Inteligência Artificial

Resolução de Problemas de Otimização

Otimização de Corte de Placas de madeira/vidro

2017/2018

Universidade do Porto

Faculdade de Engenharia

FEUP

Daniel Carvalho up201405902@fe.up.pt

Paulo Correia up201406006@fe.up.pt

Tiago Neves up201406104@fe.up.pt

19 de Maio de 2018

Índice

Objectivo

Os problemas de corte são problemas facilmente formulados. Estes surgem quando as empresas necessitam de se adaptar aos pedidos de clientes que não são constantes entre si, pedindo diferentes dimensões do material, o que colide com a produção de produtos em série. Como a produção em série é capaz de reduzir os preços de produção substancialmente, é mais rentável obter os pedidos do cliente a partir de um bloco que pode ser produzido em série.

Um problema de corte é então a minimização do desperdício de material obtido pelo corte de um bloco de grandes dimensões em blocos de dimensões pretendidas, necessariamente menores.

No entanto estes problemas aparentemente simples escondem a sua real complexidade. Trata-se de problemas do tipo NP-Hard, o que justifica o estudo destes problemas com métodos alternativos de resolução que permitam obter soluções aproximadas ou exatas numa escala temporal razoável.

É objetivo deste trabalho estudar métodos que nos permitam resolver estes problemas computacionalmente complexos.

Especificação

Para dividirmos as placas eficientemente é necessário transformar o problema complexo num problema que possa possa ser resolvido com algoritmos de otimização. Esses irão calcular a posição das peças no bloco para que o desperdício seja menor.

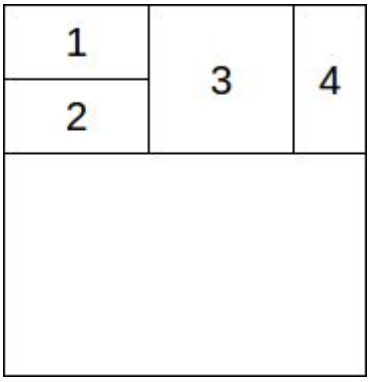

Figura 1 - Exemplo da organização de um bloco em blocos menores eficientemente

Representação

Para nos auxiliar na execução destes algoritmos resolvemos criar uma grelha que nos permite representar uma solução. A grelha permite-nos representar o bloco a partir do qual vamos cortar os objectos pretendidos. A grelha é uma matriz de largura e altura definida pelo utilizador, na qual os objetos são colocados por ordem.

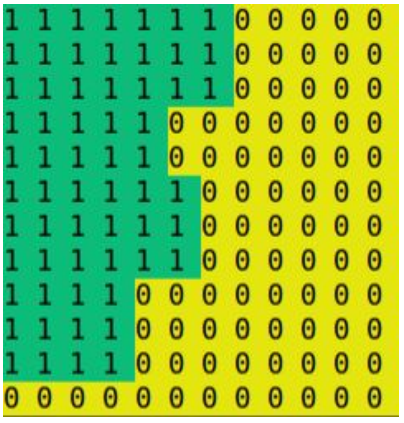

Figura 3 - Exemplo de uma grelha 10 x 10

A representação de um estado é feita por uma lista ordenada que contém os objetos. A ordem por qual os objetos estão presentes na lista é a ordem pela qual os objetos vão ser colocados na grelha. Dando como exemplo a figura 4, a lista seria [1, 2, 3, 4, 5, 6, 7]. O que os algoritmos tentam otimizar é a ordenação da lista.

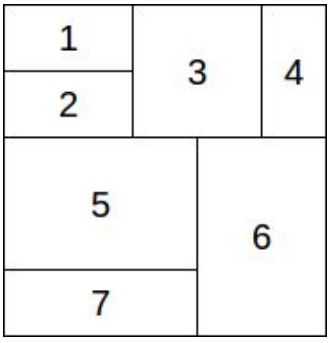

Figura 4 - Objectos numa grelha

Pontos “pivot”

Para a colocação dos objetos na grelha é necessário definir o seu ponto “pivot” que vai corresponder ao canto superior esquerdo do objeto depois de colocado. É calculada depois para cada representação uma lista de pontos que contém todos os sítios onde menos prejudicial seria colocar os pivots dos objetos. Neste caso, interessa-nos os pontos adjacentes aos objetos já colocados. No entanto, conseguimos eliminar parte desses pontos pelo facto de conseguirmos concluir que na mesma coluna o pivot do topo é melhor que todos os outros e na mesma linha o ponto mais à esquerda é melhor que todos os outros. A única excepção é quando o pivot possui na sua linha outros objetos. Quando isso isso acontece devemos explorar uma situação em que a linha esteja completamente livre.

