Utilização de Heurísticas em Algoritmos

Abner Gomes Guimarães Felipe Araujo Correia

Trabalho Interdisciplinar de Pesquisa Experimental

June 4, 2024

Conteúdos

Introdução

O problema principal que vamos analisar e definir suas soluções é o problema do caixeiro viajante (PCV). O caixeiro viajante é um problema de otimização combinatória que é intensamente investigado em matemática computacional devido à sua vasta área de aplicação e complexidade de obtenção de uma solução real.

Grafo para o Problema do Caixeiro Viajante

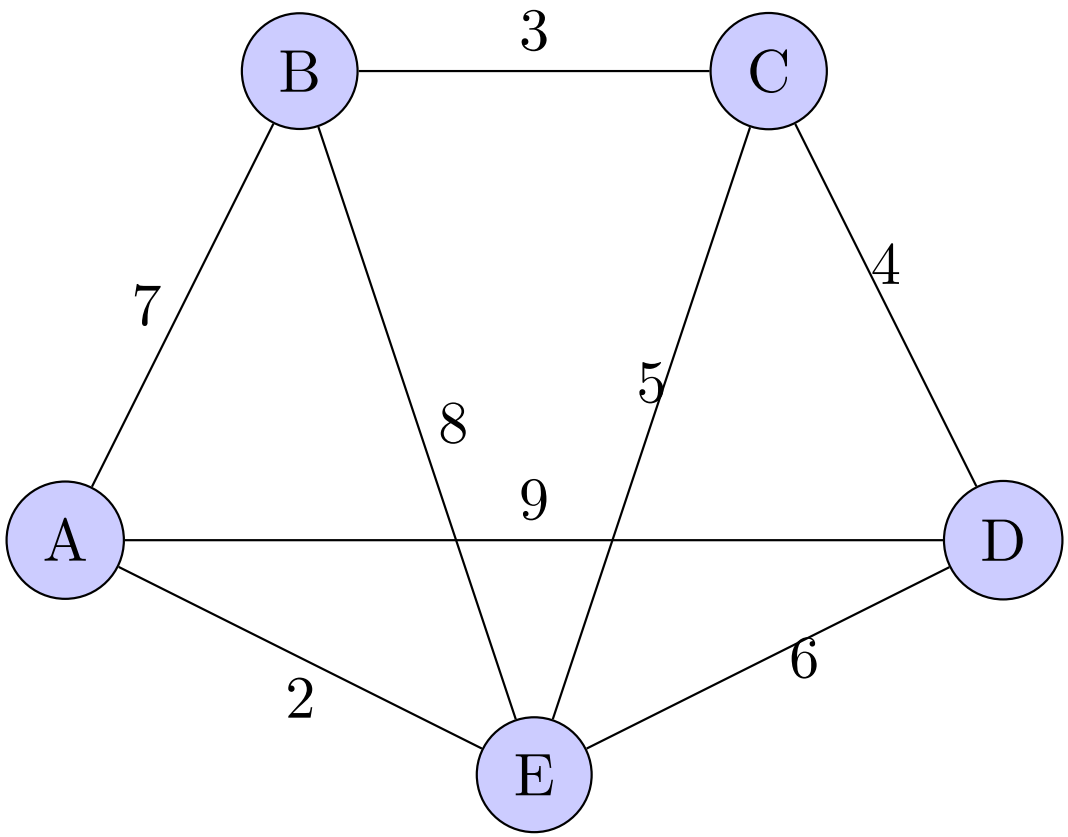

Figure: Exemplo de grafo para o Problema do Caixeiro Viajante

Geração de Grafos Completos Aleatórios

Para nossos experimentos, utilizamos um algoritmo para criar grafos completos aleatórios. Em um grafo completo, todos os vértices estão conectados entre si. Os pesos das arestas são atribuídos aleatoriamente dentro de um intervalo especificado.

Algoritmo de Grafo Aleatório

```
Grafo Criacao_grafo_aleatorio(int n, int m, int PesoMaximo) {
  Grafo G = iniciar_Grafo(n);
  while (G->QuantidadeArestas < m) {
    int v = vertice_aleatorio(G);
    int w = vertice_aleatorio(G);
    if (v != w) {
      int peso = rand() % PesoMaximo + 1;
      insere_aresta(G, v, w, peso);
    }
  }
  return G;
}
```

Figure: Algoritmo de criação de grafo aleatorio

Desenvolvimento

Neste capítulo, explicaremos as heurísticas e processos evolutivos utilizados para resolver o Problema do Caixeiro Viajante (PCV). Utilizamos uma abordagem mista que combina heurísticas construtivas, heurísticas de busca local e meta-heurísticas. Começaremos explicando a geração de grafos completos aleatórios, seguida pela descrição do método de Descida em Vizinhança Variável (VND) com a técnica de 2-opt, a heurística construtiva aleatória, o algoritmo guloso aleatório, e o algoritmo genético com seleção por torneio.

