Algoritmo Memético para o Problema do Caixeiro Viajante Assimétrico como Parte de um *Framework* para Algoritmos Evolutivos

Luciana Salete Buriol

Dissertação de Mestrado apresentada à Faculdade de Engenharia Elétrica e de Computação da Unicamp, como requisito parcial para a obtenção do título de Mestre em Engenharia Elétrica.

Área de concentração: Automação.

Banca Examinadora:

- Dr. Paulo Morelato França: orientador e professor do Departamento de Engenharia de Sistemas da Faculdade de Engenharia Elétrica e de Computação, Unicamp.
- Dr. Felipe Martins Müller: professor do Departamento de Eletrônica e Computação do Centro de Tecnologia, Universidade Federal de Santa Maria, RS.
- Dr. Marco Aurélio Amaral Henriques: professor do Departamento de Engenharia da Computação e Automação Industrial da Faculdade de Engenharia Elétrica e Computação, Unicamp.

Campinas, 21 de fevereiro de 2000.

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

B917a

Buriol, Luciana Salete
Algoritmo memético para o problema do caixeiro viajante assimétrico como parte de um *framework* para algoritmos evolutivos / Luciana Salete Buriol.-- Campinas, SP: [s.n.], 2000.

Orientador: Paulo Morelato França.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Framework (Programa de computador). 2. Algoritmos genéticos. 3. Problema do caixeiro viajante 4. Geradores. I. França, Paulo Morelato. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

RESUMO

Dentre a gama de técnicas heurísticas e exatas existentes para a resolução de problemas combinatórios, os algoritmos populacionais genéticos e meméticos têm se destacado devido a sua boa performance. Em especial, os algoritmos meméticos podem ser considerados atualmente como uma das técnicas melhores sucedidas para a resolução de vários problemas combinatórios, dentre eles, o problema do caixeiro viajante. Nesta dissertação será apresentado um algoritmo memético aplicado ao problema do caixeiro viajante assimétrico, com a proposta de uma nova busca local: *Recursive Arc Insertion*. Os resultados computacionais considerando as 27 instâncias assimétricas da TSPLIB são apresentados, analisados e comparados com resultados obtidos por outros métodos propostos para o problema. O mesmo algoritmo é também aplicado a 32 outras instâncias assimétricas e a 30 instâncias reduzidas do problema de ciclos hamiltonianos não direcionados. Um *framework* para algoritmos evolutivos é apresentado, já incluindo o algoritmo memético implementado e a redução de instâncias do problema de ciclos hamiltonianos não direcionados para o problema do caixeiro viajante simétrico. Além disso, dois geradores portáveis de instâncias com solução ótima conhecida são descritos: um para o problema do caixeiro viajante assimétrico e outro para o problema de ciclos hamiltonianos.

ABSTRACT

Among the range of heuristic and exact techniques for solving combinatorial problems, the genetic and memetic populational algorithms play an important role due to their good performance. In special, the memetic algorithms can be considered currently as one of the best techniques to solve several combinatorial problems, especially, the traveling salesman problem. In this dissertation a memetic algorithm applied to the asymmetric traveling salesman problem is developed, and a new local search is proposed: *Recursive Arc Insertion*. The computational results considering the 27 asymmetric instances from TSPLIB are presented, analysed and compared with results attained by other methods recently published. The same algorithm is also applied to 32 other asymmetric instances and to 30 reduced instances from undirect hamiltonian cycle problem. A framework for evolutionary algorithms is also presented, including the memetic algorithm implemented and the codes which performs a reduction from the undirect hamiltonian cycle problem to the symmetric traveling salesman problem. Besides, two portable instances generators with a known optimal solution are described: one for asymmetric traveling salesman problem and other for hamiltonian cycle problem.

DEDICATÓRIA

Além da Terra, além do Céu,

no trampolim do sem-fim das estrelas,

no rastro dos astros,

na magnólia das nebulosas.

Além, muito além do sistema solar,

até onde alcançam o pensamento e o coração,

vamos!

vamos conjugar

o verbo fundamental essencial,

o verbo transcendente, acima das gramáticas

e do medo e da moeda e da política,

o verbo sempreamar,

o verbo pluriamar,

razão de ser e de viver.

Além da Terra, Além do Céu / Carlos Drummond de Andrade

Ao Marcelo, com muito amor.

AGRADECIMENTOS

À minha família, em especial aos meus pais: Laurí e Orilde. A melhor forma de retribuir todo carinho, esforço e apoio que vocês me deram, é mostrar que não foi em vão. Mais do que tudo, valeu o exemplo de vida e dedicação.

Ao professor Paulo Morelato França, meus agradecimentos, minha admiração e minha confiança. Hoje sei a sorte que tive ao ser aceita sob sua orientação.

Ao Pablo Moscato, colega e amigo que participou ativamente de todas as fases deste trabalho. Seu entusiasmo, experiência e facilidade de trabalhar em grupo certamente contribuiram significativamente para o desenvolvimento desta tese.

Ao Felipe, cujo estímulo constante, confiança e apoio foram determinantes para meu gosto pela pesquisa e são estimulantes para meu crescimento contínuo.

Aos demais membros do projeto Memepool, com quem dividi muitas alegrias e muitas dúvidas: Alexandre Mendes e Andréa Toniolo.

À querida vó Dina, que não mediu esforços para tornar minha vida em Campinas mais agradável.

Aos colegas de laboratório/CAB'S/RU: Edilson, Pablo, Marcelo, Alexandre, Rodrigo, Cláudio, Joana, Moacir, Denise, Cris e também Débora, Regina, Cíntia, Fran e Vitória.

Aos funcionários da FEEC, em especial a Márcia e a Noêmia, cuja eficiência certamente contribuiu para a realização de todas as etapas do meu mestrado.

Aos amigos de vôlei e festa: Alessandro, Adilson, Lie, Andréia, Márcia, Simone, Hilson, Érico, Roberta, Léo, Alexandre, Paulo e Débora.

À Universidade Pública e Gratuita, que permitiu que eu chegasse até aqui.

À FAPESP - Fundação de Amparo à Pesquisa do Estado de São Paulo, cujo apoio financeiro foi fundamental para o desenvolvimento desse trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

Capítulo

1

INTRODUÇÃO

A denominação "algoritmo populacional" é designada aos métodos que fazem busca em uma população de indivíduos, ao contrário daqueles que exploram um único indivíduo da vizinhança a cada iteração. Com isso, ao longo das iterações, não se constrói uma trajetória única de busca, pois novas soluções são obtidas através da combinação de soluções anteriores.

Os algoritmos genéticos (*Genetic Algorithms* - GAs) são algoritmos populacionais baseados na evolução natural das espécies. Neste contexto, os indivíduos (soluções de um problema) são submetidos a processos de mutação, reprodução, competição e seleção. A reprodução (recombinação) permite a transmissão de informação genética dos pais para o filho, enquanto a mutação modifica parte da genética dos indivíduos com o objetivo único de aumentar a diversidade da população. Este mecanismo é repetido por diversas gerações, fazendo com que a cada geração somente os indivíduos mais aptos sejam selecionados para fazerem parte da próxima geração.

Um algoritmo memético (*Memetic Algorithm* - MA), além da evolução genética, utiliza o conceito de evolução cultural. A informação cultural não precisa ser transmitida por parentescos (como no caso genético), e sim por qualquer indivíduo da população. Devido aos memes (unidade de replicação da informação cultural) não estarem vinculados ao mecanismo hereditário, a informação cultural pode ser transmitida de um indivíduo para vários outros, ou até mesmo para toda população, sem que uma geração seja transcorrida. Por esse motivo, a informação memética é transmitida de forma mais rápida e flexível que a genética. Com o objetivo de agregar informação memética, um operador de busca local é adicionado ao algoritmo genético, transformando-o num algoritmo genético híbrido, melhor conhecido como algoritmo memético (Moscato, 1989, 1999).

Trabalhos recentes têm mostrado que esta técnica está ganhando espaço entre os métodos com melhor performance quando aplicados a problemas de otimização combinatória. Uma quantidade considerável de operadores de recombinação, busca local, mutação, etc., têm sido propostos, alguns para problemas específicos, outros para uso geral.

Com o objetivo de reunir um gama destes operadores em uma única ferramenta que fosse capaz de gerenciar a manipulação destes, buscou-se da engenharia de *software* o conceito de *framework*. Um *framework* é um conjunto de classes colaborativas (abstratas e concretas) que permite reutilização de projeto e provê uma infra-estrutura genérica de soluções para um conjunto de problemas específicos; é uma abstração de um conjunto de classes interrelacionadas.

O *framework* para programação evolutiva proposto nesta dissertação tem o objetivo de, recebendo a informação de quais métodos e operadores o usuário deseja utilizar, estruturar uma seqüência de execução conforme os dados recebidos. Por se tratar de um sistema grande e complexo, o paradigma da orientação a objetos se tornou imprescindível para compor módulos independentes e consistentes. Cada operador é alocado em uma classe exclusiva e independe do fluxo do restante do programa. Desta forma, o mesmo operador pode ser aplicado a soluções de diversos problemas, desde que utilizem a mesma estrutura para representação de solução.

Neste contexto, o uso de reduções se mostrou adequando. Considerando que o *framework* reúne operadores para representações de soluções para diversos problemas, além de rotinas para leitura das instâncias correspondentes, as reduções se encaixaram perfeitamente na estrutura.

1.1 PRINCIPAIS OBJETIVOS DA DISSERTAÇÃO

Os principais objetivos desta dissertação são apresentar a estruturação de um *framework* para programação evolutiva, bem como um algoritmo memético para o problema do caixeiro viajante assimétrico (*Asymmetric Traveling Salesman Problem* - ATSP).

Para apresentar a estruturação do *framework*, além de uma breve visão de orientação a objetos, os diagramas de classes que compõem o *framework* são cuidadosamente descritos e relatados. Para a modelagem do sistema foi utilizada uma linguagem visual adequada: *Unified*

Modelling Language - UML.

Com o objetivo de inserir código e testar um problema alvo, o MA proposto para o ATSP foi adicionado ao *framework*. Os resultados computacionais obtidos pela execução deste algoritmo são relatados e analisados.

Reduções também são objeto do plano inicial e por isso o conceito é apresentado e aplicado em dois casos. Para avaliar a performance do MA quando aplicado a uma instância reduzida, assim como avaliar a transformação da instância após a redução, testes computacionais são apresentados e analisados para a redução do problema de ciclos hamiltonianos não direcionados (*Undirect Hamiltonian Cycle Problem* - UHCP) para o problema do caixeiro viajante simétrico (*Symmetric Traveling Salesman Problem* - STSP).

1.2 ORGANIZAÇÃO DA DISSERTAÇÃO

O capítulo 2 é inteiramente dedicado ao problema do caixeiro viajante (*Traveling Salesman Problem* - TSP) e tem o objetivo de conceituar e caracterizar o problema, visto que durante toda dissertação esse problema é tomado como exemplo. Neste capítulo, o problema é conceituado, a formulação matemática e a complexidade do mesmo são apresentadas, algumas aplicações práticas são relatadas e a estrutura básica dos algoritmos meméticos é detalhadamente descrita. Uma revisão bibliográfica do mesmo é desenvolvida com o objetivo de situar adequadamente as contribuições propostas nesta dissertação.

O capítulo 3 descreve o algoritmo memético implementado para o ATSP e tem o objetivo principal de apresentar um novo operador de busca local: *Recursive Arc Insertion* - RAI. Esta busca local foi especialmente desenvolvida para o ATSP e constitui-se numa das principais contribuições originais deste trabalho. Todos os operadores implementados são devidamente descritos: mutação, *restart*, recombinação e busca local. Para avaliar a performance do algoritmo, resultados computacionais utilizando 4 operadores de recombinação diferentes são testados. A fim de comparar a performance do algoritmo com métodos heurísticos já propostos, são apresentados os resultados computacionais de seis algoritmos heurísticos para o ATSP com os

melhores resultados encontrados na literatura. Os resultados se concentram nas instâncias de ATSP da TSPLIB[1] - uma biblioteca de instâncias via *Internet*.

O capítulo 4 apresenta um gerador de instâncias de ATSP, a partir de instâncias simétricas da TSPLIB, e um gerador de instâncias para o problema de ciclos hamiltonianos (*Hamiltonian Cycle Problem* - HCP). O objetivo da implementação desses geradores é viabilizar a execução do algoritmo memético desenvolvido considerando instâncias grandes (com mais de 443 cidades) de ATSP e pequenas (com menos de 1000 vértices) de HCP (para serem reduzidas para TSP), visto que estas não estão disponíveis na TSPLIB. São apresentados e analisados resultados computacionais do algoritmo memético descrito no capítulo 3 aplicado às instâncias criadas pelo gerador de instâncias assimétricas de TSP.

O capítulo 5 tem o objetivo principal de conceituar reduções e apresentar a redução de HCP para TSP. São apresentados e analisados os resultados computacionais resultantes da aplicação do algoritmo memético às instâncias reduzidas de HCP (algumas provenientes da TSPLIB, outras criadas pelo gerador descrito no capítulo 4).

O capítulo 6 apresenta os diagramas de classes do *framework*, modelados com UML, e uma descrição breve de cada um. Como exemplo, estão inseridos o algoritmo memético descrito no capítulo 3 e as reduções descritas no capítulo 5. Um Apêndice, único da dissertação, foi inserido com o objetivo de sanar possíveis dúvidas que possam dificultar o entendimento da leitura do Capítulo 6. Este Apêndice apresenta Java, a linguagem de programação utilizada, e UML, linguagem visual utilizada na modelagem do *framework*. Os principais conceitos de orientação a objetos são apresentados, juntamente com uma rápida descrição de como estes são implementados em Java e modelados em UML.

O capítulo final apresenta as conclusões da dissertação, principais contribuições, publicações, bem como algumas sugestões para trabalhos futuros.

---

[1] http://www.iwr.uni-heidelberg.de/iwr/comopt/soft/TSPLIB95/TSPLIB

Capítulo

2

O PROBLEMA DO CAIXEIRO VIAJANTE

O TSP é um problema clássico da otimização combinatória e tem despertado grande interesse por parte dos pesquisadores da área. Isso se deve ao fato de ser um problema simples de descrever, mas muito difícil de resolver, além de possuir inúmeras aplicações práticas. O TSP pertence à classe de problemas NP-difíceis, ou seja, o tempo gasto para resolvê-lo pode ser exponencial em relação ao tamanho da instância. Devido a isso, a resolução do problema utilizando métodos heurísticos ganha maior importância, principalmente quando aplicado a instâncias grandes do problema.

Este capítulo tem por objetivo conceituar o problema, apresentando a formulação matemática, complexidade, aplicações práticas e os métodos de resolução do mesmo. Uma ênfase é dada aos algoritmos meméticos, visto que são objeto de análise do capítulo 3. Para finalizar o capítulo, uma revisão bibliográfica do problema analisa resultados obtidos por algoritmos exatos e heurísticos aplicados ao TSP, bem como aspectos relevantes sobre implementações e estudos teóricos do problema.

2.1. DEFINIÇÃO

A um caixeiro viajante é informado um conjunto de cidades e um custo c_{ij} associado a cada par de cidades i e j deste conjunto, representando a distância de ir da cidade i à cidade j. O

caixeiro deve partir de uma cidade inicial, passar por todas as demais uma única vez e retornar à cidade de partida. O problema consiste em fazer esta trajetória pelo menor caminho possível.

Formalmente, o TSP pode ser definido considerando um grafo completo. Dado G = (V, A) um grafo onde V é um conjunto de n vértices e A é o conjunto de arcos ou arestas que conectam cada par de cidades i e $j \in$ V. A cada arco/aresta está associado um custo c_{ij}. O TSP consiste em encontrar a rota de menor custo, passando por cada vértice uma única vez. No caso simétrico $c_{ij} = c_{ji}$ para toda cidade $i, j \in$ V, enquanto que o caso assimétrico possui pelo menos um caso em que $c_{ij} \neq c_{ji}$.

Embora a definição do problema é normalmente referenciada para um problema de minimização, o problema também é encontrado na sua forma de maximização.

2.2. FORMULAÇÃO MATEMÁTICA

A formulação matemática é a mesma para ambos os casos, apenas indicando a função objetivo como sendo de *Min* (minimização) ou *Max* (maximização), conforme o caso. Na literatura a formulação é normalmente referenciada usando uma função de *Min*.

Função objetivo: $$MIN \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}.x_{ij}$$

Sujeito a :

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad j = 1, ..., n \qquad (1)$$

$$\sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, ..., n \qquad (2)$$

$$\sum_{i \in S}\sum_{j \in S} x_{ij} \leq |S| - 1, \qquad \forall S \subset V, S \neq \varnothing \qquad (3)$$

$$x_{ij} \in \{0, 1\} \qquad i, j = 1, ..., n \qquad (4)$$

A variável inteira $x_{ij} = 1$ indica que a cidade j é visitada logo após a cidade i, caso contrário $x_{ij} = 0$. A variável n representa o número de cidades do problema, S é um subconjunto do conjunto $\{1, 2, ..., n\}$, e o símbolo "| |" denota a cardinalidade do conjunto. A função objetivo representa a minimização do somatório das distâncias entre as cidades da rota. As restrições (1) e

(2) garantem que para cada cidade $i \in n$, há exatamente uma conexão de chegada e uma partindo para outra cidade. A restrição (3) garante a não existência de subrotas (uma rota que não inclua todas as n cidades) e a restrição (4) define x como variável binária.

2.3. COMPLEXIDADE

Segundo a teoria de NP-completude (Garey e Johnson, 1979), os problemas de otimização combinatória podem ser classificados, segundo sua complexidade, em P, NP, NP-difícil e NP-completo. Os problemas pertencentes à classe P são ditos tratáveis computacionalmente, já os demais são considerados intratáveis, ou seja, o número de *computações* executadas no melhor algorimo conhecido para o problema cresce exponencialmente em função do tamanho da instância. Neste contexto, por *computações* subentende-se operações primitivas (atribuição, soma, etc.) (Cormen et al., 1996).

Antes de caracterizar as 4 classes de problemas, é necessário saber o que é um algoritmo determinístico e um não-determinístico. Dado o fluxo de controle de um algoritmo, ele é determinístico se, a cada passo deste, o próximo passo é único, enquanto que os algoritmos não-determinísticos fazem uma escolha aleatória do próximo passo, entre um número fixo de possibilidades, e, conseqüentemente, o fluxo do algoritmo depende da alternativa selecionada.

P (*Polinomial Time*): problemas que podem ser resolvidos por algoritmos determinísticos polinomiais em função do tamanho da instância; a complexidade do problema cresce polinomialmente em função do tamanho da instância.

NP (*NonDeterministic Polinomial Time*): problemas que podem ser resolvidos por algoritmos não-determinísticos polinomiais no tamanho da instância; a complexidade do problema cresce exponencialmente em função do tamanho da instância.

Conclui-se que $P \subseteq NP$, mas não é sabido se $P = NP$. Para resolver a questão, todos os problemas pertencentes à classe NP deveriam ser resolvidos em tempo polinomial, ou serem reduzidos polinomialmente para os problemas da classe P.

NP-difícil: compostos pelos problemas caracterizados por haver redução polinomial a partir de todo problema pertencente à classe NP.

NP-completo: problemas pertencentes a interseção das classes NP e NP-difícil.

Desta forma, para provar que um problema Q pertence à classe NP-completo, deve-se mostrar que Q está em NP e encontrar um problema R, que se sabe estar em NP-difícil, e reduzi-lo polinomialmente ao problema Q.

O problema do caixeiro viajante pertence à classe NP-difícil (Garey e Johnson, 1979).

2.4. APLICAÇÕES PRÁTICAS

O número de problemas que podem ser modelados como um problema de caixeiro viajante é muito grande. Além disso, o TSP possui muitas aplicações práticas. Nesta seção, algumas aplicações práticas serão vistas, com o objetivo de justificar a importância deste problema.

2.4.1. Fabricação de placas de circuitos eletrônicos

A fabricação de placas de circuitos eletrônicos é feita em diversas etapas: perfuração da placa, adição e soldagem de *chips*, conexão entre pinos, teste do circuito, etc. Quatro destas tarefas são aplicações do TSP.

Perfuração de placas: o problema da perfuração de placas de circuito impresso (*Print Circuits Board* - PCB) é uma aplicação padrão do TSP simétrico. Para conectar um condutor num nível com um condutor em outro nível, ou para posicionar os pinos dos circuitos integrados, furos são feitos através da placa. Estes podem ser de diferentes diâmetros. Para fazer dois furos de diferentes diâmetros consecutivamente, a cabeça da máquina tem de mover-se para uma caixa de

ferramentas e mudar o equipamento de perfuração, o que consome tempo. Assim, fica claro que no início é preciso escolher algum diâmetro, fazer todos os furos com esse diâmetro, mudar a broca, fazer os furos do próximo diâmetro e assim por diante. Desta forma, este problema pode ser visto como uma seqüência de TSPs simétricos, um para cada diâmetro. As "cidades" são as posições iniciais e o conjunto de todos os furos que podem ser feitos com uma broca; a "distância" entre duas cidades é o tempo gasto para mover a cabeça de uma posição para outra. A meta é minimizar o tempo de deslocamento da cabeça da máquina. Ainda, ao finalizar o perfuramento de uma placa, a peça perfuradora deve retornar ao ponto de partida, para iniciar a perfuração de uma nova placa.

Soldagem dos *chips* na placa: a máquina deve fazer a soldagem dos *chips* partindo de uma posição inicial, passando por todos os pontos de soldagem, e retornar ao ponto inicial para iniciar a soldagem em uma nova placa.

Conexão entre os pinos: as conexões de pinos têm tamanho limitado, restritos a um número de conexões. Muitas conexões a um mesmo pino dificulta futuras alterações ou correções. Em geral o número de conexões máximo é um valor pequeno e pré-estabelecido, muito menor que sua capacidade. Caso esse valor pré-estabelecido seja 2, o problema pode ser modelado como um TSP.

Teste do circuito: para verificação se o circuito não foi montado com curto-circuito ou em circuito aberto, alguns pontos são testados em cada placa. Percorrer estes pontos para fazer a verificação constitui num problema de TSP.

2.4.2. Seqüenciamento de tarefas

Suponha que n tarefas devam ser processadas seqüencialmente em uma determinada máquina. O tempo de preparo da máquina para processar a tarefa j imediatamente após a tarefa i é designado por c_{ij}. O problema de encontrar uma seqüência de execução para as tarefas, de forma a minimizar o tempo total de processamento, pode ser modelado como um TSP (Laporte, 1992).

2.4.3. Roteamento de veículos resolvido como um problema de *m*-caixeiros

Suponha que fregueses exijam certas quantidades de alguma mercadoria e um fornecedor tenha que satisfazer todas as demandas com uma frota de caminhões. Aqui temos o problema adicional de designar fregueses a caminhões e encontrar um escalonamento de entrega para cada caminhão de forma que a sua capacidade não seja excedida, bem como minimizar o custo total da viagem. O problema de roteamento de veículos pode ser resolvido como um TSP, se não há restrição de tempo ou se o número de caminhões é fixo (digamos m). Neste caso, ao invés de apenas um caixeiro, consideram-se m caixeiros, os quais se localizam numa cidade $n+1$ e têm que visitar as cidades $1,2,...,n$. A tarefa é selecionar estes caixeiros e designar-lhes rotas tais que, no conjunto destas rotas, cada cidade seja visitada exatamente uma vez. A ativação do caixeiro j inclui um custo fixo w_j. O custo da rota do caixeiro j é a soma das distâncias entre as cidades, iniciando e retornando para a cidade $n+1$. O problema dos m-caixeiros agora consiste em selecionar um subconjunto de caixeiros e designar uma rota para cada um deles de forma que cada cidade seja visitada por exatamente um caixeiro e que o custo total para visitar todas as cidades seja o menor possível. Este problema pode ser transformado em um TSP assimétrico envolvendo apenas um caixeiro (Jünger et al., 1995).

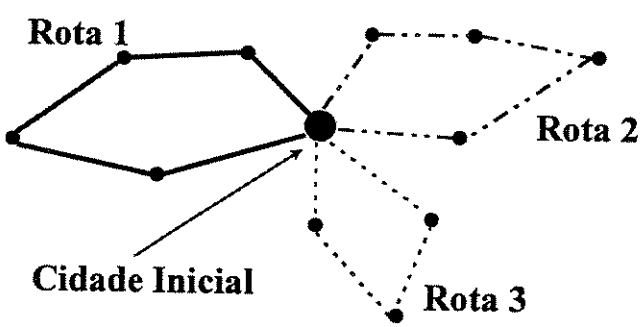


Figura 1 - Múltiplos Caixeiros Viajantes.

Outro exemplo da aplicação dos m-caixeiros é a coleta de cartas nas caixas de correio. Suponha que, numa cidade, n caixas de correio tenham que ser esvaziadas a cada dia e dentro de um certo período de tempo (digamos 1 hora). O problema consiste em encontrar o número mínimo de caminhões para fazer isto e o tempo mais curto para fazer a coleta usando este número de caminhões.

2.4.4. Cristalização com raio-X

Outra aplicação direta do TSP ocorre na análise da estrutura de cristais (Jünger et al., 1995). Aqui um difratômetro de Raio-X é usado para obter informações a respeito da estrutura do material cristalino. Para este fim, um detetor mede a intensidade das reflexões de Raio-X do cristal de várias posições. Considerando que a medição por si própria pode ser executada rapidamente, há uma considerável sobrecarga no tempo de posicionamento, pois até 30.000 posições devem ser realizadas para alguns experimentos. Neste exemplo, o posicionamento envolve a movimentação de quatro motores. O tempo necessário para o movimento de uma posição para outra pode ser computado de forma precisa. A seqüência em que as medidas são tomadas para várias posições é irrelevante. Conseqüentemente, a melhor seqüência para as medidas a fim de minimizar o tempo de posicionamento total tem de ser determinada. O problema pode ser modelado como um TSP simétrico.

2.4.5. Controle de robôs

Para que seja possível fabricar alguma peça, um robô tem que realizar uma seqüência de operações (corte de ranhuras, etc.). A tarefa resume-se em determinar uma seqüência para realizar as operações necessárias, a fim de minimizar o tempo de processamento. Surge uma dificuldade nesta aplicação, uma vez que há restrições de precedência. Desta forma, temos o problema de encontrar o caminho hamiltoniano mais curto (onde as distâncias irão corresponder aos tempos necessário na mudança de posicionamento, bem como de escolha de ferramentas) que satisfaçam certas relações de precedência. Recebendo como dado de entrada a matriz de distâncias entre cada cidade i, j, o problema de caminho hamiltoniano pode ser definido como o problema de encontrar um caminho que, partindo de uma cidade inicial, passe uma única vez por todas as demais cidades, sem retornar ao ponto de partida.

2.5. MÉTODOS DE RESOLUÇÃO

A classificação mais geral para métodos de resolução de problemas de otimização diferencia métodos exatos de heurísticos.

Os métodos exatos garantem encontrar a solução ótima para o problema. As técnicas exatas freqüentemente utilizadas para resolver o TSP são *branch-and-bound* e *branch-and-cut*.

As heurísticas (do grego *heuriskein* = descobrir) não têm prova de convergência e não garantem encontrar a solução ótima (Osman, 1991); são procedimentos para resolver problemas através de um enfoque intuitivo, em geral racional, no qual a estrutura do problema possa ser interpretada e explorada inteligentemente para obter uma solução razoável.

Alguns autores diferenciam métodos heurísticos e de aproximação. Segundo Osman (1991), métodos de aproximação são aqueles que possuem propriedades de convergência mas não garantem chegar à otimalidade.

As heurísticas podem ser divididas, segundo Osman (1991), em construtivas, de melhoramento, programação matemática, particionamento, restrição do espaço de soluções, relaxação do espaço de soluções, algoritmos compostos e metaheuríticas. No caso das implementações requeridas para este trabalho, foram implementadas heurísticas de construção e de melhoramento, além de uma metaheurística.

Heurísticas de construção geram uma solução adicionando componentes individuais (nós, arcos, variáveis, etc.) passo a passo, até que uma solução para o problema seja gerada. Esta solução pode ser factível ou não (Osman, 1991).

Uma heurística de melhoramento inicia com uma solução factível e sucessivamente são excluídos e adicionados componentes a esta solução, de forma que uma solução vizinha de melhor qualidade seja obtida. A heurística pára quando nenhum movimento de melhora pode ser executado, ou seja, a heurística atingiu um ótimo local. Esta heurística também é encontrada com a denominação de heurística de busca local ou de vizinhança.

Metaheurísticas são procedimentos heurísticos que guiam um operador de busca local na exploração do espaço de soluções para além da otimalidade local, explorando características de boas soluções e novas regiões promissoras.

Conforme as características de busca, as metaheurísticas podem ser divididas em duas classes: a primeira compreende os métodos que exploram apenas um elemento da uma vizinhança a cada iteração e a segunda compreende os métodos que exploram uma população de soluções a cada iteração (algoritmos populacionais).

No primeiro caso, a vizinhança e a forma de explorá-la é alterada de acordo com a estratégia da busca, sendo que somente um elemento da vizinhança é escolhido a cada iteração. Esse tipo de varredura do espaço de soluções gera um caminho ou trajetória de soluções, obtido pela transição de uma solução para outra de acordo com os movimentos permitidos pela metaheurística. Dentro dessa classe de métodos pode-se citar *simulated annealing* (Kirkpatrik et al., 1983), busca tabu (Glover e Laguna, 1993; Fiechter, 1994), GRASP (*Greedy Randomized Adaptive Procedure*) (Feo e Resende, 1994), redes neurais (Potvin, 1993) e, recentemente, *simulated jumping* (Amin, 1999).

Em contraposição a esses métodos, existem aqueles que exploram uma população de soluções a cada iteração. Esses métodos constituem a segunda classe de metaheurísticas e suas estratégias de busca são capazes de explorar várias regiões do espaço de soluções a cada vez. Dessa forma, ao longo das iterações não se constrói uma trajetória única de busca, pois novas soluções são obtidas através da combinação de soluções anteriores. Nessa classe estão os algoritmos genéticos (Goldberg, 1989), algoritmos meméticos (Moscato, 1989, 1999), *scatter search* (Glover, 1977) e otimização por colônia de formigas (*Ant Colony Systems* - ACO) (Stützle e Dorigo, 1999).

2.5.1. Algoritmos populacionais genéticos e meméticos

Os Algoritmos Genéticos foram inicialmente estudados na década de 60, sendo formalizados por Holland na década de 70 (Holland, 1975). O objetivo inicial não foi desenvolver algoritmos para solução de problemas específicos, mas sim estudar formalmente os fenômenos de adaptação, naturais ou artificiais, com o propósito de importar estes mecanismos de adaptação para ambientes computacionais.

Algoritmos genéticos são procedimentos computacionais inspirados na teoria Neo-Darwiniana. Esta teoria teve origem da combinação de três fenômenos naturais: evolução das espécies (Darwin), seleção natural de indivíduos (Weismann) e transmissão da informação genética (Mendel). Pela teoria Neo-Darwinista, a vida é propagada devido a atuação de alguns processos nas populações e espécies. Estes processos são: reprodução, mutação, competição e seleção (Fogel, 1995) e são inerentemente paralelos. O processo de reprodução (recombinação) envolve dois indivíduos (soluções de um problema), dando origem a um novo, o qual herdará características genéticas de seus pais. O processo de seleção avalia cada indivíduo, selecionando os mais aptos para fazerem parte da próxima geração. A mutação é introduzida neste contexto com o objetivo único de introduzir diversidade à população. Estes processos são repetidos geração a geração, de forma que a população (conjunto de soluções = população de soluções) sofre um processo evolutivo, gerando indivíduos melhor adaptados no transcorrer das gerações.

Nesta dissertação a palavra recombinação, formalizada por Moscato (1999), denota o procedimento de união de dois ou mais pais para gerar um ou mais filhos, em substituição à palavra *crossover* geralmente encontrada na literatura. Como na biologia o *crossover* é sempre realizado com exatamente dois pais, a palavra recombinação representa um caso mais geral, incluindo a possibilidade de vários pais fazerem parte desse processo. Além disso, a recombinação pode fazer uso das informações da instância, como por exemplo para o TSP, priorizar a inserção dos arcos mais promissores. Na biologia o *crossover* não exerce este papel.

Os algoritmos meméticos, além da evolução genética, utilizam o conceito de evolução cultural. A informação cultural não é transmitida através de um processo de recombinação, e sim pela comunicação entre indivíduos. Além disso, genes são transmitidos através de gerações, enquanto os memes (unidade replicante da informação cultural) podem ser transmitidos sem que uma geração seja transcorrida. Devido aos memes não estarem vinculados ao mecanismo hereditário, a informação cultural pode ser transmitida de um indivíduo para vários outros, ou até mesmo para toda população, sem que uma geração seja transcorrida. Por esse motivo, a informação memética é transmitida de forma mais rápida e flexível que a genética.

O mecanismo que permite a transmissão da informação memética é a introdução de um (ou mais) operador de busca local. Com esse objetivo, este operador é adicionado ao algoritmo genético, transformando-o num algoritmo memético, também conhecido como algoritmo

genético híbrido (Moscato, 1989, 1999).

No caso da presença de mais de um operador com a mesma função (por exemplo: dois operadores de busca local), estes podem ser executados em toda geração ou diferentes probabilidades/taxas podem ser atribuídas a cada um, de forma que nem todos operadores sejam executados a cada geração.

Na Figura 2 é apresentado um pseudocódigo ilustrativo da estrutura geral de um MA (detalhes em Moscato, 1999). O pseudocódigo apresentado pode ser considerado genérico para os MAs baseados em busca local, caracterizando muitas implementações diferentes que podem ser encontradas na literatura.

```
início
    inicializePopulação Pop usando InicialPop( );          /* Geração da população inicial */
    paraCada indivíduo i ∈ Pop faça i = OperadorDeBuscaLocal(i);   /* Otimização da população*/
    paraCada indivíduo i ∈ Pop faça Avalie(i);             /* Avaliação da população */
    repita                                                 /* loop de geração */
        Para j = 1 até #recombinações faça                 /* fase de recombinação */
            selecioneParaRecombinação um conjunto Spar ⊆ Pop;
            filho = Recombine(Spar, inst);
            filho = OperadorDeBuscaLocal(filho);
            Avalie(filho);
            adicionarNaPopulação indivíduo filho;
        fimPara;                                           /* fim da fase de recombinação */
        Para j = 1 até #mutações faça                      /* início da fase de mutação */
            selecioneParaMutar um indivíduo i ∈ Pop;
            i_m = Mutação(i);
            i_m = OperadorDeBuscaLocal(i_m);
            Avalie(i_m);
            adicionarNaPopulação indivíduo i_m;
        fimPara;                                           /* fim da fase de mutação */
        Pop = SelecionePop(Pop);                           /* seleção */
        se convergênciaPop então Pop = RestartPop(Pop);    /* verifica diversidade */
    até critérioDeParada = true;          /* testa critério de parada do loop de gerações */
fim;
```

Figura 2 - Estrutura básica de um algoritmo memético.

Algumas palavras utilizadas no pseudocódigo são palavras reservadas que podem ser adaptadas conforme a implementação. Estas palavras podem ser entendidas como comandos para uma linguagem qualquer que possa interpretá-los de acordo com a área de aplicação do problema.

