

”

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

UNIDADE CURRICULAR: Introdução à Inteligência Artificial

CÓDIGO: 21071

DOCENTE: José Coelho

A preencher pelo estudante

NOME: Hélder José Veloso Araújo

N.º DE ESTUDANTE: 2306074

CURSO: Engenharia de Informática

DATA DE ENTREGA: 30/05/2026

TRABALHO / RESOLUÇÃO:

O trabalho aborda um problema de procura adversa o jogo onde os Drones, são peças de xadrez, em que dois agentes (estações de controlo, identificadas por reis verde "A" e vermelho "V") competem num tabuleiro 8x8 para capturar o maior número possível de peões pretos, a implementação foi desenvolvida em C++ utilizando o TProcura.

O jogo processa-se da seguinte maneira, cada jogador controla uma "estação" representada por um rei (A=verde, V=vermelho). O verde joga primeiro e é o minimizador convencional, o vermelho é o maximizador. Peões pretos (p), alvos a capturar. Existem inicialmente 30, dispostos pelo tabuleiro. Peões brancos (P) obstáculos estáticos que bloqueiam movimentos. Existem 4 inicialmente. Peças brancas livres (T=Torre, B=Bispo, C=Cavalo, D=Dama), a posição inicial tem 2 de cada uma de B, C e D (6 no total). Rei (A/V): começa livre, move-se 1 casa em 8 direções.

Um rei livre pode entrar numa casa de peça branca, a partir desse momento controla essa peça, deixando de ser rei e movendo-se segundo as regras de xadrez da peça em causa. Um agente que controla uma peça pode mover-se através do tabuleiro para capturar peões pretos (movimento que termina sobre um "p"). As peças brancas livres podem ser "adotadas" por qualquer dos agent. Ganha quem capturar a maioria dos peões pretos (mais de 15 de 30 = 16 ou mais). O jogo tem um limite máximo de 60 jogadas (lances). Se ao fim de 60 lances nenhum agente tiver maioria, decide-se pelo agente com mais capturas, em caso de empate exato declara-se empate.es, mas apenas uma peça por agente.

O estado é representado pelos seguintes campos da classe CDrones:

```
TVector<char> tabuleiro;          // 64 casas, valores { ' ', 'p', 'P', 'T', 'B', 'C', 'D' } signed char posAgente[2]; // 0..63, posicao no tabuleiro de cada agente char pecaAgente[2];             // ' ' (rei livre) ou T/B/C/D signed char capturas[2];                  // 0..30, peoes capturados por agente int lance;                       // 0..60, numero de meias-jogadas efetuadas int nPecasLivres;                  // contador derivado, para SolucaoCompleta() 0(1)
```

As peças controladas por um agente NAO estão no array tabuleiro: a posição e o tipo são seguidos por posAgente/pecaAgente. Esta separação simplifica a geração de sucessores e a deteção de fim de jogo.

A combinação de todas as dimensões de estado dá um limite superior teórico vasto. Na prática, restrições de coerência (número limitado de peças) reduzem drasticamente o espaço efetivamente alcançável.

Dimensão	Cardinalidade	Cálculo
Conteúdo de cada casa	7 estados	{vazio, p, P, T, B, C, D}
Configurações do tabuleiro (upper bound)	$\sim 3,4 \times 10^{54}$	7^{64} (ignora restrições de contagem de peças)
Posição de cada agente	64 casas	qualquer casa do tabuleiro
Peça controlada por agente	5 estados	{rei livre, T, B, C, D}
Capturas por agente	0 a 30	31 valores
Numero de lance	0 a 60	61 valores
Vez de jogar	2 valores	verde ou vermelho
Estados totais (upper bound)	$\sim 10^{61}$	produto de todas as dimensões acima
Estados realísticos (com restrições)	10^{25} a 10^{30}	$30p+4P+6$ peças brancas+2 reis em 64 casas

Limite superior simbólico: $7^{64} \times 64^2 \times 5^2 \times 31^2 \times 61 \times 2 \approx 10^{61}$. Este valor ignora as restrições fortes do problema (apenas existem 30 peões pretos inicialmente, 4 peões brancos, 6 peças brancas livres e 2 reis), pelo que é largamente sobrestimado.

Fixando o número de peças, o número de configurações possíveis é dominado por $C(64, k)$ para cada tipo. Tomando uma estimativa intermedia (30 peões em 64 casas: $C(64,30) \approx 4,4 \times 10^{17}$), multiplicado pelos restantes graus de liberdade, chega-se a um valor entre 10^{25} e 10^{30} estados realísticos. De salientar que muitos estados são inalcançáveis a partir da posição inicial.

