Trabalho Prático 2 - Algoritmo de Colônia de Formigas para o problema da p-Mediana

Hugo Araujo de Sousa

Computação Natural (2017/2)
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)

hugosousa@dcc.ufmg.br

Resumo. *Este trabalho objetiva o desenvolvimento de conceitos fundamentais na aplicação do Algoritmo de Colônia de Formigas (Ant Colony Optimization - ACO) para resolução do problema da p-Mediana. A estrutura básica do ACO é apresentada e adaptada ao problema a ser resolvido. Finalmente, a partir de dados de teste, são realizados experimentos e a análise dos resultados obtidos.*

1. INTRODUÇÃO

Como definido por [Brownlee 2011], o Algoritmo de Colônia de Formigas é um método dos campos de Inteligência de Enxames, Metaheurísticas e Inteligência Computacional. Nesse método, o comportamento de formigas na natureza, em particular a comunicação baseada em feromônio que elas realizam para encontrar bons caminhos na busca por comida em um ambiente, é inspiração para encontrar potenciais soluções para um problema.

Na busca por comida, formigas se espalham inicialmente aleatoriamente em seu ambiente. Uma vez que uma fonte de comida é localizada, as formigas que a encontraram começam a depositar feromônio nesse ambiente, marcando assim o caminho que as levaram até a fonte. Quando várias um mesmo caminho é percorrido várias vezes e por várias formigas, a quantidade de feromônio nesse caminho se torna notavelmente maior do que em outras partes do ambiente, enquanto caminhos pouco percorridos perdem feromônio à medida que o tempo passa, devido à evaporação do mesmo.A comunicação das formigas, e ponto fundamental para o algoritmo, ocorre através do feromônio depositado, uma vez que elas tendem a seguir por caminhos com maior quantidade de feromônio.

Usando esse comportamento como inspiração, surge o Algoritmo de Colônia de Formigas, cuja estratégia geral é a de construir soluções candidatas para um problema de forma estocástica. Essas soluções são construídas, então, de forma probabilística e têm suas qualidades avaliadas. A partir dessas medidas de qualidade, 'feromônio' é depositado nos caminhos que geraram soluções de maior qualidade e, dessa forma, novas soluções criadas tendem a seguir pelos mesmos caminhos.

Nesse trabalho, o Algoritmo de Colônia de Formigas será utilizado para resolver o problema da p-Mediana com restrições de capacidade. Esse problema consiste em decidir onde localizar p centros em uma rede (composta por vértices e arestas) de forma a minimizar a soma de todas as distãncias de cada vértice ao centro mais próximo. Nesse problema também existem restrições de capacidade de atendimento dos centros. Esse problema é um problema de otimização combinatória NP-difícil.

2. MODELAGEM

Para utilizar o Algoritmo de Colônia de Formiga para o problema da p-Mediana, é necessário realizar uma modelagem relativa à construção de soluções, comportamento de formigas, depósito de feromônio, entre outros aspectos. No trabalho em questão, a modelagem seguiu em grande parte [de França et al. 2005].

2.1. Indivíduos

A primeira decisão de implementação em Programação Genética é a de como representar os indivíduos que representarão soluções para o problema a ser resolvido. Para o problema de Regressão Simbólica, uma solução é uma função do tipo $f : \mathbb{R}^m \rightarrow \mathbb{R}$. Dessa forma, a representação escolhida para representar uma função que resolva o problema é a de uma árvore onde os nós internos representam operadores e os nós folha são formados por terminais, isto é, variáveis da função ou constantes. Para os nós que representam operadores, os nós filhos serão os operandos. A Figura **??** mostra um exemplo de indivíduo com essa configuração.

