UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

PAULO MIRANDA E SILVA SOUSA

PERSISTENT A*: UMA MELHORIA DO ALGORITMO A* USANDO ÁRVORE DE SEGMENTO PERSISTENTE E HASHING INCREMENTAL

QUIXADÁ

2022

PAULO MIRANDA E SILVA SOUSA

PERSISTENT A*: UMA MELHORIA DO ALGORITMO A* USANDO ÁRVORE DE SEGMENTO PERSISTENTE E HASHING INCREMENTAL

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Wladimir Araújo Tavares

QUIXADÁ

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S698p Sousa, Paulo Miranda e Silva.
Persistent A*: uma melhoria do algoritmo A* usando Árvore de Segmento Persistente e Hashing incremental / Paulo Miranda e Silva Sousa. – 2022.
49 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Computação, Quixadá, 2022.
Orientação: Prof. Dr. Wladimir Araújo Tavares.

1. Algoritmos. 2. Estrutura de dados (Computação). 3. Inteligência Artificial. I. Título.

CDD 621.39

PAULO MIRANDA E SILVA SOUSA

PERSISTENT A*: UMA MELHORIA DO ALGORITMO A* USANDO ÁRVORE DE SEGMENTO PERSISTENTE E HASHING INCREMENTAL

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em: ____/____/_____

BANCA EXAMINADORA

Prof. Dr. Wladimir Araújo Tavares (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Paulo Henrique Macêdo de Araújo
Universidade Federal do Ceará (UFC)

Prof. Dr. Fábio Carlos Sousa Dias
Universidade Federal do Ceará (UFC)

Dedico este trabalho aos meus pais por tudo que fizeram e fazem por mim.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer aos meus pais, Elza Maria e Beroaldo Rodrigues, por tudo que já fizerem e ainda fazem por mim. Eles são as minhas maiores motivações.

Gostaria de agradecer também as minhas irmãs, Paloma Miranda e Patrícia Miranda, por todo o apoio e por sempre acreditarem no meu potencial.

A minha namorada, Paula Luana, que sempre me apoiou e fez esses anos de graduação serem mais felizes.

Ao professor Dr. Wladimir Araújo Tavares por me orientar em meu trabalho de conclusão de curso e me apresentar a competição da maratona de programação que mudou a minha vida.

Aos professores Fábio Carlos Sousa e Paulo Henrique Macêdo de Araújo por aceitarem participar da minha banca avaliadora e por contribuirem com várias sugestões de melhoria para este trabalho.

Aos meus amigos do grupo Resistência, Alan Nascimento, David Tavares, Doug Nóbrega, Gregório Neto, Jorge Lucas, Natan Nobre e Ruan Felipe por fazerem essa jornada parecer mais leve e divertida.

Aos amigos do PET - TI, que fizeram parte de vários momentos importantes da minha graduação, em especial, Rafaela Fernandes, Robertty Freitas e Wilton Neto.

Aos integrantes da minha equipe da Maratona de Programação, Claro Henrique e Doug Nóbrega, por além de serem grandes amigos, toparam estudar toda semana e realizar junto comigo o sonho de conseguir ir para uma mundial da ICPC.

A toda a minha família, mãe, pai, irmãs, tios, tias, primos, sobrinhos, avós e amigos, por sempre acreditar e torcer por cada conquista. Em especial as minhas duas avós, Antônia e Maria Dedice, e ao meu tio Seu Ná, que perdi durante a minha graduação, mas que torceram sempre por mim e que mesmo não estando mais aqui em terra, sei que manterei sempre vivos no meu coração.

Ao Doutorando em Engenharia Elétrica, Ednardo Moreira Rodrigues, e seu assistente, Alan Batista de Oliveira, aluno de graduação em Engenharia Elétrica, pela adequação do *template* utilizado neste trabalho para que o mesmo ficasse de acordo com as normas da biblioteca da Universidade Federal do Ceará (UFC).

Hoje, eu sei que ter escolhido ir para Universidade Federal do Ceará - Campus Quixadá foi a melhor escolha que fiz na vida. Obrigado a todos que fizeram parte dessa jornada.

“Se eu vi mais longe, foi por estar sobre ombros de gigantes.”

(Isaac Newton)

RESUMO

A busca no espaço de estados é um método usado para resolver vários problemas computacionais aplicado em várias áreas da computação, em especial na inteligência artificial. Um dos algoritmos mais estudados dessa área é o A*. Tal algoritmo de busca utiliza informações heurísticas para realizar uma melhor exploração do espaço de busca. Entretanto, o A* possui uma limitação de memória quando aborda problemas cuja representação do estado da busca é grande. Para contornar esse problema, este trabalho apresenta o *Persistent A**, cuja finalidade é reduzir a complexidade de memória e de tempo da implementação clássica do A*. Isso foi possível utilizando a técnica de *hashing* incremental e a estrutura de dados chamada árvore de segmento persistente. Para validar a implementação, foram feitos testes usando várias instâncias do problema $(N^2 - 1)$ - *Puzzle*. Por fim, fizemos uma comparação entre alguns algoritmos da literatura.

Palavras-chave: Algoritmos. Estruturas de Dados. Inteligência Artificial.

ABSTRACT

State space search is a method used to solve several computational problems applied in several areas of computing, especially in Artificial Intelligence. One of the most studied algorithms in this area is A*. Such search algorithm uses heuristic information to perform a better exploration of the search space. However, A* has a memory limitation when it addresses problems whose search state representation is large. To circumvent this problem, this work presents the *Persistent A**, whose purpose is to reduce the memory and time complexity of the classic A* implementation. This was possible using the incremental *hashing* technique and the data structure called persistent segment tree. To validate the implementation, tests were performed using several instances of the problem $(N^2 - 1)$ - Puzzle. Finally, we made a comparison between some algorithms in the literature.

Keywords: Algorithms. Artificial Intelligence. Data Structures.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS E SIGLAS

IDA*	Iterative Deepening A*
IA	Inteligência Artificial
PDB	Pattern Database
PNBA*	Parallel New Bidirectional A*
NBA*	New Bidirectional A*

SUMÁRIO

1 INTRODUÇÃO

A busca no espaço de estados é um método usado para resolver vários problemas computacionais aplicado em várias áreas da computação, em especial na Inteligência Artificial (IA). Um problema no espaço de estado pode ser caracterizado por uma tupla $P = (S, A, s, T)$, onde S é um conjunto de estados, $s \in S$ é o estado inicial, $T \subseteq S$ é um conjunto de estados finais e $A = \{a_1, a_2, \ldots, a_n\}$ é um conjunto finito de ações, tal que cada $a_i : S \to S$ é uma função que transforma um estado em outro. Uma solução para o problema é uma sequência de ações que levam do estado inicial para algum estado final com uma determinada propriedade.

Cada *estado* de um problema representa uma possível configuração do problema. A partir do conjunto de estados do problema, podemos construir um *grafo de estados* em que dois estados s_i e s_j estão conectados se existe uma ação que pode ser executada transformando o estado s_i no estado s_j. O problema de busca no espaço de estado difere do problema de busca simples porque o conjunto de todos os estados é implícito, ou seja, o grafo do espaço de estados é muito grande para ser armazenado na memória. Nesses casos, os estados são gerados à medida que eles são explorados.

Para ficar claro alguns conceitos, considere o seguinte problema:

Um fazendeiro comprou um lobo, um carneiro e uma alface. No caminho de casa, o fazendeiro chegou à margem de um rio e alugou um barco. Ele observou que o barco só pode levar ele mesmo e uma de suas compras. O lobo e o carneiro não podem ser deixados sozinhos no mesmo lado da margem, porque o lobo come o carneiro. O carneiro e a alface não podem ser deixados sozinhos na mesma margem, porque o carneiro come a alface. Como o fazendeiro pode atravessar a si mesmo e suas compras de maneira segura para o outro lado do rio com o menor número de travessias?

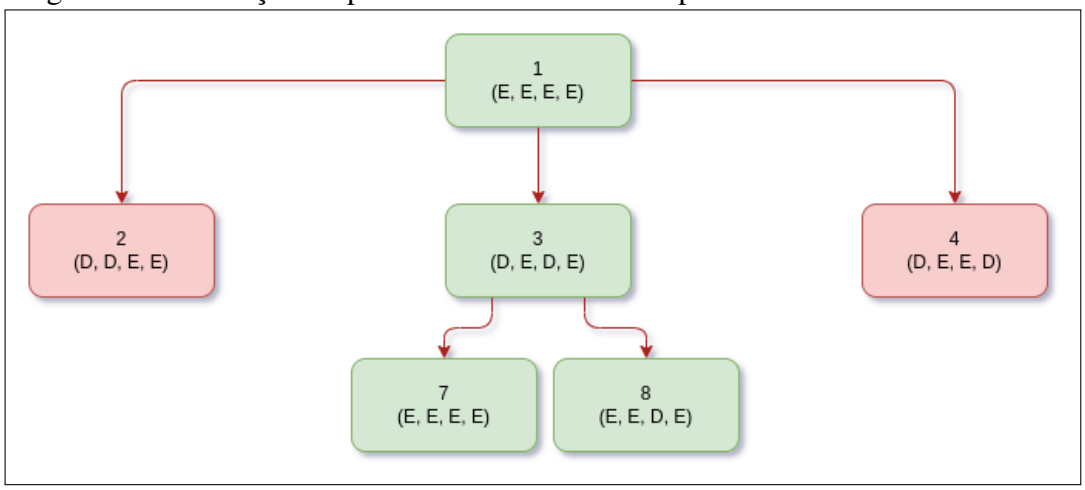
Um estado desse problema pode ser modelado como uma tupla de quatro valores:

$(lado\_fazendeiro, lado\_lobo, lado\_carneiro, lado\_alface)$

O problema começa com todos no lado esquerdo do rio representado pelo seguinte estado: (E, E, E, E). A meta é levar todos para o lado direito do rio representado pelo seguinte estado: (D, D, D, D).