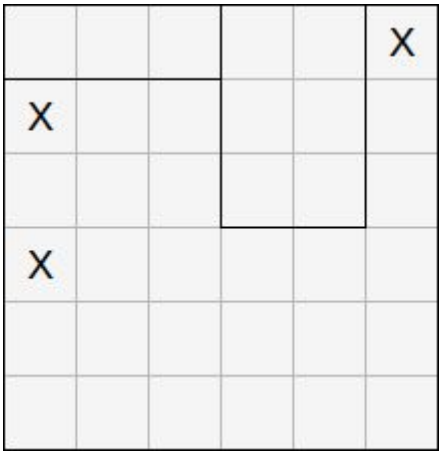

Figura 5 - Exemplo de pontos para colocar pivots (numa situação apenas ilustrativa)

Heurística

Para determinarmos quanto um estado é melhor que o outro é necessário fazer uma heurística. No nosso caso, considerando que queremos diminuir o desperdício, e considerando o desperdício é a área do menor retângulo que engloba todos os cortes. Isto faz com que o nosso desperdício forme sempre um rectângulo, o qual a sua área vai ser o principal interveniente da nossa heurística.

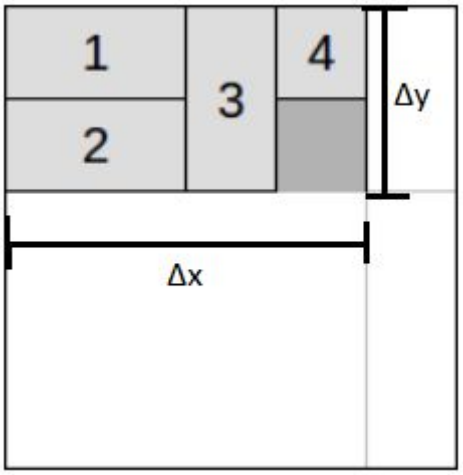

Figura 6 - Desperdício marcado a cinza. Calculado através de Δx * Δy + Δx + Δy

Outro fator que tomamos em conta no cálculo da heurística é a existência de buracos no meio de um bloco de objetos. Isto é, preferimos soluções que sejam um bloco sólido invés de uma solução que tenham desperdício no meio deste, pois geralmente este desperdício resulta em blocos que não podem ser posteriormente reutilizados.

Abordagem

Os métodos utilizados na abordagem ao problema foram o algoritmo “subir a colina”, “arrefecimento simulado” e o algoritmo genético. Que vão ser melhor detalhados abaixo.

● Subir a colina

O **“Subir a colina”** é um algoritmo que parte de uma solução inicial aleatória, gera e testa soluções possíveis de variação, chamados sucessores, e escolhe um dos sucessores para continuar. Distinguem-se duas variações deste método:

- “Subir a colina” básico;
- “Ascenção íngreme”;

No “subir a colina” básico assim que é encontrado um descendente melhor que o atual o algoritmo segue esse descendente.

Neste trabalho foi utilizado o segundo método, “Ascensão íngreme”. Partindo do estado inicial, para cada estado são calculados todos os seus sucessores, dos quais é escolhido aquele com melhor valor de heurística, este sucessor é comparado ao estado actual. Se for melhor este sucessor torna-se o estado actual e o processo repete-se, caso contrário o algoritmo é parado, e o estado final é retornado como a solução final.

No contexto do nosso problema aplicamos este algoritmo de duas maneiras, numa primeira maneira todos os objetos eram colocados na grelha de aleatoriamente, e os sucessores de um estado eram o resultado de mover cada objecto para uma posição adjacente ou rodar o mesmo, este método provou-se bastante ineficaz e foi abandonado pela especificação apresentada acima. A segunda maneira já usa tal especificação e cada sucessor é a troca de dois elementos na lista de itens.

Uma das principais fraquezas deste algoritmo é a o facto de parar em máximos locais e não na solução global ótima do problema.