Para este trabalho, o conjunto de terminais e operadores escolhidos é mostrado a seguir:

- **Operadores:**
  - **Binários:**
    - **+**: Operador de adição.
    - **-**: Operador de subtração.
    - *****: Operador de multiplicação.
    - **/**: Operador de divisão.
    - **^**: Operador de exponenciação.
  - **Unários:**
    - **log**: Função logaritmo (base e).
    - **sin**: Função seno.
    - **cos**: Função cosseno.
    - **sqrt**: Função raiz quadrada.
- **Terminais:**
  - **Constantes**: Valores reais aleatoriamente gerados.
  - **Variáveis**: Representam variáveis da função a ser aproximada. O número de variáveis aleatoriamente geradas é igual ao número de variáveis de entrada da função que o usuário deseja aproximar.

2.2. Fitness

Como dito anteriormente, cada indivíduo presente em uma determinada geração deverá ser avaliado para obter uma medida de quão bem esse indivíduo aproxima a função cujos valores de entrada e saída são fornecidos pelo usuário. O critério de avaliação da qualidade do indivíduo, também conhecido como fitness, para este trabalho, será dado pela raiz quadrada do erro quadrático médio (RMSE).

$$f(Ind) = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(Eval(Ind, x) - y)^2}$$

onde Ind é o indivíduo sendo avaliado, $Eval(Ind, x)$ avalia o indivíduo Ind no conjunto de entrada fornecido x, y é a saída correta da função para a entrada x e N é o número de exemplos fornecidos.

Dessa forma, a função $Eval(Ind, x)$ vai atribuir os valores em x às variáveis presentes em Ind, e retornar o valor y correspondente à saida da expressão simbólica que Ind representa para x.

2.3. Geração de População Inicial

Existem vários métodos para geração da população inicial de indivíduos que serão evoluídos ao longo das gerações em que o programa executará. Neste trabalho é usada uma combinação de dois deles:

- **Full:** Gera indivíduos cuja expressão simbólica é dada por uma árvore completa, i.e., todas as folhas estão na mesma profundidade.
- **Grow:** Gera indivíduos com formas e tamanhos variados, uma vez que os nós são selecionados tanto dos operadores quanto dos terminais até que a profundidade limite é atingida. Quando isso acontece, somente terminais são selecionados para compor nós.

Para garantir uma diversidade elevada de indivíduos na população inicial, é usado o método **Ramped Half-and-Half**, que combina os métodos Full e Grow, gerando metade da população com o primeiro e metade com o segundo.

2.4. Evolução

Uma vez gerada a população de indivíduos, o algoritmo de Programação Genética entra em um laço de repetição, evoluindo indivíduos e gerando populações cada vez melhores. A evolução dos indivíduos presentes em uma população se dá através de seleção e aplicação de operadores genéticos.

2.4.1. Seleção

É desejável que, a cada geração, sejam mantidas as características dos indivíduos que apresentam melhor fitness (menor, no caso da regressão simbólica). Logo, é necessário incluir um mecanismo que permita selecionar, dada uma certa população, indivíduos que se destacam.

Nesse trabalho, o mecanismo de seleção implementado foi o de **Seleção por Torneio**. Esse tipo de seleção escolhe um grupo aleatório de tamanho k dentre os indivíduos de uma população e retorna aquele, dentre os k escolhidos, que tenha a melhor fitness.

Além disso, foi adicionado ao projeto a possibilidade de se utilizar o conceito de **elitismo**. Com esse conceito, os n melhores indivíduos de cada geração são simplesmente passados para a geração seguinte, onde n é um parâmetro definido pelo usuário.

2.4.2. Operadores Genéticos

A partir dos indivíduos selecionados em cada geração, aplica-se um conjunto de operadores genéticos sobre os mesmos, a fim de garantir que novos indivíduos (baseados nesses

que se destacaram em cada geração) possam surgir. Cada um desses operadores genéticos contribui de uma forma para o algoritmo e é aplicado seguindo uma certa probabilidade pré-definida.

Os operadores genéticos implementados nesse trabalho são apresentados a seguir.

