Departamento de Engenharia de Eletrónica e Telecomunicações e de Computadores
Licenciatura em Engenharia Informática e de Computadores
Licenciatura em Matemática Aplicada à Tecnologia e à Empresa
Licenciatura em Engenharia Informática, Redes e Telecomunicações

The Kurtan game

Second Practical Project

Artificial Intelligence Course

Summer Semester 2024/2025

14 Junho 2025

Docente: Nuno Leite
Daiana Lupaiescu, A48668
João Lopes, A50457

Índice:

Introdução

O presente relatório descreve o desenvolvimento de um jogo Kurtan, implementado em Prolog, uma variação do clássico jogo Sokoban. O projeto foi realizado no âmbito da disciplina de Inteligência Artificial, com o objetivo de aplicar e comparar diferentes algoritmos de procura e otimização na resolução de níveis do jogo.

O jogo consiste na movimentação de um personagem que deve empurrar caixas para posições finais específicas. A complexidade do jogo é aumentada pela presença de um gate que exige uma chave, obtida após posicionar um número determinado de caixas corretamente. O objetivo final é alcançar a posição marcada com um ponto de interrogação, após resolver todos os desafios do nível.

Objetivos do projeto

Os principais objetivos do projeto foram:

- Implementar o jogo kurtan.
- Implementar algoritmos para resolver automaticamente o jogo Kurtan em Prolog (iterativeDeepening e A*) e em Python(Simulated Annealing e Genetic Algorithm).

- Avaliar o desempenho de diferentes algoritmos:
    - Procura em profundidade iterativa (iterative-deepening) – não informada, em Prolog.
    - Procura A* – informada, com heurística admissível, em Prolog.
    - Simulated Annealing – algoritmo de otimização, em python.
    - Algoritmos genéticos – algoritmo de otimização, em python.

Descrição do Jogo Kurtan

O Kurtan é um jogo de lógica e estratégia baseado no clássico Sokoban, no qual o jogador deve mover um personagem num labirinto para empurrar caixas de madeira até as colocar em posições finais específicas, assinaladas com o símbolo (*).

Regras principais do jogo:

- Movimentos: O jogador pode mover-se nas direções cima (u), baixo (d), direita (r) e esquerda (l), desde que a célula de destino esteja livre ou permita a movimentação.

- Empurrar caixas: Apenas caixas (@) podem ser empurradas, e apenas uma de cada vez. Caso a caixa seja colocada corretamente numa posição final, transforma-se em ($).

- Portas (G) e Chaves (K): Algumas áreas estão bloqueadas por portões que só se abrem depois de se recolher uma chave. Esta chave aparece automaticamente quando todas as caixas forem colocadas nas suas posições finais. Uma vez apanhada a chave, o portão pode ser atravessado, terminando o nível.

- Objetivo final: Atingir o portão (G) depois de recolhida a chave encerra o jogo com sucesso.

Implementação do jogo

O código é dividido logicamente em 7 módulos principais:

Menu.py
Apresenta um menu interativo, no qual o jogador pode escolher entre:

- Jogar manualmente o jogo Kurtan
- Executar uma das estratégias automáticas disponíveis (Iterative Deepening, A*, Simulated Annealing ou Genetic Algorithm)

Tabuleiro.pl
Contém toda a lógica do jogo propriamente dita, incluindo:

- Representação do tabuleiro e símbolos
- Regras de movimentos e empurrar caixas
- Condições de vitória
- Mecânica das chaves e portões

Iterative-Deepening .pl
Implementa o algoritmo de busca em profundidade iterativa (iterative deepening search).

- Gera estados sucessores possíveis a partir de um dado estado inicial;
- Explora esses estados com profundidade crescente;
- Verifica se um estado é objetivo (todas as caixas nos alvos e o jogador com a chave no portão).

astar.pl
Este ficheiro implementa o algoritmo de Busca A*, utilizando heurísticas para estimar o custo até ao objetivo. Leva em conta:

- Distância das caixas aos alvos;
- Posição do jogador em relação à chave e ao portão;
- Custo acumulado de cada movimento.

simulated_annealing.py
Implementa o SA, que é um algoritmo que explora o espaço de soluções de forma probabilística, permitindo aceitar soluções piores temporariamente para evitar mínimos locais, começa pouco eficiente, evoluindo ao longo do tempo. O algoritmo evolui com:

- get_initial_solution, para obter o estado inicial,
- get_random_neigh, para obter um estado aleatório seguinte,
- eval_func, que vai avaliar o estado gerado.

genetic_algorithms.py
Cada solução possível é representada como um cromossoma (sequência de movimentos), e o algoritmo evolui uma população de soluções com:

- select, para escolher o fittest de P(t-1),
- cross (crossover), para dar cross de P(t)
- mutate, para fazer mutação de P(t)
- evaluate_population, para avaliar P(t)

kurtanUtils.py
Este ficheiro irá conter funções que irão auxiliar o SA e o GA, como o apply_pull, visto que apesar do player não poder fazer pull, irá haver um "pull de estado", para serem analisados todos os estados, e impedir que caixas fiquem presas.

Representação do Tabuleiro

O tabuleiro é representado por uma matriz (lista de listas), onde cada célula contém um símbolo correspondente a um elemento do jogo:

- 'X' – parede (intransponível)
- ' ' – espaço vazio
- '*' – posição final de uma caixa
- '@' – caixa
- '$' – caixa colocada no alvo
- 'P' – jogador
- 'G' – portão (apenas acessível com chave)
- 'K' – chave (apenas aparece depois de todas as caixas estarem no target)

Iterative Deepening Search (IDS)

O algoritmo de Iterative Deepening Search (IDS) combina as vantagens da busca em profundidade com a exaustividade da busca em largura. Ele é especialmente útil em problemas com grande profundidade e onde a profundidade da solução não é conhecida à partida, como é o caso do jogo Kurtan.

Implementação no Projeto
A implementação deste algoritmo encontra-se no módulo interative_deepening.pl. O código foi desenvolvido em Prolog, explorando recursivamente os estados do jogo até encontrar uma solução.

A estrutura baseia-se nos seguintes predicados principais:

- *iterative_deepening*: inicia a busca com profundidade crescente, começando em 1 e incrementando até que uma solução seja encontrada.

- *depth_limited_search*: executa a busca com profundidade limitada. Este predicado verifica se o estado atual é objetivo e, caso contrário, continua expandindo os sucessores.

- *successor*: gera os sucessores de um dado estado, testando movimentos possíveis nas quatro direções (cima, baixo, esquerda, direita).

- *move:* define regras de transição entre estados. Inclui:
  - Movimento simples
  - Pegar chave
  - Empurrar caixas (com verificação se a posição seguinte está livre)

- *goal_state:* define a condição de vitória — todas as caixas devem estar nas posições-alvo, o jogador deve estar na posição do portão e a chave deve ter sido apanhada.

- *valid_pos*: garante que o jogador ou a caixa apenas se movem para posições dentro dos limites do tabuleiro e válidas conforme a posse da chave.

- *solve_ids*: ponto de entrada para resolução automática usando IDS. Obtém o estado inicial e imprime passo a passo a solução encontrada.

Algoritmo A*

O módulo astar.pl implementa a resolução do Sokoban através do algoritmo A*, uma técnica de busca informada que utiliza uma heurística para guiar a exploração do espaço de estados, procurando encontrar a solução de forma eficiente.

Funcionalidades principais:

- Representação do estado*:* Cada estado é representado pelo jogador, posições das caixas, caminho até ao estado atual, custo acumulado e o estado atual do tabuleiro.

- Geração de sucessores: O algoritmo gera todos os movimentos possíveis do jogador, incluindo movimentos normais e empurrar caixas, verificando a validade das ações com base no estado atual do tabuleiro e das caixas.

- Heurística: A heurística adotada é baseada na soma das distâncias Manhattan entre cada caixa e o seu alvo mais próximo. Esta avaliação estima o custo restante até à solução, orientando a busca para estados mais promissores.

- Filtragem de estados visitados: Para evitar ciclos e repetições, os estados já explorados são guardados e evitados em explorações futuras.

- Ordenação da fronteira: A lista de estados a explorar é mantida ordenada pelo custo total estimado (custo acumulado + heurística), permitindo ao algoritmo expandir primeiro os estados mais promissores.

- Execução da solução: Após encontrar o caminho até ao estado objetivo, a sequência de movimentos pode ser executada automaticamente no tabuleiro, com uma interface para visualizar cada passo da solução.

Este algoritmo oferece uma solução otimizada para o jogo, em contraste com a abordagem mais exploratória do iterative deepening, apresentando resultados mais rápidos em tabuleiros de complexidade média a elevada.