Heurística Construtiva Aleatória

A heurística construtiva aleatória que utilizamos começa selecionando vértices aleatórios do grafo de forma que eles não se repitam até que todos os vértices sejam visitados. Esta abordagem simples gera uma solução inicial que pode ser melhorada posteriormente com técnicas de busca local e meta-heurísticas.

Algoritmo Heurítica Contrutiva

```
void construcao(Grafo g, int *solucao){
  int n = g->MaxNumeroVertices;

  int cidadeInicial = vertice_aleatorio(g);
  solucao[0] = cidadeInicial;

  for(int i = 1; i < n; i++){
    solucao[i] = vertice_aleatorio(g);
    for(int j = 0; j < i; j++){
      if (solucao[i] == solucao[j]){
        i--;
      }
    }
  }

}
```

Figure: Heurítica Construtiva

Descida em Vizinhança Variável (VND) com 2-opt

A VND é uma técnica que explora múltiplas vizinhanças para evitar mínimos locais. Usamos a técnica de 2-opt, que envolve trocar duas arestas em uma solução atual para obter uma nova solução com um caminho potencialmente menor.

VND

```c
void VND(Grafo g, int* solucaoAtual, int r) {
    clock_t start, end;
    double cpuTime;

    start = clock();
    int n = g->MaxNumeroVertices;
    int solucao[n];
    int melhora = 0;

    for(int iteracao = 0; iteracao < r; iteracao++) {
        for(int i = 0; i < n - 1; i++) {
            for(int j= i + 1; j < n; j++) {
                replicaSolucao(solucaoAtual, solucao, n);
                twoOptSwap(solucao, i, j, n);
                if(calcDist(g, solucao) < calcDist(g, solucaoAtual)) {
                    replicaSolucao(solucao, solucaoAtual, n);
                    melhora = 1;
                }
            }
        }
        if (!melhora) break;
    }
    end = clock();
    cpuTime = ((double) (end - start)) *1000 / CLOCKS_PER_SEC;
    printf("Tempo de Execução (VND) em segundos: %f \n", cpuTime);
}
```

Figure: Descida em Vizinhança Variável

Algoritmo Genético

Os algoritmos genéticos (AG) são inspirados pela evolução biológica. Usamos um método de seleção por torneio, onde um subconjunto da população é escolhido aleatoriamente e o melhor indivíduo do subconjunto é selecionado para a reprodução.

Algoritmo Genético

```c
clock_t start, end;
double cpuTime;

start = clock();

int n = g->MaxNumeroVertices;
int **populacao = malloc(tamPop * sizeof(int *));
int *fitness = malloc(tamPop * sizeof(int));
int *novaPopulacao[tamPop];

for (int i = 0; i < tamPop; i++) {
  populacao[i] = malloc(n * sizeof(int));
  novaPopulacao[i] = malloc(n * sizeof(int));
}

inicializaPopulacao(g, populacao, tamPop);
avaliarPopulacao(g, populacao, fitness, tamPop);

for (int geracao = 0; geracao < numGeracoes; geracao++) {
  for (int i = 0; i < tamPop; i++) {
    int pai1 = torneio(fitness, tamPop);
    int pai2 = torneio(fitness, tamPop);

    if ((double)rand() / RAND_MAX < taxaCrossover) {
      crossover(populacao[pai1], populacao[pai2], novaPopulacao[i], n);
    } else {
      replicaSolucao(populacao[pai1], novaPopulacao[i], n);
    }

    if ((double)rand() / RAND_MAX < taxaMutacao) {
      mutacao(novaPopulacao[i], n);
    }
  }

  for (int i = 0; i < tamPop; i++) {
    replicaSolucao(novaPopulacao[i], populacao[i], n);
  }
```

Continuação

```
    avaliarPopulacao(g, populacao, fitness, tamPop);
  }

  int melhorIndice = 0;
  for (int i = 1; i < tamPop; i++) {
    if (fitness[i] < fitness[melhorIndice]) {
      melhorIndice = i;
    }
  }

  end = clock();
  cpuTime = ((double) (end - start)) *1000 / CLOCKS_PER_SEC;
  printf("Tempo de Execução (AG) em segundos: %f \n", cpuTime);
  //termina aqui
  replicaSolucao(populacao[melhorIndice], melhorSolucao, n);
  for (int i = 0; i < tamPop; i++) {
    free(populacao[i]);
    free(novaPopulacao[i]);
  }
  free(populacao);
  free(fitness);
}
```

Figure: Algoritmo Genético

Conclusão

Concluímos que as heurísticas e meta-heurísticas são ferramentas valiosas para resolver o Problema do Caixeiro Viajante (PCV). Durante nossa análise, observamos que as heurísticas construtivas, como algoritmos gulosos e heurísticas aleatórias, são eficazes para gerar soluções iniciais, enquanto as meta-heurísticas, como algoritmos genéticos e busca local variável, são úteis para refinar essas soluções e encontrar soluções de melhor qualidade.