Faculdade de Ciências da Universidade do Porto

Inteligência Artificial (CC2006) - 2019/2020

Métodos de Procura Para Vigilância de Partições Retangulares

Relatório do 1º Projeto de Inteligência Artificial

Énio Sousa up201405702

Mateus Almeida 201805265

Introdução

Este relatório foi realizado no âmbito do primeiro projecto da unidade curricular de Inteligência Artificial. O seu propósito é estudar diferentes algoritmos de pesquisa como métodos de resolução do problema de vigilância de partição retangulares, elaborando nas vantagens e dificuldades encontrados quanto à sua aplicação.

Um algoritmo de procura é aquele que resolve um problema de recolha de informação guardada numa estrutura de dados ou em qualquer espaço de procura específico ao problema. Para o problema a analisar, determinou-se os requisitos para a aplicação dos diferentes métodos de pesquisa e elaborou-se as estruturas e funções auxiliares necessárias para tal realizar, adaptando, quando necessário, às diferentes abordagens.

A implementação dos algoritmos e ficheiros auxiliares foi feita inteiramente em C, excepto para a implementação do modelo matemático dos exercícios 4. e 5., que foi feita em Prolog.

Índice

Estruturas de Dados

Escolheu-se uma abordagem de implementar todas as estruturas de dados de raíz, existindo na pasta "Alínea 1, Greedy" três distintos ficheiros headers e três respetivos ficheiros implementado as funções necessárias para o algoritmo.

O fichero errorMessage.h contém a API de erros detectáveis pelo algoritmo. O ficheiro errorMessage.c implementa essa API, criando as funções:

- errorMessageMem - encerra a execução do programa com EXIT_FAILURE, imprimindo para o stderr onde falhou a alocação de memória.
- errorMessageFree - encerra a execução do programa com EXIT_FAILURE, imprimindo para o stderr onde falhou a libertação de memória.
- errorMessageMaxCapacity - encerra a execução do programa com EXIT_FAILURE, imprimindo para o stderr onde se chegou à capacidade máxima.
- errorMessageIllegalSize - encerra a execução do programa com EXIT_FAILURE, imprimindo para o stderr que o tamanho da lista é menor que 0.
- errorMessageListSize - encerra a execução do programa com EXIT_FAILURE, imprimindo para o stderr que o tamanho da lista não encapacita todo os objetos a tentarem ser inseridos.
- errorMessageIndex - encerra a execução do programa com EXIT_FAILURE, imprimindo para o stderr que não foi inserido o indíce correto de um objeto na lista.

O ficheiro point.h define as estruturas:

Point, constítuida pelos inteiros x e y, representando as coordenadas de um vértices e a API de

diferentes funções de acesso e comparação de dados utilizando a estrutura.

O ficheiro point.c implementa essa API, criando as funções:

- newPoint - cria uma nova instância Point utilizando a função calloc() da biblioteca stdlib.h e instanciando o valor das coordenadas.
- equalPoint - retorna 1 se dois Point têm coordenadas com os mesmos valores e 0 caso contrário.
- comparePoint - retorna a diferença das ordenadas de dois Point se o valor das abcissas forem iguais e a diferença das ordenadas caso contrário.
- printPoint - imprime as coordenadas do Point.

O ficheiro read.h define as estruturas:

- PointMemory, estrutura constituída por um ponto Point, um inteiro nR, o número de retângulos adjacentes e o array idR, constituído pelo identificador desses retângulos. Servirá para conter a informação de cada vértice individual.
- RecMemory, estrutura constituída por um inteiro idR, identificador do retângulo, array de pontos Points idP, contém todos os pontos incidentes no retângulo e o inteiro nP, o número total de pontos. Servirá para conter a informação de cada retângulo.
- Memory, estructura constituída por um array PointMemory, inteiros pMemSize e pMemMaxSize, o número de pontos totais e o limite de pontos que poderão existir numa instância, respetivamente, um array RecMemory e um inteiro rMemMaxSize, o número de retângulos da instância do problema. Servirá para conter a informação toda de input numa única estrutura, de forma a facilitar acesso aos dados.

Contém também definida a constante MAXSIZE de valor 10240 e a API de diferentes funções de acesso e comparação de dados utilizando as estruturas acima.

O ficheiro read.c implementa essa API, criando as funções:

- newMemory - cria uma nova instância Memory utilizando a função calloc() da biblioteca stdlib.h, instanciado o limite de pontos como MAXSIZE e criando as instâncias dos arrays PointMemory e RecMemory.
- readInput - dada uma instância Memory e o número de retângulos da instância do problema, recorre à função readLine() para ler cada linha do input do stdin. De seguida, o array PointMemory é ordenado recorrendo à função sortPointMemory().
- readLine - dada uma instância Memory e RecMemory, lê uma linha de input. É lido o identificador de retângulo e o número de pontos, guardando-os em RecMemory, seguido da

informação de cada ponto. Tal informação é adicionada a PointMemory recorrendo à função addToPointMemory(), procurando de antemão se o ponto já existe com searchMemory(). Por fim, adiciona-se o ponto ao array de pontos no índice específico, em RecMemory.

