UNIVERSIDADE PRESBITERIANA MACKENZIE
Faculdade de Computação e Informática

Aplicação de Algoritmos Genéticos em Semáforos Inteligentes

Wellington Cruz

SÃO PAULO
2011

Wellington Cruz

Aplicação de Algoritmos Genéticos em Semáforos Inteligentes

Trabalho de Graduação Interdisciplinar apresentado à Faculdade de Computação e Informática da Universidade Presbiteriana Mackenzie, como exigência parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Msc. Joaquim Pessoa Filho

São Paulo - SP, Brasil

2011

UNIVERSIDADE PRESBITERIANA MACKENZIE
FACULDADE DE COMPUTAÇÃO E INFORMÁTICA

Termo de Julgamento de Trabalho de Graduação Interdisciplinar

CIÊNCIA DA COMPUTAÇÃO

Aos 01 dias do mês de dezembro de 2011, às 16:15 horas, no prédio 13, sala 201 da Universidade Presbiteriana Mackenzie, presente a Comissão Julgadora integrada pelos senhores Professores abaixo mencionados, iniciou-se a apresentação do Trabalho de Graduação Interdisciplinar do grupo formado pelos alunos a seguir relacionados. Concluída a argüição, procedeu-se ao julgamento na forma regulamentar, tendo a Comissão Julgadora atribuído as seguintes notas aos candidatos.

Título do Trabalho: ***APLICAÇÃO DE ALGORITMOS GENÉTICOS EM SEMÁFOROS INTELIGENTES***

	ALUNO	**Matrícula**	**Ênfase**
1	WELLINGTON LUIZ DA CRUZ	30698618	Ciência da Computação
2			
3			
4			

	Relatórios			Banca Examinadora						
Aluno	R1 (10%)	R2 (20%)	R3 (30%)	Apres. (40%)	Média Orientador	Examinador 1	Examinador 2	Média Banca	Partici-pação	MÉDIA FINAL
1	10,0	10,0	10,0	10,0	10,0	10,0	10,0	10,0	0.60	10,0
2										
3										
4										

Para constar, é lavrado o presente termo que vai assinado pela Comissão Julgadora e pelo Coordenador de TGI.

São Paulo, 1 de novembro de 2011

Comissão Julgadora

Orientador: **Prof. Ms. Joaquim Pessoa Filho**

Examinador 1: **Prof. Dr. Luciano Silva**

Examinador 2: **Prof. Ms. Antonio Luiz Basile**

Suplente: **Profª. Drª. Ilana de Almeida Souza**

Coordenador de TGI: **Prof. Dr. Luis Tadeu M. Raunheitte**

Resumo

O presente trabalho procura modelar o problema da otimização de tempos de semáforo, de forma que se possa aplicar os métodos de programação com algoritmos genéticos em busca da solução otimizada. Como referêncial téorico apresenta os principais métodos de controle de semáforos existentes e os fundamentos da programação evolutiva. Apresenta detalhes da implementação do algoritmo genético proposto e analisa os resultados dos experimentos realizados.

Palavras-chave: algoritmos genéticos, semáforos inteligentes, simulador, SUMO, SCOOT, Transyt, TraCI

Abstract

The actual research project persues to model the optimization problem of traffic lights timing, in a manner that can be apply the concepts of genetical algorithms programming in search for a optimized solution. As theoretical reference shows the main methods of traffic lights control and the fundamentals of evolutive computation. Presents implementation details of the proposed genetic algorithm and analyzes the results of experiments.

Keywords: genetic algorithms, intelligent traffic lights, simulator, SUMO, SCOOT, Transyt, TraCI

Agradecimentos

Agradeço primeiramente à minha família pelo apoio e alicerces familiares necessários para a conclusão desta graduação. Agradeço a minha namorada Natali pelo apoio e ajuda nos momentos de produção desta monografia. Agradeço ao meu orientador Prof. Joaquim Pessoa Filho, pela orientação precisa para o desenvolvimento deste trabalho. E agradeço ao Prof. João Cucci Neto, da Faculdade de Engenharia da Universidade Presbiteriana Mackenzie, pelas orientações e material oferecido para utilização neste trabalho.

Lista de Figuras

Lista de Tabelas

Sumário

1 Introdução

É da natureza do homem a busca de soluções para os problemas que cercam nosso dia-a-dia, bem como a otimização a processos que são realizados de forma repetitiva. Pode-se apontar inúmeros problemas que surgiram desde que o homem passou a criar processos para a interagir com a natureza, produzindo trabalho e riqueza.

Com a evolução do trabalho e da produção, a humanidade passou a encontrar novos problemas decorrentes desta evolução. A criação de cidades e de novas tecnologias de transportes acarretam muitos problemas específicos, como poluição, logística, excesso de veículos e consequentemente congestionamentos.

Atualmente os congestionamentos propiciam um dos maiores e preocupantes problemas enfrentados pelas grandes cidades, trazendo grandes prejuízos em consequência de atrasos e tempo improdutivo para deslocamento entre espaços relativamente pequenos nestas grandes cidades.

Muitos estudos são feitos na tentativa de resolver o problema de congestionamentos, em vista de que a quantidade de veículos novos que são colocados em circulação a cada dia, é cada vez maior.

Uma das principais áreas de estudo envolvidas nas soluções de redução de congestionamentos é a programação semáforica que possui o objetivo de administrar os tempos de verde / vermelho a fim de maximizar a quantidade de veículos que trafegam pelas vias, reduzindo o número de paradas e tempo de espera nessas intersecções semaforizadas.

É na Computação Natural que vamos buscar alicerces para uma abordagem a respeito do problema de otimização de tempos de semáforo, aplicando os fundamentos da computação evolutiva e algoritmos genéticos. O objetivo desta pesquisa é modelar o problema da otimização de tempos de semáforo de forma que possa ser aplicado a ele os conceitos de algoritmos genéticos.

Neste trabalho, foi utilizado como base de conhecimento o estudo de soluções de

controle semáforico já utilizadas no mercado, aplicando dentro deste trabalho conceitos já comprovadamente eficazes, para que se possa ter como tema central a implementação destes algoritmos na busca de soluções otimizadas para os tempos de semáforo.

O presente trabalho se justifica pela necessidade de estimular discussões a cerca de processos de otimização que possam culminar em uma melhor otimização e aproveitamento das vias e semáforos, e que possam colaborar de uma forma geral com o problema dos congestionamentos nos grandes centros urbanos.

O trabalho está organizado na seguinte forma: o referencial teórico pode ser encontrado nos Capítulos 1 a 3, sendo os dois primeiros referentes às tecnologias existentes de controle de semáforos inteligentes; o terceiro capítulo aborda a teoria e conceitos dos Algoritmos Genéticos. O Capítulo 4 apresenta detalhes da metologia utilizada e os conceitos do método aplicado na implementação do algoritmo, bem como seus resultados e respectivas análises. Por fim, o Capítulo 5 apresenta as considerações finais e propostas de trabalhos futuros complementares a esta pesquisa.

2 Sistemas de Controle de Semáforos

Para atender a demanda de crescimento dos grandes centros como a cidade de São Paulo, diferentes estratégias de controle de semáforos são utilizadas a fim de otimizar a fluidez do tráfego e diminuir os tempos de espera e deslocamento de veículos.

Em São Paulo, já em 1982 a cidade contava com um sistema centralizado que era responsável pelo controle de cerca de 470 semafóros nas regiões com maior dificuldade de tráfego. A este sistema se deu o nome de SEMCO, sigla para Semáforos Coordenados. Desde 1994, a cidade de São Paulo utiliza o sistema de Centrais de Tráfego em Área (CTA) (VILANOVA, 2005a).

2.1 O Sistema SEMCO

No fim da década de 1970, com a criação da CET (Companhia de Engenharia de Tráfego) iniciou-se um período de pesquisa e desenvolvimento de sistemas que pudessem contribuir e otimizar a fluidez do tráfego na cidade de São Paulo.(VILANOVA, 2005a)

Anteriormente ao sistema SEMCO, a cidade de São Paulo contava com aproximadamente 2500 cruzamentos semaforizados. Cerca de 80% destes semáforos permitiam uma única programação de tempos de verde, amarelo e vermelho. Os outros 20% eram de fabricação importada e permitiam apenas 3 programações, uma delas para horário de pico matutino, outra para o horário de pico noturno e outra para os demais horários.

Em 1982 foi implantado então, o sistema SEMCO.

O SEMCO consiste basicamente em controladores de tráfego dotados de sensores de solo que podiam captar o volume de tráfego da via onde instalados. Essas informações eram enviadas para uma única central de controle localizada dentro da CET, a qual era responsável pela compilação destes dados.

Nesta sede, haviam dois módulos centrais de processamento, o MCC (Módulo Central de Controle) e o MSG (Módulo de Supervisão Geral). O MCC era responsável por monitorar o funcionamento de todos os semáforos da rede e fornecer informações e controles para que os engenheiros pudessem alterar remotamente os planos de tráfego, seja por reconhecimento de padrões de tráfego, seja por uma tabela de horários. O sistema contava com 30 planos previamente otimizados pelos engenheiros com o auxílio do MSG.

O MSG era responsável por auxiliar os engenheiros a criar planos de tráfego otimizados com base nas curvas estatísticas extraídas da base de dados formada pelos dados captados pelos sensores de solo. Todo esse procedimento era feito *off-line*. (BOZOLA, 1983)

Desta forma os engenheiros da CET podiam intervir remotamente nos tempos e ciclos dos semáforos, visando uma melhor fluidez do tráfego.

Logo esse sistema se tornaria ineficaz para a cidade. Dada as necessidades do dia-a-dia da operação de tráfego, era necessário um sistema mais eficiente que aplicasse maior agilidade nas mudanças de fluxo e dinamismo ao trânsito.(BOZOLA, 1983)

2.2 O Sistema SEMIN / CTA

Por volta de 1995, existiam, aproximadamente, 4000 semáforos em São Paulo. Herança do SEMCO, cerca da metade ainda era controlado por equipamentos eletro-mecânicos obsoletos (VILANOVA, 2005a).

Em 1994 a CET iniciou então a pesquisa e desenvolvimento do Sistema SEMIN (Semáforos Inteligentes) com CTA (Centrais de Tráfego em Área). A principal diferença entre o sistema SEMCO e o SEMIN é o controle em *tempo real*. Isso significa que o sistema teria um dinamismo muito maior.

No modelo anterior, era necessário que muitos veículos passassem por uma via saturada para que os engenheiros de tráfego pudessem reconhecer uma alteração no padrão de tráfego e pudessem realizar a mudança necessária em seu plano. Os planos disponíveis eram previamente processados, e provavelmente o melhor plano disponível não seria o plano mais otimizado para a situação. Esse tempo entre a captação dos dados, definição das medidas adotadas e a inexatidão do plano adotado, eram motivos para um congestionamento como consequência.

O novo modelo trouxe mais dinamismo para a cidade: graças a um *software* executado na Central de Controle, o sistema é capaz de processar as informações coletadas pelos

controladores de tráfego e determinar os tempos de ciclos otimizados para os cruzamentos, *on-line e em tempo real*. Desta forma os controladores são ajustados gradativamente, diminuindo assim o tempo entre o ajuste do semáforo e o aumento de fluxo de veículos na via (VILANOVA, 2005a).

2.2.1 Tolerância à Falhas

O SEMIN (Semáforos Inteligentes) é projetado para trabalhar em modo automático em 100% do tempo, exigindo o mínimo de interferência dos engenheiros e operadores, reduzindo o risco de erros e falhas humanas.

Porém, é muito grande o número de variáveis e fatores de risco que influenciam no funcionamento desse tipo de sistema, mesmo porque, boa parte de sua estrutura física encontra-se ao ar livre. Existe o risco de dano em parte do equipamento, o que poderia deixar o sistema inoperante causando um grande transtorno à cidade.

Pensando nisso, o SEMIN possui um sistema de tolerância a falhas, o qual implementa outras formas de operação que podem ser utilizadas em caso de falhas parciais do equipamento:(VILANOVA, 2005a; BOZOLA, 1983)

- *Controle Central em Tempo real:* Operação padrão do sistema. Opera em modo automático exigindo o mínimo de intervenção dos operadores. Opera quando todos os recursos estão disponíveis e não apresentam falhas.

- *Controle Central em Tempos Fixos:* Nesta configuração, o sistema passa a operar determinando os tempos de ciclo dos controladores de forma fixa. A central dispõe de uma série de programas de tempos de ciclo fixos, criada com base em dados estatísticos do comportamento médio do fluxo apresentado anteriormente pela via e que podem ser aplicadas remotamente. Esse planos podem ser ativados por meio de uma tabela de agendamento, ou por imposição do operador do centro de controle. Essa forma de operação é especialmente importante em caso de problemas nos mecanismos de detecção de fluxo de veículos da via.

- *Controle Local em Tempos Fixos:* De forma análoga ao modo de operação anterior, o controloador passa a operar com tempos de ciclos fixos agendados ou impostos pelo operador. Porém diferentemente do controle central, este modo precisa ser aplicado em campo pelo agente de trânsito. O sistema passa a operar desta forma quando há alguma falha na rede de transmissão que impossibilite a comunicação

entre o controlador e a central de controle.

- *Controle Manual:* Esta é a forma mais trivial de funcionamento do sistema. Desta forma o agente de trânsito em campo pode manualmente acionar as mudanças de estágio do semáforo. Pode ser utilizado em caso de falhas que impossibilitem o uso dos modos de operação citados anteriormente.

- *Amarelo Intermitente:* Quando ocorre alguma falha no sistema que impeça o funcionamento pleno do semáforo. O amarelo piscante alerta os motoristas da situação de risco criada pela falta de controle semafórico.

