Introdução à Inteligência Artificial

2019/2020

Trabalho Prático 2

. . .

Leonardo Jorge Amaral Caldeira 2018011039 P5

Nuno Jerónimo Ferreira dos Santos 2018012623 P6

Índice

Introdução

Neste trabalho prático de Introdução à Inteligência Artificial, foi-nos apresentado um problema de otimização com o intuito de minimizar uma Bandwith, composta por N vértices, ligados entre si, formando uma rede complexa. Para isso, rotulamos cada vértice com um número único e aleatório (de 1 a N). Assim, e dadas as ligações entre os diversos nós conseguimos calcular o custo de cada ligação, |vn– vm|. O custo da Bandwith no geral equivale ao custo da maior ligação.

Para otimizar este custo, podemos alterar os rótulos dos nós de maneira a ter ligações menos pesadas, alterando, por exemplo, nós com rótulos maiores, ligados entre si para o custo não ser tão alto.

No entanto, para efetuar esta otimização, recorremos a métodos automáticos de otimização, que tentam através de trocas e cruzamentos chegar a soluções melhores que as iniciais.

Os métodos que utilizamos são o Trepa Colinas Normal e Probabilístico, o Evolutivo com reparação completa e incompleta, com penalização, e Híbrido, uma união entre evolutivo e trepa colinas.

Os programas são ainda compostos por um mecanismo de repetição de algoritmos. Cada repetição é chamada *run* e no fim é nos apresentado o melhor custo, solução e a média de custos.

Lógicas genéricas

Ao longo de todos os algoritmos, foram utilizados, quer para a leitura de dados, quer para dinâmica de soluções, funções comuns, ou que assentam na mesma lógica. Qualquer mudança será explicitamente anunciada nos tópicos dos algoritmos de otimização.

- Leitura de dados
  - Os dados são lidos de documentos texto que nos indicam primeiro o número de vértices e número de ligações. Seguidos pelas ligações em si, numa matriz de N:2. Esta matriz é interpretada na função e inserimos os dados numa matriz binária N:N, onde as ligações são representadas por 1.
  - Esta inserção consiste na leitura de dois números, nós ligados, e no respetivo *offset* da matriz, linha 1, coluna 2, por exemplo, colocamos o valor a *true.*

- Gerar soluções inicias
  - As soluções iniciais têm de conter números únicos de 1 a N, aleatórios na sua posição no *array*. Para isso, inicializamos um array de N posições ordenado.
  - Posteriormente, trocamos as posições aleatoriamente de lugar e utilizamos esse *array* para inicializar a nossa solução.
- Calcula custo
  - O custo é calculado através da matriz de adjacências. Ela é percorrida e se o elemento na posição x,y for *true*, vamos à posição x e y da solução e calculamos a diferença dos seus valores. Esta diferença é o custo da ligação. Permanece, como custo da BandWith o maior valor de custo.

```
int calcula_fit(int a[], int *mat, int vert)
{
        int custo=0;
        int aux=0;
        int offsetv=0;
        int i, j;
        //o custo da solucao equivale a diferenca entre duas ligacoes
        for(i=0; i<vert; i++)
        {
            for(int j= 0 ; j < vert ; j++)
            {
                offsetv = offset(i,j,vert);
                if(mat[offsetv] == 1)
                {
                    aux = abs(a[i]-a[j]);
                    if(aux > custo)
                        custo=aux;
                }
            }
        }

        return custo;
}
```

Figura 1- Código do calculo do custo, Trepa Colinas.

Os Algoritmos de Otimização

- ### Trepa Colinas

O **trepa colinas** é um algoritmo que pesquisa soluções nas imediações da sua solução atual. Se a solução vizinha for melhor é adotada.

Utilizamos ainda o **trepa colinas probabilístico**, que tem x probabilidade de adotar uma probabilidade uma solução pior, para evitar planaltos.