- **Cruzamento:** Dois indivíduos são selecionados e partes desses indivíduos (subárvores) são trocadas de lugar, gerando dois novos indivíduos. Os indivíduos pais não são alterados nesse processo.
- **Mutação:** Uma parte aleatória de um indivíduo selecionado é alterada. Essa parte pode ser removida, expandida, simplesmente ter seu conteúdo trocado, etc. Para o trabaho, foi implementada mutação de subárvore, onde um nó é escolhido para ser expandido, sendo criada uma nova subárvore a partir do mesmo.
- **Reprodução:** Um indivíduo selecionado é simplesmente passado adiante para a próxima geração.

3. IMPLEMENTAÇÃO

Dada a modelagem do problema mostrada na Seção 2 o algoritmo principal segue o pseudocódigo mostrado na Figura **??**.

Para o trabalho, o algoritmo então foi implementado usando a linguagem Python 3.

A classe **Individual** representa um indivíduo conforme descrito na Seção 2.1, e é a principal do projeto. Nela estão presentes todos os métodos para construção de indivíduos, populações, avaliação de indivíduos, etc.

Algumas decisões de implementação importantes são discutidas a seguir:

- Biblioteca Numpy utilizada no projeto intensivamente. É com ela que toda a geração de números aleatórios é feita e as operações dos nós são feitas.
- As constantes geradas para nós terminais são números reais no intervalo $[-10, 10]$.
- A fim de otimizar a cópia dos indivíduos pais durante a etapa de cruzamento, foi criada uma função copy, mais específica do que a alternativa deepcopy da linguagem.
- No método de criação de indivíduos Grow, os tipos dos nós são gerados aleatoriamente com probabilidades $0.8, 0.1$ e 0.1 para nós de operadores, variáveis e constantes, respectivamente.
- Para evitar que os indivíduos gerados cresçam indefinidamente, existe uma penalidade para indivíduos que excedam um tamanho limite. Esse tamanho (número de nós) máximo é dado por $2^(D) - 1$, onde D é a profundidade máxima das árvores geradas. Indivíduos que ultrapassem esse tamanho recebem fitness infinita.
- A produndidade máxima dos indivíduos não é variada, tendo valor 7.

4. ESTRUTURA DO PROJETO E EXECUÇÃO

Os arquivos de código-fonte do projeto se encontram na pasta **src**. Nela, o código-fonte é dividido da seguinte forma:

- **individual.py:** Define a classe Individual e todos os métodos necessários para criação e avaliação de indivíduos e criação de populações.

- **gp.py:** Métodos auxiliares para Programação Genética, tais como seleção por torneio, avaliação de população e operadores genéticos.
- **tp1.py:** Programa principal. Nele está implementado o algoritmo principal de Programação Genética, além de todos os métodos de manipulação de entrada e saída.

4.1. Execução e Parâmetros

A fim de facilitar a execução do programa, foram definidos parâmetros para alterar alguns pontos chave para Programação Dinâmica. São eles:

- **Semente:** Número inteiro usado na geração de números aleatórios durante a execução do programa. Note que, mantendo todos os outros parâmetros fixos, a saída do programa para uma mesma entrada e valor de seed será sempre igual. A semente da execução pode ser definida com a flag **-s** e tem valor padrão 0.
- **Tamanho da população:** Número inteiro que determina o tamanho de indivíduos presente em cada geração. Pode ser definido com a flag **-p** e tem valor padrão 54.
- **Tamanho do torneio:** Número de indivíduos que competem em cada seleção por torneio. Definido com a flag **-k** e tem valor padrão 7.
- **Número de gerações:** Define o número de gerações pelo qual o programa deve executar. Definido com a flag **-g** e tem valor padrão 10.
- **Taxa de cruzamento:** Probabilidade de usar o operador de cruzamento para gerar filhos em cada geração. Definida com a flag **-c**, valor padrão 0.9.
- **Taxa de mutação:** Probabilidade de usar o operador de mutação para gerar filhos em cada geração. Definida com a flag **-m**, valor padrão 0.05.
- **Tamanho da elite:** Tamanho da elite a ser transferida automaticamente para a próxima geração, a cada geração. Definido com a flag **-e**, valor padrão 2. Note que para desativar o elitismo, basta usar **-e 0**.