A primeira palavra reservada é **'inicializePopulação'**. Em alguns MAs a população dos agentes é inicializada por alguma heurística construtiva ou algoritmo com aleatoriedade, ou uma combinação de ambos, com o objetivo de obter boa solução inicial num tempo adequado. No pseudocódigo o algoritmo que gera uma solução inicial é representado por *InicialPop( )*.

A informação memética é adicionada ao algoritmo através do *OperadorDeBuscaLocal( )*. Este procedimento recebe como entrada uma solução (factível ou não) e tenta otimizá-la, fazendo uso de algum algoritmo baseado em busca local. Neste contexto, "otimizar" significa minimizar ou maximizar a função objetivo associada com o problema e/ou a medida de infactibilidade associada com a solução. O *OperadorDeBuscaLocal( )* retorna uma solução estritamente melhor ou retorna a solução passada como parâmetro, caso não conseguir otimizá-la.

Seguindo estritamente a definição de MA, no mínimo uma fase de recombinação envolvendo mais do que uma solução é aplicada. Se esta restrição for relaxada, o MA baseado em busca local poderia se reduzir a algum tipo de algoritmo de busca local de natureza aleatória. A cada fase de recombinação o número de recombinações é definido por *#recombinações*. Um MA pode recombinar os indivíduos da população corrente, considerando alguma informação de populações passadas, como em sistemas *blackboard* (Hogg e Williams, 1993) e a metodologia *Pocket-Current* utilizada em outros trabalhos, tais como Moscato e Tinetti (1992), Paechter et al. (1996) e Berretta e Moscato (1999). Observa-se que o pseudocódigo da Figura 2 não perde a generalidade desde que para todos os operadores presentes a população *Pop* pode ser composta por soluções *current* e *pocket* (a serem vistos na seção 3.1). Em adição, qualquer conjunto de subpopulações isoladas pode ser considerado como uma simples variável população.

A palavra reservada **'selecioneParaRecombinação'** representa a seleção de um subconjunto de indivíduos (chamado $Spar \subseteq Pop$) para ser usada como parâmetro de entrada para o procedimento *Recombine( )*. Existem MAs que usam uma função aleatória para selecionar os pais para recombinação, outros selecionam os melhores indivíduos da população corrente, enquanto procedimentos mais complexos tentam obter benefícios de usar população estruturada.

A função *Recombine( )* se encarrega de executar o operador de recombinação usando como pais os indivíduos selecionados por **selecioneParaRecombinação**. No pseudocódigo *inst* representa a instância do problema em questão. A decisão de adicionar ou não um indivíduo à população é obtida pelo comando **'adicionarNaPopulação'**, o qual difere muito de

implementação para implementação. Um critério geralmente utilizado é permitir que um indivíduo seja adicionado à população sob determinadas circunstâncias, como por exemplo evitar a adição de um indivíduo caso a população já tiver outro igual ou de mesma avaliação de função objetivo, a fim de manter a diversidade da mesma.

A cada *loop* de geração, um número de indivíduos (definido por *#mutações*) são selecionados (pelo comando **selecioneParaMutar**) e modificados pelo operador *Mutação( )*. Para o TSP, um dos procedimentos de mutação mais usados é baseado no movimento de dupla troca, chamado *change quad* por Kanellakis e Papadimitriou (1980). O indivíduo resultante da mutação é otimizado e adicionado ou não à população.

O procedimento *SelecionePop( )* seleciona os indivíduos que farão parte da próxima geração. O subconjunto não selecionado (a ser eliminado) nem sempre é determinado pela sua avaliação de função de avaliação (*Avalie( )*). Há características que podem influenciar de forma negativa ou positiva na seleção. Por exemplo, na dinâmica do algoritmo, é possível priorizar alguma medida que maximize a diversidade do conjunto a ser selecionado.

Uma conseqüência direta da aplicação de recombinação seguida de otimização é a perda da diversidade da população. A convergência da população é testada pelo comando **'convergênciaPop'**. Esse teste pode ser feito na população como um todo, ou em subpopulações, no caso de organização hierárquica da mesma. Este fenômeno é referenciado na literatura como *crise de diversidade* (da população ou subpopulação). Esta situação é detectada quando a população (ou subpopulação) é formada por indivíduos muito similares. Quando a crise de diversidade é detectada, a função *RestartPop( )* é usada para criar novos indivíduos e substituir alguns ou todos aqueles considerados similares. Em geral, preserva-se o melhor agente/solução encontrado (incumbente). Os novos indivíduos são subseqüentemente otimizados e avaliados e o *loop* de geração continua sua execução. O critério de parada pode ser um tempo ou um número de gerações predeterminado, assim como um procedimento dinâmico que detecta a estagnação da população.

2.6. REVISÃO BIBLIOGRÁFICA

O TSP tem despertado grande atenção dos pesquisadores nas últimas décadas. Algoritmos exatos têm sido propostos para ambos os casos: simétrico e assimétrico. Desde que o TSP foi provado pertencer à classe NP-difícil (Garey e Johnson, 1979), heurísticas e metaheurísticas ocuparam um importante lugar entre os métodos aplicados a grandes instâncias do problema. Uma visão abrangente de métodos de resolução para o TSP, pode ser encontrada em Balas e Toth (1985), Laporte (1992), Jünger et al.(1995).

Focando especificadamente os métodos exatos para o ATSP, o algoritmo *branch-and-bound* proposto por Miller e Pekny (1991) usa uma relaxação do ATSP, resultante no problema de designação (*Assignment Problem* - AP). São relatadas soluções ótimas encontradas para instâncias de até 500 mil cidades em tempos computacionais razoáveis, embora estas instâncias tenham sido aleatoriamente geradas com distribuição uniforme num dado intervalo. Instâncias geradas desta forma parecem ser fáceis para o algoritmo de Fischetti e Toth (1997), também baseado na relaxação-AP. Similarmente, Carpaneto et al. (1995) resolvem problemas com até 2 mil cidades em menos de 1 minuto. Entretanto, há instâncias no qual a relaxação-AP pode encontrar dificuldades, tais como as com distâncias quase simétricas (isto é, $d_{ij} \approx d_{ji}$ para todo i, j). Outra classe de instâncias difíceis provém de situações reais, oriundas de problemas de roteamento de veículos originados da entrega de produtos farmacêuticos na cidade de Bologna (Fischeti et al., 1994). Estas instâncias de ATSP, e o custo de suas soluções ótimas, estão disponíveis na TSPLIB (Reinelt, 1991) e estão incluídas nos testes computacionais do capítulo 3. A análise poliédrica de Fischetti e Toth (1997) obtém melhores resultados para estas classes de instâncias difíceis, quando comparadas com métodos que utilizam AP. Nos capítulos 3 e 4 serão relatados os resultados utilizando estas duas classes de instâncias difíceis.

Já para métodos heurísticos, uma gama de técnicas metaheurísticas tem sido propostas. Particularmente, publicações recentes têm apresentado algoritmos genéticos, aplicados a diversos problemas, obtendo boas soluções em tempos computacionais pequenos. Os melhores resultados têm sido obtidos pela combinação de algoritmos evolutivos com procedimentos de busca local, originando um algoritmo memético (*Memetic Algorithm* - MA) (Moscato, 1989, 1999; Moscato e Norman, 1992). Em um MA, uma busca local é aplicada a cada solução até esta se tornar um

mínimo local de uma certa vizinhança, ou seja, depois dos operadores de recombinação e mutação serem aplicados, o operador de busca local é aplicado na solução resultante.

Diversos MAs têm sido propostos para resolver o ATSP. Baseado na análise de resultados computacionais que utilizam esta técnica, pode-se dizer que um bom MA deve conter algumas características: i) operadores de recombinação e mutação apropriados e robustos; ii) um eficiente e rápido operador de busca local; iii) uma população estruturada hierarquicamente; iv) estruturas de dados e mecanismos de codificação apropriados. O primeiro é obviamente inerente a qualquer algoritmo genético, enquanto o segundo é crucial para os MAs, devido a 85% - 95% do tempo de CPU ser gasto, geralmente, com o procedimento de busca local. Muitos experimentos na literatura têm mostrado que a adoção de estruturas nas quais os agentes se relacionam conforme uma estrutura hierárquica, provaram ser mais eficazes quando comparados às implementações não estruturadas (Moscato, 1993; Gorges-Schleuter, 1997; França et al., 1999; Berretta e Moscato, 1999).

O algoritmo memético proposto por Freisleben e Merz (1996), denomidado GLS (*Genetic Local Search*) introduz um novo operador de recombinação, chamado de *DPX* (*Distance Preserving Crossover*). Neste algoritmo é usada uma variação da heurística de *Lin-Kernighan* como operador de busca local, testando instâncias do caso simétrico, e o operador de busca local *fast-3-Opt*, testando instâncias do caso assimétrico. Em um artigo mais recente, estes resultados foram melhorados, adotando uma série de mecanismos de implementação mais sofisticados que melhoraram a performance do método (Merz e Freisleben, 1997). Para o ATSP, eles adicionaram uma variante do procedimento *4-Opt* para a busca local. Em vez de utilizar procedimentos de busca local complexos, Gorges-Schleuter (1997) tem optado por investir em populações estruturadas espacialmente, usando uma busca local simples. Neste método, denomidado *Asparagos96*, a população é organizada em *demes* (populações locais) espacialmente dispostos em um anel, recebendo o nome de *ladder-population* (Gorges-Schleuter, 1989). Os operadores de busca local são o *2-Opt* e o *3-Opt*, para o STSP e ATSP, respectivamente. O operador de recombinação utilizado é o MPX2, o qual difere em alguns aspectos do também seu MPX (*Maximal Preservative Crossover*) (Gorges-Schleuter, 1989). Testes computacionais reduzidos são apresentados, limitando-se a poucas instâncias da TSPLIB. Através da análise destes resultados, conclui-se que para instâncias grandes do caso simétrico, *Asparagos96* é superior enquanto o GLS obtém melhores resultados para instâncias até 783 cidades. Para o caso

assimétrico, *Asparagos96* obteve melhores resultados que o GLS nas 5 instâncias relatadas.

Um novo operador de recombinação que pode ser usado em ambos os casos, STSP ou ATSP, foi recentemente proposto por Nagata e Kobayashi (1997). O *EAX* (*Edge Assembly Crossover*) pode gerar vários filhos a partir de dois pais. Informação heurística é adicionada durante o processo de recombinação de forma a não incluir uma fase de busca local. Entretanto, pode-se considerar este algoritmo um MA, dado que é agregada informação adicional durante o processo de evolução. Já Walters (1998) utiliza o conceito de *soft brood selection* no seu MA. Em *brood selection* os dois pais podem gerar muitos filhos e somente o melhor dentre eles, em relação à sua avaliação de função objetivo, é selecionado para permanecer na população. Na realidade, dependendo da implementação, mais de um filho pode ser selecionado e pode-se evitar aqueles que já tenham um indivíduo idêntico na população. O operador de recombinação, chamado DER (*Directed Edge Repair*), é combinado com o operador de busca local *3-Opt* modificado, e são apresentados resultados computacionais para um conjunto reduzido de instâncias da TSPLIB.

Capítulo

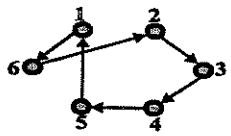
3

ALGORITMO MEMÉTICO APLICADO AO PROBLEMA DO CAIXEIRO VIAJANTE ASSIMÉTRICO

Neste capítulo será apresentado um algoritmo memético aplicado ao ATSP, com a proposta de uma nova busca local, denominada *Recursive Arc Insertion*, desenvolvida especialmente para resolver o ATSP. Quatro operadores de recombinação são testados com o propósito de escolher o mais adequado, sendo que um deles, o *Strategic Arc Crossover* - SAX, é uma adaptação do *Strategic Edge Crossover* - SEX, o qual já mostrou ser eficaz quando aplicado ao TSP simétrico (Moscato e Norman, 1992). A população é organizada hierarquicamente em uma árvore ternária completa com 13 indivíduos. Para representação da solução, utilizou-se um vetor duplo de n posições. Resultados computacionais são apresentados com população estruturada e não estruturada.

3.1. ESTRUTURA DE REPRESENTAÇÃO DA SOLUÇÃO

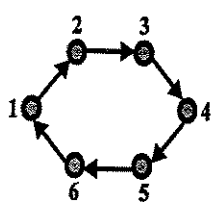
A estrutura adotada para representação de uma rota é um vetor de n posições, no qual as informações de cada cidade i da rota estão associadas à posição i do vetor. Associados a cada posição i do vetor, encontram-se os índices das cidades antecessora (*Prev*) e sucessora (*Next*) da cidade i. A Figura 3 permite visualizar uma rota e sua representação.



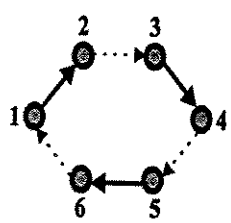
	1	2	3	4	5	6
Prev	5	6	2	3	4	1
Next	6	3	4	5	1	2

Figura 3 - Representação de uma solução para o ATSP.

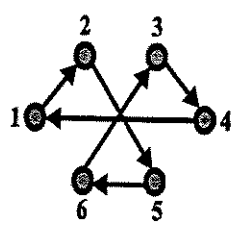
A representação em um vetor com *Prev* e *Next* é utilizada por apresentar duas características relevantes. Em primeiro lugar, por permitir que operações de troca entre 3 arcos sejam executadas em tempo O(1), como mostra a Figura 4 e, em segundo lugar, por sua simplicidade. Ainda, Java permite dimensionar vetores em tempo de execução, ou seja, seu tamanho não precisa ser previamente definido: para fazer uso dessa vantagem, a população, bem como todas as estruturas intermediárias, somente são alocadas após obterem o valor de n, para que tenham exatamente o tamanho que necessitam. Além disso, todos os vetores são, automaticamente, passados por referência às funções. Na liguagem C, essas características são obtidas utilizando-se vetores dinâmicos.



	1	2	3	4	5	6
Prev	6	1	2	3	4	5
Next	2	3	4	5	6	1



	1	2	3	4	5	6
Prev		1		3		5
Next	2		4		6	



	1	2	3	4	5	6
Prev	4	1	6	3	2	5
Next	2	5	4	1	6	3

Figura 4 - Troca entre 3 arcos executada sobre o vetor de representação de solução.

A troca entre 3 arcos é possível de ser executada em O(1) porque os índices das cidades envolvidas na troca podem ser acessadas diretamente no vetor de representação, e seus valores de *Prev* e *Next* alterados. Esta característica é muito importante, pois a maior parte do tempo de CPU (cerca de 85%) é gasto no procedimento de busca local, o qual executa recursivas trocas entre 3 arcos.

3.2. POPULAÇÃO DE AGENTES ORGANIZADA EM UMA ÁRVORE TERNÁRIA

A população é composta por 13 nós, denominados agentes, organizados em uma árvore ternária de 3 níveis (Moscato e Norman, 1992). Assim como a estrutura, o tamanho da população é fixo.

Um agente é uma estrutura composta por duas rotas, denominadas *Pocket* e *Current*, e seus respectivos custos: *PocCost* e *CurCost* (Moscato e Norman, 1992). Por custo de uma rota subentende-se a distância total por ela percorrida.

As operações de mutação, recombinação e busca local são aplicadas nas rotas *Currents*, enquanto que as rotas *Pockets* funcionam como uma memória associativa, pois armazenam rotas de custo reduzido já encontradas por alguma rota *Current* da população.

A organização em árvore ternária permite identificar 4 subpopulações distintas, sendo cada uma composta por quatro agentes. Uma subpopulação caracteriza-se por apresentar um agente líder, pertencente a uma hierarquia mais alta, e 3 agentes subordinados, pertencentes a uma hierarquia abaixo, mas no mesmo ramo da árvore. O agente 1, raiz da árvore, é o líder da subpopulação que tem como subordinados os agentes 2, 3 e 4. Já o agente 2, é líder da subpopulação 2 que tem como subordinados os agentes 5, 6, 7, e assim por diante. Cada agente da hierarquia intermediária (2, 3 e 4) é compartilhado por duas subpopulação distintas, pois são subordinados da subpopulação 1 e líderes das demais 3 subpopulações.

Pela Figura 5 é possível visualizar a organização da população em subpopulações de agentes.

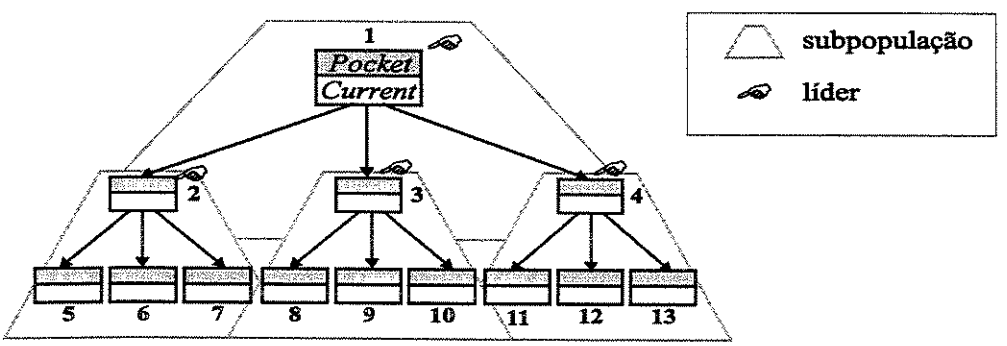


Figura 5 - Estrutura da população.

A cada geração, três operações são executadas sobre a árvore: *UpdatePocket*, *PocketPropagation* e *OrderChildren*. Estas operações são as responsáveis pela organização descrita anteriormente e todas são executadas em tempo O(1).

UpdatePocket: como as operações de mutação, recombinação e busca local são aplicadas sobre a rota *Current*, esta pode se tornar melhor que a *Pocket* do agente correspondente (*CurCost* < *PocCost*). Nesse caso, as rotas *Pocket* e *Current* são trocadas, fazendo com que o *Pocket* sempre contenha uma melhor rota que o *Current*.

PocketPropagation: sempre que um agente subordinado obtiver uma rota *Pocket* melhor que a do líder correspondente, suas rotas *Pockets* são trocadas, de forma que o líder sempre contenha o melhor *Pocket* da subpopulação.

Estas duas operações são suficientes para garantir que a melhor rota da população esteja armazenada na rota *Pocket* do agente 1 (raiz da árvore).

OrderChildren: mantém os agentes subordinados de cada subpopulação organizados em ordem crescente de seus respectivos custos de rota *Pocket* (*PocCost*). A título de exemplo, para a subpopulação 3, o *Pocket* do agente 8 é melhor do *Pocket* do agente 9 que, por sua vez, é melhor do *Pocket* do agente 10.

A organização em árvore ternária pode ser entendida como uma variação do modelo de ilhas (Gorges-Schleuter, 1989), mas em vez de estabelecer um mecanismo de migração, há trocas de agentes entre subpopulações durante a manutenção da hierarquia.

3.2.1. População inicial

A população inicial pode ser inicializada aleatoriamente, por algum método construtivo, ou uma combinação de ambos.

No caso desta implementação, a população é inicializada com a heurística do "vizinho mais próximo" (*Nearest Neighbor* - NN). Como característica dessa heurística, as últimas cidades inseridas na rota criada geralmente estão ligadas à vizinhas distantes. Desta forma, se as últimas

cidades envolvidas na construção de uma rota forem consideradas cidades de partida na construção de novas rotas, aumenta a chance de introduzir novos arcos na população inicial, tornando-a mais diversa.

Inicialmente o *Pocket* e o *Current* do agente raiz (agente 1 na Figura 5) da árvore são construídos considerando cidades de partida aleatórias. Para cada uma das quatro subpopulações, as três últimas cidades envolvidas na construção das rotas *Pocket* e *Current* do líder serão consideradas cidades de partida para a construção das rotas *Pocket* e *Current* dos três respectivos subordinados.

A inicialização aleatória foi testada, mas embora a qualidade de solução se mantivesse a mesma, o tempo computacional total era maior. Observou-se que, para a população aleatória ter uma qualidade de solução igual à obtida pela heurística NN, um número x de gerações era necessário. Ao final, o tempo total gasto era igual ao tempo total gasto com a população inicializada com a heurística do NN, mais o tempo gasto com as x gerações.

3.2.2. Adição de novos indivíduos

A fim de manter a diversidade da população, o procedimento **addInPopulation** verifica se há algum indivíduo na população (*Pocket* ou *Current*) com o mesmo custo. Se não houver, o novo indivíduo passa a fazer parte da população, caso contrário, é descartado.

3.3. OPERADORES DE RECOMBINAÇÃO

Sabe-se que a performance de um algoritmo memético depende da sinergia entre os operadores de busca local e recombinação utilizados. Da vasta quantidade de operadores de recombinação propostos, quatro foram selecionados para serem testados com a busca local descrita na próxima seção:

1. SAX: uma adaptação do SEX , originalmente proposto para o caso simétrico (Moscato e Norman, 1992);
2. DPX: operador de recombinação utilizado por Freisleben e Merz (1996);
3. MFNN: proposto por Holstein e Moscato (1999);
4. UNN: uma adaptação do crossover uniforme proposto para um vetor de booleanos.

Outros dois operadores são descritos: AinterB e AmenosB. Embora estes operadores não façam parte dos resultados computacionais, suas implementações estão disponíveis no *framework* a que faz parte o algoritmo memético descrito neste capítulo.

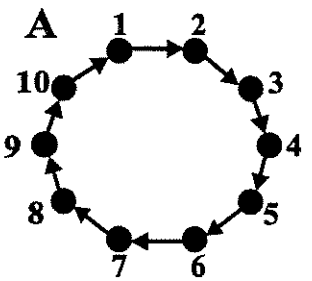
3.3.1. *Strategic Arc Crossover* (SAX)

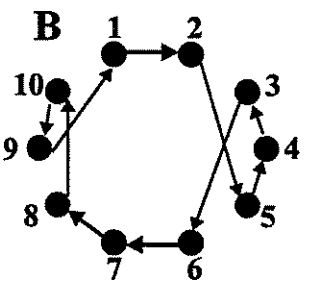
Historicamente, SEX (*Strategic Edge Crossover*) pode ser considerado como um descendente direto do *Enhanced Edge Recombination* de Karp (1977, 1979). O SAX é uma adaptação do SEX para o caso assimétrico. O objetivo deste operador é gerar um filho que tenha o maior número possível de arcos provenientes dos pais. O processo de gerar um novo indivíduo, a partir de dois pais pré-selecionados, pode ser subdividido em 3 fases:

a) construção do *ArcMap* - uma estrutura intermediária que auxilia a geração das *strings*. Por *string* subentende-se um caminho de η cidades, sendo que $1 \leq \eta \leq n$;

b) aplicação de *CreateStrings* - um procedimento responsável pela geração de *strings* a partir da leitura do *ArcMap*. Ao final desse, toda cidade deve pertencer a uma e somente uma *string*;

c) ao final, um procedimento denominado *PatchString* é aplicado, a fim de unir as *strings* e tornar a rota factível.





	1	2	3	4	5	6	7	8	9	10
Prev	10	1	2	3	4	5	6	7	8	9
Next	2	3	4	5	6	7	8	9	10	1

	1	2	3	4	5	6	7	8	9	10
Prev	9	1	4	5	2	3	6	7	10	8
Next	2	5	6	3	4	7	8	10	1	9

Figura 6 - Rotas A e B selecionadas para recombinação.

Considere o exemplo mostrado na Figura 6 onde A e B são os pais (elementos do conjunto S_{par}) selecionados pelo procedimento **selectToMerge**. O *ArcMap* gerado pela leitura das rotas de A e B é a matriz da Figura 7. *ArcMap* é composto por 6 colunas e *n* linhas. As colunas 1 e 2 contêm os índices das cidades sucessoras da cidade *i* (índice da linha) em A e B, respectivamente. Caso a mesma cidade *j* seja sucessora de *i* em A e B, o *ArcMap*[*i*][1] = *j* e *ArcMap*[*i*][2] permanece vazio. As colunas 4 e 5 são preenchidas de forma similar, com a diferença de que se referem às cidades antecessoras. As colunas 3 e 6 armazenam, para cada cidade *i*, o número de cidades sucessoras (indicadas nas colunas 1 e 2 do *ArcMap*) e antecessoras (indicadas nas colunas 4 e 5 do *ArcMap*), respectivamente.

	Sucessor			Antecessor		
	1	2	3	4	5	6
1	2		1	10	9	2
2	3	5	2	1		1
3	4	6	2	2	4	2
4	5	3	2	3	5	2
5	6	4	2	4	2	2
6	7		1	5	3	2
7	8		1	6		1
8	9	10	2	7		1
9	10	1	2	8	10	2
10	1	9	2	9	8	2

Figura 7 - *ArcMap* gerado a partir dos pais A e B.

Baseado nas informações contidas no *ArcMap*, o procedimento *CreateStrings* gera todas as *strings* possíveis, como mostrado na Figura 8. Estas *strings* são copiadas em uma nova matriz,

denominada *ChildTour*, e as cidades inicial e final de cada *string* são armazenadas num vetor duplo denominado *EndPoints*. A função *RandBetween(i, j)* escolhe *i* ou *j* com igual probabilidade.

Passo 1) faça *VisitedCities* = 0; *#strings* = 0.

Passo 2) se *VisitedCities* = *n*, vá para o passo 6. Caso contrário, selecione aleatoriamente uma cidade *i* ainda não visitada e faça: *#strings* = *#strings* + 1, *EndPoints*[*#strings*][1] = *i*, *VisitedCities* = *VisitedCities* + 1.

Passo 3) retiram-se todas as referências de *i* do *ArcMap* e atualizam-se os contadores das colunas 3 e 6. Se a coluna 1 estiver vazia e a coluna 2 não, mova o índice da coluna 2 para a coluna 1. Da mesma forma, se a coluna 4 estiver vazia e a coluna 5 não, mova o índice da coluna 5 para coluna 4.

Passo 4) se *ArcMap*[*i*][3] = 0, faça *EndPoints*[*#strings*][2] = *i* e vá para o passo 2. Se *ArcMap*[*i*][3] = 1, faça *j* = *ArcMap*[*i*][1] e vá para o passo 5. Se *ArcMap*[*i*][3] = 2 faça *j* = *RandBetween*(*ArcMap*[*i*][1], *ArcMap*[*i*][2]) e vá para o passo 5;

Passo 5) faça *VisitedCities* = *VisitedCities*+1, *ChildTour*[*i*][*Next*] = *j*, *ChildTour*[*j*][*Prev*] = *i*, *i* = *j* e vá para o passo 3;

Passo 6) execute *PatchString* (*#strings*, *EndPoints*, *ChildTour*) a fim de reconectar as *strings* contidas em *Childtour*.

Figura 8 - Procedimento CreateStrings.

No passo 3, para retirar todas as referências da cidade *i* do *ArcMap*, utilizam-se as colunas 4 a 6. Este passo é o único que utiliza as informações contidas nestas colunas. Para saber quantas vezes a cidade *i* está referenciada nas colunas 1 e 2, basta verificar o valor de *ArcMap*[*i*][6] e para saber em que linhas das colunas 1 e 2 a cidade *i* está referenciada, basta verificar os valores de *ArcMap*[*i*][4] e *ArcMap*[*i*][5]. Para retirar as referências da cidade *i* das colunas 4 e 6, o mesmo procedimento descrito acima é executado, com a diferença de que as 3 primeiras colunas são consultadas.

Suponha que a primeira cidade selecionada no passo 2 fosse 1, e que a seqüência de cidades escolhidas no passo 4 pela função *RandBetween( )* fosse 5, 6 e 9, quando *ArcMap*[*i*][3] = 2. Dessa forma, a primeira *string* gerada seria 1→2→5→6→7→8→9→10. Supondo que a segunda cidade selecionada no passo 2 fosse a cidade 3, a *string* 3→4 seria gerada. Ao final de *CreateStrings*, *ChildTour* e *EndPoints* estariam preenchidos como mostra a Figura 9.

ChildTour

	1	2	3	4	5	6	7	8	9	10
Prev		1		3	2	5	6	7	8	9
Next	2	5	4		6	7	8	9	10	

EndPoints

1	2
1	3
10	4

Figura 9 - ChildTour e EndPoints gerados pelo SAX.

Para factibilizar a rota contida em *ChildTour*, um procedimento denominado *PatchString* faz a leitura de *EndPoints* e une as *strings* contidas em *ChildTour*. O *PatchString* utilizado é baseado na heurística construtiva do "vizinho mais próximo" (*Nearest Neighbor* - NN). O *PatchString* é o mesmo para 5 dos 6 operadores descritos, com exceção do DPX.

É importante observar que todos os arcos que compõem as *strings* (geradas por *CreateStrings*) pertencem a pelo menos um dos pais. Isso acontece porque ao arcos dos pais A e B são representados através da ligação da cidade *i* (linha do ArcMap) com as cidades indicadas na colunas 1, 2, 4 e 5 correspondentes a esta linha, ou seja, o indice *i* das linhas do *ArcMap* formam arcos (i, j) com as cidades *j* localizadas em *ArcMap*[*i*][1] e *ArcMap*[*i*][2] e formam arcos (j, i) com as cidades *j* localizadas em *ArcMap*[*i*][3] e *ArcMap*[*i*][4]. Na medida em que estes arcos vão sendo incluídos nas *strings*, as cidades que formam estes arcos com os índices das linhas correspondentes são excluídos do *ArcMap*. Portanto, quando não é possível conectar uma cidade na *string* $x \rightarrow ... \rightarrow y$, significa dizer que os arcos (y, a_2) (arco pertencente à rota A) e (y, b_2) (arco pertencente à rota B) não podem ser incluídos (no algoritmo da Figura 8 a não inclusão está associada a obter *ArcMap*[*i*][3] = 0).

Ainda, há somente um caso em que um arco comum entre dois pais pode não ser herdado pelo filho. Suponha que o arco (i, j) seja um arco comum entre os pais. Caso a cidade *j* seja selecionada (no passo 2 de *CreateStrings*) como cidade inicial de uma *string*, o arco (i, j) não poderá ser inserido nessa *string* pois uma subrota seria formada. Nesse caso, o arco só poderá pertencer ao filho, caso o procedimento *PatchString* faça essa inserção.

Ainda, o número de *strings* formadas não é conhecido *a priori*, mas através de observações práticas pode-se afirmar que o número de *strings* formadas é inversamente proporcional ao número de geracões.

Na versão original do SEX, após o algoritmo não poder conectar a cidade *y* da *string* $x \rightarrow ... \rightarrow y$ a uma cidade *j*, o algoritmo faz o caminho inverso: adiciona tantas cidades antes da cidade *x* quanto possível, fazendo com que o tamanho da *string* aumente. Na versão do SAX utilizada nos resultados computacionais, este caminho inverso não era feito. Mas na versão final disponível no Memepool, o caminho inverso é implementado.

3.3.2. *Distance Preserving Crossover* (DPX)

A idéia do DPX (Freisleben e Merz, 1996) provém da observação que a distância média entre rotas que são ótimo locais é semelhante à distância média entre uma rota de ótimo local e uma rota de ótimo global. Neste caso, distância entre duas rotas representa o número de arcos que pertencem à uma rota mas não à outra. O objetivo do operador DPX é gerar um filho com distância dele a ambos pais igual à distância dos pais entre si. Inicialmente todos os genes comuns aos dois pais são copiados para o filho e a rota é factibilizada por uma heurística gulosa - uma variação da heurística NN, com a diferença de que nenhum arco pertencente a somente um dois pais é incluído na solução

Mais formalmente, considere $A \cap B$ o conjunto de arcos comuns aos pais A e B. O conjunto de arcos utilizados para gerar o filho é $A \cap B + E - (A-B) \cup (B-A)$, onde E representa o grafo completo de arcos. Um exemplo detalhado pode ser encontrado em Merz e Freisleben (1997) e em Freisleben e Merz (1996).

Nesta descrição utilizou-se o termo 'arco' e não 'aresta' que foi o termo originalmente proposto para o caso simétrico.

3.3.3. *Multiple Fragment - Nearest Neighbor Repair Edge Recombination* (MFNN)

Este operador de recombinação (Holstein e Moscato, 1999) introduz uma estratégia diferente na construção do filho. O filho gerado recebe inicialmente todos os arcos comuns aos dois pais, ou seja, pertencentes ao conjunto $A \cap B$ (assim como no DPX). Num próximo passo, todos os arcos pertencentes a $(A-B) \cup (B-A)$ são adicionados em ordem não-decrescente, de forma a priorizar os arcos de menor custo. Ainda, uma inserção somente é efetivada caso esta não causar a formação de subrotas e o grau de inserção e de adjacência das cidades envolvidas na inserção não se torne maior que 1. Em resumo, após os arcos comuns aos pais A e B serem copiados para o filho, testa-se a inserção dos arcos existentes em apenas um dos pais.

Suponha que no exemplo da Figura 6, os arcos do conjunto (A-B)∪(B-A) ordenados segundo seus custos sejam: (2,3), (8,10), (4,3), (4,5), (5,4), (9,10), (3,6), (10,1), (3,4), (10,9), (9,1), (8,9), (5,6) e (2,5). Segundo o operador MFNN, as duas *strings* formadas seriam 1→2→3→6→7→8→10→9 e 4→5. Neste caso, as matrizes correspondentes *ChildTour* e *EndPoints* podem ser visualizadas na Figura 10.

ChildTour

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Prev		1	2		4	3	6	7	10	8
Next	2	3	6	5		7	8	10		9

EndPoints

1	**2**
1	4
9	5

Figura 10 - ChildTour e EndPoints gerados pelo MFNN.

3.3.4. *Uniform Nearest Neighbor* (UNN)

No operador de recombinação uniforme (Syswerda, 1989; Goldberg, 1989), um filho é gerado através de uma escolha aleatória entre os genes dos pais. Cada gene do filho depende do valor aleatório de uma variável boleana: se verdadeiro (*true*), o gene do pai A é copiado para o filho, caso contrário (*false*), o filho recebe o gene do pai B.

O operador de recombinação UNN foi adaptado para ser aplicado a uma rota e não a um vetor de boleanos. Nesta implementação, um filho é gerado da seguinte forma: para cada gene *i*, uma variável boleana é gerada. Se esta variável for verdadeira (*true*), então o arco que parte da cidade correspondente *i* do pai A é copiado para o filho, respeitando algumas restrições: não formação de subrotas e o grau de inserção e de adjacência das cidades envolvidas ≤ 1. Se a variável for falsa (*false*), então o arco correspondente do pai B é copiado para o filho, também respeitando as restrições acima descritas. Em ambos os casos, se a violação ocorrer, testa-se a inserção do arco proveniente do outro pai.

Considere a Figura 6, supondo que a seqüência de variáveis boleanas geradas para os 10 genes seja 1010010101, as *strings* formadas seriam: 1→2→5→6→7→8→9→10 e 3→4.

ChildTour

	1	2	3	4	5	6	7	8	9	10
Prev		1		3	2	5	6	7	8	9
Next	2	5	4		6	7	8	9	10	

EndPoints

1	2
1	3
10	4

Figura 11 - ChildTour e EndPoints gerados pelo UNN.