Passos do algoritmo:

1. Leitura de dados e construção da matriz de adjacências;
2. Construção da solução inicial;
3. Início do trepa colinas, efetua por *num_iter* vezes:
    a. Gera um vizinho;
        i. Troca duas posições da solução entre si;
    b. Calcula o seu custo
        i. Maior diferença entre custo de ligações;
    c. Se o custo for menor adota o vizinho como solução;
        i. Ou pode adotar um pior se for o método probabilístico;
    d. Continua o ciclo com a melhor solução;
4. Acabado o trepa colinas apresenta a solução com o menor custo;

```
int trepa_colinas(int sol[], int *mat, int vert, int num_iter)
{
    int *nova_sol, custo, custo_viz, i;


        nova_sol = malloc(sizeof(int)*vert);
    if(nova_sol == NULL)
    {
        printf("Erro na alocacao de memoria");
        exit(1);
    }
        // Avalia solucao inicial
    custo = calcula_fit(sol, mat, vert);
    for(i=0; i<num_iter; i++)
    {
                // Gera vizinho
                gera_vizinho(sol, nova_sol, vert);
                // Avalia vizinho
                custo_viz = calcula_fit(nova_sol, mat, vert);
                // Aceita vizinho se o custo diminuir (problema de minimizacao)
        if(custo_viz < custo)
        {
            substitui(sol, nova_sol, vert);
            custo = custo_viz;
            break;
        }
    }
    free(nova_sol);

    return custo;
}
```

Figura 2- Código do Trepa Colinas.

- **Evolutivo**

O Algoritmo Evolutivo parte de um conjunto, de número de elementos predefinidos, a população, cada individuo desta população tem N vértices, ou genes. Esta população passa por um conjunto de algoritmos, nomeadamente, crossover, mutação e torneio com intuito de encontrar na população uma solução ideal, passando de geração em geração (geração é o conjunto da população nos diferentes níveis de melhoramento).

  - Crossover - Dois progenitores cruzam as suas soluções de maneira a formar dois filhos, com metade de dados de cada um.
  - Torneio – O torneio permite a uma solução passar diretamente para a geração seguinte, e são sujeitos a crossover, se o seu custo for melhor que o custo da solução que compete consigo.
  - Mutação – A mutação consiste na alteração de valores numa solução, com a intenção de alteração para um possível melhoramento.

Passos do algoritmo:

1. Leitura dos dados e preenchimento das estruturas, campos importantes à execução do algoritmo, como probabilidades de mutação e cruzamento, construção da matriz de adjacências;
2. Construção da população inicial;
    a. Igual à do trepa colinas, no entanto multiplicado ao tamanho da população;
3. Início do algoritmo evolutivo, que se repete por *x* gerações:
    a. Inicia-se o torneio para decidir os melhores progenitores.
    b. Fazem-se as operações genéticas, mutação e crossover (com a probabilidades predefinidas);
        i. Na mutação adotamos o *swap* entre elementos para não invalidar a solução.
        ii. No crossover copiamos em integra os elementos do primeiro progenitor até um ponto aleatório do *array*, e na segunda metade colocamos os elementos do segundo pai; (Este algoritmo levantou problemas que levaram a adoção de um método de reparação inevitável que explicaremos posteriormente)
    c. Avalia-se a população:
        i. A população é avaliada individuo a individuo(solução a solução), determinando também validades;
    d. Guarda-se a melhor opção e contam-se os inválidos para estatística;
    e. Repetir;

Evolutivo - Reparação no *Crossover*

No algoritmo de *crossover*, o objetivo é copiar parte de dois progenitores para uma solução comum. No entanto, isto levanta problemas de repetição de elementos, algo que não pode acontecer no contexto do enunciado, e que, consequentemente, leva à invalidação da solução.

Originalmente, o nosso crossover copia ate ao ponto N os elementos do 1º pai diretamente, e passado esse ponto copiamos apenas os que ainda não constam na solução. Se na sequência do 2º pai houver um repetido, colocamos o numero daquela posição do 1º pai.

Por exemplo:

Pai 1 - 1 2 3 4 5

Pai 2 – 3 2 4 5 1

Filho – 1 2 3 5 5 ✖

Conseguimos evitar que o 2º pai não repita os genes do 1º, mas não que o 1º repita os do 2º, seria necessário um algoritmo muito mais complexo de implementação.

```
while(count != vert)
    {
       count =0;
        for(int k =0 ; k<vert ; k++)
        {
          if(checkVal[k] == 1) count++;                    //se for ==1(como desejado, sem repetições)
                                                             //aumenta um contador que serve para ver
                                                           //quando já não há mais inválidos
          if(checkVal[k] > 1) //se for maior que 1 repete se
             for(int j=0; j < vert; j++)                          //no solução procuramos o valor
                                                                //correspondente ao valor de k+1,
                                                              //aka a 1a vez que o valor repetido se repete
              if(a[j] == k+1){
                 for(int x=0; x<vert; x++)                       //para um repetir tem de haver um outro que não
                    if(checkVal[x]==0)                           // tem ocorrência procuramos o que não tem
ocorrência
                    {
                       a[j]=x+1;                                 //e colocamo-lo no lugar do repetido na solução
                        for(int k=0; k< vert; k++)checkVal[k] = 0;
                       for(int k=0; k< vert; k++) checkVal[a[k]-1] += 1; //atualiza o checkRep array
                       break;
                    }break;}
```

