A C++ Library for Developing Evolutionary Algorithms to the QCaRS Problem

Tiago Funk[1], **Fernando Santos**[1]

[1]Departamento de Engenharia de Software
Universidade do Estado de Santa Catarina (UDESC)
Campus Alto Vale – Ibirama – SC – Brasil

tiagoff.tf@gmail.com, fernando.santos@udesc.br

Abstract. *The quota traveling car renter problem (QCaRS) consists of minimizing the cost of a trip between a group of cities, visiting only a subgroup, and ensuring the visit of cities that meet with minimum satisfaction for the traveler. Evolutionary algorithms are methods often used to solve the QCaRS problem. Reproducing the results obtained by existing algorithms in the literature can be difficult if the authors did not make their implementations available. This paper presents a C++ library for the development of evolutionary algorithms applied to QCaRS. The library uses the Strategy design pattern to provide the exchange of components of evolutionary algorithms and thus facilitate the evaluation of new solution proposals for the QCaRS problem.*

Resumo. *O problema do caixeiro viajante com quota (QCaRS) consiste em minimizar o custo de uma viagem entre um grupo de cidades, visitando apenas um subgrupo, e garantindo a visita de cidades que cumpram com uma satisfação mínima para o viajante. Algoritmos evolucionários são métodos frequentemente utilizados para resolver o problema QCaRS. No entanto, reproduzir os resultados obtidos por algoritmos existentes na literatura pode ser difícil caso os autores não disponibilizaram suas implementações. Este artigo apresenta uma biblioteca C++ para desenvolvimento de algoritmos evolucionários aplicados no QCaRS. A biblioteca utiliza o padrão de projeto Strategy para proporcionar o intercâmbio de componentes dos algoritmos evolucionários e deste modo facilitar a avaliação de novas propostas de solução para o QCaRS.*

1. Introdução

A utilização de um algoritmo que elabora um caminho ótimo entre pontos em um mapa com custo de tempo baixo pode impactar a vida das pessoas. Por exemplo, uma empresa turística que oferece roteiros customizados para seus clientes. Se este roteiro for construído de forma rápida e fácil, a empresa terá vantagem competitiva. Um outro exemplo é um funcionário que precisa visitar filiais da empresa. A rápida criação de um roteiro que minimize o deslocamento permite iniciar a viagem o mais breve possível.

Este trabalho é sobre o problema do Caixeiro Alugador com Quota – do inglês, *Quota Car Traveling Salesman (QCaRS)*. O problema consiste em visitar um subgrupo entre um grupo de cidades, minimizando o custo com a viagem. As cidades do subgrupo visitado e o tamanho deste subgrupo dependem de uma quota que cada cidade possui. A quota é um valor numérico que indica o quão boa para o visitante é visitar a cidade. A

rota criada deve atender um valor mínimo que o viajante deseja obter ao visitar todas as cidades do subgrupo escolhido. O valor minimizado são os custos com a viagem entre as cidades, custos de aluguel de veículos, e multas por devolver veículos em cidades diferentes de onde foi alugado [Goldbarg et al. 2016].

Reproduzir os resultados obtidos por algoritmos propostos para o problema QCaRS pode ser uma tarefa difícil caso os autores não tenham disponibilizado o código fonte de suas implementações. Replicar experimentos é uma parte fundamental da ciência e trabalhos que não apresentam o código fonte para consulta dificultam que seus experimentos sejam replicados e validados. Sem códigos fonte, é necessário implementar os algoritmos do zero, o que pode introduzir erros de implementação. Além disso, disponibilizar os códigos e os experimentos é importante para a transparência do método científico. Até o momento, não há implementação publicamente disponível do problema QCaRS, dificultando a execução de experimentos e comparação com novos algoritmos.

Neste trabalho propomos uma biblioteca C++ com interface de implementação para o problema QCaRS. A biblioteca visa facilitar que outros autores que forem trabalhar com o problema QCaRS possam focar na implementação de seus algoritmos e não precisem se preocupar por exemplo com a leitura de instâncias e estruturas para representação dos dados em memória. A biblioteca se baseia em algoritmos evolucionários. Um algoritmo evolucionário utiliza uma população de soluções e realiza iterações sobre esta população até chegar em soluções satisfatórias [Linden 2008]. A biblioteca também tem a intenção de facilitar inserções ou remoções de componentes dos algoritmos evolucionários. Para isto, utiliza o padrão de projeto *Strategy*, que facilita a troca de componentes de um algoritmo [Gamma et al. 2000]. Embora bibliotecas para algoritmos genéticos estejam disponíveis, o diferencial da biblioteca proposta é ser focada no problema QCaRS, fornecendo as facilidades já mencionadas.

Este artigo está organizado da seguinte forma. Na Seção 2 é apresentada a fundamentação teórica, onde é descrita a formalização do problema QCaRS, a formalização dos conceitos de algoritmos evolucionários e apresentação do padrão de projeto *Strategy*. Na Seção 3 é apresentada a biblioteca C++ proposta neste trabalho. Por fim, a Seção 4 apresenta as conclusões e trabalhos futuros.

