Relatório do exercício de Algoritmos Genéticos e Hill Climbing

Guilherme de Brito Freire

1) Algoritmo genético

Introdução

Usando o algoritmo genético com os valores padrões (pi = 50, ng = 100, pc = 80%, pm = 1%) não obtive resultados muito bons. Em nenhuma das execuções o algoritmo conseguiu chegar na solução certa, sempre ficava preso em um máximo local. A solução para um sudoku de 17 “dicas” é única, então se o algoritmo chegasse nela, seria apenas uma.

Implementação

Usei uma biblioteca de python chamada DEAP. Achei muito boa e fácil de entender/usar a interface dessa biblioteca. Ela permite muita flexibilidade para modificar qualquer parâmetro que possa existir em um algoritmo genético. A vantagem de usar essa biblioteca é que ela já vem com muitas coisas básicas e comumente usadas em um AG prontas. E qualquer customização que queira fazer, posso trocar a função usada dentro do algoritmo por meio do que a documentação chamada de *toolkit*.

Inicialmente tinha optado pela codificação do DNA dos indivíduos através de um vetor de 64 posições, no qual cada posição pode assumir um valor de 1 a 9. A inicialização do vetor era aleatória e as restrições do tabuleiro adicionadas posteriormente.

Entretanto essa versão ficou com um desempenho que eu julguei ruim e resolvi fazer uma segunda versão (genetic2.py). Nessa versão a inicialização é feita de forma que cada linha da tabela tenha apenas uma instância de cada valor de 1 a 9. Assim, já começo em um estado que é melhor do que um 100% aleatório. Com essa mudança, a população inicial começou com uma avaliação de tabuleiro muito maior e consequentemente chegou no ponto de convergência (máximo local) comparativamente mais rápido. Isso resultou também em um desempenho um pouco melhor em termos de solução final.

Análise de parâmetros

Analisando e modificando os parâmetros um de cada vez, percebi como os parâmetros influenciam na busca. Se aumentarmos a chance de ocorrer uma mutação, o algoritmo melhora consideravelmente. 1% de chance simplesmente se provou ser muito baixo. Entretanto não é recomendado aumentar muito essa probabilidade, pois o algoritmo começa a ficar muito ruidoso e se perder na busca. Empiricamente, posso dizer que um valor bom de mutação para esse problema com os outros parâmetros fixos é 10%. Talvez isso se dê devido ao fato de que a população tem tamanho padrão de 50 indivíduos. Como 50 indivíduos não permite uma fronteira de busca muito larga, a mutação se encarrega de explorar ramos mais remotos da árvore.

O que nos leva ao segundo parâmetro para modificar, o tamanho da população. Modificar esse parâmetro é o que mais tem efeito sobre o desempenho do algoritmo. Baseado

nos teste, cheguei à conclusão de que 300 indivíduos é um tamanho bom para esse problema com os outros híper-parâmetros fixos (e a mutação de volta em 1%). Com essa população maior, o algoritmo consegue ter uma fronteira de busca maior no início, o que é importante, pois torna o melhor caminho mais fácil de ser encontrado. Aumentar muito mais a população, não significa aumentar muito mais o desempenho também. Parece que cada parâmetro tem um valor bom e quanto mais nos afastamos dele (para mais ou para menos), pior fica o resultado.

Analisando a quantidade de iterações, percebi que também não melhora afeta muito o algoritmo. Com os parâmetros padrões, o algoritmo converge muito rápido, perto de 30 iterações. Portanto, a partir desse valor, aumentar o número de iterações não vai melhorar a resposta do algoritmo, mas trás os indivíduos para mais próximos uns dos outros. Essencialmente diminuindo o desvio padrão.

Finalmente, alterar o crossover influência tanto na resposta quando na velocidade de convergência. Botando a probabilidade de crossover muito baixa, o algoritmo se estabiliza rapidamente e não progride muito. Ao passo que aumentar a chance de ocorrer o crossover aumentou significativamente o desempenho do algoritmo. Afinal isso força a população a quase sempre se atualizar, e por tanto achar outro estado, possivelmente melhor se torna mais fácil.

Alterando o tabuleiro

Foi interessante trocar a quantidade de valores fixos no tabuleiro. A priori, eu achava que retirar restrições fosse ajudar o algoritmo, uma vez que existem agora mais soluções possíveis (17 é o número mínimo de “dicas” para existir uma solução única). Entretanto, o desempenho quase não foi afetado removendo restrições. O que mudou foi a velocidade de convergência. Com menos restrições, o algoritmo pode convergir para qualquer uma das soluções possíveis (e consequentemente a quantidade de mínimos locais também aumenta). Assim, o algoritmo acabou chegando mais rapidamente em um mínimo local e estagnou lá.