3.3.5. Operadores AinterB e AmenosB

O operador AinterB copia para o filho somente os arcos comuns aos dois pais (arcos do conjunto A∩B) e o operador AmenosB copia para o filho todos os arcos do pai A que não são comuns ao pai B (arcos do conjunto A-B).

Considerando a Figura 6, as *strings* geradas pelo operador AinterB seriam apenas duas: 1→2 e 6→7→8. Já para o operador AmenosB, as *strings* formadas seriam 3: 2→3→4→5→6, 7, 8→9→10→1.

ChildTour

	1	2	3	4	5	6	7	8	9	10
Prev	10		2	3	4	5			8	9
Next		3	4	5	6			9	10	1

EndPoints

1	2	3
2	7	8
6	7	1

Figura 12 - ChildTour e EndPoints gerados pelo operador AmenosB.

Observe que neste exemplo uma *string* gerada é composta apenas pela cidade 7. Neste caso, no vetor *ChildTour* a cidade 7 não apresenta cidade antecessora e nem sucessora. No vetor *EndPoints* a cidade 7 aparece nas duas posições, pois como é formada por apenas uma cidade, 7 representa a cidade inicial e a final da *string*.

3.4. NOVO OPERADOR DE BUSCA LOCAL: *RECURSIVE ARC INSERTION* (RAI)

Há muitas vizinhanças usadas em algoritmos de busca local para o TSP. A literatura normalmente referencia a vizinhança de Lin-Kernighan como a melhor (Lin e Kernighan, 1973). Variações desta vizinhança têm mostrado boa performance, particularmente em instâncias grandes de STSP (Merz e Freisleben, 1997). Para as instâncias assimétricas, a heurística proposta por Kanellakis e Papadimitriou (1980) (e algumas variantes dela) tem mostrado ser eficaz. Entretanto, muitos pesquisadores preferem usar uma vizinhança simples, como a *k-Opt*, devido à sua baixa complexidade computacional.

Nesta seção, como *OperadorDeBuscaLocal( )* do MA proposto, utilizaremos um novo procedimento denominado Inserção Recursiva de Arcos (*Recursive Arc Insertion* - RAI). Desde que a busca executa a retirada de 3 arcos e a inserção de outros 3, o movimento básico é sempre uma vizinhança *3-Opt*. Como a maior parte do tempo de CPU, cerca de 85%, é gasto na fase de busca local, seu objetivo consiste em restringir a atuação exclusivamente a algumas cidades e, a partir delas, otimizar recursivamente outras detectadas durante a execução do procedimento.

Este novo operador de busca local necessita de pontos iniciais para iniciar a otimização. No caso desta implementação, estes pontos são as cidades marcadas com *don't look bits* (Bentley, 1992) durante as fases de recombinação e mutação. Os *don't look bits* são uma estrutura de dados com *flags* associados a cada cidade do problema. Usando sua idéia original, inicialmente todos *flags* não estão marcados (*false*). Se para alguma cidade nenhum movimento de melhora pode ser efetivado pelo operador de busca, o *flag* correspondente a esta cidade é marcado (*true*) e esta será desconsiderada como cidade inicial para o procedimento de busca da próxima iteração.

A utilização dos *don't look bits*, representados como um vetor de boleanos nesta implementação, mantém a idéia original, mas diferencia em alguns aspectos. Inicialmente todos são marcados como 'verdadeiros'. Durante o procedimento de recombinação, todas as cidades pertencentes à matriz de *EndPoints* têm seus *don't look bits* marcados como 'falsos'. Durante a mutação, a cidade *j* escolhida para ser *Next(i)* (onde *i* é a cidade selecionada para mutação), terá seu *don't look bit* marcado como 'falso'. As cidades marcadas com *don't look bits* 'falsos' serão consideradas críticas e serão, uma a uma, as cidades iniciais para o procedimento de busca local RAI. Antes da aplicação do operador de recombinação e depois da aplicação do operador de

busca local, todas as cidades têm seus *don't look bits* marcados como 'verdadeiros'. Para cada rota, cada cidade marcada como ponto crítico será cidade inicial para execução do operador de busca local. Pela natureza recursiva do algoritmo, uma cidade pode ser usada mais de uma vez como cidade inicial para a execução do procedimento de busca local (veja passo 4 da Figura 14).

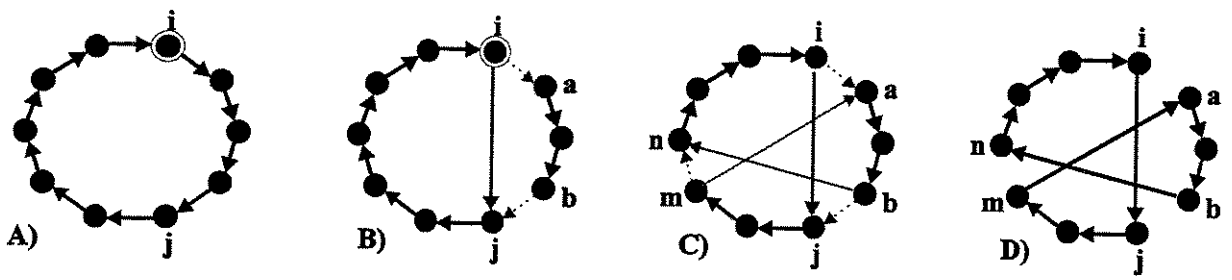


Figura 13 - Passos da busca local RAI: **A)** Passo 1; **B)** Passo 2; **C)** Passo 3; **D)** Passo 4.

Para cada cidade *i*, define-se uma matriz bidimensional (*s-vizinhos[s][n]*) *s-vizinhos*(*i*) como o conjunto das *s* cidades que são extremos dos *s* menores arcos adjacentes de *i* (as cidades contidas em *s-vizinhos*(*i*) são ditas sucessoras de *i*). Analogamente, a matriz bidimensional *p-vizinhos*(*i*) é o conjunto das *p* cidades mais próximas incidentes em i (as cidades contidas em *p-vizinhos*(*i*) são ditas antecessoras de *i*).

Na Figura 14 são descritos os passos do algoritmo aplicados à Figura 13. Vale lembrar que este procedimento é executado para cada cidade *i* marcada com *don't look bit* 'falso'.

Passo 1: (Figura 13A) Escolha $j \in$ *s-vizinhos*(*i*). Se j = *Next*(*i*), faça *Cont* = 1 e vá para o passo 5. Caso contrário, faça *Cont* = 0 e vá para o passo 2;

Passo 2: (Figura 13B) Teste a inserção do arco (i, j) e a remoção dos arcos (i, a) e (b, j), onde a = *Next*(*i*) e b = *Prev*(*j*). Calcule $\Delta_1 = d_{ia} + d_{bj} - d_{ij}$. Faça *Cont* = *Cont*+1 e vá para o passo 3;

Passo 3: (Figura 13C) Partindo do arco (*j*, *Next*(*j*)), selecione o primeiro arco $(m, n) \neq (i, j)$ na subrota {j, ..., i, j} tal que $\Delta_2 < \Delta_1$, $\Delta_2 = d_{ma} + d_{bn} - d_{mn}$. Se algum arco (m, n) é encontrado, tal que $\Delta_2 < \Delta_1$, vá para o passo 4. Se $\Delta_2 \geq \Delta_1$ para todos os arcos na subrota {j, ..., i, j}, então vá para o passo 5.

Passo 4: (Figura 13D) Execute o movimento, isto é, insira os arcos (i, j), (m, a) e (b, n) e remova os arcos (i, a), (b, j) e (m, n). Repita o procedimento recursivamente usando como cidades iniciais (referenciada como *i* no passo 1) as cidades *a*, *b*, *n*, *m* e *j*, nesta ordem. Vá para o passo 5;

Passo 5: Se Cont=2, pare. Caso contrário, selecione $j \in$ *p-vizinhos*(*i*). Invertem-se os índices *i* e *j* (fazendo $x = i$; $i = j$; $j = x$) e vá para o passo 2;

Figura 14 - Procedimento recursivo básico do RAI.

As matrizes de *s-vizinhos*(*i*) e *p-vizinhos*(*i*) são calculadas uma única vez, no início do programa, visto que estas não mudam durante a execução do mesmo. As cidades do conjunto *s*,

assim como as do conjunto p, estão ordenadas de forma não-decrescente de distância da cidade i, a fim de facilitar a seleção da cidade j nos passos 1 e 5. Nos experimentos que constam desta dissertação, s e p são fixados em 5 para as instâncias até 500 cidades e em 3 para as maiores. A escolha da cidade j é feita da seguinte forma: há 40% de chances que a cidade j seja a primeira da lista de vizinhanças (forma o menor arco partindo ou chegando em i, considerando a matriz de *s-vizinhos*(s) ou *p-vizinhos*(i), respectivamente), 30% de chances de ser a segunda, 15% a terceira, 10% a quarta e 5% a quinta. Estes valores foram estipulados após vários valores terem sido utilizados em testes de performance e são baseados nas informações empíricas fornecidas por Walter (1998): numa solução ótima, 40% das cidades estão ligadas a suas vizinhas mais próximas, 80% estão ligadas a uma das 3 vizinhas mais próximas e 95% estão ligadas a uma das 10 vizinhas mais próximas.

Com o objetivo de não fixar uma percentagem para cada cidade vizinha, um procedimento adaptativo foi testado, no qual todos os vizinhos iniciavam com a mesma percentagem e, na medida que a seleção de um obtivesse sucesso na aplicação da busca, sua percentagem era aumentada (em taxas pequenas). Este procedimento se adaptou muito bem ao algoritmo, mas não registrou melhora nos resultados computacionais. Optou-se, então, por deixar os valores estipulados acima, visto que são de implementação mais simples.

Note que durante a execução do passo 3, a inserção somente acontece se a soma dos custos dos arcos a serem inseridos é menor que a soma do custo dos arcos a serem removidos. Como uma troca está vinculada à diminuição do custo da rota, a cidade i (cidade inicial do procedimento) pode ser ligada a duas novas cidades (uma sucessora e uma antecessora), ou somente a uma nova sucessora, ou somente a uma nova antecessora, ou a nenhuma. Caso o movimento seja efetivado, toda inserção do arco (i, j) e a exclusão dos arcos (i, a) e (b, j) transforma a rota em uma subrota e uma *string*. Esta última pode conter muitos, um ou nenhum arco (no caso de $a = b$). Em todos os casos, ao final do procedimento, é necessário que a factibilidade da rota seja re-estabelecida.

No passo 3, outra alternativa para a busca seria considerar para exclusão o arco (m, n) da subrota que obtivesse o maior Δ_2, mas experimentos práticos mostraram que além do tempo médio por geração aumentar, há perda de diversidade. Testes envolvendo um valor constante de arcos também não obtiveram sucesso.

Acredita-se que a característica marcante desta busca, e talvez a principal razão de sua boa performance, é combinar a escolha gulosa dos arcos envolvidos no cálculo de Δ_1 com a diversidade gerada pelos arcos de Δ_2.

No caso da implementação aqui descrita, a nova busca local *Recursive Arc Insertion* é aplicada ao ATSP, não impedindo que seja utilizada em outros domínios. Sua aplicação é válida para problemas suscetíveis a permutações, tal como o problema de *scheduling* (Mendes et al., 1999).

Antes que esta busca fosse utilizada, heurísticas como 3-opt, inserção-de-uma-cidade e diversas variações destas foram implementadas e testadas. Não foram construídas as tabelas de resultados utilizando tais buscas, mas pode-se afirmar que para as instâncias com mais de 100 cidades a solução ótima não era encontrada e, para instâncias menores, quando a solução ótima era encontrada, o tempo era dispendioso.

3.5. MANUTENÇÃO DA DIVERSIDADE

Robustez é um elemento chave de qualquer metaheurística. É desejável que o método mostre ser eficaz, obtendo soluções ótimas ou sub-ótimas de boa qualidade em pequenos tempos computacionais, de acordo com o tamanho e características da instância. Outro aspecto da robustez, não menos relevante, é a habilidade do método em resolver um conjunto de instâncias com os mesmos parâmetros previamente usados na resolução de outras instâncias de tamanho e características diferentes. Em todos os experimentos que constam desta dissertação, o tamanho da população manteve-se sempre em 13 agentes (13 indivíduos que funcionam como memória e 13 indivíduos sobre os quais o MA executa os operadores de recombinação e busca local). Se comparado com outros MAs propostos para o TSP, uma população de 13 indivíduos é considerada muito pequena. A vantagem de se ter uma população pequena é que o tempo gasto por geração é menor; por outro lado, como o número de indivíduos é menor, a população tem mais chances de perder a diversidade prematuramente. Para evitar isso, a implementação aqui descrita se utiliza de alguns procedimentos descritos nas próximas subseções.

3.5.1. Procedimento selecioneParaRecombinação

O procedimento **selecioneParaRecombinação** é responsável pela seleção do conjunto de agentes *Spar* utilizado como parâmetro de entrada para o procedimento *Recombine( )*. Uma escolha adequada dos agentes é requerida para preservação da diversidade da população.

Considere a notação abaixo usada para cada subpopulação da árvore ternária da Figura 5.

- *líder*: o agente líder de uma subpopulação;
- *filho1*, *filho2*, *filho3*: os três agentes subordinados de uma subpopulação;
- *Pocket* (*i*): a rota *Pocket* do agente *i*;
- *Current* (*i*): a rota *Current* do agente *i*.

Na implementação relatada neste capítulo, a recombinação resulta num novo indivíduo que substitui alguma rota *Current* da população. Desta forma, a recombinação do *Pocket* do primeiro subordinado com o *Current* do segundo subordinado substitui a rota *Current* do agente raiz da árvore. As rotas *Currents* das 4 subpopulações, obtidos através da fase de recombinação, seguem a seguinte política:

Current do líder ⇐ *Recombine* (*Pocket* (*filho2*), *Pocket* (*filho3*));

Current do *filho1* ⇐ *Recombine* (*Pocket* (*líder*), *Current* (*filho2*));

Current do *filho2* ⇐ *Recombine* (*Pocket* (*filho1*), *Current* (*filho3*));

Current do *filho3* ⇐ *Recombine* (*Pocket* (*filho2*), *Current* (*filho1*)).

A designação de *filho1*, *filho2* e *filho3* para os 3 subordinados é feita de forma aleatória a cada recombinação, para cada subpopulação. Exemplificando, para uma geração qualquer *x*, o *filho*1 pode ser o primeiro subordinado da subpopulação 1, o terceiro da subpopulação 2, o primeiro da subpopulação 3 e o segundo da subpopulação 4.

3.5.2. Procedimento *selecioneParaMutar*

No contexto de MAs baseados em busca local, o operador de mutação tem o objetivo único de manter a diversidade da população. Para todos os testes computacionais que constam desta dissertação *#mutações* = 5. O procedimento **selecioneParaMutar** escolhe aleatoriamente cada agente de *Pop* com 5% de probabilidade de aplicar a mutação. No caso do agente ter sido selecionado, uma cidade *h* da rota *Current* deste agente é escolhida aleatoriamente e o operador de mutação é aplicado 5 vezes (*#mutações* = 5). A cidade *i* descrita no procedimento da Figura 15 é considerada a cidade *h* na primeira execução, *h+1* na segunda execução, *h+2* na terceira, *h+3* na quarta e *h+4* na quinta (caso *h+x* seja maior que *n*, continuar a seleção a partir do índice 1). Note que a escolha consecutiva dos índices não significa cidades consecutivas na rota. Então, o operador *Mutação( )* executa o procedimento de inserção-de-uma-cidade da seguinte forma: uma cidade $j \neq Next(i)$, selecionada aleatoriamente, é inserida entre *i* e *Next(i)* e o arco (*Prev(j)*, *Next(j)*) é inserido. O procedimento executado pelo operador de mutação pode ser visualizado na Figura 15.

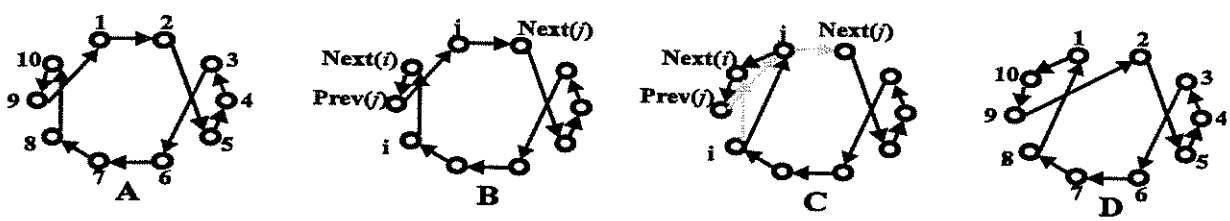


Figura 15 - Operação de Mutação.

No exemplo da Figura 15, supõe-se que a rota *Current* do agente selecionado seja a Figura 15A e que a cidade *h* aleatoriamente selecionada seja a cidade 8. Desta forma, as cidades que sofrerão mutação serão as cidades 8, 9, 10, 1 e 2. A primeira cidade a sofrer mutação é a cidade 8. Como $i = 8$ e supondo que a cidade *j* selecionada aleatoriamente seja a cidade 1, então $Next(i) = 10$, $Prev(j) = 9$ e $Next(j) = 2$, conforme pode ser visto na Figura 15B. Ao inserir *j* entre (*i*, *Next(i)*), os arcos (*i*, *Next(i)*), (*Prev(j)*, *j*) e (*j*, *Next(j)*) automaticamente são excluídos (Figura 15C). Para factibilizar a rota, o arco (*Prev(j)*, *Next(j)*) é incluído. A rota final é apresentada na Figura 15D.

Como o operador de *Mutação( )* é aplicado 5 vezes para cada rota *Current* selecionada para mutação, 5 cidades serão marcadas como 'falso' no vetor de *don't look bits*, pois somente a

cidade j do procedimento é marcada como 'falso'. Para aplicação do operador de mutação na Figura 15, a cidade $j = 1$ terá seu *don't look bit* marcado como falso. Aplicando-se o operador de mutação considerando as cidades $h = 9$, 10, 1 e 2, mais quatro cidades terão seus *don't look bits* marcados como falsos (as cidades j selecionadas aleatoriamente em cada aplicação do operador). As cidades são consideradas como seqüência dos índices do vetor para que pontos não consecutivos da rota sejam considerados sem gasto de tempo na geração de números randômicos (que seriam outra alternativa).

3.5.3. *Restart* da população

Depois de decorrido um certo número de gerações, naturalmente a população perde diversidade. O critério utilizado para detectar a crise de diversidade de uma subpopulação (**convergênciaPop** no pseudocódigo da Figura 2) é verificar se o número de vezes que o procedimento *UpdatePocket* foi executado sem que o *Current* do agente líder tenha sido trocado pelo *Pocket* correspondente (*PocCost* < *CurCost*) ultrapassou um limite pré-estipulado. Quando este número é igual a 30 (limite utilizado na implementação), o procedimento de mutação é aplicado na subpopulação correspondente e o contador é reinicializado. Neste caso, a mutação é aplicada da mesma forma e com as mesmas probabilidades descritas na seção anterior, com a diferença que é a rota *Pocket*, e não a rota *Current* do agente selecionado, que sofre mutação.

Ainda, de 300 em 300 gerações, todos os *Pockets* dos agentes do nível intermediário da árvore (veja Figura 5) são substituídos por novos, gerados pela heurística NN. Devido as operações *UpdatePocket* e *PocketPropagation* para organização da árvore, informações consideradas relevantes (bons arcos no caso deste problema) das rotas de uma hierarquia são transmitidas para as hierarquias em um nível imediatamente superior e inferior. Desta forma, considera-se que em 300 gerações as informações relevantes da hierarquia intermediária já tenham sido transmitidas para o indivíduo raíz e para os indivíduos folhas da árvore e a troca dos *Pockets* da hieraquia intermediária por novas soluções introduziria maior diversidade na população.

3.6. RESULTADOS COMPUTACIONAIS CONSIDERANDO POPULAÇÃO ESTRUTURADA

O objetivo dos testes computacionais é analisar a performance do algoritmo, com ênfase principalmente no novo operador de busca local. Para isso, o algoritmo foi testado com 4 operadores de recombinação diferentes, descritos nas seções 3.3.1 a 3.3.4. Os testes são compostos pelas 27 instâncias de ATSP da TSPLIB (dimensões entre 17 e 443 cidades). O algoritmo memético foi executado 30 vezes para cada instância. Toda programação foi feita utilizando a linguagem de programação Java, com o compilador do *Java Software Development Kit* 1.2. O computador é um *Pentium* II, com 128 MB de RAM e 450 Mhz para a Tabela 2 e 400 Mhz para todos os demais testes computacionais descritos nesta dissertação. Como critérios de parada estabeleceram-se 1000 gerações ou 300 segundos (seg.) de tempo de CPU para instâncias até 500 cidades, e 1000 gerações ou 1000 seg. para problemas maiores. A execução é interrompida assim que o primeiro critério de parada (tempo ou n° de gerações) for satisfeito. Obviamente, como se tem conhecimento da solução ótima, o algoritmo interrompe a execução caso esta seja encontrada. Em geral, nos casos em que a solução ótima não é encontrada, se a instância é grande (com mais de 200 cidades) o primeiro critério de parada satisfeito é o tempo.

As médias dos resultados utilizando cada operador de recombinação são apresentadas na Tabela 1, onde *Otm/#exe* representa o número médio de vezes que a execução encontrou a solução ótima (*Otm*), antes do programa satisfazer qualquer critério de parada, considerando o número de execuções definido por *#exe*; *Qual (%)* representa o desvio percentual médio da solução ótima; *Ger* é a média do número de gerações e *Tempo (seg.)* denota o tempo total médio de CPU em segundos.

Tabela 1- Médias obtidas pelo MA utilizando RAI e 4 diferentes operadores de recombinação.

Operador de Recombinação	*Otm/#exe*	*Qual (%)*	*Ger*	*Tempo (seg.)*
SAX	25,5/30	0,071	222	8,08
DPX	24,2/30	0,033	276	22,91
MFNN	18,8/30	0,195	495	6,86
UNN	18,9/30	0,160	447	7,02

O desvio percentual médio, o qual é apresentado pela coluna *Qual (%)*, é calculado pela fórmula seguinte:

$$Qual\ (\%) = \frac{\sum_{i=1}^{\#exe} \frac{(\text{Sol. Final}_i - \text{Sol. Ótima})}{(\text{Sol. Ótima})} * 100}{\#exe}$$

Para uma melhor visualização, os critérios mais importantes para medir a performance do método, que são a qualidade da solução final e o tempo de CPU, são plotados na Figura 16. A melhor relação entre qualidade e tempo de CPU é encontrada pelo operador de recombinação SAX. Desta forma, nas demais comparações envolvendo o MA proposto e outros métodos da literatura, vamos utilizar a combinação SAX/RAI.

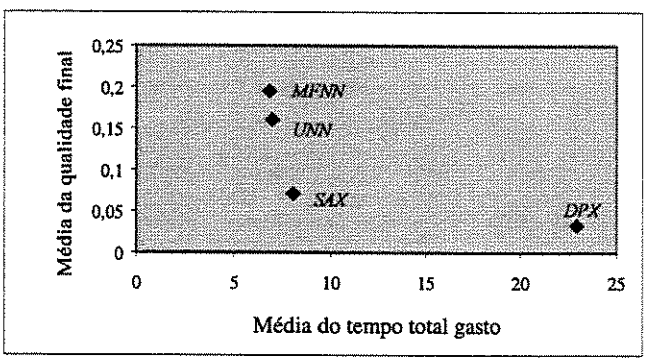


Figura 16 - Performance do MA em relação à qualidade e tempo de CPU em segundos.

Uma interessante observação pode ser feita através destes experimentos. RAI demonstrou boa performance, independente do operador de recombinação utilizado, mostrando que pode ser considerado um operador de busca local robusto para MAs aplicados ao ATSP.

A Tabela 2 apresenta os resultados obtidos pelo MA usando SAX/RAI. Os resultados representam as médias de 30 execuções para cada uma das 27 instâncias de ATSP disponíveis na TSPLIB. As três primeiras colunas se referem ao nome da instância, dimensão da mesma e seu custo de solução ótima (valores disponíveis na TSPLIB), respectivamente. *CPU BL* denota a percentagem do tempo de CPU gasto na fase de busca local e *IniPop (%)* apresenta o desvio percentual médio da população inicial (obtida pela heurística NN), calculado pela fórmula:

$$IniPop\ (\%) = \frac{\sum_{i=1}^{\#exe} \frac{(\text{Sol. Inicial Média}_i - \text{Sol. Ótima})}{(\text{Sol. Ótima})} * 100}{\#exe}$$

As colunas *Otm/#exe*, *Qual (%)*, *Ger* e *Tempo (seg.)* foram descritas para a Tabela 1. O tamanho da população é fixo em 13 indivíduos para todas as execuções.

Tabela 2 - Resultados computacionais do SAX/RAI aplicado às instâncias de ATSP da TSPLIB.

Instância	*#Cid.*	*Ótima*	*Otm/#exe*	*IniPop (%)*	*Qual (%)*	*Ger*	*CPU BL (%)*	*Tempo (seg.)*
br17	17	39	30/30	69	0,00	1,9	-	0,00
ftv33	34	1286	30/30	51	0,00	11,1	80	0,06
ftv35	36	1473	30/30	44	0,00	80,5	76	0,39
ftv38	39	1530	13/30	39	0,07	785,7	75	3,82
p43	43	5620	30/30	2	0,00	22,7	83	0,17
ftv44	45	1613	28/30	62	0,04	381,1	76	2,22
ftv47	48	1776	30/30	50	0,00	52,1	81	0,42
ry48p	48	14422	30/30	33	0,00	111,8	80	0,92
ft53	53	6905	30/30	50	0,00	92,6	80	0,80
ftv55	56	1608	30/30	65	0,00	21,2	84	0,24
ftv64	65	1839	30/30	47	0,00	27,7	81	0,35
ft70	70	38673	30/30	16	0,00	305,9	84	4,64
ftv70	71	1950	30/30	55	0,00	67,1	81	0,90
ftv90	91	1579	30/30	75	0,00	29,2	85	0,60
kro124p	100	36230	30/30	38	0,00	62,4	87	1,71
ftv100	101	1788	28/30	73	0,02	94,7	83	2,02
ftv110	111	1958	29/30	77	0,00	78,0	84	2,05
ftv120	121	2166	22/30	76	0,07	342,4	84	9,35
ftv130	131	2307	25/30	89	0,04	270,7	85	8,63
ftv140	141	2420	01/30	87	0,30	969,9	85	33,73
ftv150	151	2611	02/30	81	0,71	940,9	85	35,51
ftv160	161	2683	03/30	92	0,64	911,7	86	42,33
ftv170	171	2755	28/30	91	0,02	228,7	88	12,86
rbg323	323	1326	30/30	35	0,00	32,6	94	12,61
rbg358	358	1163	30/30	59	0,00	64,5	94	28,78
rbg403	403	2465	30/30	41	0,00	6,8	95	6,01
rbg443	443	2720	30/30	44	0,00	6,4	91	7,07
Médias			25,5/30	57,07	0,07	222,2	84,4	8,08

A Tabela 3 apresenta os resultados computacionais do MA com RAI e os demais operadores de recombinação testados, para as mesmas instâncias da Tabela 2.

Tabela 3 - Resultados computacionais para o MA com RAI usando os outros três operadores de recombinação testados.

	DPX						MFNN						UNN					
Instância	*Otm/ #exe*	*IniPop (%)*	*Qual (%)*	*Ger*	*CPU BL (%)*	*Tempo (seg.)*	*Otm/ #exe*	*IniPop (%)*	*Qual (%)*	*Ger*	*CPU BL (%)*	*Tempo (seg.)*	*Otm/ #exe*	*IniPop (%)*	*Qual (%)*	*Ger*	*CPU BL (%)*	*Tempo (seg.)*
br17	30/30	69	0,00	1,9	67	0,00	30/30	66	0,00	2,1	33	0,00	30/30	71	0,00	1,9	18	0,00
ftv33	30/30	51	0,00	13,7	80	0,09	30/30	51	0,00	18,5	57	0,05	30/30	50	0,00	48,3	87	0,16
ftv35	30/30	44	0,00	32,5	82	0,20	30/30	44	0,00	44,4	59	0,13	15/30	45	0,07	628,3	85	1,84
ftv38	19/30	39	0,05	622,0	83	3,09	26/30	39	0,02	360,1	62	1,03	06/30	39	0,10	884,5	84	2,65
p43	30/30	2	0,00	109,7	81	1,03	30/30	2	0,00	71,8	55	0,26	29/30	2	0,00	166,5	88	0,88
ftv44	29/30	62	0,02	296,9	84	2,36	29/30	62	0,02	212,4	66	0,75	06/30	62	0,59	952,0	86	3,75
ftv47	30/30	51	0,00	10,3	88	0,13	22/30	51	0,06	491,0	64	1,74	30/30	51	0,00	43,0	88	0,24
ry48p	30/30	33	0,00	54,8	86	0,60	23/30	33	0,06	446,6	66	2,12	24/30	33	0,03	481,4	90	2,57
ft53	30/30	50	0,00	32,5	89	0,42	19/30	50	0,01	676,1	64	2,68	30/30	50	0,00	81,6	89	0,47
ftv55	30/30	65	0,00	36,4	86	0,46	30/30	65	0,00	42,4	68	0,21	30/30	65	0,00	31,0	91	0,23
ftv64	30/30	47	0,00	45,3	87	0,66	30/30	47	0,00	211,0	68	0,98	30/30	47	0,00	36,0	89	0,28
ft70	06/30	16	0,04	880,2	84	19,06	00/30	16	0,09	1000	68	7,56	26/30	16	0,01	294,8	91	2,73
ftv70	30/30	57	0,00	105,6	87	1,66	15/30	56	0,11	745,6	71	3,83	30/30	56	0,00	140,3	90	1,13
ftv90	26/30	76	0,04	258,4	87	4,25	25/30	75	0,02	582,5	75	4,59	09/30	76	0,20	722,0	88	5,93
kro124p	30/30	38	0,00	67,8	90	2,29	30/30	38	0,00	289,8	74	2,53	30/30	39	0,00	159,3	93	2,59
ftv100	24/30	73	0,02	308,8	88	6,40	20/30	73	0,04	702,8	77	6,52	07/30	73	0,21	790,0	89	8,29
ftv110	19/30	77	0,07	485,1	89	12,89	00/30	77	0,80	1000	77	10,24	02/30	77	0,26	939,5	90	11,32
ftv120	15/30	76	0,11	649,8	90	24,12	00/30	76	1,24	1000	79	11,60	00/30	76	0,26	1000	90	14,01
ftv130	19/30	89	0,08	573,3	90	22,71	02/30	89	0,21	982,9	81	14,51	00/30	89	0,26	1000	91	16,36
ftv140	05/30	87	0,24	863,2	88	36,93	01/30	87	0,90	981,1	81	15,25	00/30	87	0,50	1000	91	17,47
ftv150	29/30	81	0,00	371,0	86	17,44	01/30	81	0,48	987,2	81	16,68	03/30	80	0,82	930,5	92	20,15
ftv160	18/30	92	0,06	739,6	86	43,00	00/30	91	0,24	1000	76	14,38	01/30	92	0,81	978,9	93	25,69
ftv170	18/30	91	0,09	648,4	87	46,19	00/30	91	0,94	1000	81	17,99	23/30	91	0,21	668,4	93	19,54
rbg323	29/30	36	0,00	72,8	90	83,59	26/30	35	0,01	371,6	79	32,13	30/30	35	0,00	33,1	96	7,94
rbg358	08/30	59	0,07	172,5	89	271,23	30/30	59	0,00	117,5	72	10,46	30/30	60	0,00	43,3	97	11,64
rbg403	30/30	40	0,00	4,7	93	8,18	30/30	41	0,00	11,4	77	3,27	30/30	40	0,00	7,8	96	4,79
rbg443	30/30	43	0,00	4,1	92	9,60	30/30	44	0,00	4,8	85	3,61	30/30	45	0,00	8,8	96	6,97
Médias	24,2/30	57,2	0.033	276,3	86,3	22,91	18,8/30	57,0	0,19	494,6	70,2	6,86	18,9/30	57,3	0,16	447,1	87,8	7,02

A Figura 17 particulariza a performance do MA, utilizando os quatro operadores de recombinação, aplicados às duas maiores instâncias de ATSP da TSPLIB. A solução ótima foi sempre encontrada em todas as execuções, independente do operador de recombinação utilizado. Os tempos computacionais médios ficaram entre 3 e 9 seg. Estes resultados reforçam ainda mais a observação da robustez do MA implementado, demonstrando a habilidade da busca RAI em encontrar soluções de qualidade, mesmo em instâncias grandes.

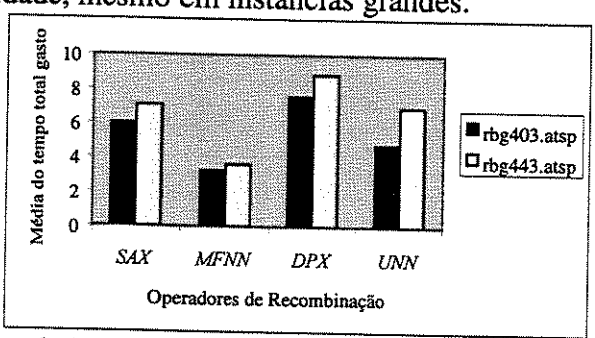


Figura 17 - Peformance do MA aplicado às 2 maiores instâncias de ATSP da TSPLIB.

A fim de comparar performance, é feita uma comparação entre os resultados obtidos pelo SAX/RAI e por outras metaheurísticas recentemente aplicadas ao ATSP. Foram selecionados 6 algoritmos metaheurísticos com os melhores resultados relatados na literatura (até outubro/1999). Estes resultados são oriundos dos MAs de Gorges-Schleuter (1997), Merz e Freisleben (1997), Nagata e Kobayashi (1997) e Walters (1998). As comparações também incluem uma otimização por colônia de formigas de Stützle e Dorigo (1999) e a técnica de *simulated jumping* (Amin, 1999), originada de idéias de *spin-glasses, simulated-annealing* e *self-organization.*

A Tabela 4 sumariza os resultados publicados por estes métodos, aplicados às instâncias de ATSP da TSPLIB. A coluna *Tam. Pop* denota o tamanho da população (as informações contidas nas outras colunas foram descritas em tabelas anteriores). Os resultados de cada método foram obtidos usando os computadores e linguagens de implementação indicados. Os autores testaram diferentes número de execuções: Gorges-Schleuter, Merz e Freisleben e Walters fizeram 20 execuções, Nagata e Kobayashi fizeram 30 execuções, enquanto que Stützle e Dorigo fizeram 25 execuções. Amin apresenta somente os melhores resultados selecionados dentre 5 execuções.

A penúltima linha da Tabela 4 apresenta a qualidade e tempo de CPU médios obtidos pelos 6 métodos, enquanto que na última linha aparecem as médias obtidas pelo MA SAX/RAI aplicado apenas ao conjunto de instâncias da coluna correspondente. Assim é possível comparar os resultados obtidos pelo MA SAX/RAI com as mesmas instâncias utilizadas pelos outros métodos.

