424	263	169	424	0	269	3	0,586
Fácil 1	6	sim	18	423	263	171	423	0	265	2	0,551
Fácil 2	2	sim	20	615	379	240	615	0	393	5	0,587
Fácil 2	4	sim	20	748	465	294	748	0	484	7	0,629
Fácil 2	6	sim	20	764	479	298	764	1	498	9	0,558
Fácil 3	2	sim	20	945	587	363	945	0	618	7	0,842
Fácil 3	4	sim	20	877	547	340	877	0	572	8	1,027
Fácil 3	6	sim	20	1094	685	424	1094	1	717	9	0,921
Difícil 1	2	sim	31	18162	11967	6231	18162	30	13267	456	0,573
Difícil 1	4	sim	31	21094	14095	7093	21094	82	15745	661	0,719
Difícil 1	6	sim	31	18824	12472	6436	18824	70	13799	556	1,077
Difícil 2	2	sim	31	18115	11974	6186	18115	39	13303	492	0,688
Difícil 2	4	sim	31	23051	15443	7681	23051	62	17322	717	0,781
Difícil 2	6	sim	31	19599	13000	6671	19599	55	14418	568	1,255
Impossível 1	2	não	0	181391	192648	23643	181391	11256	304364	27573	1,324
Impossível 1	4	não	0	181385	196969	23789	181385	15582	311438	31608	2,434
Impossível 1	6	não	0	181385	191105	22817	181385	9718	302600	25045	3,648
Impossível 2	2	não	0	181396	192593	23158	181396	11196	304376	27489	1,282
Impossível 2	4	não	0	181381	196672	23296	181381	15289	310787	31421	2,383
Impossível 2	6	não	0	181381	190102	22352	181381	8720	301508	23563	3,737

.2. Tabelas – Problema *NumberLink*

Tabela 2.1. *NumberLink*: Medições versão sequencial do algoritmo.

Problema	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Tempo Execução
numberlink-1	sim	8	33	17	18	33	0	6	0	0,00004
numberlink-2	sim	12	158	44	116	158	0	19	0	0,00014
numberlink-3	sim	31	1723	434	1292	1723	0	589	0	0,001129
numberlink-4	sim	24	3482	774	2711	3482	0	1428	0	0,00191
numberlink-5	sim	22	112890	41277	71615	112890	0	41885	0	0,153251

Tabela 2.2. *NumberLink*: Medições versão paralela do algoritmo – procura exaustiva.

Problema	Threads	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Soluções	Soluções Piores	Soluções Melhores	Speed-up
numberlink-1	2	sim	8	63	54	27	63	0	32	0	5	4	1	0,294
numberlink-1	4	sim	8	59	55	24	59	0	29	0	4	3	1	0,339
numberlink-1	6	sim	8	56	54	21	56	0	30	0	4	3	1	0,336
numberlink-2	2	sim	12	951	561	380	951	0	387	0	41	40	1	0,145
numberlink-2	4	sim	12	819	507	318	819	0	349	0	35	34	1	0,152
numberlink-2	6	sim	12	798	500	318	798	0	380	0	34	33	1	0,184
numberlink-3	2	sim	31	20870	7086	13567	20870	0	13313	0	11	10	1	0,104
numberlink-3	4	sim	31	23600	8430	13784	23600	0	16984	0	18	17	1	0,127
numberlink-3	6	sim	31	27657	10733	17029	27657	0	23354	0	18	17	1	0,132
numberlink-4	2	sim	24	79084	22036	56739	79084	0	52368	0	48	47	1	0,035
numberlink-4	4	sim	24	81344	23528	56930	81344	0	56707	0	60	59	1	0,047
numberlink-4	6	sim	24	83651	24553	57778	83651	0	61332	0	66	65	1	0,051
numberlink-5	2	sim	22	751760	323640	4E+05	751760	0	349084	0	704	703	1	0,027
numberlink-5	4	sim	22	756512	328637	4E+05	756512	0	362807	0	704	703	1	0,049
numberlink-5	6	sim	22	759004	331311	4E+05	759004	0	369436	0	704	703	1	0,065

Tabela 2.3. *NumberLink*: Medições versão paralela do algoritmo – primeira solução.

