instituto superior de engenharia de lisboa

Segundo trabalho prático

Cadeira de Inteligência Artificial

Francisco Lufinha nº49447

Rodrigo Vitorino nº49448

Guilherme Barradas nº49458

Introdução

No âmbito da unidade curricular de Inteligência Artificial do 3º Ano / 6º Semestre da licenciatura em Engenharia Informática de Computadores, foi proposta a implementação de algoritmos de resolução aplicados ao jogo *Kurtan*.

O jogo *Kurtan*, fortemente inspirado no clássico *Sokoban,* consiste num puzzle em que o jogador deve empurrar caixas para posições-alvo dentro de um mapa limitado por paredes, seguindo regras definidas de movimentação e interação com o ambiente.

Neste trabalho, iremos descrever detalhadamente a metodologia de desenvolvimento e a implementação dos algoritmos utilizados para resolver o jogo e um foco na representação do estado do jogo e o gerar e avaliar dos movimentos (através da metodologia própria de cada um dos algoritmos).

Deste modo, iremos terminar o trabalho com uma reflexão crítica sobre as abordagens tomadas demonstrando os conhecimentos adquiridos ao longo da unidade curricular.

Iterative-Deepening

De modo a implementar o *iterative-deepening*, foi utilizada a linguagem *Prolog*. Este algoritmo foi aplicado ao jogo *Kurtan*, cuja lógica foi modelada com recurso a predicados e pares de coordenadas para representar a localização de cada um dos elementos do mapa.

Esta abordagem apresenta uma diferença significativa em relação ao trabalho anterior, onde foi desenvolvida a lógica do jogo das damas através de uma estrutura baseada em matrizes (*arrays* de *arrays*). A mudança de representação tem impacto direto na forma como os estados do jogo são definidos e manipulados, influenciando a lógica de navegação entre estados.

O algoritmo *iterative deepening* combina os benefícios da profundidade limitada com a completude da procura em largura. Sumamente, este algoritmo realiza buscas sucessivas em profundidade com profundidades máximas crescentes até encontrar uma solução. Cada iteração explora a árvore de estados até à profundidade atual, reiniciando a procura com uma profundidade maior caso a solução ainda não tenha sido encontrada.

No contexto do jogo *Kurtan,* cada estado representa uma configuração do tabuleiro e cada ação possível gera um novo ramo na árvore de estados. Assim, um estado com quatro movimentos válidos gera quatro novos estados onde cada um corresponde a um movimento distinto.

A lógica do algoritmo baseia-se em três predicados principais: *path*, *s* e *m*.

O predicado *m* é responsável por aplicar a lógica do jogo, incluindo o movimento do jogador, o empurrar das caixas e o aparecimento de elementos especiais como a chave.

S, gera os estados sucessores a partir de um estado atual. Este chama internamente o predicado *m* para realizar a transição de estado.

Por fim, *path* implementa a lógica de procura, explorando os estados gerados e aplicando o mecanismo de profundidade limitada com incremento progressivo.

Iterative deepening trata todos os movimentos com custo uniforme, o que simplifica a implementação, mas pode tornar-se ineficiente em jogos como o *Kurtan*, onde o número de estados possíveis cresce exponencialmente. Apesar de ser um algoritmo completo e mais eficiente do que *breadth-first search*, a sua implementação continua a visitar múltiplas vezes estados já vistos e considerando jogadas sem valor relevante para o contexto do problema, incrementando o custo computacional total.

*Best-first search / A**

Contrariamente ao algoritmo *iterative deepening*, o *best-first search*, particularmente na sua variante *A**, não realiza uma procura exaustiva sobre todos os ramos possíveis da árvore de estados. Em vez disso, foca-se em expandir os nós mais promissores com base numa função de avaliação heurística, tentando assim encontrar a solução de forma mais eficiente em termos de tempo.