Figura 3- Ciclo reparador do crossover, na função, void adjustCrossover(int a[], int vert)

Apesar deste problema, aproveitamos a oportunidade e a partir desta falha implementamos o nosso sistema de **Reparação** e **Penalização**, cuja análise trataremos mais à frente neste relatório.

A reparação da solução centrou-se num algoritmo que consiste em:

1. Criar um *array*, denominado a partir de agora *checkVal*, com o tamanho da solução;
2. Dado o elemento, x, sol[i], da solução, incrementar a posição correspondente ao seu valor no *checkVal*;
3. Verificar se algum elemento do *checkVal* é diferente de 1. Ou seja, tem repetições.
4. Quando ocorre a repetição, guardamos através de um novo ciclo o elemento repetido.
5. Verificamos o *checkVal* para um elemento igual a 0, para ser acrescentado.
6. E acrescentamos o valor que tem 0 ocorrências no 1º elemento que se repete.

Evolutivo - Penalização do custo

Como já referimos, utilizamos esta oportunidade de ter soluções inválidas sob controlo, podendo decidir como penalizá-las.

Assim, decidimos punir as soluções com repetições na função do calculo do custo, que não apenas verifica os custo mas também a validade das soluções, com a mesma estratégia utilizada na função *void adjustCrossover(int a[], int vert)*, sem a parte de reparação obviamente. Às soluções invalidas são-lhe, portanto, atribuídas custos elevados e é-lhe ativada a *flag* de invalidade na sua estrutura individual.

• Híbrido

No algoritmo híbrido aplicamos Trepa Colinas e Evolutivo, no mesmo algoritmo. Utilizamos como base o algoritmo evolutivo e colocámos o trepa colinas em dois diferentes momentos, trabalhando independentemente um momento do outro, ou seja, num teste só aplicado num dos momentos. Mas também aplicamos em conjunto.

Estes momentos foram:

- Durante a inicialização da população inicial, para esta começar já otimizada pelo trepa coinas de início;
- Na solução final, depois de todo o algoritmo genético ter evoluído a população inicial;
- Nestas situações em conjunto.

Utilizamos o evolutivo com os parâmetros que obtiveram melhores resultados e o trepa colinas, para não sobrecarregar os dispositivos nas instâncias com mais vértices, utilizamos 200 iterações.

Análise de resultados

Dados a ter em consideração:

- Grafos:
  - Bscpwr01 – 39 nós; 85 arestas;
  - Bscpwr02 – 49 nós; 108 arestas;
  - Bscpwr03 – 118 nós; 297 arestas;
  - Dwt234– 234 nós; 534 arestas;
  - Will199– 199 nós; 701 arestas;

Os nossos testes tiveram como base as instâncias fornecidas pelos professores. "Corremos" cada algoritmo 20 vezes (20 runs), com parâmetros diferentes para retirar as seguintes elações:

Trepa Colinas

O nosso trepa colinas principal foi implementado para procurar vizinhos melhores durante as iterações, *it*, para tentar ao máximo encontrar a melhor solução possível.

(1º critério de vizinhança)

		100 it	1000 it	5000 it
bcspwr01	Melhor	22	18	15
	MBF	28,43	25,83	24,46
bcspwr02	Melhor	31	25	22
	MBF	37,73	34,8	31,63
bcspwr03	Melhor	92	85	80
	MBF	104,56	98,76	93,66
dwt234	Melhor	199	172	172
	MBF	211,83	202,83	200,6
will199	Melhor	178	173	166
	MBF	188,23	183	180

Figura 4- Testes do Trepa Colinas, nas várias instâncias e com diferentes números de iterações.

Perante os resultados apresentados, observamos a **qualidade da solução** (fitness, custo, assinalado por *Melhor*, no gráfico) é **inversamente proporcional ao número de iterações**. O que seria de esperar, tendo em conta que mais iterações permitem explorar um maior leque de opções de combinação de rótulos.