2.2.2 Centrais Distribuídas

O SEMCO centraliza todas as suas informações em uma única central de controle, o que prejudica o desempenho dos serviços de operação e manutenção dos controladores de tráfego. Além disso, aumentando o número de controladores no sistema, poderia-se gerar um volume de processamento muito grande para uma única central. A figura 1 mostra um exemplo do ambiente de uma central de controle.

Figura 1: CTA1 Centro Expandido.
Fonte: (NETO, 2011)

Em São Paulo, a CET distribuiu o processamento destas informações em 5 CTAs (Centrais de Tráfego de Área):

- CTA-1 - Centro Expandido

- CTA-2 - Oeste e Norte
- CTA-3 - Leste
- CTA-4 - Sudeste
- CTA-5 - Sul

Para diminuir os custos de instalação de redes de transmissão de dados, cada central foi instalada o mais próximo do centro geográfico da região por ela comandada (VILANOVA, 2005a).

2.2.3 Monitoramento de Vídeo

Para auxiliar os operadores na interpretação dos dados enviados e processados, o sistema conta com uma rede de monitamento de vídeo com câmeras instaladas pelas vias por onde atua.

Essa rede de monitamento de vídeo permite aos operadores confrontar as informações capturadas pelo sensores de solo com as imagens das vias monitoradas. O monitoramento de vídeo também é importante para levar aos operadores outros tipos de informações e ocorrências relevantes que impactam diretamente na fluidez do tráfego e que não podem ser capturadas pelos sensores automáticos, como acidentes, enchentes, etc.

Apesar de serem complementares quanto à importância e forma de atuação, o sistema de semáforo e o sistema de vídeo são completamente independentes. O mal funcionamento de um dos sistemas não afeta tecnicamente o funcionamento do outro sistema (VILANOVA, 2005a).

2.2.4 Arquitetura do Sistema

2.2.4.1 Estrutura Física

O Sistema CTA é um sistema inteligente projetado para trabalhar de modo automático em 100% do tempo. Para isso é necessário contar com uma estrutura física confiável. Caso alguma falha na estrutura seja detectada, o sistema passa a funcionar de acordo com os modos de operação descritos na seção 2.2.1.

A seguir na figura 2 pode-se ver o diagrama de estrutura física do sistema SEMIN / CTA.

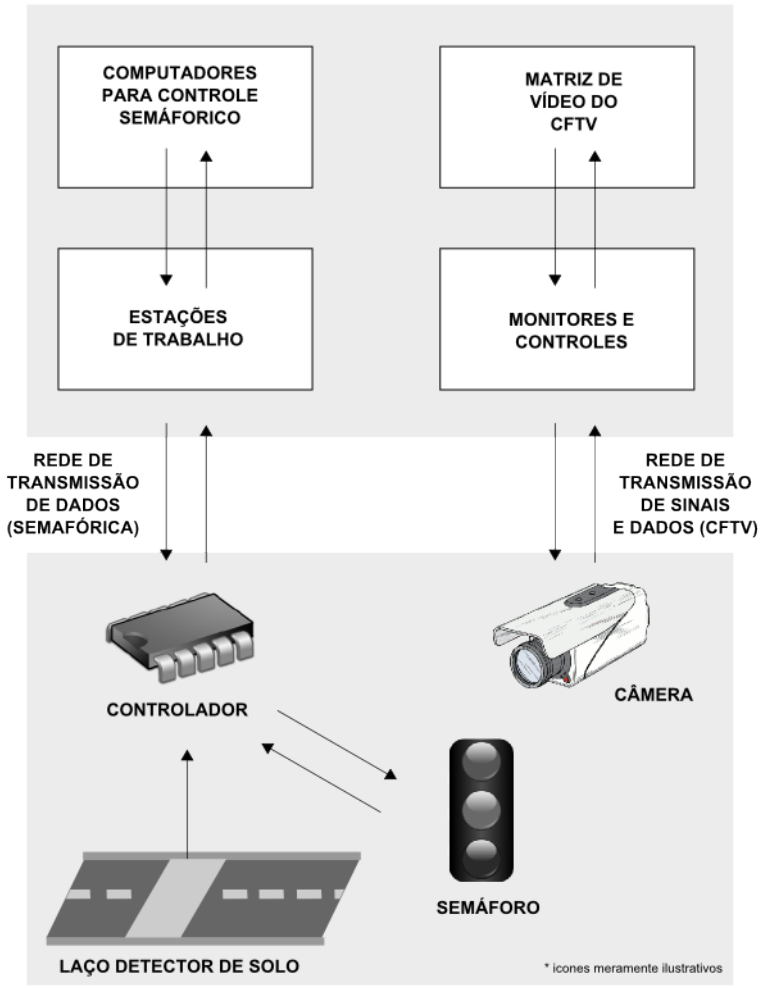

Figura 2: Estrutura Física do Sistema
Adaptado de: (VILANOVA, 2005a)

A arquitetura física do sistema, é basicamente constituída de um controlador de tráfego, um laço detector de solo, um semáforo (conjunto de luzes de sinalização) e na central de controle, o computador central de processamento e as estações de trabalho interligadas, que podem enviar intervenções ao sistema e oferecendo o monitoramento em tempo real para os engenheiros de trânsito. Os dois lados são ligados através de uma rede de transmissão de dados, tipicamente uma rede telefônica.

Paralelamente ao sistema de controle semafórico, existe o sistema de transmissão de vídeo, totalmente independente ao controle de semáforo, composto pela matriz de vídeo de sistema e os monitores de vídeo que exibem as imagens recebidas pelas câmeras, interligado pela rede de transmissão de sinal de vídeo.(VILANOVA, 2005a)

2.2.4.2 Componentes do Sistema

- *Controlador de Tráfego:* responsável por processar e interpretar os dados recebidos pela Central de Controle, captar os dados dos sensores de solo, enviar para a central de controle e temporizar adequadamente o chaveamento das luzes do semáforo. Um controlador de tráfego típico, como o SIEMENS T400 utilizado em São Paulo, é composto por: modem, CPU, módulo de potência, módulo de I/O, cartão detector e fonte de alimentação. Um controlador de tráfego possui ainda microcontrolador programável com sofware de controle de tráfego que permite os modos de operação local descritos no item 2.2.1 em caso de problemas na comunicação com a central.(CAPELLI, 2009)

  Na Figura 3, pode-se ver a estrutura de um controlador de tráfego típico.

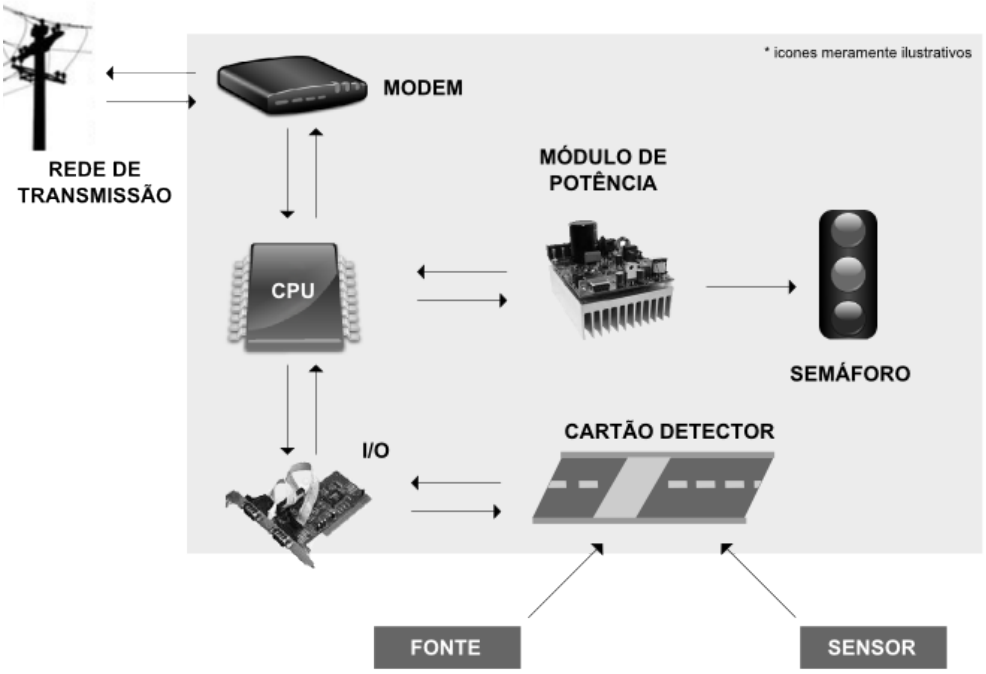

Figura 3: Estrutura de um Controlador de Tráfego
Adaptado de: (CAPELLI, 2009)

- *Sensores de Solo:* Os cartões detectores de solo são os equipamentos responsáveis

pela captação do fluxo de veículos que trafegam sobre a via. Seu funcionamento é muito semelhante ao funcionamento dos detectores de metais. Uma bobina é colocada sob a pista e quando um veículo passa sobre ela, é detectado. Isso acontece porque o equipamento possui um oscilador interno que gera um sinal em forma de senoíde para a bobina. Quando uma massa metálica passa por cima da bonina, ocorre uma defasagem na senóide. Um conversor Analógico / Digital converte a defasagem da senóide para sinal digital e assim a quantidade de veículos é contada.(CAPELLI, 2009)

3 Softwares de Gerenciamento de Tráfego

Softwares de Gerenciamento de Tráfego são algoritmos que atuam sobre a estrutura física descrita no capítulo anterior, aplicando uma lógica que visa a otimização da fluidez de veículos nas intersecções semaforizadas. Para a compreensão deste algortimos, são definidas algumas terminologias a respeito do funcionamento geral da programação de semáforos (VILANOVA, 2011):

- Tempo de verde: intervalo de tempo onde o semáforo permite que os veículos de uma determinada via avancem sobre uma intersecção.
- Tempo de vermelho: intervalo de tempo onde o semáforo bloqueia a passagem de uma determinada via, para que os veículos da outa via da intersecção tenham prioridade.
- Tempo de amarelo: intervalo que antecede o tempo de vermelho, como sinal de alerta. Serve para que a via fique disponível para o próximo tempo de verde.
- Tempo de ciclo: tempo total usado por um semáforo para percorrer os tempos de verde, amarelo e vermelho, retornando ao primeiro estágio. Tempo de verde + tempo de amarelo + tempo de vermelho = Tempo de ciclo

Podemos notar que numa mesma intersecção, duas ou mais vias são controladas pelo mesmo controlador de semáforo, cada uma delas possuindo o seu próprio conjunto luminoso. Neste trabalho, vamos considerar somente intersecções com duas vias. Neste caso específico, o tempo de verde de uma via corresponde ao tempo de vermelho da outra, e vice-versa.

Não é objetivo deste trabalho se aprofundar nos conceitos de programação semáforica. Portanto, os conceitos são expostos de forma simplificada para melhor aplicação. Outros

conceitos avançados como Tempo Morto, Verde Efetivo, Verde de Segurança, etc, não serão expostos pois não serão utilizados no desenvolvimento desta pesquisa.

3.1 TRANSYT

Criado em 1969 pelo Prof. D. I. Robertson no TRL (*Transport Research Laboratory* - Orgão de pesquisas de tráfego do Ministério de Transportes da Inglaterra), o TRANSYT (*Traffic Network Study*) é um software usado para determinar planos de tráfego em *tempos fixos* com o objetivo de otimizar o fluxo de tráfego na rede viária, através de uma função de otimização que define a desafagem e os melhores tempos de verde.

O software foi utilizado na sua versão Transyt/6 na cidade de São Paulo juntamente com o sistema SEMCO para preparação de planos *off-line* de tempos de semáforos. (MUNHOZ, 1978)

Apesar de sua primeira versão ser bem antiga, o software continua sendo utilizado e desenvolvido até hoje pela TRL, e está na sua versão Transyt/14.

É importante observar os aspectos primários de funcionamento do *Transyt* porque muitos dos seus conceitos são utilizados atualmente em softwares mais modernos como o SCOOT (Seção 3.2).

O software utiliza basicamente dois elementos: uma modelagem de comportamento do fluxo veicular e uma função matemática de otimização.

3.1.1 Modelagem e Simulação Comportamental do Tráfego

Segundo Munhoz (1978), para que o software TRANSYT funcione corretamente, algumas hipóteses precisam ser assumidas como verdadeiras:

- Todas as importantes intersecções de via são sinalizadas.
- Todos os semáforos trabalham em ciclo comum.
- Os veículos entram na rede que será otimizada com uma taxa constante de chegada.
- Os fluxos de conversão nos cruzamentos têm porcentagens constantes.
- As filas formadas em certa aproximação são sempre escoadas no primeiro período de verde; não existem ciclos atrasados.

No *Transyt* a malha viária é representada por um conjunto finito de vértices e arestas. Os vértices representam intersecções viárias sinalizadas e as arestas as correntes de tráfego unidirecionais entre dois vértices. O ciclo é subdividido em unidades iguais de tempo e os cálculos do software são feitos tendo como base os valores médios de fluxo e filas previstas em cada um dos vértices.

O software simula o comportamento do tráfego, observando e manipulando três tipos de valores para cada vértice, que são chamados de histogramas de fluxo:

- Quantidade de fluxo de veículos que cruzariam o vértice se os véiculos não fossem retidos pelo fechamento do semáforo.
- Quantidade de escoamento de tráfego de um vértice.
- Saturação de fluxo: quantidade de fluxo de escoamento se durante o tempo de verde, o tráfego fluísse na capacidade máxima.

Todos os cálculos realizados no software são feitos com base nesses histogramas. (MUNHOZ, 1978)

3.1.1.1 Função Matemática de Otimização

De acordo com esse modelo defindo de tráfego, pode-se notar portando que o fluxo liberado de um vértice X, percorrerá uma determinada aresta até atingir o vértice Y na outra extremidade da aresta, onde será represado novamente por outro semáforo, até ser liberado novamente em seu período de verde.