2. Fundamentação Teórica

Esta seção apresenta a base teórica, contextualizando os assuntos para uma melhor entendimento do problema QCaRS e as técnicas utilizadas no desenvolvimento. Serão apresentados os seguintes tópicos; fundamentação teórica do problema QCaRS; formalização dos conceitos de algoritmos evolucionários; e apresentação do padrão de projeto *Strategy*.

2.1. O Problema QCaRS

O problema QCaRS envolve criar uma rota entre um conjunto de cidades, com a restrição de minimizar os custos da viagem sempre realizando o aluguel de veículos. Estes custos são o aluguel de carros, custos com gasolina, pedágios e multas caso seja realizada uma devolução de veículo fora da cidade do aluguel original. Outra restrição é garantir um valor mínimo de satisfação. Cada cidade possui um valor numérico chamado *quota*. O roteiro possui um valor de satisfação, que é a soma da *quota* de todas as cidades que pertencem na rota. A seguir é apresentada a formulação matemática do problema.

Seja um grafo completo $G = (V, A)$, onde V é um conjunto com n nós simbolizando as cidades e A é um conjunto de arcos simbolizando as estradas entre as cidades. Existe também um conjunto C de veículos que está disponível para aluguel. Cada veículo possui seus gastos já citados. O objetivo no QCaRS é completar um caminho hamiltoniano em G pagando o mínimo possível com relação aos custos com os veículos. O QCaRS é um problema NP-Difícil [Menezes et al. 2017].

As características do problema QCaRS são: não existe limitação de deslocamento entre cidades; é possível ir de uma cidade para qualquer outra cidade; qualquer carro pode ser alugado em qualquer cidade; o carro alugado pode ser devolvido em qualquer cidade; cada carro só pode ser alugado uma vez; os custos de utilizar um carro para ir de cidade a para cidade b é igual ao valor de ir da cidade b até a cidade a; um carro devolvido em uma cidade diferente daquela que foi alugado implica uma multa a ser paga [Menezes et al. 2017]. A definição formal e o modelo matemático do problema QCaRS são apresentados em [da Silva Menezes et al. 2014] e atualizados em [Goldbarg et al. 2016]. As equações 1 a 14 apresentam o modelo matemático do QCaRS.

$$\text{Minimize: } \sum_{c \in C} \sum_{i,j \in V} d_{ij}^c * f_{ij}^c + \sum_{c \in C} \sum_{i,j \in V} y_{ij}^c * w_{ij}^c \tag{1}$$

$$\text{Sujeito a: } \sum_{c \in C} \sum_{j \in V} f_{1j}^c = \sum_{c \in C} \sum_{i \in V} f_{i1}^c = 1 \tag{2}$$

$$\sum_{c \in C} \sum_{i \in V} f_{ih}^c = \sum_{c \in C} \sum_{j \in V} f_{hj}^c \leq 1 \; \forall h \in V \tag{3}$$

$$a_i^c = \left(\sum_{j \in V} f_{ij}^c \right) \left(\sum_{c' \in C c' \neq c} \sum_{h \in V} f_{hi}^{c'} \right) \; \forall c \in C, i \in V, i > 1 \tag{4}$$

$$e_i^c = \left(\sum_{j \in V} f_{ji}^c \right) \left(\sum_{c' \in C c' \neq c} \sum_{h \in V} f_{ih}^{c'} \right) \; \forall c \in C, i \in V, i > 1 \tag{5}$$

$$w_{ij}^c = a_j^c * e_i^c \; \forall c \in C, \forall i, j \in V \tag{6}$$

$$\sum_{c \in C} a_1^c = 1 \tag{7}$$

$$\sum_{i \in V} a_i^c \leq 1 \; \forall c \in C \tag{8}$$

$$\sum_{i \in V} a_i^c = \sum_{i \in V} e_i^c \; \forall c \in C \tag{9}$$

$$\sum_{i \in V} \left(\left(\sum_{c \in C} \sum_{j \in V} f_{ij}^c \right) q_i \right) \geq \omega \tag{10}$$

$$2 \leq u_i \leq n \; \forall i = 2, \ldots, n \tag{11}$$

$$u_i - u_j + 1 \leq (n-1)(1 - \sum_{c \in C} f_{ij}^c) \; \forall i, j = 2, \ldots, n \tag{12}$$

$$f_{ij}^c * w_{ij}^c * a_i^c * e_i^c \in [0, 1] \tag{13}$$

$$u_i \in \mathbb{N} \tag{14}$$

A variável d_{ij}^c representa o custo de usar o carro $c \in C$ para ir da cidade i até a cidade j. Um custo y_{ij}^c deve ser pago caso um carro c seja alugado em uma cidade i e entregue em j, onde as cidades i e j são diferentes. A variável u_i indica a posição na rota. As seguintes variáveis são binárias: f_{ij}^c recebe valor 1 para o caso que o carro c viaja da cidade i para a cidade j e valor 0 caso contrário; w_{ij}^c representa se o carro c é alugado na cidade j e entregue na cidade i; a variável a_i^c representa se o carro c é alugado na cidade i; e a variável e_i^c representa se o carro c foi entregue na cidade i [Goldbarg et al. 2016].