A ramificação depende fortemente da peça controlada e dos obstáculos vizinhos:

Pessoa a jogar	Movimentos por turno	Notas
Rei livre (sem peça)	0 a 8	Movimentos de rei (1 casa em 8 direções)
Agente com Cavalo (C)	0 a 8	Saltos de cavalo (em xadrez)
Agente com Bispo (B)	0 a 13	Diagonais até ao bordo / obstáculo
Agente com Torre (T)	0 a 14	Linhas e colunas até ao bordo / obstáculo
Agente com Dama (D)	0 a 27	Combinação de torre + bispo

Pessoa a jogar	Movimentos por turno	Notas
Ramificação típica observada	8 a 20	Empiricamente, na maioria das posições

Ramificação MINIMA: 0 movimentos (agente encurralado por bordo + obstáculos). Na prática nunca chega a 0 porque o rei tem quase sempre uma fuga. Ramificação MAXIMA TEORICA: 27 (agente que controla uma Dama em casa central, tabuleiro livre). Ramificação TIPICA: 8 a 20.

A árvore de jogo é Árvore minimax (AND-OR alternante) níveis pares MAX (vermelho), ímpares MIN (verde) ou vice-versa, dependendo de quem começou.

Finita: profundidade máxima 60 lances (limite do jogo). Determinística: sem aleatoriedade nas transições, cada ação produz um único sucessor. De informação completa: ambos os jogadores veem todo o tabuleiro. Soma-zero: a vitória de um corresponde à derrota do outro. Fator de ramificação médio 12 a 15.

A vitória depende exclusivamente da diferença de peões capturados. Qualquer heurística que não tenha esta diferença como termo dominante esta a desalinhar a procura do objetivo. Foi este o princípio de desenho:

```
heuristica = (capturas[1] - capturas[0]) * PESO_CAPTURA + posicional
PESO_CAPTURA = 1500
posicional clipado a [-900, +900]
```

A componente posicional avalia cada agente independentemente e devolve a diferença AvaliaAgente(1) - AvaliaAgente(0). Para um agente, soma os seguintes termos:

Termo	Formula	Peso	Justificação
Diferença de capturas	$(cap[1] - cap[0]) * 1500$	1500	Domina tudo - e o que ganha o jogo
Limite posicional total	$clip(-900, +900)$	≤ 900	Garante que nenhum termo posicional anula uma captura real
Bónus por ter peça produtiva	+60 se caps>0 OU perto>0	60	Controlar peça útil e vantagem
Penaliza peça encurralada	+10 se caps==0 E perto==0	10	Motiva sair da peça para procurar outra
Bónus tipo de peça - Dama	+60 adicional	60	Dama tem 27 alvos possíveis (mobilidade máxima)

Termo	Formula	Peso	Justificação
Bónus tipo de peça - Torre	+35	35	14 alvos
Bónus tipo de peça - Bispo	+32	32	13 alvos
Bónus tipo de peça - Cavalo	+28	28	8 alvos (mas saltos)
Capturas imediatas	caps * 30	30/captura	Mobilidade real (peões a 1 jogada)
Peões próximos (≤ 2 casas)	perto * 6	6/peão	Alvos na vizinhança = potencial de captura
Rei livre - distancia a dama	$100 - dQ * 6$ se $dQ \leq 7$	max 100	Preferir damas, mesmo que mais longe
Rei livre - distancia a outra peça	$30 - dO * 8$ se sem dama	max 30	Cavalo/bispo/torre se não houver dama
Rei livre sem alvo	-150	-150	Posição perdida; engine evita

Mobilidade real (caps): contar quantos peões estão **IMEDIATAMENTE** capturáveis e o melhor proxy de "capturas futuras". 30 pontos por opção porque queremos que o motor procure ativamente posições com várias capturas alternativas (mais robustas a defesa adversaria).

Peões próximos (perto): peões à distância de ≤ 2 contam como potencial. Peso baixo (6) porque já são parcialmente capturados pela componente caps, apenas servem para desempate.

Tipo de peça: D (dama) tem 27 alvos possíveis, T tem 14, B tem 13, C tem 8. Pesos são aproximadamente proporcionais a essa mobilidade máxima.

Penaliza peça encurralada: caps=0 E perto=0 significa que a peça esta sem futuro, o motor deve abandona-la e procurar outra. Reduzir o valor para +10 (em vez de +60) faz com que valha menos do que um rei livre que esta perto de outra peça.