Seja V um vértice pertencente à malha simulada, A o valor da função de *demanda* (número de veículos que chegam até o vértice V) e B o valor da função de *serviço* (número de veículos que são escoados durante o período de verde do vértice V), então a área definida pelo gráfico das funções A e B é dito como atraso de cruzamento no vértice V. Representado na figura 5.

Pode-se notar que variando o valor e o intervalo de tempo da função A (chegada do veículo ao vértice) e da função B (escoamento do veículo do vértice) o valor do atraso irá variar consequentemente. Desta forma pode-se manipular o valor de A - B para minimizar o tempo de atraso.

O modelo de otimização do *Transyt* procura chegar a um valor de atraso otimizado em relação à sua função objetivo, que é chamada de Índice de Performance (IP). Para

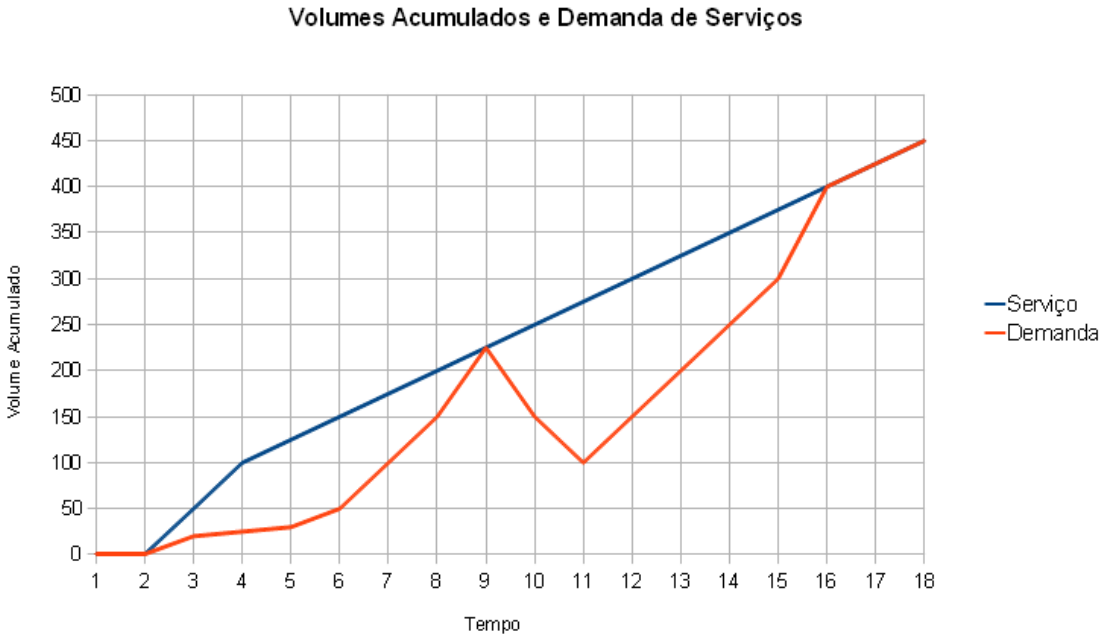

Figura 4: Volume Acumulado e Demanda de Serviço
Adaptado de: (MUNHOZ, 1978)

essa função são utilizados os valores das funções de demanda e de serviço geradas pelo simulador de tráfego.

A função objetivo - Índice de Performance (IP) é definida por (MUNHOZ, 1978):

$$\text{IP} = \sum_{i=1}^{n} (D_i + KS_i).$$

Sendo:

- n = número de vértices da malha representada
- D_i = valor médio de atraso total no vértice i
- K = fator de penalidade de parada
- S_i = número médio de paradas no vértice i

O fator de penalidade K é utilizado para aplicar prioridade a vias que tem prioridade no escoamento de tráfego.

O valor do IP pode ser composto de duas formas diferentes:

- *Atraso médio uniforme* - calculado em função dos fluxos nos vértices

- *Atraso médio aleatório* - considera os efeitos da flutuação do volume de fluxo de acordo com os ciclos, considerando a natureza estocástica da função de demanda (chegada ao vértice).

3.1.1.2 Algoritmo

O algoritmo de otimização do *Transyt* utiliza uma técnica matemática de otimização conhecida como *Hill Climbing* para determinar os tempos otimizados de semafóros para os vértices da malha representada (MUNHOZ, 1978).

Os algoritmos iterativos do tipo *Hill Climbing* são utilizados para encontrar uma solução otimizada para problemas de busca local de complexidade exponencial. Parte-se de uma solução arbitrária dada como otimizada, e então, tenta-se encontrar uma solução melhor a cada iteração, alterando um único elemento da fórmula. Caso a alteração produza um melhor resultado, essa solução é mantida como a mais otimizada e outras alterações são feitas a partir dessa solução.

O processo se repete até que não seja encontrado melhores resultados dentro de um limite de passos, e então, a solução encontrada é dada como solução ótima (RUSSELL; NORVIG, 2003).

3.2 SCOOT

SCOOT-UTC (*Split, Cycle, Offset, Optimization e Technique Urban Traffic Control*) é um software desenvolvido pela TRL (*Transport Research Laboratory*), Orgão de Pesquisas de Tráfego do Ministério de Transportes da Inglaterra, onde também é desenvolvido o TRANSYT (Seção 3.1) para controle otimizado de semáforos urbanos em *tempo real*.

O SCOOT é tido atualmente como principal software para controle de semáforos inteligentes. Esse projeto coordena a operação de todos os semáforos em determinada área provendo boa fluídez enquando os veículos trafegam pela malha, fazendo alterações na coordenação, sincronismo e tempos de ciclos nos semáforos, continuamente conforme as flutuações de fluxo de veículos. Com isso elimina-se a dependência de sistemas menos sofisticados, baseados em planos de tráfego e precisam ser constantemente atualizados.(BRETHERTON, 2011)

Sua primeira aplicação prática foi no início da década de 1980 e desde então vem sendo aperfeiçoado com base nas dificuldades encontradas e nas mudanças de mobilidade das grandes cidades. Desde então novas versões são lançadas incorporando novas funções e corrigindo problemas anteriores. No Brasil este sistema é utilizado com sucesso nas cidades de São Paulo e Fortaleza. (VILANOVA, 2005a)

3.2.1 Terminologia

Alguns termos específicos são utilizados para denominar itens e componentes específicos para a modelagem de tráfego com o SCOOT. Estes termos são definidos a seguir, e serão utilizados durante o capítulo (VILANOVA, 2005a):

- Área: é chamado de área, o conjunto de todos os itens gerenciados pelo SCOOT em um determinado sistema.
- Região: uma região é um conjunto de nós, que possuem o mesmo tempo de ciclo, ou seja, são coordenados entre si. Este conceito é muito importante para que o SCOOT possa sincronizar os estágios dos nós.
- Nó: tipicamente é uma interrupção do tráfego gerenciado por um semáforo. Uma intersecção de via semaforizada é um nó, vias semaforizadas sem intersecção (como por exemplo um semafóro de travessia de pedestres) também são denomizadas como nós.

- Link: é a ligação unidirecional entre dois nós, o nó-origem e o nó-destino. Note que uma via de mão dupla pode conter dois links, um em cada sentido, ligando dois nós, que se alternam entre origem e destino.
- Detector: equipamento responsável pelo registro de passagem dos veículos ao longo de um link.

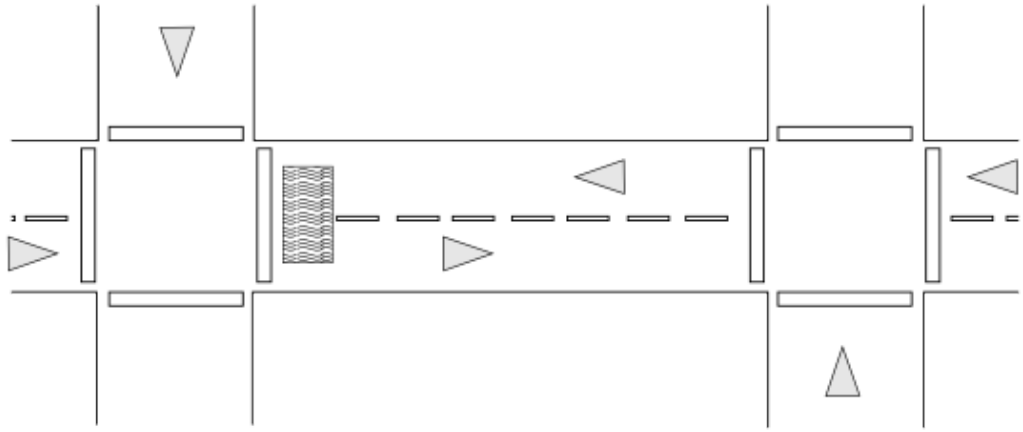

Figura 5: Representação comum de dois cruzamentos com a indicação de mãos de direção das vias
Adaptado de (VILANOVA, 2005a)

A via representada na Figura 5, de acordo com a terminologia de modelagem do SCOOT pode ser representada como apresentado na Figura 6.

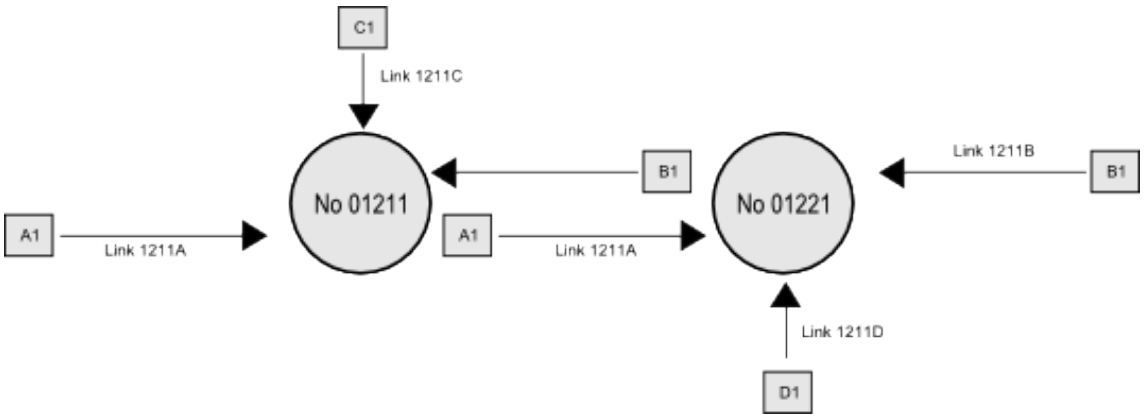

Figura 6: Representação da via de acordo com a terminologia SCOOT.
Adaptado de (VILANOVA, 2005a)

3.2.2 Conceitos

O SCOOT se baseia em um conjunto de conceitos básicos para seu funcionamento (VILANOVA, 2005a).

- Minimização de Transientes: o SCOOT é um modelo de programa do tipo adaptativo e aplica pequenas e frequentes modificações nos parâmetros da programação de semáforos, como o tempo de ciclo, tempos de estágios e defasagem. Este funcionamento não elimina o problema dos transientes, mas o minimiza, pois é capaz de se adaptar as demandas de fluxo ao longo do tempo.
- Predições em Curto Prazo: As decisões são baseadas na situação monitorada naquele exato momento.
- Respostas Ágeis: No final de todo estágio são recalculados os tempos de fases de semáforo para se adaptar aos veículos que foram detectados naquele momento.
- Trânsito em tempo real: o sistema utliza dados lidos através de detectores de solo para estimar a fila de veículos aguardando o semáforo. Traçando este perfil, os módulos internos de otimização serão alimentados.
- Falha de Detectores: Todos os detectores são testados continuamente e em caso de falhas no equipamento, a leitura destes passa a ser ignorada e o sistema, alimentado por valores padrão ou pelo usuário do sistema.
- Planos de partida: Para iniciar o processo de otimização, o SCOOT precisa de planos de tráfico previamente preparados que serão utilizados como ponto de partida.

3.2.2.1 Posicionamento dos Laços Detectores

Um dos fatores que difere o SCOOT dos demais sistemas de gerenciamento de tráfego, é o posicionamento dos seus laços detectores. Enquanto a maior parte do sistema realiza a contagem de veículos em uma posição muito próxima do nó (semáforo), o SCOOT realiza a contagem dos veículos através de um laço detector posicionado o mais longe possível do nó, mas ainda dentro do mesmo link, desde que não haja outros nós entre o nó e o detector.

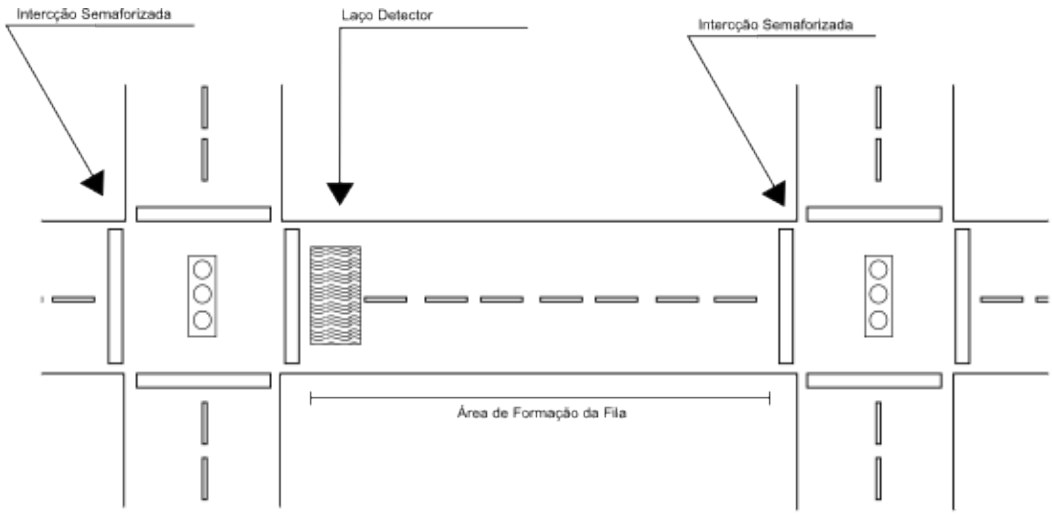

Figura 7: Representação do Posicionamento dos Laços Detectores

Posicionando o detector desta forma, conforme visto na Figura 7 o SCOOT consegue prever em alguns segundos de antecedência, o tamanho da fila que irá se formar na retenção de veículos no nó.