O primeiro termo da função objetivo (1) expressa o custo das cidades visitadas e o segundo termo é o custo das multas pelos carros devolvidos em cidades diferentes daquelas em que foram pegos. A cláusula (2) obriga que a cidade 1 é a cidade que a rota deve começar e terminar. A cláusula (3) afirma que cada vértice deve ser visitado no máximo uma vez e se um carro chega ao vértice i, então um carro deve deixar esse vértice. A cláusula (4) restringe se o carro c foi alugado na cidade i. A cláusula (5) limita se o carro foi entregue na cidade i. A cláusula (6) delimita se um carro foi alugado em uma cidade e entregue em outra. A cláusula (7) obriga que um carro seja alugado na cidade de início da rota. A cláusula (8) limita que um carro seja alugado apenas uma vez. A cláusula (9) obriga que um carro alugado seja entregue. A cláusula (10) obriga que o mínimo de quota seja coletado. As cláusulas (11) e (12) evitam sub-rotas. A cláusula (13) indica que as variáveis são binárias e a cláusula (14) indica que a variável u_i é um inteiro positivo.

O formato utilizado para armazenar instâncias de problemas QCaRS em arquivos é uma extensão do formato TSPLIB [Reinelt 1995]. O formato TSPLIB é utilizado na comunidade de otimização para representar instâncias do problema do caixeiro viajante e relacionados. O formato estendido, proposto por [Asconavieta 2011], inclui dados adicionais para representar os detalhes específicos do QCaRS.

Cada arquivo de instância, que possui a extensão *pcar*, é dividido em duas seções: especificação e dados. A seção de especificação contém dados sobre o tipo de problema e suas características. Estes dados são organizados no formato *<palavra-chave> : <valor>*. As seguintes palavras-chave são encontradas nesta seção:

- *NAME:* nome da instância, por exemplo, *Mali11n.pcar*.
- *TYPE:* tipo dos dados da instância, que no caso do QCaRS é *CaRS*.
- *COMMENT:* comentário sobre a instância.
- *DIMENSION:* número de vértices do grafo, que no QCaRS é o número de cidades.
- *CARS_NUMBER:* número de carros do problema.
- *EDGE_WEIGHT_TYPE:* indica como valores de custos das arestas são representados. O valor *EXPLICT* indica que os custos estão explicitamente demonstrados na instância. Outro possível valor é o *EUC_2D*, e indica que os valores são representados em um vetor e ao ler o arquivo é realizando um cálculo.
- *EDGE_WEIGHT_FORMAT:* especifica o formato de apresentação dos custos das arestas. O valor *VECTOR* indica que os custos são apresentados de forma compacta e deve-se aplicar uma fórmula para obter os valores. O valor *EUC_2D* indica que as coordenadas dos vértices estão representados em coordenadas de um plano Euclidiano de 2 dimensões. Por fim, *FULL_MATRIX* indica que os custos são mostrados em forma de matriz, sem necessidade de aplicação de fórmulas.

As instâncias utilizadas são divididas em dois grupos: instâncias euclidianas e

instâncias não euclidianas. As instâncias representam coordenadas de cidades no mapa. Nas instâncias euclidianas, a distância entre cidades é calculada pela fórmula da distância euclidiana entre dois pontos. A distância entre cidades nas instâncias não euclidianas é calculada de forma diferente, pois nestas instâncias considera-se a curvatura do planeta e são utilizadas coordenadas gaussianas [Wolfe 2012].

A seção de dados contém os valores das instâncias. Esses valores são os valores de custo de viagem entre as cidades, custos de multa e valor de satisfação. As seguintes palavras-chave são encontradas nesta seção:

- *EDGE_COORD_SECTION:* seção que apresenta as matrizes com os valores que representam os custos das viagens entre cidades. Cada matriz representa um carro com seus respectivos custos.
- *RETURN_RATE_SECTION:* representa o custo de multa caso um veículo seja devolvido fora da cidade de origem do aluguel. Cada matriz representa um veículo.
- *BONUS_SATISFACTION_SECTION:* um vetor com o valor numérico de satisfação ao visitar cada cidade da instância.

2.2. Algoritmos Evolucionários

Algoritmos evolucionários são uma categoria de metaheurísticas baseadas em população. O seu foco é realizar melhorias em uma população de possíveis soluções. Essas melhorias são realizadas enquanto um critério de parada não for atingido. Este critério pode ser um número de gerações ou uma aptidão média da população [Talbi 2009].