Na Figura 1, temos todos os estados alcançados usando até 2 passos. Os estados em vermelhos são aqueles que invalidam os requisitos. O estado 2 é inválido, pois o carneiro comeu a alface. Já o estado 4 é inválido, pois o lobo comeu o carneiro. Os estados em verde são os estados válidos.

Figura 1 – Simulação do problema efetuando até 2 passos



Fonte: elaborado pelo autor (2021) utilizando o site draw.io.

Uma possível solução do problema do fazendeiro com o menor número de travessias é a seguinte:

$$(E,E,E,E) \Rightarrow (D,E,D,E) \Rightarrow (E,E,D,E) \Rightarrow (D,D,D,E) \Rightarrow (E,D,E,E) \Rightarrow (D,D,E,D) \Rightarrow (E,D,E,D) \Rightarrow (D,D,D,D)$$

Uma solução é considerada ótima se ela é uma solução que possui o menor custo dentre todas as soluções válidas possíveis. No caso do problema acima, consideramos que o custo é igual à quantidade de movimentos realizados do estado inicial até o final.

Existem duas principais categorias de métodos de buscas utilizado em problemas de espaço de estado: busca não-informada e busca informada.

Na busca não-informada, não utilizamos nenhuma informação sobre o quão perto do objetivo cada estado está. Já na busca informada, utilizamos uma estimativa de quão perto do estado meta um estado se encontra. Essas estimativas são chamadas funções heurísticas sendo usadas para guiar os algoritmos. Os algoritmos A* e IDA* são dois exemplos de algoritmos da categoria de busca informada que utilizam essas informações para encontrar uma solução de forma mais eficiente.

De modo geral, esses algoritmos exploram um largo espaço de busca seguindo uma sequência de ações até alcançar o objetivo. Um dos algoritmos mais estudados dessa área é o A*. Tal algoritmo de busca utiliza funções heurísticas para realizar uma melhor exploração do espaço de busca. Se essas funções respeitam algumas condições, então é provado que o A* encontrará a solução ótima (RIOS; CHAIMOWICZ, 2011).

As funções heurísticas podem reduzir o número de estados visitados sem precisar sacrificar a otimalidade da solução. Entretanto, dependendo das características do problema

ou da função heurística, a complexidade de tempo do A* cresce de forma exponencial. Várias extensões e variações do A* foram propostas para melhorar o desempenho do algoritmo em diferentes cenários (RIOS; CHAIMOWICZ, 2010).

Ao executar os algoritmos de busca, podemos encontrar um estado já visitado que chamaremos estado redundante. Na Figura 1, os estados 1 e 7 são redundantes. Para detectar se um estado é redundante precisamos realizar uma busca na lista dos estados já visitados pelo algoritmo. Para facilitar essa busca podemos mapear um estado para um valor numérico (*hashing*) e, com isso, utilizar estruturas de dados como Tabela de Dispersão ou Árvore de Busca Balanceada para verificar elementos repetidos. Contudo, o cálculo do *hashing* pode ser o gargalo da exploração para problema que possuem um estado cuja quantidade de informações necessárias para representá-lo seja grande (MEHLER; EDELKAMP, 2006).

No problema acima, podemos calcular o valor *hashing* de um estado qualquer usando *hashing* polinomial. Hashing polinomial é uma ferramenta muita utilizada em problemas de *matching* de *string*. Dado um estado qualquer (a,b,c,d), adotando $E = 0$ e $D = 1$, podemos calcular a função hashing h de uma tupla com $h((a,b,c,d)) = a \cdot 2^0 + b \cdot 2^1 + c \cdot 2^2 + d \cdot 2^3$. Assim, a tupla (D, E, D, E) teria valor *hashing* igual a $h((D,E,D,E)) = 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 = 1 + 4 = 5$.

Mesmo que o tempo para gerar um sucessor do estado atual seja $O(1)$, uma abordagem de *hashing* tradicional terá custo de $O(K)$ para ser computada, onde K é o tamanho do vetor que representa o estado. Para solucionar esse problema, em alguns casos, é possível calcular um valor de *hashing* a partir de outro cálculo já feito, essa técnica é chamada *hashing* incremental.

Usando o mesmo problema como exemplo, dado que já calculamos o hashing de (a,b,c,d) e precisamos calcular o valor para (a,b,x,y), não é necessário refazer todo o cálculo, podemos reaproveitar o que já foi calculado. Logo, temos que $h((a,b,x,y)) = h((a,b,c,d)) - c \cdot 2^2 - d \cdot 2^3 + x \cdot 2^2 + y \cdot 2^3$. Quanto maior for a representação do estado, maior será o ganho ao usar essa abordagem. Com isso, é possível reduzir de $O(K)$ para $O(1)$, a transição entre estados sucessores.

Entretanto, ao utilizar *hashing* pode acontecer de dois estados diferentes gerar o mesmo valor de *hashing*, chamamos isso de colisão. Algumas soluções para colisões necessitam que seja armazenada o estado. Assim, teremos o custo de $O(K)$ para comparar essas informações. Uma possível solução é escolher o método de busca parcial, o qual a otimalidade pode ser perdida com a finalidade de melhorar o desempenho da busca. Apesar de não ser possível

garantir a otimalidade, a probabilidade de colisão pode ser muito pequena dependendo da função de *hashing*. Uma forma simplificada de estimar a probabilidade é dado pela fórmula: $\frac{C}{M}$, em que C é quantidade de estados visitados e M é o módulo usado no *hashing* polinomial. Com isso, é possível calcular o próximo estado em $O(1)$ (EDELKAMP; MEHLER, 2004).

Entretanto, a técnica de *hashing* incremental funciona bem para algoritmos como IDA*, pois os estados são explorados utilizando a estratégia LIFO (*Last In, First Out*). Nessa estratégia, é possível implementar o algoritmo de forma recursiva. Assim, antes de chamar para os estados sucessores, é feito o ajuste no valor do *hashing* e, após processá-lo e voltar da recursão, essa alteração é desfeita. Com isso, é possível transitar de um estado para outro em $O(1)$.

Já com o A* isso não é possível, pois o próximo estado a ser explorado pode ser um estado totalmente diferente do anterior. Isso acontece, pois o A* processa o estado mais promissor entre os estados ainda não explorados.

Neste trabalho, propomos o algoritmo *Persistent A**, cuja finalidade é reduzir a complexidade de memória e de tempo da implementação clássica do A*. Isso foi possível utilizando a técnica de hashing incremental e a estrutura de dados chamada árvore de segmento persistente. Com essa estrutura, podemos recuperar qualquer estado já visitado com custo de $O(log(K))$, onde K é o tamanho do vetor que representa o estado. Assim, o cálculo do novo estado pode ser melhorado de $O(K)$ para $O(log(K))$ tanto em memória quando em tempo de execução para o algoritmo A*.

Além disso, analisamos o comportamento da abordagem de *hashing* incremental no algoritmo A* e comparamos com a mesma abordagem no algoritmo IDA*. Com a utilização da árvore de segmento persistente conseguimos superar as limitações relacionadas com a utilização do *hashing* incremental no algoritmo A* melhorando o tempo de execução e a memória utilizada.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo geral deste trabalho é propor e avaliar uma solução para a redução da memória e tempo consumido no algoritmo de A* utilizando árvore de segmento persistente e função de *hashing*.

1.1.2 Objetivos Específicos

Como objetivos específicos estão:

1. Propor e avaliar uma abordagem para reduzir a complexidade de memória e de tempo do algoritmo A* clássico utilizando uma árvore de segmento persistente.
2. Implementar uma árvore de segmento persistente para representar um estado da busca.
3. Investigar o comportamento da abordagem de *hashing* incremental no algoritmo proposto e compará-la com a mesma abordagem nos outros algoritmos da literatura.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta seção, serão apresentados alguns dos principais conceitos para o entendimento e desenvolvimento deste trabalho.

2.1 Algoritmo A*

O algoritmo A^* é um algoritmo de busca heurística clássico proposto em (HART *et al.*, 1968). Nesse algoritmo, os estados são avaliados conforme a combinação de $g(n)$, o custo do início até alcançar o estado n, e $h(n)$, o custo estimado para ir do estado n até o objetivo. Sendo expresso por: $f(n) = g(n) + h(n)$. Com isso, podemos interpretar $f(n)$ como sendo o custo estimado para solução de menor custo do início até o destino passando pelo estado n (RUSSEL; NORVIG, 2013).

Como estamos buscando encontrar a solução de menor custo, é razoável pensar que para encontrar a solução ótima é melhor tentar primeiro o estado cujo $f(n)$ é o menor possível. Essa estratégia pode retornar a solução ótima se a função $h(n)$ possuir algumas condições (RUSSEL; NORVIG, 2013).

A primeira condição para otimalidade é que a função $h(n)$ tem que ser admissível. Uma função heurística é admissível quando ela nunca supera o custo ótimo para alcançar o objetivo. Além disso, a segunda condição é chamada de consistência. Uma função heurística $h(n)$ é dita consistente se para cada estado u e para todo sucessor v gerado a partir de uma ação x, o custo estimado de alcançar o objetivo a partir de u não é maior que o custo da operação x somado com o custo de alcança o objetivo a partir de v, ou seja, $h(u) \leq custo(x) + h(v)$. É mais comum uma função não ser consistente quando o problema possui custos negativos (RUSSEL; NORVIG, 2013).

Bu e Korf (2021) apontam que o algoritmo A* tem duas vantagens principais:

- Capacidade de podar nós utilizando as listas de nós abertos e nós fechados.
- Ordenação dos nós para a realização da expansão utilizando a função f.