Tabela 4 - Resultados computacionais encontrados por métodos de outros pesquisadores.

Autor	Gorges-Schleuter (1997)			Merz e Freisleben (1997)			Nagata e Kobayashi (1997)			Walters (1998)			Stützle e Dorigo (1999)		Amin (1999)	
Método	MA			MA			MA			MA			*Ant Colonies*		*Simul. Jumping*	
Computador	SUN *UltraSparc* 170MHz			DEC *Alpha* 233MHz			*Pentium* 200 MHz			*Pentium* II 300 MHz			*UltraSparc* II *167MHz*		*Dell computer* 300 MHz	
Linguagem	C			C++			C			C			C		C	
Instância	*Tam. Pop*	*Qual (%)*	*Tempo (seg.)*	*Tam. Pop*	*Qual (%)*	*Tempo (seg.)*	*Tam. Pop*	*Qual (%)*	*Tempo (seg.)*	*Tam. Pop*	*Qual (%)*	*Tempo (seg.)*	*Qual (%)*	*Tempo (seg.)*	*Qual (%)*	*Tempo (seg.)*
br17															0,00	1
ftv33															0,00	71
ftv35															0,00	107
ftv38															0,00	65
p43	20	0,00	2,1	40	0,00	11									0,00	10
ftv44															0,00	940
ftv47															0,00	1450
ry48p	20	0,00	3,5	40	0,20	72	100	0,00	3	20	0,00	0,4	0,00	2,3	0,00	78
ft53							100	0,56	5						0,00	185
ftv55															0,00	1067
ftv64															0,00	70
ft70	20	0,00	4,4	40	0,00	111	150	0,00	10	20	0,00	2,0	0,00	37,2	0,00	430
ftv70															0,00	190
ftv90																
ftv100																
kro124p	20	0,00	6,0	40	0,00	35	200	0,02	15	20	0,00	1,6	0,00	7,3	0,00	1223
ftv110																
ftv120																
ftv130																
ftv140																
ftv150																
ftv160																
ftv170	20	0,00	9,5	40	0,26	100	350	0,00	60	50	0,00	7,3	0,00	56,2	1,49	482
rbg323							300	0,00	129						0,00	11140
rbg358							350	0,00	172						0,09	4308
rbg403							400	0,00	256						0,00	544
rbg443							450	0,00	268						0,00	11280
Médias		**0,00**	**5,1**		**0,09**	**65,8**		**0,06**	**102**		**0,00**	**2,8**	**0,00**	**25,7**	**0,08**	**1771**
SAX/RA1		***0,00***	***4,1***		***0,00***	***4,1***		***0,00***	***8,4***		***0,00***	***5,0***	***0,00***	***5,0***	***0,00***	***4,8***

3.7 RESULTADOS COMPUTACIONAIS CONSIDERANDO POPULAÇÃO NÃO ESTRUTURADA

Para finalizar a fase de testes, o MA utilizando SAX/RAI é testado com uma população não estruturada. As operações *PocketPropagation* e *OrderChildren*, utilizadas para manter a árvore organizada como uma árvore ternária, são omitidas.

Tabela 5 - Resultados computacionais do SAX/RAI aplicado às instâncias de ATSP da TSPLIB, considerando população não estruturada.

Instância	*#Cid.*	*Ótima*	*Otm/#exe*	*IniPop (%)*	*Qual (%)*	*Ger*	*CPU BL (%)*	*Tempo (seg.)*
br17	17	39	30/30	64	0,00	21,7	41	0,14
ftv33	34	1286	30/30	51	0,00	176,9	55	1,75
ftv35	36	1473	13/30	49	0,09	709,8	56	7,46
ftv38	39	1530	02/30	44	0,16	988,2	54	10,73
p43	43	5620	20/10	4	0,01	613,9	65	9,99
ftv44	45	1613	13/30	74	0,47	828,2	57	11,18
ftv47	48	1776	28/30	56	0,01	322,9	62	5,32
ry48p	48	14422	28/30	33	0,00	413,2	65	7,31
ft53	53	6905	24/30	59	0,01	565,2	63	10,50
ftv55	56	1608	29/30	88	0,01	329,4	59	5,94
ftv64	65	1839	28/30	52	0,01	377,6	61	8,17
ft70	70	38673	17/30	17	0,03	677,6	64	16,64
ftv70	71	1950	26/30	133	0,03	407,8	64	10,05
ftv90	91	1579	28/30	85	0,01	307,3	65	10,21
kro124p	100	36230	30/30	78	0,00	385,1	67	16,13
ftv100	101	1788	30/30	39	0,00	345,2	66	13,75
ftv110	111	1958	27/30	163	0,02	434,4	67	21,93
ftv120	121	2166	21/30	84	0,05	607,3	65	36,99
ftv130	131	2307	24/30	94	0,04	557,7	64	38,05
ftv140	141	2420	01/30	93	0,36	970,6	69	67,61
ftv150	151	2611	10/30	87	0,58	821,9	70	61,27
ftv160	161	2683	09/30	98	0,60	814,4	73	69,93
ftv170	171	2755	15/30	97	0,09	722,3	75	70,66
rbg323	323	1326	08/30	50	0,07	354,5	92	273,51
rbg358	358	1163	05/30	65	0,15	299,6	92	274,61
rbg403	403	2465	15/30	42	0,03	173,0	94	215,03
rbg443	443	2720	13/30	49	0,03	137,0	94	224,37
Médias			19,41/30	68,44	0,11	494,9	67,4	55,52

Como o procedimento **selecioneParaRecombinação** é aplicado considerando uma árvore ternária, uma alteração nesse procedimento é necessária. A cada geração um novo indivíduo para cada rota *Current* é gerado, da mesma forma que foi descrita anteriormente. A mudança aparece na política de seleção dos pais e designação do filho a um agente. Considerando 13 indivíduos, a fase de recombinação gerou 3 novos indivíduos por uma recombinação entre duas rotas *Pockets* aleatoriamente selecionadas e 10 novos indivíduos por uma recombinação entre uma rota *Pocket* e uma *Current* aleatoriamente selecionadas.

3.8. ANÁLISE DOS RESULTADOS

As tabelas de resultados apresentadas neste capítulo comprovam a robustez do MA proposto, pois apresentam bons resultados para todas as instâncias de ATSP da TSPLIB. Esta conclusão é reforçada quando os resultados obtidos pelo MA proposto são comparadas com os resultados heurísticos apresentados na Tabela 4. Os métodos de Gorges-Schleuter e Walters são comparáveis em termos de qualidade de solução e tempo computacional. Se a performance dos métodos forem comparados, as instâncias testadas pelos demais pesquisadores são consideradas fáceis pelo MA proposto, enquanto que as instâncias `ftv120`, `ftv130`, `ftv140`, `ftv150` e `ftv160` são difíceis. Observa-se que os resultados para estas últimas não foram publicados por nenhum dos 6 pesquisadores mencionados na Tabela 4.

Outro critério desejável em um implementação é manter os parâmetros fixos, ou dependentes de alguma função, para todas as instâncias. Por exemplo, os testes apresentados por Nagata e Kobayashi e Walters, o tamanho da população muda conforme a instância. Nos experimentos com SAX/RAI, utilizam-se exatamente os mesmos parâmetros para todas as instâncias. O MA SAX/RAI encontrou no mínimo uma solução ótima para todas as instâncias. Considerando as 810 execuções dos experimentos computacionais com SAX/RAI, o MA encontrou a solução ótima em 85% das execuções e o desvio percentual médio da solução final foi é de 0,07%. Caso as 5 instâncias difíceis de `ftv` anteriormente mencionadas fossem excluídas dos experimentos, o desvio percentual médio da solução final ficaria em 0,007% e a percentagem de soluções ótimas encontradas seria de 96% e o tempo computacional médio cairia para 4,03

seg.

Pela Tabela 3 é possível verificar que o MA obteve bons resultados também com os demais operadores de recombinação testados. Embora o tempo encontrado pelo DPX tenha sido bem maior, o aumento considerável da média se deve ao elevado tempo gasto com a instância `rbg358`. Ainda, para este operador, mesmo que a razão *otm/#exe* seja menor que a razão encontrada pelo SAX, para todas as instâncias no mínimo 5 soluções ótimas foram encontradas, enquanto que, para as instâncias `ftv140`, `ftv150` e `ftv160` o algoritmo com SAX encontrou apenas 1, 2 e 3 vezes a solução ótima, respectivamente.

Considerando as 2.430 execuções dos experimentos computacionais envolvendo os demais operadores de recombinação testados, a solução ótima é encontrada em 69% das execuções (DPX: 81%, UNN e MFNN: 63%) e o desvio percentual médio da qualidade final é de 0,13% (DPX: 0,033%, UNN: 0,16% e MFNN: 0,19%).

Se a média dos resultados encontrados para os 4 operadores de recombinação testados (Tabelas 2 e 3) fossem avaliados, a solução ótima teria sido encontrada em 73% das execuções, obtendo uma qualidade de solução final de 0,11%, em um tempo computacional médio de 11,22 seg. Estes resultados são bons se comparados com os apresentados na Tabela 4, pois tratam-se das médias para todas as instâncias da TSPLIB e não para um conjunto particular em que o algoritmo tenha obtido bons resultados.

Outra observação interessante se refere às médias da qualidade de solução inicial e tempo gasto com busca local obtidos e apresentados nas Tabelas 2 e 3. A diferença entre a maior e a menor média de qualidade de solução inicial foi de 0,3% (UNN: 57,3%, DPX: 57,2%, SAX: 57,07% e MFNN: 57,0%). Já o tempo gasto com busca local apresentou algumas diferenças. Para os operadores UNN, DPX e SAX, a percentagem dos tempos gastos com busca local são muito parecidos (87,8%, 86,3% e 84,4%, respectivamente), enquanto que para o MFNN a percentagem média foi de 70,2%.

A Tabela 5 mostra que os resultados obtidos com população estruturada são melhores que os obtidos sem estruturação, mas a boa performance do algoritmo se mantém. Uma observação interessante é que, para todas as instâncias, embora o tempo computacional tenha aumentado, para as instâncias de `ftv` consideradas difíceis, a média da qualidade de solução melhorou. A qualidade de solução média para estas instâncias, considerando a população estruturada, é de

0,352. Considerando a população não estruturada, a média passou para 0,326%. Esses números estão relacionados com o número de vezes que a solução ótima é encontrada: na população não estruturada, para estas instâncias consideradas difíceis, o número médio de soluções ótimas encontradas é de 13 em 30 execuções, enquanto que para população não estruturada, para o mesmo número de execuções, esta média caiu para 10,6. Embora a qualidade de solução final tenha melhorado, o tempo médio gasto, para estas instâncias, aumentou de 25,91 para 54,77 seg. Mas este aumento é pequeno se comparado com o aumento de tempo que as outras instâncias tiveram.

Quando as médias dos resultados obtidos pelas populações estruturada e não-estruturada são comparados, se torna fácil verificar o melhor desempenho do MA utilizando população estruturada: a qualidade de solução final considerando população estruturada é de 0,07% enquanto que para não estruturada é de 0,11%; o número de soluções ótimas encontradas diminuiu de 25,5 para 19,4; o número médio de gerações aumentou de 222,2 para 494,9 e a média do tempo gasto aumentou de 8,08 para 55,52. Esse aumento de tempo se deve, em grande parte, pelo aumento dispendido com as 4 maiores instâncias (de 323, 358, 403 e 443 cidades). Considerando os resultados médios sem estas instâncias, a média foi de 7,12 seg. para população estruturada e 22,24 seg. para não estruturada. Por outro lado, para a população estruturada, a média de tempo gasto com estas 4 instâncias foi de 13,62 seg., enquanto que, para a população não estruturada, esta média cresceu substancialmente para 246,88 seg. Assim mesmo, para as duas maiores instâncias, os tempos encontrados pela população não estruturada (215,03 e 224,27 seg.) são menores que os melhores resultados heurísticos encontrados na literatura até outubro/1999 (256 e 268 seg.), encontrados por Nagata e Kobayashi (1997).

É importante observar que nos testes computacionais realizados pelos diversos pesquisadores foram utilizados computadores e compiladores/linguagens de implementação diferentes. Considera-se que a performance dos computadores é proporcional ao *clock*, enquanto que o compilador Java, utilizado para os resultados computacionais que constam nesta dissertação, certamente é mais lento que os compiladores GNU C/C++. Até mesmo os defensores de Java admitem que a segurança provida pela linguagem, além das facilidades oriundas da disponibilidade de um *garbage collector*, podem influenciar na performance da mesma (veja em *http://www.stl.nps.navy.mil/lists/dis-java-vrml/0213.html*).

3.9 RESUMO E CONCLUSÕES

Neste capítulo apresentou-se a proposta de um novo algoritmo de busca local denominado *Recursive Arc Insertion* - RAI. No caso desta dissertação, o algoritmo de busca local é utilizado como operador num algoritmo memético (MA) aplicado ao problema do caixeiro viajante assimétrico (ATSP). A implementação do MA introduz procedimentos novos para melhorar a performance do algoritmo, tal como a população organizada hierarquicamente em uma árvore ternária completa de 13 agentes, agrupados em 4 subpopulações. Cada subpopulação é formada por um líder e três subordinados. Cada agente é composto por duas rotas factíveis e seus respectivos custos: *Current*, rota onde as operações de mutação, recombinação e busca local são aplicadas, e *Pocket* que funciona como uma memória que armazena uma rota com informações de gerações anteriores. A fim de manter uma organização lógica da árvore, três operações são aplicadas a esta, a cada geração, de forma que a rota *Pocket* sempre seja melhor que a *Current*, o *Pocket* do líder sempre seja melhor que o dos seus subordinados e os subordinados encontram-se ordenados de forma crescente de custo de rota *Pocket*. Com esta organização, a melhor rota da população encontra-se, ao final de cada geração, na rota *Pocket* do agente raiz da árvore. Quatro operadores de recombinação foram testados com a nova busca local, e os resultados das 30 execuções de cada instância de ATSP da TSPLIB com cada operador foram apresentados. Dentre os operadores, os melhores resultados foram obtidos pelo *Strategic Arc Crossover* - SAX, considerando tempo de CPU e qualidade da solução final obtida. O MA com SAX/RAI encontrou, em média, a solução ótima em 85% das execuções, num tempo computacional de 8,08 seg. e com uma qualidade de solução igual a 0,07%.

A comparação com resultados de 6 outros algoritmos, todos aplicados às mesmas instâncias da TSPLIB, demonstrou que o MA aqui proposto pode ser considerado robusto e de boa qualidade. Caso os parâmetros fossem adaptados instância a instância, certamente os resultados obtidos pelo MA seriam melhores.

Ainda, os resultados da execução do MA utilizando SAX/RAI sem população estruturada, apresentou resultados muito bons, ainda que não teve atenção especial (apenas retirou-se *PocketPropagation* e *OrderChildren* do código e o procedimento **selecioneParaRecombinação** foi alterado). Caso o objetivo fosse trabalhar com uma população não estruturada, certamente

outras adaptações teriam de ser feitas ao algoritmo (como adaptação de parâmetros) de tal forma que os resultados tenderiam a melhorar.

Ainda o fato do MA sem população estruturada ter qualidade de solução melhor para os problemas de `ftv` considerados difíceis, abre uma questão: talvez pequenas modificações no algoritmo pudessem alterar positivamente a performance do mesmo.

Os resultados encontrados pelo algoritmo quando aplicado a instâncias simétricas não foram apresentados, mas são comparáveis aos resultados encontrados pelo algoritmo quando aplicado às instâncias assimetrizadas da Tabela 6 a ser apresentada no próximo capítulo.

Os resultados do algoritmo quando aplicado a instâncias assimétricas são melhores se comparados com resultados obtidos pelo algoritmo quando aplicado à instâncias simétricas de mesma dimensão. Não chegou-se a uma conclusão sobre o porque isso acontece, mas algumas observações podem ser feitas quando são analisados algoritmos de outros pesquisadores. Existem algoritmos que se adaptam muito bem com instâncias assimétricas e não tão bem para o caso simétrico (caso do MA SAX/RAI). Entretanto o contrário também é válido. Por exemplo, dos pesquisadores mencionados na Tabela 6 que testaram seus algoritmos com instâncias simétricas (com exceção de Amin (1999)), os melhores resultados simétricos são obtidos por Merz e Freisleben (1997), que também apresentam os piores resultados para o caso assimétrico. Portanto, a possibilidade de haver características relacionadas à simetridade da instância que influenciam a performance do método é válida.

Capítulo

4

GERADORES PORTÁVEIS DE INSTÂNCIAS

Neste capítulo serão descritos dois geradores portáveis de instâncias implementados: um de instâncias para o problema de ciclos hamiltonianos (*Hamiltonian Cycle Problem* - HCP) e outro de instâncias para o ATSP. Ambos os geradores lêem e gravam arquivos no formato TSPLIB, geram uma instância com solução ótima conhecida, possuem interface gráfica e estão disponíveis na *Internet*. A necessidade da utililização desse geradores surgiu devido a TSPLIB dispor de apenas 27 instâncias de ATSP e 9 de HCP, sendo que a maior de ATSP tem 443 cidades e a menor de HCP tem 1000 vértices. Como foi desenvolvida uma redução de UHCP (*Undirect Hamiltonian Cycle Problem*) para o STSP, e um MA para resolver instâncias de ATSP, tornou-se necessário testar instâncias maiores de ATSP e menores de HCP.

As instâncias de HCP geradas podem ser consideradas direcionadas ou não. Isso acontece porque, inicialmente, uma rota direcionada é criada aleatoriamente. Logo, cada vértice i recebe um número x de arestas aleatoriamente escolhidas, adjacentes ao vértice i, tal que $minEdges \leq x \leq maxEdges$. As variáveis *minEdges* e *maxEdges* são informadas pelo usuário e contêm, respectivamente, o número mínimo e máximo de arestas adjacentes a cada vértice i. Ao se fazer a leitura desta instância, pode-se considerar o caso direcionado, assim como o caso não direcionado, pois a rota gerada é uma solução para ambos os casos.

Já as instâncias de ATSP são geradas a partir de instâncias de TSP com solução ótima conhecida. Uma instância de TSP é lida e, aos valores da matriz triangular inferior, que não fazem parte da rota ótima, é adicionado um valor aleatoriamente selecionado no intervalo [0,

$\alpha_{max}]$, onde α_{max} representa uma percentagem (pré-estabelecida pelo usuário) da média das distâncias da triangular superior da matriz de distâncias. Novos testes computacionais são apresentados utilizando o MA descrito no capítulo 3, aplicado às instâncias de ATSP geradas.

4.1. GERADOR DE INSTÂNCIAS ASSIMÉTRICAS DE TSP

Geradores de instâncias ótimas de ATSP são mencionados por alguns pesquisadores, tais como Glover et al. (1999), Fischetti e Toth (1997) e Miller e Pekny (1991). No entanto, tais geradores não obtêm a solução ótima, e sim uma solução que corresponde a um limitante inferior. No caso dos três artigos acima citados, todos utilizam um algoritmo *Branch & Bound* com limitante inferior calculado através do problema de designação (AP).

Embora os resultados computacionais obtidos, utilizando instâncias desses geradores, são de grande valia para análise, não é simples a sua reutilização por outros pesquisadores. Essa reutilização seria importante para fins de comparação de resultados.

A reutilização de instâncias grandes de ATSP não é usual devido ao problema com o transporte dessas, visto o espaço considerável que requerem. Diferente de uma instância de TSP, que é um vetor de coordenadas, uma instância de ATSP é uma matriz n x n cheia. Para se ter idéia, o arquivo da instância assimétrica `pr2392.atsp`, gerada a partir do arquivo da instância simétrica `pr2392.tsp` da TSPLIB, ocupa um espaço de 30 MB. Cabe lembrar que Miller e Pekny (1991) geraram instâncias de até 500.000 cidades.

Outra possibilidade seria utilizar um gerador e não as instâncias. Mas a reutilização de geradores não se mostra apropriada, pois não há garantia que a função de geração de números aleatórios utilizada tenha as propriedades de repetitividade e portabilidade desejáveis a um gerador. Por essas propriedades, sempre o mesmo comportamento estatístico é produzido e apresenta, para a mesma semente, a mesma seqüência de bits, independente da arquitetura da máquina onde está sendo executado. Se o gerador não tiver uma função de geração de números aleatórios com estas propriedades, não há garantia de obter duas vezes a mesma instância.

Com a finalidade de testar instâncias maiores de ATSP e de dispor de um gerador portável, implementou-se um gerador de instâncias ATSP a partir de instâncias TSP. Este gerador é portável e utiliza uma função de números aleatórios (*Random( )* do pacote *java.util*) com a propriedade da repetitividade, além da portabilidade. Como semente para o gerador foi escolhido o número 5. Não há um motivo especial para a escolha deste número; foi uma escolha aleatoria visto que uma semente fixa deveria ser indicada. Todas essas características tornaram viável sua reutilização. Por ser portável, devido a implementação em Java, este gerador pode ser executado em máquinas com diferentes arquiteturas e sistemas operacionais. Por ser repetitivo, uma mesma instância pode ser gerada quantas vezes for requisitada, sempre que os parâmetros de entrada forem os mesmos.

São requeridos dois dados de entrada: um arquivo contendo um instância de TSP e um arquivo contendo a (ou uma) rota ótima para esta instância. Devido a uma mesma instância de TSP poder possuir mais de uma rota ótima, duas instâncias de ATSP geradas são iguais se, além dos parâmetros serem os mesmos, também a rota ótima de TSP for a mesma.

O gerador foi implementado baseado nas instâncias da classe B de Fischetti e Toth (1997). Tais instâncias são geradas da seguinte forma: $d_{ij} = \sigma_{ij} + \alpha_{ij}$, com $\sigma_{ij} = \sigma_{ji}$ aleatoriamente gerado entre [1, 1000] e α_{ij} aleatoriamente gerado entre [0, 20].

No caso da implementação aqui relatada, a matriz de distâncias do ATSP é obtida a partir da matriz de distâncias da instância de STSP. A matriz triangular inferior é calculada da seguinte forma: para cada d_{ij} (com $i > j$), sendo que d_{ij} representa a distância da cidade *i* à cidade *j*, é adicionado um valor α_{ij} aleatoriamente gerado entre [0, α_{max}]. α_{max} é definido como o valor inteiro da porcentagem ρ da média das distâncias $D_{média}$ da matriz triangular superior, ou seja $\alpha_{max} = int\ (D_{média} * \rho/100)$.

Para que a rota ótima da instância simétrica fornecida seja a rota ótima para a instância assimétrica, para todo arco *(i, j)* presente na solução ótima, a distância d_{ij} da matriz triangular inferior é igualada a d_{ji}.

Os dados de entrada que o gerador necessita são: um arquivo de uma instância simétrica de TSP no formato *nome_da_instância.tsp* e um arquivo contendo uma rota ótima desta instância no formato *nome_da_instância.opt.tour*.

A Figura 18 mostra o arquivo da instância `ulysses16.atsp` gerado a partir do arquivo da instância `ulysses16.tsp` da TSPLIB com percentagem ρ=5%. As distâncias correspondentes aos arcos/arestas da solução ótima estão sombreadas na matriz.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	509	501	312	1019	736	656	60	1039	726	2314	479	448	479	619	150
2	538	0	126	474	1526	1226	1133	532	1449	1122	2789	958	941	978	1127	542
3	504	126	0	541	1516	1184	1084	536	1371	1045	2728	913	904	946	1115	499
4	330	492	569	0	1157	980	919	271	1333	1029	2553	751	704	720	783	455
5	1030	1556	1524	1194	0	478	583	996	858	855	1504	677	651	600	401	1033
6	772	1235	1219	997	493	0	115	740	470	379	1581	271	289	261	308	687
7	696	1168	1095	929	607	115	0	667	455	288	1661	177	216	207	343	592
8	60	533	563	289	1010	752	672	0	1066	759	2320	493	454	479	598	206
9	1053	1477	1379	1334	865	479	456	1093	0	328	1387	591	650	656	776	933
10	749	1155	1070	1055	866	407	328	778	331	0	1697	333	400	427	622	610
11	2348	2811	2748	2587	1511	1582	1668	2356	1387	1735	0	1838	1868	1841	1789	2248
12	502	958	936	751	708	283	177	528	615	344	1838	0	68	105	336	417
13	461	972	943	706	654	322	256	485	657	401	1870	68	0	52	287	406
14	511	984	978	751	633	275	211	489	661	464	1862	122	52	0	237	449
15	639	1150	1154	818	401	330	382	611	787	660	1808	344	325	266	0	636
16	157	561	499	486	1055	696	619	231	948	636	2251	441	436	476	652	0

Rota ótima:1→14→13→12→7→6→15→5→11→9→10→16→3→2→4→8→1

Custo da rota ótima = 479 + 52 + 68 + 177 + 115 + 308 + 401 + 1504 + 1387 + 328 + 610 + 499 + 126 + 474 + 271 + 60 = 6859

Figura 18 - Matriz assimetrizada, rota ótima e custo da rota ótima obtidos a partir do arquivo da instância simétrica `ulysses16.tsp` com ρ = 5%. O arquivo da instância assimétrica `ulysses16.atsp` gerada com ρ = 5% é sempre o mesmo, independente da máquina ou sistema operacional utilizados.

Como $D_{média}$ = 814,27, então o valor de α_{max} obtido é: $\alpha_{max} = int\ (814{,}27 * 5/100) = int(40{,}7135) = 40$.

Para uma instância assimetrizada qualquer, a diagonal superior da matriz assimétrica é exatamente igual à diagonal superior da matriz simétrica. Já a diagonal inferior possui suas distâncias acrescidas, com exceção das distâncias dos arcos pertencentes à rota ótima. Portanto, na matriz assimétrica da Figura 18, as distâncias sombreadas pertencem à solução ótima, e por isso nenhuma é alterada.

As instâncias são lidas (TSP) e gravadas (ATSP) conforme o formato da TSPLIB. Segundo este formato, uma instância possui um cabeçalho com informações sobre a mesma, seguido do dados necessários para preencher a matriz de distâncias.

4.1.1. Formatos dos arquivos de TSP lidos e de ATSP gerados

O gerador pode ler instâncias armazenadas em arquivos de TSP conforme os formatos da TSPLIB. Uma instância de TSP da TSPLIB possui, além dos dados para matriz de distâncias, um cabeçalho composto por campos indispensáveis e outros opcionais. Abaixo estão listados todos os campos possíveis (11 no total) e a informação que cada um fornece.

Campos presentes em todas as instâncias (1°, 2° , 3° e 4°):

1°) **NAME**: *<nome do arquivo>*

2°) **COMMENT**: *<comentários sobre a instância, em geral, sobre seu(s) criador(es) >*

3°) **TYPE**: *<problema em questão>*

4°) **DIMENSION**: *<n° de cidades do arquivo>*

Um ou os dois campos abaixo (5° e 6°) sempre aparece(m):

5°) **EDGE_WEIGHT_TYPE**: *<especificação de como os custos dos arcos podem ser obtidos>*

A especificação consiste em um dos itens abaixo:

a) EXPLICIT: matriz de distâncias armazenada explicitamente;

b) EUC_2D: usar cálculo de distância *euclidiana* em 2 dimensões;

c) EUC_3D: usar cálculo de distância *euclidiana* em 3 dimensões;

d) MAX_2D: usar cálculo de distância de *máximo* em 2 dimensões;

e) MAX_3D: usar cálculo de distância de *máximo* em 3 dimensões;

f) MAN_2D: usar cálculo de distância *manhattan* em 2 dimensões;

g) MAN_3D: usar cálculo de distância *manhattan* em 3 dimensões;

h) CEIL_2D: usar cálculo de distância *euclidiana* em 2 dimensões. Os valores são arredondados pela função de maior inteiro;

i) GEO: usar cálculo de distância *geográfica*;

j) ATT: usar cálculo de distância *pseudo-euclidiana*;

k) XRAY1: usar cálculo de distância para problemas cristalográficos com motores de mesma velocidade;

l) XRAY2: usar cálculo de função de distância para problemas cristalográficos com motores com velocidades diferentes;

m) SPECIAL: cálculo por outra função de distância qualquer.

As funções de cálculo para as distâncias de *b)* a *j)* estão todas implementadas no gerador, e encontram-se descritas em Reinelt (1991). As funções XRAY1, XRAY2 não foram implementadas, assim como nenhuma função SPECIAL.

6°) **EDGE_WEIGHT_FORMAT**: *<especifica o formato dos dados, se estes estiverem armazenados explicitamente>*

A especificação consiste em um dos itens abaixo:

a) FUNCTION: distâncias são obtidas por uma das funções descritas nos itens de *b)* a *m)* do 5° campo;

b) FULL_MATRIX: distâncias estão armazenadas como uma matriz cheia;

c) LOWER_DIAG_ROW: distâncias estão armazenadas como uma matriz triangular inferior armazenada por linhas, incluindo diagonal de zeros;

d) LOWER_DIAG_COL: distâncias estão armazenadas como uma matriz triangular inferior armazenada por colunas, incluindo diagonal de zeros;

e) LOWER_ROW: distâncias estão armazenadas como uma matriz triangular inferior armazenada por linhas, sem diagonal de zeros;

f) LOWER_COL: distâncias estão armazenadas como uma matriz triangular inferior armazenada por colunas, sem diagonal de zeros;

g) UPPER_DIAG_ROW: distâncias estão armazenadas como uma matriz triangular superior armazenada por linhas, incluindo diagonal de zeros;

h) UPPER_DIAG_COL: distâncias estão armazenadas como uma matriz triangular superior armazenada por colunas, incluindo diagonal de zeros.

i) UPPER_ROW: distâncias estão armazenadas como uma matriz triangular superior armazenada por linhas, sem diagonal de zeros;

j) UPPER_COL: distâncias estão armazenadas como uma matriz triangular superior armazenada por colunas, sem diagonal de zeros.

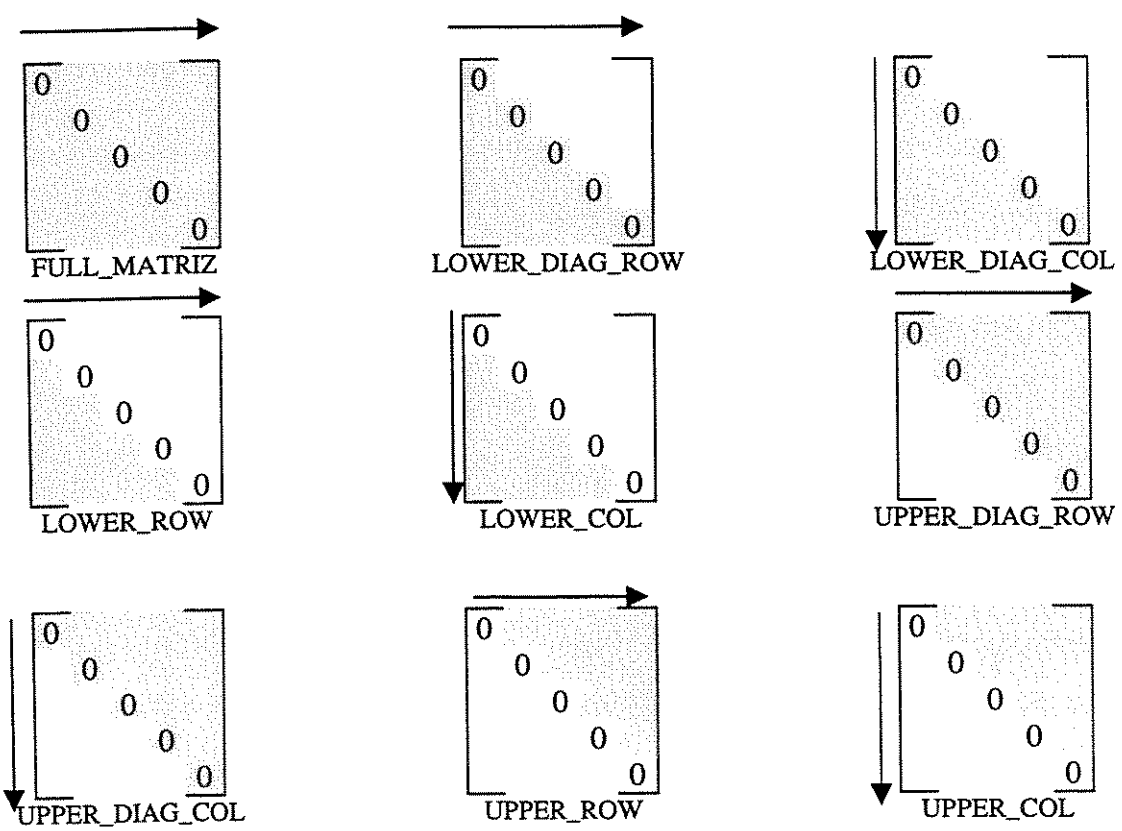


Figura 19 - Armazenamento explícito da matriz de distâncias de uma instância de TSP simétrico. As células cinzas representam a matriz armazenada em arquivo e a seta → indica leitura por linha e ↓ indica leitura por coluna.

Os dois campos abaixo (7° e 8°) são opcionais:

7°) **NODE_COORD_TYPE**: *<informa como as coordenadas estão associadas às cidades>*

a) TWO_COORDS: as cidades estão especificadas por coordenadas em 2 dimensões;

b) THREED_COORDS: as cidades estão especificadas por coordenadas em 3 dimensões;

c) NO_COORDS: as cidades não têm coordenadas associadas.

8°) **DISPLAY_DATA_TYPE**: *<especifica como obter visualização gráfica das cidades>*

a) COORD_DISPLAY: visualização gráfica é gerada a partir das coordenadas;

b) TWOD_DISPLAY: coordenadas explícitas são fornecidas em 2 dimensões sem índice de linha;

c) NO_DISPLAY: não é possível obter visualização gráfica.

Um, e somente um, dos três campos abaixo (9°, 10° e 11°) deve aparecer:

9°) **NODE_COORD_SECTION**: especifica que as coordenadas das cidades estão armazenadas por linha como *<inteiro> <real> <real>*;

10°) **DISPLAY_DATA_SECTION**: especifica que as coordenadas das cidades estão armazenadas segundo DISPLAY_DATA_TYPE. Se DISPLAY_DATA_TYPE é TWO_DISPLAY, cada linha é composta por *<inteiro> <real> <real>*;

11°) **EDGE_WEIGHT_SECTION**: especifica que a matriz de distâncias é do tipo EDGE_WEIGHT_FORMAT com valores inteiros e sem o índice de linha.

Após o cabeçalho, os dados são fornecidos na forma de coordenadas ou matriz, e sua finalização é indicada pelo finalizador de arquivo EOF.

Verificando os arquivos de instâncias da TSPLIB, observa-se que nem sempre a ordem dos campos e a formatação são mantidas. Por exemplo, na Figura 20 o cabeçalho do arquivo da instância *ulysses16.tsp* tem TYPE como 2° campo, possui a extensão "*.tsp*" no seu nome e não possui espaço em branco antes dos dois pontos ":" de alguns campos. Observe a diferença com o cabeçalho da instância *att48*.