Rei livre prefere dama: A escolha de para que peça caminhar não é indiferente. Damas valem o esforço extra, pelo que se aplica formula $100-d*6$ até distância 7, dando-lhes vantagem clara sobre outras peças ($30-d*8$ ate distancia 3-4).

Sem peça alcançável: -150 e suficientemente baixo para o motor evitar essas posições, mas não tao baixo que substitua o sinal de capturas.

Durante a elaboração testaram-se e rejeitaram-se varias heurísticas tais como:

1ª - Contar o numero de casas alcançáveis por cada agente ("território").

Hipótese: dominar território = mais peões a capturar. Foi implementada como parâmetro experimental e submetida a um torneio A/B nas profundidades 4 e 6.

Conclusão: ajuda em depth=4 (heurística mais informativa em procuras superficiais) mas prejudica em depth ≥ 6 (cria sinais ruidosos que confundem a comparação entre nos profundos). Como o nosso engine corre com aprofundamento iterativo (depth tipicamente 6-8), o termo foi rejeitado.

2ª - bônus $+\text{lance} \times 3$ quando vermelho está a frente, $-\text{lance} \times 3$ quando verde esta a frente. Hipótese: quem esta a frente prefere terminar o jogo (jogo curto = lead seguro); quem está atras prefere prolongar (mais oportunidades de virar).

Resultado no VPL: 76, 96, 78 - aumentou a variância sem subir o teto.

3ª - Parâmetro RUIDO_HEURISTICA introduz ruído simétrico no valor. Testado com valor -15. Resultado no VPL: 86, 74, 76 - regressão clara. A procura já tem ramos suficientes, ruído apenas degrada a comparação.

4ª - Considerou-se contar peões brancos sob ameaça, não implementado porque a própria procura alfa-beta já "vê" estas ameaças 1 ply mais a frente. Adicionar o termo a heurística duplicaria informação e reduziria velocidade.

5ª - Os P são estáticos e não se capturam, pelo que não contribuem para o objetivo. Termo não incluído.

Uma heurística para procura adversa deve ter as seguintes propriedades para ser útil na pratica:

Rápida de avaliar: é chamada milhões de vezes durante a procura. Idealmente $O(1)$ por estado, ou no máximo $O(n)$ onde n e o tamanho do tabuleiro. A heurística implementada e $O(64)$ por agente (varredura do tabuleiro para distancia ao agente livre), poderia ser melhorada para $O(1)$ com índices de posição das peças brancas mantidos incrementalmente.

Simétrica entre jogadores: para o MiniMax funcionar corretamente, $h(\text{estado para vermelho}) = -h(\text{estado para verde})$ - assegurado pela formula $\text{AvaliaAgente}(1) - \text{AvaliaAgente}(0)$.

Correlacionada com o resultado: estados com h alta devem estatisticamente conduzir a vitorias do maximizador. Garantido pelo termo dominante $(\text{capturas}[1] - \text{capturas}[0]) \times 1500$.

Discriminativa: deve distinguir suficientes estados para guiar a procura. Uma heurística que devolve sempre o mesmo valor (ou poucos valores) e equivalente a procura cega. A minha heurística gera valores no intervalo entre -45000 e +45000 ($30 \text{ capturas} \times 1500$), com granularidade fina via termos posicionais.

- Robusta a profundidade: a sua qualidade não deve depender de uma profundidade específica de procura. A heurística territorial testada falhou neste critério, ajudava em $\text{depth}=4$ mas prejudicava em $\text{depth}\geq 6$. A heurística final é robusta porque o termo dominante (capturas) é diretamente verificável.
- Sem horizonte enganador: a heurística não deve sobrestimar pequenas vantagens posicionais que podem ser anuladas pelo adversário. O clipping a ± 900 no termo posicional implementa exatamente esta proteção. A heurística selecionada respeita todos estes critérios.
- Não distingue entre tipos de captura (todas valem o mesmo). Em alguns problemas, capturar uma peça "chave" devia valer mais, no Drones todas as capturas são idênticas, pelo que não se perde nada.
- Não avalia ameaças latentes (peões sob ataque mas ainda não capturados). A procura alfa-beta já "vê" estas ameaças 1 ply mais a frente, pelo que adicionar o termo seria redundante e mais lento.