Além disso, Bu e Korf (2021) apontam que a principal desvantagem do algoritmo A* é a sua exigência de espaço exponencial uma vez que ele armazena na memória todos os nós gerados durante a busca. Por isso, A* pode preencher 8 GB de memória em poucos minutos em domínios de planejamento e pesquisa heurística comuns.

O algoritmo A* também pode ser visto como um algoritmo de busca informado que

encontra o menor caminho entre dois vértices em um grafo. O termo informado diz respeito ao fato de ser possível estimar o quão perto do objetivo o vértice atual está. Isso ajuda a guiar o algoritmo, fazendo o A* ter um comportamento melhor que o algoritmo de Dijkstra na maioria dos casos. A eficiência do A* está diretamente relacionada com a função de estimativa. O algoritmo de Dijkstra pode ser visto como sendo o A* com $h(n) = 0$.

No Algoritmo 1, mostramos como o A* pode ser implementado. A instrução *PriorityQueue.new* cria uma fila de prioridade de tuplas com 4 posições. Dada a tupla (*f, g, h, s*), os elementos são ordenados usando a abordagem lexicográfica. O f representa a estimativa para resolver o problema, o g é o custo para chegar no estado, o h é o valor da função heurística para o estado e o s é o estado da busca.

Ao analisar o algoritmo, vemos uma similaridade muito grande com o algoritmo de *Dijkstra*, a principal diferença está no critério de ordenação da fila de prioridade.

Algoritmo 1: Pseudocódigo do A*

```
Function aStar(init, final):
    dist[init] = 0
    pq = PriorityQueue.new
    pq.insert(0 + getH(init), 0, getH(init), init)
    while !pq.empty() do
        [f0, g0, h0, state] = pq.top()
        pq.pop()
        if state == final then
            return g0
        end
        if g0 > dist[state] then
            continue
        end
        for each to in state.neighbors() do
            newH = getH(to)
            if g0 + 1 < dist[to] then
                dist[to] = g0 + 1
                pq.insert(dist[to] + newH, dist[to], newH, to)
            end
        end
    end
    return -1
```

Fonte: elaborado pelo autor.

Neste trabalho, o A* é o algoritmo utilizado para realizar a busca de uma solução ótima para o problema do $(N^2 - 1) - puzzle$.

2.2 Algoritmo IDA*

O IDA* é um algoritmo de busca por aprofundamento iterativo proposto por (KORF, 1985). Em cada iteração é escolhido um dos vizinhos para ser explorado de forma recursiva até atingir um valor de corte. Para realizar o corte nesse algoritmo, é utilizado a mesma função $f(n) = g(n) + h(n)$ do A*. Em cada iteração, o valor de corte é o máximo valor de $f(n)$ que um nó pode alcançar durante uma busca. Assim, para encontrar uma solução, iniciamos o valor de corte em 1 e esse valor é incrementado enquanto a solução não é encontrada (RUSSEL; NORVIG, 2013).

O IDA* é prático para vários problemas que possuem custos unitários e, além disso, evita a sobrecarga em relação à manutenção de uma fila de prioridade, como é feita no A* (RUSSEL; NORVIG, 2013).

Bu e Korf (2021) apontam que o algoritmo IDA* tem como vantagem uma exigência linear de memória em relação ao valor de corte. A principal desvantagem do IDA* seria gerar muitos estados duplicados.

No Algoritmo 2, mostramos como o IDA* pode ser implementado.

Algoritmo 2: Pseudocódigo do IDA*

```
Function dfs(state, final, g, h, limit):
    if g + h > limit then
        return false
    end
    if state == final then
        return true
    end
    for each to in state.neighbors() do
        newH = getH(to)
        response = dfs(to, final, g + 1, newH, limit)
        if response then
            return true
        end
    end
    return false
End Function
Function IDAStar(init, final):
    initH = getH(init)
    for it=1; true; it++ do
        if (dfs(init, final, 0, initH, it)) then
            return it
        end
    end
    return -1
End Function
```

Fonte: elaborado pelo autor.

A implementação do IDA* é utilizada neste trabalho com a finalidade de comparar com a implementação do A*. Ambas utilizam a abordagem de *hashing* incremental.

2.2.1 IDA com Memorização*

O IDA* com Memorização é uma versão mais eficiente em tempo de execução, contudo menos eficiente em memória sido inspirado em (CHAKRABARTI *et al.*, 1989; KORF, 1993; RUSSELL, 1992). A ideia é basicamente salva o menor valor de g alcançado por um estado até o momento, e se um mesmo estado é visitado com valor igual ou maior, a recursão é interrompida.

Com isso, o algoritmo evita ramificações desnecessárias. Entretanto, a complexidade de memória fica linear em relação à quantidade de estados visitados.

No Algoritmo 3, podemos ver como o IDA* como Memorização pode ser implementado. A principal mudança é a adição de um dicionário *seen* que vai dizer o menor valor de g alcançado por aquele estado durante a recursão. Esse dicionário é resetado a cada iteração e seu

valor padrão é infinito.

Algoritmo 3: Pseudocódigo do IDA* com Memorização

```
Function dfs(state, final, g, h, limit):
    if g + h > limit then
        return false
    end
    if state == final then
        return true
    end
    if seen[state] <= g then
        return false
    end
    seen[state] = g
    for each to in state.neighbors() do
        newH = getH(to)
        response = dfs(to, final, g + 1, newH, limit)
        if response then
            return true
        end
    end
    return false
End Function
Function IDAStar(init, final):
    initH = getH(init)
    for it=1; true; it++ do
        seen.reset()
        if (dfs(init, final, 0, initH, it)) then
            return it
        end
    end
    return -1
End Function
```

Fonte: elaborado pelo autor.

2.3 Função *Hashing*

Uma função *hashing* é uma função que mapeia uma entrada de tamanho variado em uma saída de tamanho fixa (STEVENS *et al.*, 2007). Funções *hashing* possuem diversas aplicações na área da computação. Essa função pode ser usada para validar a integridade de um dado. Além disso, é muito usado em estruturas de dados como conjuntos e dicionários, onde a função da *hashing* é responsável por codificar o endereço para onde uma determinada chave vai ficar na estrutura. Além das aplicações clássicas, também podemos usá-lo para fazer *hashing* do espaço de estado.

2.3.1 Função de Hashing Polinomial

Função de *hashing* polinomial é muito utilizada para fazer *hashing* de *string*, foi criado por (KARP; RABIN, 1987) para resolver problemas de *maching* de *strings*. Seja $s[1..n]$ uma *string* qualquer. Uma forma de fazer o cálculo de sua *hashing* está na Equação 2.1:

$$\text{hash}(s) = s[0] + s[1] \cdot p + s[2] \cdot p^2 + ... + s[n-1] \cdot p^{n-1} \mod m = \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m \quad (2.1)$$

Os valores de p e m podem ser escolhidos pelo projetista da função *hashing*. Entretanto, existem algumas recomendações. Primeiro, é recomendado que p e m sejam primos. Isso ajuda a reduzir as chances de colisões. Além disso, quando maior o m, menor é a probabilidade de colisão. Por fim, é vantajoso mapear os valores de $s[i]$ para valores entre 1 e $p-1$.

2.3.2 Hashing do Espaço de Estado

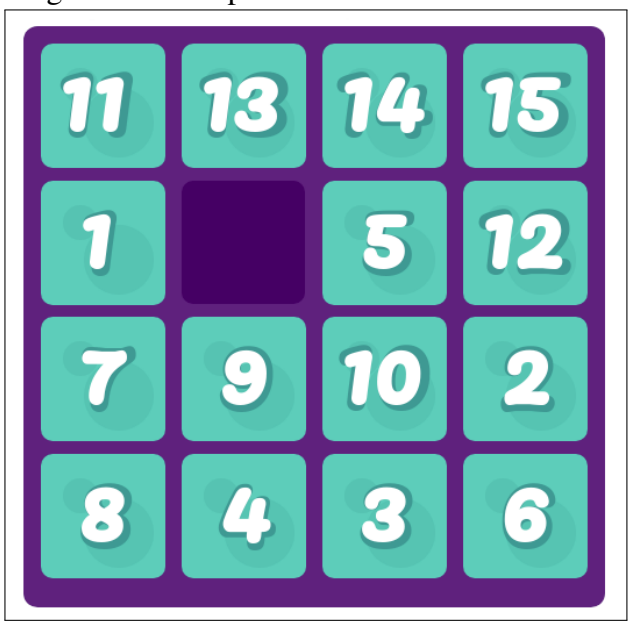
A *hashing* também pode ser expandida para espaço de estado. Primeiro, podemos mapear a representação de um estado para um vetor. Após isso, fazemos o *hashing* desse vetor (EDELKAMP; MEHLER, 2004). Para ficar mais claro, podemos ver o exemplo da Figura 2. O vetor desse estado seria $V = (11,13,14,15,1,0,5,12,7,9,10,2,8,4,3,6)$. Além disso, podemos escolher $p = 16$ e $m = 10^9 + 7$. Assim, temos que o valor de *hashing* desse estado é $(11 \cdot 16^0 + 13 \cdot 16^1 + 14 \cdot 16^2 + 15 \cdot 16^3 + 1 \cdot 16^4 + 0 \cdot 16^5 + 5 \cdot 16^6 + 12 \cdot 16^7 + 7 \cdot 16^8 + 9 \cdot 16^9 + 10 \cdot 16^{10} + 2 \cdot 16^{11} + 8 \cdot 16^{12} + 4 \cdot 16^{13} + 3 \cdot 16^{14} + 6 \cdot 16^{15}) \, mod \, 10^9 + 7$.