● Arrefecimento Simulado

O **"Arrefecimento Simulado"** é um algoritmo usado para encontrar o máximo global de um função. Primeiro é inicializada uma temperatura inicial igual à área da grelha onde os objectos vão ser colocados. Para cada estado o algoritmo gera um sucessor aleatório como explicado anteriormente e vê a variação da heurística entre o novo sucessor e o estado atual. Se o novo sucessor for melhor que o estado atual, este passa a ser o próximo estado, caso isto não aconteça há uma probabilidade de $e^{(\Delta h/T)}$ onde T é a temperatura atual. A temperatura é decrementada todas as iterações e a solução final e o estado actual de quando a temperatura chega a 0. Esta abordagem é semelhante à anterior só que menos propícia a parar em máximos locais.

● Algoritmo Genético

O algoritmo genético inspira-se na evolução biológica descrita por Darwin. Simula a evolução das espécies na natureza através dos princípios da variabilidade genética, diferentes soluções, e da seleção natural, escolha das melhores soluções de acordo com a sua adaptabilidade ao meio.

Para a variabilidade genética contribuem o cruzamento entre diferentes indivíduos e a mutação dos genes dos indivíduos. Para a seleção natural é necessário quantificar o quão ideal é uma solução. Para isso utiliza-se uma heurística que nos permite atribuir um valor a cada indivíduo para depois poder ao longo do algoritmo ir selecionando os melhores com uma razão de probabilidade correspondente à sua adaptabilidade ao meio.

Adaptação

Para o cálculo da adaptabilidade utilizamos a heurística que já foi definida atrás. Quanto menor a adaptação, mais apto é o indivíduo.

Seleção

Posteriormente, com vista à seleção dos indivíduos, ordenou-se a população em função da sua adaptação. Utilizou-se uma estratégia elitista em que os melhores resultados sobrevivem inalterados. Todos os outros têm uma probabilidade de serem eliminados.

Cruzamento

No cruzamento os passos utilizados foram: seleção de dois indivíduos, geração de um ponto de crossover, cópia de a lista ordenada do primeiro pai os primeiros objetos até ao crossover point e depois copia do outro os objetos que faltam pela ordem que o segundo pai os tem.

Exemplo:
Pai A = [A B C D E F]
Pai B = [F E C D B A]
Filho com crossover point em 3 = [A B C F E D]

Mutação

A mutação é feita trocando as posições de dois objetos entre si na lista ordenada.

Exemplo:
Indivíduo = [A B C D E F]
Indivíduo mutado = [A E C D B F]

Critério de paragem

Para parar o algoritmo o método utilizado foi a espera pela estabilização do algoritmo, isto é, quando a heurística permanece igual durante um determinado número de gerações, o algoritmo considera-se estabilizado e termina-se a sua execução. Outro critério de paragem possível, mas não implementado, é parar o algoritmo quando a sua heurística atinge o valor mínimo possível. No entanto como nem sempre é possível calcular a sua heurística mínima não utilizamos este método.

Desenvolvimento

O desenvolvimento de todos os algoritmos descritos foi feito em Python 3.6. O programa foi desenvolvido de raiz com base nos conhecimentos adquiridos na unidade curricular. Utilizamos o módulo *"term"* para fazer a interface gráfica no terminal. O ambiente de desenvolvimento utilizado foi o Microsoft Visual Studio Code.