Os algoritmos evolucionários representam uma classe de algoritmos que simulam a evolução das espécies, e utilizam a noção de competição. Um algoritmo evolucionário possui uma população de soluções, e a cada iteração são realizadas transformações nos indivíduos. Para criar a população pode ser utilizado algum procedimento ou heurística que for melhor para o problema. Cada indivíduo representa uma solução, e sua aptidão representa o valor que a função objetivo retorna ao ser submetida ao indivíduo.

Partindo da população inicial, um algoritmo evolucionário realiza iterações sobre os indivíduos. Em cada iteração é calculada a aptidão, feita a seleção dos indivíduos e realizado o cruzamento e mutação. No cálculo da aptidão, cada indivíduo é submetido ao cálculo de sua aptidão utilizando a função objetivo. Na seleção, são escolhidos indivíduos da população para reprodução. Na reprodução, indivíduos são selecionados par a par para trocar suas características e gerar novos indivíduos com uma mescla de características de ambos. Esses novos indivíduos são chamados de prole. A prole também passa pelo processo de calculo de aptidão. Por último ocorre a substituição, quando indivíduos com melhor aptidão substituirão indivíduos da população original com pior aptidão [Talbi 2009]. O *template* de um algoritmo evolucionário é mostrado na figura 1.

Na linha 1 no algoritmo é onde ocorre a geração da população. Entre a linha 2 e 8 está o laço de repetição, que para apenas quando o critério de parada for satisfeito. Na linha 3 ocorre o cálculo de aptidão. Na linha 4 ocorre a seleção dos indivíduos da população para reprodução. Na linha 5 ocorre a reprodução. Na linha 6 é calculada a aptidão da prole. Na linha 7 é realizada a substituição na população. A prole é inserida e os piores indivíduos da população original são removidos.

1: $P_0 = gerarPopulacao()$/* Gera a população inicial*/
2: **while** Critério de parada for satisfeito **do**
3: $avaliacao(P_i)$ /*Calcula a aptidão dos indivíduos*/
4: $P'_i = selecao(P_i)$ /*Seleciona indivíduos para reprodução*/
5: $P'_i = reproducao(P'_i)$ /*Realiza o cruzamento e mutação*/
6: $avaliacao(P'_i)$ /*Calcula a aptidão da prole*/
7: $P_{(i+1)} = substituicao(P_i, P'_i)$/*Insere os indivíduos da prole na população*/
8: **end while**

Figura 1. Template de um algoritmo evolucionário. Adaptado de [Talbi 2009]

2.3. Padrão de Projeto *Strategy*

Padrões de projeto são descrições de objetos e classes que se comunicam com a finalidade de resolver um problema geral de projeto em um contexto particular [Gamma et al. 2000]. Um padrão de projeto identifica as classes e papéis e distribui responsabilidades entre eles. Cada padrão resolve um problema recorrente do projeto orientado a objetos. Baseiam-se em soluções reais que foram implementadas nas principais linguagens de programação orientadas a objetos.

Dentre os padrões de projeto apresentados por [Gamma et al. 2000] está o *Strategy*. Este padrão faz parte do grupo dos padrões comportamentais, que se preocuparam com algoritmos e a atribuição de responsabilidades entre objetos envolvidos em algoritmos. Não descrevem apenas objetos ou classes, mas também a comunicação entre eles, visando afastar o foco do fluxo de controle para permitir apenas se concentrar em como os objetos são interconectados [Gamma et al. 2000].

O padrão *Strategy* tem a intenção de definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis [Gamma et al. 2000]. A figura 2 apresenta um diagrama de classes com a estrutura geral deste padrão de projeto. A interface *Strategy*, que é o núcleo deste padrão de projeto, disponibiliza a assinatura da função *AlgorithmInterface* que deve ser implementada pelas estratégias concretas. As classes *ConcreteStrategyA*, *ConcreteStrategyB* e *ConcreteStrategyC* são as implementações. Cada uma implementa o método *AlgorithmInterface* de acordo com a necessidade ou particularidade. A classe *Context* depende apenas da interface *Strategy*. Em tempo de execução, qualquer objeto que implemente *Strategy* poderá ser fornecido para *Context*, que portanto fica independente de qualquer implementação concreta.

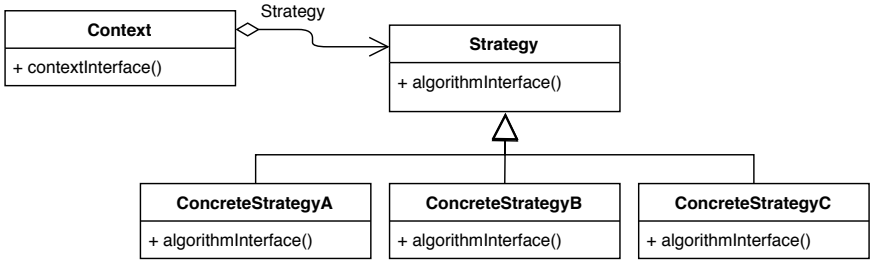

Figura 2. Diagrama de classes do padrão *Strategy*. Fonte: [Gamma et al. 2000]