2.3.3 Hashing Incremental

A vantagem de usar *hashing* polinomial é que em alguns problemas a mudança no vetor é pontual, ou seja, apenas algumas posições são alteradas (EDELKAMP; MEHLER, 2004). Assim, seja um vetor v_1 e a hashing polinomial $h(v_1)$. Suponha agora v_2 igual a v_1 em todas as posições exceto pelas posições i e j que foram trocadas. Usando a ideia de *hashing* incremental podemos calcular $h(v_2)$ em $O(1)$ reaproveitando o valor de $h(v_1)$. Com isso, temos que $h(v_2) = h(v_1) - v[i] \cdot p^i - v[j] \cdot p^j + v[j] \cdot p^i + v[i] \cdot p^j \; mod \; m$. Isso pode ser facilmente implementado pre-calculando os valores de p^i para todo $0 \leq i < n$, onde n é o tamanho do vetor v_1 e v_2.

A *hashing* incremental será utilizada neste trabalho para verificar de forma eficiente

Figura 2: Exemplo de um estado do 15-Puzzle



Fonte: elaborado pelo autor (2021) utilizando o site 15puzzle.netlify.app.

se um estado já foi visitado e, assim, evitar ramificações desnecessárias. Dado o valor da *hashing*, podemos utilizar uma tabela de dispersão para verificar se o estado já foi processado ou não.

2.4 (N^2-1)-Puzzle

Na década de 1870, Sam Loyd causou uma grande agitação nos Estados Unidos e Europa com seu quebra-cabeça 15-puzzle. O 15-puzzle consiste em tabuleiro com dimensão 4x4. Nesse jogo, existem 15 posições preenchidas com valores de 1 até 15 e uma célula vazia. Para vencer o jogo é necessário deixar o tabuleiro com uma configuração específica. Em geral, consideramos como configuração final a forma apresentada na Figura 3.

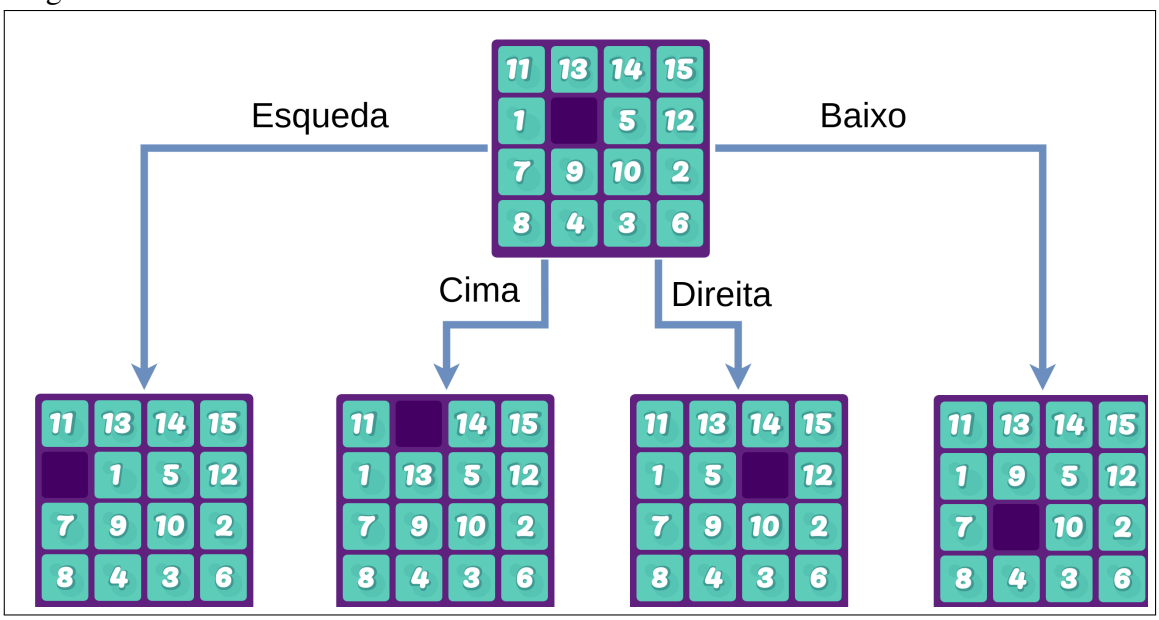
Como isso, dado uma configuração inicial você pode fazer alguns movimentos utilizando a célula vazia. Você pode trocar a posição da célula vazia com alguma célula que possui um lado em comum. Para mais clareza, veja o exemplo na Figura 4, em que temos uma configuração inicial e as configurações alcançáveis após um movimento. O desafio que este trabalho tenta resolver é encontrar o menor número de passos para solucionar o problema.

O problema (N^2-1)-*Puzzle* é uma versão generalizada do $15-puzzle$ no qual temos células numeradas de 1 até N^2-1 e um espaço vazio, no tabuleiro NxN. O objetivo do jogo é deixar o tabuleiro em uma ordenação crescente de modo que o espaço vazio ocupe a última posição do tabuleiro (BALL; COXETER, 2016). O problema de encontrar a solução com o menor número de movimentos foi provada ser NP-Completo por (RATNER; WARMUTH, 1986).

Figura 3: Configuração do 15-Puzzle desejada

Fonte: elaborado pelo autor (2021) utilizando o site 15puzzle.netlify.app.

Figura 4: Movimentos no 15-Puzzle



Fonte: elaborado pelo autor (2021) utilizando o site 15puzzle.netlify.app.

Drogoul e Dubreuil (1993) aponta que o 15-puzzle tem sido usado por anos para teste de técnicas de buscas devido à sua simplicidade de manipulação e por ser um bom representante para uma classe particular de problemas.

Na literatura, podemos encontrar diversas técnicas de buscas aplicadas ao problema 15-puzzle:

– **Aprofundamento Iterativo**: Korf (1985) combina o algoritmo A* com a técnica de aprofundamento iterativo criando o algoritmo IDA*. Ele testou o A* e o IDA* utilizando 100

instâncias geradas de maneira aleatória. Nos testes computacionais realizados o algoritmo IDA* resolveu todas as instâncias, enquanto o A* não resolveu nenhuma instância, pois o algoritmo ficou sem memória. O tamanho médio da solução nessas instâncias geradas aleatórias foi de 53 movimentos.

– **Técnicas de Gerenciamento de Memória**: Reinefeld e Marsland (1994) propõe técnicas para melhorar a exploração das informações obtidas nas profundidades anteriores. No problema 15-puzzle, evitar ciclos e transposições (quando diferentes caminhos terminam na mesma posição). Os ciclos e as transposições podem ser evitadas guardando a posição e o custo em uma tabela.

– ***Pattern Database (PDB)***: Culberson e Schaeffer (1998) propõe a utilização da base de dados de padrões para o problema 15-puzzle. A utilização de PDB consiste em pré-computar a distância de um conjunto de padrões. Em seguida, essas distâncias são usadas para definir heurísticas admissíveis para o problema. A utilização do pattern databases juntamente com IDA^* obtém uma grande redução do tamanho da árvore de busca. Mais recentemente, Felner e Adler (2005) melhora a técnica PDB permitindo o crescimento dinâmico da base de padrões aplicando no problema 24-puzzle conseguindo uma melhoria em 40 vezes do algoritmo anterior.

– **Heurísticas Admissíveis Mais Poderosas**: Korf e Taylor (1996) apresentam técnicas para obter heurísticas admissíveis mais poderosas. Além disso, ele mostra como obter novas heurísticas admissíveis a partir da heurística de *Manhattan*. Ele encontra soluções ótimas para um conjunto de 10 instâncias do problema 24-puzzle.

– ***State Space Hashing***: Edelkamp e Mehler (2004) propõe a utilização de *hashing* como método para endereçar e recuperar os estados já visitados de maneira eficiente. Como esta técnica, cada estado $S \in \mathscr{S}$ é associado a uma chave $key(S)$. O cálculo $key(S)$ pode ser otimizado realizando o cálculo de maneira incremental de $key(S)$. Para evitar armazenar os estados já visitados para o tratamento de colisões podemos sacrificar a completude da busca optando por um método de busca parcial armazenando apenas $key(S)$. Nesse artigo, ele realiza algumas considerações para o problema (N^2-1)-*puzzle* das técnicas

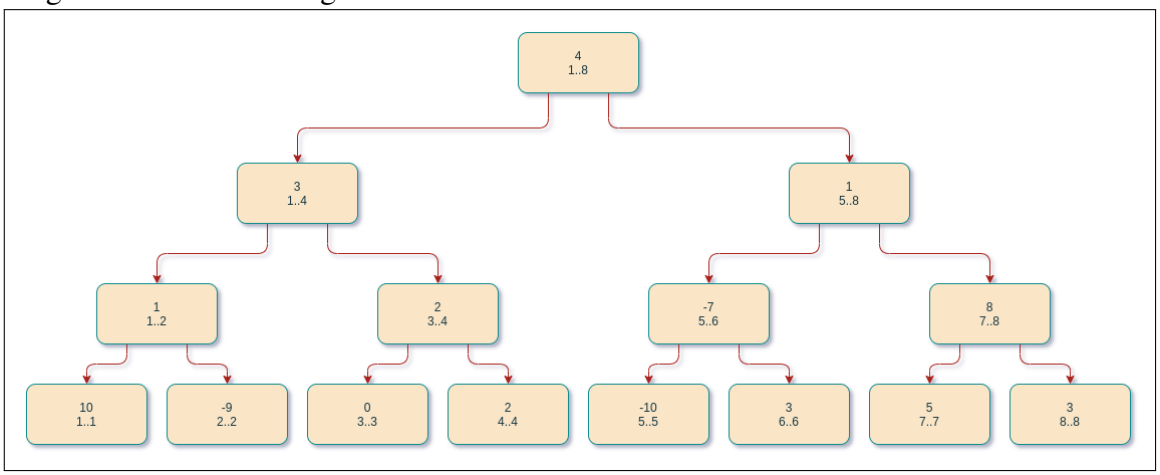
supracitadas.