3.2.3 Como funciona

De modo geral, o SCOOT determina através dos laços detectores, um perfil de fluxo para cada link do sistema. Este perfil irá compor a principal estrutura de dados do sistema. Para traçar este perfil, o sistema precisa considerar o estado do semáforo no momento da passagem do veículo pelo detector:

- *Durante o tempo de vermelho:* o sistema considera que o veículo irá seguir toda a extensão do link (espaço entre o laço detector e o semáforo) e irá compor a fila de espera pelo verde.
- *Durante o tempo de verde:* o sistema considera que os veículos estão deixando a fila a uma velocidade constante de fluxo de saturação.

No sistema SCOOT, os laços detectores coletam as informações a cada 250 milisegundos. Este dados são processados e armazenados sob uma estrutura de dados chamada de Perfil Cíclico de Fluxo - *Cyclic Flow Profile*, utilizada para alimentar os componentes otimizadores do sistema. Esta estrutura contém as informações necessárias para que os componentes possam decidir o melhor tempo de estágio de cada nó, bem com a forma de sincronismo entre o nó-origem e nó-destino de cada link.

Graças ao posicionamento específico dos laços detectores do SCOOT, é possível prever com antecedência o tamanho da fila que será formada no momento de retenção do nó, projetando um perfil de fluxo para cada link. Para isso, é considerado que o veículo percorra o link do detector até o semáforo em uma velocidade constante. Também é aplicada uma função de amortecimento para simular uma situação de dispersão dos veículos.

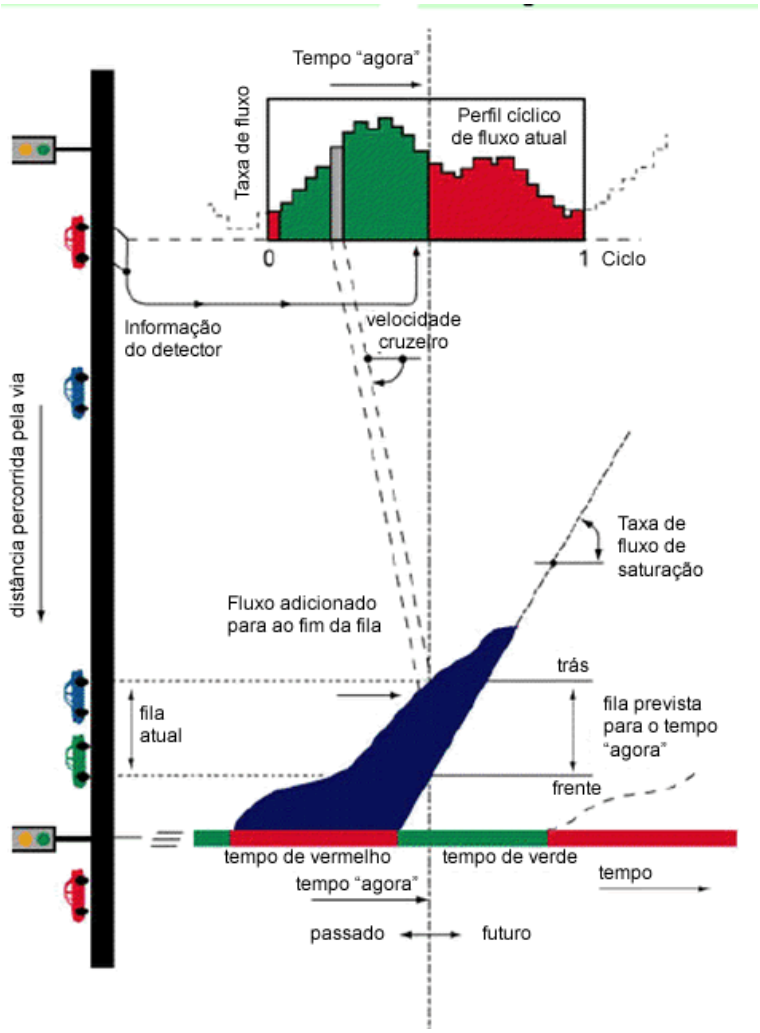

Figura 8: Representação da composição de um Perfil Cíclico de Fluxo
Fonte: (BRETHERTON, 2011)

A figura 8 mostra como é composto o perfil ciclico de fluxo: o gráfico em azul mostra o crescimento da vila de veículos no semáforo durante o tempo de vermelho, e como essa fila vai reduzindo durante o tempo de verde. Com essas informações, é possível formar um perfil de fluxo cíclico, pois essas informações são captadas a cada ciclo.

3.2.3.1 Detecção de Congestionamento

Os componentes de otimização de sistema se comportam de forma diferente para vias saturadas e para vias congestionadas. Para isso, é importante que o sistema tenha mecanismos de identificação de congestionamento. No SCOOT, o fator de congestionamento é interpretado diretamente pelo laço detector.

Com os laços posicionados o mais longe possível do semáforo (normalmente mais distante do que a extensão normal da fila de espera pelo semáforo), a presença constante de veículos sobre o laço detector pode ser interpretada como presença de congestionamento.

A presença de um veículo por mais de quatro segundos sobre um detector registra uma unidade de congestionamento no sistema. Com o passar do tempo, o acúmulo destas unidades determina o nível do congestionamento encontrado na via. De acordo com esses dados, os componentes otimizadores farão correções na programação de semáforos, considerando a situação da via como congestionada.

3.2.4 Componentes Otimizadores

O objetivo do algoritmo do SCOOT é predizer o comportamento do tráfego para os próximos instantes após o efeito de pequenas modificações nas configurações dos *Tempos de Estágio*, *Tempos de Ciclo* e *Defasagem*. Os valores de uma dessas configurações não depende das outras duas, e são calculadas por componentes individuais do sistema.

- Tempos de Estágio (*SPLIT*): é referente a quantidade de verde em cada via semaforizada.
- Tempos de Ciclo (*CYCLE*): é referente ao tempo de ciclo do semáforo, a soma de todos os estágios.
- Desafagem (*OFFSET*): é valor é referente ao tempo de atraso entre os semáforos, para que fiquem coordenados. Esse valor é especialmente importante em semáforos consecutivos, para que seja possível criar a "onda verde", onde os semáforos vão se abrindo em sequência após alguns segundos, diminuindo a quantidade de paradas.

3.2.4.1 Otimizador de Tempos de Estágio

O objetivo do Algoritmo Otimizador de Tempos de Estágio (SPLIT) é determinar um melhor tempo de estágio para cada link do sistema, ou seja, a quantidade de tempo

de verde para o semáforo. Este componente é acionado pouco tempo antes da mudança prevista de estágio. No momento em que é acionado, o algoritmo analisa quais das possibilidades é considerada a melhor para o próximo ciclo: antecipar a mudança do estágio (diminuir a quantidade de verde), manter a mudança de estágio da forma que está (mesma quantidade de verde) ou atrasar a mudança do estágio (aumentar a quantidade de verde).

O algoritmo considera como melhor opção aquela que melhor equaliza os níveis de saturação entre todos os links que chegam até o nó, ou seja: o objetivo do algoritmo é manter semelhante a saturação entre as vias que chegam ao semáforo. Note que isso não necessariamente quer dizer que o escoamento das vias será o mesmo: vias principais e com um maior fluxo de veículos precisam de um maior escoamento de carros para atingir um bom nível de saturação, enquanto vias com menor fluxo atingem esse nível mais facilmente.

O algoritmo precisa também considerar o nível de congestionamento dos links ligados ao nó. Neste caso, é observado não só o nível de saturação do link, mas também o nível de congestionamento. Desta forma, vias congestionadas tendem a ter maior prioridade em tempos de verde. Além disso, o algoritmo pode ainda, minimizar os tempos de verde das vias que alimentam a via congestionada, a fim de evitar o disperdício de tempos de verde e minimizar o problema do fechamento de cruzamentos.

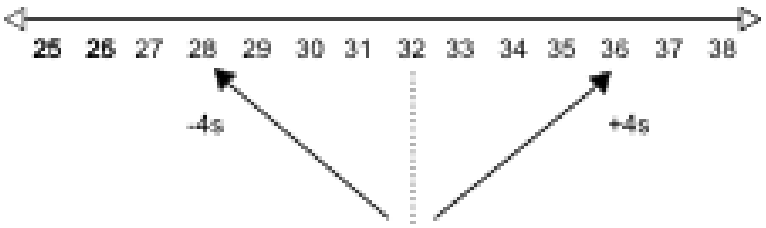

Figura 9: Representação da escolha de mudança de tempo de estágio (SPLIT)
Fonte: (VILANOVA, 2005a)

Na figura 9 pode-se ver o exemplo de um semáforo previsto para mudança de estágio aos trinta e dois segundos. Pouco antes desta mudança, o algoritmo avalia, se antecipar ou atrasar essa mudança em quatro segundos ajudará a equalizar o nível de saturação entre os links que chegam ao nó.

3.2.4.2 Otimizador de Defasagens

O Otimizador de Defasagens (OFFSET) é acionado uma vez por ciclo, e tem um funcionamento bem semelhante ao Otimizador de Tempos de Estágio, porém seu objetivo é avaliar os efeitos de atrasar, avançar o tempo de defasagem entre o ciclo atual e o próximo ciclo do semáforo, ou permanecer na configuração atual.

Ao ser acionado, o Otimizador de Defasagens utiliza o *Perfil de Fluxo Cíclico* para calcular o tamanho das filas de todos os links que chegam e saem de um determinado nó. Com base nessas informações o algoritmo adota uma configuração de defasagem que leva a um menor atraso e quantidade de paradas de veículos que chegam até a fila, em todos os links que levam até o nó.

O atraso de alguns segundos no ciclo de um semáforo em relação a outro pode evitar que os veículos que partiram no estágio de verde do primeiro semáforo precisem ser retidos no estágio de vermelho do segundo semáforo. Graças a defasagem entre os ciclos, o segundo semáforo deverá atingir o tempo de verde com alguns segundos de atraso, na aproximação dos veículos, evitando paradas desnecessárias.

Este componente também leva em consideração o fator de congestionamento em cada link. Vias congestionadas recebem prioridade para obter uma maior taxa de saturação, com sua defasagem entre seu nó-origem e nó-destino recebendo um valor especial.(VILANOVA, 2005a)

3.2.4.3 Otimizador de Tempo de Ciclo

O sistema SCOOT possui uma grande quantidade de váriaveis de configuração, entre elas o nível máximo aceitável de grau de saturação nos links, e o tempo máximo de ciclo para todos os semáforos da rede. De acordo com a documentação do sistema, o SCOOT configura como grau de saturação limite o valor de 90%, mas este valor pode ser alterado de acordo com as necessidades do engenheiros de tráfego.

O Otimizador de Tempos de Ciclo (CYCLE) atua sobre uma região que possui um conjunto de semáforos, todos com o mesmo tempo de ciclo. A cada cinco minutos este otimizador é acionado, e calcula o nível de saturação de todos os links que chegam a cada nó pertencente a essa região. Se pelo menos um destes links exceder o limite de saturação configurado no sistema, o otimizador irá então incrementar este tempo de ciclo em alguns segundos, não podendo este ultrapassar o valor de tempo máximo pré configurado no sistema. Se todos eles estiverem abaixo do limite estabelecido, o tempo de ciclo será

então decrementado em alguns segundos.

Este otimizador passa a operar automaticamente a cada dois minutos e meio, quando o sistema detecta uma grande variação de trânsito (como em horários de pico). O fator de congestionamento não afeta diretamente este componente, por se tratar de uma configuração mais global, trabalhando em conjuntos de semáforos. Por isso apenas os outros dois componentes, Otimizadores de Estágios e Defasagens, atuam diretamente na solução de vias congestionadas.(VILANOVA, 2005a)

4 Algoritmos Genéticos

Algoritmos de busca local são algoritmos que procuram resolver problemas de otimização de forma iterativa, geralmente partindo de um estado inicial (normalmente gerado de forma aleatória) e realizar pequenas modificações em soluções potenciais para o problema até encontrar a solução ótima (máximo global) (COPPIN, 2010).

Os algoritmos genéticos são algoritmos de busca local, que fazem parte do ramo da Ciência da Computação chamado de Computação Evolucionária, uma área de estudo que se baseia nos mecanismo evolutivos encontrados no mundo natural. Este ramo da computação procura simular através de algoritmos os mecanismos relacionados com a Teoria da Evolução de Darwin, que afirma diretamente que a vida e evolução dos seres vivos se deu graças a um processo de seleção, realizado pelo meio ambiente, selecionando somente os indivíduos mais adaptados, tendo dessa forma maiores chances de sobreviver e se reproduzir.

Além dos conceitos da Teoria da Evolução de Darwin, os algoritmos genéticos também fazem uso dos fundamentos da Genética de Gregor Mendel, que define os conceitos de hereditariedade e suas probabilidades, que é a capacidade de um indivíduo transmitir suas características às gerações seguintes.