- addToPointMemory - adiciona um novo ponto caso ainda não exista no array PointMemory. Caso contrário, adiciona a informação sobre os retângulos adjacentes.
- comparePointMemory - compára duas posições de memória do array PointMemory.
- searchMemory - dada uma instância PointMemory e um Point, procura no array se existe algum Point igual, retornando o seu índice, caso contrário retorna a primeira posição não válida.
- sortPointMemory - ordena as posições de memoria do array PointMemory usando a função comparePointMemory() e qsort() built in do C.
- printMemory - imprime o PointMemory e RecMemory, recorrendo às funções auxiliares printPointMemory() e printRecMemory().
- printPointMemory - imprime todas as posições de memória do array PointMemory, chamando as funções auxiliares printPoint() e printArray().
- printArray - imprime um array de inteiros.
- printRecMemory - imprime todas as posições de memória do array RecMemory e o conteúdo das suas variáveis.
- freeMemory - liberta o espaço ocupado pela estrutura Memory, recorrendo às funções auxiliares freePointMemory(), freeRecMemory() e free() da biblioteca stdlib.h.
- freePointMemory - liberta individualmente o espaço ocupado por cada Point e pela estrutura PointMemory.
- freeRecMemory - liberta o espaço ocupado pela estrutura RecMemory.

Na pasta BlindSearch, os ficheiros errorMessage.h, errorMessage.c, point.h, point.c, read.h e read.c servem o mesmo propósito, implementando as mesmas estruturas, sendo que a estrutura Point e a sua API foram exportadas para os ficheiros point.c e point.h, respetivamente.

O ficheiro state.h define as estructuras que representarão os estados a serem utilizados pelos métodos de pesquisa e a API das funções que acedem à informação neles guardados.

A estrutura State é constituída por diferentes variáveis: um array de inteiros idR, em que o seu índice representa o identificador do rectângulo e a sua imagem o número de vezes que o vimos; o inteiro iLim que define o tamanho do array idR; o inteiro index, índice do último ponto visto; o inteiro g, a geração atual do estado; o inteiro h, subestimação do custo para chegar a uma solução; o inteiro f, a soma de g

e f; o inteiro seen, o número de diferentes retângulos cobertos pelo estado e, por fim, pai, um apontador para o estado-pai.

O ficheiro state.c implementa as funções do API:

- newState - Aloca dinamicamente memória para um novo estado e retorna o seu apontador. Esta função irá copiar toda a informação contida em pai e irá adicionar um novo ponto representado por index. Caso o pai seja null, iremos apenas buscar a informação contida no ponto.
- copyIntArray - copia o conteúdo de um array de inteiros para outro.
- getNewRectangles - Adiciona os retângulos adjacentes do novo ponto ao array idR do estado.
- calculateSeen - calcula o número de retângulos que um dado estado está a vigiar.
- freeState - liberta toda a memória alocada dinamicamente na função newState().
- freeStateIdR - liberta o espaço alocado para o array idR de um dado estado.
- printState - imprime o conteúdo de um estado.
- printIntArray - imprime o conteúdo de um array de inteiros.

O ficheiro stateArray.h define a estrutura que conterá os diferentes estados a ser referenciados nos métodos de procura, organizados num array. StateArray contém o array de estados, state, e os inteiros size e maxSize que representam o tamanho atual e o tamanho máximo do array, respetivamente. A sua API define os métodos a serem implementados em stateArray.c.

O ficheiro stateArray.c implementa as funções do API:

- newStateArray - aloca dinamicamente a memória necessária para o StateArray e inicializa os seus valores com o limite de estados 10 milhões, retornando o seu apontador. Poderá ser vantajoso iniciar com um número menor de posições.
- addToStateArray - adiciona um estado ao array, StateArray..
- freeStateArray - liberta a memória alocada para o StateArray recorrendo à função auxiliar freeState().
- printStateArray - imprime os conteúdos de StateArray, recorrendo à função printState() para imprimir cada posição/estado.

O ficheiro stateLinkedList.h define as estruturas SNode e SList a, organizados numa lista ligada e a API para aceder e adicionar informação à lista. Define as estruturas:

- SNode - constituído por um estado State state e dois apontadores SNode prev e next, que apontam para o nó anterior e próximo, respetivamente.
- SList - constituído por um inteiro size, o número de nós na lista, e dois apontadores SNode first e last, que apontam para o primeiro e último nó respetivamente.

O ficheiro stateLinkedList.c implementa as funções do API:

- newStateList - aloca dinamicamente o espaço para a lista de estados, retornando o seu apontador.
- newStateNode - aloca dinamicamente o espaço para o nó, retornando o seu apontador.
- insertStateFirst - insere um novo nó no início da lista.
- pushState -sendo a lista utilizada como um stack, insere um novo nó no início da lista recorrendo à função auxiliar inserFirstLast().
- insertStateLast - insere um novo nó no fim da lista.
- enqueueState - sendo a lista utilizada como uma queue, insere um novo nó no fim da lista recorrendo à função auxiliar inserStateLast().
- removeFirstState - retira o primeiro nó da lista.
- popState - sendo a lista utilizada como um stack, retira o primeiro nó da lista, recorrendo à função auxiliar removeFirstState().
- dequeueState - sendo a lista utilizada como uma queue, retira o primeiro nó da lista, recorrendo à função auxiliar removeFirstState.()
- stateListSize - retorna o número de nós na lista.
- printStateList - imprime to conteúdo de todos os estados em cada respetivo nó, referenciados por índice.
- freeStateList - liberta a memória alocada para a lista, recorrendo às funções auxiliares dequeueState() e freeState().