Podemos ainda retirar daqui que quantos **mais vértices** temos **maior** é a **diferença de qualidade** entre incremento de iterações.

Trepa Colinas probabilístico

Nesta vertente exploramos a qualidade da solução se saltássemos para uma solução pior satisfeita uma certa probabilidade. (2º critério de vizinhança)

	it	100	5000	100	5000
	prob	0,01	0,01	0,0005	0,0005
bcspwr01	Melhor	22	17	22	18
	MBF	28,93	22,86	29,1	24
bcspwr02	Melhor	29	23	29	24
	MBF	37,033	36,73	30,5	31,83
bcspwr03	Melhor	97	77	96	84
	MBF	104,76	89,1	104,133	104,36
dwt234	Melhor	200	167	200	180
	MBF	214,93	172,6	214,76	196,56
will199	Melhor	179	165	181	162
	MBF	187,93	176,9	188,8	178,56

Figura 5- Resultados dos testes do Trepa Colinas Probabilístico.

Em análise destes resultados podemos concluir, que novamente, **mais iterações significam melhor soluções.**

Observámos ainda que, contra intuitivamente, **por vezes sair da melhor solução atual tem vantagens**, pois permite sair de um ciclo vicioso, um pico não máximo. Muitas vezes **esta passagem** para uma qualidade pior **nem sempre valoriza** e temos então valores muito díspares de relação, porque foi atribuída uma qualidade muito pior à solução, disparando as médias, no sentido errado.

Comparando com os testes onde não havia probabilidade de piorar, podemos então confirmar estas especulações.

Exemplifique-se:

- MBF_prob_inst bcspwr01_5000it_0.01 = 22.86;
  MBF_5000it=24.46;
  Melhorou a média em 20 *runs.*

- MBF_prob_inst bcspwr03_5000it_0.005 = 104.36;
  MBF_5000it=93.66;
  Piorou a média em 20 *runs.*

Conclusão: Verifica-se pontualmente um melhoramento.

Evolutivo

Nos testes do evolutivo fizemos na mesma as 20 runs para cada instância, com exceção da duas mais pesadas, porque não conseguimos acabar a execução.

Para estes testes diversificamos vários parâmetros probabilísticos para analisar qual a melhor alternativa e combinação para crossover e mutação, por exemplo.

Modelo A - Reparação completa em crossover e mutação por *swap*. Torneio de tamanho 2.

		bcspwr01		bcspwr02		bcspwr03		
Fixed		Best	MBF	Best	MBF	Best	MBF	
ger = 2500	pr=0.3	20	21,4	26	28,45	85	87,06	Media MBFs
pop = 100	pr=0.5	18	21,25	27	28,85	86	87,53	98,862 *
pm= 0.01	pr=0.7	20	21,5	26	28,55	85	87	99,686
								98,955

		bcspwr01		bcspwr02		bcspwr03		
Fixed		Best	MBF	Best	MBF	Best	MBF	
ger = 2500	pm=0.0	24	26,8	29	33,9	93	96,4	Media MBFs
pop = 100	pm=0.001	22	22,9	28	30,05	87	89,8	107,57
pr= 0.3	pm=0.01	20	21,15	25	28,45	86	88,1	101,43
	pm=0.05	18	20,65	26	27,9	84	86,9	98,84
	pm=0.1	19	20,65	26	28,05	85	87,1	98,51 *
								98,7

			bcspwr01		bcspwr02		bcspwr03		
Fixed			Best	MBF	Best	MBF	Best	MBF	
	25k ger.	pop=10	19	20,7	27	27,85	84	87,5	Media MBFs
pm=0.05	5k ger.	pop=50	19	20,7	26	28,05	86	87,125	98,81
pr= 0.3	2,5k ger.	pop=100	18	20,65	26	27,9	84	86,9	98,975
									98,51 *

Figura 6-Testes das 3 primeiras instâncias, seguindo o modelo A. E média de MBF de todas as instâncias.

Como podemos observar por estes dados, quanto **menor a probabilidade de crossover, menor serão os custos em média**, apesar de ser uns valores muito próximos decidimos envergar por aquilo que testámos e não por aquilo que pensávamos ser o caminho certo.

Na **mutação é imediato o sucesso da solução** com o seu aumento, estagnando este grau de melhoramento algures entre 0.05 e 0.1 de probabilidade.