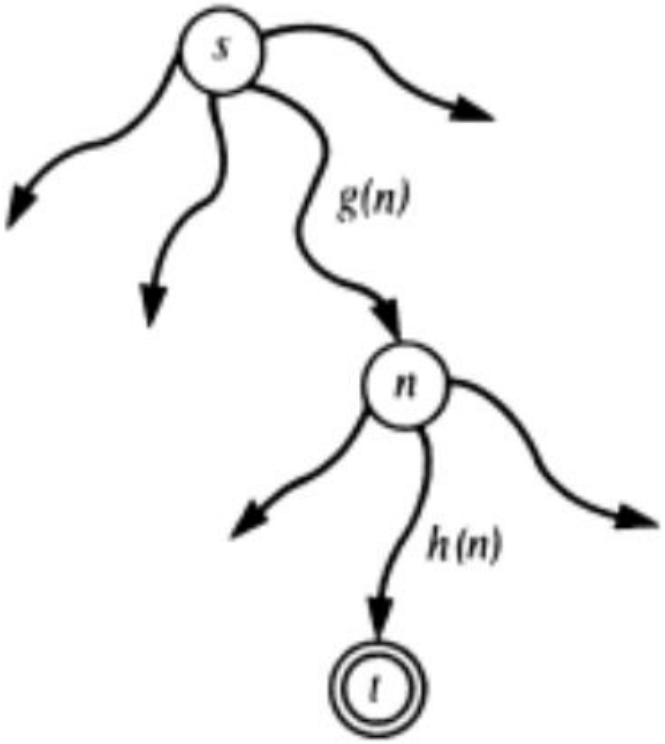

*Figura 1 – Lógica do algoritmo A**

Enquanto a abordagem de *depth-first* garante sempre a descoberta de uma solução (desde que esta exista e o espaço de procura seja finito), o algoritmo A* toma decisões baseadas em avaliações locais, podendo em alguns casos descartar caminhos que, mais adiante, levariam à solução mais eficiente.

Para a implementação da função de avaliação heurística foi utilizada a distância de *Manhattan*, calculada entre cada caixa e o *goal* mais próximo, representando a distância mínima (em termos de movimentos horizontais e verticais) necessária para posicionar as caixas, e a verificação de *deadlocks* simples. Para a verificação dos *deadlocks*, é feita uma análise das caixas que ainda não estão em posições-alvo. Se alguma destas se encontrar num canto ou rodeada por paredes de forma a impedir movimentos futuros, o estado é penalizado. Esta verificação não cobre todos os casos de *deadlock* possíveis, mas ajuda a filtrar uma parte significante de estados sem possível solução.

O uso da heurística permite ao *A** navegar de forma mais direcionada no espaço de estados, reduzindo de foram significantes o número de estados explorados em

comparação com abordagens exaustivas. No entanto, esta eficiência tem um custo: se a função heurística não for suficientemente precisa, o algoritmo pode seguir caminhos não ideais.

Mesmo com a verificação parcial de *deadlocks* implementada na heurística, a complexidade do jogo pode levar a que estados com potenciais soluções sejam ignorados, resultando num compromisso entre eficiência e resolução do problema.

Simulated Annealing

Embora as duas primeiras tarefas deste trabalho tenham sido desenvolidas em *Prolog*, a implementação do algoritmo *simulated annealing* (bem como do algoritmo genético, abordado na secção seguinte) foi realizada em *matlab* devido à maior complexidade e natureza destas abordagens. A linguagem *matlab*, com sintaxes mais simples para operações com matrizes, revelou-se mais adequada para a manipulação e avaliação de estados no jogo *kurtan*.

Sinulated annealing é um algoritmo de otimização que consiste em explorar o espaço de soluções através da geração aleatória de vizinhos do estado atual, aceitando ou rejeitando novos estados com base numa função de avaliação e uma variável de controlo chamada temperatura.

Ao contrário de algoritmos deterministas como o *A** ou *DFS,* o *simulated annealing* permite, especialmente em fases iniciais, aceitar soluções com pior desempenho que a atual. Esta estratégia tem como objetivo evitar ficar preso em posições benéficas apenas do ponto de vista do algoritmo de avaliação (um problema comum em algoritmos como o *best-first search*). A probabilidade de aceitar uma solução pior diminui com a redução da temperatura ao longo do tempo, regulando assim o grau de exploração do espaço de estados.

A implementação em *matlab* recorre à representação do estado do jogo através de matrizes, facilitando o controlo do mapa e a aplicação de movimentos. O algoritmo foi modularizado em várias funções principais:

- *getInitialSolution*: gera uma sequência inicial aleatória de movimentos a partir do estado inicial;

- *evalFunc:* avalia a qualidade de um estado com base em critérios como número de caixas posicionadas corretamente e distância aos objetivos;
- *getRandomNeigh:* produz uma solução vizinha válida;

- *p*: calcula a probabilidade de aceitar uma solução pior com base na diferença de avaliação e na temperatura atual;

- *newTemp:* atualiza a temperatura de acordo com um calendário de arrefecimento;

- *sa:* função principal que integra todas as anteriores, realizando iterações até atingir uma solução aceitável ou detetar um *deadlock*.

