Instituto Tecnológico de Aeronáutica

Departamento de Engenharia de Software
Divisão de Ciência da Computação
Laboratório de CTC-17

Relatório da atividade

Projeto 5:
Algoritmos genéticos

Lucas França de Oliveira, lucas.fra.oli18@gmail.com
Matheus Felipe SBF Rodrigues, matheus.felipesbf@gmail.com
Professores:
Prof. Paulo André Castro , pauloac@ita.br
Prof. Paulo Marcelo Tasinaffo, tasinaffo@ita.br

4 de dezembro de 2017

1 Introdução

Os algoritmos genéticos consistem em classes de algoritmos que resolvem problemas em que se deseja maximizar uma função de valor ou minimizar uma função de custo. Nesta atividade, ela será denotada **função de fitness**. Existe um espaço de estados, possivelmente infinito e denso, em que cada possível estado possui um valor de fitness associado a ele. O algoritmo deve achar um estado que maximiza a função de fitness ou que torna ela maior que um certo valor desejado.

A ideia dos algoritmos genéticos consiste na evolução de uma população de agentes os quais cada um possui um estado associado. O algoritmo executa rodadas de evolução, em que a população - o conjunto de todos os agentes - sofre uma transformação de forma que o fitness de todos os seus integrantes, médio ou máximo, aumente, até que se alcance um valor aceitável.

Nos algoritmos genéticos clássicos, essa transformação é caracterizada por um processo de reprodução cruzada e de mutação. O primeiro serve para garantir o aumento do fitness: os agentes com melhor desempenho são escolhidos para ter suas características referentes aos seus estados misturadas, a fim de que se gerem novos estados com um fitness tão bons quanto aos da geração anterior. Os agentes com pior desempenho são eliminados. Por fim, dada uma população escolhida para reprodução, é possível que o estado ótimo não seja alcançável pelos descendentes desta população, então se adicionam mutações - pequenas alterações aleatórias - nos estados dos descendentes. O termo "algoritmo genético" se deve à semelhança desse processo com a reprodução natural cruzada de seres vivos, em que o agentes são os indivíduos e o estado é o amterial genético.

Existem, também, outras possíveis transformações aplicáveis à população que podem ser usadas com resultados bons. Neste trabalho, serão implementados dois algoritmos para resolver um mesmo problema: o Particle Swarm Optimization (PSO) e o Differential Evolution (DE). No fim, estes serão comparados em termos de desempenho para a resolução do problema. Mais especificamente, tempo de execução em milissegundos. Para isso, foi passado no roteiro do laboratório um mesmo problema a ser resolvido por ambos: o problema das N rainhas, que será descrito na próxima seção.

Este labpratório tem por objetivos:

- Estudo e implementação do algoritmo Particle Swarm Optimization (PSO);
- Estudo e implementação do algoritmo Differential Evolution (DE);
- Aplicação dos algoritmos acima para o problema das N rainhas;
- Estudo comparativo dos algoritmos acima em relação à performance para resolver o problema das N rainhas.

Os códigos podem ser visualizados no repositório: https://github.com/splucs/Lab5_CTC17

2 Descrição do problema

O problema das N rainhas consiste em, dado um tabuleiro de xadrez de tamanho NxN, deseja-se colocar N rainhas - peças que atacam todas as outras peças pertencentes à mesma linha, coluna ou diagonal que a peça atual - no tabuleiro de forma a que nenhuma delas se ataque.

Modelando o desafio como um problema de otimização, adotamos como estado uma possível configuração do tabuleiro, que pode ser descrita como uma permutação dos números de 1 a N: o i-ésimo termo dessa permutação representa a posição de uma rainha na i-ésima linha. Assim, a restrição de linhas já é satisfeita. Como as permutações não possuem elementos repetidos, a restrição de colunas também é satisfeita. Resta apenas trabalhar pra achar um estado que satisfaz as condições de diagonal.

3 Particle Swarm Optimization (PSO)

Este método foi desenvolvido por R. C. Eberhar e J. Kennedy, e foi inspirado pelo comportamento de enxame dos pássaros. Em uma população de pássaros que procura o melhor ponto para se aninhar, cada pássaro voa com uma velocidade e olha por um caminho quais pontos já forma visitados e qual o melhor ponto para construir o ninho já visitado. Ele sabe não apenas o melhor ponto visitado por ele mesmo, mas também o melhor já visitado por todos os pássaros do enxame.