2.4.1 *Árvore de Segmento*

Árvore de segmento é uma estrutura de dados que permite fazer consultas em um intervalo e atualização de uma posição do vetor em $O(log(K))$, onde K é o tamanho do vetor (BERG *et al.*, 1997; MEHTA; SAHNI, 2004; PREPARATA; SHAMOS, 2012). Uma aplicação clássica da árvore de segmento é poder atualizar qualquer posição do vetor e, também, ser possível descobrir a soma dos elementos no subvetor $V[i..j]$. Tudo é feito em $O(log(K))$ por operação. Além da soma, a árvore de segmento resolve $f(V[i..j])$ para qualquer operação f que seja associativa.

A árvore de segmento utiliza a ideia de divisão e conquista. A árvore de recursão é semelhante à implementação recursiva do *merge sort*. Entretanto, ao invés de fazer o *merge* dos vetores ordenados, na árvore de segmento você irá fazer $f(valor\_esquerdo, valor\_direiro)$, onde f é a operação desejada. A construção da árvore de segmento do vetor $[10, -9, 0, 2, -10, 3, 5, 3]$ para operação de soma pode ser observada na Figura 5, onde cada nó possui o range que abrange e o valor da soma dos elementos naquele intervalo.

Figura 5: Árvore de Segmento de soma



Fonte: elaborado pelo autor (2021) utilizando o site draw.io.

Para atualizar uma posição do vetor basta primeiro encontrar o nó folha que representa aquela posição. Após atualizar a folha, será necessário atualizar todos os nós afetados com essa modificação. Os nós afetados são todos os ancestrais do nó atualizado. Assim, basta refazer os cálculos de forma *bottom-up*. Isso pode ser facilmente implementado em $O(log(K))$.

Para fazer uma consulta nessa árvore, você pode aproveitar do fato de que alguns

segmentos já foram calculados. Assim, para calcular o valor de $f(V[i..j])$ basta quebrar os segmentos em partes já conhecidas e depois faz a união dessas informações. Pela forma construída, é possível quebrar o subvetor $V[i..j]$ em x partes, tal que todas as partes do vetor estão na árvore. Além disso, escolhendo os intervalos de forma ideal, temos que x é da ordem de $O(log(K))$.

No Algoritmo 4, mostramos como a árvore de segmento pode ser implementada. Para construir, basta chamar a função $build(1,1,k,v)$, onde o primeiro 1 é a posição da raiz, v é o vetor e k é o tamanho do vetor v. Para fazer uma consulta da soma dos elementos de $v[a..b]$, basta chamar a função $query(1,1,k,a,b)$. Por fim, para fazer uma atualização da posição idx para x, basta fazer $update(1,1,k,idx,x)$.

A árvore de segmento tradicional não é usada neste trabalho. Apresentamos os seus conceitos com a finalidade de fundamentar a explicação da árvore de segmento persistente.

Algoritmo 4: Pseudocódigo do Árvore de Segmento para Operação de Soma

```
Function build(node, i, j, v):
    if i == j then
        st[node] = v[i]
        return
    end
    m = (i + j) / 2
    l = (node*2)
    r = l + 1
    build(l, i, m)
    build(r, m + 1, j)
    st[node] = st[l] + st[r]
End Function
Function query(node, i, j, a, b):
    if (i > b) or (j < a) then
        return 0
    end
    if (a <= i) and (j <= b) then
        return st[node]
    end
    m = (i + j) / 2
    l = (node*2) r = l + 1
    return query(l, i, m, a, b) + query(r, m + 1, j, a, b))
Function update(node, i, j, idx, value):
    if i == j then
        st[node] = value
        return
    end
    m = (i + j) / 2
    l = (node*2)
    r = l + 1
    if idx <= m then
        update(l, i, m, idx, value)
    end
    else
        update(r, m + 1, j, idx, value)
    end
    st[node] = st[l] + st[r]
End Function
```

Fonte: elaborado pelo autor.

2.4.2 Árvore de Segmento Persistente

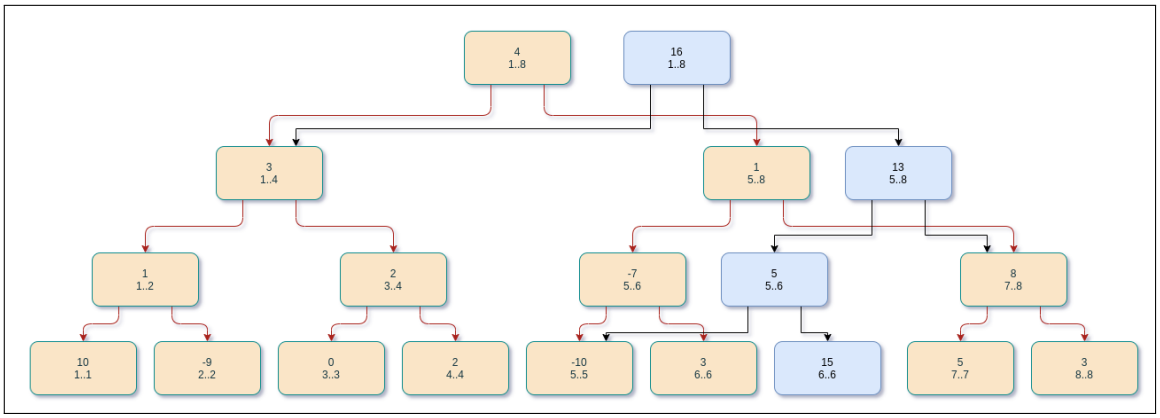
Uma estrutura de dados é persistente se possui suporte para acessar múltiplas versões. Essas versões são geradas após alguma atualização. Existem dois tipos de persistência: parcial e total. A persistência parcial ocorre quando todas as versões podem ser consultadas, mas só a última versão pode ser atualizada. Já a persistência total acontece quando todas as versões

podem ser consultadas e atualizadas (DRISCOLL *et al.*, 1989).

É possível fazer a persistência total da árvore de segmento de forma eficiente em $O(log(K))$ de memória e de tempo, onde K é o tamanho do vetor inicial. Durante a atualização, ao invés de atualizar o vértice original, é criado um nó e o novo valor será salvo nesse nó. Além disso, um dos ponteiros para os filhos será mantido e o outro será atualizado para o filho que foi gerado e faz parte da recursão da atualização. Assim, cada versão da árvore terá a raiz como representante da versão. Dessa forma, como a árvore possui profundidade $O(log(K))$ então só será necessário criar $O(log(K))$ nós (LAAKSONEN, 2020).

Para ficar mais claro, veja a Figura 6. Nessa figura, temos que a parte com cor vermelha é a árvore antes da atualização e a parte azul representa os nós que foram gerados após a atualização da posição 6 do vetor para o valor 15.

Figura 6: Árvore de Segmento Persistente de soma após update



Fonte: elaborado pelo autor (2021) utilizando o site draw.io.

Com isso, se for necessário fazer alguma consulta na versão antiga da árvore, bastaria começar a consulta a partir da raiz vermelha. Entretanto, se o objetivo for utilizar os dados da árvore de segmento atualizada, bastaria acessar a estrutura a partir da raiz azul. Assim, temos uma estrutura de dados bastante poderosa e versátil.

No Algoritmo 5, mostramos como a Árvore de Segmento Persistente pode ser implementada para operação de soma. Para construir, basta chamar a função $build(v,0,1,k)$, onde 0 é o identificador da raiz original, v é o vetor e k é o tamanho do vetor v. Para fazer uma consulta da soma dos elementos de $v[a..b]$ em uma versão t da árvore, basta chamar a função $query(a,b,t,1,k)$. Por fim, para fazer uma atualização da posição idx para x na versão t da árvore, basta fazer $update(idx,x,t)$. A função de *update* retorna o identificador da nova versão da árvore.

A árvore de segmento persistente é utilizada neste trabalho para representar o estado atual de uma busca. Com a árvore de segmento persistente foi possível reduzir a complexidade de memória e de tempo do A*. Isso é possível, pois para representar cada modificação de um vetor só é necessário utilizar $O(log(K))$ nós adicionais.

Algoritmo 5: Pseudocódigo da Árvore de Segmento Persistente para Soma

```
Function build(v, p, l, r):
    if l == r then
        return seg[p] = v[r]
    end
    cnt++
    L[p] = cnt
    cnt++
    R[p] = cnt
    mid = (l + r) / 2
    seg[p] = build(v, L[p], l, mid) + build(v, R[p], mid + 1, r)
End Function
Function query(a, b, p, l, r):
    if (b < l) or (r < a) then
        return 0
    end
    if (a <= l) and (r <= b) then
        return seg[p]
    end
    m = (l + r) / 2
    return query(a, b, L[p], l, m) + query(a, b, R[p], m + 1, r)
End Function
Function _update(a, x, lp, p, l, r):
    if l == r then
        seg[p] = x
        return seg[p]
    end
    m = (l + r) / 2
    if a <= m then
        cnt++
        L[p] = cnt, R[p] = R[lp]
        seg[p] = _update(a, x, L[lp], L[p], l, m) + seg[R[p]]
        return seg[p]
    end
    else
        cnt++
        L[p] = L[lp], R[p] = cnt
        seg[p] = seg[L[p]] + _update(a, x, R[lp], R[p], m + 1, r)
        return seg[p]
    end
End Function
Function update(a, x, version):
    cnt++
    newRoot = cnt
    _update(a, x, version, newRoot, 1, n)
    return newRoot
End Function
```

Fonte: elaborado pelo autor.

3 TRABALHOS RELACIONADOS

Nesta seção, alguns trabalhos relacionados com este serão apresentados.

