

”

E-fólio A | Folha de resolução para E-fólio

UNIDADE CURRICULAR: Introdução à Inteligência Artificial

CÓDIGO: 21071

DOCENTE: José Coelho

A preencher pelo estudante

NOME: Hugo José Costa Correia

N.º DE ESTUDANTE: 2405276

CURSO: LEI – Licenciatura em Engenharia Informática

DATA DE ENTREGA: 29/04/2026

1. Aspetos Críticos e Análise do Problema

O principal desafio desta missão é a enorme explosão combinatória causada pela dicotomia de movimento: o agente tanto atua como um Rei (navegação no mar) como incorpora Drones (peças com movimentos de xadrez clássico). Numa abordagem rápida, conseguimos perceber desde logo que cada passo do Rei na água corresponde a um nó na árvore de procura, que gera ciclos infinitos e esgota rapidamente a memória (128 MB de limite no VPL). Além disso, problemas com um elevado número de alvos (como as instâncias 9 e 10) onde avaliar todas as permutações de captura (se existirem 20 manchas existem 20! permutações teóricas possíveis) é inviável e impossível em 10 segundos.

O facto de o Rei se poder mover infinitamente na água implica que se o algoritmo principal modelasse cada passo do Rei como um nó na árvore geraria loops infinitos e uma explosão de estados irrelevantes.

Certas zonas do tabuleiro pode estar bloqueadas por obstáculos (peões brancos) tais como sucede nas instâncias 8 e 10, que obrigam ao uso específico do Cavalo, o que obriga o algoritmo a prever as trocas de Drones essenciais para não deixar áreas isoladas.

2. Opções Tomadas na Implementação

Inicialmente comecei por desenvolver o programa em Python, mas rapidamente percebi que dificilmente ia conseguir atingir o nível de performance necessário para resolver o problema. Desta forma acabei por optar por fazer em C++ por ser mais rápido e eficiente.

Analisei e testei vários algoritmos de procura construtiva (incluindo os do TProcura) e heurísticas, entre eles AStar(A*), IDAStar (IDA*), Melhor Primeiro, Profundidade Primeiro, Largura Primeiro, Branch and Bound, e outros sugeridos pela IA generativa.

Os dois algoritmos que melhores resultados tiveram, e onde tentei mais optimizações foram:

A*: O algoritmo mais eficiente para instâncias pequenas/médias. Contudo, nas instâncias finais (9 e 10), a fila de prioridade rapidamente esgotava os 128 MB do VPL (ultrapassando 1.3M de nós) devido aos milhares de empates no custo quando havia demasiadas permutações possíveis.

IDA*: Inicialmente ponderado por usar menos memória e tentar contornar a restrição dos 128MB do VPL, mas descartado, pois a necessidade de recalculiar nós devido ao elevado fator de ramificação excedia o limite de tempo (10s) nas instâncias intermédias.

Foram estudadas, ponderadas e testadas algumas heurísticas como contagem de casas sujas (admissível, mas gera muitos estados com o mesmo valor/empates), Manhattan (não admissível - sobrestima o custo real - assume que não pode andar em diagonal), Euclidiana (admissível, mas subestima o número de passos - cada passo é de custo 1) e distância de Chebyshev pura (admissível, mas ao focar-se apenas no alvo mais próximo subestima o custo total de capturar todas as restantes manchas, gera demasiados empates na fila de prioridade e levava ao esgotamento rápido da memória).

3. Seleção de Algoritmo, Heurística e Configurações

Perante a dificuldade de encontrar um modelo que resolvesse de forma satisfatória todas as instâncias de teste, usei a IA generativa para adotar um modelo combinado/híbrido e definir algumas regras para melhorar a performance do algoritmo.

Assim, para garantir eficácia e eficiência sob os limites de RAM do VPL, a escolha final recaiu num algoritmo Beam Search (Procura em Feixe) que é uma variante restritiva da Procura do Melhor Primeiro (Best First Search), com um tamanho do feixe definido para `BEAM_WIDTH = 20000`.