Unindo essas duas teorias fundamentais da Ciência Biológica, espera-se que os indivíduos selecionados se reproduzam mais e transmitam suas características de boa adaptação para sua geração seguinte, enquanto indíviduos menos adaptados se reproduzam menos, e suas características sejam perdidas geração pós geração, criando assim indívuduos cada vez mais próximos a perfeição em relação a uma qualidade específica. (ARTERO, 2009)

4.1 Aplicabilidade dos Algoritmos Genéticos

Algoritmos Genéticos são frequentemente usados a fim de determinar soluções otimizadas para problemas onde não há um algoritmo conhecido. Para isso parte-se de um

conjunto inicial de dados gerados por um processo estocástico, que são avaliados quanto à satisfação do problema proposto, combinando os dados mais próximos a solução esperada. (ARTERO, 2009)

Em diversas áreas da computação os Algoritmos Genéticos são aplicados com sucesso. Muitos problemas de busca combinatória podem ser resolvidos com eficiência através desta técnica, entre eles, o problema do Caixeiro-viajante, O Problema da Mochila e o problema da Satisfatibilidade FNC. (COPPIN, 2010)

4.2 Nomenclatura e Terminologia

Para ilustrar o conceito e manter o paralelo com a genética e ciência evolutiva biológica, algoritmos genéticos usam terminologia semelhante ao da Biologia para representar seus conceitos, estruturas de dados e procedimentos.

- Genes: Os genes são a representação mínina utilizada e correspondem a algum parâmetro ou característica de interesse ou importância que afeta a solução do problema de otimização. Podem ser representados através de bits (0,1), números reais, inteiros, ou até mesmo cadeias de caracteres, sendo que a representação binária é a mais comum.

- Cromossomos: Cromossomos são cadeias de genes. Podem conter genes de qualquer tipo, desde que o cromossomo possua o mesmo tipo em todos os genes, a fim de manter a compatibilidade dos novos cromossomos gerados durante o processo de cruzamento (ARTERO, 2009).

  - cromossomoA = {1.2;0.1;2.0} - Representação com números reais
  - cromossomoB = {1;3;7}- Representação com números inteiros
  - cromossomoC = {010101010000}- Representação com binários

- Indivíduos: Indivíduos são conjuntos de um ou mais cromossomos, e no contexto dos algoritmos genéticos, representam soluções encontradas para o problema de otimização. Segundo o modelo de John Henry Holland (conhecido como o criador dos Algoritmos Genéticos) cada indivíduo representa um único cromossomo. Porém, para resolução de problemas mais complexos podem combinar vários cromossomos em um único indivíduo, se aproximando ainda mais da genética real (COPPIN, 2010).

- População: Chamamos de população o conjunto de indivíduos (no caso dos algoritmos genéticos, soluções candidatas para o problema de otimização) que irão competir entre si para se reproduzir e dar origem a uma nova geração. Para a primeira iteração do algoritmo, essa população inicial pode ser baseada em uma solução otimizada pré-existente, ou gerada através de um processo estocástico.

- Geração: É chamada de geração uma população em um determinado período de tempo. No contexto dos algoritmos genéticos, é chamada de geração a população obtida a cada iteração do algoritmo.

4.2.1 Função de Avaliação

As funções de avaliação, são fórmulas matemáticas utilizadas para medir e testar a qualidade e nível de aptidão de um determinado indivíduo. Nos algoritmos genéticos são utilizadas para avaliar os melhores indivíduos (soluções propostas para o problema de otimização). Estes indivíduos possuirão uma probabilidade de cruzamento maior, esperando assim que seus genes gerados de uma solução otimizada sejam transmitidos à próxima geração de indivíduos (ARTERO, 2009).

4.2.2 Falha

Um problema conhecido dos algoritmos genéticos é chamado de *falha*. Este problema acontece quando o algoritmo é induzido a descartar um indivíduo aparentemente subótimo, gerado a partir de dois indivíduos considerados com alto nível de aptidão.

Por exemplo, vamos ilustrar os seguintes cromossomos de 8 bits e seus valores de avaliação:

$S_1 = 11****** \ f(S_1) = 50$

$S_2 = ******11 \ f(S_2) = 40$

$S_3 = 11****11 \ f(S_3) = 5$

$S_4 = 00****00 \ f(S_4) = 65$

Os cromossomos S_1 e S_2 são utilizados para gerar o cronossomo S_3, porém o cromossomo S_3 é considerado bem menos apto do que S_1 e S_2.

Se a solução ótima para este problema de otimização for $S_5 = 11111111$ com $f(S_5) = 100$, então dificilmente o algoritmo genético irá convergir para esta solução ótima.

Para evitar este tipo de problema existem técnicas que podem ser aplicadas aos algoritmos genéticos, tais como a *Inversão* e a *Mutação* que diminuem a probabilidade de *Falha* e evita que o algoritmo pare um máximo local, e não converja para o máximo global. (COPPIN, 2010)

4.3 Operadores e Etapas

4.3.1 Visão Geral

A visão geral por trás do algoritmos genéticos é muito semelhante a Teoria da Evolução de Darwin. Consiste basicamente em selecionar os indíviduos mais bem adaptados e dar a eles uma chance maior de sobrevivência e reprodução, esperando que seus genes sejam passados para a futura geração.

Mais especificamente aplicado aos algoritmos, isto significa avaliar cada uma das soluções candidatas para o problema de otimização (indivíduos) através de uma função de avaliação. As soluções melhores avaliadas terão uma probabilidade maior de serem selecionadas para o cruzamento. Cada etapa de seleção e cruzamento equivale a uma iteração do algoritmo, que dará origem a uma nova população de indivíduos.

Para evitar que o algoritmo não converja única e precocemente para um máximo local, uma função de mutação deve ser aplicada a uma probabilidade constante aos indivíduos da nova população gerada.

De forma geral, um algoritmo genético pode ser descrito da seguinte forma:(COPPIN, 2010)

1. Determine a população inicial. Pode ser definida por um conjunto de soluções otimizadas por outros métodos, ou geradas atavés de um processo estocástico
2. Determine o grau de aptidão de cada indivíduo
3. Avalie o críterio de terminação. Caso satisfeito, pare. Caso não, continue para a próxima etapa.
4. Aplique o cruzamento e mutação aos indivíduos selecionados, gerando uma nova população
5. Retorne à etapa número 2.

4.3.2 Seleção

O processo de seleção dos indivíduos mais aptos é iniciada pela sua avaliação. Ao optar e escrever um algoritmo genético é importante encontrar métricas para que a função de avaliação possa examinar corretamente e objetivamente a qualidade de um indivíduo.

Note que em algoritmos genéticos, os indivíduos com baixo nível de adaptação não são descartados do processo de seleção, apenas devem possuir uma probabilidade menor de serem selecionados, de acordo com o método de seleção utilizado.

Os métodos mais comuns de seleção são:

- Seleção Aleatória: neste método de seleção são sorteados aleatoriamente dois indivíduos dentre a população. Desta forma a probabilidade de seleção de cada indivíduo é igual, independente do valor de sua função de avaliação.
- Seleção por Torneio: neste método também são sorteados aleatoriamente dois indivíduos dentre a população. Através da função de avaliação, o indivíduo considerado mais apto é selecionado. O processo é repetido para que seja encontrado o segundo indivíduo que irá compor o cruzamento.
- Método da Roleta: neste método os valores de aptidão de todos os indivíduos é calculado e seus valores irão corresponder a setores proporcionais de uma roleta imaginária. Desta forma, indivíduos com menor aptidão serão representados por setores menores da roleta, tendo portanto uma probabilidade menor de serem selecionados. Pode-se visualizar gráficamente na Figura 10 a apresentação as probabilidades de seleção de cada indivíduo.

$S_1 = 11****** \text{ f}(S_1) = 50$

$S_2 = ******11 \text{ f}(S_2) = 40$

$S_3 = 11****11 \text{ f}(S_3) = 5$

$S_4 = 00****00 \text{ f}(S_4) = 65$

O resultado de um sorteiro aleatório entre os valores 1 e 160 irá determinar o setor da roleta do indivíduo que será selecionado.

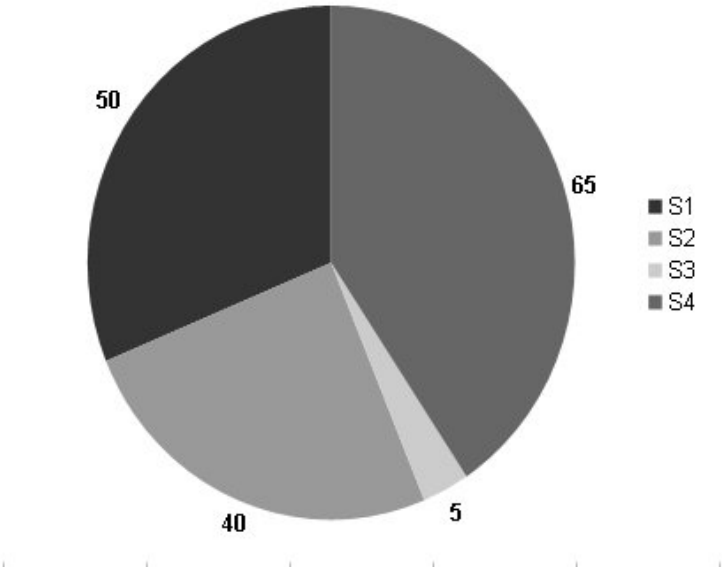

Figura 10: Representação da roleta imaginária para a seleção dos Indivíduos S_1,S_2,S_3,S_4
Adaptado de: (ARTERO, 2009)

4.3.3 Cruzamento

Após o processo de seleção de dois indivíduos entre os elementos da população, é dado início ao processo de cruzamento, que levará à formação de um novo indivíduo, com características de ambos. Este processo de cruzamento origina a uma nova população, com um nova geração de indivíduos, que será utilizada no processo de seleção na próxima iteração do algoritmo.

O Cruzamento é um dos pontos centrais do algoritmo genético, e seu desempenho depende muito desta etapa. Existem várias formas de se realizar o cruzamento, a forma adotada e o tipo de implementação depende diretamente do problema a ser resolvido.

- Ponto de cruzamento único: é definido aleatoriamente um único ponto de cruzamento entre os pais Pai_1 e Pai_2. Os genes são copiados de Pai_1, em sequência desde o primeiro gene até o ponto de cruzamento e de Pai_2 do ponto de cruzamento até o fim do cromossomo.

  $Pai_1 = \underline{11001}100$

  $Pai_2 = 11110\underline{000}$

  $Filho = 11001—000$

- Ponto de cruzamento duplo: são definidos aleatoriamente dois pontos de cruzamentos entre os pais Pai_1 e Pai_2. Os genes são copiados de Pai_1, em sequência desde

o primeiro gene até o primeiro ponto de cruzamento, de Pai_2 do primeiro ponto de cruzamento até o segundo ponto, e do segundo ponto até o fim os genes são copiados novamente de Pai_1.

$Pai_1 = \underline{11}001\underline{00}$

$Pai_2 = 11\underline{110}00$

$Filho = 11—1100—00$

- Cruzamento uniforme: os genes são copiados aleatoriamente de Pai_1 ou Pai_2 diretamente para o filhos.

  $Pai_1 = \underline{1}1\underline{0}0\underline{1}100$

  $Pai_2 = 1\underline{1}1\underline{10}\underline{00}$

  $Filho = 1—1—0—10—1—00$

- Cruzamento aritmético: os genes de Pai_1 e Pai_2 são combinados através de uma função aritmética e o seu resultado transmitidos para os filhos.

  Ex: Cruzamento aritmético utilizando a função OR

  $Pai_1 = 11001100$

  $Pai_2 = 11110000$

  $Filho = 11111100$

4.3.4 Mutação

O processo de mutação é um processo muito importante no algoritmo genético, pois este garante a diversidade genética da população alterando arbitrariamente um ou mais genes dos filhos. Com esta operação, é possível introduzir indíviduos com novas características na população, assegurando que a probabilidade de se chegar a um ponto do universo de busca nunca será zero. Além disso evita que o algoritmo fique estagnado em algum mínimo local, pois a mutação altera de forma sútil a direção da busca.

A mutação não é aplicada a todos os indivíduos resultantes de cruzamento. A mutação de cada indivíduo é determinada por uma taxa de probabilidade de mutação, que geralmente é pequena, para que os valores transmitidos de pais para filhos são sejam distorcidos por frequentes mutações.

Durante o processo, o gene que sofrerá a mutação é determinado de forma aleatória e o seu valor é arbitrariamente invertido: Durante o processo, o gene que sofrerá a mutação é determinado de forma aleatória e o seu valor é arbitrariamente invertido (ARTERO, 2009):

$Filho = \underline{1}1111100$

Após mutação:

$Filho = 01111100$

5 Aplicação de Algoritmos Genéticos em Controle de Semáforos

O objetivo deste trabalho é aplicar os conceitos apresentados no capítulo 4 (Algoritmos Genéticos) aos conceitos dos sistemas apresentados no Capítulo 3 - Softwares de Gerenciamento de Tráfego - a fim de encontrar uma solução otimizada para os tempos de semáforo em um determinado cenário, utilizando a linguagem de programação Python, a API de acesso a dados TraCI *Traffic Control Interface* e o simulador SUMO *Simulator of Urban Mobility*.

Como exposto no Capítulo 4, o objetivo dos algoritmos genéticos é encontrar uma solução máxima global para problemas de otimização. Neste trabalho de pesquisa, o problema de otimização que buscamos resolver é a configuração tempos de semáforos que permitem uma maior fluidez de tráfego de acordo com o fluxo de veículos detectados nas vias.

Contextualizando o problema, percebe-se que cada combinação de tempos de semáforo é equivalente a um indivíduo entre uma população de configurações possíveis. Para avaliar todas essas soluções é necessário o auxílio do simulador SUMO, que aplicará essa solução dentro do ambiente de simulação pré-configurado por um período pré-determinado de tempo. Durante o período de simulação, o software irá através da TraCI API, efetuar a leitura de sensores em tempo de execução, e através de um algoritmo adaptado do software *TRANSYT* (Seção 3.1) irá atribuir um valor de *fitness* para cada um dos indívuos.

A partir desta avaliação, cruzamentos entre as soluções serão realizadas durante um número limite de gerações, a fim de encontrar a solução otimizada para o cenário. Os detalhes de modelagem do cenário e implementação do software são apresentados neste capítulo.