Simulated Annealing

A implementação do algoritmo de Simulated Annealing (SA) foi realizada em Python, aproveitando uma estrutura modular e orientada a objetos para representar o estado do jogo Kurtan. O objetivo do algoritmo é encontrar uma sequência de movimentos que resolva o puzzle minimizando uma função custo que avalia a qualidade do estado do jogo.

Estrutura Geral:

- Classe GameState: Representa o estado atual do jogo, incluindo o tabuleiro, posições do jogador, caixas, chave e portão. Permite clonar estados e executar movimentos válidos.
- Função simulated_annealing: Controla o processo de otimização, começando com uma temperatura inicial alta (Tmax) e diminuindo progressivamente até um limite inferior (Tmin), de acordo com uma taxa de arrefecimento R. Em cada passo, o algoritmo gera um vizinho aleatório do estado atual e decide aceitá-lo com base na diferença de custo e na temperatura.
- Função de avaliação (eval_func): Calcula o custo do estado atual considerando:

    - A soma das distâncias Manhattan das caixas que não estão sobre os alvos.
    - Penalizações para caixas bloqueadas (presas em cantos).
    - Penalizações quando a chave está visível mas não apanhada, proporcional à distância do jogador até à chave.
    - Penalizações quando a chave foi apanhada mas o jogador ainda não alcançou o portão.

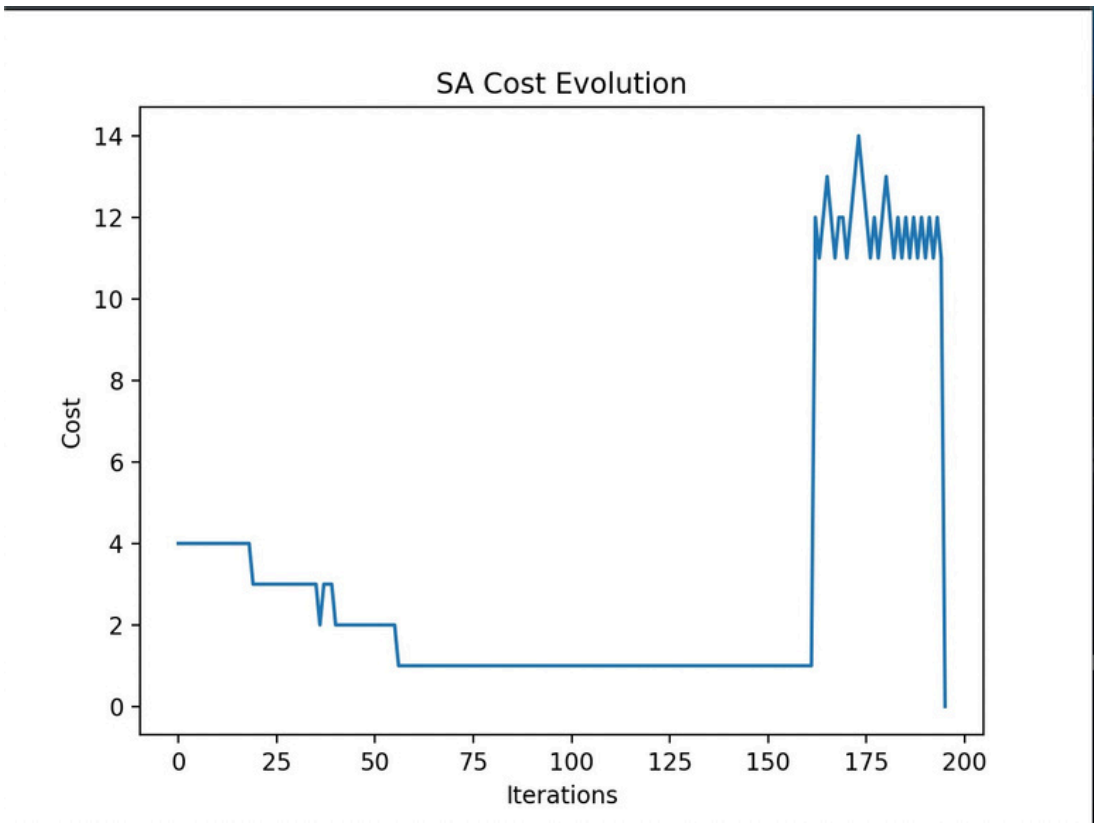
- Geração de vizinhos (get_random_neigh): Gera estados vizinhos ao atual realizando movimentos aleatórios do jogador. Caso uma caixa fique presa, tenta aplicar uma técnica de "pull" para libertá-la, aumentando a robustez do algoritmo.