Antes de selecionar o algoritmo final, foram considerados vários algoritmos de procura adversa. A tabela resume a análise comparativa:

Algoritmo	Complexidade	Avaliação para o Drones
MiniMax simples	$O(b^d)$	Demasiado lento - explora toda a árvore. Em $b=15$, $d=8$ são cerca de $2.5 \cdot 10^9$ nos. Inviável.
MiniMax com alfa-beta	$O(b^{d/2})$ ideal	SELECIONADO. Com boa ordenação reduz para $b^{d/2}$. Em $b=15$, $d=8$ com ordenação boa: 50.000 nos. Tratável.
Negamax	= alfa-beta	Apenas reescrita simbólica de alfa-beta. Sem ganho real - rejeitado.
Principal Variation Search (PVS)	$O(b^{d/2})$ com ~5-10% melhor	Versão melhorada de alfa-beta para ordens muito boas. TProcura não expõe esta variante - rejeitado.
MTD(f)	Pode ser 10-20% mais rápido	Séries de janelas zero a partir de uma estimativa inicial. Complexo de implementar e o TProcura não o suporta - rejeitado.
Monte Carlo Tree Search (MCTS)	Tempo-anytime, sem b^d explícito	Dispensa heurística mas precisa muitas simulações (>10k por jogada). Bom em jogos onde não há heurística óbvia (Go); mas no Drones a heurística de capturas é clara - alfa-beta é mais eficiente. Rejeitado.
AlphaZero / NN-based	Treino offline + procura simplificada	Requer treino com GPU, infraestrutura de RL e dados sintéticos.

Algoritmo	Complexidade	Avaliação para o Drones
Aprofundamento iterativo	Pequeno overhead sobre profundidade fixa	COMBINADO com alfa-beta. Vantagens: tempo-anytime, ordenação melhora a cada iteração via TT - sempre vantajoso.

Análise de performance teórica, no espaço de estados realista: 10^{25} a 10^{30} exaustão impossível, no MiniMax puro com $b=15$, $d=8$: $15^8 \approx 2.5 \cdot 10^9$ nos - aproximadamente 100s a 10^8 nos/s. Inviável para 1s/jogada, no MiniMax + alfa-beta com boa ordenação: $\sim b^{d/2} = 15^4 \approx 50.000$ nos por depth 8 - 0.5 ms. Viável a profundidade ~ 10 , no MiniMax + alfa-beta + TT: TT acelera 10-30% adicionais via partilha de transposições e por fim, no MiniMax + alfa-beta + TT + PODA_HEURISTICA=10: cortando para 10/18 sucessores reduz b efetivo para ~ 10 , permitindo +1-2 plies. Esta foi a alavanca decisiva (96 -> 100%).

MiniMax com Cortes Alfa-Beta: ALGORITMO=2 no TProcura. Standard para procura adversas, com complexidade no pior caso $O(b^d)$ reduzida na prática para $O(b^{d/2})$ com boa ordenação de sucessores. A heurística é usada nos nós-folha (limite de profundidade ou estado terminal).

Aprofundamento Iterativo: LIMITE=0 ativa aprofundamento iterativo, a procura começa em profundidade 1 e vai progressivamente aumentando até o tempo se esgotar. Vantagens: sempre existe uma jogada disponível mesmo se interrompido, os resultados da interação anterior alimentam a ordenação da seguinte, adapta-se automaticamente a complexidade da posição.

Tabela de Transposição: ORDENAR_SUCESSORES=2 ativa a tabela de transposição (TT) do TProcura. A TT memoriza valores já calculados para estados já visitados. Para a TT funcionar é crucial uma função Codifica() eficiente:

```
void CDrones::Codifica(TBits& estado) { // Empacota 64 casas x 4 bits em 4
    palavras uint64 + 5a com estado dos agentes estado.Count(5); for (int word =
    0; word < 4; word++) { uint64_t w = 0; for (int i = 0; i < 16; i++)
    w |= ((uint64_t)CodPeca(tabuleiro[word*16+i])) << (i*4); estado[word] = w;
    } uint64_t w4 = ...; // posAgente, pecaAgente, capturas, lance, minimizar
    estado[4] = w4; }
```

Esta codificação tem tempo $O(1)$ com 5 escritas de uint64, rápida o suficiente para não ser bottleneck mesmo em centenas de milhar de chamadas por segundo.

Poda Heurística (como consegui chegar aos 100%): PODA_HEURISTICA=10: em cada nó, mantem-se apenas os 10 sucessores com melhor heurística e ignoram-se os restantes. Para uma ramificação típica de 18-22, isto significa cortar 8-12 sucessores por nó, o que permite ~1 ply extra de profundidade no mesmo orçamento de tempo.