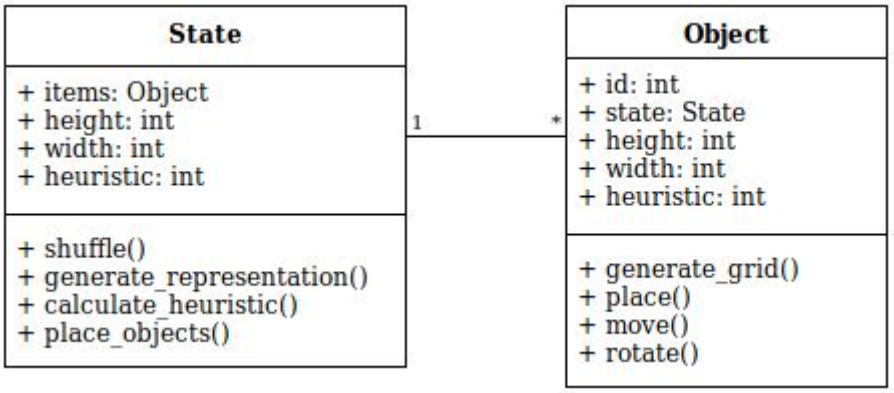

Figura 7 - Diagrama de classes

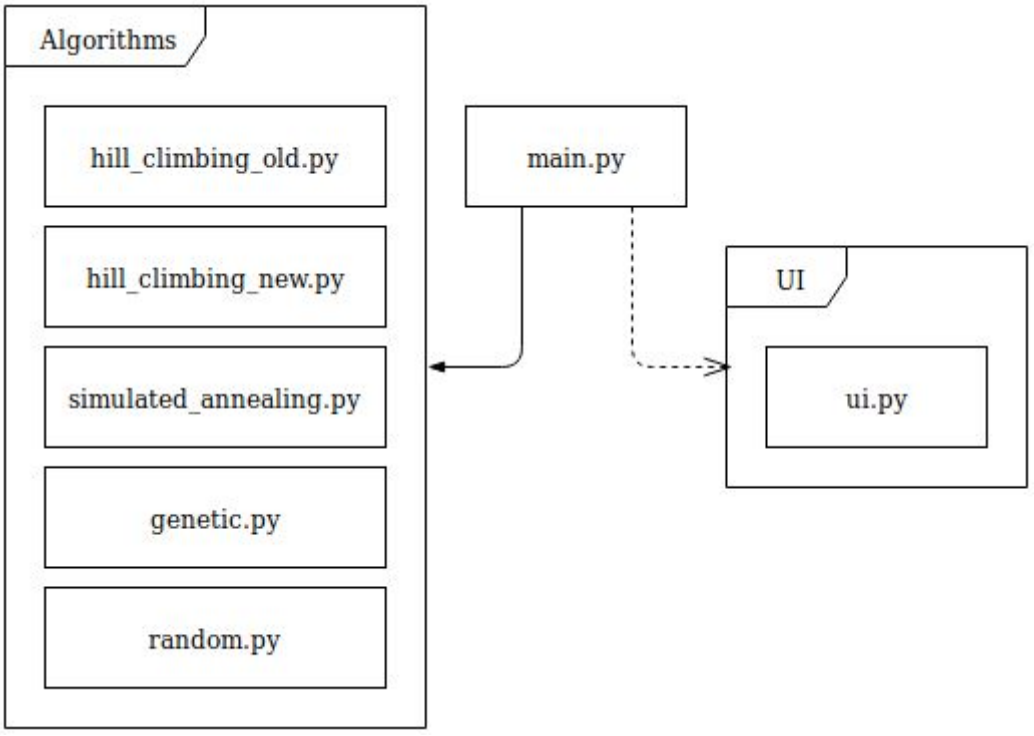

Figura 8 - Estrutura da aplicação

Experiências

Para compararmos os diferentes métodos criamos um ambiente de simulação que nos permite obter o tempo despendido para chegar a uma solução em cada um dos algoritmos. É também relevante para a comparação entre os métodos saber o quão facilmente o algoritmo chega à solução ótima.

Uma das situações de teste é o ficheiro “fill12”, incluído com o programa. Este dataset tenta colocar 18 objetos numa grelha 12x12. Encontra-se numa situação em que toda a grelha é preenchida por objetos, ou seja, o desperdício ideal é nulo. Pode nesta situação acontecer a obtenção de uma solução impossível e em que o algoritmo teria que ser reiniciado para obter uma nova solução.

A outra situação de teste feita é o “mem10”, também incluído com o programa. É um dataset muito mais leve que apenas tenta colocar 8 objectos numa grelha 10x10. Neste caso a grelha não é totalmente cheia, pelo que se espera que nunca existam soluções impossíveis. Este teste é muito menos pesado computacionalmente que o anterior e pretende testar o desempenho numa situação em que obter a solução é fácil.

Para verificarmos o comportamento das nossas soluções com a complexidade do problema, criamos uma solução “random” em que apenas ordena a lista de forma aleatória e testa a sua solução 150 vezes.

Para cada algoritmo fizeram-se 20 testes com ambos os ficheiros. Os resultados estão explícitos na tabela abaixo.