O ficheiro integerLinkedList.h define as estruturas INode e IList, que conterão os índices dos pontos e a API de funções a ser referenciados nos métodos de procura. Definiu-se as estruturas:

- INode - constituída por um inteiro value e dois apontadores INode, prev e next, que apontam para o nó anterior e próximo, respetivamente.
- IList - constituída por um inteiro size, o número de nós na lista e dois apontadores first e last, que apontam para o primeiro e último nó, respetivamente.

O ficheiro integerLinkedList.c implementa as funções da API:

- newIntegerList - aloca dinamicamente o espaço para a lista, retornando o seu apontador.
- newIntegerNode - aloca dinamicamente o espaço para o nó, retornando o seu apontador.
- insertIntegerFirst - insere um novo nó no início da lista.
- pushInteger - estando a lista de inteiros ser tratada como um stack, insere um novo nó no início da lista, recorrendo à função auxiliar insertIntegerFirst().
- insertIntegerLast - insere um novo nó no fim da lista.
- enqueueInteger - estando a lista de inteiros a ser tratada como uma queue, insere um novo nó no fim da lista, recorrendo à função auxiliar insertIntegerLast().
- removeFirstInteger - remove o primeiro nó da lista.
- popInteger - estando a lista de inteiros a ser tratada como um stack, remove um novo nó do início da lista, recorrendo à função auxiliar insertIntegerLast().
- dequeueInteger - estando a lista de inteiros a ser tratada como uma queue, remove o primeiro nó da lista, recorrendo à função auxiliar removeFirstInteger().
- removeLastInteger - remove o último nó da lista.
- integerListSize - retorna o número de nós da lista.
- getFirstInteger - retorna o valor do primeiro nó da lista.
- topInteger - retorna o valor do primeiro nó da lista, recorre à função auxiliar getFirstInteger().
- getLastInteger - retorna o valor do último nó da lista.
- searchIntegerList - procura se um certo valor está contido na lista, retornando 1 se estiver e 0 se não.

- printIntegerList - imprime os conteúdos de cada nó da lista e o número de nós.

- freeIntegerList - liberta o espaço alocado na memória para a lista, recorrendo à função auxiliar dequeueInteger().

O ficheiro extraFunctions.h implementa a API de funções auxiliares para a execução dos diferentes métodos de procura, sendo implementadas no ficheiro extraFunctions.c como:

- isSolution - retorna 1(true) quando o estado em causa já visualizou todos os retângulos, isto é, a solução na forma do caminho até ele garante uma total cobertura dos retângulos.

- updatePath - auxilia o backtracking necessário a métodos de pesquisa em profundidade.

- bestPath - obtém os valores dos estados no caminho-solução do algoritmo a ser analisado, garantindo que a lista sol, onde ficarão os valores dos estados, tem o tamanho da lista path, onde estão os estados-solução.

- randomSolution - escolhe pontos aleatórios até gerar um estado que seja solução, retornando o seu apontador.

- pickRandomPoint - escolhe um índice aleatório e utiliza a função moveIndexLeft para retornar o índice de um ponto que ainda não foi escolhido.

- moveIndexLeft - decrementa o índice atual, retornando ao valor do número de pontos caso seja menor do que 0.

- printPath - imprime o caminho destruindo o stack.

- printPath1 - imprime caminho sem destruir os elementos da lista.

- printPath 2 - imprime o caminho do estado filho ao estado pai até chegar a NULL.

- printSolution - imprime o caminho de estados na forma de pontos e o seu tamanho.

- feasible - verifica se ao tirar o ponto de índice igual ao index de um dado estado state, se ele continua feasible, isto é, continua a ver todos os rectangulos.

- takePoint - tira o ponto pMem[index] de um estado state e retorna um novo estado sem esse ponto.

O ficheiro converter.c lê todas as instâncias de input, recorrendo às estruturas Memory e funções auxiliares readInput() e eclipseMode(), que cria 4 diferentes predicados para conter os dados de input e os imprime no ficheiro data.ecl. Os predicados são nrec/1, o número de retângulos da instância do

problema, p/3, que atribui um identificador a cada ponto e guarda as suas coordenadas, rec/3, que contém o identificador de retângulo, o número de vértices incidentes e uma lista com os identificadores desses vértices e guardar/1 que contém uma lista de identificadores de retângulos.

Nas pastas Alínea 1, Alínea 2 e Alínea 3 existe uma sub-pasta Performance Evaluation. Nela estão guardados ficheiros para a avaliação da performance de cada algoritmo.