Ao perceber que o ponto atual está distante de um ponto bom observado por ele ou por outro pássaro, ele tende a aumentar a velocidade de forma a sair desse local rápido. Pode-se modelar sua função de velocidade, portanto, por: um fator de inércia que consiste em uma parcela da velocidade anterior, um fator proporcional à diferença de estado entre o atual e o ótimo local, e um fator proporcional à diferença de estado entre o atual e o ótimo global.

Modelando o problema das N rainhas para o PSO, pode-se imaginar agentes que "voam" pelo espaço de estados de permutações de uma array de 1 a N. Modela-se o estado como uma permutação. O conceito de velocidade está associado ao quão rápido deve-se sair do estado atual dependendo do quão distante ele está dos estados ótimos observados global e local. A movimentação, portanto, será um conjunto de trocas aleatórias dentro da permutação e a velocidade, o número de trocas. A distância entre dois estados é o número de posições da permutação em que os termos diferem. A **função de fitness** será o número de pares de rainhas que não se atacam, e esta deve ser maximizada até o valor $N(N-1)/2$.

Com o intuito de se regular o peso de cada fator sobre a velocidade, adicionam-se os parâmetros ω para a inércia, $C1$ para a distância entre o estado atual e o ótimo observado localmente, e $C2$ para a distância entre o estado atual e o ótimo observado globalmente. Um último parâmetro adicionado é o tamanho da população S.

Assim, o algoritmo pode ser modelado da seguinte forma:

1. Inicializa-se a população com S permutações aleatórias $x[i], 1 \leq i \leq S$, $p[i] = x[i]$, em que $p[i]$ é o máximo observado local e $g = max_{1 \leq i \leq S}(p[i])$, em que g é o máximo observado global.

2. Loop enquanto o melhor g tiver um fitness menor que $N(N-1)/2$, se falhar retorna o resultado g:

3. Para cada agente i em S, $velocidade[i] = \omega * velocidade[i] + C1 * dist(x[i], p[i]) + C2 * dist(x[i], g)$.

4. Para cada agente i em S, aplica-se sua velocidade: trunca-se $velocidade[i]$ e este valor é o número de trocas aleatórias aplicadas.

5. Atualiza-se $p[i] = max(p[i], x[i]), 1 \leq i \leq S$ e $g = max(g, max_{1 \leq i \leq S}(p[i]))$.

6. Volta para o item 2.

Com isso, foram realizados múltiplos testes variando-se o valor dos parâmetros fornecidos. Foi utilizado o valor de $N = 15$ para todos os testes, ou seja, o tabuleiro possui tamanho $15x15$ e há 15 rainhas nele.

O programa *PSO.cpp*, implementado em C++, pode ser acessado pelo anexo enviado ou pelo repositório no github. O código implementado em C++ utilizando o compilador GNU g++ 6.3 (-O3) em uma máquina 64 bits com processador Intel(R) Core(TM) i5-6300 CPU @ 2.40Hz apresentou os seguintes resultados (tabelas com tempos de execução até a solução em milissegundos, cada tempo é a média de 10 iterações):

$S = 20, \omega = 0.4$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	407	365	592	546	886
0.4	662	402	927	887	1123
0.6	510	1260	882	679	892
0.8	529	573	1131	950	1260
1.0	552	799	623	690	581

$S = 20, \omega = 0.6$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	411	1296	1585	1088	1588
0.4	319	990	861	1023	972
0.6	854	1409	895	1032	977
0.8	788	990	1328	1541	994
1.0	734	871	1345	727	2026

$S = 20, \omega = 0.8$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	937	1070	1839	2146	2224
0.4	823	1795	1494	2264	1899
0.6	1300	2096	2031	3539	2241
0.8	1337	1619	1697	1556	3207
1.0	1669	1915	2234	4409	2411

$S = 50, \omega = 0.4$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	464	373	957	815	707
0.4	537	570	685	1005	891
0.6	712	288	520	908	1488
0.8	804	794	485	599	1329
1.0	860	619	894	1271	970