Agora, quando aumentamos, o algoritmo obteve um desempenho melhor. A princípio isso parece contra-intuitivo. Entretanto, ao restringir mais as condições do problema, o espaço de busca fica consideravelmente menor e a quantidade de máximos locais cai bastante. Os reslutados dos experimentos mostram exatamente isso: em geral o algoritmo teve um êxito melhor do que com menos restrições.

Elitismo

Por alguma razão, usando elitismo a performance caiu extremamente. Não consegui entender o motivo. Entretanto essa parte do trabalho fiz com certa pressa e posso não ter percebido algo óbvio ou ter feito alguma bobeira na hora de executar.

Pelos exemplos rodados (dá para ver pelos gráficos gerados), o desempenho caiu enormemente.

Outras Variações

Após realizar os experimentos pedidos, resolvi modificar mais de um parâmetro ao mesmo tempo para tentar chegar ao melhor resultado possível. Os gráficos referentes a essas modificações estão na pasta multi_parametros.

Depois de muito variar os parâmetros, cheguei à conclusão de que uma quantidade um conjunto bom de parâmeteros é ng = 1000, pi = 300+, pc = 80% e pm = 10%. Com parâmtros próximos desses, obtive bons resultados. O melhor deles apenas diferindo na população, com o valor 1000.

Também cheguei a implementar a ideia dada em aula de aumentar o fator de mutação quando a população fica estagnada. Isso ajudou a melhorar um pouco o resultado e conseguiu cumprir o papel de tirar a população de uma convergência para um máximo local e fazer ela continuar até outro máximo local com pontuação mais alta.

Usando essa técnica, consegui uma pontuação de 233 de um total de 243 pontos possíveis. O que possivelmente significa que o tabuleiro final tinha 3 casas erradas (uma vez que na minha função de fitness eu conto os erros por linha, coluna e quadrante - permitindo que uma casa errada contribua com até 3 erros).

2) Hill Climbing

Introdução

Inicialmente, achei que o Hill Climbing fosse obter um resultado pior do que o algoritmo genético. Entretanto, me surpreendi ao ver que o inverso aconteceu. Isso se deu principalmente pela forma como escrevi ambos os códigos. No Hill Climbing, diferentemente do genético, como apenas meus vizinhos são possibilidades de crescimento, eu tenho mais controle, como programador, de como selecionar o caminho de subida. A tática usada foi inicialmente começar uma uma matriz cujas linhas todas têm números de 1 a 9 sem repetição em uma ordem aletória. A cada passo, eu apenas permitia o algoritmo fazer um swap entre duas posições de uma mesma linha. Dessa forma, garanto que a integridade das linhas permanecerá ao longo da execução inteira. E portanto, o crescimento do hill climbing será bastante eficiente.

Realmente a prática comprovou essa teoria. Justamente por perder o controle da integridade das linhas no genético (devido ao crossover e mutações), o caminho até a solução não é tão bem controlado. Ao passo que no hill climbing esse controle é mantido e a progressão se mostrou consistente.

Mesmo assim, o Hill Climbing não conseguiu encontrar nenhuma solução. Conseguiu chegar bem próximo, mas não na solução.

Implementação

O algoritmo de Hill Climbing foi todo implementado por mim usando apenas o numpy como biblioteca externa.

Alterando o tabuleiro

Fiz as mesmas alterações do tabuleiro com o hill climbing para verificar mudanças. Os reslutados foram consistentes com o do algoritmo genético. Quando eliminava restrições - e consequentemente gerava mais máximos locais - o hill climbing ficava preso em um máximo

local com mais frequência. E adicionando restições, o hill climbing conseguiu chegar muito próximo da solução, superando todas as minhas espectativas para o desempenho desse algoritmo.

3) Execução

Para rodar o algoritmo de hill climbing, basta abrir o arquivo `hill_climbing.py` com o python (no meu computador eu sempre uso o python3, então não sei se existe alguma incompatibilidade com o python2.7). Para rodar o algoritmo genético é a mesma coisa, só que com o arquivo `genetic2.py`. Para rodar o genetic2.py é necessário baixar a biblioteca DEAP para python. Isso pode ser facilmente feito com `pip install deap` ou `pip3 install deap` dependendo da versão do python.

4) Observações

As imagens estão divididas por pastas com nomes auto explicativos referentes aos experimentos realizados. Pode acontecer de alguns títulos de gráficos estarem ligeiramente equivocados (erro meu na hora de fazer o gráfico). Entretanto a sua localização nas pastas descreve exatamente o seu conteúdo. Se houver divergência entre título e localização, confie mais na localização.