Note que a taxa de reprodução não é definida pelo usuário, mas calculada em função das taxas de cruzamento e mutação, como mostrado na Figura **??**.

Para executar o programa, o comando abaixo é usado:

tp1.py [h] [s RSEED] [p POP_SIZE] [k KTOUR] [g NGEN] [c CROSSR] [m MUTR] [e ELIT] train_file test_file

Onde RSEED indica a semente, POP_SIZE o tamanho da população, KTOUR o número de competidores em torneios, NGEN o número de gerações, CROSSR a probabilidade de cruzamento, MUTR a probabilidade de mutação, ELIT o tamanho da elite em cada geração, train_file o nome do arquivo com as entradas de treino e test_file o nome do arquivo com as entradas de teste.

Todos os parâmetros entre colchetes acima são opcionais.

4.2. Entrada e Saída

Os arquivos de entrada, tanto de treino quanto de teste, devem possuir a mesma estrutura. O arquivo de treino é usado para evoluir as soluções do programa até o número máximo de gerações ser alcançado. Quando essa etapa é finalizada, as melhores soluções encontradas são avaliadas com o arquivo de teste.

Em ambos os arquivos, cada linha representa uma amostra da função a se aproximar. Sendo $x + 1$ valores de ponto flutuante separados por vírgula, onde x é o número de variáveis da função. A última coluna, então, representa a saída y da função para os valores de entrada das colunas anteriores.

Em relação à saída do programa, é impresso, a cada geração, um resumo de estatísticas. Primeiramente é impresso o número da geração atual, em seguida o valor da fitness do melhor indivíduo da geração, da fitness do pior indivíduo, a fitness média considerando todos os indivíduos da geração, o número de indivíduos repetidos, a taxa de melhoria para mutação e cruzamento.

As taxas de melhoria são calculadas da seguinte forma:

$$Imp(op) = \frac{NImp}{NGen}$$

Sendo Imp(op) a taxa de melhoria para o operador op, NGen representa o número de indivíduos criados em determinada geração utilizando-se o operador genético op e NImp o número, dentre esses indivíduos gerados, daqueles que apresentaram fitness melhor do que seus pais (indivíduos sobre os quais o operador op foi aplicado).

Exemplo de estatísticas para uma certa geração:

Generation 7
Best fitness: 2.77477679664e-09
Worst fitness: inf
Average fitness: 1.70803185868e+17
Number of repeated individuals: 15
Mutation improvement rate: 0.0%
Crossover improvement rate: 6.25%

5. EXPERIMENTOS

Nessa seção serão apresentados os experimentos realizados. Todos eles foram executados em uma máquina Intel Core i7-5500U, 2.40GHz, 4 cores, 8GB de memória RAM e sistema operacional ubuntu 14.04 LTS.

5.1. Metodologia

Muitas instâncias de teste foram executadas para cada uma das bases de teste. Alguns exemplos de saídas obtidas estão presentes na pasta **test**.

Um script para teste de todas as bases presentes no diretório **datasets** com todas as possíveis configurações de parâmetro e 30 execuções/sementes foi desenvolvido. Para executá-lo, basta digitar o comando a seguir no terminal:

./run_tests.py

A metodologia para execução dos testes seguiu muitas das orientações presentes em [Brownlee 2011].

5.2. Experimentos

Abaixo são mostrados os principais experimentos realizados. Os resultados são apresentados na Seção 6. Os valores apresentados foram obtidos da média de 30 execuções.

- **Experimento 1:** No primeiro experimento, objetivou-se simplesmente a verificação de convergência da melhor indivíduo em relação ao número de gerações.Os parâmetros utilizados estão listados abaixo.