3.1 Incremental Hashing in State Space Search

Em Edelkamp e Mehler (2004), os autores introduzem o *hashing* de estado incremental para calcular os valores de *hashing* em tempo constante. Esse método é mais eficaz na busca em grafos de espaço de estados quando utilizamos algoritmos como o IDA*, onde o cálculo do conjunto de sucessores e suas estimativas heurísticas é extremamente rápido. A abordagem ajuda a decidir rapidamente se um determinado estado está presente em uma tabela *hash* e acelera a pesquisa bem-sucedida. Esse trabalho aplica a técnica usada no *15 - Puzzle* e no *Atomix*.

Este trabalho também utiliza *hashing* incremental e aplica as técnicas no *15 - Puzzle*. A principal diferença deste trabalho para dos autores citados é a utilização do algoritmo de A* ao invés do IDA*. Além disso, apresentamos uma técnica mais genérica, pois não é necessário que os estados apareçam de forma sequencial. Isso é possível devido à utilização de uma árvore de segmento persistente. Assim, será possível usar o A* independente do tamanho do vetor que representa o estado atual da busca.

3.2 PNBA*: A Parallel Bidirectional Heuristic Search Algorithm

Rios e Chaimowicz (2011) propuseram o algoritmo nomeado de *Parallel New Bidirectional A* (PNBA*)*. Esse algoritmo combina os benefícios de uma busca bidirecional e de uma execução em paralelo. Com isso, foi desenvolvido um algoritmo eficiente baseado no $A*$. Nos experimentos desenvolvidos em diferentes domínios pelos autores, foi mostrado que o PNBA* é mais eficiente que A* original e o New Bidirectional A* (NBA*).

Tal trabalho também aplica uma solução para o *15-Puzzle*. O principal diferencial deste trabalho em relação ao trabalho de Rios e Chaimowicz (2011) é que apresentamos uma forma de reduzir a memória. Assim, pode-se aplicar o A* para os problemas que a representação do espaço do estado seja larga.

3.3 A* + IDA*: A Simple Hybrid Search Algorithm

Bu e Richard (2019) propuseram um algoritmo de busca hibrido, fazendo uma combinação do algoritmo A* com o IDA*. A ideia da abordagem é, primeiramente, executar o A* até que quase toda a memória do computador seja consumida. Após isso, para cada estado da fronteira é executado o IDA* sem checar estados duplicados. Em seus testes, os autores mostram que A*+IDA* é mais rápido que o IDA* no *24-puzzle* por cerca de 5 vezes. Além disso, é apresentado uma abordagem do A* + IDA* baseada em disco que permite o uso de mais memória e, com isso, apresenta melhorias para instâncias difíceis do *24-puzzle*.

O trabalho dos autores citados apresenta também uma solução para o *24-puzzle*. Diferente deste trabalho, eles não apresentam uma forma de reduzir a memória no A* e não utilizam *hashing* incremental para verificar estados duplicados.

3.4 Análise Comparativa

Na Tabela 1, temos uma análise comparativa entres este trabalho e os trabalhos relacionados. Assim, é possível analisar quais algoritmos e técnicas são utilizadas para solucionar o problema.

Tabela 1: Análise comparativa entre os trabalhos relacionados e este trabalho

Trabalhos	*Puzzle*	IDA*	A*	*Hashing* Incremental	Árvores Persistentes
(EDELKAMP; MEHLER, 2004)	*15-puzzle*	sim	não	sim	não
(RIOS; CHAIMOWICZ, 2011)	*15-puzzle*	não	sim	não	não
(BU; KORF, 2019)	*24-puzzle*	sim	sim	não	não
Este Trabalho	(N^2-1)-*puzzle*	sim	sim	sim	sim

Fonte: elaborado pelo autor.

Como podemos ver na Tabela 1, o Edelkamp e Mehler (2004) e Rios e Chaimowicz (2011) utilizam o $15-puzzle$ para validar as abordagens propostas. Entretanto, em Bu e Richard (2019), os autores realizam os seus testes de desempenho no 24-puzzle. Já este trabalho aborda o problema para várias versões do (N^2-1)-*Puzzle*.

Podemos também, analisar em relação aos algoritmos utilizados. O algoritmo IDA* é abordado nos trabalhos de Edelkamp e Mehler (2004), Bu e Richard (2019) e neste trabalho. Já o A*, é utilizado nos trabalhos Rios e Chaimowicz (2011), Bu e Richard (2019) e neste trabalho. Os únicos trabalhos que usam ambos os algoritmos são Bu e Richard (2019) e este trabalho.

Já em relação ao *hashing* incremental, temos que apenas este trabalho e Edelkamp e Mehler (2004) utilizam essa abordagem. Por fim, vale destacar que este trabalho é o único entre os trabalhos relacionados que visa a redução de memória e, para isso, utiliza árvore de segmento persistente.

4 ALGORITMO PERSISTENT A*

O algoritmo A* possui uma limitação quando aborda problema cuja representação do estado é larga. Isso acontece pois ele possui complexidade de tempo e memória de $O(M \cdot K \cdot log(M))$, onde M é a quantidade de estados visitados durante a busca, K é o tamanho do vetor que representa um estado e o $log(M)$ é o custo de usar a fila de prioridade. Logo, nesse cenário, o A* dificilmente conseguiria resolver um tabuleiro $1000x1000$ do (N^2-1)-*Puzzle*.

Para contornar essa limitação, este trabalho propõe a utilização de uma Árvore de Segmento Persistente com a técnica de *Hashing* Incremental para conseguir reduzir a complexidade de tempo e memória para $O(M \cdot log(K \cdot M))$. Na Tabela 2, podemos ver em que parte do A* cada estratégia vai contribuir na resolução do (N^2-1)-*Puzzle*. Nesta análise, assumimos que o A* é implementado utilizando uma fila de prioridade com operações de custo O($log(M)$).

Tabela 2: Análise de complexidade das variações de implementação do A*

Abordagens	Gerar o sucessor (A)	Achar o próximo para processar (B)	Colocar na fila (C)	Verificar se é final (D)	Remover estados redundates (E)	Complexidade final
A* Clássico (AC)	O(K)	O($K \cdot log(M)$)	O($K \cdot log(M)$)	O(K)	O(K)	O($M \cdot K \cdot log(M)$)
A* com Árvore de Segmento Persistente (AASP)	O($log(K)$)	O($log(M)$)	O($log(M)$)	O(K)	O(K)	O($M \cdot (K + log(M))$)
A* com hashing Incremental (AHI)	O(K)	O($K \cdot log(M)$)	O($K \cdot log(M)$)	O(1)	O(1)	O($M \cdot K \cdot log(M)$)
A* com Árvore de Segmento Persistente e Hashing Incremental (AASPHI)	O($log(K)$)	O($log(M)$)	O($log(M)$)	O(1)	O(1)	O($M \cdot log(K \cdot M)$)

Fonte: elaborado pelo autor.

Nos algoritmos AC e AHI temos o custo de O(K) para gerar o sucessor, pois após trocar as duas posições do vetor, precisamos fazer a cópia dele. Entretanto, em AASP e AASPHI temos que o custo para gerar um sucessor é O(log(K)), pois só precisamos fazer dois *updates* na árvore de segmento persistente e só precisamos guardar a raiz que representa a versão da árvore.

Para achar o próximo estado para ser processado, é necessário acessar o topo da fila

de prioridade e depois remover o mesmo. Com isso, nos algoritmos AC e AHI temos o custo de O($K \cdot log(M)$), pois O(K) é o custo para trocar dois estados de posição e precisamos fazer até log(M) trocas. Já com os algoritmos AASP e AASPHI temos custo O(log(K)), pois para trocar dois estados de posição na fila de prioridade é O(1) já que só precisamos trocar o ponteiro para a raiz da árvore. No total, vai ser necessário fazer até O(log(M)) trocas.

Quando olhamos do ponto de vista de inserir na fila, temos a mesma análise relacionada a encontrar o próximo para processar. Assim, temos que os algoritmos AC e AHI tem o custo de O($K \cdot log(M)$) e os algoritmos AASP e AASPHI tem o custo de O($log(M)$).

Para verificar se um determinado estado é final, precisamos comparar ele como o nosso objetivo. Assim, temos que as abordagem que utilizam o *hashing* incremental conseguem fazer isso em O(1). Em contrapartida, os demais algoritmos precisam comparar todo o vetor, sendo O(K) para fazer essa verificação.

Além disso, temos que verificar se um estado é redundante. Isso pode ser feito de forma mais rápida quando utilizando o hashing incremental. Assim os algoritmos AHI e AASPHI podem fazer isso em O(1), usando a hashing gerada para acessar uma tabela de dispersão. Já no caso dos algoritmos AC e AASP seria necessário percorrer todos o vetor para descobrir essa informação, podemos fazer essa verificação O(K) usando a estrutura de dados Trie (FREDKIN, 1960). Nessa estrutura é possível verificar se um vetor está presente em um conjunto. Isso pode ser feito em complexidade linear em relação ao tamanho do vetor.

Por fim, para calcular a complexidade final de cada algoritmo, podemos perceber que cada estado acessado pode gerar até 4 sucessores, remover ele próprio da fila de prioridade, verificar se é final, se é redundante e colocar até 4 sucessores na fila. Assim, temos que a complexidade é O($M \cdot (4 \cdot A + B + 4 \cdot C + D + E)$).

No Algoritmo 6, temos a implementação do algoritmo proposto. Alguns detalhes de implementação foram omitidos para facilitar a leitura. A seguir será explicado cada parte que o defere do A*.

