TRABALHO DE MODELAGEM E OTIMIZAÇÃO ALGORÍTMICA

Ant System e Simulated Annealing

DIOGO BRUMASSIO 120122
CAIO AUGUSTO CANO 117416
JOÃO PEDRO PERES BERTONCELO 112650
VINICIUS OKAGAWA RODRIGUES 122944

MARINGÁ
08/04

SUMÁRIO

ALGORITMO DE OTIMIZAÇÃO POR COLÔNIA DE FORMIGAS

Introdução:

O Algoritmo de Otimização por Colônia de Formigas (ACO) é uma técnica meta-heurística baseada no comportamento das formigas reais. Essas formigas usam feromônios para comunicar informações sobre o caminho que estão seguindo. Ao seguir um caminho, elas depositam feromônio, tornando-o mais atrativo para outras formigas. Esse comportamento é usado no ACO para encontrar a melhor solução para um problema.

Desenvolvimento:

O código em Python apresentado implementa o algoritmo ACO para encontrar a melhor solução para um problema de TSP (Problema do Caixeiro Viajante). O algoritmo funciona da seguinte maneira:
Define as constantes do algoritmo, incluindo a constante de intensificação de feromônio, a taxa de evaporação de feromônio, o peso do feromônio e a distância na escolha da próxima cidade, o número de formigas na população e o número de iterações do algoritmo. Define as informações das cidades, incluindo uma matriz de distâncias entre as cidades. Inicializa as trilhas de feromônio com um valor constante. Define uma função para calcular a distância total percorrida por uma formiga.

Executa o algoritmo ACO. Para cada iteração, para cada formiga na população, a cidade inicial é escolhida aleatoriamente. A formiga segue um caminho construído usando a regra de escolha de probabilidade baseada na quantidade de feromônio depositado pelas formigas anteriores e a distância entre as cidades. A qualidade da solução é medida pela distância total percorrida pela formiga. A cada iteração, as trilhas de feromônio são atualizadas com base na qualidade da solução encontrada pela formiga, e uma certa quantidade de feromônio evapora. O processo é repetido até que o número máximo de iterações seja atingido.

O melhor caminho e a distância total percorrida pela formiga são impressos na tela.

Conclusão:

O algoritmo ACO é uma técnica meta-heurística eficaz para resolver problemas de otimização, incluindo o problema do caixeiro viajante. O código em

Python apresentado implementa o algoritmo ACO para encontrar a melhor solução para um problema de TSP e produz bons resultados. É importante destacar que, embora o algoritmo apresente um bom desempenho em muitos casos, não garante a obtenção da melhor solução global em todas as situações.

COMPARAÇÃO ENTRE ALGORITMO GENÉTICO E ANT SYSTEM

Tanto o algoritmo genético quanto o Ant System são técnicas populares de otimização para resolver problemas do Caixeiro Viajante. Vamos comparar as duas abordagens para ver suas semelhanças e diferenças.

Algoritmo Genético:

O algoritmo genético (AG) é um algoritmo de otimização baseado na teoria da evolução. Ele usa a seleção natural e a reprodução para encontrar soluções ótimas para um problema. No caso do problema do Caixeiro Viajante, o AG cria uma população de possíveis soluções e, em seguida, aplica operadores de crossover e mutação para criar novas soluções. O processo de seleção é usado para determinar quais soluções sobrevivem e quais são eliminadas. A cada geração, as soluções são avaliadas e as melhores soluções são mantidas.

Ant System:

O Ant System é uma técnica de otimização inspirada no comportamento de formigas. Ele usa uma colônia de formigas virtuais para encontrar o caminho mais curto entre os pontos de um grafo. No caso do problema do Caixeiro Viajante, o Ant System usa uma colônia de formigas virtuais para encontrar o caminho mais curto entre as cidades. Cada formiga constrói uma solução caminho parcial baseada na escolha de uma cidade próxima, de acordo com regras baseadas em feromônios deixados pelas formigas anteriores. As melhores soluções são atualizadas com feromônios para aumentar sua probabilidade de serem escolhidas novamente.

Comparação:

O Algoritmo Genético e o Ant System são técnicas de otimização que compartilham muitas semelhanças. Ambos usam uma população de soluções candidatas, avaliam essas soluções, e selecionam as melhores para continuar a busca. Ambos também permitem que novas soluções sejam geradas usando regras de cruzamento e mutação.

No entanto, existem algumas diferenças importantes. O Algoritmo Genético é mais flexível e pode ser usado em muitos tipos diferentes de problemas de otimização, enquanto o Ant System é mais específico para problemas de roteamento em grafos. O Ant System é mais inspirado em uma abordagem biológica

e depende do comportamento das formigas para explorar o espaço de soluções, enquanto o Algoritmo Genético depende da seleção natural e da reprodução para criar novas soluções.

Em geral, o Algoritmo Genético é uma escolha melhor para problemas de otimização mais gerais, enquanto o Ant System é mais adequado para problemas de roteamento em grafos, como o problema do Caixeiro Viajante. Ambas as técnicas podem fornecer soluções eficazes e eficientes, dependendo do problema em questão

ALGORITMO SIMULATED ANNEALING

Introdução:

O código que implementa o algoritmo Simulated Annealing para resolver o problema de corte de placas em Python é uma solução simples e eficiente para o problema.

Desenvolvimento:

O algoritmo funciona encontrando a melhor solução possível para o problema de corte de placas, em que é necessário encontrar a disposição mais econômica dos cortes na placa, minimizando a área não utilizada da placa.

O código consiste em uma função principal, simulated_annealing, que executa o algoritmo Simulated Annealing. A função recebe como entrada a lista de cortes, a largura e a altura da placa, bem como alguns parâmetros opcionais, como a temperatura inicial, a temperatura final, o fator de redução de temperatura e o número máximo de iterações.

O algoritmo começa com uma solução inicial e, em seguida, gera soluções vizinhas aleatórias e calcula a diferença de custo entre a solução atual e a vizinha. Se a solução vizinha tiver um custo menor, a solução atual é atualizada para a vizinha. Se a solução vizinha tiver um custo maior, ela pode ainda assim ser aceita com uma certa probabilidade, dependendo da temperatura atual e da diferença de custo.

Ao longo do processo, a temperatura é gradualmente reduzida, o que significa que a probabilidade de aceitar soluções piores diminui. Isso ajuda a garantir que o algoritmo explore soluções mais promissoras no início e, em seguida, se concentre em soluções cada vez melhores à medida que avança. A função cut_cost calcula o custo da solução, que é a área da placa não utilizada, enquanto a função neighbor gera uma solução vizinha aleatória a partir da solução atual. A função acceptance_probability calcula a probabilidade de aceitação de uma solução pior, com base na diferença de custo e na temperatura atual.

Conclusão:

Em geral, a implementação é eficiente e razoavelmente rápida, embora o tempo de execução possa ser maior para placas grandes ou soluções com muitos cortes. No entanto, o algoritmo geralmente encontra uma solução muito próxima da ótima em um número relativamente pequeno de iterações.

Portanto, a implementação do algoritmo Simulated Annealing em Python é uma solução eficaz para o problema de corte de placas e pode ser facilmente adaptada para resolver problemas semelhantes em outras áreas.