Parâmetros e Funcionamento:

- A temperatura inicial Tmax controla a probabilidade de aceitar soluções piores no início, permitindo exploração ampla.
- A temperatura diminui exponencialmente em cada iteração (T = Tmax * exp(-R * t)).
- Em cada temperatura, até k vizinhos são avaliados.
- O algoritmo termina quando atinge a temperatura mínima Tmin ou encontra uma solução ótima onde a chave está apanhada e todas as caixas estão nos alvos.

Resultados e Visualização:

Durante a execução, o custo dos estados explorados é armazenado e no final é apresentada uma curva da evolução do custo ao longo das iterações. Além disso, o estado final é exibido visualmente, permitindo análise e validação da solução.



Genetic Algorithm

A implementação do Genetic Algorithm (GA) foi realizada em Python, com o objetivo de encontrar uma sequência de movimentos que leve à resolução do puzzle do jogo Kurtan, minimizando uma função de custo associada ao estado final resultante da execução desses movimentos.

Estrutura Geral:

- Classe GameState, já referido anteriormente.
- Função simulated_annealing: Controla o processo de evolução ao longo de várias gerações. Começa por gerar uma população inicial de sequências de movimentos aleatórias e aplica operadores de seleção, cruzamento e mutação para gerar novas soluções ao longo do tempo.
- Função de avaliação (eval_func): Avalia a qualidade de uma sequência de movimentos, aplicando-a a um estado inicial do jogo e medindo:

    - A soma das distâncias de cada caixa aos seus alvos (caso ainda não estejam todas no lugar).
    - Penalizações por caixas bloqueadas (presas por paredes ou outras caixas).
    - Penalizações caso a chave esteja visível mas ainda não tenha sido apanhada.
    - Penalizações se a chave tiver sido apanhada, mas o portão ainda não tiver sido alcançado.

- Geração inicial (get_initial_population): Cria uma população inicial com sequências de N movimentos aleatórios.
- Seleção (select): Utiliza torneios binários para selecionar os indivíduos mais aptos da população atual para formar a nova geração.
- Cruzamento (cross): Combina pares de indivíduos com uma probabilidade definida, utilizando crossover de ponto único para trocar segmentos das suas sequências de movimentos.
- Mutação (mutate): Com uma dada probabilidade, altera aleatoriamente um movimento dentro da sequência de um indivíduo, incentivando a diversidade da população.

Parâmetros e Funcionamento:

- tmax: Número máximo de gerações.
- popSize: Tamanho da população em cada geração.
- crossProb: Probabilidade de cruzamento entre pares de indivíduos.
- mutProb: Probabilidade de mutação por indivíduo.
- N: Comprimento de cada sequência de movimentos (genoma).

A cada geração, a população é avaliada, os melhores indivíduos são selecionados, cruzados e mutados, e os resultados da nova população são novamente avaliados. O algoritmo termina quando se atinge uma geração ótima ou o número máximo de gerações (tmax).

Resultados e Visualização:

Durante a execução, são registados:

- O melhor valor de fitness (menor custo) por geração.
- A média do fitness da população por geração.

Estes dados são apresentados em dois gráficos:

1. Melhor fitness absoluto ao longo das gerações.
2. Fitness percentual comparado com o valor ótimo (se definido), tanto para o melhor indivíduo como para a média da população.

Além disso, o número total de avaliações realizadas e a melhor solução encontrada são apresentados no final. O estado final correspondente à melhor sequência pode ser impresso para análise.

Conclusão

O projeto permitiu explorar a aplicação prática de diferentes técnicas de Inteligência Artificial na resolução de um problema clássico. Cada algoritmo demonstrou forças e fraquezas distintas, sendo adequados a diferentes contextos e objetivos.

A implementação em Prolog e Python permitiu compreender melhor as abordagens baseadas em procura e otimização, respetivamente. Em projetos futuros, seria interessante expandir a complexidade do jogo, incluindo múltiplos níveis e elementos como escadas, ou ainda integrar os algoritmos num sistema híbrido.

No geral, o projeto contribuiu significativamente para o desenvolvimento de competências em planeamento, raciocínio heurístico e otimização baseada em busca.