Ao expandir a árvore o algoritmo avalia milhares de ramificações, mas ordena-as e mantém apenas os 20000 melhores candidatos por nível. O resto é descartado da memória. Sabendo que $m = P$ (no máximo 30), a memória total acumulada nunca excede 600000 nós. Calculando o peso do código na RAM:

A struct State ocupa 48 bytes;

Os arrays `parents_pool` e `next_hash` (`uint32_t`) ocupam 4 bytes;

O array `g_costs` ocupa 2 bytes (`int16_t`).

Portanto temos um total de $48+4+4+2=58$ bytes por nó, que nos 600000 nós perfazem um total de 34.8MB de RAM. A isto somamos o tamanho fixo da tabela de dispersão que tem um tamanho fixo de 2^{22} posições de 4bytes: $4194304 \times 4 = 16.7\text{MB}$.

Portanto o programa consome no máximo um total de $34.8 + 16.7 = 51.5\text{MB}$ de RAM ficando bastante abaixo do limite de 128MB do VPL, contornando as limitações de memória e mantendo uma largura suficiente para garantir soluções ótimas. A reconstrução detalhada do trajeto do Rei é adiada e feita apenas quando a solução final é validada, poupando a propagação de strings pesadas pelos nós.

Para ordenar o feixe, usou-se uma função de avaliação heurística customizada com distância de Chebyshev com peso inflacionado: $\text{Score} = g(n) + (h_{\text{geo}}(n) \times 3)$, onde $g(n)$ é o custo real acumulado (garante que não desperdiça ações no mar) e a componente h_{geo} inflacionada funciona como um radar magnético que atrai os drones para zonas com maior densidade de petróleo, forçando desta forma o algoritmo a preferir caminhos que deixem a estação móvel geograficamente mais perto da próxima captura.

Em vez de gerar sucessores passo a passo, implementou-se uma arquitetura de Macro Capture: Um sucessor é definido estritamente como: "Com o drone D, capturar a mancha na casa C". A árvore de procura não tem conhecimento de água. O custo de transição entre estados é calculado por um micro navegador (uma BFS secundária com operações bitwise e Bitboards de obstáculos dinâmicos) que devolve a distância exata em passos do Rei. Se um drone estiver bloqueado, o ramo é imediatamente cortado.

Em vez de representar cada movimento unitário no tabuleiro, decidi remodelar o espaço de estados: um sucessor na árvore de procura corresponde estritamente à ação combinada de "saltar para um drone e capturar uma mancha

de imediato". A navegação intermédia do Rei na água é tratada como um sub problema, resolvido em segundo plano por uma Micro BFS (procura em largura) dedicada, e só é usada para calcular o custo (g) da aresta que liga a ação. Esta abstração reduz drasticamente a profundidade e a ramificação da árvore, eliminando todos os estados em que o Rei anda sem utilidade pela água. Também para lidar com o limite de memória do VPL o estado foi modelado de forma ultra compacta usando Bitboards (inteiros de 64 bits), onde cada bit representa uma casa do tabuleiro. Um estado guarda apenas a máscara de manchas, as máscaras de cada drone, a posição atual e o modo (peça ativa), garantindo que cada estado ocupa o mínimo de memória possível.

4. Análise da Árvore de Procura: b, d, m

A redefinição do espaço de estados para Macro Capture altera de forma drástica a topologia da árvore. Considerando P o número de manchas iniciais de petróleo e D o número de drones disponíveis:

Profundidade da Solução (d): Como cada transição no nosso espaço de estados resulta na captura obrigatória de exatamente uma mancha, a profundidade da solução é exatamente igual ao número de manchas no tabuleiro. Logo, $d = P$. (Ex: na instância 10, $d = 27$).

Profundidade Máxima (m): Numa modelação clássica passo a passo a árvore seria infinita pois o Rei poderia andar em círculos perdido no oceano para sempre. Nesta modelação Macro a árvore é estritamente finita e igual ao número de manchas, logo $m = P$ e assim garantimos que $d = m$.

Fator de Ramificação (b): O número máximo de sucessores num determinado nó depende de quantas manchas estão na linha de visão (movimentos legais) dos drones. No pior caso, se todos os drones ($D \leq 4$) conseguirem ver e alcançar todas as manchas restantes no nível atual ($P_{\text{restantes}}$), o fator de ramificação máximo é $b_{\text{max}} = D \times P_{\text{restantes}}$. O valor médio desce à medida que aprofundamos na árvore, sendo $b_{\text{min}} = 0$ (beco sem saída).