  - Base keijzer-7:
    Tamanho da população:
    Competidores em torneios:
    Número de gerações:
    Taxa de cruzamento:
    Taxa de mutação:
    Tamanho da elite:

  - Base keijzer-10:
    Tamanho da população: 100
    Competidores em torneios: 2
    Número de gerações: 100
    Taxa de cruzamento: 0.8
    Taxa de mutação: 0.1
    Tamanho da elite: 2

  - Base house:
    Tamanho da população: 100
    Competidores em torneios: 7
    Número de gerações: 50
    Taxa de cruzamento: 0.9
    Taxa de mutação: 0.05
    Tamanho da elite: 2

- **Experimento 2:** Com o experimento 2, procurava-se determinar a melhor configuração, dos parâmetros de número de gerações e tamanho de população inicial. Para isso, a base house foi utilizada. Os parâmetros são mostrados abaixo.
  Base house
  Tamanho da População: {5, 100, 500}
  Competidores em torneios: 7
  Número de gerações: 50
  Taxa de cruzamento: 0.8
  Taxa de mutação: 0.1
  Tamanho da elite: 2

- **Experimento 3:** Já com valores de tamanho de população e número de gerações fixados, o próximo passo foi determinar os parâmetros de probabilidade de aplicação dos operadores genéticos.
  Foram testadas duas configurações:
  Base house
  Tamanho da População: 50
  Competidores em torneios: 7
  Número de gerações: 50

Cruzamento, mutação: $\{0.9, 0.05\}$ e $\{0.6, 0.3\}$
Tamanho da elite: 2

- **Experimento 4:** Uma vez estabelecidas as taxas de cruzamento e mutação que promovem um melhor resultado do programa, é necessário testar os valores de tamanho de torneio. Esse valor refere-se ao número de competidores que competem a cada seleção de indivíduos por torneio.
  Base keijzer-7
  Tamanho da População: 50
  Competidores em torneios: 3 e 7
  Número de gerações: 50
  Taxa de cruzamento: 0.9
  Taxa de mutação: 0.05
  Tamanho da elite: 2

- **Experimento 5:** Em seguida, vemos como é o comportamento do melhor indivíduo de cada geração, e, consequentemente, da melhor fitness, utilizando ou não de elitismo. Parâmetros utilizados:
  Base house
  Tamanho da População: 50
  Competidores em torneios: 7
  Número de gerações: 50
  Taxa de cruzamento: 0.9
  Taxa de mutação: 0.05
  Tamanho da elite: 0 e 2

- **Experimento 6:** Uma vez que a diversidade dos indivíduos presentes em uma população cai muito, a melhor fitness de cada geração tende a permanecer sem grandes modificações. Para este trabalho, a única medida, ainda que não muito eficaz, de diversidade, é o número de indivíduos repetidos. Um indivíduo é considerado repetido aqui quando já existe na mesma geração um indivíduo que apresenta a mesma estrutura (mesma expressão simbólica - genótipo do indivíduo).
  Base keijzer-7
  Tamanho da População: 500
  Competidores em torneios: 7
  Número de gerações: 50
  Taxa de cruzamento: 0.9
  Taxa de mutação: 0.05
  Tamanho da elite: 2

6. RESULTADOS

6.1. Experimento 1: Convergência de melhor fitness

Esse experimento comprova que, de fato, o algoritmo implementado leva as populações geradas inicialmente a convergirem eventualmente, tentando sempre se aproximar do valor ótimo de fitness igual a zero para as bases keijzer (o que indica que a solução ótima pode ter sido encontrada).

Podemos ver que, em um primeiro momento a melhor fitness cai de forma bem acentuada, em seguida convergindo a um valor específico para cada uma das bases.

6.2. Experimento 2: Tamanho de população e número de gerações

A partir do experimento 2 foi possível constatar que o tamanho da população tem grande impacto na melhor fitness das primeiras gerações. Isso se deve ao fato de que aumentando o número de invivíduos estamos aumentando a diversidade e explorando melhor o espaço de busca. Também podemos observar na Figura **??** que, para todos os valores de tamanho de população, o programa converge para valores muito próximos, e, além disso, isso ocorre sempre por volta da geração 30. Concluímos então que o custo adicional de avaliar mais indivíduos não compensa muito quando vemos que o ponto de convergência é muito próximo para os valores de tamanho de população testados.