Na última tabela concluímos finalmente que a **qualidade da solução não está relacionada com a quantidade de indivíduos**:

- 25k ger. * 10 = 250,000 indivíduos.
- 5k ger. * 50 = 25,000 indivíduos.
- 2,5k ger. * 100 = 25,000 indivíduos.

Apesar a disparidade de 250k para 25k não se observou diferença significativa.

Modela B - Reparação incompleta no crossover(prob. de pRep% de reparação), para avaliação de penalização.

			bcspwr01			bcspwr02		
Fixed			Best	MBF	Inv. (%)	Best	MBF	Inv. (%)
2,5k ger.	pop=100	pRep.=0.3	20	21,5	63,25	28	28,5	60,96
pm=0.05		pRep.=0.5	20	21,05	32,4	27	32,7	28,5
pr= 0.3		pRep.=0.7	19	20,5	11,55	25	27,85	14,3

Figura 7- Testes do modelo B, nas duas primeiras instâncias.

No modelo B, tomamos partido da situação referida atrás e não reparamos totalmente o crossover. Aplicamos como penalização um elevado custo às soluções inválidas.

A melhores soluções apresentadas, são as da população inicial por vezes, na probabilidade menor de reparação.

Podemos ainda observar que a **média de custos aumenta quando a probabilidade de reparação é menor**.

Em matérias comparativas revela-se uma **clara desvantagem de um algoritmo penalizado perante um algoritmo totalmente reparado**.

Híbrido

			bcspwr01		bcspwr02		bcspwr03		dwt234		will199	
Fixed			Best	MBF	Best	MBF	Best	MBF	Best	MBF	Best	MBF
2,5k ger.		Met.1	17	19,35	22	26,4	80	85,5	181	185,5	166	170,75
pm=0.05												
pr= 0.3		Met.2	19	20,6	25	27,9	85	86,9	185	187,25	171	171,875
pop=100												
		Met.3	17	18,5	23	26	80	84,9	181	185,37	169	171,5

Figura 8- Testes Do algoritmo Hibrido.

Método 1- Trepa Colinas na população inicial, de 200it.

Método 2 - Trepa Colinas na solução final, de 200it.

Método 3 - Trepa Colinas na pop inicial e na solução final.

No hibrido adotamos as melhores opções tomadas no evolutivo e 200 iterações de trepa colinas para não ficar muito pesado e acabar por não concluir a execução.

No algoritmo hibrido torna-se claro o poder destes algoritmos combinados. Podemos por exemplo analisar a instância will199 que no trepa colinas em 5000 iterações teve MBF de 180, e na combinação não saiu da casa dos 170.

O **modelo 1** é claramente o mais poderoso, ou seja, aplicar o trepa colinas à população inicial**, começa logo os indivíduos em vantagem**.

O **modelo 2**, é apresenta também um melhoramento perante os dois algoritmos trabalhando sozinhos, **mas não demonstra qualquer vantagem sobre o método 1**.

Para concluir, coloquei **os dois métodos a trabalhar em conjunto**, o que produziu as **melhores médias de todo o projeto**.

Análise Geral dos algoritmos e Conclusão

Os algoritmos explorados no trabalho prático demonstraram-se capazes de melhorar uma solução. No entanto, os dados que tiramos deles não foram da nossa perspetiva muito conclusivos, retiramos as elações que estaríamos à espera.

Vários testes pareceram encaixar sempre na mesma solução e custo, mesmo mudando os parâmetros. Acreditamos que será parcialmente culpa do facto que para alguns testes 20 ou 40 runs não são suficientes para ter resultados determinísticos.

Deparamo-nos então com esse problema de tirar conclusões exatas nalgumas etapas deste projeto.

Apesar de tudo, fomos capazes de tirar algumas conclusões concretas.

O **trepa colinas** é um ótimo algoritmo para pesquisar soluções locais, é apesar disso, muito dispendioso, na medida, em que são necessárias imensas iterações para obter os melhores resultados possíveis. Podemos ainda combiná-lo com um algoritmo regressivo que aceita piores custos e que possibilita a saída de um falso pico máximo.

No **algoritmo genético** podemos concluir que a **mutação é fundamental** à **despadronização de resultados obtidos do crossover**. O crossover forma a partir de uma geração determinados padrões que se propagam por gerações e gerações que são dificilmente quebrados sem um mecanismo de mutação.

No **algoritmo híbrido**, concluímos que **partir de uma solução relativamente otimizada** é a maneira mais correta de proceder.