$S = 50, \omega = 0.6$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	411	403	1117	729	999
0.4	456	347	1057	1452	1301
0.6	592	301	836	1368	878
0.8	794	2555	1262	1010	1013
1.0	1110	1489	1270	1389	784

$S = 50, \omega = 0.8$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	600	1713	1510	1175	2008
0.4	1028	1699	1717	1810	2325
0.6	1039	1272	1579	1765	2219
0.8	1649	2415	2263	2116	2029
1.0	1420	1855	2433	2274	2252

$S = 80, \omega = 0.4$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	889	607	827	735	1295
0.4	735	455	469	957	935
0.6	323	779	453	993	1064
0.8	473	693	659	1008	1818
1.0	561	543	1151	542	774

$S = 80, \omega = 0.6$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	785	621	499	898	555
0.4	616	512	971	1070	854
0.6	708	634	1275	1679	733
0.8	957	1074	983	1796	1734
1.0	1480	1374	808	808	1468

$S = 80, \omega = 0.8$

$C2\backslash C1$	0.2	0.4	0.6	0.8	1.0
0.2	780	716	909	1222	1267
0.4	913	1549	1560	1619	2301
0.6	1109	975	902	1489	2492
0.8	1148	696	1800	2812	3005
1.0	1494	2190	4075	2843	2788

O tempo geral médio foi de 1204.490 milissegundos e o mínimo foi de 301. Observa-se que o tempo é menor para valores menores de $C1$ e $C2$. Quanto a $C1$ e $C2$, isso se deve ao fato de que, quando a constante é alta, a velocidade será alta e o número de trocas será o suficiente para que o agente execute uma permutação aleatória a cada passo. Isso faz com que o algoritmo se reduza à busca completa sobre o espaço de estados. Houve uma diferença de tempo em relação a S, mas isso se deve à complexidade do algoritmo de $O(ISN)$, em que I é o número de iterações.

No geral, uma vez que se consolida um valor de $C1$ e $C2$ tais que o algoritmo não se torna a busca completa aleatória, o algoritmo se mostrou mais rápido e com um tempo menor. Cada entrada nas tabelas é uma média de 10 iterações, uma vez que é necessário amortizar o efeito do fator aleatório do método. Ressalta-se que, em todos os testes, uma solução ótima foi encontrada.

Foram feitos, também, testes com o tabuleiro tradicional, com $N = 8$, mas estes deram tempo de execução da ordem de microssegundos. Neste caso, o algoritmo de busca completa aleatória em si já possui um tempo de execução relativamente baixo.

4 Differential Evolution (DE)

Este método foi desenvolvido em 1995 por Prince e Storn, e se caracteriza por uma transformação diferencial aleatória de cada agente a cada passo.

A modelagem é semelhante à do PSO: cada agente, equivalente à partícula, representa um possível estado, uma permutação de N elementos. A **função de fitness** será o número de pares de rainhas que não se atacam, e esta deve ser maximizada até o valor $N(N-1)/2$.

O método se assemelha a uma busca gulosa aleatória de vários agentes em paralelo: cada um, a cada passo, faz algumas trocas aleatórias sobre a permutação que ele atualmente representa. Em seguida, compara-se o novo estado com o antigo e, caso seja melhor, o substitui. A forma como ele gera essa nova permutação afeta como o algoritmo funciona. Neste caso, como devemos manter a propriedade de permutação para satisfazer a restrição das colunas, devemos realizar trocas, e não um Crossover Binomial ou Exponencial como sugerido em sala. Para cada termo da permutação, se $rand(0,1) < CR$ para um CR dado, ele é trocado com outra posição aleatória da permutação.

Assim, o algoritmo pode ser modelado da seguinte forma:

1. Inicializa-se a população com S permutações aleatórias $x[i], 1 \leq i \leq S$ e $g = max_{1 \leq i \leq S}(p[i])$, em que g é o máximo observado global.

2. Loop enquanto o melhor g tiver um fitness menor que $N(N-1)/2$, se falhar retorna o resultado g:

3. Para cada agente i em S, $u[i] = GetDonorVector(x[i])$, em que GetDonorVector é uma função que aplica trocas aleatórias sobre uma permutação para cada elemento com probabilidade CR.