5.1 Ferramentas utilizadas

5.1.1 Simulador SUMO

Para este trabalho foi eleito o SUMO - *Simulator of Urban Mobility* como software responsável por simular o comportamento dos tempos de semáforo para que os indíviduos do Algoritmo Genético possam ser avaliados. O SUMO é um simulador free, opensource, desenvolvido em C++ pelo Instituto de Pesquisas em Transporte do Centro Aeroespacial Alemão (DLR) em parceria com Centro de Informática Aplicada de Colônia. O simulador foi criado com o objetivo de ser uma plataforma para testes de novos produtos e soluções aplicada à modelagem de trânsito e tem grande aceitação na comunidade de simulação microscópica e mesoscópica de engenharia de tráfego.(BEHRISCH, 2011)

A documentação do simulador é bem completa e abrangente, e possui dificuldade moderada para composição de cenários de simulação, através de arquivos XML.

Algumas características importantes do simulador influenciaram na escolha para este trabalho:

- Realiza simulação microscópica
- Suporta diferente tipos de fluxos e velocidade de veículos
- Permite o uso de multiplas faixas por via
- Controle dinâmico de semáforos através da API TraCI
- Leitura dinâmica de sensores de solo
- Rápido desempenho de processamento - gerencia aproximadamente 100.000 veículos/s em uma máquina de 1Ghz (segundo documentação oficial)
- Iteração com outros processos em tempo de execução
- Multi-plataforma: possui pacotes para Windows e Linux
- Possui interface gráfica (GUI) e Interface texto
- Gratuito e OpenSource, distribuído pela GPL

5.1.2 TraCI API

A TraCI (sigla para Traffic Control Interface) (BEHRISCH, 2011) é uma API escrita na linguagem Python, distribuida junto com o simulador SUMO, que permite integração de scripts Python escritos por desenvolvedores com o simulador.

A API fornece métodos que permitem aos desenvolvedores capturar dados e intervir na simulação que está sendo executada em outro processo, em tempo de execução. Essa comunicação é feita pelo protocolo TCP/IP em uma porta previamente configurada no ambiente de simulação.

Entre os métodos disponíveis pela API estão: alterar a configuração das luzes de semáforo em tempo real, efetuar a leitura dos sensores de solo, avançar os passos no ambiente de simulação, entre outros.

5.2 Ambiente de Simulação

A primeira etapa para a produção deste trabalho de pesquisa, é criar um ambiente de simulação onde as possíveis soluções de tempos de verde possam ser testadas e avaliadas. Para isso foi utilizada a documentação disponível do simulador SUMO. Pode-se visualizar na Figura 11 o simulador operando com interface gráfica.

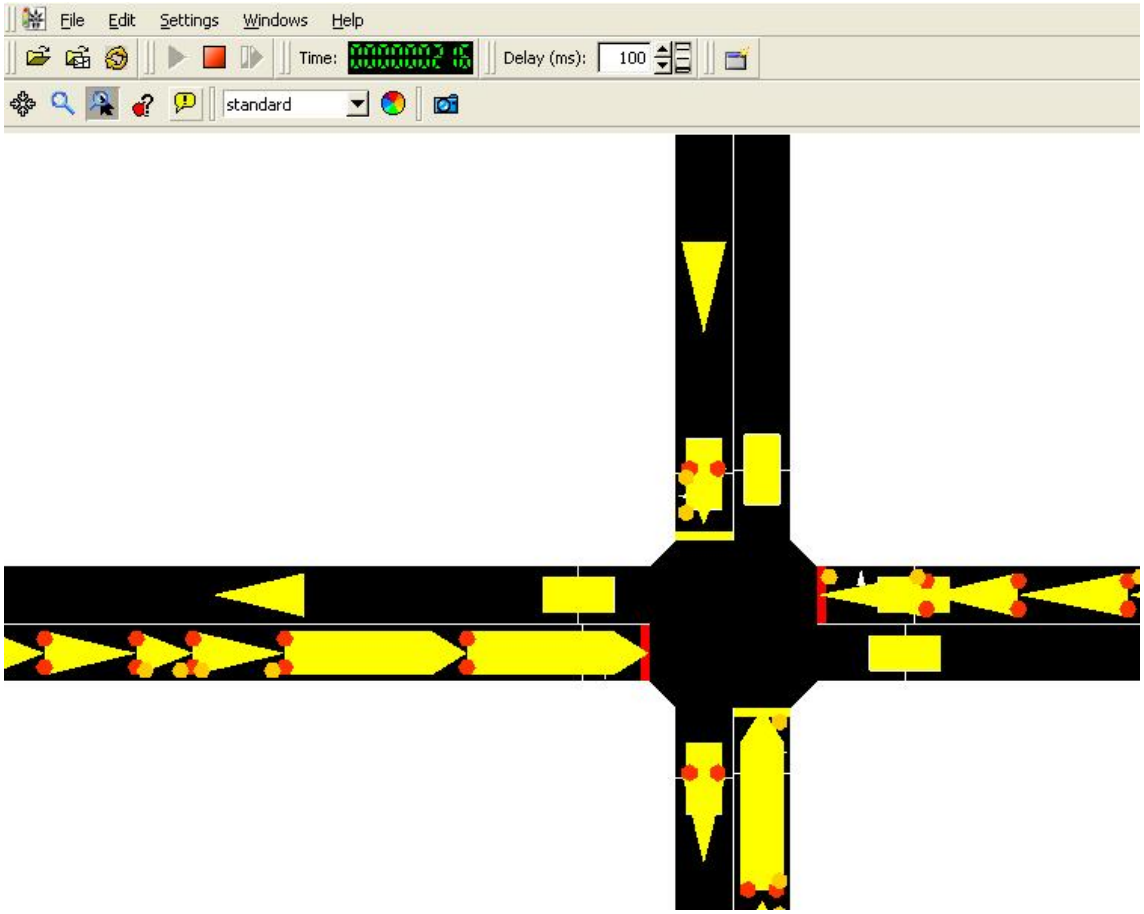

Figura 11: Ambiente do simulador SUMO com interface gráfica

Para a avaliação das possíveis soluções de combinações de tempo de semáforo, foi criado um ambiente de simulação com as seguintes características:

- 3 vias de tráfego de mão dupla, com faixa de rolagem única para cada sentido
- duas intersecções de vias semaforizadas
- simulação de fluxo com 4 tipos de veículos com velocidades diferentes: ônibus, carros lentos, carros médios e carros rápidos, com fluxo adequado a capacidade máxima da via
- tempo de ciclo de semáforo fixo, com tempo de 60 segundos
- tempos de amarelo fixados em 2 segundos entre tempo de verde e tempo de vermelho
- rotas não cíclicas

Para que o ambiente de simulação fosse desenvolvido corretamente, foi criado um modelo conceitual usando grafos, que facilia a visualização das vias como vetores. Essa visualização favorece a criação do ambiente de acordo com a documentação do SUMO. Na Figura 12 pode-se ver este modelo conceitual.

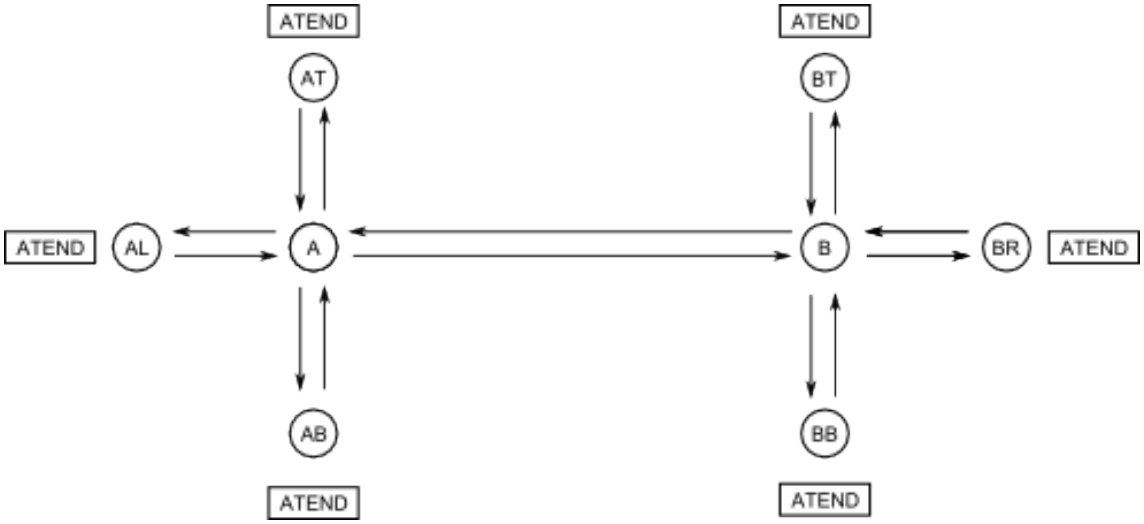

Figura 12: Representação em forma de grafo do ambiente de simulação

Para configuração do ambiente, existem conceitos utilizados pelo simulador, expostos em sua documentação. Estes conceitos são listados abaixo e serão utilizados durante ao decorrer do capítulo:

- *Nodes*: são a representação dos vértices do grafo exposto no modelo conceitual. No ambiente de simulação, esses vétices representam os extremos e a intesecção das vias.

- *Edges*: são a representação das arestas do grafo exposto no modelo conceitual, ligando um vértice a outro. No ambiente de simulação, essas arestas representam um único sentido de uma via. Note que, vias de mão dupla possuem duas arestas com sentidos opostos.

- *Lanes*: *lanes* representam as faixas de rolagem de cada sentido de cada via. Esse elemento não é representado no modelo conceitual. Note que uma única via pode ser composta com até dois edges (com sentidos opostos) e cada edges pode ter até n lanes.

- *Connection*: o elemento connetion representa a forma como os vértices estão interligados. Esse elemento permite definir por quais *Edges* o fluxo de trânsito pode seguir de forma contínua. Por exemplo, pode-se restringir uma conversão proibida entre dois *Edges*.

- *Routes*: o elemento *route* é equivalente às trilhas que se deseja permitir que o tráfego circule de acordo com o modelo conceitual. É formado por um conjunto de *Edges* sequenciais, que representam as vias e os sentidos por onde passará o fluxo de veículos.

5.2.1 Convenção de Nomes

Para facilitar o entendimento e a implementação do algoritmo de função de avaliação, neste trabalho foram utilizados alguns padrões de nomes para identificação dos elementos. Esses padrões foram definidos com o objetivo de tornar o reconhecimento de tipos, posições, direções, etc, de todos os elementos declarados no ambiente de simulação durante a implementação do código. A Tabela 1 exibe esse padrões utilizados.

5.2.2 Configuração das Vias

O ambiente de simulação do SUMO é todo configurado através de arquivos XML, onde são declaradas todas as informações e coordenadas de vias. Essas configurações são distribuídas em diversos arquivos separadamente, que posteriormente são utilizados por um aplicativo do próprio simulador - *netconvert* - que gera um novo arquivo XML contendo todas as informações necessárias para iniciar a simulação.

Os arquivos de configuração utilizados neste trabalho são:

Tabela 1: Convenção de Nomes para elementos do ambiente de simulação

Convenção de Nomes para elementos do ambiente de simulação	
Semáforo	Letras Sequenciais do Alfabeto {A, B,..., X, Z}
Nodes	São declarados em função à posição {Top (T), Bottom (B), Left (L), Right (R)} relativa ao semáforo em que está interligado. Ex: Node posicionado à esquerda do semáforo A: "AL". Os nodes terminais recebem o sufixo END. Ex: Node terminal posicionado à direita do semáforo B: "BREND".
Edges	São declarados com a identificação do node de partida e node de destino, separados por {_} (underline). Por exemplo, o edge que vai do node "AL" até o semáforo "A", recebe o nome "AL_A".
Lanes	São nomeadas automaticamente pelo simulador. Recebem a mesma identificação do Edge ao qual pertence, mais um número inteiro único e sequencial, separados por um "." (ponto). Ex: "AL_A.0","AL_A.1".
Detectores	Recebem o mesmo nome do Edge ao qual pertence, seguido do sufixo "IN" para detectores de entrada, e "OUT" para detectores de saída: Ex: "AL_A_IN" e "BT_B_OUT".
Routes	Seguindo o padrão adotado, os routes recebem os nomes do edge de partida e edge de destino, separados por dois {_} (underline). Por exemplo, uma rota que parte do edge "AL_A" até o edge "A_AT" recebe o nome "AL_A__A_AT".
Tipo de Veículo	Descrição livre do veículo, sem regras de padronização. {CarroLento, CarroRapido}
Flows	O padrão de nome para os fluxos de veículos declarados, concatenado o prefixo FLOW, Tipo de Veículo e rota que usado no fluxo, separados por {_}. Ex. "FLOW_ONIBUS_AT_A__A_AL"
Veículos	São nomeados automaticamente pelo simulador. Recebem a mesma identificação do Flow ao qual pertence, mais um número inteiro único e sequencial, separados por um "." (ponto). Ex: "FLOW_ONIBUS_AT_A__A_AL.0", "FLOW_ONIBUS_AT_A__A_AL.1"

- *tgiv1.nod.xml*: Este arquivo é utilizado para declarar os vértices do modelo de simulação. O elemento "node" do esquema XML é utilizado para essa declaração. São declarados dois tipos de vértice: vértices regulares (padrão) e os vértices que representam uma intersecção semaforizada.

```
<node id="A" x="-100" y="0" type="traffic_light" />
```

  Os parâmetros utilizados neste elementos são:

  - id: identificação única do vértice. Será utilizado posteriormente para referenciar o vértice no restante da configuração
  - x: posição do vértice no eixo X do plano de simulação
  - y: posição do vértice no eixo Y do plano de simulação
  - type: parâmetro opcional "traffic light" que indica que o vértice representa uma intersecção semaforizada.