Algoritmo 6: Pseudocódigo do Persistent A*

```
Function persitentAStar(init):
    pst = PersitentSegmentTree.new(init)
    initHash = getHash(init)
    dist[initHash] = 0
    pq = PriorityQueue.new
    pq.insert(dist[initHash] + getH(init), dist[initHash], getH(init), pst.lastVersion(),
      getPos(init, 0), initHash)
    while !pq.empty() do
        [f0, g0, h0, version0, pos0, hash0] = pq.top()
        pq.pop()
        if hash0 == finalHash then
            return g0
        end
        if g0 > dist[hash0] then
            continue
        end
        for each to in adj[pos0] do
            x = pst.get(to, version0)
            version = pst.update(to, 0, version0)
            version = pst.update(pos0, x, version)
            newHash = getNewHash(hash0, pos0, to, x)
            newH = getNewH(h0, pos0, to, x)
            if g0 + 1 < dist[newHash] then
                dist[newHash] = g0 + 1
                pq.insert(dist[newHash] + newH, dist[newHash], newH, version, to,
                  newHash)
            end
        end
    end
    return -1
End Function
```

Fonte: elaborado pelo autor.

A instrução *PersitentSegmentTree.new(init)* é responsável por construir uma árvore de segmento persistente. Essa construção é feita usando como base o estado inicial do *puzzle*.

A função *getHash(s)* calcula o *hashing* polinomial do estado s. Para implementação foi usado $p = n^2$ e $m = 2^{61} - 1$. Esse módulo foi escolhido porque é um primo grande e pode ser representado por um inteiro de 64 bits. Logo, temos que:

$$getHash(s) = (\sum_{i=0}^{s.size-1} s[i] \cdot p^i) \mod m. \tag{4.1}$$

O objeto *dist* é um dicionário que mapeia um inteiro para outro inteiro. De forma específica para o problema do (N^2-1)-*Puzzle*, o *dist[x]* = y pode ser interpretado da seguinte

forma: y é a menor quantidade de movimento para alcançar o estado que possui hashing x. Quando o estado que possui hashing x sai da fila de prioridade, o valor de y é a resposta ótima para aquele estado. O valor da hashing é utilizado ao invés do próprio estado, pois isso torna o algoritmo mais eficiente.

A instrução *PriorityQueue.new* inicializa uma fila de prioridade de tuplas com 6 posições. Dado a tupla (a_1, a_2, a_3, a_4, a_5, a_6), os elementos são ordenados usando a abordagem lexicográfica. Na posição a_1 é salvo o f que representa a estimativa para resolver o problema. Em a_2 temos o custo g para chegar no estado. No a_3 temos o valor da heurística h para o estado. Na posição a_4 temos o número que indica qual versão da árvore de segmento persistente pode ser usada para recuperar qualquer elemento do estado. Além disso, em a_5 salvamos em qual posição o elemento 0 está. Isso é importante para que não seja necessário percorrer todo o vetor para encontrar essa posição. Por fim, temos que a_6 guarda o valor do hashing do estado.

Ao executar *adj[pos0]* temos como resultado um vetor com todas as posições que são adjacentes a posição *pos0*. Agora, com essa informação é possível simular todas as possibilidades de movimentos.

Para cada movimento possível, temos que explorar todas as opções. Para isso, primeiramente, precisamos recuperar o valor do elemento que está em uma determinada posição. Logo, podemos usar a instrução *pst.get(p, v)* no qual p é a posição que queremos consultar e v é a versão da árvore de segmento persistente.

Após descobrir essa informação, precisamos trocar dois elementos de posição de forma eficiente. Assim, uma forma de fazer isso usando árvore de segmento persistente é executar duas operações de *update*. A função de update tem o seguinte formato: *pst.update(p, x, v)*, onde p é a posição que queremos atualizar, x é o novo valor e v é a versão da árvore de segmento. Além disso, essa função retorna o valor da nova versão da árvore gerada após a atualização.

Seja A o estado atual e B o seu sucessor. Para conseguir manter a complexidade é necessário calcular o valor do *hashing* e da função heurística h do novo estado B de forma eficiente. Para isso, foram implementadas duas funções.

A primeira função é *getNewHash($hash_0$, p_0, t, x)* em que $hash_0$ é o valor da *hashing* de A, p_0 é a posição do 0 em A, t é a posição para onde o 0 foi em B e x é o valor que esta na posição t em A. Assim, podemos facilmente calcular o valor do novo *hashing* usando a ideia de

hashing incremental. Temos que:

$$getNewHash(hash_0, p_0, t, x) = (hash_0 - x \cdot p^t + x \cdot p^{p_0} \; mod \; m). \tag{4.2}$$

onde p e m são os valores adotados no *hashing*.

A segunda função é *getNewH(h_0, p_0, t, x)* em que h_0 é o valor da heurística do A, p_0 é a posição do 0 em A, t é a posição para onde o 0 foi em B e x é o valor que estava na posição t de A. Assim, podemos facilmente calcular o novo valor de h. Temos que:

$$getNewH(h_0, p_0, t, x) = h_0 - D(t, x) + D(p_0, x). \tag{4.3}$$

onde $D(a,b)$ é a distância *Manhattan* que o elemento de valor b está da sua posição final assumindo que atualmente ele está na posição a.

O restante do código possui a mesma estrutura do algoritmo A*.

5 RESULTADOS

5.1 Implementação

Os algoritmos IDA* e A* foram implementados utilizando a linguagem C++. O IDA* foi implementado de forma recursiva e o A* utilizou a *priority_queue* do *Standard Template Library* (STL) do C++.

A função heurística usada nos algoritmos A* e IDA* para resolver o problema *($N^2 - 1$) - Puzzle* foi a distância *Manhattan*. Esta função é muito utilizada para abordar o $15 - Puzzle$.

Para obter o valor da função, primeiramente, é feito o cálculo da distância *Manhattan* de cada elemento no tabuleiro para sua posição final. Após isso, para obter o valor heurístico de uma configuração do tabuleiro, fazemos o somatório desses valores. A posição vazia é desconsiderada nesse cálculo.

Uma função pode ser utilizada no nosso contexto se ela for admissível. Uma forma fácil de ver que essa função é admissível é perceber que ao fazer um movimento, apenas uma peça é movida, e apenas ela pode ficar uma unidade mais perto do objetivo. Com isso, para chegar na configuração final, seria necessário fazer esse movimento para todas as peças. Assim, é preciso pelo menos o valor da função para resolver o problema.

Para avaliar a eficiência da técnica proposta foi feita uma análise comparativa entre o algoritmo proposto e os algoritmos A*, IDA* e IDA* com Memorização. A implementação de todos os algoritmos usam o *hashing* incremental e o cálculo da função *hashing* de forma eficiente. Além disso, todos os algoritmos utilizaram a mesma função heurística. Tais implementações podem ser acessadas no repositório do GitHub do autor. [1]

5.2 Geração dos Casos de Teste

Os testes foram gerados para tabuleiros de tamanho 4x4, 5x5, 10x10, 50x50, 100x100, 500x500 e 1000x1000. Para cada tamanho, foram geradas grupos com 90, 92, 94, 96, 98 e 100 iterações. Além disso, para cada grupo foram gerados 5 casos de testes para diminuir as chances de fatores aleatórios ajudarem algum algoritmo. Com isso, temos $210 = 7 \cdot 6 \cdot 5$ casos de testes.

Para produzir um caso de teste de uma tabuleiro NxN e com K iterações, primeiramente, inicializamos uma matriz NxN na configuração padrão final. Após isso, pegamos a

[1] https://github.com/PauloMiranda98/TccPersistentAStar/tree/master/code

posição do zero e trocamos de forma aleatória com alguma posição vizinha. Repetimos esse último processo K vezes, evitando o movimento que anula o último movimento realizado.

5.3 Execução

Para executar os casos de teste, compilamos todos os algoritmos usando o mesmo compilador *g++ (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0*. Para executar os programas desenvolvidos, limitamos todos os algoritmos para executar com no máximo 16 GB de memória e executando por até 60 segundos. Para fazer isso, utilizamos o programa *ulimit* do *Linux*. O computador usado para executar os testes é um computador com processador *Intel® Core™ i7-10610U CPU @ 1.80GHz × 8*, com 16 GB de RAM e 1 TB de SSD. O sistema operacional é um Ubuntu 20.04.2 LTS sendo configurado com 19 GB de memória SWAP.

5.4 Resultados

Na Tabela 3, temos o resultado obtido por cada algoritmo dependendo do caso de teste. A coluna *Tamanho da Matriz* especifica a versão do $(N^2 - 1)$ - *Puzzle* usada no teste e a coluna *Iterações* representa a quantidade de iterações usada para gerar o caso de teste, conforme especificada na Seção 5.2.

O símbolo *?* significa que o algoritmo não conseguiu encontrar uma resposta nas restrições de tempo e memória determinada. Já no padrão *x # y*, temos que: o *y* é a quantidade de casos dentre os 5 casos que o algoritmo conseguiu executar nos limites de memória e tempo, e o *x* é a média do tempo em segundos que o algoritmo demorou para executar os *y* testes.

Nas versões 4x4, 5x5 e 10x10 temos que o algoritmo *Persistent A** possui um desempenho inferior ao Algoritmo A* em todos os casos. Isso é esperado, pois, para casos pequenos, o uso de uma árvore de segmento persistente vai adicionar uma sobrecarga desnecessária ao algoritmo.

Entretanto, ao olhar as versões 50x50, 100x100 e 500x500 temos que o algoritmo *Persistent A** é superior ao algoritmo A* em todos os casos de teste. A vantagem ao utilizar o algoritmo *Persistent A** fica ainda mais evidente quando analisamos em relação à versão 1000x1000 do $(N^2 - 1)$ - *Puzzle*.

Com isso, vemos que o algoritmo proposto cumpre com o esperado, pois supera a implementação do algoritmo A* clássico quando abordamos matrizes grandes. Além disso,

supera também o IDA* clássico em todas as versões do $(N^2 - 1)$ - *Puzzle*, tornando-se uma opção viável. Apesar de não superar o IDA* com memorização não tínhamos como ambição ser o melhor algoritmo para resolver o problema, mas sim, uma opção alternativa para abordar os problemas de busca.

