Pontifícia Universidade Católica de Goiás
Escola Politécnica e de Artes
Ciência da Computação
Time Spectree
Alunos: Allann Barbosa Cintra, Danilo Matias Barcelos

2º Desafio em Otimização com Meta-heurística
Problema do Empacotamento – Solução com Recozimento Simulado

Tópicos:

Introdução

Este documento é a entrega única do nosso grupo, ele contém todas as informações requisitadas para a submissão do desafio.

Ambiente de Teste & Desenvolvimento

Os computadores pessoais dos dois integrantes foram utilizados em todos os processos deste desafio.

O computador do integrante Danilo Barcelos é um computador de mesa sem modelo, possui o processador Ryzen 5 3400G, a placa-mãe ASUS A320M, e 12 GB de memória RAM com 2666 MHz de velocidade. Não possui GPU dedicada. Usa o sistema operacional Windows 10, e o IDE Visual Studio Code.

O integrante Allann Cintra possui um Acer Aspire Z1-752. Ele usa o sistema operacional Linux, com a distribuição Gentoo, e programa na IDE NeoVim.

Instâncias Utilizadas

No processo de teste para a obtenção dos resultados na seção 2.4, foram utilizados os 6195 casos de teste disponíveis no site da UNIBO, além das soluções ótimas contidas no mesmo.

Meta-heurística Utilizada

Descrição

A Meta-heurística escolhida foi a chamada Recozimento Simulado (em inglês, Simulated Annealing), um algoritmo que tem base teórica no processo de recozimento no contexto da metalurgia. É uma técnica que consiste em aquecer um metal acima da sua zona crítica (temperatura onde ocorre a transição do estado sólido para o líquido), e depois é resfriado lentamente para atingir a forma desejada. A lógica deste processo é bem simples: um metal em baixas temperaturas não é maleável, e não muda de forma facilmente, enquanto um metal muito quente não consegue manter sua forma. Portanto, para que se mude a forma do metal para algo desejado, ele deve transitar lentamente da sua forma instável para a melhor possível, de maneira controlada.

O Recozimento Simulado trata a resposta para um problema computacional como o metal do recozimento real, ele existe para diminuir a diferença entre uma resposta qualquer para o problema e a resposta ótima, como o ferro lentamente tomando a forma de uma espada. O algoritmo consegue isso aplicando pequenas alterações aleatórias em uma resposta inicial, e toda variação da resposta atual é considerada, e será escolhida se for melhor que a atual, ou se a "temperatura" estiver alta o suficiente. Cada iteração do algoritmo diminui a temperatura. A forma de obtenção da resposta inicial não é relevante.

Esta Meta-heurística utiliza o conceito de temperatura para retirar a resposta atual de um "máximo local", que é uma resposta excelente se forem consideradas apenas as vizinhas, mas não é a melhor se for considerar todas as respostas possíveis. Isso é possível pois maiores temperaturas fazem com que a resposta mude drasticamente, procurando grandes partes do espaço amostral de forma aleatória, e enquanto a temperatura diminui, espaços mais curtos são considerados, obtendo assim uma resposta próxima da ótima.

O algoritmo se encerra quando a temperatura atinge um ponto mínimo, onde se pode garantir com uma quantidade razoável de certeza que não serão encontradas soluções melhores.

Implementação e Escolhas

A implementação da heurística foi feita na linguagem de programação C++, e foi separada em 5 arquivos de código-fonte:

- "random.cpp" e "random.hpp" definem os métodos de geração de números aleatórios uniformes usados na heurística. Eles usam a implementação pré-feita do C++ do algoritmo Mersenne Twister chamada "mt19937_64" e não serão discutidos.
- "sa.hpp" e "sa.cpp" definem a classe que define uma caixa, com todos os seus métodos e itens, a classe que define uma solução do problema, e o algoritmo em si. Esta é a parte mais importante do código, e contém toda a heurística.
- "main.cpp" apenas lê da entrada padrão o número de itens, a capacidade de cada caixa, e para todo item lê o seu tamanho. Então ele chama a classe da solução definida em "sa" e imprime a resposta que ela calcula na saída padrão.

A razão para ser feito dessa maneira é que o executável resultante é extremamente flexível. Pode-se usá-lo executando diretamente, escrevendo um caso de teste, e vendo a saída; mas também é possível realizar testes automatizados através de scripts.

A classe da caixa contém uma lista de itens, e a somatória dos pesos de todos os itens atuais, Além de métodos para tirar e colocar itens, trocar itens com outras caixas e imprimir seu conteúdo.

A classe da solução contém uma permutação dos itens, que é interpretado como a ordem em que eles serão inseridos nas caixas, da esquerda para a direita. Contém também o método para gerar a primeira solução, gerar um vizinho aleatório da solução, de imprimi-la, e a função heurística.