NAME: ulysses16.tsp TYPE: TSP COMMENT: Odyssey of Ulysses (Groetschel/Padberg) DIMENSION: 16 EDGE_WEIGHT_TYPE: GEO DISPLAY_DATA_TYPE: COORD_DISPLAY NODE_COORD_SECTION	NAME : att48 COMMENT : 48 capitals of the US (Padberg/Rinaldi) TYPE : TSP DIMENSION : 48 EDGE_WEIGHT_TYPE : ATT NODE_COORD_SECTION

Figura 20 - Exemplos de dois cabeçalhos de instâncias de TSP simétrico da TSPLIB.

A formatação e a ordem dos campos diferentes foram tratadas, de tal forma que não há problemas para o gerador. Além disso, os campos 7° a 11° não são analisados, visto que, com a leitura dos campos anteriores, é possível deduzir a informação contida nestes campos.

O cabeçalho de uma instância ATSP criada pelo gerador (veja Figura 21) é sempre a mesma, e possui sempre 7 campos: 1° a 6° descritos anteriormente, além do 11°. Os três últimos campos são iguais em todas as instâncias. O campo COMMENT é composto por *file generated from the file*, seguido do comentário presente na instância de TSP e, ao final, seguido do valor da percentagem selecionada pelo usuário.

NAME : ulysses16.atsp TYPE : ATSP COMMENT : file generated from the file Odyssey of Ulysses (Groetschel/Padberg) with PERCENTAGE CHOSEN = 5.0 DIMENSION : 16 EDGE_WEIGHT_TYPE: EXPLICIT EDGE_WEIGHT_FORMAT: FULL_MATRIX EDGE_WEIGHT_SECTION

Figura 21 - Cabeçalho do arquivo da instância assimetrizada `ulysses16.atsp`.

Nesta implementação, a diagonal que separa a matriz diagonal inferior da superior é sempre preenchida por zeros (0's).

O valor que segue PERCENTAGE_CHOSEN no 3° campo informa a percentagem ρ selecionada pelo usuário. Numa primeira implementação, esta informação havia sido adicionada como um novo campo no cabeçalho. Optou-se por colocar esta informação no campo COMMENT pois em geral não há necessidade do seu conhecimento pelo algoritmo que fizer leitura do aquivo, além do que os algoritmos atuais deveriam ser adaptados para fazer leitura de mais este campo.

A descrição do gerador usando *JavaDoc*, gerador de documentação para programas Java (faz parte do *kit* do Java SDK), bem como seu código, estão disponíveis no endereço *internet* *http://www.densis.fee.unicamp.br/~buriol/ATSPGenerator.html.*

4.1.2. Resultados computacionais

Nesta seção são apresentados os resultados computacionais que avaliam a performance do algoritmo memético RAI/SAX sobre um novo conjunto de instâncias, gerado a partir do gerador implementado. Para todas as instâncias simétricas que possuiam uma rota ótima disponível na TSPLIB, as instâncias assimétricas correspondentes foram geradas.

A Tabela 6 apresenta os resultados da execução do algoritmo memético descrito no capítulo 3. Duas percentagens são utilizadas: $\rho = 5\%$ e $\rho = 50\%$. O valor numérico que aparece no nome da instância corresponde à dimensão do problema.

Os critérios de parada são testados ao final de cada geração. Devido a isto, caso o tempo for o critério satisfeito, normalmente alguns segundos a mais são considerados.

Tabela 6 - Resultados computacionais do MA com SAX/RAI aplicado às instâncias de ATSP geradas a partir de instâncias simétricas de TSP da TSPLIB.

	α_{max}: 5%					α_{max}: 50%				
Instância	*Otm/#exe*	*IniPop (%)*	*Qual (%)*	*Ger*	*Tempo (seg.)*	*Otm/#exe*	*IniPop (%)*	*Qual (%)*	*Ger*	*Tempo (seg.)*
ulysses16	30/30	39	0,00	2,2	0,01	30/30	30	0,00	2,9	0,01
ulysses22	30/30	37	0,00	2,1	0,01	30/30	37	0,00	2,6	0,01
gr24	30/30	40	0,00	3,4	0,02	30/30	48	0,00	2,5	0,01
fri26	30/30	35	0,00	3,4	0,03	30/30	79	0,00	2,9	0,02
bayg29	30/30	40	0,00	3,7	0,03	30/30	46	0,00	3,4	0,02
bays29	30/30	36	0,00	4,7	0,04	30/30	68	0,00	5,4	0,04
att48	30/30	52	0,00	24,8	0,34	30/30	43	0,00	88,2	0,77
gr48	30/30	46	0,00	14,8	0,19	30/30	61	0,00	15,3	0,17
eil51	30/30	33	0,00	59,4	0,60	29/30	44	0,02	76,5	0,67
berlin52	30/30	46	0,00	18,1	0,27	30/30	51	0,00	9,1	0,15
st70	30/30	42	0,00	19,9	0,39	30/30	64	0,00	61,3	0,79
eil76	24/30	42	0,10	447,2	6,68	26/30	49	0,23	227,8	3,27
pr76	30/30	73	0,00	30,4	0,65	21/30	73	0,30	469,6	6,04
gr96	27/30	45	0,06	133,9	2,82	20/30	64	0,83	548,4	10,28
kroa100	29/30	54	0,05	68,5	1,70	29/30	76	0,05	110,0	2,79
kroc100	30/30	56	0,00	19,1	0,73	23/30	84	0,38	281,4	5,34
krod100	20/30	50	0,32	364,7	8,20	29/30	80	0,04	149,0	3,17
rd100	30/30	53	0,00	35,7	1,12	23/30	74	0,21	341,6	6,44
eil101	26/30	34	0,04	340,9	8,82	29/30	41	0,05	88,6	2,22
lin105	30/30	59	0,00	30,0	1,22	30/30	93	0,00	28,6	1,22
gr120	29/30	43	0,01	169,3	6,25	26/30	63	0,18	251,6	7,20
ch130	27/30	55	0,18	217,7	9,02	23/30	67	0,42	409,0	12,08
ch150	18/30	51	0,38	600,5	23,82	16/30	73	0,47	565,8	18,33
brg180	00/30	1520	9,38	1000	125,57	00/30	1758	10,33	1000	143,87
gr202	05/30	38	0,41	892,8	61,35	00/30	57	1,28	1000	72,31
tsp225	02/30	50	0,65	938,9	80,80	01/30	73	0,02	967,6	64,68
a280	06/30	49	0,53	834,9	89,16	03/30	78	0,68	963,1	101,08
pcb442	00/30	56	1,61	1000	605,46	00/30	81	2,68	1000	637,70
pa561	00/30	31	2,44	995,9	856,71	00/30	39	3,40	1000	729,76
gr666	00/30	27	2,49	674,5	998,24	00/30	51	5,82	522,1	1000,60
pr1002	00/30	34	7,56	188,1	1001,80	00/30	49	10,20	177,5	1002,60
pr2392	00/30	34	10,77	26,6	1016,80	00/30	45	13,21	21,0	1024,70
Médias	20,7/30	90,6	1,16	286,4	153,40	19,6/30	113,7	1,59	324,7	151,82

4.1.3. Análise dos resultados

A análise dos resultados permite algumas conclusões:

a) para instâncias de até 100 cidades, o método apresenta performance similar quando comparado com os resultados da Tabela 2;

b) para instâncias grandes, a qualidade média decresce;

c) a versão assimétrica da instância `brg180` pode ser classificada como difícil para o MA proposto, possivelmente devido a baixa qualidade obtida na solução inicial construída pela heurística NN;

d) observa-se que a performance do método é pouco sensível ao parâmetro α_{max}, o que é mais uma característica de robustez do método.

Para as 32 instâncias de TSP simétricas que possuem uma rota ótima (juntamente com a instância) na TSPLIB, a solução ótima foi encontrada em média em 67% das execuções (α_{max} = 5%: 69% e α_{max} = 50%: 65%). A qualidade média da solução inicial foi de 102,15% (α_{max} = 5%: 90,6% e α_{max} = 50%: 113,7%) e passou para 1,37% na solução final (α_{max} = 5%: 1,16% e α_{max} = 50%: 1,59%). Os resultados considerando α_{max} = 50% são levemente piores, possivelmente devido à qualidade da solução inicial ser pior. O tempo computacional médio gasto foi de 152,61 seg.

Em conclusão, este novo conjunto de instâncias é bem resolvido pelo algoritmo proposto, com exceção da instância `brg180` que é um caso patológico que deve ser melhor estudado. Para as instâncias maiores que 200 cidades o percentual de instâncias resolvidas na otimalidade diminuiu (na verdade há poucos casos onde se encontra a solução ótima), mas estão dentro dos valores normalmente aceitos para os tempos de CPU utilizados como critério de parada (1000 segundos).

4.2. GERADOR DE INSTÂNCIAS DE HCP

O problema de ciclos hamiltoniandos (*Hamiltonian Cycle Problem* - HCP) pode ser definido como: dado um grafo G, o problema consiste em encontrar uma rota que passe por cada vértice uma única vez e retorne ao vértice de partida.

A TSPLIB dispõe de 9 instâncias de HCP e todas possuem pelo menos um ciclo hamiltoniano, sendo que uma delas (`alb4000`) possui dois (Reinelt, 1991). Ainda, a TSPLIB dispõe do código de um gerador de instâncias de HCP, implementado por M. Jünger e G. Rinaldi (Reinelt, 1991). Este gerador produz uma matriz quadrada completa de zeros (0) e uns (1) gerados aleatoriamente, sendo que a célula $i = j$ tem sempre o valor 0 (zero). Desta forma, não há garantia se a instância produzida possui ou não um ciclo hamiltoniano. Como o HCP é um problema de decisão, se um ciclo hamiltoniano de custo zero (0) for encontrado, a resposta é SIM, caso contrário, é NÃO.

O gerador aqui descrito produz uma instância que sempre possui pelo menos um ciclo hamiltoniano e fornece a rota formada por este ciclo em um arquivo *nomeDaInstância.otp.tour*.

Uma interface gráfica é fornecida para permitir interação com o usuário. Através desta, deve-se informar três dados fundamentais para o funcionamento do gerador:

- *dimensão*: o número de vértices que terá a instância;
- *minEdges*: o número mínimo de arcos adjacentes a cada vértice;
- *maxEdges*: o número máximo de arcos adjacentes a cada vértice;

Para cada vértice, o número de arcos adjacentes será um número aleatoriamente selecionado entre os valores de *minEdges* e *maxEdges*. Se o usuário desejar que o número de arcos adjacentes seja o mesmo para todos os vértices, deve informar *mimEdges* = *maxEdges*.

Inicialmente um ciclo hamiltoniano qualquer é criado e adicionado à lista de adjacências dos vértices correspondentes. Logo, a lista de adjacências de cada vértice é completada por vértices aleatórios. Teve-se o cuidado de não gerar dois arcos iguais e nem *loops* (um arco apontado para ele mesmo). Ao final, um arquivo com o nome *random*η*.hcp*, onde η representa sua dimensão, é criado, segundo o padrão TSPLIB, e recebe a informação contida na lista de adjacências dos vértices, seguido do valor -1 e do finalizador de arquivo EOF (padrão TSPLIB).

Da mesma forma que o gerador de instâncias de ATSP, este gerador foi implementado em Java e utilizando o mesmo gerador de números aleatórios, de tal forma que para os mesmos parâmetros de entrada, a instância gerada é sempre a mesma, independente da máquina onde é feita a execução.

Para exemplificar, as Figuras 22 e 23 mostram, respectivamente, uma instância do problema e um ciclo hamiltoniano gerados conforme os formatos da TSPLIB. Os parâmetros de entrada são 10, 2 e 4, para a dimensão, *minEdges* e *maxEdges*, respectivamente.

```
NAME : random10
COMMENT : Hamiltonian cycle instance generated randomicaly (Buriol, França and Moscato - 1999) with MinEdges: = 2 e maxEdges = 4
TYPE : HCP
DIMENSION : 10
Max e Min Edges: minEdges = 2 e maxEdges = 4
EDGE_DATA_FORMAT : EDGE_LIST
EDGE_DATA_SECTION
10 6
10 3
9 7
9 8
8 4
8 5
8 3
8 7
7 3
7 8
7 9
6 9
6 5
5 10
5 9
5 3
5 4
4 2
4 10
3 1
3 5
2 5
2 8
2 3
2 10
1 8
1 3
1 9
1 4
-1
EOF
```

Figura 22 - Arquivo `random10.hcp` gerado com dimensão 10, minEdges = 2 e maxEdges = 4.

Cada linha de dados indica um arco válido. Por exemplo, a primeira linha de dados (10 6) indica que a posição [10][6] da matriz recebe o valor 1, indicando a existência de um arco conectando estas duas cidades.

```
NAME: random10.tour
TYPE: TOUR
COMMENT: Optimal solution for random10
DIMENSION: 10
TOUR_SECTION
1 8 4 2 5 10 6 9 7 3
-1
EOF
```

Figura 23 - Arquivo random10.tour fornecido pelo gerador ao criar a instância da Figura 22.

Uma aspecto interessante deste gerador é que, além de gerar uma instância e fornecer uma solução, a instância, bem como a solução, são as mesmas tanto para o caso do problema de ciclos hamiltonianos direcionados quanto para o caso não direcionado.

A descrição do gerador usando *JavaDoc*, bem como seu código, estão disponíveis no endereço *internet http://www.densis.fee.unicamp.br/~buriol/HCPGenerator.html.*

4.3. CONCLUSÕES

O principal benefício de ambos geradores é que, devido à portabilidade de Java, sempre a mesma instância é criada, independente do computador ou compilador utilizados. Esta característica é desejável a qualquer gerador, visto que, necessitando das instâncias, o usuário poder gerá-las diretamente. Como já mencionado, o transporte de instâncias, principalmente de ATSP, é dificultado devido ao tamanho delas.

O gerador de instâncias de ATSP descrito necessita de uma instância simétrica e uma solução ótima para a mesma. Já o gerador de instâncias de HCP requer apenas que sejam fornecidos os parâmetros de entrada que são a dimensão e o número mínimo e máximo de arcos adjacentes a cada vértice.

Os resultados computacionais, utilizando o MA proposto nesta dissertação, demonstraram que, para instâncias de até 100 cidades, o comportamento é similar ao encontrado para as instâncias de ATSP disponíveis na TSPLIB. Para as instâncias maiores, o tempo computacional é maior e a qualidade da final ainda deve ser melhorada. A qualidade de solução final foi de 1,37%.

Capítulo

5

REDUÇÕES ENTRE PROBLEMAS

Duas medidas válidas podem ser adotadas no sentido de resolver um problema: aplicar uma técnica algorítmica específica para este problema ou reduzi-lo a outro problema que já tenha um algoritmo desenvolvido.

Um problema Q pode ser reduzido a um problema P em tempo $f(n)$ se uma instância genérica do problema Q puder ser transformada em uma instância específica do problema P em tempo $O(n)$, e, a partir de uma solução encontrada ao resolver o problema P, uma solução para o problema Q é possível de ser obtida também em tempo $O(n)$ (Karp, 1972). A variável n utilizada denota a dimensão da instância.

Diz-se instância genérica, pois a redução deve ser válida para qualquer instância daquele problema para poder usar a generalização de *redução entre problemas* e não *redução entre instâncias*.

A fim de exemplificar uma redução, considere a notação abaixo:

- I_P: conjunto de instâncias do problema P;
- $\vartheta_{f(n)}$: $I_Q \rightarrow I_P$: algoritmo transformador de instâncias genéricas do problema Q em instâncias específicas do problema P, executado em tempo $O(n)$;
- S_P: conjunto de soluções do problema P;
- S'_P: $S'_P \subset S_P$ é o conjunto de soluções do problema P geradas a partir de $\vartheta_{f(n)}$: $I_Q \rightarrow I_P$.
- $\xi_{f(n)}$: $S_P \rightarrow S_Q$: algoritmo transformador de soluções do problema P em soluções do problema Q, executado em tempo $O(n)$;

- *Resolve(P)*: algoritmo para resolver o problema *P*.

A redução do problema *Q* ao problema *P* pode ser representada na Figura 24, usando a sintaxe definida anteriormente:

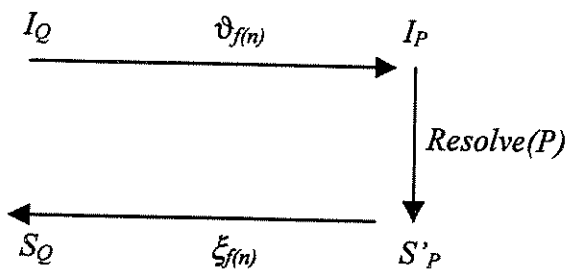


Figura 24 - Representação da redução do problema Q ao problema P.

A complexidade **C** de uma redução é calculada por: $\mathbf{C} = \mathbf{C}_{\vartheta} + \mathbf{C}_{Resolve} + \mathbf{C}_{\xi}$. Pela propriedade de soma de complexidades (Cormen et al., 1996, capítulo 2), a complexidade resultante é dominada pelo maior termo, ou seja, $\mathbf{C} = Max(\mathbf{C}_{\vartheta}, \mathbf{C}_{resolve}, \mathbf{C}_{\xi})$.

Segundo Allender et al. (1999) uma redução pode ser efetivada por restrição, por pequeno ajuste ou por transformação combinatorial.

Redução por restrição: a redução consiste de uma restrição do problema original para um caso especial.

Ex: HCP → TSP: considerar o custo das arestas originais da instância de HCP como tendo valor 1 (um) e completar o grafo com arestas de custo 2 (dois). O grafo possui um ciclo hamiltoniando se uma rota de custo igual a *n* (número de cidades) for encontrada.

Redução por pequeno ajuste: embora os problemas pareçam ser bem diferentes, a redução consiste numa pequena adaptação de um problema para outro. Para o exemplo, considere as definições dos problemas de clique e de conjunto independente.

Definição do problema de clique: dado um grafo G e um inteiro *k*, o problema consiste em encontrar um subconjunto K de *k* vértices tais que cada dois vértices em K sejam adjacentes em G.

Definição do problema de conjunto independente: dado um grafo G e um inteiro *k*, o problema consiste em encontrar um subconjunto de U de *k* vértices tais que não haja dois vértices em U que sejam adjacentes em G.

Ex: clique → conjunto independente: a partir do grafo G, gere seu complementar G'. Um grafo G tem um clique de tamanho *t* se, e somente se, seu grafo complementar G' tem um conjunto independente de tamanho *t*.

Redução por transformação combinatorial: o mecanismo combinatorial de um problema pode ser transformado por uma redução em um outro mecanismo aparentemente muito diferente do outro problema; é mais difícil de ser deduzida que as reduções por restrição ou por pequeno ajuste. Para o exemplo, considere a definição do problema 3SAT.

Definição de 3SAT: dada uma expressão boleana na forma normal conjuntiva com três literais por cláusula, o problema consiste em saber se a expressão pode ser satisfeita ou não.

Ex: 3Sat → conjunto independente: dada uma fórmula boleana (instância do 3Sat) composta pelas variáveis x_1, ..., x_n e pela cláusulas C_1, ..., C_m. O grafo G da redução é feito em três fases: (1ª) constrói-se um grafo composto por $2n$ vértices rotulados por x_1, $\sim x_1$, ..., x_n, $\sim x_n$, e pelos arcos (x_i, $\sim x_i$) para $1 \leq i \leq n$; (2ª) formam-se m subgrafos completos, cada um constituído por 3 vértices representado as variáveis x_i ou $\sim x_i$ que compõem cada cláusula; (3ª) para cada vértice dos subgrafos completos, adiciona-se um arco ligando este a seu complementar introduzido na 1ª fase. Caso um conjunto independente de tamanho $n + m$ seja encontrado no grafo G, as cláusulas de 3Sat são satisfeitas. Neste exemplo, $\sim i$ é utilizado como sendo negação de i.

Em geral, o procedimento de redução nos dois primeiros casos costumam ser simples, enquanto que o terceiro caso, em geral, consiste de um mecanismo mais complexo.

As reduções são implementadas de cima para baixo na árvore, ou seja, um problema que está no penúltimo nível da árvore é reduzido para o problema folha (da árvore) subseqüente. Portanto, a árvore da Figura 25 representa 13 reduções. Ainda pela árvore, um algoritmo para o TSP poderia resolver 5 problemas, além do próprio TSP. Isso é possível devido ao caminho de recursão transcrito por:

bounded tiling ➔ *satisfiability* ➔ *exact cover* ➔ *hamilton cycle* ➔ *undirected hamilton cycle*

⬇

traveling salesman problem

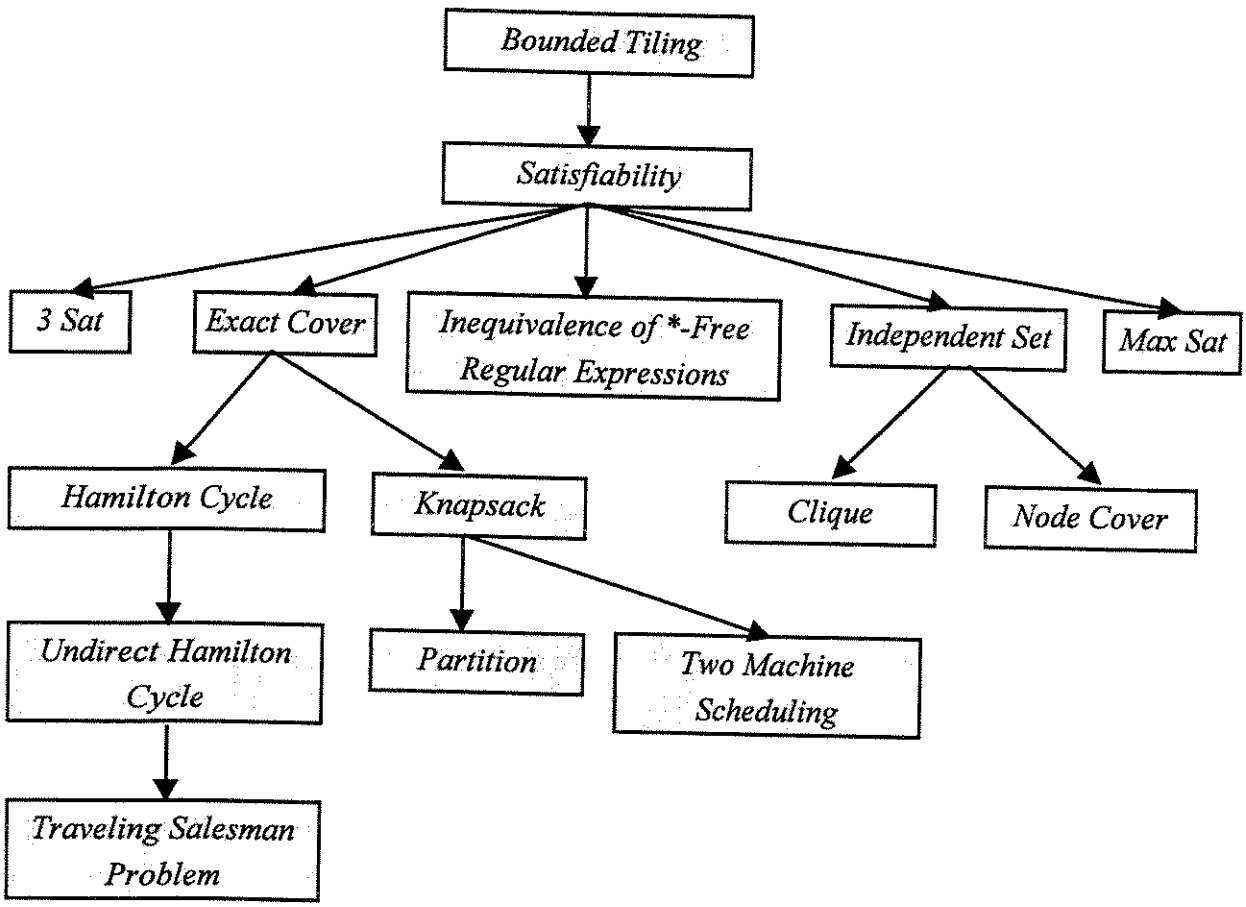


Figura 25 - Uma árvore de reduções entre alguns problemas combinatórios.

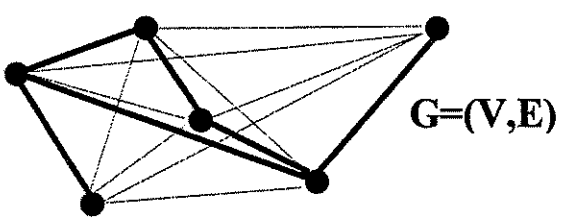
A partir de uma redução, é possível demostrar a complexidade intrínseca de problemas. Em vez de fazer a prova direta da complexidade de um problema, sua complexidade pode ser provada a partir de uma redução. Se um problema Q for reduzido a um problema P, há transferência de cota superior do problema P para o problema Q e de conta inferior do problema Q para o problema P (Lewis e Papadimitriou, 1998).

5.1. REDUÇÃO DE UHCP PARA STSP

Considerando a notação para reduções definida no início deste capítulo, esta será considerada para exemplificar a redução do problema de ciclos hamiltonianos não direcionados (*Undirect Hamiltonian Cycle Problem* - UHCP) para o STSP.

- I_{UHCP}: grafo simétrico G = (V, E), onde V representa o conjunto de n vértices e E o conjunto de m arestas.
- I_{STSP}: matriz n x n de distâncias D_{ij} entre cada cidade i e j.
- ϑ: $I_{UHCP} \rightarrow I_{STSP}$: $n = |V|$ e a distância d_{ij} entre cada cidade i e j é preenchida da seguinte forma (veja Figura 26): $d_{ij} = \begin{cases} 0 & \text{se } i = j; \\ 1 & \text{se } (v_i, v_j) \text{ ou } (v_j, v_i) \in G; \\ 2 & \text{caso contrário.} \end{cases}$

Desde que G seja um grafo simétrico sem *loops*, a matriz de distâncias será simétrica.



— Arcos do grafo original ⇒ custo 1

— Arcos adicionados pela redução ⇒ custo 2

Figura 26 - Instância de UHCP resultante da redução de UHCP para STSP.

- *Resolve(STSP)*: qualquer algoritmo, heurístico ou exato, para resolução do STSP. É importante observar que toda solução gerada por *Resolve(STSP)* terá custo igual a $n + \varepsilon$, sendo que ε representa o número de arcos da rota com custo igual a 2.
- S_{STSP}: qualquer ciclo hamiltoniano que conecte todas as cidades sem repetir nenhuma.
- S'_{STSP}: qualquer ciclo hamiltoniano, composto apenas por arcos de custo 1, ou seja, o custo do ciclo obrigatoriamente deverá ser igual a n, pois $\varepsilon = 0$.
- $\xi_{f(I)}$: $S_{STSP} \rightarrow S_{UHCP}$: não há necessidade desta transformação visto que os dois problemas têm um ciclo hamiltoniano como solução.

5.1.1 Resultados computacionais

A Tabela 7 apresenta os resultados computacionais encontrados com a utilização do MA SAX/RAI descrito no capítulo 3, aplicado à redução de UHCP para o STSP. As duas últimas instâncias, alb1000 e alb2000 são provenientes da TSPLIB. Já as demais, são instâncias criadas pelo gerador de instâncias de HCP descrito na seção 4.2. A dimensão da instância é representada pelo número que aparece após a palavra *random*, *MinEdges* = 4 e *MaxEdges* = 8.

Tabela 7 - Resultados computacionais da utilização de SAX/RAI à redução de UHCP para STSP.

Instância	*Otm/#exe*	*IniPop (%)*	*Qual (%)*	*Gen*	*CPU BL (%)*	*Tempo (seg.)*
random20	30/30	5	0,00	1,4	3	0,03
random40	30/30	7	0,00	1,9	10	0,03
random60	30/30	8	0,00	2,1	28	0,04
random80	30/30	8	0,00	2,0	27	0,06
random100	30/30	9	0,00	2,2	30	0,07
random150	30/30	8	0,00	3,5	51	0,23
random200	30/30	9	0,00	10,2	70	1,29
random250	30/30	9	0,00	26,4	79	5,91
random300	30/30	9	0,00	75,8	85	27,98
random350	16/30	9	0,13	362,2	89	216,37
random400	01/30	10	0,32	359,9	90	297,89
random450	00/30	10	0,45	276,0	92	300,59
random500	00/30	10	0,59	218,9	73	300,75
random600	00/30	6	0,70	492,6	93	1001,04
random700	00/30	9	0,87	355,7	94	1001,33
random800	00/30	10	1,05	269,27	94	1001,48
random900	00/30	10	1,24	211,07	95	1002,42
random1000	00/30	14	1,24	171,87	95	1003,46
alb1000	00/30	17	4,29	235,90	94	1002,14
alb2000	00/30	16	5,23	51,766	96	1009,48
Médias	14,3/30	9,6	0,80	156,5	69,4	408,63

Assim como para todos os demais resultados computacionais que constam nesta dissertação, os resultados contidos na Tabela 7 representam a média de 30 execuções para cada instância. Os parâmetros do algoritmo e o computador utilizado são os mesmos mencionados no capítulo 3. A coluna de solução ótima é suprimida porque, como visto na redução, a solução ótima é igual a *n*. Por exemplo, para a instância *random20* a solução ótima é 20.

5.1.2 Análise dos resultados

Para instâncias de até 300 cidades, a solução ótima foi sempre encontrada, num tempo computacional médio de 3,96 seg. Para problemas com mais de 400 cidades, a solução ótima não foi encontrada em qualquer execução, mesmo para os problemas com mais de 500 cidades que exigem tempo de execução de 1000 seg. Em média, a solução ótima foi encontrada em 48% das execuções.

A solução inicial média está mais próxima da ótima (*Qual (%)* = 9,6%), se comparada com resultados para instâncias de mesmo tamanho vistos anteriormente nesta dissertação, porque não há arcos de custo elevado, apenas arcos de custo 1 ou 2. Nas outras tabelas já apresentadas na dissertação, a população inicial tinha maior custo porque, caso um arco não pertencente à rota fosse incluído, este poderia ser dezenas de vezes maior que outros arcos da rota. No caso das instâncias da Tabela 7, o maior arco tem custo 2 e o menor tem custo 1.

A média da qualidade de solução na Tabela 7 foi de 0,8%, com um tempo médio de 408,63 seg.

Uma análise mais completa poderia ser feita se os resultados da Tabela 7 fossem comparados com resultados obtidos por um algoritmo especialmente desenvolvido para o problema de ciclos hamiltonianos não direcionados. Essa análise não é simples devido a variedade de algoritmos existentes para HCP. Como referência, pode-se citar Vandegriend (1998), pois reúne uma série de técnicas, estudos teóricos e algoritmos aplicados à instâncias (fáceis e difíceis) do problema.

Uma análise mais aprofundada ainda poderia fazer a comparação entre os resultados de algoritmos para o TSP aplicados diretamente ao problema e também a instâncias reduzidas de HCP.

Para fazer estas comparações seria necessário uma extensa pesquisa bibliográfica. Esta, porém, será colocada como sugestão para trabalhos futuros.

5.2. REDUÇÃO DE DHCP PARA O ATSP

A mesma notação para reduções definida no início deste capítulo será considerada para exemplificar a redução do problema de ciclos hamiltonianos direcionados (*Direct Hamiltonian Cycle Problem* - DHPC) para o ATSP. Esta redução foi disponibilizada motivada pela sua fácil implementação; computacionalmente se trata de pequenas diferenças entre o código da redução de UHCP para STSP.

- I_{DHCP}: grafo direcionado G = (V, E), onde V representa o conjunto de *n* vértices e E o conjunto de *m* arcos.
- I_{ATSP}: matriz *n* x *n* de distâncias d_{ij} entre cada cidade *i* e *j*.
- ϑ: $I_{DHCP} \rightarrow I_{ATSP}$: $n = |V|$ e a distância d_{ij} entre cada cidade *i* e *j* é preenchida da seguinte forma:

$$d_{ij} = \begin{cases} 0 & \text{se } i = j; \\ 1 & \text{se } (v_i, v_j) \in G; \\ 2 & \text{caso contrário.} \end{cases}$$

- *Resolve(ATSP)*: qualquer algoritmo, heurístico ou exato, para resolução do ATSP.

É importante observar que toda solução gerada por *Resolve(ATSP)* terá custo igual a $n + \varepsilon$, sendo que ε representa o número de arcos da rota com custo igual a 2.

- S_{ATSP}: qualquer ciclo hamiltoniano que conecte todas as cidades sem repetir nenhuma.
- S'_{ATSP}: qualquer ciclo hamiltoniano, composto apenas por arcos de custo 1, ou seja, o custo do ciclo obrigatoriamente deverá ser igual a *n*, pois $\varepsilon = 0$.
- $\xi_{f(I)}$: $S_{ATSP} \rightarrow S_{DHCP}$: não há necessidade desta transformação visto que os dois problemas têm um ciclo hamiltoniano como solução.

Esta redução é muito semelhante à apresentada na seção anterior, com a diferença de que apenas $d_{ij} = 1$ se $(v_j, v_i) \in E$. Por isso, considerou-se suficiente analisar os resultados da Tabela 7 para se chegar a uma conclusão sobre a performance do algoritmo quando aplicado à instâncias reduzidas de HCP. Embora os resultados para esta redução não tenham sido gerados, os resultados para a redução para STSP tendem a ser melhores, pois as instâncias reduzidas contêm o dobro de arcos de custo 1 que as instâncias reduzidas para ATSP.

5.3. CONCLUSÕES

A complexidade da transformação das instâncias é, para ambos os casos, O(max(n^2, $n*m$)), onde m representa o número de arcos/arestas do vértice com maior grau da instância de HCP (Cormen et al., 1996, capítulo 23). Em geral, a complexidade é O(n^2), mas caso a instância de HCP a ser reduzida contiver algum arco/aresta com $m > n$, a complexidade será O($n*m$). Já a complexidade da transformação da solução é O(1), visto que a solução encontrada é solução do problema reduzido. No caso desta redução, certamente a complexidade dominante será dos algoritmos *Resolve(ATSP)* e *Resolve (STSP)*, pois estes problemas pertencem à classe de problemas NP-difíceis e não possuem um algoritmo de resolução em tempo O(n^2) ou O($n*m$).

Embora não tenha sido feita uma comparação dos resultados da Tabela 7 com resultados obtidos por outros métodos, eles podem ser considerados satisfatórios. Para 20 instâncias, de 20 a 2000 cidades, a média da qualidade da solução foi de 0,8% em um tempo computacional de 408,63 seg. Até 300 cidades a solução ótima foi encontrada em todas as instâncias, enquanto que para instâncias maiores de 400 cidades, a solução ótima não foi encontrada nenhuma vez.

Como resultados computacionais para instâncias reduzidas são pouco publicados, a possibilidade de um método ter sucesso na resolução de instâncias reduzidas é muito grande. Por exemplo: assim como as instâncias difíceis de ATSP tiveram melhor qualidade com população não estruturada, um método poderia resolver com melhor performance instâncias reduzidas que instâncias do problema original de mesma dimensão.

Capítulo

6

PROJETO INICIAL DO FRAMEWORK

Um *framework* é um conjunto de classes colaborativas (abstratas e concretas) que faz reutilização de projeto e provê uma infra-estrutura genérica de soluções para um conjunto de problemas específicos; é uma abstração de um conjunto de classes interrelacionadas (Szyperky, 1998; Gamma et al., 1995).