O gerador de teste, os algoritmos e os *scripts* de execução e compilação podem ser encontrados no repositório do GitHub do autor.[2]

[2] https://github.com/PauloMiranda98/TccPersistentAStar

Tabela 3: Análise comparativa entre os algoritmos

Tamanho da Matriz	Iterações	IDA*	IDA* com Memorização	A*	Persistent A*
4x4	90	0.313 # 1	1.334 # 5	2.486 # 5	3.213 # 5
4x4	92	3.548 # 1	4.474 # 5	6.11 # 5	7.892 # 5
4x4	94	11.758 # 2	0.675 # 5	2.622 # 5	3.331 # 5
4x4	96	7.495 # 4	0.174 # 5	0.399 # 5	0.562 # 5
4x4	98	2.573 # 3	3.265 # 5	0.192 # 4	0.28 # 4
4x4	100	30.393 # 2	2.328 # 5	2.791 # 5	3.598 # 5
5x5	90	3.462 # 1	15.424 # 3	7.677 # 2	9.397 # 2
5x5	92	0.013 # 1	4.637 # 3	17.498 # 3	20.962 # 3
5x5	94	?	1.291 # 2	3.082 # 2	3.856 # 2
5x5	96	?	?	?	?
5x5	98	9.171 # 1	8.453 # 3	16.019 # 3	19.258 # 3
5x5	100	?	16.054 # 1	?	?
10x10	90	12.809 # 4	0.045 # 4	0.567 # 4	0.686 # 4
10x10	92	?	19.443 # 2	2.779 # 1	3.137 # 1
10x10	94	20.394 # 1	19.039 # 2	3.383 # 1	4.17 # 1
10x10	96	?	20.893 # 2	19.743 # 1	21.36 # 1
10x10	98	?	4.658 # 3	26.043 # 3	15.543 # 2
10x10	100	?	2.303 # 1	13.107 # 1	13.779 # 1
50x50	90	2.834 # 1	1.276 # 3	3.652 # 2	12.221 # 3
50x50	92	1.559 # 3	0.022 # 3	0.264 # 3	0.096 # 3
50x50	94	?	1.502 # 3	?	6.37 # 3
50x50	96	36.249 # 1	0.23 # 1	?	5.203 # 1
50x50	98	0.001 # 1	14.877 # 3	0.011 # 1	0.003 # 1
50x50	100	?	6.316 # 1	?	29.748 # 1
100x100	90	0.084 # 2	4.294 # 3	0.955 # 2	0.103 # 2
100x100	92	16.109 # 4	2.711 # 5	0.403 # 1	3.31 # 4
100x100	94	?	6.301 # 1	?	39.561 # 1
100x100	96	0.003 # 1	0.276 # 2	0.433 # 1	3.203 # 2
100x100	98	?	1.341 # 3	?	10.49 # 3
100x100	100	?	3.373 # 2	?	11.627 # 2
500x500	90	31.012 # 2	4.032 # 5	?	1.41 # 4
500x500	92	0.046 # 2	2.462 # 4	0.62 # 1	0.572 # 3
500x500	94	12.886 # 3	0.613 # 4	?	6.4 # 4
500x500	96	?	2.597 # 1	?	24.055 # 1
500x500	98	8.544 # 1	7.645 # 3	?	7.529 # 2
500x500	100	22.171 # 2	0.512 # 3	?	4.125 # 3
1000x1000	90	12.366 # 2	1.881 # 3	?	3.68 # 2
1000x1000	92	13.144 # 2	0.953 # 4	?	9.02 # 4
1000x1000	94	12.843 # 2	10.301 # 5	?	10.308 # 4
1000x1000	96	0.753 # 1	0.179 # 1	?	0.403 # 1
1000x1000	98	8.016 # 4	0.417 # 4	?	2.693 # 4
1000x1000	100	0.149 # 1	0.268 # 2	?	2.151 # 2

Fonte: elaborado pelo autor.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, realizou-se a implementação do A* usando árvore de segmento persistente e hashing incremental. Com ela, foi possível remover uma limitação do algoritmo A*, tornando viável resolver problemas cuja a representação do estado é grande. Assim, temos mais um algoritmo para abordar problemas de busca no espaço de estado.

Apesar de não conseguir superar o algoritmo IDA* com memorização, nos resultados gerais, isso não significa que ele seja um algoritmo inferior. Assim, pode existir outro problema diferente do que foi testado neste trabalho, para o qual o algoritmo *Persistent A** tenha um desempenho melhor.

Para trabalhos futuros, pretende-se implementar uma versão mais eficiente do *Persistent A**. Essa versão irá usar memória estática ao invés de memória dinâmica, evitando assim a sobrecarga da alocação de memória. Outro ponto de melhoria, seria trocar a fila de prioridade por um *bucket list* já que a operação de troca possui custo unitário. Por fim, podemos analisar outras funções heurísticas e ver como os algoritmos vão se comportar.

REFERÊNCIAS

BALL, W. R.; COXETER, H. S. M. **Mathematical recreations & essays**. [*S. l.*]: University of Toronto Press, 2016.

BERG, M. d.; KREVELD, M. v.; OVERMARS, M.; SCHWARZKOPF, O. **Computational geometry**. [*S. l.*]: Springer, 1997. 1–17 p.

BU, Z.; KORF, R. E. A* + ida*: A simple hybrid search algorithm. In: **IJCAI**. [*S. l.*: *s. n.*], 2019. p. 1206–1212.

BU, Z.; KORF, R. E. A* + bfhs: A hybrid heuristic search algorithm. **arXiv preprint arXiv:2103.12701**, 2021.

CHAKRABARTI, P. P.; GHOSE, S.; ACHARYA, A.; SARKAR, S. D. Heuristic search in restricted memory. **Artificial intelligence**, Elsevier, v. 41, n. 2, p. 197–221, 1989.

CULBERSON, J. C.; SCHAEFFER, J. Pattern databases. **Computational Intelligence**, Wiley Online Library, v. 14, n. 3, p. 318–334, 1998.

DRISCOLL, J. R.; SARNAK, N.; SLEATOR, D. D.; TARJAN, R. E. Making data structures persistent. **Journal of computer and system sciences**, Elsevier, v. 38, n. 1, p. 86–124, 1989.

DROGOUL, A.; DUBREUIL, C. A distributed approach to n-puzzle solving. In: **PROCEEDINGS OF THE DISTRIBUTED ARTIFICIAL INTELLIGENCE WORKSHOP**. [*S. l.*: *s. n.*], 1993.

EDELKAMP, S.; MEHLER, T. Incremental hashing in state space search. In: **WORKSHOP NEW RESULTS IN PLANNING, SCHEDULING AND DESIGN**. [*S. l.*: *s. n.*], 2004.

FELNER, A.; ADLER, A. Solving the 24 puzzle with instance dependent pattern databases. In: **INTERNATIONAL SYMPOSIUM ON ABSTRACTION, REFORMULATION, AND APPROXIMATION**. [*S. l.*: *s. n.*], 2005. p. 248–260.

FREDKIN, E. Trie memory. **Communications of the ACM**, ACM New York, NY, USA, v. 3, n. 9, p. 490–499, 1960.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE transactions on Systems Science and Cybernetics**, IEEE, v. 4, n. 2, p. 100–107, 1968.

KARP, R. M.; RABIN, M. O. Efficient randomized pattern-matching algorithms. **IBM journal of research and development**, Ibm, v. 31, n. 2, p. 249–260, 1987.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. **Artificial intelligence**, Elsevier, v. 27, n. 1, p. 97–109, 1985.

KORF, R. E. Linear-space best-first search. **Artificial Intelligence**, Elsevier, v. 62, n. 1, p. 41–78, 1993.

KORF, R. E.; TAYLOR, L. A. Finding optimal solutions to the twenty-four puzzle. In: **PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE**. [*S. l.*: *s. n.*], 1996. p. 1202–1207.

LAAKSONEN, A. **Guide to competitive programming**. [*S. l.*]: Springer, 2020. 274–274 p.

MEHLER, T.; EDELKAMP, S. Dynamic incremental hashing in program model checking. **Electronic Notes in Theoretical Computer Science**, Elsevier, v. 149, n. 2, p. 51–69, 2006.

MEHTA, D. P.; SAHNI, S. **Handbook of data structures and applications**. [*S. l.*]: Chapman and Hall/CRC, 2004.

PREPARATA, F. P.; SHAMOS, M. I. **Computational geometry**: an introduction. [*S. l.*]: Springer Science & Business Media, 2012.

RATNER, D.; WARMUTH, M. K. Finding a shortest solution for the n$\times$ n extension of the 15-puzzle is intractable. In: **AAAI**. [*S. l.*: *s. n.*], 1986. p. 168–172.

REINEFELD, A.; MARSLAND, T. A. Enhanced iterative-deepening search. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, IEEE, v. 16, n. 7, p. 701–710, 1994.

RIOS, L. H. O.; CHAIMOWICZ, L. A survey and classification of a* based best-first heuristic search algorithms. In: **BRAZILIAN SYMPOSIUM ON ARTIFICIAL INTELLIGENCE**. [*S. l.*: *s. n.*], 2010. p. 253–262.

RIOS, L. H. O.; CHAIMOWICZ, L. Pnba*: A parallel bidirectional heuristic search algorithm. In: **ENIA VIII ENCONTRO NACIONAL DE INTELIGÊNCIA ARTIFICIAL**. [*S. l.*: *s. n.*], 2011.

RUSSEL, S.; NORVIG, P. **Inteligência Artificial.** [*S. l.*]: Rio de Janeiro: Elsevier Editora Ltda, 2013.

RUSSELL, S. J. Efficient memory-bounded search methods. In: **ECAI**. [*S. l.*: *s. n.*], 1992. v. 92, p. 1–5.

STEVENS, M. *et al.* **On collisions for MD5**. [*S. l.*]: Citeseer, 2007.