FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Relatório do Projeto de Programação (2ª fase)

Knapsack

Desenho de Algoritmos para Problemas de Otimização
Prof. Pedro Barahona
2018/2019

Filipe Medeiros nº47967
Miguel Carrega nº47830

Índice

Introdução

Neste projeto foi feita a continuação da análise do problema de otimização Knapsack 0/1. Nesta fase foi utilizado o método de procura local, para o qual implementamos duas meta-heurísticas: Simulated Annealing e Ant Colony. Para testar estes algoritmos, foram usados 4 ficheiros de inputs, cada um com diferentes características (tamanho do conjunto, valores máximos, etc), de forma a obter resultados distintos.

O relatório contém uma explicação do problema, seguido de uma descrição dos algoritmos (modelo, soluções iniciais e meta-heurísticas). Também serão apresentados alguns testes, bem como a análise dos mesmos e a comparação entres estes e os algoritmos de aproximação implementados na primeira fase do projeto.

Por fim, serão explicadas as conclusões retiradas ao desenvolver o projeto e ao analisar os testes executados.

Problema (Knapsack)

Knapsack (ou MAX-Knapsack): Dado *n* objetos (*i* = 1,…,*n)*, cada um com um peso w_i e valor v_i associado (números inteiros positivos), e um saco com capacidade *C* (inteiro positivo), encontrar um subconjunto de objetos com o máximo valor cujo peso total não exceda *C*. (Assumir que $w_i \leq C$, para cada *i* = 1,…,n). Pode ser escrito como:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0,1\}.$$

onde x_i representa se o objeto *i* foi escolhido ou não.

Este problema de otimização é NP-Hard, a sua resolução é pelo menos tão difícil quanto o seu problema de decisão (pode um valor de pelo menos *V* ser atingido sem exceder a capacidade *C)*, que é NP-Complete. Não existe um algoritmo polinomial que, dada uma solução, verificar se esta é a solução ótima.

Algoritmos

Os algoritmos foram implementados sobre a mesma base, na qual existem objetos do tipo *Item* que contém o valor e o peso do mesmo. Os *Itens* são recebidos como input no formato de um ficheiro de texto. Ambos os algoritmos foram desenvolvidos em Java.

Simulated Annealing:

O algoritmo de simulated annealing tem como objetivo explorar os vizinhos de uma solução, de forma a encontrar uma solução global ótima. Ao selecionar uma solução vizinha, o algoritmo aceita esta caso seja melhor que a solução atual. Caso contrário irá aceita a solução com uma certa probabilidade. Com o número de iterações, esta probabilidade vai decrementar até um dado limite, no qual a melhor solução descoberta será o resultado.

Para executar o algoritmo são necessários vários parâmetros:

- Temperatura inicial
- Temperatura final
- Fator de arrefecimento
- Número de vizinhos

O algoritmo começa com uma temperatura inicial. Quando é descoberta uma nova solução, a temperatura é reduzida, multiplicando pelo fator de arrefecimento. Soluções são procuradas até a temperatura chegar ao valor da temperatura final. O número de vizinhos será o número de soluções vizinhas à solução atual, soluções que serão avaliadas.

O algoritmo é executado da seguinte forma:

- Começa numa solução vazia (existe opção de começar com uma solução aleatória ou o resultado de um algoritmo greedy).
- Enquanto a temperatura for superior à temperatura final:
    - Calcula soluções de *n* vizinhos e avalia cada solução; caso seja melhor, esta é atualizada para a solução atual; caso contrário aplica a expressão $e^{-delta/temp}$ para calcular a probabilidade de aceitar esta solução (*delta = bestSol – currentSo, temp = temperatura atual).*
    - Arrefece a temperatura usando o fator de arrefecimento
- Devolve a melhor solução encontrada

Ant Colony:

Neste algoritmo, um conjunto de formigas tenta trabalhar em conjunto para encontrar a solução ótima do problema. Em geral, o algoritmo utiliza a seguinte lógica. Em cada iteração, as formigas começam com uma solução vazia e com uma vizinhança igual ao conjunto total de itens, e, segundo uma função probabilística (pseudo-aleatória) vão escolhendo um novo item para acrescentar à solução. Cada vez que um item é adicionado à solução, uma nova vizinhança, retirando o item acabado de escolher, e qualquer item que já não caiba na mala. Ao fim de alguns passos, as formigas descobrem a sua solução. Serialmente, todas as formigas encontram uma solução. Depois de todas as formigas o fazerem, cada uma delas irá depositar uma certa quantidade de feromonas em todos os itens que a sua solução inclui. Devido à função que determina a probabilidade de uma formiga escolher um item ter em conta a quantidade de feromona nele presente, ao longo das iterações, as formigas têm tendência a escolher mais itens que anteriormente outras formigas escolheram e designaram como bons.