Aplicações que fazem uso de bibliotecas de classes possuem o controle (fluxo de eventos) localizado na própria aplicação e fazem chamadas de rotinas de uma biblioteca. Isto é, o programador da implementação é encarregado de implementar o fluxo de controle da aplicação específica. Aplicações baseadas em conjunto de classes colaborativas (*frameworks*) reutilizam o fluxo de eventos já embutido. O programador que utiliza um *framework* para construir uma aplicação específica implementa apenas o código que será chamado por este (*callbacks*). Ainda, um cliente que queira adicionar código a esta, apenas deve entender como funciona seu fluxo e qual o procedimento para inserção de uma nova parte.

As três maiores vantagens no uso de *frameworks* são:

a) provê uma infra-estrutura bem projetada, reduzindo codificação, testes e gastos com *debugging*. Aplicações desenvolvidas com *frameworks* tendem a ser menores, mais fáceis de manter e reutilizar;

b) como provê um código flexível (com as propriedades de extensibilidade e reutilidade), adições de partes podem ser feitas sem grandes esforços, pois suas interfaces são bem definidas e seu código bem modularizado;

c) manutenção reduzida: por todas suas características já mencionadas, um *framework* reduz de forma considerável os custos de manutenção do *software*. Além disso, como adições de novo código não faz com que todo código tenha que se adaptar, e sim só uma parte, a chance da ocorrência de novos erros diminui.

O *framework* para programação evolutiva aqui proposto é descrito em termos do seu diagrama de classes em UML. O *software* utilizado para fazer a modelagem em UML é o *Rational Rose* da *Rational*, disponível gratuitamente na *Internet*[2]. Para eventuais dúvidas sobre Java, UML e orientação a objetios, o Apêndice 1 pode ser consultado, pois foi adicionado com esse intuito.

Atualmente, alguns *frameworks* para algoritmos evolucionários estão sendo construídos, alguns especificamente para algoritmos meméticos. Dentre eles, pode-se citar como referências os *frameworks* de Krasnogor e Smith (1999) e Costa et al. (1999).

6.1. *Memepool*: A CLASSE CENTRAL DO *FRAMEWORK*

O método *main( )*, que dá inicio à execução do *framework*, encontra-se na classe *Memepool*. Esta classe considera dados supostamente fornecidos pelo usuário e seleciona as operações e os operadores que serão utilizados no restante da aplicação.

Por se tratar de um algoritmo populacional, um algoritmo construtivo e um operador de recombinação obrigatoriamente deverão ser selecionados. O algoritmo construtivo pode ser acessado por uma referenciada da classe *DiCycleConstructionAlgorithms*, validada (recebe uma referência válida) pelo método *selectConstructAlgoritm( )*; já o operador de recombinação pode ser acessado por uma referência da classe *DiCycleCrossoverOperator*, validada pelo método *selectCrossoverOperator( )*.

Caso o usuário selecione um operador de busca local, um algoritmo memético será executado, caso contrário, a execução consiste de um algoritmo genético. No caso de ter

selecionado um operador de busca local, o operador selecionado poderá ser acessado através de uma referência da classe *DiCycleLocalSearchOperator*, validada pelo método *selectLocalSearchOperator( )*.

Os operadores de *restart* e mutação são opcionais, e somente serão executados caso o usuário tenha feito alguma seleção. No caso de um operador de *restart* e um de mutação, a instância da classe correspondente à cada seleção poderá ser acessada por referências das classes *DiCycleRestarOperator* e *DiCycleMutationOperator*, respectivamente validadas pelos métodos *selectRestarOperator( )* e *selectMutationOperator( )*.

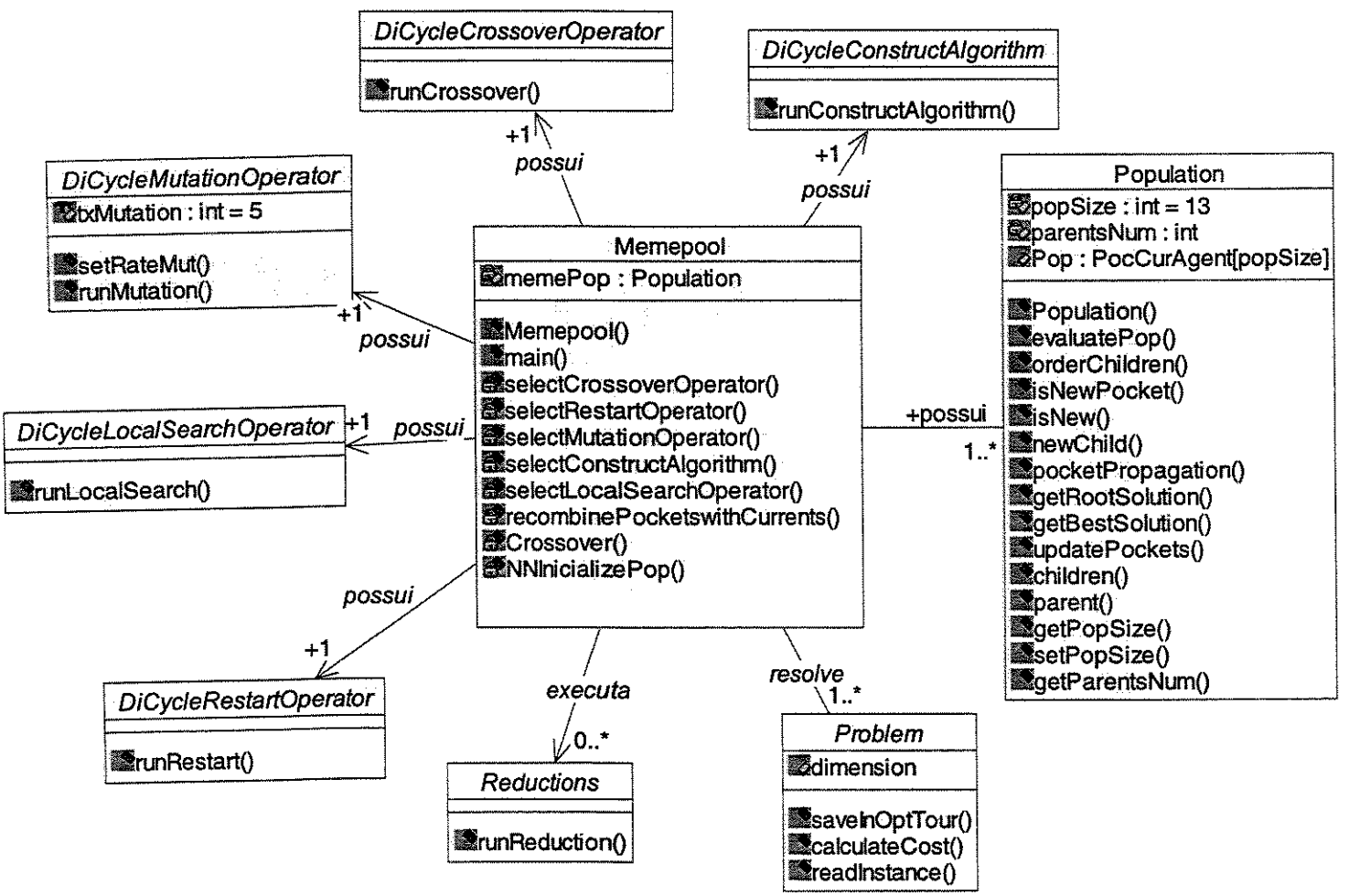


Figura 27 - *Memepool*: a classe central do *framework*.

[2] http://www.rational.com/products/rose

Uma variável que faz referência a um objeto do tipo População é sempre definida, por se tratar de um algoritmo populacional. O tamanho dessa população também é indicado pelo usuário e cada algoritmo populacional pode conter mais que uma população, embora, em geral, tais algoritmos contenham uma única população.

No caso de uma redução, uma instância da classe da redução selecionada será alocada, passando os dados de entrada do problema a ser reduzido e retornado os dados de entrada para o problema a que foi reduzido. A multiplicidade de cada redução é de *0...n* pois o usuário pode não fazer redução assim como pode fazer tantas quanto possível. A classe *Problem* possui multiplicidade *1...n* pois o número de instâncias que serão alocadas desta classe depende do número de reduções da execução do MA.

6.2. A CLASSE *Population*

A classe *Memepool* aloca instância(s) da classe *Population* e a esta(s) é informado um valor que define o número de agentes dessa população, estabelecido pelo usuário. Caso nenhum valor seja informado, a população é inicializada com 13 agentes. O relacionamento entre estas duas classes é uma associação em que cada classe *Memepool* pode conter uma ou mais populações.

Os métodos dessa classe são todos públicos e são os responsáveis por estruturar a população em uma árvore ternária completa (*orderChildren( ), pocketPropagation( )* e *updatePocket( )*), verifica se há indivíduos na população com o mesmo custo de uma solução nova (*isNew()*), verificar se há algum *pocket* na população com o mesmo custo de uma solução nova (*isNewPocket( )*), inserir uma solução nova na população (*newChild( )*), avaliar a população conforme a função de *fitness* definida (*evaluatePop( )*), obter a solução do *Pocket* raiz e a melhor solução da população (*getRootSolution( )* e *getBestSolution( )*), obter os índices dos filhos de um pai informado ou o pai de um filho informado (*children( )* e *parent( )*), obter e modificar o valor de *popSize* (*getPopSize( )* e *setPopSize( )*), obter o valor do número de indivíduos não folhas (*getParentsNum( )*), além do método construtor, responsável pela inicialização da população.

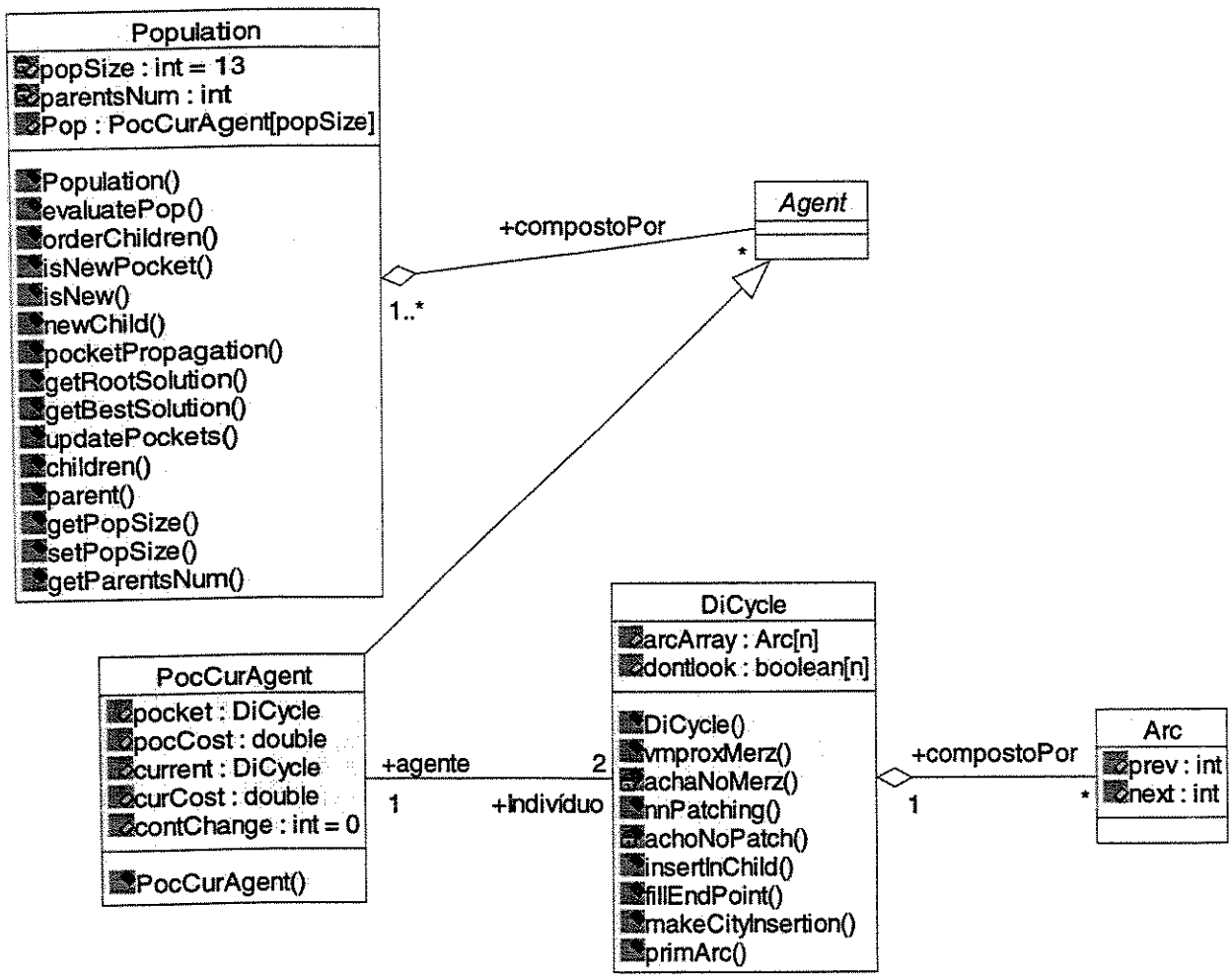


Figura 28 - As classes envolvidas na criação da população.

A classe *Population* possui três atributos: *popSize*, *parentsNum* e *Pop*. O atributo *popSize* armazena o número de agentes da população e *parentsNum* o número de agentes pais (não folhas) da mesma. Este último é calculado por uma fórmula (menor inteiro de *popSize*/3). Isto é possível devido a árvore ser ternária completa. *Pop* é um vetor de agentes que representa a população.

A população é composta por agentes, necessitando para isso de um relacionamento de composição entre as classes *Population* e *Agent*, indicando que cada instância da classe *Population* pode ter quantas instâncias da classe *PocCurAgent* desejar (o número de instâncias não é aceito caso não corresponda ao número de indivíduos que completam uma árvore ternária).

A classe *PocCurAgent* é herdada de *Agent*, pois outros tipos de *Agent* poderiam ser implementados. Entre as classes *PocCurAgent* e *DiCycle* há um relacionamento de associação indicando que cada instância da classe *PocCurAgent* possui duas instâncias da classe *DiCycle*

(*pocket e current*). A classe *PocCurAgent* ainda possui três atributos: *pocCost*, *curCost* e *contChange* que representam, respectivamente, o custo da rota *pocket*, o custo da rota *current* e um contador do número de vezes que o *pocket* do agente correspondente não foi trocado com o *current* (*curCost* < *pocCost*) no procedimento *updatePocket*. A classe possui um construtor que faz a inicialização dos atributos da classe.

A classe *DiCycle* é um vetor de *n* posições, cada posição contendo um objeto do tipo *Arc*, o que permite representar uma rota (Figura 3) e por isso possui um relacionamento de composição com a classe *Arc*. *Arc* é uma classe que possui como atributos dois inteiros (*prev* e *next*) que permitem indicar as cidades extremos de um arco. *DiCycle* ainda possui um vetor de boleanos, denominado *dontlook*, que permite fazer a manipulação dos *don't look bits*.

Se o usuário desejar utilizar outro tipo de agente, a classe correspondente ao novo agente deve ser colocada ao lado de *PocCurAgent*, ou seja, herdando da classe *Agent*. Da mesma forma, a adição de uma nova representação de solução deve ser herdada do agente a que está associada.

6.3. SELEÇÃO DE UM OPERADOR DE RECOMBINAÇÃO

Quando um operador de recombinação é alocado, na realidade é feito um *upcasting* (veja seção 6.3.10). Dessa forma, o método *selectCrossoverOperator( )*, definido na classe *Memepool*, aloca uma instância da classe correspondente ao operador selecionado (UNN, SAX, etc.) e faz um *upcasting* para uma referência da classe *DiCycleCrossover*.

O método *selectCrossoverAlgorithm( )* da classe *Memepool* possui dois parâmetros: *refCrossover*, que é uma referência da classe *DiCycleCrossoverOperator*, e *OPCrossover*, que é uma *string* contendo a seleção do usuário. Dependendo do operador selecionado pelo usuário, uma instância da classe correspondente é alocada e é feito um *upcasting* para a referência da classe *DiCycleCrossoverOperator* previamente criada. Devido ao *upcasting*, durante a execução do programa, a invocação do operador de recombinação é feita pelo método *runCrossover( )* da classe *DiCycleCrossoverOperator*. Ao acessar este método, na realidade o método polimórfico correspondente ao operador de recombinação selecionado é executado.

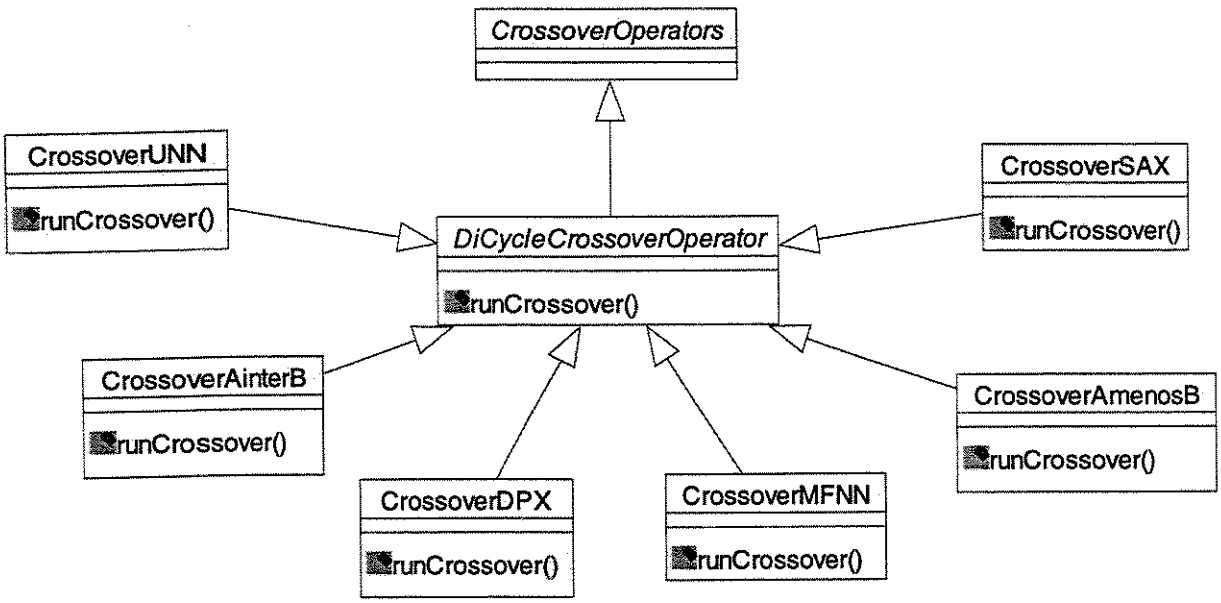


Figura 29 - Seleção do operador de recombinação.

O uso de *upcasting* é utilizado com o mesmo objetivo para seleção dos operadores de busca local, mutação, restart e para seleção do problema que está sendo tratado.

A classe abstrata *DiCycleCrossoverOperator* possui um método polimórfico *runCrossover( )* que é redefinido pelas subclasses *CrossoverUNN*, *CrossoverSAX*, *CrossoverDPX*, *CrossoverMFNN*, *CrossoverAinterB* e *CrossoverAmenosB*. Devido ao *upcasting*, quando este método for invocado a partir de uma referência da classe *DiCycleCrossoverOperator*, o método *runCrossover( )* de uma das classes filhas executará.

```
public DiCycleCrossover selectCrossoverOperator(DiCycleCrossover refCrossover, String OPCrossover){
  if (OPCrossover.equals("Strategic Arc Crossover - SAX")){
        CrossoverSAX sax = new CrossoverSAX();
        sax.setProblem(prob);
        refCrossover=sax;  /*upcasting*/
  }
  else if (OPCrossover.equals("Distance Preserving Crossover - DPX")){
        CrossoverDPX dpx = new CrossoverDPX();
        refCrossover = dpx; /*upcasting*/
  }
```

```
else if (OPCrossover.equals("Multiple Fragment Nearest Neighbour - MFNN")){
        CrossoverMFNN mfnn = new CrossoverMFNN();
        refCrossover = mfnn; /*upcasting*/
  }
else if (OPCrossover.equals("Uniform Nearest Neighbor - UNN")){
        CrossoverUNN unn = new CrossoverUNN();
        refCrossover = unn; /*upcasting*/
}
else if (OPCrossover.equals("AinterB")){
        CrossoverAinterB AiB = new CrossoverAinterB();
        refCrossover = AiB; /*upcasting*/
  }
else if (OPCrossover.equals("AmenosB")){
        CrossoverAmenosB AmB = new CrossoverAmenosB();
        refCrossover = AmB; /*upcasting*/
  }
else {
        Print("Erro: nenhum operador de recombinação foi selecionado");
        RefCrossover = null;
}
return(refCrossover);
```

Figura 30 - Método *selectCrossoverOperator()* da classe *Memepool*.

Os operadores de recombinação possuem métodos auxiliares e por isso são todos privados. Estes métodos são omitidos dos diagramas para tornar a representação mais clara.

Se o usuário desejar incluir outro operador de recombinação, apenas deve-se herdá-lo da classe *DiCycleCrossoverOperator*. No caso da inclusão de um operador para uma estrutura que não seja *DiCycle*, uma classe abstrata para esta estrutura (assim como *DiCycleCrossoverOperator*) deve ser criada e herdada de *CrossoverOperators* e o operador implementado deve ser herdado desta nova classe.

6.4. SELEÇÃO DE UM ALGORITMO CONSTRUTIVO

Quando um algoritmo construtivo é selecionado, na realidade é feito um *upcasting*. Dessa forma, o método *selectConstructAlgorithm( )*, definido na classe *Memepool*, aloca uma instância da classe correspondente ao algoritmo selecionado e faz um *upcasting* para uma referência da classe *DiCycleConstructAlgorithm*.

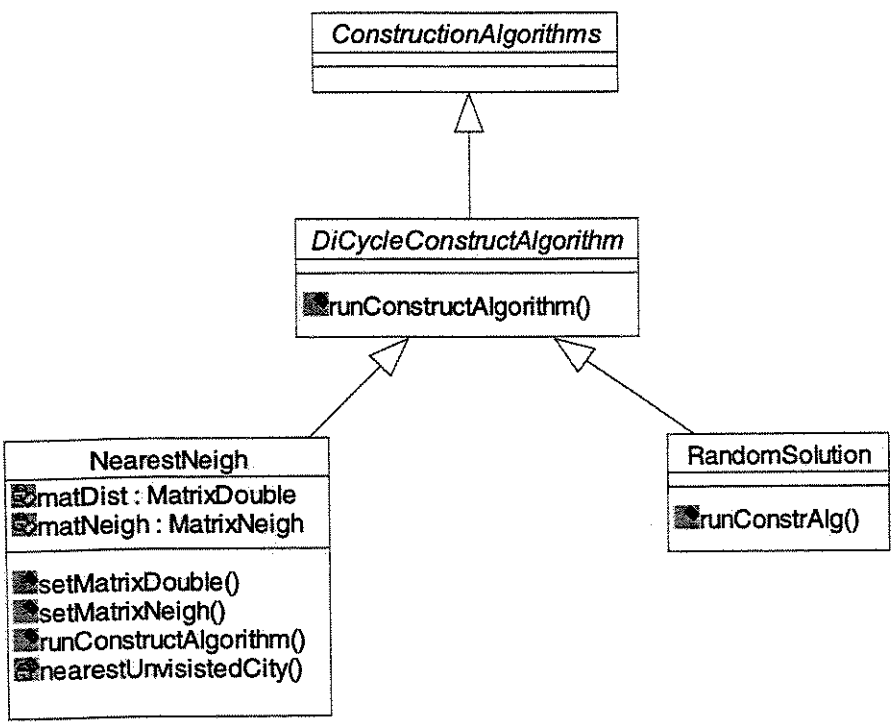


Figura 31 - Seleção de um algoritmo construtivo.

A classe abstrata *DiCycleConstructAlgorithm* possui um método polimórfico *runConstructAlgorithm( )* que é redefinido pelas subclasses *RandomSolution* e *NearesNeigh*. Devido ao *upcasting*, quando este método for invocado a partir da referência da classe *DiCycleConstructAlgorithm*, somente o método *runConstructAlgorithm( )* de uma das classes filhas executará.

A classe *RandomSolution* possui o método *runConstrAlg( )* que gera uma solução factível aleatoriamente. Já a classe *NearestNeigh* gera uma solução pela heurística do "vizinho mais próximo". Esta heurística requer a matriz de distâncias (passada pelo método *setMatrixDouble( )*) e, no caso desta implementação, utiliza a matriz de vizinhança (passada pelo método

setMatrixNeigh( )), caso ela tenha sido inicializada. O método *nearestUnVisitedCity( )* é um método auxiliar do método *runConstrAlg( )* e por isso é privado.

Se o usuário desejar incluir um novo operador de recombinação, apenas deve herdá-lo da classe *DiCycleConstructinAlgorithm( )*. No caso da inclusão de um operador para uma estrutura que não seja *DiCycle*, uma classe abstrata para esta estrutura (assim como *DiCycleConstructinAlgorithm*) deve ser criada e herdada de *ConstructinAlgorithms* e o operador implementado deve ser herdado desta nova classe.

6.5. SELEÇÃO DE UM OPERADOR DE BUSCA LOCAL

Assim como para a seleção de um operador de crossover e um algoritmo construtivo, quando um operador de busca local é selecionado, na realidade é feito um *upcasting*. Dessa forma, o método *selectLocalSearchOperator( )*, definido na classe *Memepool*, aloca uma instância da classe correspondente ao algoritmo selecionado e faz um *upcasting* para a classe *DiCycleLocalSearchOperator*.

A classe abstrata *DiCycleLocalSearchOperator* possui um método polimórfico *runLocalSearch( )* que é redefinido pela subclasse *LocalSearchRAI*. Devido ao *upcasting*, quando este método for invocado a partir de uma referência da classe *DiCycleLocalSearchOperator*, somente o método *runLocalSearch( )* da classe *LocalSearchRAI* executará.

Na implementação, somente a classe *LocalSearchRAI* herda de *DiCycleLocalSearchOperator*. Essa classe possui dois métodos públicos: *runLocalSearch( )* e *LocalSearchRAI( )*. O método *runLocalSearch( )* invoca o método *LocalSearchRAI( )* para cada cidade *i* marcada com o atributo *dontlook*[*i*] setado para falso (da rota passada para o método). O método *LocalSearchRAI( )* é declarado público, pois pode ser acessado externamente, sem a necessidade do método *runLocalSearch( )*, desde que seja informado, além da rota a ser otimizada, uma cidade inicial para a busca.

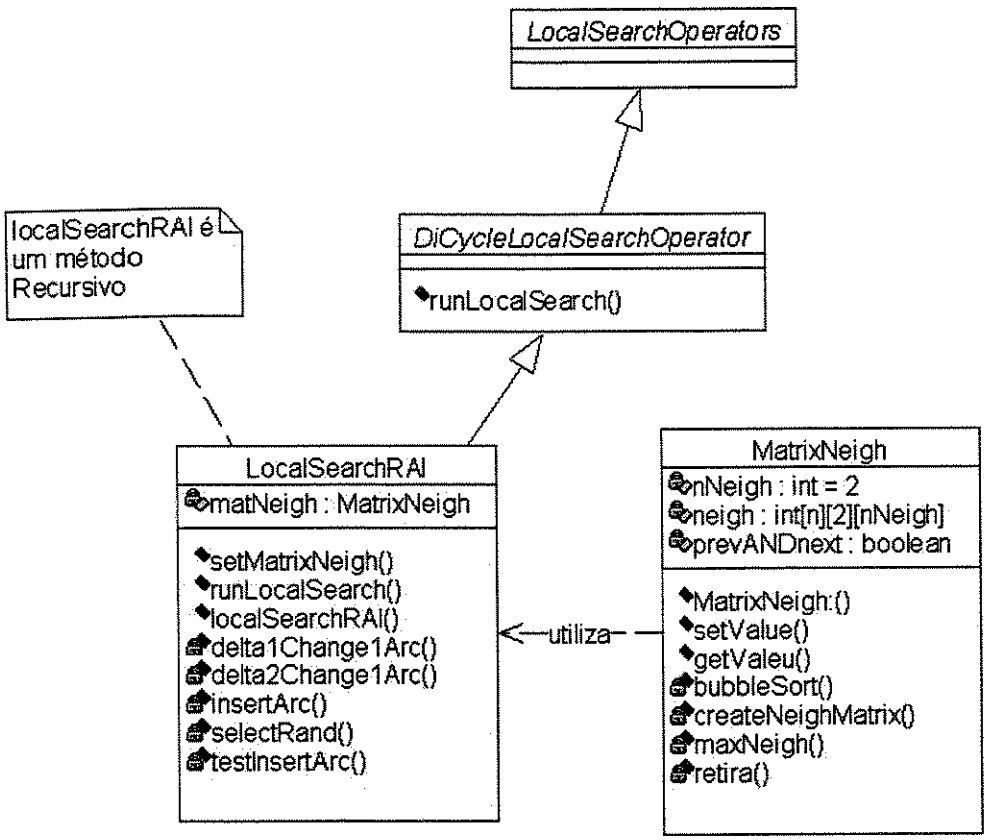


Figura 32 - Seleção do operador de busca local.

O único atributo da classe *LocalSearchRAI* é uma referência a um objeto da classe *MatrixNeigh*. Quando esta referência é alocada, o atributo *neigh*, que representa a matriz de vizinhança é alocada e devidamente preenchida. O atributo *neigh* é uma matriz tridimensional que armazena, para cada cidade, as vizinhas sucessoras e antecessoras (*s-vizinhos* e *p-vizinhos*, veja seção 3.4) mais próximas. Se o atributo boleano *prevANDnext* for falso (caso simétrico), somente a matriz de *s-vizinhos* é alocada, caso contrário (caso assimétrico) as matrizes de *s-vizinhos e p-vizinhos* são alocadas. O atributo *nNeigh* da classe *MatrixNeigh* indica o tamanho da vizinhaça que será criado. No caso da implementação, *s* e *p* sempre terão o mesmo tamanho. O método *createNeighMatrix( )* é invocado pelo construtor da classe *MatrixNeigh( )* e por isso é privado, assim como seus métodos auxiliares *maxNeigh( )* e *retira( )*. Como matriz *neigh* é um atributo privado, esta somente pode ser acessada por métodos da classe. No caso, *neigh* é preenchida e consultada pelos métodos *setValue( )* e *getValue( )*, respectivamente.

6.6. SELEÇÃO DE UM OPERADOR DE MUTAÇÃO

O operador de mutação também é submetido a um *upcasting*, pois o método *selectConstructionAlgorithm( )*, definido na classe *Memepool*, aloca uma instância da classe correspondente ao algoritmo selecionado e faz um *upcasting* para um referência da classe *DiCycleConstructAlgorithm*.

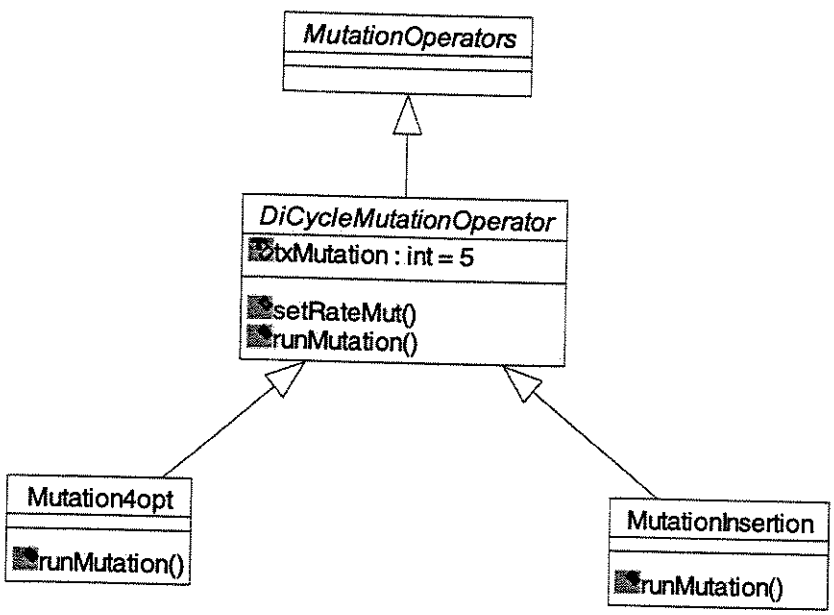


Figura 33 - Seleção do operador de mutação.

A classe abstrata *DiCycleMutationOperator* possui um único atributo que representa a taxa de mutação selecionada pelo usuário (caso nenhuma seleção seja feita, a taxa é setada para 5%). Este atributo é protegido pois somente deve ser acessado pelas classes filhas. O método polimórfico *runMutation( )* é redefinido pelas subclasses *Mutation4opt* e *MutationInsertion*. Devido ao *upcasting*, quando este método for invocado a partir da instância da classe *DiCycleMutationOperator*, somente o método *runMutation( )* de uma das classes filhas executará.

A classe Mutation4opt, embora não tenha sido descrita no capítulo 3, está disponível no código do *framework*. É um operador simples que executa o procedimento 4-opt entre quatro cidades aleatoriamente selecionadas.

6.7. SELEÇÃO DE UM OPERADOR DE *RESTART*

O operador de *restart* é submetido a um *upcasting* de forma que o método *selectRestartOperator( )*, definido na classe *Memepool*, aloca uma instância da classe correspondente ao algoritmo selecionado e faz um *upcasting* para uma referência da classe *DiCycleRestartOperators*.

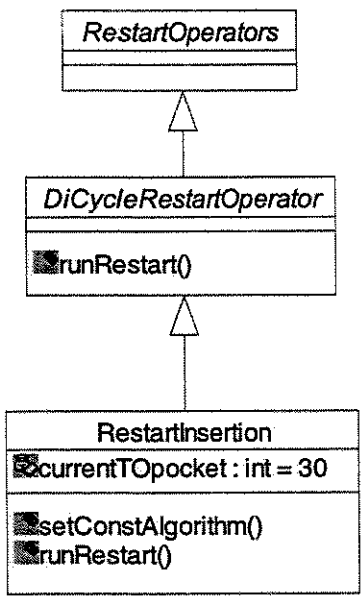


Figura 34 - Seleção do operador de restart.

A classe abstrata *DiCycleRestartOperator* possui um único atributo, *currentTOpocket*, que representa o número máximo de iterações que a rota *Pocket* de um agente líder pode permanecer sem ser trocada pela rota *Current* (pelo procedimento *updatePocket*). O operador de *restart* é aplicado quando o atributo *contChange*, da classe *PocketCurrentAgent*, tiver o mesmo valor que *currentTOpocket*. O método polimórfico *runRestart( )* é redefinido pela subclasse *RestartInsertion*. Devido ao *upcasting*, quando este método for invocado a partir da instância da classe *DiCycleRestartOperator*, o método *runRestart( )* da classe filha RestartInsertion executará.

O método *setConstAlgorithm( )* informa o algoritmo construtor selecionado para gerar novas soluções (veja operador de *restart* na seção 3.5.3.).