  Para os vértices regulares, basta omitir o parâmentro *type*:

```
<node id="AL" x="-200" y="0" />
```

- *tgiv1.edg.xml*: Este arquivo é utilizado para declarar as arestas (vetores de direção das vias) do modelo de simulação. O elemento "edge" do esquema XML é utilizado para essa declaração.

```
<edge id="A_B" from="A" to="B" type="ADJ" />
```

  Os parâmetros utilizados neste elemento são:

  - id: identificação única do vetor. Será utilizado posteriormente para referenciar esta aresta no restante da configuração
  - from: vértice de origem
  - to: vértice de destino
  - type: parâmetro opcional, onde é possível definir características específicas para a aresta. Estes tipos podem ser personalizados no arquivo *tgiv1.typ.xml*.

- *tgiv1.typ.xml*: Este arquivo é utilizado para declarar características específicas para as arestas previamente configuradas no arquivo tgiv1.edg.xml. O elemento "type" do esquema XML é utilizado para essa declaração.

```
<type id="ADJ" priority="2" numLanes="1" speed="60" /
    >
```

Os parâmetros utilizados neste elemento são:

- id: identificação única para o tipo de aresta. É referenciada no parâmetro "type" do arquivo tgiv1.edg.xml.
- priority: nível de prioridade deste sentido da via. Este valor é utilizado pelo simulador para priorizar o fluxo de veículos entre as vias, em intersecções sem semáforo.
- numLanes: número de faixas de rolagem neste sentido da via
- speed: velocidade máxima permitida aos veículos nesta via

- *tgiv1.con.xml*: Este arquivo é utilizado para declarar as informações referentes as conexões entre os sentidos das vias (*edges*). Somente as conexões declaradas são permitidas. O elemento "connection" do esquema XML é utilizado para declarar as conexões:

```
<connection from="B_A" to="A_AL" fromLane="0" toLane
    ="0" />
```

Os parâmetros utilizados neste elemento são:

- from: *edge* de origem do veículo.
- to: *edge* de destino do veículo.
- fromLane: faixa de rolagem de origem do veículo.
- toLane: faixa de rolagem de destino do veículo.

- *tgiv1.netc.cfg*: Este arquivo é utilizado pelo aplicativo "*netconvert*" para consolidação do arquivo "*tgiv1.net.xml*", que contém todas as informações relativas às vias do ambiente que será simulado. Aqui é determinada a localização dos arquivos descritos anteriormente, para que o aplicativo possa formar um novo arquivo único que será efetivamente utilizado pelo simulador.

```
<input>
    <edge-files value="tgiv1.edg.xml" />
    <node-files value="tgiv1.nod.xml" />
    <type-files value="tgiv1.typ.xml" />
    <connection-files value="tgiv1.con.xml" />
</input>
<output>
```

```
        <output-file value="tgiv1.net.xml"/>
    </output>
```

O elemento "input" determina o conjunto de arquivos de entrada de informações, enquanto o elemento "output" determina o nome do arquivo consolidado, usado pelo simulador.

5.2.3 Configuração de Fluxo

A configuração de fluxo aqui utilizada é feita com base na observação prévia das vias, sem nenhuma metodologia específica, pois não é este o objetivo deste trabalho. Para este trabalho, foi adotado os seguintes modelos de veículos, expostos na Tabela 2, que irão compor o fluxo total de veículos que transitarão pelas vias:

Tabela 2: Tipos de veículos utilizados no ambiente de simulação

Tipos de veículos utilizados no ambiente de simulação				
ID	F. Aceleração	F. Desaceleração	Comprimento	Velocidade Max
CarroLento	$1.0m/s^2$	$6.0m/s^2$	$6.0m$	40 km/h
CarroMedio	$2.0m/s^2$	$8.0m/s^2$	$5.0m$	60 km/h
CarroRapido	$3.0m/s^2$	$9.0m/s^2$	$3.0m$	100 km/h
Onibus	$0.1m/s^2$	$2.0m/s^2$	$10.0m$	20 km/h

Para declarar estes tipos de veículos no simulador, é utilizado o arquivo *tgiv1.rou.xml*. Neste arquivos são declarados os tipos de veículos, as rotas possíveis e a quantidade de fluxo específica para cada veículo e cada rota do ambiente de simulação:

- *tgiv1.rou.xml*: Este arquivo é utilizado para declarar as informações referente ao fluxo de veículos que serão utilizados na simulação. O elemento "vType" do esquema XML é utilizado para declarar as caracterísicas específicas dos tipos de veículos que serão utilizados no ambiente:

```
<vType id="CarroLento" accel="1.0" decel="6.0"
    minGap="0.2" length="6.0" maxSpeed="40.0" sigma="
    0.0" />
```

  Os parâmetros utilizados neste elemento são:

  - id: identificação única para o tipo de veículo. É referenciada posteriormente na configuração de rotas.
  - accel: fator de aceleração do veículo. Expresso em m/s^2.

- decel: fator de desaceleração do veículo. Expresso em m/s^2.
- minGap: distância mínima entre o veículo e o veículo à sua frente. Expresso em m.
- length: largura total do veículo
- maxSpeed: velocidade máxima permitida para este tipo de veículo. Expresso em km/h.
- sigma: fator de aleatoriedade e imprevisibilidade do veículo. É utilizada para simular a imprevisibilidade e variação do comportamento humano. Neste trabalho, essa variação é mantida em 0.0, para que tenhamos exatamente o mesmo cenário de simulação na avaliação de todas as possíveis soluções de otimização.

Para garantir a configuração heterogênia do fluxo de veículos, este foi modelado a fim de seguir aproximadamente a seguinte proporção da Tabela 3 entre os tipos de veículos:

Tabela 3: Proporção entre diferentes tipos de veículos

Proporção entre diferentes tipos de veículos	
Tipo	Proporção
CarroLento	10%
CarroMedio	50%
CarroRapido	35%
Onibus	5%

O elemento "route" do esquema XML é utilizado para determinar as rotas pré-definidas que serão percorridas pelos veículos na simulação. Essas rotas são definidas por um conjunto de vértices previamente declarados no arquivo *tgiv1.edg.xml*

```
<route id="AT_A__A_AL" edges="AT_A_A_AL_AL_END" />
```

Os parâmetros utilizados neste elementos são:

- id: identificação única para a rota será referenciada posteriormente na declaração de fluxo de veículos.
- edges: grupo de arestas, declaradas em sequência, que serão percorridas pelos veículos durante o tempo de simulação.

O elemento "flow" do esquema XML é utilizado para especificar as características de determinado fluxo de veículos que participarão do ambiente de simulação. O

fluxo é responsável por lançar um determinado número de veículos em determinada rota, em um intervalo de tempo definido.

```
<flow id="FLOW_CARROLENTO_AT_A__A_AL" type="
    CarroLento" route="AT_A__A_AL" period="200" />
```

Os parâmetros utilizados neste elemento são:

- id: identificação única para o fluxo do veículos.
- type: tipo de veículo previamente declarado que será utilizado nesta rota.
- route: rota previamente declarada por onde passarão os veículos definidos neste fluxo. O veículo parte do primeiro *edge* declarado na rota e deixa o ambiente de simulação quando atingir o último *edge* no grupo declarado na rota.
- period: o intervalo de tempo entre os lançamentos de veículos na rota declarada. Expresso em *segundos*

Neste exemplo, é declarado que a cada 200 ciclos de simulação, 1 veículo do tipo "CarroLento" será lançado automaticamente na rota "AT_A__A_AL". Este veículo possuirá uma identificação única (importante para o processamento correto da função de avaliação, Seção 5.3.5) e irá iniciar seu trajeto no primeiro "edge" declarado na rota "AT_A__A_AL", neste caso "AT_A", irá fazer a conversão para percorrer todo "A_AL" e ao atingir "AL_END", o veículo é retirado automaticamente do ambiente de simulação.

5.2.4 Configuração de Laços Detectores

O correto posicionamento dos laços detectores é um fator muito importante para que possamos extrair informações sobre o fluxo de veículos da simulação que está sendo executada. Para este trabalho, o posicionamento dos laços detectores será uma adaptação dos conceitos utilizados pelo *SCOOT* (BRETHERTON, 2011), conforme exposto na seção 3.2.2.1:

- O Detector de contagem de veículos será posicionado o mais longe possível do nó (intersecção semaforizada), porém ainda dentro da via. Este detector fornecerá informações sobre a quantidade de veículos que está entrando na via, em direção ao semáforo.

- Outro detector será posicionado na via, o mais próximo possível do semáfoto. Este detector fornecerá informações sobre a quantidade de veículos que efetivamente sairam da via, em direção a outro link.

No ambiente de simulação os detectores foram posicionados de acordo com a Figura 13.

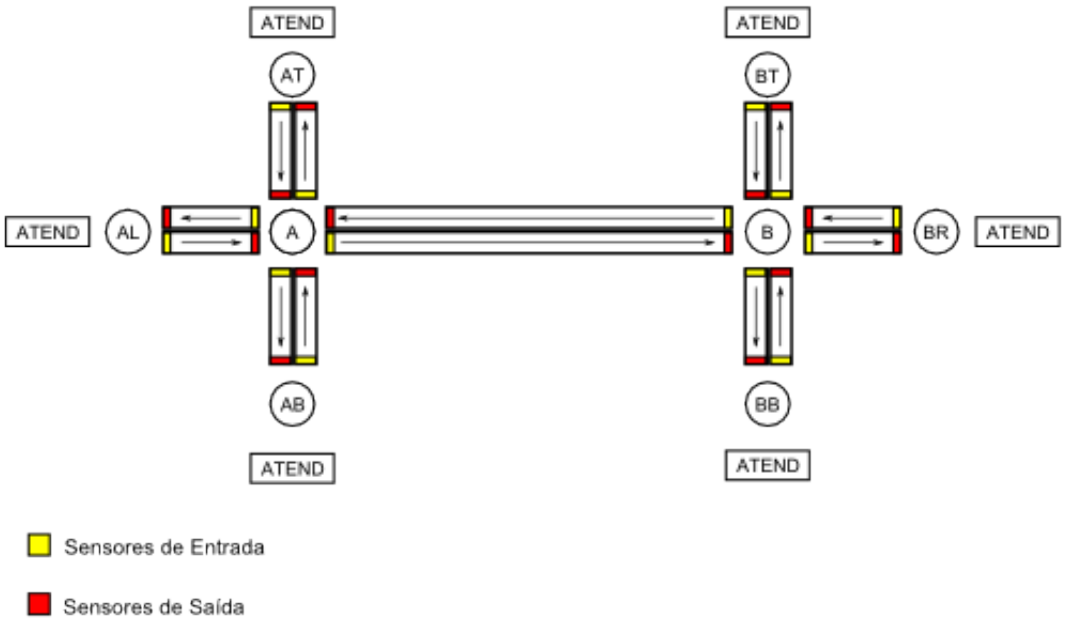

Figura 13: Mapa de localização dos laços detectores

Para declarar estes laços detectores no simulador, utilizamos o arquivos tgiv1.add.xml. Este arquivo é utilizado para declarar qualquer tipo de elemento adicional para o ambiente, incluindo os laços detectores.

- *tgiv1.add.xml*: Arquivo é utilizado para declarar todas as outras configurações adicionais para o cenário de simulação, como laços detectores, programação de semáforos estáticos, etc. Para este trabalho, este arquivo foi utilizado para declaração dos laços detectores de solo, utilizando o elemento "inductionLoop".

```
<inductionLoop id="AT_A_OUT" lane="AT_A_0" pos="90"
    freq="50" file="NUL" />
```

Os parâmetros utilizados neste elementos são:

- id: identificação única do sensor. Este ID é utilizado posteriormente no script Python, para leitura dos veículos e entram e saem de determinada via.
- lano: faixa de rolagem onde será posicionado o sensor.
- pos: a posição em *metros* em relação ao início da via.
- freq: frequência de leitura do sensor.
- file: nome do arquivo de relatório do sensor, gerado automaticamente pelo simulador. O valor NUL informa que este arquivo será ignorado.

5.2.5 Configuração Global do Ambiente de Simulação

- *tgiv1.sumo.cfg*: Arquivo de configuração global do ambiente de simulação, utilizado pelo simulador para unir a configuração das vias, fluxos e integração com aplicativos de terceiros, como a TraCI API.

```
<configuration>
    <input>
        <net-file value="tgiv1.net.xml"/>
        <route-files value="tgiv1.rou.xml"/>
                <additional-files value="tgiv1.add.xml"/>
    </input>
```

O elemento "input" determina os arquivos de configuração do ambiente para vias, rotas e elementos adicionais.

```
    <time>
        <begin value="0"/>
        <end value="10000"/>
    </time>
```

O elemento "time" é usado para determinar o início e fim do período de simulação. Estes valores serão ignorados, caso o controle da simulação seja feito através da TraCI API.

```
    <traci_server>
        <remote-port value="8813"/>
    </traci_server>
</configuration>
```

O elemento “traci server” é usado para determinar as configurações da comunicação entre o simulador e a TraCI API. O elemento “remote port” é usado para configurar a porta TCP/IP de configuração.

5.3 Implementação do Algoritmo Genético

A implementação do Algoritmo Genético foi realizada utilizando a linguagem de programação Python, com estruturas de dados simples e disponíveis na biblioteca padrão da linguagem, utilizando recursos do paradigma imperativo e funcional. A coleta de dados sempre que possível é armazenada em listas, a fim de reduzir a complexidade do algoritmo e aumentar a performance de execução.

A estruta de arquivos de implementação contém os seguintes arquivos:

- tgiv1.py: Classe principal do software onde são instâncias as classes do argoritmo genético.
- config.py: Classe que contém todas as constantes de configuração do sistema.
- guy.py: Contém a classe Guy, que representa um indivíduo único na população.
- fitness.py: Contém a classe Fitness, que realiza a função de avaliação em cada um dos indíviduos do algoritmo.
- population.py: Classe que representa uma população, que agrega um conjunto de n indivíduos.