Há vários parâmetros e funções que podem ser alteradas e que irão afetar a performance do algoritmo, tanto em tempo de execução como na sua eficácia (proximidade ao valor ótimo). A lista destes parâmetros e funções segue-se:

- α - é utilizado na função de cálculo da probabilidade de escolha de um item, e representa a importância que queremos dar às feromonas
- β - é utilizado na função de cálculo da probabilidade de escolha de um item, e representa a importância que queremos dar à atratividade de um item
- ρ - é utilizado na função de evaporação e representa o ritmo ao qual queremos “esquecer” valores calculados pelas formigas, e é apenas multiplicado pelo valor de feromonas do item após cada iteração
- τ_i - representa a quantidade de feromona que o item *i* tem neste momento
- τ_{max} - representa a quantidade máxima de feromona que um item pode ter
- τ_{min} - representa a quantidade mínima de feromona que um item pode ter

- μ_i - representa a atratividade do item *i* e é calculado pela fórmula $\frac{z}{w^2}$, em que *z* é o valor do item e *w* é o seu peso
- p_i - representa a probabilidade de uma formiga escolher o item *i*, quando este está na sua vizinhança, e é dada por $\frac{\tau_i^{\alpha} \mu_i^{\beta}}{\sum_j \tau_j^{\alpha} \mu_j^{\beta}}$, em que o somatório do denominador simboliza toda a vizinhança corrente
- $\Delta\tau$ - representa a variação de feromonas em cada iteração, ou seja a quantidade de feromonas que uma formiga irá depositar em todos os itens da sua solução, e é dada por $\frac{1}{1 + \frac{z_{best} - z_s}{z_{best}}}$, em que z_{best} é o valor da melhor solução encontrada até ao momento, e z_s é o valor da solução onde se vai colocar feromonas.

Outros aspetos a notar sobre a implementação são que foi implementado um sistema de Threads no Java, para que, na mesma iteração, várias formigas pudessem encontrar a sua solução em paralelo, e que havia outras maneiras de implementar o algoritmo, por exemplo fazer com que apenas a melhor formiga deposite feromonas, ou fazer com que as formigas depositam feromonas assim que encontram a solução, dentro da iteração. Chegamos à conclusão que este conjunto de "definições" escolhidas era o que trazia melhor resultados, apesar de nunca termos obtido o valor ótimo nos testes (a não ser no teste data_1).

Testes

Para testar estes algoritmos foram usados como input ficheiros de texto no seguinte formato: primeira linha contém o número total de objetos (*n*) e a capacidade do saco (*C*), as linhas seguintes contêm o peso (w_i) e o valor de cada objeto (v_i), um por cada linha.

Foram usados 4 ficheiros de input, todos utilizados para testar dois algoritmos implementados na 2ª fase, assim como ambos os algoritmos desenvolvidos na 1ªfase. Os testes foram gerados automaticamente, usando um gerador de testes aleatórios (com parametrizações) desenvolvido por nós.

Os testes gerados têm as seguintes informações:

	nº items	*capacity*	*max value*	*max weight*	*opt value*	*opt weight*
data_1	100	5.000	1.000	1.000	2.650	4.985
data_2	500	11.000	300	4.000	9.200	10.997
data_3	500	10.000	600	3.000	21.461	9.987
data_4	500	15.000	500	30.000	14.272	29.978

Para Simulated Annealing executamos cada ficheiro de input usando 3 soluções iniciais distintas: solução a zeros (array de bytes[] da solução inicial começa a zero); solução random (array de bytes[] da solução inicial começa a 0/1, com 50% de probabilidade); solução greedy (array de bytes[] da solução inicial começa com a solução do algoritmo greedy implementado na primeira fase).

Para o Ant Colony, apenas foi executado o teste standard (começar com todas as soluções a zeros) para cada um dos ficheiros de teste.

Resultados dos testes

Todos os testes dos algoritmos implementados na 2ªfase do projeto foram executados 100 vezes. Os resultados apresentados são valores médios de cada iteração.

Parâmetros usados para a meta-heurística Simulated Annealing:

- Fator de arrefecimento: 0.99
- Temperatura inicial: 100.0
- Temperatura final: 0.2
- Número de vizinhos: 100

Parâmetros usados para a meta-heurística Ant Colony:

- Número de formigas: 15
- Número de iterações: 20
- α: 2.0
- $\boldsymbol{\beta}$: 4.0
- ρ: 0.95
- $\boldsymbol{\tau}_{max}$: 10.0
- $\boldsymbol{\tau}_{min}$: 0.1

data_1	calculated weight	calculated value	time taken (ms)
Greedy	4.981	2.638	1
FPTAS	4.985	2.650	60
SA-0's	4.967	2.548	34
SA-R	4.959	2.560	34
SA-G	4.981	2.638	30
AC	4.985	2.650	169
Optimum	4.985	2.650	-

data_2	calculated weight	calculated value	time taken (ms)
Greedy	10.975	9.148	8
FPTAS	10.997	9.200	710
SA-0's	10.971	6.612	82
SA-R	10.973	6.087	79
SA-G	10.974	9.148	81
AC	10.984	9.018	
Optimum	10.997	9.200	-

data_3	calculated weight	calculated value	time taken (ms)
Greedy	9.999	21.461	7
FPTAS	9.987	21.461	707
SA-0's	9.985	11.639	80
SA-R	9.987	10.280	82
SA-G	9.999	21.461	84
AC	10.000	20.883	905
Optimum	9.987	21.461	-

data_4	calculated weight	calculated value	time taken
Greedy	29.864	14.250	12
FPTAS	29.978	14.272	722
SA-0's	29.932	7.650	81
SA-R	29.930	6.330	87
SA-G	29.864	14.250	90
AC	14.955	9.803	406
Optimum	29.978	14.272	-

Os resultados respetivos a Simulated Annealing dependeram muito da solução inicial. Podemos observar que começar com uma solução inicial vazia é preferível a ter uma solução aleatória. Isto deve-se provavelmente ao facto de estarmos a "enganar" o algoritmo com uma solução que pode não ter qualquer relação com a solução ótima. Para além disso, observamos que, ao disponibilizar o resultado da execução do algoritmo greedy como solução inicial, a solução encontrada irá ser igual. Podemos então concluir que não deveremos esperar resultados diferentes quando usamos este tipo de solução inicial.

Para além disto, também foram estudados os diferentes parâmetros usados no algoritmo. O fator de arrefecimento, pertencente a]0,1[, define a velocidade a que a temperatura vai arrefecer, sendo que, quanto menor, mais rápido é o decréscimo. Isto, juntamente com a temperatura inicial e a temperatura, irão definir o número de iterações que o algoritmo executa antes de atingir a temperatura mínima. O número de vizinhos irá decidir o número de soluções exploradas que foram baseadas na solução atual. Quanto maior o número de vizinhos, maior será a dispersão da procura. Isto influencia a proximidade da solução atual, sendo que um pequeno número de vizinhos pode levar a uma solução ótima global, ao invés de global. Por outro lado, um elevado número de vizinhos leva uma procura muito alargada, demorando um tempo de execução mais elevado.

Quanto aos resultados do algoritmo Ant Colony, podemos observar primeiro que os tempos de execução são muito superiores a quase todos os outros algoritmos, e que os resultados nem sempre são melhores. No caso específico do teste data_4, é muito provável que algures na implementação haja um pequeno erro (ou mau uso de tipos de dados, como *int* onde devia ser usado o *long*) que causa o péssimo resultado. Depois de muitas tentativas, o resultado 9.803 foi sempre o melhor, sem variação, mas não conseguimos identificar a causa. De resto, os resultados são bastante satisfatórios.

No geral, é possível observar que, para os testes realizados, os dois algoritmos de otimização estudados na primeira fase são bastante melhores na sua eficiência (resultado / sobre tempo demorado) que todos os algoritmos de procura local, exceto o SA-G. No entanto, este tem uma certa vantagem pois já começa com uma solução extremamente boa.

Apesar disto, podemos dizer que um grande grau de confiança que isto se deve aos testes realizados, que, apesar de serem aleatórios, não representam todos os tipos de dados que se encontram no mundo real, e há muitas situações, especialmente para *data sets* muito grandes, em que o algoritmo Greedy se torna pouco eficaz, e o FTPAS pouco eficiente no tempo de execução. Nestes casos, achamos que algoritmos de procura local, com ou sem ajuda dos anteriores, podem obter resultados muito mais satisfatórios.

Conclusão

Ao desenvolver este projeto pudemos concluir que os diferentes algoritmos têm diferentes propósitos, dependendo do tipo de execução que pretendemos. Apesar de nos nossos testes só termos uma média de resultados dos algoritmos de procura local, a ideia seria guardar os resultados da melhor solução calculada num dado número de iterações. No entanto, desta forma conseguimos comparar melhor o resultado esperado ao executar os algoritmos, nomeadamente comparar a diferença entre diferentes inputs (no caso de simulated annealing).

Apesar de tudo, tivemos dificuldades ao tentar implementar os algoritmos em Comet devido à falta de informação disponível online, mais especificamente ao procurar erros de compilação. Decidimos então recorrer a Java por ser uma linguagem mais conhecida por nós. No entanto sabemos que se o projeto fosse feito em Comet, seria mais simples, curto e otimizado, mas aproveitamos esta oportunidade para elevar ainda mais as nossas capacidades de implementar algoritmos de raíz em Java, tendo cuidado com todas as otimizações possíveis.