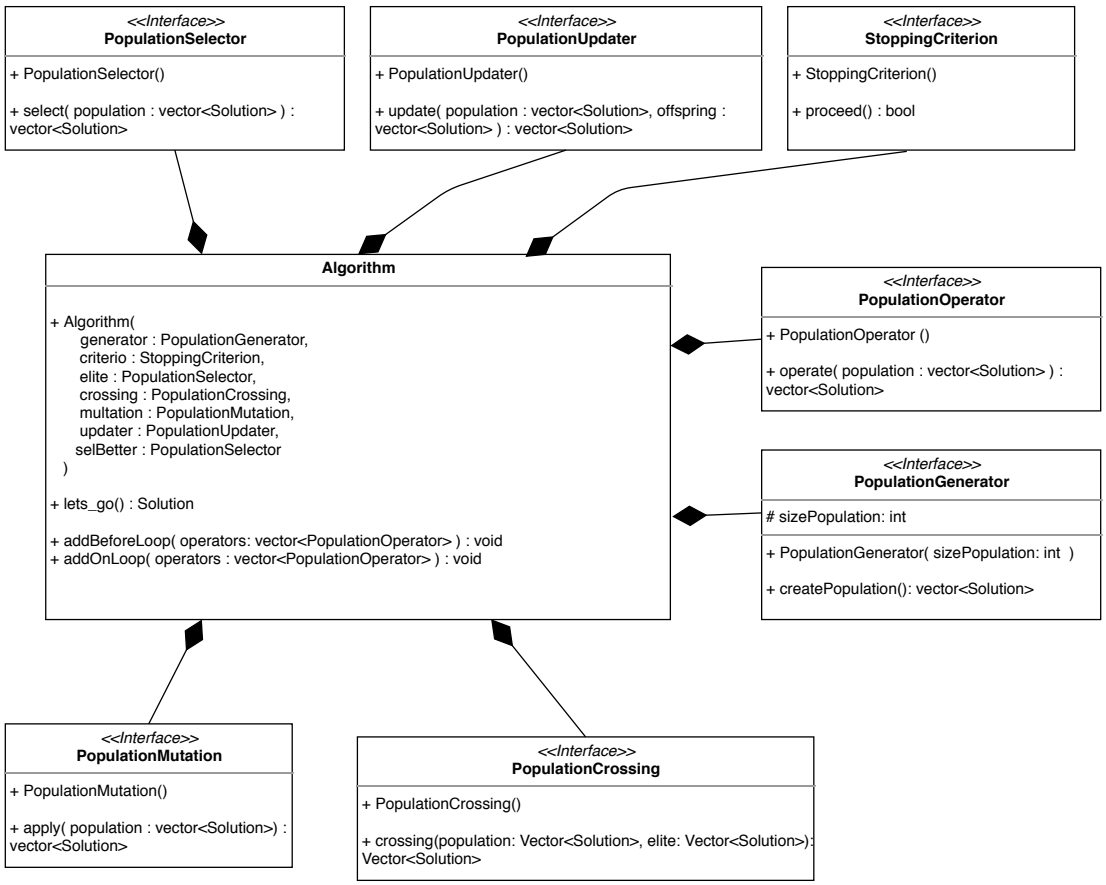

Figura 3. Diagrama de classes com as principais interfaces

3. Biblioteca C++ para o Problema QCaRS

Nesta seção será apresentada a contribuição deste artigo, que é a biblioteca C++ para o problema QCaRS. Inicialmente é descrita a especificação e implementação da biblioteca, e em seguida um exemplo de utilização e um experimento para testar a sua funcionalidade.

3.1. Especificação e Implementação

A biblioteca apresentada nesta seção tem a intenção de facilitar a implementação de novos algoritmos evolucionários para o problema QCaRS, mas pode ser utilizado para outros problemas. Para isso, ela utiliza o padrão de projeto *Strategy* para fornecer interfaces e as assinaturas de funções importantes. Além da estrutura do algoritmo, a biblioteca também disponibiliza a implementação da leitura das instâncias do problema e estruturas de dados. A biblioteca foi implementada na linguagem C++ e está disponível no Github.[1]

A figura 3 apresenta o diagrama de classes com as principais entidades da biblioteca. A nomenclatura destas entidades está relacionada com as operações existentes em algoritmos evolucionários, descritos anteriormente na seção 2.2. A interface *PopulationGenerator* é a abstração da geração da população inicial. A interface *PopulationSelector* é a abstração da realização uma seleção dentro da população. A interface *PopulationUpdater* é similar a interface anterior, mas recebe duas listas de indivíduos, abstraindo a

[1] `https://github.com/TiagoFunkUdescCeavi/Qcars-Interface`

realização da seleção considerando estas duas listas. A interface *StoppingCriterion* representa o critério de parada. A interface *PopulationCrossing* representa o processo de cruzamento dos indivíduos da população. A interface *PopulationMutation* é a abstração do processo de mutação nos indivíduos. A interface *PopulationOperator* representa uma operação qualquer sobre a população. Por exemplo, um processo de correção de soluções inválidas ou melhoria de soluções. Nesta implementação é possível adicionar operações que serão executadas em duas situações: operações que serão executados antes do processo de iteração inicie; e operações que serão executadas enquanto a iteração já está ocorrendo. Os métodos *addBeforeLoop* e *addOnLoop* servem para adicionar os operadores. Por fim, a classe *Algorithm* utiliza todas estas interfaces e é responsável por executar o fluxo principal do algoritmo.