6.8. SELEÇÃO DO PROBLEMA A SER RESOLVIDO

Problem é uma classe abstrata que contém os métodos necessários para diferienciar um problema de outro. Para tanto, três métodos são fornecidos para que sejam redefinidos em cada problema: *saveInOptTour()*, *calculateCost()* e *readInstance()*.

A classe *MatrixDouble* contém como atributo uma matriz de *doubles* denominda *mat*. Por ser privada, seus dados somente podem ser adicionados ou consultados pelos métodos públicos da classe: *setDimension( )* (método sobrecarregado), *setValue( )* e *getValue( )*. Esta matriz armazenará a matriz de distâncias para o TSP (simétrico e assimétrico) e o grafo do HCP (direcionado e não direcionado). Todas as classes derivadas de *Problem* possui um atributo público, *matDist*, que é uma referência à instância de *MatrixDouble*. Essa alocação é feita na classe *Memepool*, pois somente uma instância dessa classe é necessária para a execução do MA.

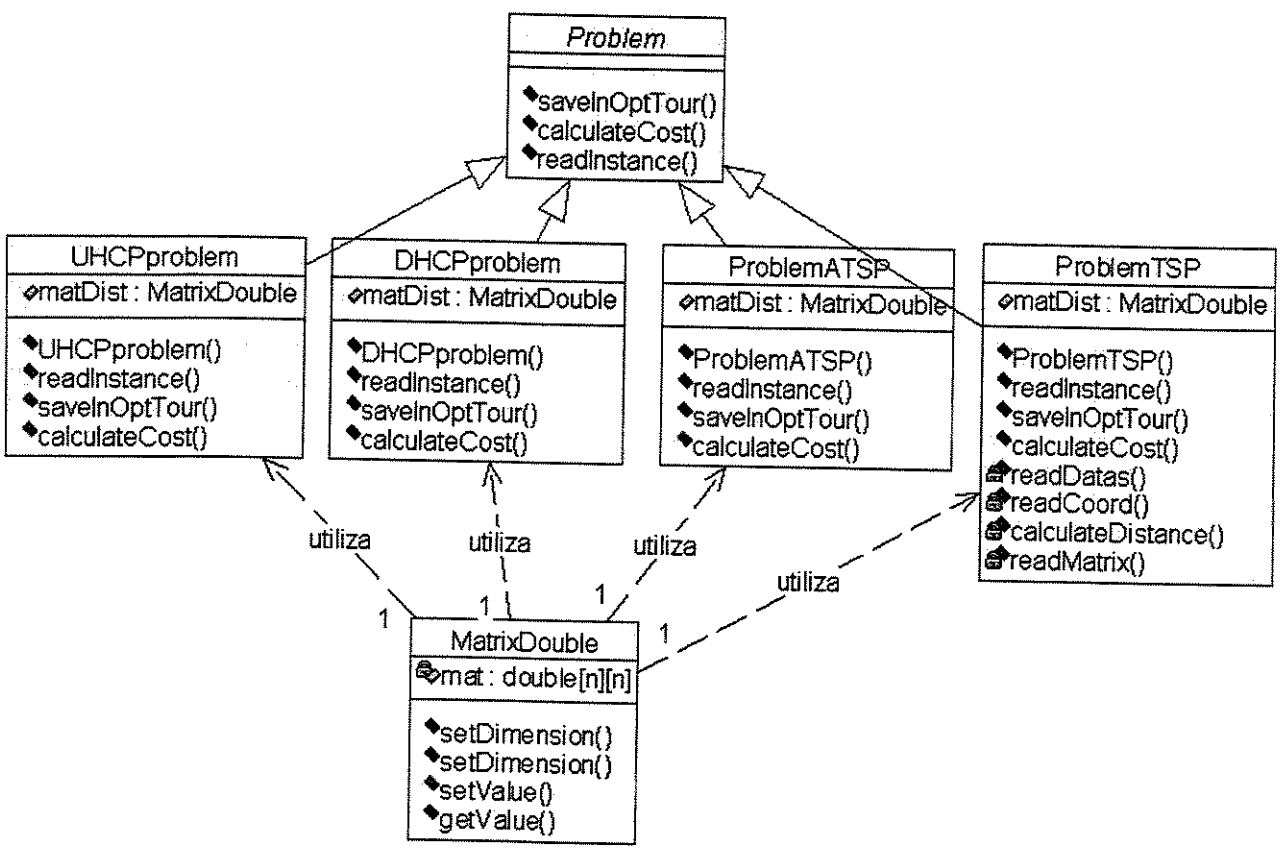


Figura 35 - Seleção de um problema.

As subclasses de *Problem* possuem, além do construtor, a redefinição dos métodos da superclasse. A classe *ProblemTSP* possui alguns métodos auxiliares do método *readInstance( )* e por isso são privados. O método *readDatas( )* verifica se os dados estão armazenados por coordenadas ou por matriz; *readCoord( )* faz a leitura das coordenadas, caso os dados estejam armazenados por coordenadas); *readMatrix( )* faz a leitura da matriz de distâncias, caso os dados estejam armazenados em uma matriz, e *calculateDistance( )* calcula a matriz de distâncias pela função indicada no cabeçalho do arquivo, no caso dos dados estarem por coordenadas.

6.9. REDUÇÕES

Da mesma forma e com o mesmo objetivo de como é utilizado para os operadores e seleção de problema, o *upcasting* é utilizado para redefinir os métodos das classes filhas da classe *Reductions*.

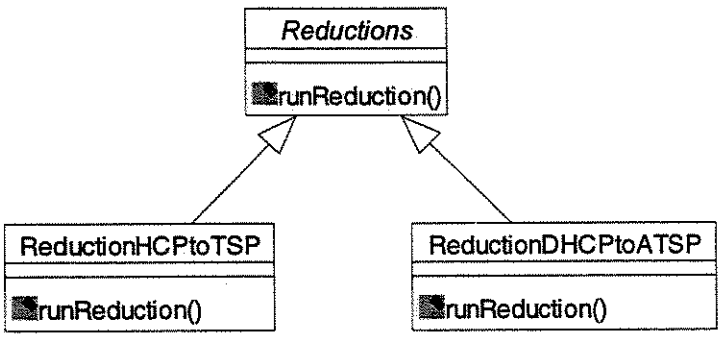


Figura 36 - Seleção da redução.

A inclusão de novas reduções é feita de forma simples. Uma classe para a nova redução é feita e esta é herdada da classe *Reductions*, e o método *runReduction( )* é redefinido.

6.10. CONSIDERAÇÕES FINAIS E CONCLUSÕES

A programação foi desenvolvida de forma a explorar, tanto quanto possível, as características de orientação a objetos. O encapsulamento e ocultamento de informação tiveram atenção especial, pois são as características de um programa orientado a objetos que mais contribuem para tornar os módulos mais independentes. As únicas classes que possuem atributos públicos são: *Problem*, *Population*, *PocCurAgent*, *DiCycle* e *Arc*. Somente a variável *dimension*, localizada na classe *Problem*, é definida global (*static*). Polimorfismo é amplamente utilizado, permitindo a possibilidade de *upcasting* entre referências de 6 classes. Com *upcasting*, o programa fica melhor estruturado.

A adição de novos operadores, bem como novos problemas e reduções, é feita de forma simples e fácil. A adição de uma nova representação de solução, embora seja feita facilmente, recebe um tratamento diferenciado. Como na implementação atual a classe *Memepool* acessa operadores representados por instâncias da classe *DiCycle*, uma outra representação qualquer acarretaria a mudança da classe *Memepool*. Essa manteria seu fluxo normal, mas as instâncias dos operadores passariam a ser alocadas para operadores da nova representação, e não para *DiCycle*, como se encontra atualmente.

Isso acontece porque a herança permite polimorfismo, ou seja, os métodos nas classes filhas são redefinidos. Mas em se tratando de uma nova representação, os parâmetros dos métodos seriam de tipos diferentes. Por exemplo, o método *runCrossover( )* possui como parâmetros três rotas representadas em *DiCycle*: *parentA*, *parentB* e *child*. Se uma nova representação, por exemplo *HashTable*, fosse usada, os parâmetros não seriam mais do tipo *DiCycle*, e sim do tipo *HashTable*, não permitindo ser possível redefinir os métodos nas classes filhas.

Para resolver esse problema seria necessário um mecanismo que tornasse possível utilizar um tipo genérico de dados e fazer um *cast* para o tipo selecionado pelo usuário. Esse *cast* seria feito dentro do operador (mutação, recombinação, etc.) para que o operador considerasse o tipo em que é implementando.

Esse fato acontece também com as classes *Problems* e *Reductions*. Para os problemas até então disponíveis no *framework*, os tipos passados como parâmetros para os métodos dessas classes podem ser os mesmos. Mas na medida que novos problemas forem sendo incluídos, haverá necessidade de tratar com tipos genéricos de dados nos métodos redefinidos pelas classes filhas.

Com essa alteração, o diagrama central do *framework* seria:

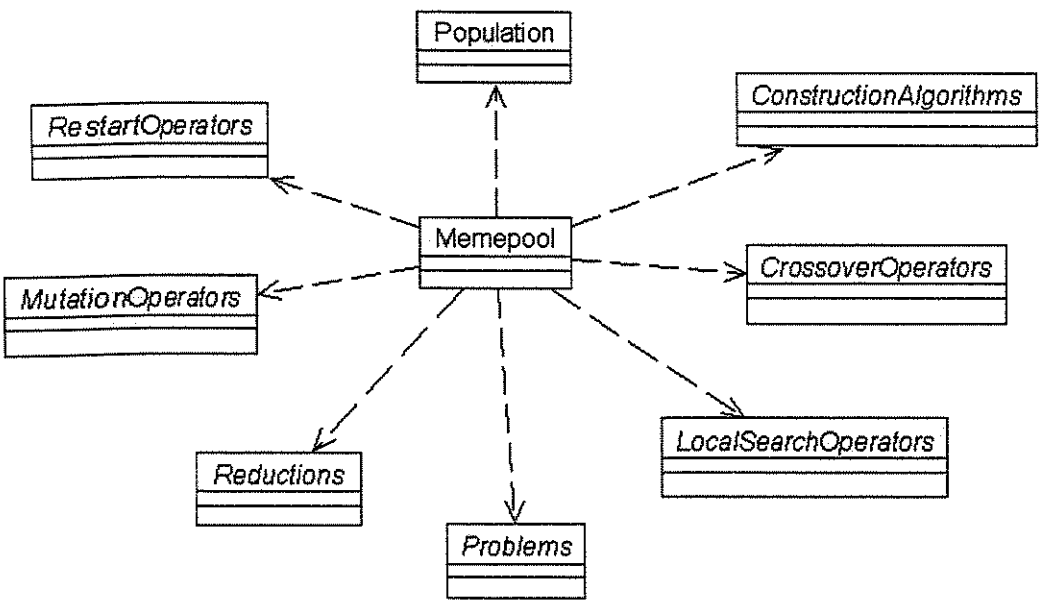


Figura 37 - Diagrama central alternativo para o *framework*.

Esse problema não é simples e certamente requer amplo conhecimento da linguagem e experiência em programação orientada a objetos.

Ainda, a criação de uma classe para controle da seleção dos indivíduos que farão parte da fase de recombinação também é necessária. Dessa forma, o método *recombinePocketsWithCurrentes*, atualmente na classe *Memepool*, estará disponível somente se a população estiver organizada em agentes formados por *pockets* e *currents*. Ainda, diferentes políticas de seleção poderão estar disponíveis, o que requer uma classe para melhor estruturação do código.

No caso dessa implementação, o algoritmo está limitado à seleção de somente um operador de cada especialidade (recombinação, busca local, mutação e *restart*), assim como somente um algoritmo construtivo. Essa limitação poderia não existir, pois o usuários pode

mesclar tai métodos que tenham o mesmo objetivo, inclusive com taxas diferentes de utilização.

O *JavaDoc* foi utilizado para documentar o *framework*. A página *html* contendo tal documentação pode ser acessada a partir de *http://www.densis.fee.unicamp.br/~Memepool.html*.

A convenção sugerida para dar nomes em Java é seguida, a fim de tornar mais fácil o entendimento dos diagramas e do código.

Capítulo

7

CONCLUSÕES E SUGESTÕES PARA TRABALHOS FUTUROS

O trabalho de tese reuniu duas áreas bem diferentes, mas que se beneficiam mutuamente quando combinadas: engenharia de *software* e otimização. Com isso, os algoritmos meméticos encontraram uma ferramenta que gerenciasse seu fluxo de execução e os *frameworks* encontraram mais uma aplicação para seu conceito. A programação orientada a objetos realizou este "casamento", tornando possível a estruturação adequada do *framework*, adicionando a este todas as características provenientes do seu uso.

O algoritmo memético desenvolvido introduziu, além do novo operador de busca local RAI, uma série de artifícios novos, tais como: população estruturada em árvore ternária completa e agentes formados por duas rotas, onde uma funciona como uma memória associativa e a outra como indivíduo corrente da população. A robustez da busca local foi comprovada, pois 4 operadores de recombinação são testados e, para todos, os resultados apresentados pelo algoritmo obteve boa qualidade de solução num tempo pequeno. Ainda, os resultados comprovaram que uma população estruturada apresenta vantagens sobre a não estruturada, pois, para todas as instâncias testadas, o tempo computacional da última aumentou. A qualidade de solução também teve um piora quando a população não é estruturada, mas para as 5 instâncias de `ftv` consideradas difíceis para o algoritmo, a média da qualidade de solução passou de 0,352% para 0,326%. Quando comparados com os resultados obtidos por outros pesquisadores, a maior diferença de performance entre o RAI/SAX e os outros algoritmos pode ser observada para os maiores problemas de ATSP da TSPLIB. No caso do maior problema, `rbg443`, a média do tempo total gasto pelo MA proposto, em 30 execuções com cada operador de recombinação

implementado, obtendo o solução ótima para as 120 execuções, foi de 6,76 seg. O melhor tempo apresentado por outros pesquisadores é de 268 seg. (4,47 minutos), também encontrando sempre a otimalidade em 30 execuções, por Nagata e Kobayashi (1997). Ou seja, a diferença de tempo é de 40 vezes.

Os resultados encontrados pelo algoritmo quando aplicado às instâncias assimétricas de TSP da TSPLIB não foram tão bons como os resultados encontrados para instâncias de mesma dimensão originadas da redução de HCP para TSP e para instâncias assimétricas de ATSP geradas a partir de instâncias simétricas de TSP.

O algoritmo memético também foi aplicado a instâncias assimétricas geradas a partir de instâncias simétricas da TSPLIB. Os testes foram executados considerando 32 instâncias com dimensões entre 16 e 2.392 cidades, que correspondem às 32 instâncias de TSP simétrico com rota ótima disponível na TSPLIB. Os resultados computacionais demonstraram que a boa performance do algoritmo, considerando instâncias de até 100 cidades, foi a mesma apresentada anteriormente, enquanto que para instâncias maiores, a qualidade de solução decresce. Como média para as 32 instâncias, considerando os dois valores de α_{max} testados, o desvio percentual médio da solução inicial foi de 102,15%, enquanto que o da solução final foi de 1,37%. Para as 30 execuções com cada instância testada, a solução ótima foi encontrada em 67% das execuções e o tempo médio gasto foi de 151,82 seg.

A análise da performance do algoritmo aplicada à redução de UHCP para STSP também foi realizada. Para os testes foram consideradas 20 instâncias de dimensões entre 20 e 2.000 cidades. Para instâncias de até 300 cidades, a solução ótima foi sempre encontrada, num tempo computacional médio de 3,96 seg. Para problemas com mais de 400 cidades, a solução ótima não foi encontrada em qualquer execução. Em média, a solução ótima foi encontrada em 48% das execuções, a qualidade de solução final foi de 0,8% em um tempo médio de 408,63 seg.

Dois geradores de instâncias de problemas foram implementados: um para instâncias de ATSP e outro para instâncias de HCP. Os dois geradores foram implementados em Java e por isso são portáveis. Para o gerador de instâncias de ATSP, a matriz de distâncias é obtida a partir da instância de TSP, com a diferença que, aos valores da matriz triangular inferior que não pertencem a rota ótima, é adicionado um valor α aleatoriamente gerado entre $[0, \alpha_{max}]$. O valor de α_{max} é obtido como o valor inteiro do produto da porcentagem ρ pela média das distâncias da

matriz triangular inferior. Já o gerador de HCP produz uma instância que sempre possui pelo menos um ciclo hamiltoniano. Além do ciclo hamiltoniano, são adicionados k arcos aleatoriamente selecionados, para cada vértice i. O número k é um valor aleatório entre *minEdges* e *maxEdges*, informados pelo usuário. Os geradores estão disponíveis nos endereços *http://www.densis.fee.unicamp.br/~buriol/ATSPGenerator.html* e *http://www.densis.fee.unicamp.br/~buriol/HCPGenerator.html*.

7.1. PRINCIPAIS CONTRIBUIÇÕES

Como síntese das contribuições que o trabalho de tese possibilitou, pode-se citar:

1) implementação de um algoritmo memético implementado em Java para ATSP, juntamente com os resultados computacionais obtidos quando aplicado às 27 instâncias disponíveis na TSPLIB, bem como a análise e comparação com resultados encontrados por outros pesquisadores e desempenho do algoritmo com população estruturada e não-estruturada.

2) descrição de uma nova busca local para o ATSP: *Recursive Arc Insertion* (RAI);

3) criação do operador de recombinação SAX através da assimetrização do SEX;

4) Implementação de um gerador portável de instâncias de ATSP com solução ótima conhecida;

5) implementação de um gerador portável de instâncias de HCP com solução ótima conhecida (problema de decisão);

6) implementação da redução do problema do problema de ciclos hamiltonianos não direcionados para o problema do caixeiro viajante simétrico, bem como os testes computacionais obtidos, utilizando o MA desenvolvido;

7) implementação e documentação do *framework* orientado a objetos para algoritmos evolutivos.

7.2. PUBLICAÇÕES E APRESENTAÇÕES RELACIONADAS À TESE

Como os resultados da redução de UHCP para STSP foram recentemente obtidos, ainda não constam em nenhum trabalho apresentado.

Palestras:

Mendes, Alexandre, Felipe Müller, Luciana Buriol, Paulo França e Pablo Moscato. *O projeto MEMEPOOL: um Framework para Otimização Combinatória.* VI ELAVIO - Escola Latino-Americana de Verão de Pesquisa Operacional, Mendes/RJ - 11 a 15 de janeiro de 1999;

Buriol, Luciana, Paulo França e Pablo Moscato. *Uma implementação em Java para resolução de um Algorítmo Memético para o Problema do Caixeiro Viajante Assimétrico.* II Semana Acadêmica do Curso de Informática, Universidade Federal de Santa Maria, Santa Maria/RS - 22 a 26 de novembro de 1999.

Resumo e palestra:

Mendes, Alexandre, Felipe Müller, Luciana Buriol, Paulo França e Pablo Moscato. *O projeto MEMEPOOL: um Framework para Otimização Combinatória.* XXX SOBRAPO - Simpósio Brasileiro de Pesquisa Operacional, Curitiba, PR, Brazil, November, 1998, pgs. 20-21.

Buriol, Luciana S., Paulo M. França, e Pablo Moscato. (1999). *New heuristic results for the asymmetric traveling salesman problem.* Resumo aceito para o 4th International Conference on Operations Research Optimization. Havana, March 6-10, 2000. Organized by Universidad de La Habana e Humboldt-Universität zu Berlin.

Artigo e palestra:

Buriol, Luciana, Paulo França e Pablo Moscato. *Algoritmo Memético para Resolução do Problema do Caixeiro Viajante Assimétrico.* I Oficina de Planejamento e Controle da Produção em Sistemas de Manufatura, Campinas/SP - 15 e 16 de abril de 1999, pgs. 37-41.

Buriol, Luciana, Paulo França e Pablo Moscato. *Algoritmo Memético para o problema do caixeiro viajante.* XXXI SOBRAPO - Simpósio Brasileiro de Pesquisa Operacional, Juiz de Fora, MG, Brazil, 20-22 October, 1999.

Artigo submetido para revista

Buriol, Luciana S., Paulo M. França, e Pablo Moscato. (1999). *Recursive Arc Insertion: A New Local Search Embedded in a Memetic Algorithm for the Asymmetric Traveling Salesman Problem.* Artigo submetido (outubro/1999) para a revista Journal of Heuristics, 34 pgs.

7.3 SUGESTÕES PARA TRABALHOS FUTUROS

Pelo fato da população utilizada para o MA implementado ser muito pequena (13 indivíduos), poder-se-iam gerar as soluções iniciais por algum procedimento mais robusto, em vez da heurística construtiva do "vizinho mais próximo". Para as instâncias assimétricas da TSPLIB, o desvio percentual da qualidade da solução inicial é de 57,07% (Tabela 2), enquanto que Glover et al. (1999) descreve e testa 6 novos algoritmos construtivos, com qualidade de solução inicial de 3,36%, 4,29%, 4,77%, 17,36%, 18,02% e 30,62%, considerando 26 das 27 instâncias assimétricas da TSPLIB (com exceção da instância `ftv90`).

Embora os resultados do MA utilizando RAI foram bem sucedidos, este não foi testado como uma busca local pura (sem estar inserido num MA). Desta forma, o operador RAI poderia ser adaptado e testado como uma busca local pura para o TSP ou problemas de seqüenciamento.

Como a população é muito pequena, a possibilidade de perda de diversidade aumenta e o a população pode ficar estagnada. O operador de *restart( )* tem o papel de devolver a diversidade à população, mas sem que para isso, um procedimento muito agressivo seja usado. Por agressivo subentende-se aquele que age de forma muito intensa, como por exemplo, excluir muitas soluções da população, fazer modificações substanciais na maior parte das soluções, etc. Quando o procedimento é muito agressivo, a população ganha diversidade, mas a convergência da solução demora para se reestabelecer. O procedimento *restart( )* implementado não é considerado um procedimento agressivo, mas foi pouco explorado, de forma que outros operadores poderiam contribuir positivamente para a melhora da performance do algoritmo.

Como já mencionado no capítulo 5, seria muito importante que uma revisão bibliográfica do problema de ciclos hamiltonianos fosse desenvolvida. O objetivo é comparar os resultados da Tabela 7 com os resultados obtidos por algoritmos específicos para o problema e por reduções do problema para o STSP.

Outro ponto importante que pode ser modificado são as alterações sugeridas no capítulo 6, a fim de obter o diagrama de classes da Figura 37.

O *framework* não dispõe de uma interface gráfica, sua implementação contribuiria muito para a utilização do mesmo.

Os artigos apresentados não compreendem a parte que diz respeito às reduções e a estruturação do *framework*. Seria interessante que pelo menos um artigo abordando cada assunto fosse escrito.

Os algoritmos meméticos são intrinsicamente paralelos. No caso do *framework* a paralelização poderia ser analisada de diversas formas. O paralelismo poderia ser obtido dentro de um algoritmo, considerando subpopulações ou operadores em processadores/máquinas diversas. Outro tipo de paralelismo menos granular seria executar instâncias ou problemas em processadores/máquinas diferentes. Este trabalho de paralelização pode ser colocado como sugestão para trabalhos futuros, mas certamente compreenderia uma tese de mestrado inteiramente dedicada ao assunto.

REFERÊNCIAS BIBLIOGRÁFICAS

Allender, Eric, Michael C. Loui e Kenneth W. Regan. (1998). "Reducibility and Completeness". In Mikhail J. Atallah (ed.), *Algorithms and Theory of Computation Handbook*, (chapter 28). CRC Press.

Amin, S. (1999). "Simulated Jumping." *Annals of Operations Research* 86, 23-38.

Balas, E. e P. Toth. (1985). "Branch-and-Bound Methods." In E. L. Lawler, J.K. Lentra, A. H. G. Rinnoy Kan e D. B. Shmoys (eds.), *The Traveling Salesman Problem,* New York: John Wiley e Sons.

Bentley, J. L. (1992). "Fast Algorithms for Geometric Traveling Salesman Problems." *ORSA Journal on Computing* **4**, 387-411.

Berretta, R. e P. Moscato. (1999). "The Number Partitioning Problem: An Open Challenge for Evolutionary Computation." In D. Corne, M. Dorigo e F. Glover (eds.), *New Ideas in Optimization*, (chapter 17). Washington: McGraw-Hill.

Boock, Grady, James Rumbaugh e Ivar Jacobson. (1999). *The Unified Modeling Language User Guide: the ultimate tutorial to the UML from the original designers*. Massachusetts: Addison-Wesley.

Carpaneto, G., M. Dell´Amico e P. Toth. (1995). "Exact Solution of Large-Scale, Asymmetric Traveling Salesman Problems." *ACM Transactions on Mathematical Software* **21** (4), 394-409.

Cormen, Thomas H., Charles E. Leiserson e Ronald L. Rivest. (1996). *Introduction to Algorithms.*" 7ª edição, Massachusetts: McGraw-Hill.

Costa, João, Nuno Lopes, Pedro Silva e Agostinho Cláudio da Rosa. (1999). *JDEAL: The Java Distributed Evolutionary Algorithms Library, (http://laseeb.ist.utl.pt/sw/jdeal/).*

Deitel, H. M e P. J. Deitel. (1998). *Java: how to program.* 2nd ed., New Jersey: Prentice-Hall.

Feo, T, Mauricio Rezende. (1994). "A greed randomized adaptative search procedure for

maximum independent set." *Operations Research* **42**, 860-879.

Fiechter, C. N. (1994). "A Parallel Tabu Search Algorithm for Large Traveling Salesman Problems." *Discrete Applied Mathematics* **51**, 243-267.

Fischetti, M. e P. Toth. (1997). "A Polyhedral Approach to the Asymmetric Traveling Salesman Problem." *Management Science* **43** (11), 1520-1536.

Fischetti, M., P. Toth e D. Vigo. (1994). "A Branch-and-Bound Algorithm for the Capacitated Vehicle Routing Problem on Directed Graphs." *Operations Research* 42, 846-859.

Fogel, David B. (1995). *Evolutionary Computation : Towards a New Philosophy of Machine Intelligence*. IEEE Press, 1st. ed. *http://www.natural-selection.com/*

França, P. M., A. S. Mendes e P. Moscato. (1999). "Memetic Algorithms to Minimize Tardiness on a Single Machine with Sequence-Dependent Setup Times." In *Proceedings of the 5th International Conference of the Decision Sciences Institute*, Athens, Greece, 1708-1710.

Freisleben, B. e P. Merz. (1996). "A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problems." In *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, Nagoya, Japan, 616-621.

Gamma, Erich, Richard Helm, Ralph Johnson e John Vlissides. (1995). *Desing Patterns: elements of reusable object-oriented software*. Addison-Wesley Publizhing Company.

Garey, M.R. e D. S. Jonhson. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: Freeman.

Glover, F. (1977). "Heuristics for Integer Programming Using Surrogate Constraints." *Decision Sciences* **8** (1), 156-166.

Glover, F. e M. Laguna. (1993). "Tabu Search". Colin Reeves (ed.), em *Modern Heuristic Techniques*. Blackwell Scientific Publications, Oxford, Blackwell, 70-150.

Glover, F., G. Gutin, A. Yeo e A. Zverovich. (1999). "Construction Heuristics and Domination Analysis for the Asymmetric TSP." In J. S. Vitter e C. D. Zaroliagis (eds.), *Proceedings of the Algorithm Engineering: 3rd International Workshop*, London, UK, *Lecture Notes in Computer Science* **1668**, 85-94, Springer.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning.*

Addison-Wesley.

Gorges-Schleuter, M. (1989). "ASPARAGOS: An Asynchronous Parallel Genetic Optimization Strategy." In J. D. Schaffer (ed.), *Proceedings of the Third International Conference on Genetic Algorithms*, 422-427.

Gorges-Schleuter, M. (1997). "Asparagos96 and the Traveling Salesman Problem." In T. Baeck, Z. Michalewicz e X. Yao (eds.), *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation* , Indianapolis, USA, 171-174.

Hogg, T. e C. P. Williams. (1993). "Solving the Rally Hard Problems with Cooperative Search." *In Proceedings of AAAI93*, Menlo Park, EUA, 231-236.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. The University of Michigan Press, 1st ed. (The MIT Press, 2nd. ed. 1992).

Holstein, D. e P. Moscato. (1999). "Memetic Algorithms Using Guided Local Search: A Case Study." In . D. Corne, M. Dorigo e F. Glover (eds.), *New Ideas in Optimization,* (chapter 15). Washington: McGraw-Hill.

Horstmann, Cay S. e Gary Cornell. (1997). "Core Java 1.1", Califórnia: Sun Microsystem Press.

Jünger, M., G. Reinelt e G. Rinaldi. (1995). "The Traveling Salesman Problem." In M. Ball, T. Magnanti, C.L. Monma e G. L. Nemhauser (eds.), *Handbooks in Operations Research and Management Sciences: Networks.* North-Holland.

Kanellakis, P. C. e C. H. Papadimitriou. (1980). "Local Search for the Asymmetric Traveling Salesman Problem." *Operations Research* **28** (5), 1086-1099.

Karp, R. "Reducibility among combinatorial problems". (1972). In R. Miller e J. Thatcher, edts. *Complexity of Computer Computation.* New York: Plenum Press, 85-103.

Karp, R. M. (1977). "Probabilistic Analysis of Partitioning Algorithms for the Traveling Salesman Problem in the Plane." *Math. Operational Research* **2**, 209-224.

Karp, R. M. (1979). "A Patching Algorithm for the Nonsymmetric Traveling Salesman Problem." *SIAM J. Comput.* **8**, 561-573.

Kirkpatrick, S., C. D. Gelatt Jr., M. P. Vecchi. (1983). *Optimization by Simulated Annealing. Science* **220** (4598), 671-679.

Krasnogor, Natalio e Jim Smith. (1999). *A Java Framework for Memetic Alforithms*. A ser publicado. *http://www.ics.uwe.ac.uk/~natk/MAFRA/mafra.html.*

Laporte, G. (1992). "The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms." *European Journal of Operational Research* **59** (2), 231-247.

Lewis, Harry R. e Christos H. Papadimitriou. (1998). "Elements of the theory of computation". 2nd ed., Prentice-Hall, New Jersey.

Lin, S. e B. W. Kernighan. (1973). "An Effective Heuristic Algorithm for the Traveling Salesman Problem." *Operations Research* **21** (2), 498-516.

Mendes, A. S., Müller, F. M., França, P. M. e P. Moscato. (1999). "Comparing Metaheuristic Approaches for Parallel Machine Scheduling Problems with Sequence Dependent Setup Times." In M. F. Carvalho e F. M. Müller (eds.), *Proceedings of the 15th International Conference on CAD/CAM, Robotics & Factories of the Future*, Aguas de Lindoia, Brazil, 1-6.

Merz, P. e B. Freisleben. (1997). "Genetic Local Search for the TSP: New Results." In *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, Indianapolis, EUA, 159-164.

Miller, D. L. e J. F. Pekny. (1991). "Exact Solution of Large Asymmetric Traveling Salesman Problems." *Science* **251**, 754-761.

Moscato, P. (1989). "On Evolution, Search, Optimization, Genetic Algorithms, and Martial Arts: Towards Memetic Algorithms." *Technical Report, Caltech Concurrent Computation Program*, C3P Report 826.

Moscato, P. e M. G. Norman. (1992). "A Memetic Approach for the Traveling Salesman Problem. Implementation of a Computational Ecology for Combinatorial Optimization on Message-Passing Systems." In M. Valero, E. Onate, M. Jane, J. L. Larriba e B. Suarez (eds.), *Parallel Computing and Transputer Applications* 187-194. Amsterdam: IOS Press.

Moscato, P. e F. Tinetti. (1992). "Blending Heuristics with a Population-Based Approach: A Memetic Algorithm for the Traveling Salesman Problem." CeTAD, *Report 92-12*. Universidad Nacional de La Plata, Argentina.

Moscato, P. (1993). "An Introduction to Population Approaches for Optimization and Hierarchical Objective Functions: A Discussion on the Role of Tabu Search." *Annals of Operations Research* **41**, 85-121.

Moscato, P. (1999). "Memetic Algorithms: A Short Introduction." In D. Corne, M. Dorigo e F. Glover (eds.), *New Ideas in Optimization*. Washington: McGraw-Hill.

Nagata, Y. e S. Kobayashi. (1997). "Edge Assembly Crossover: A High-power Genetic Algorithm for the Traveling Salesman Problem." In *Proceedings of the Seventh International Conference on Genetic Algorithms*, East Lansing, EUA, 450-457.

Osman, Ibrahim. (1991). "Heuristics for Combinatorial Optimization Problems: Developments and New Directions". *In Proceedings of the first seminar on Information Technology and Applications*, Marfield Conference Centre, September.

Paechter, B., A. Cumming, M. G. Norman e H. Luchian. (1996). "Extensions to a Memetic Timetabling System." In E. K. Burke e P. Ross (eds.), *The Practice and Theory of Automated Timetabling, Lecture Notes in Computer Science* **1153**, 251-265, Springer Verlag.

Potvin, J. Y. (1993). "The Traveling Salesman Problem: A Neural Network Perspective." *ORSA Journal on Computing* **5**, 328-348.

Reinelt, G. (1991). "TSPLIB - A Traveling Salesman Library." *ORSA Journal on Computing* **3**, 376-384.

Stützle, T. e M. Dorigo. (1999). "ACO Algorithms for the Traveling Salesman Problem." To appear in K. Miettinen, M. Mäkelä, P. Neittaanmäki e J. Periaux, (eds), *Evolutionary Algorithms in Engineering and Computer Science: Recent Advances in Genetic Algorithms, Evolution Strategies, Evolutionary Programming, Genetic Programming and Industrial Applications,* John Wiley & Sons.

Syswerda G. (1989). "Uniform crossover in genetic algorithms." In *Proceedings of the Third International Conference on Genetic Algorithms*. In Schaffer J. D. (ed.), Morgan Kaufmann, San Mateo CA, 2-9.

Szypersky, Clemens. (1998). *Component Software beyond object-oriented programming.* Addison-Wesley Publizhing Company.

Vandegriend, Basil. (1998). *Finding Hamiltonian Cycles: Algorithms, Graphs and Performance.* Tese defendida na Faculty of Graduate Studies and Research, Department of Computing Science, University of Alberta, Alberta, Edmonton, Canadá. (*http://web.cs.ualberta.ca/~joe/Theses/vandegriend.html*).

Vogel, Andreas e Keith Duddy. (1997). *Java Programming with CORBA.* John Wiley & Sons.

Walters, T. (1998). "Repair and Brood Selection in the Traveling Salesman Problem." In A. E. Eiben, T. Bäch, M. Schoenauer, H. P. Schwefel (eds.), *Proceedings of the Fifth International Conference on Parallel Problem Solving from Nature*, Amsterdam, The Netherlands, 813-822.

Whitley, D., T. Starkweather e D. Shaner. (1991). "The Traveling Salesman and Sequence Scheduling: Quality Solutions Using Genetic Edge Recombination." In L. Davis (ed.), *Handbook of Genetic Algorithms*, New York, 350-372.