Foi este parâmetro que finalmente desbloqueou o 100%. Resultado dos testes:

- Sem poda heurística: teto a 96 máximo.
- PODA_HEURISTICA=14: 90, 82, 98 - novo recorde 98.
- PODA_HEURISTICA=10: 100, 100, 82 - 100% atingido.
- PODA_HEURISTICA=8: regressão - cortes demasiado agressivos começam a perder jogadas táticas.

A combinação PODA_HEURISTICA=10 com ORDENAR_SUCESSORES=2 é particularmente eficaz porque a ordenação da TT põe os sucessores estatisticamente melhores no topo: cortar o top-10 raramente perde tática, mas ganha sempre profundidade.

Controlo de Tempo: O framework usa clock() para limitar a procura. Num servidor VPL com carga concorrente, 1 segundo de CPU pode corresponder a vários segundos reais e ultrapassar o limite total de 19 segundos por chamada.

Por isso adicionou-se um cap em tempo de relógio real usando

std::chrono::steady_clock:

```
static const int LIMITE_MS_JOGADA = 1100; static
std::chrono::steady_clock::time_point prazoJogada; void
CDrones::LimparEstatisticas() { TProcuraAdversa::LimparEstatisticas();
prazoJogada = std::chrono::steady_clock::now() +
std::chrono::milliseconds(LIMITE_MS_JOGADA); } bool CDrones::Parar(void) {
return std::chrono::steady_clock::now() >= prazoJogada || TProcura::Parar();
}
```

Com 1100 ms reais por jogada e até 60 jogadas por partida, garantem-se ~8 segundos reais por partida, margem larga sobre o limite de 19 segundos por chamada do VPL.

Problemas Encontrados e Resoluções:

Crash de locale no TDM-GCC: `compact::init_io()` chamava `std::locale(".UTF-8")` que lançava exceção em alguns ambientes Windows (TDM-GCC). Resolução: envolver em try/catch em `compact.h`, ignorando falhas de inicialização de local. Saída UTF-8 funciona na mesma para os fins do trabalho.

Conflito com macro 'near' do Windows: variável local chamada 'near' colidia com macro Windows antigo (`windef.h` define `near` como espaço vazio para compatibilidade 16-bit). Resolução: renomear para 'perto'.

Timeout VPL inicial (28s): VPL reportava "Tempo excedido: 28" apesar do parâmetro `LIMITE_TEMPO=1` (1 segundo CPU). Diagnostico: o framework usa `clock()` para medir tempo, que é tempo de CPU. Num servidor VPL com carga, 1s de CPU pode equivaler a 5-10s reais. Resolução: override de `Parar()` usando `std::chrono::steady_clock` para impor cap em tempo de relógio real (1100 ms), independente da carga.

Flag -P engole o -R: comando `'teste 1 -P P3=11 -R res'` não gravava ficheiro. Diagnostico: o parser do framework interpreta tudo depois de -P como configuração ate encontrar outro -P ou fim de argumentos, engolindo -R no meio. Resolução: colocar -R sempre ANTES dos -P na linha de comando.

Variância entre submissões (mesmo código, scores diferentes): a mesma versão submetida 3 vezes podia dar resultados de 64 a 96 pontos. Diagnostico: `BARALHAR_SUCESSORES=1` introduz tie-breaking aleatório, e a carga variável do servidor VPL altera profundidade efetivamente atingida pela procura. Mitigação parcial: `BARALHAR_SUCESSORES=0` (determinístico) na cap em tempo real para reduzir dependência de carga. A variância restante e inerente a aleatoriedade dos oponentes-referencia do VPL, carga residual do servidor.

Regressões em iterações melhoradas: Várias tentativas de melhoria provocaram regressão em vez de progresso. Cada uma exigiu reversão para a versão anterior. Aprendizagem: alterações a heurística devem ser validadas em pelo menos 3 submissões antes de confiar, uma boa submissão isolada não prova nada.