O ficheiro getTimes.sh é um scrip de bash que executa um ficheiro n vezes e calcula o tempo de cada execução, colocando-os no ficheiro times.txt para posteriormente ser analisado.

O ficheiro rating.c recebe 3 argumentos: número de vezes a executar o programa, o caminho relativo do programa a executar e o caminho relativo do caso de teste. Iremos criar um novo processo que executa getTimes.sh usando a função execv() e posteriormente é aberto e lido o ficheiro times.txt para calcular o tempo médio de execução.

1. Estratégias Greedy

Um algoritmo greedy é qualquer algoritmo cujo objetivo é encontrar a solução ótima local em qualquer estado do problema a ser resolvido, com intenção de encontrar um ótimo global.

Implementação

O ficheiro greedy.c implementa o algoritmo. As funções auxiliares aqui implementadas executam efetivamente o algoritmo, sendo:

- initiateArray - dado um array de inteiros, um tamanho e um valor, inicializa o array com o valor.

- initiateHeap - insere na heap os vértices, tendo como chave um índice de 1 a N, sendo N o número de vértices únicos no input, e o valor o número de retângulos adjacentes desse ponto. Essa inserção recorre à função insertInHeap da API da estrutura heap, ordenando assim de modo a forma uma heapMax em que o máximo valor será aquele do vértice com mais retângulos a ele adjacente.

- member - dado um inteiro e um array de inteiros, retorna 1 se o valor está contido no array ou 0 se não está.

- copyArray - copia os conteúdos de um array para outro.

- updatePosition - atualiza o número de retângulos adjacentes não vigiados de um vértice no array PointMemory.

- updateMember - dada uma instância Memory e um array de identificadores, atualiza um dos valores da heap, decrementando o número de retângulos adjacentes caso a o array de retângulos adjacentes partilhe um identificador com o array.

- update - atualiza o número de retângulos adjacentes não vigiados de cada ponto e reinsere a nova informação na heap, recorrendo à função auxiliar updatePosition() e changeValue(), da API da estructura heap, para atualizar a heap.

- solve - executa o algoritmo.

Para n instâncias, é criado uma instância da estrutura Memory, recebe-se o input para Memory, recorrendo à função readInput(). De seguida, chama-se a função solve() que irá executar o algoritmo, imprimindo os vértices onde colocar as guardas. Por fim, liberta-se o espaço de memória, desconstruindo-se o array mem, freeMemory(), e a heap, destroyHeap().

A função solve encontra uma solução recorrendo a métodos greedy. A execução do algoritmo consiste em inicializar a heap com índices como chaves e o número de retângulos adjacentes de cada vértice como valor ordenante, de seguida retirar o elemento máximo da heap, isto é, o ponto com maior número de retângulos adjacentes e atualizar os restantes pontos na heap para não considerarem os retângulos no qual o vértice retirado incide. Encontra-se uma solução quando todos os retângulos estão vigiados, isto é, não existe mais nenhum elemento do heap com número de retângulos adjacentes positivo.

Resultados

O algoritmo greedy não é óptimo, portanto existe instâncias em que não retorna a melhor solução possível. No entanto apresenta os melhores resultados temporais. Podemos também afirmar que tem uma boa gestão espacial, porque, como veremos mais a frente para outros algoritmos, o factor limitante é a eficiência espacial.

O tempo médio de execução de alguns testes foram calculados usando o programa rating.

- G75dados10_500: 0.01 segundos
- G75dados20_500: 0.04 segundos
- G75dados100_200: 0.26 segundos
- G75dados150_200: 0.54 segundos
- G75dados200_200: 0.97 segundos
- G75dados300_100: 1.15 segundos
- G75dados450_100: 2.69 segundos
- G75dados700_50: 3.28 segundos
- G75dados2500_10: 9.29 segundos
- G75dados5000_10 39.12 segundos

2. Métodos de Pesquisa Não Informada e Informada

Um método de pesquisa não-informada é aquele a ser aplicado a problemas onde está determinado o grafo, o nó inicial e o destino, mas não o caminho a percorrer. O método de pesquisa define a ordem pela qual os diferentes possíveis caminhos são percorridos, sendo que o método de seleção dos caminhos difere dependendo do método de pesquisa.

Por outro lado, um método de pesquisa informada, ou pesquisa heurística, utiliza informação heurística no momento de seleção do nó para guiar a pesquisa, nomeadamente, o custo de um caminho resultante da aplicação da função heurística ao nó. A função heurística toma um nó e retorna um número não-negativo real que é a estimativa do custo do caminho-solução de custo menor, sendo a função admissível se os seus resultados forem sempre menor ou iguais ao custo verdadeiro do caminho-solução de custo menor a ser analisado.

Breadth-First Search

No método de pesquisa Breadth-First Search, o conjunto de caminhos são implementados numa queue First-In-First-Out, logo o caminho a ser selecionado a cada iteração do ciclo é aquele que foi primeiro adicionado. Esta abordagem implica que os caminhos são gerados a partir do nó inicial por ordem do número de arcos num caminho, sendo o caminho com menor número de arcos selecionado a cada iteração.