Problema	Threads	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Soluções	Soluções Piores	Soluções Melhores	Speed-up
numberlink-1	2	sim	8	47	30	25	47	0	10	0	1	0	1	0,381
numberlink-1	4	sim	8	48	35	23	48	0	14	0	1	0	1	0,44
numberlink-1	6	sim	8	49	42	21	49	0	19	0	1	0	1	0,388
numberlink-2	2	sim	12	185	74	116	185	0	27	0	1	0	1	0,509
numberlink-2	4	sim	12	120	44	84	120	0	19	0	1	0	1	0,683
numberlink-2	6	sim	12	260	104	165	260	0	45	0	1	0	1	0,437
numberlink-3	2	sim	31	2226	540	1697	2226	0	674	0	1	0	1	0,73
numberlink-3	4	sim	31	1942	465	1526	1942	0	683	0	1	0	1	0,771
numberlink-3	6	sim	31	5263	1296	3977	5263	0	1676	0	1	0	1	0,632
numberlink-4	2	sim	24	5019	1165	3858	5019	0	1756	0	1	0	1	0,702
numberlink-4	4	sim	24	2205	479	1737	2205	0	539	0	1	0	1	1,473
numberlink-4	6	sim	24	609	124	496	609	0	123	0	1	0	1	2,412
numberlink-5	2	sim	22	120546	45243	75306	120546	0	49317	0	1	0	1	1,195
numberlink-5	4	sim	22	133863	52167	81701	133863	0	62067	0	1	0	1	1,681
numberlink-5	6	sim	22	140029	55241	84795	140029	0	66625	0	1	0	1	2,085

3. Tabelas – Problema Labirinto

Tabela 3.1. Labirinto: Medições versão sequencial do algoritmo

Problema	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Tempo Execução
maze-1 (10x10)	sim	38	45	45	2	45	0	42	0	1,7E-05
maze-8 (100x100)	sim	986	3639	3633	10	3639	0	3630	0	0,00162
maze-9 (200x200)	sim	3778	15399	15397	17	15399	0	15394	0	0,00911
maze-15 (800x800)	sim	32574	188126	188122	26	188126	0	188119	0	0,27894
maze-17 (1000x1000)	sim	50210	404494	404485	37	404494	0	404482	0	1,89221
maze-21 (5000x5000)	sim	928642	9652345	9652336	60	9652345	0	9652333	0	498,388
maze-22 (10000x10000)	sim	4805482	34240548	34240529	66	34240548	0	34240526	0	4113,54

Tabela 3.2. Labirinto: Medições versão paralela do algoritmo – procura exaustiva

Problema	Threads	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Speed-up
maze-1 (10x10)	2	sim	38	45	46	3	45	0	42	0	0,195
maze-1 (10x10)	4	sim	38	45	46	5	45	0	42	0	0,224
maze-1 (10x10)	6	sim	38	45	46	7	45	0	42	0	0,239
maze-8 (100x100)	2	sim	986	3650	3648	14	3650	0	3642	0	0,766
maze-8 (100x100)	4	sim	986	3711	3711	25	3711	0	3705	0	0,803
maze-8 (100x100)	6	sim	986	3747	3747	31	3747	0	3741	0	0,813
maze-9 (200x200)	2	sim	3778	15418	15418	26	15418	0	15413	0	0,803
maze-9 (200x200)	4	sim	3778	15432	15433	39	15432	0	15427	0	0,937
maze-9 (200x200)	6	sim	3778	15441	15442	48	15441	0	15436	0	1,058
maze-15 (800x800)	2	sim	32574	188176	188174	43	188176	0	188169	0	1,134
maze-15 (800x800)	4	sim	32574	188280	188278	67	188280	0	188272	0	1,647
maze-15 (800x800)	6	sim	32574	188302	188301	79	188302	0	188293	0	1,937
maze-17 (1000x1000)	2	sim	50210	404501	404494	69	404501	0	404488	0	1,263
maze-17 (1000x1000)	4	sim	50210	404532	404530	111	404532	0	404521	0	2,056
maze-17 (1000x1000)	6	sim	50210	404551	404552	120	404551	0	404539	0	2,681
maze-21 (5000x5000)	2	sim	928642	9652454	9652450	113	9652454	0	9652444	0	1,343
maze-21 (5000x5000)	4	sim	928642	9652454	9652453	213	9652454	0	9652445	0	2,212
maze-21 (5000x5000)	6	sim	928642	9652468	9652467	264	9652468	0	9652459	0	2,969
maze-22 (10000x10000)	2	sim	4805482	34240555	34240538	126	34240555	0	3,4E+07	0	1,371
maze-22 (10000x10000)	4	sim	4805482	34240826	34240814	237	34240826	0	3,4E+07	0	2,162
maze-22 (10000x10000)	6	sim	4805482	34241065	34241054	299	34241065	0	3,4E+07	0	2,74