A figura 4 apresenta o laço principal do algoritmo, que é implementado pela função *lets_go* da classe *Algorithm*, e que será executado enquanto o critério de parada não for satisfeito. Na linha 2 é executado a função que cria a população inicial. Nas linhas 3 a 5, existe um laço de repetição que executa todos os operadores necessário antes de começar o processo iterativo de melhora da população. Nas linhas 6 a 14 está o laço que vai realizar as iterações sobre a população seguindo o critério de parada. Na linha 7 é realizada a seleção da população de elite. Na linha 8 ocorre o cruzamento da população com a população de elite, que gera a prole. Na linha 8 ocorre a mutação sobre a prole. Nas linhas 10 a 12 está o laço que executa as operações sobre a prole. Na linha 13 ocorre a atualização da população levando em conta a prole gerada. Na linha 15 ocorre a seleção no melhor individuo, que representa a melhor solução encontrada na execução do algoritmo. O padrão *Strategy* se aplica neste algoritmo ao utilizar as interfaces da figura 3. O método *lets_go* desempenha o papel do *ContextInterface* na classe *Context* da figura 2.

```
1  Solution lets_go(){
2      population = generator->createPopulation();
3          for( PopulationOperator * op: operatorsBeforeLoop ){
4            population = op->operate( population );
5          }
6          while( criterio->proceed() ){
7            elitePop = elite->select( population );
8            offspring = crossing->crossing( population, elitePop );
9            offspring = multation->apply(  offspring );
10           for( PopulationOperator * op: operatorsOnLoop ){
11             offspring = op->operate( offspring );
12           }
13           population = updater->update( population, offspring );
14         }
15         return selBetter->select( population ).at( 0 );
16 }
```

Figura 4. Implementação de algoritmo evolucionário disponível na biblioteca

Além das entidades para implementar o algoritmo evolucionário, a biblioteca disponibiliza entidades para leitura de instâncias de problema QCaRS e também estruturas de dados que auxiliam a utilização da biblioteca. Estas entidades são apresentadas no diagrama de classes da figura 5 e são detalhadas a seguir.

A classe *InstanceReader* realiza a leitura das instâncias. Para este problema, existe