Para gerar a primeira solução, a versão atual do código cria uma caixa, e começa colocando nela os itens mais pesados que cabem na mesma, até não sobrar nenhum item que é menor ou igual à sua capacidade atual, nesse ponto ele cria outra caixa e recomeça esse processo, até não sobrar nenhum item. A ideia inicial era usar uma permutação aleatória de itens e colocá-los de forma sequencial em caixas, mas este método não resultava em soluções próximas dos resultados ótimos.

Para gerar um vizinho da solução atual, isto é, gerar uma solução fazendo apenas 1 alteração aleatória na solução, o seguinte algoritmo é usado: primeiro seleciona-se uma caixa aleatória, depois se criam dois vetores, chamados “Sequência 10” e “Sequência 11”. No vetor 10 são colocadas todas as caixas que tem espaço o suficiente para receber o menor item da caixa atual, sem exceder o seu limite, enquanto no vetor 11 são colocadas as caixas que podem trocar o seu menor item com o menor item da caixa atual, sem que nenhuma exceda o seu limite, e de forma que dois itens de tamanho igual não sejam trocados. A mesma caixa pode aparecer nos dois vetores. Então se escolhe de forma aleatória, mas com preferência para o primeiro, um dos vetores que não está vazio, e se realiza a ação para a qual ele foi criado com uma caixa aleatória dele.

Anteriormente, o código de gerar vizinho apenas trocava dois itens de lugar na sequência de itens, mas ele não garantia que os dois itens não estavam na mesma caixa, resultado em muitos “vizinhos” que eram efetivamente idênticos à solução original. Uma tentativa de melhorar a solução anterior foi fazer múltiplas trocas, mas isso gerava soluções completamente diferentes da melhor.

O cálculo para determinar a aptidão da solução (o quão boa ela é) é feito simplesmente obtendo o tamanho do vetor de caixas, como não existe uma operação para criar uma caixa, nunca haverá uma solução pior do que a atual, portanto a temperatura da heurística apenas define a chance de mudar de solução ou ignorar e tentar de novo, caso a temperatura mínima seja muito alta, o algoritmo sempre trocará de solução.

Essa variação foi escolhida pois após vários testes, pois é a que deu os melhores resultados e atingiu o melhor equilíbrio entre uma solução viável e tempo de execução. É possível atingir resultados melhores aumentando a temperatura ou diminuindo a velocidade de resfriamento e a temperatura mínima, porém isto acarretará num aumento drástico no tempo de execução do algoritmo.

Experimentos Finais

A experimentação foi realizada no computador do Danilo. Para testar todos os 6,1 mil casos de forma rápida foi usado o script “run.bat” para chamar o executável

repetidamente com todos os casos de teste. Este script também é responsável por imprimir no arquivo de saída o tempo de execução de cada teste, além de encontrar a solução ótima baixada e imprimi-la também.

Os parâmetros de temperatura inicial, variação de temperatura, e de temperatura mínima são pré-programados no código-fonte. No momento dos destes eles eram, respectivamente: 1000, 0.999, e 10. A variação de temperatura é um fator que será multiplicado na temperatura atual no final de toda iteração.

Como métricas da performance do programa, foram calculados a média do tempo de execução (em segundos), a precisão da resposta do algoritmo, e a maior diferença entre uma resposta ótima e uma resposta do programa dentre todos os casos de teste. A precisão das respostas representa o quão perto elas estavam do resultado ótimo, em média. Para calcular isso, foi utilizado a fórmula do Gap. O cálculo final da precisão pode ser representado pela fórmula:

$\frac{\sum_{i=1}^{N} 1 - \frac{R_i - P_i}{P_i}}{N}$, onde *N* é o número de casos de teste, R_i é a resposta do programa para o caso de teste *i*, e P_i é a solução ótima para o caso de teste *i*. $\frac{R_i - P_i}{P_i}$ representa a distância (ou o "gap") entre a resposta do programa e a solução ótima. Como queremos o inverso disso (o quão **próxima** a reposta está, colocamos 1 menos isso no somatório.

Por último, a média do tempo de execução do programa na verdade é a média dos segundos que o programa levou para executar, pois o método de medição atual não leva milissegundos em consideração.

Resultados

Após processar os 6195 casos de teste, o programa demonstrou uma média de 0,293462 segundos para calcular a resposta de cada caso, e obteve uma precisão de 95,389549% nas respostas, com a maior discrepância entre resposta e solução ótima sendo 287 caixas. O maior tempo de execução foi 3 segundos, registrado em 28 casos.

Conclusão

Ao final da experimentação da versão atual do algoritmo, concluímos que é possível aumentar a precisão das respostas, mas o preço em tempo disso não vale a pena. Consideramos que 95,4% de precisão é o bastante, e 3 segundos sendo o maior tempo de execução é muito bom.

Compreendemos que o processador do computador do Danilo é mais rápido que os dos computadores que executarão o programa (3,4 GHz vs 3,7 GHz), mas estamos confiantes que a performance de tempo se manterá adequada.