Algoritmo	Subir a colina (diferente heurística)	Subir a colina	Arrefecimento Simulado	Algoritmo Genético	Geração de soluções random
Tempo Médio de Convergência para a solução	11.08	12.05	3.52	17.66	3.37
Número de soluções ótimas	0/20	15/20	13/20	17/20	6/20
Número de soluções possíveis	0/20	15/20	13/20	17/20	6/20

Tabela 1 - Teste com ficheiro “fill12”

Algoritmo	Subir a colina (diferente heurística)	Subir a colina	Arrefecimento Simulado	Algoritmo Genético	Geração de soluções random
Tempo Médio de Convergência para a solução	0.72	0.38	0.57	2.60	0.72
Número de soluções ótimas	0/20	18/20	9/20	20/20	20/20
Número de soluções possíveis	0/20	20/20	20/20	20/20	20/20

Tabela 2 - Teste com ficheiro “mem10”

Resultados

Baseado na estatística obtida conseguimos observar que o algoritmo genético é o que nos permite mais fiável e consistentemente obter resultados ótimos e possíveis. Apesar de quando o problema é computacionalmente fácil o “subir a colina” e o random obterem resultados igualmente bons, quando o problema cresce de complexidade o algoritmo genético demarca-se dos outros e demonstra melhores resultados quando se pretende obter uma solução.

Essa melhoria em termos de obtenção de solução não se verifica no comportamento temporal. O algoritmo genético cresce temporalmente muito mais rápido que o arrefecimento simulado. Na mudança do teste “mem10” para o teste “fill10” é possível verificar que o arrefecimento simulado demora apenas 6x mais. Isto faz com que consiga claramente derrotar todos os outros algoritmos em termos de tempo necessário para a execução com o crescimento da complexidade do problema.

No entanto todos os valores de tempo medido estão a tomar em conta o tempo desde que uma solução foi encontrada até que o algoritmo pare de executar. No teste “fill12”, isto significa mais 1 segundo em média no algoritmo “arrefecimento simulado” e em média mais 10 segundos no algoritmo genético! Em casos em que é possível calcular a heurística final, como por exemplo quando toda a grelha é preenchida por objetos, os algoritmos tornam-se mais eficientes temporalmente se terminarmos o algoritmo assim que a heurística atinge o seu máximo absoluto. No entanto, como nem sempre é possível calcular a heurística mínima e para não influenciar as conclusões dos resultados obtidos, todos os testes continuam com os seus critérios de paragem originais em ambos os testes.

Outra conclusão final a retirar é que os nossos processos de heurística e colocação das peças na grelha, que é comum a todos os algoritmos, influencia positivamente os nossos resultados. Aquele que no início era para nós um teste de controlo, o algoritmo “random”, acaba por ter excelentes resultados nas situações mais fáceis de computar. Isto deve-se ao facto de a nossa maneira de colocar as peças no tabuleiro já por si otimizar imenso o processo de convergência para uma solução e transformar um problema complexo, colocação de peças num bloco, num problema mais básico, ordenação de uma lista de forma a que se minimize o desperdício.

Melhoramentos

Embora o trabalho realizado apenas tenha abrangido dois algoritmos e as suas variações, este é um problema que é possível de se resolver com outros métodos. Por exemplo, caso tenhamos exemplos suficientes, podemos treinar uma rede neuronal que, depois de treinada, iria obter um resultado de forma mais rápida que todos os outros algoritmos. Poderíamos inclusive criar um dataset a partir dos resultados dos algoritmos que possuímos e criar uma aproximação dos algoritmos que seria de mais rápida execução.

As possibilidades não se ficam por aí e podemos encontrar outros algoritmos, como por exemplo, o GRASP.

Recursos

Bibliografia

> Slides da cadeira de Iart sobre métodos de resolução de problemas:
- https://web.fe.up.pt/~eol/IA/1718/APONTAMENTOS/2MRPeAG.pdf

Elementos do grupo

- Daniel Carvalho: 33,33%
- Paulo Correia: 33,33%
- Tiago Neves: 33,33%

Manual de utilizador

A aplicação está disponível em duas variantes, uma versão para windows e uma versão para linux, ambas são corridas com o python3.

Para correr a versão do linux é necessário primeiro instalar uma dependência com o seguinte comando `pip install py-term`.

A versão para linux é corrido com o comando `python3 main.py` e a versão do windows com o comando `python3 main_windows.py`

Estes comandos podem ser corridos com 2 argumentos, um primeiro argumento é o nome do ficheiro para ler data e o segundo argumento o id do algoritmo a ser usado. Caso estes argumentos não forem proporcionados o utilizador é solicitado por estas informações.