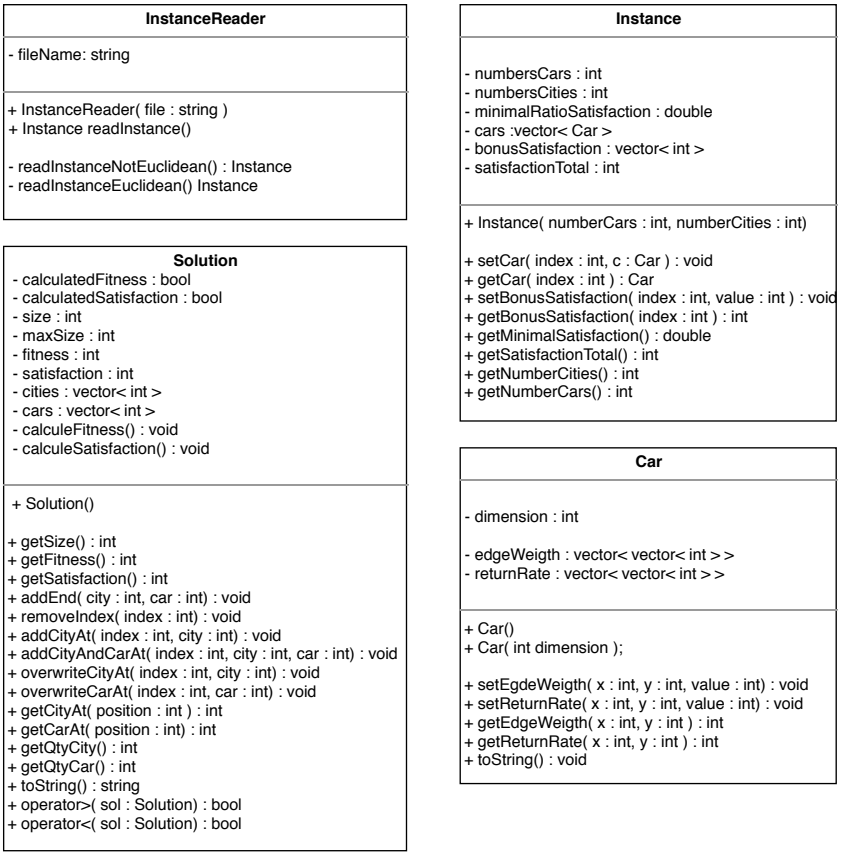

Figura 5. Diagrama com as classes de leitura de instâncias e estruturas de dados

dois tipos, a saber: euclidianas e não euclidianas [Goldbarg et al. 2016]. Na biblioteca basta passar o nome do arquivo da instância no construtor da classe e já é identificado o tipo e realizado a leitura a partir do arquivo. A leitura retorna um objeto do tipo *Instance*. A classe *Instance* possui os seguintes atributos: *numberCars* é o número total de carros; *numberCities* é o número de cidades; *minimalRatioSatisfaction* é o valor que representa o valor mínimo proporcional ao total de satisfação; *bonusSatisfaction* é a lista de inteiros que representam o valor de satisfação de cada cidade; *satisfactionTotal* é a soma dos valores de satisfação de cada cidade; e *cars* é a lista de objetos *Car*. A lista representa a lista de carros da instância.

A classe *Car* representa o carro com seus custos e multas. Esta classe possui as matrizes com os valores de custos de viagem entre cidades e valores de multa: *edgeWeight* é a matriz com os custos de viagens entre as cidades; *returnRate* é a matriz com os valores de multa se ocorrer a devolução em uma cidade diferent; *dimension* é o número de cidades e por consequência, a ordem das matrizes.

Por fim há a classe *Solution*, que representa a possível solução do problema. O atributo *calculatedFitness* é uma variável que controla o cálculo da aptidão pela função objetivo, para evitar o recálculo caso não houver alteração da solução. O atributo *calculatedSatisfaction* é semelhante, mas para o cálculo de satisfação. O atributo *size* salva a quantidade de cidades que estão na solução. *maxSize* indica o número máximo de cidades que a solução pode aceitar. *fitness* é o valor da aptidão. *satisfaction* é o valor da satis-

fação. *cities* e *cars* são listas de cidades e carros, respectivamente, presentes na solução. Importante frisar que a cidade *0* representa a cidade inicial e final do problema, assim a solução sempre deve ter esta cidade no início e fim da lista de cidades. Esta classe também possui métodos para calcular o valor da aptidão e valor de satisfação da solução, além de métodos para inserir e remover cidade e carros das listas.

3.2. Exemplo de Utilização

Para utilizar e executar esta biblioteca, é necessário um compilador C++. Neste exemplo utilizamos o G++ MingGW versão 6.3.0. Para iniciar a implementação, basta sobrescrever as interfaces da figura 3 de acordo com a implementação pretendida e passar no construtor da classe *Algorithm*.

A figura 6 apresenta um exemplo de uso da biblioteca. Entre as linhas 2 e 33

```
1  int main(int argc, char *argv[]) {
2      try{
3          string file = "Mali1ln.pcar";
4          int sizePopulation = 100;
5          double sizePlasmideo = 0.5;
6          double cross = 0.5;
7          double ratio = 0.3;
8          int limitIterations = 1000;
9
10         InstanceReader reader( file );
11         Instance inst = reader.readInstance();
12         GlobalVarables::instance = &inst;
13
14         PopulationGenerator * gen =
15           new GeneratePopulationWithHeuristic( sizePopulation );
16         StoppingCriterion* count = new Counter( limitIterations );
17         PopulationOperator* mul = new MultiOperatorsLocalSearch();
18         PopulationSelector* elite = new EliteSelector(ratio, cross);
19         PopulationCrossing* mem = new Memplas( sizePlasmideo );
20         PopulationMutation* mun = new EmptyMutation();
21         PopulationUpdater* upd = new BinaryTournament();
22         PopulationSelector* better = new SelectBetter();
23
24         vector< PopulationOperator * > operators;
25         operators.push_back( mul );
26
27         Algorithm alg(en,count,mul,elite,mem, mun,upd,better);
28
29         alg.addBeforeLoop( operators );
30         alg.addOnLoop( operators );
31
32         Solution s = alg.lets_go();
33     }catch (exception &e){
34         cerr << e.what() << endl;
35         return 1;
36     }
37     return 0;
38 }
```

Figura 6. Método main da biblioteca

estão um *try-catch* para capturar erros que a aplicação venha a lançar, como por exemplo, erro ao ler instâncias. Nas linhas 3 a 8 está os valores de argumentos. Os argumentos servem para fornecer parâmetros para a aplicação, por exemplo, tamanho da população. O usuário que for utilizar esta biblioteca, pode utilizar os argumentos que achar necessário para o problema. Nas linhas 10, 11 e 12, o leitor da instância realiza a leitura e salva a instância em uma variável global para que outras classes possam utilizá-la. A partir da linha 14 inicia a instanciação das classes criadas pelo usuário. Na linha 24 é criada o vetor com os operadores que devem ser passados para o algoritmo. Na linha 27 é instanciada a classe *Algorithm* como todos os parâmetros necessários. Nas linhas 29 e 30 são passados os operadores que executarão antes e durante o *laço* do algoritmo evolucionário. Na linha 32 esta a execução do algoritmo em si, ao chamar o método *lets_go*. Ao final do processo é retornado o melhor individuo.