Implementação

O ficheiro bfs.c implementa o algoritmo. As funções auxiliares aqui implementadas são:

- initiateQueue - dado um StateArray, Memory e lista de inteiros IList, inicializa todos os estados state do StateArray recorrendo à função newState() e adiciona o respetivo index desse state a IList recorrendo à função enqueueInteger().

- solve - executa o algoritmo.

A função solve executa o algoritmo, inicializando o StateArray e uma lista de inteiros como uma queue, recorrendo à função initiateQueue(). Retira então o primeiro índice da queue e, para o correspondente estado presente no State Array, quebra o ciclo se o número de retângulos vigiados no caminho até ao estado é igual ao número total de retângulos. Caso sim, imprime o caminho do estado atual ao seu pai até chegar a NULL e liberta o espaço da memória locado. Caso não, adiciona os novos estados vizinhos e adiciona-os à queue.

Resultados

O algoritmo bfs neste contexto do problema irá encontrar soluções ótimas, no entanto devido o árvore de geração ser demasiado “larga” a sua gestão espacial é horrível e será o grande fator limitante da aplicação deste algoritmo para este exercício. Veremos também que tem um melhor eficiência temporal que o dfs, pelo simples facto de termos alterado o dfs para encontrar soluções ótimas.

Calculamos o tempo médio de execução para dois pequenos casos de teste.

- G75dados10_500: 0.37 segundos

- G75dados20_1: O algoritmo fica sem memoria.

Depth-First Search

No método de pesquisa Depth-First Search, o conjunto de caminhos são implementados num stack Last-In-First-Out, isto é, os elementos são adicionados e retirados sempre do topo do stack. A sua utilização significca que o caminho selecionado e removido é sempre o último a ser adicionado.

Esta implementação implica a procura num caminho até chegar ao seu fim e depois fazer backtracking para a procura num outro caminho diferente, sendo perigosa a implementação em casos de caminhos de comprimento infinito ou na presença de ciclos.

Implementação

O ficheiro dfs.c implementa o algoritmo. As funções auxiliares aqui implementadas são:

- initiateStack - cria N estados state, sendo N o número de retângulos da instância, recorrendo à função newState() e empurra-os para a instância SList, o stack.

- solve - executa o algoritmo.

A função solve cria um stack onde são colocados e do qual serão retirados todos os estados em conjunto com duas listas de inteiros que contém os índices dos estados: path, onde é guardado um caminho corrente que é constantemente atualizado, e sol, o caminho hipótese solução e define como o tamanho mínimo do caminho minPath, um terço do número de retângulos. De seguida, procura o caminho solução, retirando o último estado a ser colocado no stack e, recorrendo à função updatePath(), atualiza o caminho de estados até ao último retirado. Se o estado retirado for uma solução, isto é, vigia todos os retângulos, então obtém-se um caminho solução que, se tiver tamanho menor ou igual a minPath, é ótimo. Caso contrário, adiciona os novos estados ao stack. Por fim, imprime o caminho solução e liberta o espaço na memória das estruturas.

Resultados

O algoritmo dfs não é suposto ser ótimo, no entanto modificamos os parâmetros de procura, para encontrar uma solução ótima. Isto afeta significativamente a eficiência temporal. A sua gestão de memória também foi alterada de modo a obter a melhor eficiência possível. Correntemente estamos a usar uma stack para guardar o caminho de procura e atualizamos esta stack sempre que entramos num novo ramo de procura.

Tentamos calcular o tempo médio de execução de alguns testes, mas infelizmente este algoritmo modificado para encontrar soluções óptimas, não é ideal para grandes casos de teste.

- G75dados10_500: 19.50 segundos

-

Iterative DFS/Iterative Deepening

O método de pesquisa Iterative Deepening baseia-se na abordagem DFS, procurando combater os seus problemas como a possibilidade de não encontrar solução em grafos infinitos ou cíclios. Recorre, então, a um inteiro para designar a profundidade e nunca explorar para lá dessa profundidade até procurar em todos os arcos dela. Se não encontrar uma solução, cria novos estados e procura em novos caminhos de maior pfoundidade. Eventualmente, caso exista, encontrará uma solução e, semelhante a BFS, como está a criar novos caminhos por ordem do número de arcos, o caminho com o menor número de arcos será encontrado primeiro.

Implementação

O ficheiro iterativeDfs.c implementa o algoritmo. As funções auxiliares aqui implementadas são:

- solve - executa o algoritmo.

A função solve cria um stack onde são colocados e do qual serão retirados todos os estados em conjunto com duas listas de inteiros que contém os índices dos estados: path, onde é guardado um caminho corrente que é constantemente atualizado, e sol, o caminho hipótese solução. De seguida, retira um estado do stack e atualiza o caminho de estados percorrido até ao atual. Se o estado retirado for uma solução, isto é, vigia todos os retângulos, quebra o ciclo e imprime o caminho-solução. Caso contrário, se o custo do caminho for menor que o valor da geração, adiciona os novos estados. Se chegar ao fim do stack sem encontrar uma solução, avança o valor da geração.

Resultados

A-Star

O método de procura A-Star, ou A*, utiliza ambos o custo de um caminho, para procura por custo menor primeiro, e informação heurística na seleção de caminho. Para cada conjunto de caminhos, A* usa uma estimação do custo total do caminho do nó inicial ao nó destino.