Comparando com algoritmos deterministas implementados anteriormente, o *simulated annealing* apresenta como vantagem que permite encontrar soluções mais eficientes por divergir da função de avaliação e é geralmente mais rápido por evitar a exploração exaustiva de todos os ramos da árvore de estados.

Com isso em mente, existem também limitações: a natureza aleatória pode gerar *deadlocks* ou estados inválidos, especialmente se a função de vizinhança não for cuidadosamente definida; requer afinação de parâmetros, como a temperatura inicial, a taxa de arrefecimento e critério de paragem, os quais têm impacto direto na eficácia e desempenho do algoritmo.

Apesar destes desafios, quando o algoritmo converge corretamente, os resultados obtidos podem ser tão bons ou melhores do que os gerados por outras abordagens, e a um custo computacional significativamente inferior.

Genetic Algorithm

Tendo em consideração os restantes algoritmos mencionados neste relatório, *genetic algorithm* apresenta-se como uma solução eficiente de modo semelhante ao *simulated annealing* mas com uma vantagem sobre o quão aleatórias são os movimentos realizados. Contrariamente ao *simulated annealing* que utiliza o gerar de movimentos aleatórios durante toda a sua execução, o *genetic algorithm* usa este apenas como um ponto de partida. Após a primeira geração são usadas funções de mutação e de *crossover* para melhorar as soluções aleatórias geradas (mantendo algum grau de aleatoriedade, mas mais reduzido).

O processo começa com a geração de uma população inicial de soluções. De forma semelhante ao exercício anterior, utilizamos uma função *getInitialSolution* para criar este conjunto aleatório. Cada solução é um vetor de inteiros de tamanho *N*, onde cada inteiro codifica um movimento específico: 0 para sem movimento, 1 para cima, 2 para baixo, 3 para esquerda e 4 para direita.

Após a criação das soluções iniciais, a função *evalFunc* entra em ação para avaliar cada uma delas. Essa avaliação simula a sequência de movimentos codificada no vetor e calcula um custo associado à posição final gerada. Esse custo é uma métrica abrangente que considera os passos utilizados (o número de movimentos diferentes de 0 realizados), a distância de Manhattan de cada caixa até o gol mais próximo, penalidades ou recompensas baseadas em caixas que estão fora ou dentro dos *goals*, e custos adicionais para situações onde as caixas ficam presas ou em posições sem saída, os chamados *deadlocks*. Com as soluções avaliadas, passamos para a função de seleção, que é responsável por escolher os indivíduos mais promissores para a próxima geração. Atualmente, esta função seleciona aleatoriamente dois pais da população atual.

Uma vez selecionados os pais, o processo de reprodução começa com a função *onePointCrossover*. Com uma probabilidade *crossProb*, um ponto de corte escolhido aleatoriamente no vetor de movimentos. As "caudas" dos dois pais são então trocadas, combinando características de ambos para criar duas soluções. Para garantir a diversidade genética e evitar que o algoritmo fique preso em mínimos locais, as novas soluções passam pela função de mutação. Com uma probabilidade *mutProb*, a função escolhe uma posição aleatória no vetor de movimentos e atribui a ela um novo valor, selecionado aleatoriamente entre 0 e 4**.** Este ciclo de geração, avaliação, seleção, cruzamento e mutação é repetido ao longo de várias gerações, permitindo que o algoritmo evolua soluções cada vez mais eficientes para o problema em questão.

Conclusão

Tendo em conta os resultados obtidos nas diversas implementações realizadas por este grupo, consideramos fundamental fazer uma reflexão crítica sobre as soluções desenvolvidas. Embora as implementações demosntrem um conhecimento sólido, tanto teórico como prático, relativamente a cada um dos algoritmos estudados, é importante reconhecer as limitações existentes.

Em particular, o trabalho não apresenta uma solução completa para o jogo *kurtan*, focando-se maioritariamente na solução do *sokoban*, que constitui uma parte do problema e não inclui a componente da chave final (embora exista no *prolog* a implementação do portão). Ainda assim, acreditamos que o projeto evidencia uma compreensão técnica adequada dos algoritmos abordados, bem como uma capacidade de adaptação à estrutura da implementação proposta, especialmente nas versões desenvolvidas em *matlab*.

Por fim, este trabalho serviu como uma importante experiência pedagógica, permitindo-nos consolidar conhecimentos em inteligência artificial aplicada a problemas como jogos e explorar diferentes paradigmas algorítmicos, desde a procura exaustiva até métodos heurísticos.