Apêndice

1

CONCEITOS DE ORIENTAÇÃO A OBJETOS: UMA VISÃO EM JAVA E UML

A programação orientada a objetos tem sua principal aplicabilidade em sistemas grandes e complexos. O uso dessa metodologia torna o sistema mais modular e mais confiável, permite reutilização de código, o desenvolvimento e custo de manutenção são reduzidos e facilita a organização de sistemas complexos. Uma linguagem para programação orientada a objetos é composta por um conjunto de regras e convenções que, quanto melhor utilizadas, mais independentes e simples se tornam as partes que compõem o sistema. Uma estruturação "simples" não significa "fácil". Pelo contrário, quanto mais experiência o programador tiver no assunto, mais simples se torna a estruturação. Várias são as linguagens orientadas a objetos: Java, C++, Eiffel, CLOS, Smalltalk-80. Java é a mais recente e a única que permite programação para a *Internet*, além de prover portabilidade, segurança e facilidade de manipulação.

Embora a linguagem de implementação possa ser considerada uma decisão secundária de projeto por alguns pesquisadores, a utilização da linguagem Java foi uma decisão fudamentada e muito importante para o trabalho. Com o propósito de justificar sua escolha, uma breve descrição e as principais vantagens de sua utilização são sumarizadas na seção 6.1

Para apresentar visualmente a organização do sistema, dispõe-se de uma linguagem para modelagem unificada (*Unified Modelling Language* - UML). UML descreve um padrão de representação para a modelagem do sistema.

Este apêndice objetiva introduzir os conceitos de orientação a objetos, bem como sua utilização em Java e representação em UML, para que o leitor possa entender os diagramas de classes do *Framework*, apresentados no capítulo 6.

1.1. A LINGUAGEM DE PROGRAMAÇÃO JAVA

Os programas escritos em Java, quando compilados, não são convertidos para o conjunto de instruções de nenhum processador específico. Eles são traduzidos para um formato intermediário e universal, chamado *bytecode*. Quando um compilador Java traduz um arquivo fonte de classe para *bytecodes*, o arquivo que contém os *bytecodes* da classe pode ser executado em qualquer máquina que possua um Máquina Virtual Java (*Java Virtual Machine* - JVM[3]). Isto permite que o código em Java possa ser escrito independente da plataforma onde é executado. Para uma arquitetura cliente/servidor, essa facilidade elimina os ciclos de compilar e rodar no cliente, porque os *bytecodes* não são específicos de uma máquina, mas interpretados (Vogel e Duddy, 1997).

A partir da versão do JDK (*Java Development Kit*) 1.1.6, o compilador passou a gerar código para a arquitetura da máquina onde é compilado, além da opção da geração dos *bytecodes*. Com isso ganha-se em velocidade, mas perde-se a portabilidade. É o usuário que define qual a opção que melhor se adequa à sua aplicação. Cabe salientar que o código é exatamente o mesmo, a opção de compilação é que vai definir se será portável ou não.

A linguagem Java foi criada para desenvolvimento de programas em ambientes heterogêneos ligados em rede. Desde que o objetivo inicial da linguagem era ser utilizada em sistemas isolados, com quantidade mínima de memória, a linguagem Java foi projetada para ser pequena e utilizar uma quantidade reduzida de recursos do sistema. A execução de seu código pode ser por *applets* ou por aplicação.

Aplicações Java são programas autônomos, que podem ser interpretados diretamente (pelo interpretador Java) a partir de um método *main( )*. Este método é o ponto de início de execução de qualquer aplicação Java. Ao contrário do que acontece em C e C++, toda e qualquer classe pode ter um método *main( )* definido.

Numa execução por *applet*, ao acessar a página HTML com um *tag APPLET*, o *browser* carrega todos os arquivos de classe relacionados ao *applet*, e também outros possíveis arquivos utilizados por este. O *browser* também aloca espaço para o *applet* na página e disponibiliza os

---

[3] JVM é uma máquina de computação abstrata e um ambiente de execução independente de plataforma.

parâmetros do HTML. Os *applets* podem ser de dois tipos: *applets* carregados via rede e *applets* carregados via sistema de arquivos (Vogel e Duddy, 1997).

Para garantir a segurança nas máquinas e evitar que *applets* realizem operações não desejadas (apagar os arquivos da máquina, enviar *mails*, etc.), os *applets* carregados via rede devem rodar em um ambiente limitado. Por isso, os *browsers* e visualizadores de *applets* cuidadosamente restringem as ações que estes podem executar. Diferentes navegadores e visualizadores de *applets* podem colocar diferentes restrições, mais ou menos rígidas, mas, em geral, *applets* não podem: carregar bibliotecas do sistema operacional nem definir métodos nativos, ler ou gravar arquivos do *host* onde executam, executar qualquer programa no *host* onde estão carregados, fazer conexões de rede (por exemplo, usando *sockets*), exceto com o *host* origem, ler variáveis de ambiente (por exemplo, o *path* ou o nome do usuário), definir propriedades do sistema.

Todo o código Java recebido a partir da rede passa por um processo de verificação do seu *bytecode*. Este processo garante que o código carregado não viole restrições de espaço de endereçamento ou restrições de conversão de tipos. Pode também verificar se o código é realmente código da JVM, se não converte tipos de dados de forma ilegal, se não estoura a pilha de execução e se não usa registradores incorretamente. O propósito desta verificação é garantir que o código carregado não forje ponteiros ou faça aritmética de memória que poderia dar acesso privilegiado à máquina onde executa.

Outra vantagem da programação Java é que o programador não precisa se preocupar com a remoção explícita de objetos. Ao alocar um objeto, uma quantidade de memória é destinada a armazenar as informações pertinentes ao mesmo. Para desalocá-lo, duas abordagens possíveis são: dedicar essa tarefa ao programador ou deixar que o sistema seja o responsável por esta retomada de recursos. O problema da primeira abordagem é que o programador, erroneamente, pode desalocar uma posição de memória indevida ou esquecer de desalocar uma área previamente alocada que não mais seria necessária para o restante da execução do programa. Na segunda abordagem, recursos adicionais do sistema são necessários, pois este deve identificar quando uma posição de memória não mais será referenciada no restante da execução. Java segue a segunda opção, pois possui um *garbage collector* que verifica os objetos que não têm nenhuma referência válida, retomando o espaço dispensado para cada um desses objetos

A linguagem C++ foi desenvolvida a partir de C e, para que se difundisse rapidamente, a compatibilidade com os progrmas escritos em C foi mantida. Como C não é orientada a objetos, C++ fez concessões na implementação da tecnologia. Porém, apesar dos desenvolvedores da linguagem Java terem se baseado na linguagem de programação C++, ela não foi criada com a necessidade de ser compatível com nada já existente, e por isso apresenta melhores resultados do que C++ na implementação orientada a objetos. Em Java foram removidas muitas das características de orientação a objeto que são raramente usadas, ou são usadas parcialmente, tornando-a mais clara e mais simples do que as outras linguagens. Uma fonte de *bugs* em programas desenvolvidos em C++ é o gerenciamento de memória; em Java ele foi aperfeiçoado e facilitado.

A segurança é essencial em programas disponibilizados na *Internet*. Por exemplo, quando o usuário executa em seu computador um programa que veio via *Internet*, não há garantia de que aquele programa não tenha vírus embutido, capaz de apagar arquivos ou danificar programas já gravados no disco rígido local. Java eliminou alguns recursos de C++ (por exemplo, ponteiros) que facilitavam a confecção de programas de vírus.

Ao contrário de C++, a orientação a objetos em Java tem poucas responsabilidade com gerenciamento de memória, não tem ponteiros, menos confusão de sintaxe, e possui formas simples de resolução de métodos.

O componente básico para o desenvolvimento de *software* em Java é o *Java Software Development Kit* - Java SDK (nas versões anteriores à 1.2, o *Kit* era chamado de *Java Development Kit* - JDK). Este *software*[4] encontra-se disponível via *Internet*.

Para uma melhor análise de Java para a *Internet* consultar Vogel e Duddy (1997) enquanto que a programação básica pode ser encontrada, escrita de forma clara e de fácil entendimento, em Horstmann e Cornell (1997) e Deitel & Deitel (1998).

Como pode-se concluir, Java é uma linguagem de programação que reúne uma série de características desejáveis. Pode-se destacar: portabilidade, programação orientada a objetos, possui *garbage collection*, permite programação via *Internet*, segurança, etc. Além de tudo, C++ é uma linguagem muito conhecida entre os programadores. Como Java e C++ têm afinidades, é fácil migrar para a nova linguagem. Por todos estes motivos, Java foi a linguagem escolhida para implementação do *framework* descrito no Capítulo 6.

---

[4] http://java.sun.com/jdk/

1.2. UML - UMA LINGUAGEM PARA MODELAGEM DE SISTEMAS

Especificação é a descrição do sistema feita de forma rigorosa, consistente e completa, utilizando uma linguagem adequada. Uma especificação é um modelo abstrato de um *software*, *hardware* ou de qualquer produto. Quando a especificação utiliza uma linguagem de sintaxe formal, mas cuja semântica é definida de maneira informal, esta linguagem é dita semi-formal. UML é um exemplo de linguagem semi-formal.

UML é o acrônimo para *Unified Modelling Language* (Linguagem de Modelagem Unificada) proposta pelo OMG (*Object Management Group*) para ser um padrão para modelagem[5] de sistemas de *software*. As pesquisas para a criação desta linguagem iniciaram em outubro de 1994, sendo que a versão de UML 1.0 somente foi lançada em janeiro de 1997, sob a autoria de Grady Booch (*Rational Software Corporation*), Ivar Jacobson (*Objectory*) e James Rumbaugh (*General Eletronic*). Com a UML podemos especificar, visualizar, construir e documentar sistemas de *software* utilizando diversos diagramas. Uma descrição completa e concisa da linguagem pode ser encontrada em Boock(1999).

Além da representação em UML dos principais conceitos de orientação a objetos que serão descritos na próxima seção, UML possui outras representações, tais como nota e multiplicidade.

Uma nota é um símbolo para agregar um comentário e/ou anotações sobre um elemento ou um conjunto de elementos. Em UML, uma nota é representada por um retângulo com a ponta superior direita dobrada, contendo o comentário no seu interior.

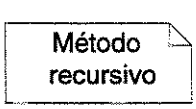


Figura 38 – Uma nota em UML.

Multiplicidade é um número ou uma faixa de valores que podem estar associados a uma classe ou a um atributo. Multiplicidade indica o número de cópias de instâncias (quando associado a uma classe) ou de atributos (se associado com algum atributo).

Quando associado a uma classe, a multiplicidade pode estar indicada nos relacionamentos dos diagramas de classes (serão vistos na seção 1.3.11), ou indicada no canto superior direito da classe. Quando associada ao atributo, encontra-se entre colchetes, após o identificador do mesmo.

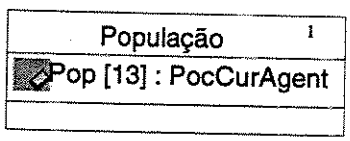


Figura 39 – Exemplo de multiplicidade de classe e atributo em UML.

Um diagrama de classes permite visualizar o relacionamento entre um conjunto de classes, interfaces, bem como seus métodos e atributos. Uma modelagem pode ser composta por diversos outros diagramas (Booch et al., 1999), mas o diagrama de classes é o mais encontrando em modelagens de sistemas orientados a objetos. O diagrama de classes descreve um estado estático do sistema.

1.3. CONCEITOS DE ORIENTAÇÃO A OBJETOS

A programação de sistemas utilizando o paradigma de orientação a objetos facilita a organização de sistemas complexos, pois promove uma melhor estruturação de seus componentes, beneficiando a reutilização de código, bem como sua manutenção. As principais vantagens de sua utilização são:

a) reutilização de código;

b) desenvolvimento e custo de manutenção reduzidos;

c) maior confiabilidade;

d) facilidade de lidar com programas complexos.

Nas seções a seguir serão descritos os principais conceitos de orientação a objetos, bem como sua utilização em Java e a modelagem UML correspondente.

---

[5] O termo modelagem é normalmente utilizado quando a especificação é feita utilizando uma notação gráfica.

1.3.1. Classes

Classe é o conceito mais amplo da programação orientada a objetos, representando uma abstração de um elemento do mundo real. O termo abstração é usado para indicar que tal elemento será representado apenas pelas suas características relevantes em relação ao que se deseja na modelagem. Através da definição de uma classe, descreve-se que propriedades e atributos os objetos dessa classe terão.

Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento dos objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe. Essas funcionalidades são descritas através de métodos. Um método nada mais é que o equivalente a um procedimento ou função, com a restrição que ele manipula apenas as variáveis locais e os atributos definidos para a classe.

A especificação de uma classe é composta por três regiões: nome da classe (identificador), atributos e métodos. Em Java, classes são definidas através do uso da palavra-chave *class*. Para definir uma classe, utiliza-se a construção:

```
class nomeDaClasse {
    // corpo da classe...
}
```

Após a palavra-chave *class*, segue o nome da classe, que deve ser um identificador válido para a linguagem. A definição da classe propriamente dita está entre as chaves { e }, que delimitam blocos na linguagem Java. No corpo da classe são definidos os atributos e métodos, podendo incluir a implementação destes. Tipicamente, uma classe é definida em um arquivo que tem o mesmo nome da classe, com a extensão *.java* como sufixo. Quando a classe for pública, essa regra não tem exceções.

A classe *java.lang.Object* é a raiz a partir da qual todas as classes são definidas em Java. Desse modo, suas definições estão disponíveis para objetos de todas as demais classes. Por exemplo, o método *equals( )*, que permite comparar objetos por seus conteúdos, e o método *clone( )*, que permite criar duplicatas de um objeto, são métodos definidos nessa classe.

Em UML, uma classe é representada por uma caixa contendo três campos que indicam o nome, atributos e métodos.

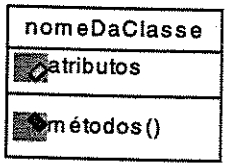


Figura 40 - Representação de uma classe em UML.

1.3.2. Métodos

Em Java todo método deve pertencer a uma classe. Para cada método, especifica-se sua assinatura, composta por:

- nome: um identificador para o método;
- tipo de retorno: quando o método tem um valor de retorno identifica-se o tipo desse valor. Caso não haja valor de retorno, o tipo de retorno é identificado como *void*;
- lista de argumentos: caso o método receba parâmetros para sua execução, para cada parâmetro deverá ser informado o tipo e um identificador;
- visibilidade (opcional): utilizando os identificadores de acesso (serão vistos na seção 1.3.5), define-se o quão visível é um método a partir de objetos de outras classes. Caso este não for definido, Java o considera público automaticamente.

A forma genérica para a definição de um método em uma classe é:

```
[modificador] tipo nomeDoMétodo(argumentos) {
    // corpo do método
}
```

O método construtor é um método especial, definido para cada classe. O corpo desse método determina as atividades associadas à inicialização de cada objeto criado. Assim, esse método é apenas invocado no momento da criação do objeto através do operador *new*. A assinatura do

método construtor é diferente das assinaturas dos outros métodos: não tem nenhum tipo de retorno (nem mesmo *void*) e o nome do método deve ser o próprio nome da classe.

Mais de um método construtor pode ser definido para uma classe e toda classe tem pelo menos um construtor. No caso de nenhum construtor ser explicitamente definido pelo programador, o compilador Java se encarrega de criar um.

Em UML um método pode ser apresentado de várias formas. Abaixo seguem alguns exemplos:

localSearchRAI	Somente nome
+ *localSearchRAI*	Visibilidade e nome
localSearchRAI(Ind: DiCycle, startCity: int)	Nome e parâmetros
LocalSearchRAI( ): DiCycle	Nome e tipo de retorno
LocalSearchRAI( ) {recursivo}	Nome e propriedade

Uma boa prática de programação é manter a funcionalidade de um método simples, desempenhando uma única tarefa. O nome do método deve refletir de modo adequado a tarefa realizada. Se a funcionalidade do método for simples, será fácil nomeá-lo.

1.3.3. Atributos

Os atributos são as variáveis que formam o conjunto de propriedades da classe ou do método. Para cada atributo, especifica-se:

- nome: um identificador para o atributo;
- tipo: o tipo do atributo (inteiro, real, caracter, etc.);
- valor inicial (opcional): pode-se especificar um valor inicial para o atributo;
- modificadores de acesso (opcional): pode-se especificar o quão acessível é um atributo de um objeto.

Em UML um atributo pode ser apresentado de diversas formas. Abaixo seguem alguns exemplos:

Pop	Somente nome
+ Pop	Visibilidade e nome
Pop: Agent	Nome e tipo
*tour: *Edges*	Nome e tipo complexo
Pop[13]: *Agent*	Nome, multiplicidade e tipo
bestSolution: double = 0.0	Nome, tipo e valor inicial

1.3.4. Objetos

Um objeto é a representação de um único elemento do mundo real, criado de acordo com o molde definido pela classe. Estritamente falando, ao se definir uma classe, define-se na realidade um tipo de dados, e a criação de um objeto corresponde à declaração de uma variável do tipo definido pela classe. Os objetos são também chamados de instâncias.

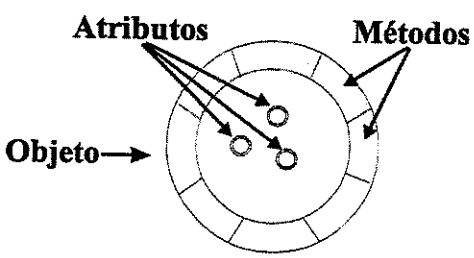


Figura 41 - Um objeto.

A parte externa do objeto é sua interface com o mundo exterior, composta pelo conjunto de métodos e dados que são declarados como públicos e/ou protegidos.

Enquanto uma classe é uma abstração que descreve os atributos (variáveis) e os comportamentos (métodos) de elementos do mundo real sendo modelados, um objeto representa um exemplar único, específico de uma classe.

A criação de objetos em Java ocorre em duas etapas. Primeiro declara-se normalmente uma variável do tipo de uma classe existente. Quando isso é feito não há alocação de espaço em memória para armazenar o objeto. Essa informação será utilizada para verificação dos tipos em

tempo de compilação, além de ser uma declaração de ponteiro implícita (em Java os nomes dos objetos funcionam como ponteiros, mas não são manipulados explicitamente como na linguagem C++). Na segunda etapa é feita a alocação de memória utilizando-se o operador *new*, que é feito dinamicamente durante a execução.

```
Ex: tspProblem tsp = new tspProblem(dimension, matrixCost);
```

Essa expressão é uma invocação do construtor. A aplicação do operador *new* ao construtor da classe retorna uma referência para o objeto. Para que o objeto possa ser efetivamente manipulado, essa referência deve ser armazenada por quem determinou a criação do mesmo. Nesse exemplo, *tsp* é uma variável que guarda uma referência para um objeto do tipo *tspProblem.*

No paradigma de orientação a objetos, tudo pode ser potencialmente representado como um objeto e, sob este ponto de vista, um objeto não é muito diferente de uma variável normal. Por exemplo, quando define-se uma variável do tipo *int* em Java, essa variável tem um espaço em memória para registrar o seu estado (valor) e um conjunto de operações que podem ser aplicadas a ela, através dos operadores definidos na linguagem que podem ser aplicados a valores inteiros. Da mesma forma, quando se cria um objeto, este adquire um espaço em memória para armazenar seu estado (os valores de seu conjunto de atributos definidos pela classe) e um conjunto de operações que podem ser aplicadas a ele (o conjunto de métodos definidos pela classe).

Quando declara-se uma variável cujo tipo é o nome de uma classe, como em

```
tspProblem tsp;
```

não está se criando um objeto dessa classe, mas simplesmente uma referência (*tsp*) para um objeto da classe *tspProblem*, a qual inicialmente não faz referência a nenhum objeto válido.

Quando um objeto dessa classe é alocado, obtém-se uma referência válida, que é armazenada na variável cujo tipo é o nome da classe do objeto. Por exemplo, em

```
tsp = new tspProblem( );
```

tsp é uma variável que armazena uma referência para um objeto específico da classe *tspProblem.*

É importante observar que a variável *tsp* mantém apenas a referência para o objeto e não o objeto em si. Assim, uma atribuição como

```
tspProblem tsp1 = tsp;
```

não cria outro objeto, mas simplesmente uma outra referência para o mesmo objeto.

O único modo de aplicar os métodos a um objeto é através de uma referência ao objeto. Seguindo com o mesmo exemplo, para criar um novo objeto com o mesmo conteúdo do objeto existente, o método *clone( )* pode ser aplicado da seguinte forma:

```
TspProblem tsp2 = tsp1.clone( );
```

Para essas três variáveis apresentadas nos exemplos acima, a expressão

```
tsp == tsp1
```

resulta *true*, pois as duas referências são iguais (se referem ao mesmo objeto). No entanto,

```
tsp == tsp2 ou tsp1 == tsp2
```

resulta *false*, mesmo que os objetos tenham o mesmo conteúdo, pois as duas referências comparadas são distintas (referem-se a objetos iguais, mas não são o mesmo objeto). Para efetivamente comparar o conteúdo de objetos, o método *equals( )* deve ser utilizado. Assim, as expressões

tsp1.equals(*tsp2*) e *tsp*.equals(*tsp1*)

resultam *true*, pois comparam efetivamente os conteúdos dos objetos.

1.3.5. Modificadores de acesso

Cada classe deve definir quais de seus dados e métodos podem ou não serem acessados externamente. Isto é feito fazendo uso dos modificadores de acesso protegidos (*protected*), privados (*private*) e públicos (*public*). Caso nenhum modificador de acesso seja indicado, o compilador Java automaticamente trata como *public*.

- protegido: os métodos e atributos declarados como protegidos somente poderão ser acessados por métodos de subclasses da classe onde foram declarados.
- privado: os métodos e atributos que forem declarados como privados só poderão ser acessados por métodos da própria classe.

- público: os métodos e atributos declarados como públicos podem ser acessados por métodos de qualquer classe.

Em UML, os modificadores de acesso são representados por caracteres: + (*public*) - (*private*), ou # (*protected*). O software *Rational Rose* para modelagem em UML oferece símbolos alternativos para a representação dos modificadores de acesso.

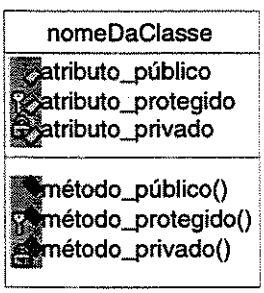


Figura 42 - Representação dos modificadores de acesso em UML.

Um retângulo inclinado indica um atributo (inclinado para direita) ou método (inclinado para esquerda) público; se acompanhado de uma chave indica protegido ou, se acompanhado de um cadeado, indica privado.

As técnicas de programação orientada a objetos recomendam que a estrutura de um objeto e a implementação de seus métodos devem ser tão privativos quanto possível. Normalmente, os atributos de um objeto não devem ser visíveis externamente. Da mesma forma, para um método deve ser suficiente conhecer apenas sua especificação, sem necessidade de saber detalhes de como a funcionalidade que ele executa é implementada.

1.3.6. Identificador

Nomes de classes, atributos e métodos devem ser identificadores válidos da linguagem. Um identificador é uma seqüência de caracteres que segue regras para definição:

- pode ser composto por letras (minúsculas e/ou maiúsculas), dígitos e os símbolos _ e $;
- não pode ser iniciado por um dígito (0 a 9);

- letras maiúsculas são diferenciadas de letras minúsculas;
- uma palavra-chave da linguagem Java não pode ser um identificador.

Embora não seja obrigatório, o conhecimento e uso da seguinte convenção para atribuir nomes em Java pode facilitar bastante a compreensão de um programa:

- nomes de classes são iniciados por letras maiúsculas;
- nomes de métodos, atributos e variáveis são iniciados por letras minúsculas;
- em nomes compostos, cada palavra do nome é iniciada por letra maiúscula, sendo que as palavras não são separadas por nenhum símbolo.

1.3.7. Ocultamento da informação

Ocultamento da informação é o princípio pelo qual cada componente deve manter oculta, sob sua guarda, uma decisão de projeto única. Para a utilização desse componente, apenas o mínimo necessário para sua operação deve ser revelado, restringindo o projetista à interface pública do objeto.

1.3.8. Encapsulamento

Encapsulamento é o princípio de projeto pelo qual cada componente de um programa deve agregar toda a informação relevante para sua manipulação como uma unidade (caixa preta). Através do encapsulamento são minimizadas as interdependências entre módulos, restringindo o acesso entre os mesmos apenas pela interface. Aliado ao conceito de ocultamento de informação, o encapsulamento é um poderoso mecanismo da programação orientada a objetos.

1.3.9. Polimorfismo X sobrecarga

O polimorfismo, juntamente com encapsulamento, ocultamento da informação e herança, é um dos princípios fundamentais da programação orientada a objetos.

Através do uso do polimorfismo, duas ou mais classes podem invocar métodos que têm a mesma identificação mas comportamentos distintos, especializados para cada classe. A identificação de um método é dada por seu nome e assinatura (tipos dos argumentos). No caso de polimorfismo, é necessário que os métodos tenham a mesma identificação, sendo utilizado o mecanismo de redefinição de métodos (*overriding*).

Sobrecarga (*overloading*) é o tipo mais comum de polimorfismo no qual dois ou mais métodos têm o mesmo identificador mas não a mesma identificação, ou seja, suas assinaturas são diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos seus argumentos. Um exemplo de sobrecarga de métodos é a utilização de mais de um construtor para uma classe: seus identificadores são os mesmos mas as assinaturas não.

1.3.10. *Upcasting*

A operação de atribuir uma referência de uma classe mais especializada à uma classe ancestral é denominada *upcasting*. Por exemplo, suponha que a classe *tspProblem* é derivada da classe *problem*. O procedimento abaixo gera um *upcasting*:

```
problem prb;
tspProblem tsp = new tspProblem(dimension, matrixDist);
prb = tsp;
```

Com isso, a referência *prb* poderá manipular os atributos e métodos da classe *tspProblem*.

1.3.11. Relacionamentos

As três formas de relacionamento mais utilizadas são: dependência, associação e generalização.

Dependência é o relacionamento entre dois elementos em que uma mudança na especificação de um elemento (independente) pode afetar a semântica do outro (dependente), mas não necessariamente o contrário. Em UML este relacionamento é representado por uma seta tracejada (de ponta em >) que aponta para o elemento dependente, podendo apresentar um rótulo e a multiplicidade.

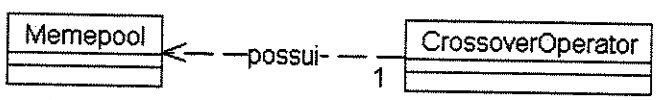


Figura 43 – Representação do relacionamento de dependência em UML.

Associação é o relacionamento que especifica que elementos de um tipo estão conectados a elementos de outro tipo. Em UML uma associação é representada por uma linha sólida, podendo conter um rótulo e, normalmente, contém a multiplicidade. Embora no exemplo abaixo não seja visualizado, uma associação pode informar os nomes dos elementos aos quais cada classe associada representa.

Figura 44 – Representação do relacionamento de associação em UML.

Agregação é um tipo de associação que representa um relacionamento estrutural entre o todo e suas partes. Numa agregação, um objeto pode fazer parte de apenas um elemento composto a cada vez e, caso o elemento composto seja excluído, tal objeto deixa de existir. Uma agregação possui a mesma representação da associação, adicionada de um losango no extremo que conecta a classe agregadora. O losango pode aparecer preenchido, caso a agregação siga rigorosamente as características descritas por esse tipo de relacionamento (Booch et al., 1999).

Figura 45 – Representação do relacionamento de agregação em UML.

Associação do tipo navegação é uma associação onde somente é possível navegar de um lado *a* para um lado *b* da associação (não se pode navegar de *b* para *a*). Neste tipo de associação, a navegação só pode ser feita no sentido da seta. Uma associação do tipo navegação possui a mesma representação da associação, adicionada de uma seta (com ponta em >) indicando o sentido da navegação permitida.

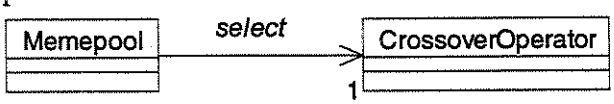


Figura 46 – Representação de uma associação do tipo navegação em UML.

Generalização é um relacionamento de especialização/generalização entre uma superclasse (classe mais genérica - chamada de classe pai) e uma subclasse (classe mais específica - chamada de classe filha). A classe mais específica faz uso da estrutura e dos métodos da classe mais genérica. Esse relacionamento determina o mecanismo de herança. Em UML uma generalização é representada por uma seta (com ponta em ∇) direcionada para a classe mais genérica.

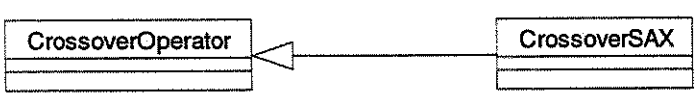


Figura 47 – Representação do relacionamento de generalização em UML.

A forma básica de herança em Java é a extensão simples entre uma superclasse e sua classe derivada. Para tanto, utiliza-se na definição da classe derivada a palavra-chave *extends* seguida pelo nome da superclasse.

Para definir uma classe derivada *tspProblem* da superclasse *problem* o seguinte código é requerido:

```
class tspProblem extends problem {
    // corpo da classe
}
```

Java não oferece o mecanismo de herança múltipla, ou seja, não é possível criar uma classe derivada com mais de uma classe base. Por esse motivo, é simples fazer uma referência da classe derivada para sua superclasse; o mecanismo para tal é o uso da palavra-chave *super*. Construtores da superclasse podem ser explicitamente invocados usando o método *super( )*. Outro uso dessa palavra-chave é como prefixo para referenciar métodos da superclasse.

1.3.12. As Palavras-chave *abstract, static* e *final*

A palavra-chave *abstract* modifica uma declaração para indicar que não há uma definição concreta do que vem a seguir. Pode ser aplicada a classes ou a métodos.

Uma classe abstrata é uma classe que não pode ser instanciada, ou seja, não há objetos que possam ser construídos diretamente de sua definição. Uma classe abstrata pode conter um ou mais métodos abstratos. Um método abstrato não cria uma definição, mas apenas uma declaração de um método que deverá ser implementado em uma classe derivada. Se esse método não for implementado na classe derivada, esta permanece como uma classe abstrata mesmo que não tenha sido assim declarada explicitamente. Quando uma classe só contém métodos abstratos, ela pode ser uma classe abstrata ou uma interface Java.

Em UML, a identificação de métodos e classes abstratas é feita com seus identificadores escritos em itálico.

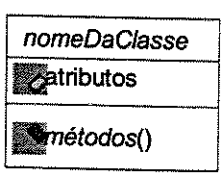


Figura 48 - Representação de uma classe e um método abstratos em UML.

Em Java um atributo precedido da palavra reservada *final* é definido como constante, ou seja, não pode ter seu valor alterado durante a execução do programa. O valor do atributo deve ser definido no momento da declaração, pois não é permitida nenhuma atribuição em outro momento.

A partir de Java 1.1, é possível ter atributos de uma classe que sejam *final* mas não recebem

valor na declaração, mas sim nos construtores da classe. (A inicialização deve obrigatoriamente ocorrer em uma das duas formas.) São os chamados *blank finals*, que introduzem um maior grau de flexibilidade na definição de constantes para objetos de uma classe, uma vez que essas podem depender de parâmetros passados para o construtor.

A utilização de *final* para uma referência a objetos é permitida. Como no caso de constantes, a definição do valor (ou seja, a criação do objeto) também deve ser especificada no momento da declaração. No entanto, é preciso ressaltar que o conteúdo do objeto em geral pode ser modificado, apenas a referência é fixa. O mesmo é válido para os vetores.

Argumentos de um método que não devem ser modificados podem ser declarados como *final*, também, na própria lista de parâmetros. A palavra chave *final* pode ser utilizada como uma indicação de algo que não deve ser modificado ao longo do restante da hierarquia de descendentes de uma classe. Pode ser associada a atributos, métodos e classes.

Cada objeto definido a partir de uma classe terá sua cópia separada dos atributos definidos para a classe. No entanto, há situações em que é interessante que todos os objetos compartilhem a mesma variável, similarmente ao que ocorre com variáveis globais em linguagens de programação tradicional. O mecanismo para realizar esse compartilhamento é ter a declaração da variável precedida pela palavra-chave *static*.

Variáveis constantes e visíveis para todas as classes são definidas em Java como *public static final*.

1.3.13. Interfaces

Na orientação a objetos, o uso da encapsulamento e ocultamento da informação recomenda que a representação do estado de um objeto deve ser mantida oculta. Cada objeto deve ser manipulado exclusivamente através dos métodos públicos do objeto, dos quais apenas a assinatura deve ser revelada. O conjunto de assinaturas dos métodos públicos da classe constitui sua interface.

Dessa forma, detalhes internos sobre a operação do objeto não são conhecidos, permitindo que o usuário do objeto trabalhe em um nível mais alto de abstração, sem preocupação com os detalhes internos da classe. Essa facilidade permite simplificar a construção de programas com funcionalidades complexas (tais como interfaces gráficas ou aplicações distribuídas).

Uma interface Java é uma classe abstrata para a qual todos os métodos são implicitamente abstratos e públicos, e todos os atributos são implicitamente *static* e *final*. Em outros termos, uma interface Java implementa uma classe abstrata pura.

A sintaxe para a definição de uma interface Java equivale àquela da definição de uma classe, apenas usando a palavra chave *interface* ao invés da palavra chave *class*.

Por exemplo, para definir uma interface chamada *interface1* que declara um método *void met1()* a sintaxe é:

```
interface Interface1 {
    void met1();
}
```

A diferença entre uma classe abstrata e uma interface Java é que a interface obrigatoriamente não tem um corpo associado. Para que uma classe seja abstrata basta que ela seja assim declarada, mas a classe pode incluir atributos e definição de métodos, públicos ou não. Na interface, apenas métodos públicos podem ser declarados, mas não definidos. Da mesma forma, não é possível definir atributos, apenas constantes públicas.

Enquanto uma classe abstrata é estendida (palavra-chave *extends*) por classes derivadas, uma interface Java é implementada (palavra chave *implements*) por outras classes.

Outra diferença essencial entre classes e interfaces Java é que uma classe pode implementar múltiplas interfaces, mas pode estender apenas uma superclasse. Quando mais de uma interface está sendo implementada, a sintaxe de declaração da classe especifica que as interfaces implementadas sejam separadas por vírgulas:

```
class Derivada extends SuperClasse implements Interface1, Interface2 {
    // corpo da classe;
}
```

Neste exemplo, a classe Derivada deve implementar todos os métodos da *Interface1* e todos os métodos da *Interface2*. Adicionalmente, se SuperClasse também tiver métodos abstratos,

esses também deverão ser implementados por métodos da classe Derivada.

Em UML uma interface pode ser representada da mesma forma que uma classe, com a diferença que a palavra interface aparece entre << e >>.

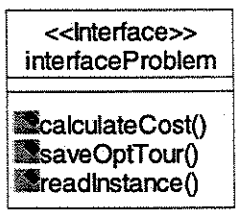


Figura 49 - Representação de uma interface em UML.

Embora a representação seja similar ao de uma classe, no *software Rational Rose* o objeto de seleção deve ser diferenciado para classes e interfaces.