Implementação

O ficheiro aStar.c implementa o algoritmo. As funções auxiliares aqui implementadas são:

- solve - executa o algoritmo.

A função solve executa o algoritmo, inicializando o StateArray e uma heap com um índice como chave e o resultado da função heurística que retorna a subestimação do custo a chegar a uma solução como valor. Enquanto a heap não está vazia, retira um índice da heap e confirma se o estado de mesmo índice vigia todos os retângulos, terminando o ciclo e imprimindo a solução, recorrendo à função printPath2, se sim. Caso contrário, adiciona os novos estados ao StateArray e os respetivos valores à heap.

Resultados

O algoritmo Astar baseia-se no método uniforme cost e usa uma heurística de subestimação do custo de chegar ao seu objetivo, para direcionar a procura. Esta subestimação é extremamente importante pois garante que obtemos soluções óptimas. Astar apresenta a segunda melhor eficiência temporal, no entanto o seu factor limitante é a sua gestão de memória.

Calculamos o tempo médio de execução de alguns casos de teste.

- G75dados10_500: 0.15 segundos
- G75dados20_500: 40.52 segundos
- G75dados30_4: 12.85 segundos. Para alguns casos fica sem memória
- G75dados40_1: Fica sem memoria suficiente.

Branch and Bound

O método de procura Branch-and-Bound combina a característica de poupança de memória do Depth-First Search e informação heurística para encontrar caminhos ótimos. O método guarda o caminho de menor custo e o seu custo, deprezando qualquer caminho de custo maior e guardado o próximo caminho cujo custo seja menor, garantido que a solução, quando encontrada, é ótimo.

Implementação

O ficheiro branchAndBound.c implementa o algoritmo. As funções auxiliares aqui implementadas são:

- solve - executa o algoritmo.

A função cria um stack onde são colocados e do qual serão retirados todos os estados em conjunto com duas listas de inteiros que contém os índices dos estados: path, onde é guardado um caminho corrente que é constantemente atualizado, e sol, o caminho hipótese solução. De seguida, retira um estado do topo do stack e atualiza o caminho até ao path com a função updatePath(). Se já se tiver encontrado um caminho-solução, obtém os valores recorrendo a bestPath(), ajusta o valor do upperBound caso seja maior que o custo do caminho percorrido e se o tamanho da solução obtida for menor que o lowerBound, termina a procura. Caso não, se o a subestimação do custo para chegar a uma solução for menor que o upperBound, adiciona estados state ao stack com pushState(). Imprime a solução quando obtida e liberta o espaço na memória dedicado às estruturas.

Resultados

O algoritmo branch and Bound usa o método DFS para fazer a expansão dos ramos de procura, no entanto temos duas grandezas, limite superior e limite inferior que nos vai restringir a expansão dos ramos. O limite inferior, caso seja atingido, garante que qualquer solução que possamos encontrar posteriormente, será pior ou igual a que correntemente temos. O limite superior indica o melhor tamanho das soluções que já encontramos e um ramo de procura só será expandido caso o seu possível comprimento, seja menor ou igual ao nosso limite superior. Usamos este limites para cortar tempo de procura. O algoritmo tem a mesma gestão espacial que o DFS e no entanto apresenta melhores resultados de execução.

Calculamos o tempo médio de execução de alguns casos de teste.

- G75dados10_500: 0.57 segundos
- G75dados20_2: 80.66 segundos

3. Métodos de Pesquisa Local

Os métodos de pesquisa local são designados a problemas de tamanho espacial muito grande. Ao não procurarem todo o espaço sistematicamente, encontram soluções num espaço mais rápido, sendo então apropriados a problemas onde já se sabe que existe, ou é muito provável existir, uma solução.

Local Search

O método de pesquisa Local Search é utilizado em problemas de procura de soluções de acordo com um critério entre um número de soluções candidatas. Experimenta, então, as diferentes soluções como resposta a mudanças locais até que solução considerada ótima é encontrada ou um limite de tempo ou memória o impede.

Implementação

O ficheiro localSearch.c implementa o algoritmo. As funções auxiliares aqui implementadas são:

- countRepeted - calcula o somatório de um estado state, que consiste na quantidade de vezes que os retângulos adjacentes ao ponto de índice state->index foram vistos.
- whichIsBest - compara dois estado e retorna 1 se o somatório do primeiro for maior que o do segundo e 0 caso contrário.
- findLocalMinimum - executa o algoritmo, dependo da solução aleatório calculada.

Define-se um array de índices de pontos seen e procura-se uma solução aleatório recorrendo à função randomSolution() que utiliza a função pickRandomPoint() para escolher um ponto do array PointMemory, gerando estados correspondente até se encontrar um estado que seja uma solução. De seguida, utiliza-se o array de pontos vistos,seen, e a solução aleatória, cur, para encontrar a solução local com a função findLocalMinimum().