5 . Estruturas de Dados, Vetores e Matrizes

```
struct State: Representação extra compacta do tabuleiro
    uint64_t bp;           //Bitboard dos peões pretos (manchas de petróleo)
    uint64_t wt, wb, wc, wd; //Bitboards dos drones brancos (T, B, C, D)
    int8_t pos;           //Indice da casa atual onde o Rei está (0 a 63)
    char mode;            //Drone atualmente ativo ('T', 'B', 'C', 'D')
```

```
struct Action: Estrutura auxiliar usada apenas durante a geração de jogadas
```

```
    State s;                //Estado do sucessor
    int cost;               //custo isolado da transição
```

```
struct Candidate: Representa um nó na árvore de pesquisa. Usada para classificar os nós durante o Beam Search.
```

```
    int score;              // Pontuação heurística (menor, melhor)
    int cost_step;          // Custo real acumulado até aqui (g)
    uint32_t parent_id;     // ID do nó pai (para reconstruir o caminho no fim)
    State s;                // O estado do tabuleiro resultante
```

Para evitar a lentidão e fragmentação de memória causadas pela alocação dinâmica tradicional aplicou-se uma arquitetura usando vetores paralelos. Os nós não são objetos no heap ligados por ponteiros mas sim inseridos sequencialmente nestes vetores, e as relações de parentesco são mantidas através de índices garantindo que a RAM cresce de forma contígua, maximizando o uso da memória cache do processador.

```
vector<State> states_pool;
vector<uint32_t> parents_pool; // ID do nó anterior para reconstruir o caminho
```

Tabelas de hash com tratamento de colisões por encadeamento usando índices de arrays em vez de listas ligadas com ponteiros, mantendo a regra de zero alocações dinâmicas durante a pesquisa:

```
vector<uint32_t> next_hash; // Para tratar colisões na tabela de hash
vector<uint32_t> head_hash; // A tabela de hash propriamente dita

vector<int16_t> g_costs; // Custos g(n) correspondentes a cada estado
```

Matrizes de dimensão estática que funcionam como memória muscular do programa

rays_N, knight_m, king_m, etc.: Arrays 2D que guardam os índices das casas legais para onde uma peça se pode mover a partir de qualquer uma das 64 posições do tabuleiro.

`cheb_dist`: Matriz 64x64 pré-calculada no arranque do programa que devolve a distância de entre quaisquer dois pontos em tempo.

6. Funções e Variáveis Principais

`precompute()`: Pré-calcula todas as jogadas legais de xadrez para cada uma das 64 casas do tabuleiro.

`sq_to_str()`: Converte o índice 0-63 na string final exigida (ex: 0 -> "a1", 63 -> "h8")

`bfs_king_dist()`: Calcula a distância exata em movimentos do Rei entre dois drones evitando obstáculos. Retorna -1 se for impossível chegar lá. Utilizado pelo motor para prever o custo de uma mudança.

Variáveis principais:

`visited (uint64_t)`: Registo de casas já processadas. É inicializado com o Bitboard de obstacles, a BFS trata os obstáculos físicos como casas "já visitadas", impedindo que o Rei nade por cima deles.

`q e dist (Arrays Estáticos int[64])`: O `q` funciona como uma fila circular muito rápida (FIFO) que guarda os índices das casas a avaliar e o `dist` guarda a distância da origem até essa casa.

`head e tail (int)`: Os dois ponteiros de gestão da fila estática `q`. O `tail` empurra novos vizinhos para o fim da fila, e o `head` avança para ler e retirar o nó seguinte.

`bfs_king_path()`: Executada apenas na reconstrução final da solução. Refaz o caminho exato do Rei para preencher a string de output.

`push_capture()`: Aplica a macro ação de captura. Move o drone selecionado para a casa alvo, destrói a mancha e adiciona o novo estado à lista de sucessores.

`add_captures()`: Analisa as linhas de visão de um drone e encontra todas as manchas capturáveis.

`get_macro_successors()`: A lógica principal de sucessão. Abstrai a árvore de pesquisa de movimentos de 1 casa. Uma transição de estado só acontece se resultar obrigatoriamente numa captura de mancha.