6.3. Experimento 3: Operadores Genéticos

Com esse experimento podemos observar como a velocidade de convergência é afetada diretamente pela escolha dos parâmetros relativos às probabilidades de uso de cada um dos operadores genéticos.

Vemos que, ao utilizar uma configuração onde a taxa de cruzamento é muito elevada (garantindo que gerações posteriores possam apresentar grandes semelhanças com as os indivíduos de melhor fitness das gerações anteriores) e a de mutação é reduzida (somente para garantir que haja uma certa diversidade para explorar o espaço de busca) o programa não só converge mais rapidamente, como chega a um valor bem melhor de fitness. A Figura **??** mostra os resultados para esse experimento.

6.4. Experimento 4: Torneio

No experimento 4, ilustrado na Figura **??**, vemos que é importante selecionar um tamanho de torneio que seja grande o suficiente para incluir bons indivíduos, porém não muito grande que sempre selecione aqueles de maior fitness (condição importante para manter diversidade).

6.5. Experimento 5: Elitismo

Com esse experimento vemos que o uso do elitismo é importante para garantir que a solução seja sempre melhorada ou mantenha-se constante. Sem o uso do mesmo, a melhor fitness pode piorar de uma geração para a próxima. A Figura **??** ilustra o experimento.

6.6. Experimento 6: Indivíduos repetidos

O resultado desse experimento, como indicado na Figura **??**, mostra que, uma vez que o programa encontra uma solução muito boa (próxima da solução ótima), todos os outros indivíduos de uma geração tendem ter seus genótipos aproximados a essa solução.

6.7. Observações

Os experimentos realizadas possibilitaram observar diversas características importantes, tanto de Programação Genética em geral, quanto da implementação desse trabalho e das bases de dados utilizadas.

Vemos que a escolha da probabilidade dos operadores genéticos e tamanho de torneio pode melhorar a diversidade dos indivíduos, promovendo uma maior exploração do espaço de busca.

Além disso, à medida que a melhor fitness de uma população converge para um determinado valor, é de se esperar que o número de indivíduos repetidos também tenda a aumentar, dada uma taxa de cruzamento alta.

Um ponto importante sobre os dados coletados diz respeito à discrepância entre os valores de melhor, pior e fitness média das gerações. Mesmo que uma determinada geração possua melhor fitness muito baixa, nada impede que a fitness média e pior sejam muito elevadas. De fato, é isso que se espera quando utilizamos operadores como multiplicação e exponenciação no conjunto de operadores dos indivíduos.

7. CONCLUSÃO

O trabalho aqui apresentado foi de grande utilidade para fixar vários conceitos vistos em aula na disciplina de Computação Natural. Creio que a habilidade de modelar um problema como uma instância de Programação Genética foi reforçada e poderá ser aplicada com maior facilidade no futuro.

Pontos chave de aprendizado devem ser ressaltados. Na Programação Genética é importante definir como representar os indivíduos que representam soluções, e, para isso, é importante identificar como mapear soluções já conhecidas para soluções genéricas.

Além disso, pode ser constatado que a escolha dos parâmetros do algoritmo afetam significativamente os resultados obtidos. Sendo o processo de teste algo iterativo, é importante definir a cada momento a configuração de parâmetros que afeta de maneira mais positiva a saída do programa.

A maior dificuldade encontrada foi lidar com a grande quantidade de testes necessária para avaliar o projeto. Visto que cada geração pode demorar muito tempo para executar (cada indivíduo deve ser avaliado e o tamanho da entrada afeta a performance diretamente), foi necessário começar o processo de testes assim que possível para obter dados e poder analisá-los.

De maneira geral, o aprendizado obtido tem grande apelo, não só para conceitos de Programação Genética, mas também de Computação Evolucionária em geral.

8. REFERÊNCIAS

Brownlee, J. (2011). *Clever Algorithms*. LuLu, 1st edition.

de França, F. O., Zuben, F. J. V., and de Castro, L. N. (2005). Max min ant system and capacitated p-medians: Extensions and improved solutions. *Informatica (Slovenia)*, 29(2):163–172.