Dados dois estados State, best e temp, que representam o melhor estado solução e o estado hipótese, respetivamente, para todos os pontos, procura-se se é possível desconsiderar um certo ponto no estado e o estado continuar sendo uma solução, recorrendo a feasible(), e caso possível, retira-se esse ponto do estado com takePoint(), redirecionando o novo estado para temp.. De seguida atualiza-se o state best com whichIsBest().

No final do ciclo, se o estado best for NULL então não se encontrou um estado melhor que cur e esse é retornando. Caso contrário, pode existir uma solução ainda melhor e chama-se a função findLocalMinimum() recursivamente, sendo best o novo cur.

Resultados

O algoritmo Local Search irá procurar um mínimo local para uma dada solução aleatória. Basicamente consiste em retirar pontos da solução e comparar se estamos mais perto de um mínimo local. Este algoritmo de maneira nenhuma garante soluções ótimas, no entanto é bastante útil para encontrar uma melhor solução. A qualidade do mínimo local está diretamente relacionada com a solução aleatória gerada inicialmente. Com este método tentamos gerar soluções o mais rapidamente possível em detrimento da qualidade de solução, no entanto gostava de mencionar que para grande casos de teste, a diferença entre a solução obtida e a solução mínima, compensa pois é bastante rápido determinar o mínimo local.

Calculamos o tempo médio de execução de alguns casos de teste.

- G75dados10_500: 0.01 segundos
- G75dados20_500: 0.04 segundos

- G75dados100_200: 0.85 segundos
- G75dados150_200: 2.74 segundos
- G75dados200_200: 6.49 segundos
- G75dados300_100: 10.90 segundos
- G75dados450_100: 32.13 segundos

Iterative Local Search

O método de pesquisa local iterada é um algoritmo de procura que seleciona um sucessor do estado atual que mais melhora a solução segundo uma função de avaliação. Se houverem vários possíveis sucessores, um é escolhido aleatóriamente. Deste modo, procura um ótimo local, tal que não exista nenhum possível sucessor que possa melhorar a solução.

A sua adaptabilidade à procura de um ótimo local, no entanto, pode levar ao algoritmo ficar preso entre dois mínimos sem encontrar vizinhos, sendo então difícil ao algoritmo encontrar mínimos globais.

Implementação

O ficheiro iterativeLocalSearch.c implementa o algoritmo. As funções auxiliares aqui implementadas são:

- copyCharArray - copia um array de chars para outro.
- moveAway - dado uma instância State minSol e um inteiro index, cria um novo estado com índice index e pai minSol.
- findPointToPut - encontra um ponto que ainda não foi testado e retorna o seu índice.
- ILS - executa o algoritmo, retornando o apontador do estado-solução mínima.

Tal como na função localSearch, define-se um array de índices de pontos seen e procura-se uma solução aleatório recorrendo à função randomSolution(). Utiliza-se esses valores para encontrar a solução iterative com ILS().

Define-se um array de chars pointsPushed, que representa os pontos a adicionar nas iterações, e curSeen, a lista de pontos a retirar do estado para testar como solução, e procura-se uma solução mínima inicial, minSol, com findLocalMinimum() e um ponto a testar com findPointToPut(). Enquanto o índice desse ponto for maior ou igual que 0, recorre-se a copyCharArray() para copiar os elementos de seen para curSeen, forma-se um novo estado com a solução mínima inicial e o índice atual, recorrendo-se a moveAway() e procura-se uma solução com este estado e curSeen usando findLocalMinimum().

Se o somatório do estado-solução encontrado for menor que o somatório de minSol então a nova

solução mínima é o estado encontrado. Nesse caso, copia-se os conteúdos de curSeen para seen para a próxima iteração. Caso contrário, procura-se um novo ponto para testar. No final do ciclo, retorna-se o apontador para o minSol.

Resultados

O algoritmo ILS, consiste em gerar uma solução aleatória, de seguida encontrar um mínimo local para essa solução, dar um pequeno empurrão a esse mínimo e voltar a gerar um mínimo local. Nós obviamente iremos escolher o mínimo que apresenta melhores resultados. O ILS é um pouco mais lento que o Local Search, mas garante soluções melhores ou iguais. Infelizmente continua a não obter soluções mínimas, pelo simples facto de nos fazermos a escolha consciente que preferíamos ter soluções rápidas do que soluções óptimas e lentas.

Calculamos o tempo médio de execução de alguns casos de teste.

- G75dados10_500: 0.02 segundos
- G75dados20_500: 0.06 segundos
- G75dados100_200: 0.98 segundos
- G75dados150_200: 6.68 segundos
- G75dados200_200: 10.09 segundos
- G75dados300_100: 17.44 segundos
- G75dados450_100: 36.95 segundos

4. Programação por Restrições (CLP)

4.1 Modelo Matemático

Seja A uma matriz com I linhas

- $x_{i,j} \in A$ se, e somente se, $i \in I \land j \leq \text{length}(A[i])$

Seja P um array de tamanho k

- $\forall\ p_k \in P\ \exists\ x_{i,j} \in A \mid p_k = x_{i,j}$
- $\forall\ p_k \land p_{k+1} \in P \mid p_k < p_{k+1}$
- $\forall\ i \in I, \exists\ x_{i,j} \in A\ \exists\ p_k \in P \mid x_{i,j} = p_k$
- alldifferent(P)

- $k \geq I/10 + (1 \text{ se } I\% \mathrel{!}= 0) \wedge k \leq I$

Queremos encontrar o mínimo valor que K pode tomar.