3.3. Verificação da Biblioteca

Para verificar se a implementação da biblioteca está funcional, foi realizado a implementação do algoritmo MemPlas apresentado em [Goldbarg et al. 2016]. Esta implementação também está disponível no Github[1]. Foi realizado um experimento com quatro instâncias do problema para verificar se o resultado se aproximava do estado da arte apresentado por [Goldbarg et al. 2016]. As quatro instâncias foram escolhidas por serem pequenas, apenas para fim de verificar a implementação. A implementação do MemPlas foi executada 30 vezes para cada instância.

Os resultados do experimento são apresentados na tabela 1. A linha *Instância* indica o nome da instância que foi utilizada. A linha *Média* é o valor da média da aptidão da melhor solução encontrado nas trinta execuções do algoritmo. A linha *Melhor* é o valor do melhor resultado encontrado pela implementação da biblioteca. A linha *Estado da arte* contém os valores considerados *estado da arte* para a instância, determinados por [Goldbarg et al. 2016]. Assim, das quatro instâncias, o algoritmo obteve a solução estado da arte em três. Na instância *Mali11n* a implementação não atingiu o estado da arte. Isto pode estar relacionado a diferentes recursos de implementação, tais como construções de código e estruturas de dados, utilizados por [Goldbarg et al. 2016] (como não há acesso a implementação destes autores, não é possível identificar os recursos utilizados). Lembramos que este trabalho não tem a intenção de apresentar uma implementação do MemPlas, e sim uma biblioteca que facilite a implementação de algoritmos evolucionários.

Tabela 1. Resultados do experimento de verificação da biblioteca

Instância	AfricaSul11n	Bolivia10n	Niger12n	Mali11n
Média	539,63	452,97	495,27	671,97
Melhor	537	448	494	622
Estado da arte	537	448	494	607

4. Conclusão

Este artigo propôs uma biblioteca C++ para fornecer uma interface para o problema QCaRS. Este problema consiste em construir uma rota de cidades que são visitadas utilizando aluguel de veículos. O problema necessita garantir uma satisfação miníma ao

visitar as cidades e minimizando o custo da viagem. A biblioteca proposta fornece a leitura das instâncias, estrutura de dados para utilização do algoritmo e abstrações dos componentes de algoritmos evolucionários para facilitar a implementação. A biblioteca foi submetida a um experimento para verificar sua funcionalidade e ela obteve resultados próximos ao estado da arte em três de quatro instâncias de problemas QCaRS utilizadas.

Como trabalhos futuros, sugere-se estender a biblioteca para suportar outras variações de problemas do tipo caixeiro viajante. Ao estudar novos problemas, poderia aumentar a abstração do código, permitindo, por exemplo, mais facilidade para trocar a implementação de leitura de instâncias. Outra abordagem, estudar se a biblioteca funciona em problemas diferentes que o QCaRS.

Referências

Asconavieta, P. (2011). *O Problema do Caixeiro Alugador: um estudo algorítmico*. PhD thesis, Tese (Doutorado em Ciência da Computação)–Universidade Federal do Rio Grande do Norte.

da Silva Menezes, M., Goldbarg, M. C., and Goldbarg, E. F. (2014). A memetic algorithm for the prize-collecting traveling car renter problem. *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 3258–3265.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (2000). *Padrões de projeto: Soluções Reutilizáveis de Software Orientado a Objetos*. Bookman, Porto Alegre, Brasil.

Goldbarg, M. C., Goldbarg, E. F., Menezes, M. d. S., and Luna, H. P. (2016). Quota traveling car renter problem: Model and evolutionary algorithm. *Information Sciences*, 367:232–245.

Linden, R. (2008). *Algoritmos genéticos (2a ediçao)*. Brasport, Brasil.

Menezes, M. d. S., Goldbarg, M. C., Goldbarg, E. F., Ferreira, V. E. S., and ColaÇo, G. C. (2017). Abordagens GRASP aplicadas ao problema quota cars. *Anais do 49 Simpósio Brasileiro de Pesquisa Operacional*, pages 1807–1818.

Reinelt, G. (1995). Tsplib95. *Interdisziplinäres Zentrum für Wissenschaftliches Rechnen (IWR), Heidelberg*, 338:1–16.

Talbi, E.-G. (2009). *Metaheuristics: from design to implementation*, volume 74. John Wiley & Sons, United States.

Wolfe, H. E. (2012). *Introduction to non-Euclidean geometry*. Courier Corporation.