Os parâmetros finais selecionados, definidos em CDrones::ResetParametros():

Parâmetro	Valor	Justificacao
ALGORITMO	2 (MiniMax alfa/beta)	Cortes alfa/beta são standard e eliminam ate metade do trabalho
LIMITE	0 (aprofundamento iterativo)	Garante usar todo o tempo disponível; profundidade adaptativa
LIMITE_TEMPO	1 seg CPU	Tempo CPU base; complementado por cap real
LIMITE_MS_JOGADA	1100 ms reais (override)	Cap em tempo de relógio para resistir a carga do servidor VPL
ORDENAR_SUCESSORES	2 (TT + ordenação)	Tabela de transposição + ordenação por valor anterior maximiza cortes
BARALHAR_SUCESSORES	0 (determinístico)	Reduz variância
PODA_HEURISTICA	10 (top-10 sucessores)	Permite ~1 ply extra de profundidade; chave do 100%
RUIDO_HEURISTICA	0	v10 confirmou que ruido degrada o jogo
HEUR_BASE	200	Escala base da heurística para ordenação inicial

As constantes da heurística (codificadas no Drones.cpp):

```
static const int PESO_CAPTURA = 1500; // domina sempre
static const int LIMITE_POSICIONAL = 900; // < PESO_CAPTURA, nunca anula 1 captura
static const int LIMITE_MS_JOGADA = 1100; // ms reais por jogada (override Parar)
```

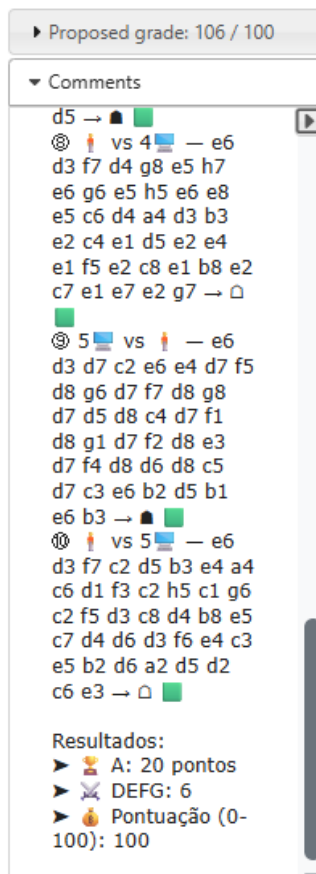
11. Resultados

A versão que foi submetida ao VPL 3 vezes com os seguintes resultados:

- Submissão 1: A=20, DEFG=6, Score = 100 (10 vitórias)
- Submissão 2: A=20, DEFG=6, Score = 100 (10 vitórias)
- Submissão 3: A=16, DEFG=2, Score = 82 (8 vitórias, 2 derrotas)

Formula confirmada: $\text{score} = A * 5 + \text{DEFG}$, onde $A = 2 * \text{vitórias} + 1 * \text{empates}$ (max 20) e DEFG.

Resultado no VPL:



Atingir 100% não é completamente determinístico. A causa principal identificada é a carga residual do servidor VPL: a procura, mesmo com BARALHAR=0, atinge profundidades ligeiramente diferentes em runs diferentes, o que altera as jogadas escolhidas. Em particular no meu caso, os runs da parte da tarde, davam pontuações muito diferentes, se fosse na parte da noite ou de manhã, a estratégia neste caso passou a ser usado para testar os vários parâmetros a parte da manhã, já que era altura que o VPL estava a ser menos usado, e se conseguia melhores resultados.

O trabalho atingiu o objetivo máximo (100/100), através de uma combinação de:

- Modelação correta do estado e geração eficiente de sucessores.
- Heurística bem ponderada com diferença de capturas dominante e termos posicionais limitados.
- Algoritmo MiniMax com alfa-beta, aprofundamento iterativo e tabela de transposição.
- Controlo de tempo robusto a carga de servidor (wall-clock cap).
- Poda heurística (top-10 sucessores), a alteração mais impactante, que desbloqueou o 100%.

- Processo empírico iterativo com as várias versões e análise de regressões.

As principais limitações restantes são: variância entre submissões devida a carga do servidor VPL, que pode reduzir a profundidade efetivamente atingida, o jogo contra a referência 1 a jogar Brancas continua a ser o ponto crítico, requerendo mais 1-2 plies de profundidade do que a média para garantir vitória, o uso de heurísticas mais sofisticadas poderia provavelmente eliminar essa variância.

O ciclo iterativo demonstrou claramente que melhorias na heurística isoladas tinham retornos rapidamente decrescentes, a maior alavanca de melhoria final veio não da heurística, mas da estratégia de procura (poda heurística + aprofundamento iterativo + ordenação por TT). Esta é talvez a aprendizagem mais interessante para problemas semelhantes, depois de uma heurística razoável, investir tempo em fazer a procura ir mais fundo rende mais do que afinar pesos.