4.2 Implementação (sem módulos de programação por restrições)

Implementou-se 6 módulos distintos para responder ao problema proposto, sendo estes:

- isGoal/1 - recorre às funções auxiliares guardar/1, onde estão contidos os dados dos retângulos, e aux_isGoal/2 para obter a lista de pontos solução.
- aux_isGoal/2 - retorna a lista de pontos Points cuja interseção com a lista de pontos, List, correspondente a cada retângulos não é nula.
- sorted/1 - ordena a lista por ordem decrescente.
- generate/1 - gera uma hipótese ao problema, recorrendo a aux_generate/2
- aux_generate/2 - gera uma lista de pontos únicos e ordenados de forma decrescente com os identificadores dos pontos contidos no predicado p/3.
- solve/2 - executa o algoritmo.

Acendendo aos dados em data.ecl, solve/2 gera uma lista que terá de comprimento 1 a N, sendo N o número de retângulos na instância do problema, e generate/1 instância essa lista com vértices, sendo que a lista será retornada como true em isGoal/1 apenas se for uma solução do problema.

Resultados

4.3 Implementação em CLP

Implementou-se 2 módulos distintos para auxiliar à resposta ao problema proposto, sendo estes:

- intersect - retorna true se a interseção de duas listas não for nula.
- isSolution - retorna true se a lista de pontos Points contiver pelo menos um ponto em cada uma das linhas da matriz M.
- getMatrix - cria uma matriz a partir das listas List correspondentes a cada identificador de retângulos, rec(R,_,List).
- model1 - executa o algoritmo.

Acedendo aos dados em data.ecl, model1/2 acede à matriz com getMatrix/1, "achata-a" com flatten/2,

tornando numa única lista, e ordena-a com sort/2, e constrói uma lista Points, ordenada e de todos valores únicos de tamanho entre o nº de retângulos a dividir por 3 e o nº de retângulos a dividir por 2, valores que representam respetivamente o mínimo número de vértices suficientes para a cobertura dos retângulos e o número máximo. É depois instanciada por todas as permutações dos valores da matriz com labeling/1, encontrando-se a solução mínima e redunzindo-se as simetrias com minimize/2 utilizando-se branch_and_bound.

Resultados

5. Mínimo número de cores

Implementação

Implementou-se 3 novos módulos, extendendo-se para a resolução dos outros passos, o uso daqueles implementados anteriormente, sendo os novos:

- restrictColor - restringe a lista Color a cores correspondentes a vértices que estão nos retângulos, sendo as cores todas diferentes no mesmo retângulo.
- getColor - retorna a lista de cores.
- model1 - executa o algoritmo.

Obtém a matriz com getMatrix/1, "achata-a" com flatten/2, tornando numa única lista, e ordena-a com sort/2. De seguida, constrói uma lista Points e uma lista Color de tamanhos iguais, sendo o domínio de Points a matriz modificada e o de Color todos os valores entre 1 e N, sendo N o número de retângulos. Assim, cada ponto na lista Points tem uma cor correspondente de mesmo índice na lista Color. Adiciona-se a restrição a Points de ser uma lista ordenada por ordem de crescente com ordered/2, instância-se com labeling/1 e restringe-se às soluções da matriz com isSolution/2. De seguida, restringe-se as cores com restrictColor/3 que utiliza getColor/4 e calcula-se a soma dos elementos da lista Color com sumlist/2 que irá restringir o número de iterações de Color instanciadas com labeling/1. Por fim, minimiza-se as simetrias com minimize/2.

Resultados

Conclusão

A aplicação dos diferentes métodos de pesquisa ao problema da vigilância de partição retangulares retornou resultados muito diferentes. Para encontrar soluções rápidas é necessário abdicar soluções ótimas, como na aplicação de métodos greedy, ou utilizar instâncias de problemas menores de modo a evitar limitações de espaço de memória, ocorrente em método de pesquisa não-informadas.

Por outro lado, métodos de pesquisa local são mais eficientes em procura de mínimos locais, tendo a desvantagem de poder ficar limitados a um mínimo local e nunca conseguir encontrar a solução ótima global.

A implementação em prolog sem restrições encontrou problemas semelhantes a métodos de procura não-informados, sendo o seu maior limite a rápida expansão do número de estados necessários a analisar de modo a encontrar uma resposta. No entanto, comprovou-se como a programação lógica com restrições traduz um algoritmo mais bem elaborado que, embora ainda com os seus defeitos,

produz resultados melhores.

A elaboração da resposta às mínimas cores produziu soluções óptimas a custo de soluções rápidas, consolidando os conhecimentos de Constraint Logic Programming.

No desenvolvimento do trabalho encontrou-se algumas dificuldades em implementar algoritmos como o BFS, que encontram limites de memória rapidamente, sendo aplicáveis apenas a instâncias do problema curtas.