Tabela 3.3. Labirinto: Medições versão paralela do algoritmo – primeira solução

Problema	Threads	Solução	Custo	Estados Gerados	Estados Explorados	Heap	Nós Novos	Nós Reinseridos	Caminhos Piores	Caminhos Melhores	Speed-up
maze-1 (10x10)	2	sim	38	45	46	3	45	0	42	0	0,23
maze-1 (10x10)	4	sim	38	45	46	5	45	0	42	0	0,212
maze-1 (10x10)	6	sim	38	45	46	7	45	0	42	0	0,254
maze-8 (100x100)	2	sim	986	3648	3646	14	3648	0	3641	0	0,771
maze-8 (100x100)	4	sim	986	3704	3704	25	3704	0	3698	0	0,799
maze-8 (100x100)	6	sim	986	3742	3743	31	3742	0	3736	0	0,814
maze-9 (200x200)	2	sim	3778	15415	15416	26	15415	0	15411	0	0,793
maze-9 (200x200)	4	sim	3778	15432	15432	42	15432	0	15427	0	0,938
maze-9 (200x200)	6	sim	3778	15442	15442	51	15442	0	15436	0	1,05
maze-15 (800x800)	2	sim	32574	188179	188177	42	188179	0	188172	0	1,133
maze-15 (800x800)	4	sim	32574	188282	188281	67	188282	0	188273	0	1,647
maze-15 (800x800)	6	sim	32574	188286	188285	78	188286	0	188277	0	1,966
maze-17 (1000x1000)	2	sim	50210	404501	404494	69	404501	0	404489	0	1,265
maze-17 (1000x1000)	4	sim	50210	404464	404462	110	404464	0	404454	0	2,059
maze-17 (1000x1000)	6	sim	50210	401829	401829	122	401829	0	401822	0	2,68
maze-21 (5000x5000)	2	sim	928642	9652431	9652427	111	9652431	0	9652422	0	1,318
maze-21 (5000x5000)	4	sim	928642	9644501	9644500	213	9644501	0	9644493	0	2,24
maze-21 (5000x5000)	6	sim	928642	9615459	9615459	280	9615459	0	9615453	0	2,996
maze-22 (10000x10000)	2	sim	4805482	34240556	34240542	124	34240556	0	3,4E+07	0	1,385
maze-22 (10000x10000)	4	sim	4805482	34162579	34162568	231	34162579	0	3,4E+07	0	2,17
maze-22 (10000x10000)	6	sim	4805482	34085706	34085695	282	34085706	0	3,4E+07	0	2,747

Carlos Pires, Finalista no curso de licenciatura em Engenharia Informática da Universidade Aberta ano letivo 2022/23. Trabalha no ramo das tecnologias de informação desde o ano 2000, tendo realizado variados projetos no âmbito de infraestrutura de TI, sistemas distribuídos e sistemas na nuvem. Tem como áreas de interesse engenharia de sistemas, algoritmos e segurança de sistemas informáticos.

Paulo Shirley, Professor Auxiliar no Departamento de Ciências e Tecnologia (DCeT), Secção de Informática, Física e Tecnologia (SIFT). Licenciado em Engenharia Electrotécnica e de Computadores em 1988 pelo IST-UTL. Obteve os graus de Mestre (perfil de Controlo e Robótica) e de Doutor em Eng. Electrotécnica e de Computadores, pelo IST-UTL em 1993 e 2003 respetivamente. Tem como áreas de interesse, a intersecção da Informática (Computer Science) com a área do Controlo Automático, nomeadamente a área da "Computação de Alto Desempenho" (HPC) aplicada a problemas de otimização e cálculo científico.