4. Atualiza-se $x[i] = max(u[i], x[i]), 1 \leq i \leq S$ e $g = max(g, max_{1 \leq i \leq S}(x[i]))$.

5. Volta para o item 2.

Assim, os únicos parâmetros são o tamanho do conjunto de agentes, S, e a probabilidade de troca, CR. O tamanho do tabuleiro foi adotado como $N = 15$, ou seja, temos 15 rainhas e um tabuleiro de tamanho $15x15$. O motivo da adoção deste valor é a ordem de grandeza com que os valores do tempo crescem. Um valor maior que esse levaria a um grande tempo de execução para testes. Valores menores, como o tabuleiro tradicional com $N = 8$, são resolvidos na ordem de microssegundos, como verificado em testes.

O programa *DE.cpp*, implementado em C++, pode ser acessado pelo anexo enviado ou pelo repositório no github. O código implementado em C++ utilizando o compilador GNU g++ 6.3 (-O3) em uma máquina 64 bits com processador Intel(R) Core(TM) i5-6300 CPU @ 2.40Hz apresentou os seguintes resultados (tabelas com tempos de execução até a solução em milissegundos, cada tempo é a média de 50 iterações):

$S \backslash CR$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
20	709	778	844	760	898	856	872	685	695	661
40	788	702	1043	739	811	800	739	633	744	640
60	776	730	723	640	719	693	542	821	720	806
80	885	793	844	783	694	883	910	960	892	873

Para um número menor de agentes, observou-se uma demora maior para valores intermediários de CR. Para valores maiores, não foi observado um padrão, mesmo o valor imprimido sendo a média de 50 iterações, uma amortização feita para reduzir a influência do fator aleatório.

Em média, o tempo de execução foi de 777.186 milissegundos. O mínimo foi de 633 milissegundos. O aumento do tamanho do conjunto de agentes pouco influenciou o tempo, apesar da complexidade do algoritmo ser de $O(ISN)$, em que I é o número de iterações. Isso mostra que, quando o número de agentes aumenta, eles necessitam de menos iterações para achar a solução, pois a busca tornou-se mais paralelizada.

5 Comparação entre os métodos

Particle Swarm Optimization, para valores de S baixo e com valores adequados de $C1$ e $C2$, possui um resultado melhor que o método Differential Evolution. O primeiro também possui um valor mínimo melhor que o segundo. Mesmo com valores maiores de S, o PSO se mostrou no mínimo tão rápido quanto o DE, bastando apenas o ajuste dos demais parâmetros.

O método de Differential Evolution, por outro lado, possui uma média menor e, para todo o espaço de parâmetros, possui resultados melhores que se comparado com todo o espaço de parâmetros do PSO. No entanto, uma vez conhecido o valor que otimiza o Particle Swarm Optimization, é possível ter um resultado melhor.

Uma das vantagens do PSO é o fato de que existe comunicação entre os agentes. Cada partícula conhece o melhor estado alcançado globalmente até aquele momento e usa esse valor pra direcionar melhor sua busca. O DE, por outro lado, não transmite informações entre os agentes, consolidando-se como um algoritmo guloso em paralelo. Essa é a principal diferença entre os dois, uma vez que, caso o DE levasse em consideração o melhor valor global em vez do atual, ele se tornaria semelhante ao PSO.

6 Conclusão

Todas as atividades foram concluídas com êxito. O problema foi resolvido para tamanhos menores que ou iguais a $N = 15$ em tempo hábil para os casos em que existe solução. O tempo de execução considerando a variação de parâmetros foi feita e os algoritmos foram comparados em desempenho. O PSO se mostrou melhor se bem ajustado, enquanto o DE se mostrou melhor no caso geral de possibilidades de parâmetros.

O trabalho se mostrou útil para aprimorar o conhecimento sobre os métodos de inteligência artificial apresentados - algoritmos genéticos, Particle Swarm Optimization e Differential Evolution. O problema estudado é um clássico da literatura, considerado um teste de benchmark para diversos paradigmas de solução de problemas. A execução deste laboratório foi um excelente exercício de aplicação das técnicas abordadas. O laboratório se mostrou simples de implementar, tendo os pseudo-códigos dos algoritmos apresentados no material didátio do curso. Não houve impedimentos.