Variáveis:

`occupied (uint64_t)`: Um Bitboard aglutinador gerado na hora (faz OR entre manchas, drones e obstáculos fixos). Serve como mapa de colisões para perceber se a linha de visão de uma Torre/Bispo/Dama está bloqueada.

`others (uint64_t)`: Um Bitboard que guarda as posições de todos os drones disponíveis exceto aquele onde o Rei está atualmente. É usado como mapa de destinos para simular as trocas de corpo.

`obstacles (uint64_t)`: Um Bitboard de colisões específico para o Rei na água. Ignora os outros drones porque o Rei pode sobrepor-se a eles na água (seja para os controlar seja para os atravessar), mas inclui as manchas e obstáculos fixos (peões).

`solve_instance()`: Implementa a Procura em Feixe que garante não esgotar a RAM, abdicando da matemática do A* clássico para conseguir descartar ramos inúteis.

Variáveis:

`total_targets (int)`: Calcula-se logo no início contando os bits das manchas (`__builtin_popcountll(bp)`). Como cada nível da árvore captura exatamente uma mancha, esta variável dita o número exato de iterações do ciclo for principal (a profundidade limite `d`).

`beam (vector<uint32_t>)`: É o próprio Feixe da pesquisa. Guarda apenas os IDs (índices da Memory Pool) dos até 20000 melhores estados que sobreviveram ao corte no nível de profundidade atual.

`next_cands` (vector<Candidate>): Uma lista temporária que recolhe todos os sucessores gerados a partir do beam atual onde os estados vão ser avaliados e ordenados antes de passarem à fase de corte.

`h_geo` (int): Atua como radar, na avaliação de um candidato guarda a distância de Chebyshev mínima entre o drone ativo e a mancha de petróleo mais próxima, sendo a componente chave da heurística.

`score` (int): O valor final da função de avaliação ($g(n) + h_geo(n) \times 3$) que vai decidir se este candidato fica nos 20000 lugares no feixe.

`macro_path` e `final_path` (vector): Usados exclusivamente no bloco de sucesso final. O `macro_path` faz o backtracking dos IDs dos nós do fim até ao início, enquanto o `final_path` traduz esses IDs nos índices inteiros das casas (0-63), colando as viagens da água pelo meio.

7. Análise de Resultados

No percurso para implementar a solução final foram encontrados vários entraves que exigiram repensar todo o programa/algoritmo por diversas vezes. Foram testadas várias abordagens conforme descrito no capítulo 2, e foi utilizada a IA para perceber alguns comportamentos errados do programa, corrigir bugs e chegar ao algoritmo final. Foi um processo de tentativa e erro perceber o que funcionava e o que não, e sobretudo porquê, tendo sempre em mente a submissão final no VPL e as suas limitações, requerendo vários testes e afinações até chegar à solução (quase) ótima.

A versão final é capaz de resolver as instâncias de teste em mais ou menos tempo mudando por exemplo só o parâmetro `BEAM_WITH`. Depois de vários testes no VPL, concluí que o valor ótimo é 20000, pois foi onde consegui o melhor compromisso de custo/tempo. Colocar valores bastante mais baixos poupa bastante memória, e é bastante mais rápido, mas tem um custo em movimentos extra. Uma vez que o valor 20000 mantém o uso de memória bastante abaixo dos limites do VPL, optei por manter esse valor na versão final. A execução com as 10 instâncias de teste gera a seguinte Tabela de Resultados:

Instancia	I2(Tempo(ms))	Solucao	Observações
1	22	b4 c2 a3 b1 d2 f1 g3 h1 f2 d1 c3 b5 c7	Resolvida! Analisando o mapa do enunciado vemos que todos os movimentos são válidos e todos os peões pretos foram capturados
2	24	b7 c8 g4 b4 c5 e7 a3 a4 a1	Resolvida! Analisando o mapa do enunciado vemos que todos os movimentos são válidos e todos os peões pretos foram capturados
3	62	a1 c1 c7 d7 e8 a8 b7 c7 f7 h7 g8 g2 f3 f1	Resolvida! Analisando o mapa do enunciado vemos que todos os movimentos são válidos e todos os peões pretos foram capturados
4	88	h3 f3 f6 f7 g8 e6 g4 c4 b5 d6 b7 a5 b3 a1	Resolvida! Analisando o mapa do enunciado vemos que todos os movimentos são válidos e todos os peões pretos foram capturados
5	2304	h7 g6 h5 h4 h2 f4 f3 e2 e6 c4 b5 b6 c5 f8 c8 c6 a4 d7 d4 b2 b1	Resolvida! Analisando o mapa do enunciado vemos que todos os movimentos são válidos e todos os peões pretos foram capturados
6	1392	d3 c4 b5 a4 b4 a3 a5 b6 c5 d4 e5 c7 d8 c8 b7 b8 h2 e2 d1 f3 h3 c3 c6 e6 f7	Resolvida! Analisando o mapa do enunciado vemos que todos os movimentos são válidos e todos os peões pretos foram capturados
7	2375	e3 d3 f5 g4 g3 g2 g6 f7 f8 e8 d8 c7 b8 a8 b7 b5 c6 a4 b3 b1 b4 c3 d2 e1 c1 f4 h2 e5 d6	Resolvida! Analisando o mapa do enunciado vemos que todos os movimentos são válidos e todos os peões pretos foram capturados
8	20		Impossível! Analisando o mapa do enunciado, vemos que existem peões pretos barricados por peões brancos no canto superior direito. Uma vez que não existem Cavalos é impossível o agente entrar (se iniciar fora) ou sair (se iniciar dentro) desta barreira. Repare-se que esta instância termina com um tempo inferior aos 10000ms, o que indica que o algoritmo desistiu não por ter excedido o tempo limite mas por ter concluído que não existem mais movimentos possíveis
9	1627	h7 h6 h8 g7 e5 h2 f2 f3 e2 e3 d4 c5 a5 a4 a6 b6 c6 d6 a3 a2 a1 b1 c1 d1 d2 d3 c4 c7 d8 e8 c8 a8	Resolvida! Analisando o mapa do enunciado vemos que todos os movimentos são válidos e todos os peões pretos foram capturados
10	2085	h3 g5 e4 g3 f1 e3 g2 h4 f3 d4 c2 b4 d3 e5 c4 d6 c7 b7 a7 d7 d6 e8 g7 e6 f4 d5 e7 d7 d2 c2 b2 b1 a1 c1 d2 d8 c7 b8 a8 b7 c7 d6 e7 g8 h8 h7	Resolvida! Apesar de existir uma barreira similar à instância 8 no canto superior direito, uma vez que existem Cavalos disponíveis, a solução já é possível. Repare-se na 23ª ação onde o agente salta de e8>g7 por cima da barreira, e mais tarde na 44ª onde o Cavalo volta a saltar de e7>g8 para limpar o resto do canto.

Destes resultados verifica-se que a abordagem adotada com Beam Search aliada ao Macro Capture atingiu o objetivo proposto de forma bastante eficiente. Todas as instâncias foram resolvidas muito abaixo da barreira dos 10000ms, com consumos de memória marginais, provando que abandonar a abordagem clássica do A* (e a sua garantia teórica de otimalidade absoluta) em prol de um Beam Search aliado a uma heurística de Chebyshev com peso inflacionado é a estratégia correta em domínios sujeitos a explosões combinatórias puras e limites de hardware severos.

─	🔍	Soluções:10	📄	Instâncias: 10.
─	📄	:1	🔍	🔥 13 ⌚ 39
─	📄	:2	🔍	🔥 9 ⌚ 46
─	📄	:3	🔍	🔥 14 ⌚ 76
─	📄	:4	🔍	🔥 14 ⌚ 109
─	📄	:5	🔍	🔥 21 ⌚ 1614
─	📄	:6	🔍	🔥 25 ⌚ 1294
─	📄	:7	🔍	🔥 29 ⌚ 3038
─	📄	:8	🔍	🔥 🚫 ⌚ 40
─	📄	:9	🔍	🔥 32 ⌚ 2496
─	📄	:10	🔍	🔥 46 ⌚ 2161
─	🔍	Válidas:10	📄	Instâncias: 10.
─	🔥	Melhor:203	🔥	Pior: 203.
─	⌚	Tempo(ms):10913.		
─	📊	99.7%	(🔍	100.0% 🔥 100.0% ⌚ 99.0%)

Resultado do evaluate no VPL