5.3.1 Representação dos Indivíduos

Cada possível solução para os tempos de semáforo representa um indivíduo entre um conjunto finito de indivíduos de uma população. Neste projeto, a classe Guy representa uma única possível solução com tempos de verde e vermelho para cada semáforo do ambiente de simulação.

Cada indivíduo possui um cromossomo para cada semáforo presente na representação do cenário. Estes cromossomos são cadeias de bits de comprimento N, sendo:

- N = tamanho de ciclo do semáforo (tempo total onde o semáforo passa pelo tempos verde, amarelo, vermelho e retorna para o tempo de verde, pré configurado na classe Config.)

- bit de valor 1 = representa um segundo de tempo de fase de verde na configuração de tempos do semáforo. Ou seja, uma cadeia com N bits de valor 1, representa um cromossomo com N segundos de tempo na fase de verde.
- bit de valor 0 = representa um segundo de tempo de fase de vermelho na configuração de tempos do semáforo. Ou seja, uma cadeia com N bits de valor 0, representa um cromossomo com N segundos de tempo na fase vermelho.

Note que um único cromossomo representa a configuração para uma intersecção entre duas vias, que chamaremos de via A e B. Os tempos de verde e vermelho para a via B são exatamente o inverso dos tempos da A, sendo que o tempo de ciclo é fixo. Por ex:

Um indivíduo possui o seguinte cromossomo para a configuração de um semáforo: 11011011011011011011011011011011011011011011011011011011

Quantidade de bits de valor 1: 38 Quantidade de bits de valor 0: 18

No ambiente de simulação, este indvivíduo irá compor a seguinte configuração para a o semáforo:

Tempo de verde para a via A: 38 segundos Tempo de vermelho para a via A: 18 segundos Tempo de amarelo para a via A: 2 segundos (fixo)

Tempo de verde para a via B: 18 segundos Tempo de vermelho para a via B: 38 segundos Tempo de amarelo para a via B: 2 segundos (fixo)

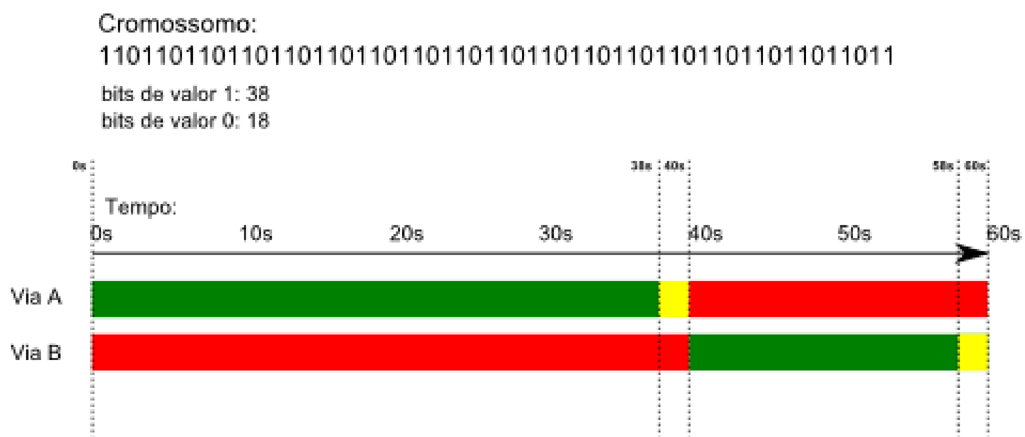

Figura 14: Diagrama de tempos de verde para vias A e B representada por 1 cromossomo.

Conforme pode ser visto na Figura 14, durante os primeiros 38 segundos do tempo de semáforo, a semáforo estará verde para a via A, e vermelho para a via B. Nos segundos 39 e 40, a via B permanece com luz vermelha, enquanto a via A tem luz amarela. Nos segundos 40 a 58, a luz verde será acesa para a via B e vermelha para a via A. Nos segundos 59 e 60, a via A permanece com luz vermelha, enquanto a via B tem luz amarela, e então o ciclo recomeça.

Cada indíviduo é representando por uma estrutura de dados na Classe Guy. Há basicamente dois atributos:

- *lights*: o atributo *lights* é uma lista de listas que contém as configurações de tempos de semáforos para cada uma das intersecções semaforizadas. Cada uma dessas listas possuem os seguintes elementos (em ordem): (cromossomo, tempo de verde, tempo de vermelho). "Cromossomo" é do tipo *string*, enquanto os outros dois elementos são números *integers*, usados para evitar que o cromossomo precise ser percorrido toda vez que for necessário encontrar os valores das configurações, o que diminuiria a performance do software.
  Ex: [["110110110110110110110110110110110110110110110110110110110011", 38, 18],
  ["110110110110110110110110110110110110110110110110110110110011", 30, 16]]

- *score*: o atributo *score* é um valor *float* e representa o valor de aptidão do indivíduo. Esse valor é setado pela função de avaliação, após a configuração do indivíduo ser testada no simulador.

5.3.2 Representação da População

É chamada de população a representação de um grupo de possíveis soluções para o problema de otimização. Esse grupo de soluções possuem indivíduos com a estrutura semelhante, mas com valores diferentes entre si. Neste trabalho, a população é composta por um grupo de N (previamente configurado) indivíduos da classe Guy.

A população é representada pela classe Population (population.py), responsável por aplicar a maior parte dos operadores genéticos à população, como a seleção de indivíduos, cruzamento, criação de uma nova geração e aplicação das políticas de Elitismo.

A população é representanda por uma estrutura de dados na classe Population. Há basicamente três atributos:

- *members*: o atributo *members* é uma lista de indíviduos (instâncias da classe Guy).

Esta lista representa os indivíduos da geração atual.

- *nextgeneration*: o atributo *nextgeneration* é mais uma lista de indíviduos, desta vez representando os indivíduos da próxima geração.

- *generation*: o atributo *generation* é número *integer* usado como contador para a quantidade de gerações que já foram evoluídas pelo algoritmo.

5.3.3 Parâmentros Genéticos

Todos os paramentos genéticos e configurações de comportamento do algoritmo estão previamento configurados na classe Config (config.py) que guarda o valor de todas as constantes utilizadas pelo software, além de realizar a configuração prévia do simulador sumo (que é uma aplicação indepentente, portanto é utilizada através de um outro processo).

Essas constantes de configuração de logs podem ser observados na Tabela 4:

Tabela 4: Constantes de Configuração do Sistema de logs

Constantes de Configuração do Sistema de logs	
PORT	Número da porta TCP/IP que será utilizada para comunicação entre o algoritmo e o simulador SUMO.
DEBUG	{True — False} Liga ou desliga o output de mensagens de debug durante a execução do algoritmo.
LOG	{True — False} Liga ou desliga a geração de arquivos com os logs a cada ciclo dos semáforos.
DEEPLOG	{True — False} Liga ou desliga a geração de logs "profundos". Estes logs são gravados a cada passo da simulação.
GENERATIONLOG	{True — False} Liga ou desliga a geração de logs com informações do algoritmo a cada geração criada.
SPEEDLOG	{True — False} Liga ou desliga a geração de logs com informações sobre o tempo de execução do algoritmo.
LOGFILE, DEEPLOGFILE, SPEEDLOGFILE, GENERATIONLOGFILE	{String} Nome do arquivo do respectivo arquivo de log que será gravado na pasta do algoritmo. Ex: "deeplog.log"

As constantes de configuração que definem regras de negócio do sistema podem ser observados na Tabela 5:

E as constantes de configuração dos parâmentros dos algoritmos genéticos logs podem ser observados na Tabela 6:

Tabela 5: Constantes de Configuração de Regras de Negócio

Constantes de Configuração de Regras de Negócio	
JAMDETECTION	{Integer Padrão: 25} Quantidade de ciclos necessários com o mesmo veículo posicionado sobre um detector, para que a via seja considerada congestionada.
JAMPENALTY	{Integer Padrão: 1}Fator de penalidade aplicada ao nível de saturação de uma via congestionada.
YELLOW_TIME	{Integer Padrão: 2} Tempo pré-fixado para a fase em amarelo do semáforo.
MINFASETIME	{Integer Padrão: 1} Quantidade de ciclos mínimos para cada fase do semáforo (evita soluções degenerativas com tempo de verde zero)
SIMULATION_TIME	{Integer Padrão: 720} Quantidade total de ciclos de simulação para cada indivíduo.
CYCLE_TIME	{Integer Padrão: 60} Quantidade de ciclos total para cada fase de semáforo.
OFFSET	{Integer Padrão: 0}Atraso entre os ciclos do semáforo. (veja Seção 3.2.4.2)
IGNORECYCLES	{Integer Padrão: 2}Quantidade de ciclos iniciais que serão ignorados, para que os valores de saturação com as vias vazias não deturpem a avaliação final do indivíduo.
LIGHTSID	{List}Lista com os IDs dos semáforos previamento declarados no ambiente de simulação.
DETECTORSID	{List}Lista com os IDs dos edges que possuem detectores, previamente declarados no ambiente de simulação.

Tabela 6: Constantes de Configuração de Parâmentos Genéticos

Constantes de Configuração de Parâmentos Genéticos	
MAXPOPULATION	{Integer Padrão: 20} Número de indíviduos que compõem uma população.
MAXGENERATIONS	{Integer Padrão: 10} Número máximo de gerações que o algoritmo irá criar, caso ainda encontre o indivíduo mais adaptado.
MUTATIONGENERATE	{Float Padrão: 0.05} Porcentagem máxima de genes que poderão ser alterados aleatoriamente durante a mutação.
CREATIONMETHOD	{0—1 Padrão: 1} Seleciona o algoritmo de inicialização de indivíduos.
SELECTIONMETHOD	{0—1 Padrão: 1} Seleciona o algoritmo de seleção de casais de indivíduos.
PERFECTRATE	{Float Padrão: 0.8} Porcentagem de desempenho (fitness) que um indivíduo precisa ter para ser considerado o mais adaptado, em relação a solução ótima.

5.3.4 Operadores Genéticos

5.3.4.1 Inicialização de Indivíduos

A inicialização de indivíduos aleatórios é muito importante para o algoritmo genético, pois a forma como esses indivíduos são criados, pode afetar a cobertura da busca da solução otimizada. Quanto maior a cobertura dessas soluções maiores são as chances do algoritmo convergir para a solução otimizada.

O processo de inicialização de indíviduos é realizado no momento da criação de uma nova instancia da classe Population, que recebe em seu construtor o parâmetro opcional *brandnew* que assume o valor *True* caso seja omitido.

```
1 class Population:
2 def __init__(self, brandnew=True):
3  if(brandnew):
4   for i in range(0, self.config.MAXPOPULATION):
5    self.members.append(guy.Guy())
6  self.orderYourGuys()
```

Construtor da classe *Population*

Seu construtor cria um número N (previamento definido na classe de Configuração, constante MAXPOPULATION) de instâncias da classe Guy e as armazena na lista *members*. Neste momento, a classe Guy é instanciada sem nenhum parâmetro para construtor, apesar de poder receber os seguintes parâmetros que alteram a forma como o indivíduo é criado:

```
1 class Guy:
2 def __init__(self, autoFitness=True, randomize=True, mother=
    False, father=False):
```

Assinatura do método construtor da classe *Guy*

- *autoFitness*: esse parâmetro determina se o indivíduo deverá ou não chamar automaticamente sua função de avaliação no momento da sua criação.
- *randomize*: esse parâmetro determina se o indivíduo é criado aleatoriamente, ou é fruto de um cruzamento entre outros dois individuos.

- *mother* e *father*: nestes dois parâmetros o construtor recebe outros dois objetos desta mesma classe, que serão utilizados para cruzar os cromossomos e gerar este novo indivíduo.

Os membros da primeira geração do algoritmo é composta por indivíduos com cromossomos gerados aleatoriamente:

```
1  class Guy:
2  def __init__(self, autoFitness=True, randomize=True, mother=
       False, father=False):
3   self.lights = [[],[]]
4   for i in range(0, len(self.config.LIGHTSID)):
5    if(randomize):
6     self.lights[i].append(self.generateCromossome())
7    else:
8     self.lights[i].append(self.crossCromossome(mother.
         getCromossome(i), father.getCromossome(i)))
```

Construtor da classe *Guy*

No laço *for* do trecho acima, percebe-se que o código irá adicionar à lista de informações sobre os semáforos (*lights*), um cromossomo gerado aleatoriamente pelo método *generateCromossome()* para cada semáforo declarado previamente na classe *Config*.

Existem dois algoritmos de inicialização dos indivíduos disponíveis, que serão selecionados através do valor da variável CREATIONMETHOD da classe Config:

```
1   def generateCromossome(self):
2    cromossome = str()
3     max_green_time = self.config.CYCLE_TIME - (self.config.
         YELLOW_TIME * 2) - (self.config.MINFASETIME)
4     total_fase_time = self.config.CYCLE_TIME - (self.config.
         YELLOW_TIME * 2)
5
6     if(self.config.CREATIONMETHOD==0):
7      for i in range(0, max_green_time):
8       rand1 = random.randrange(0, 100)
9       rand2 = random.randrange(0, 100)
10      if(rand2 < rand1):
```

```
11          cromossome += str("1")
12        else:
13          cromossome += str("0")
```

Método *generateCromossome()* da classe *Guy*

Nas linhas 3 e 4, são declaradas as variavéis *max_green_time* e *total_fase_time*. A variavél *max_green_time* é referente ao tempo máximo de verde que uma fase pode ter (para evitar que uma das vias fique infinitamente em tempo de vermelho e amarelo) e *total_fase_time* determina o tempo total de ciclo, descontados os tempos de amarelo.

Caso a configuração esteja setada em *CREATIONMETHOD*==0, o algoritmo de inicialização do indivíduo será processado da seguinte forma:

Para cada gene do cromossomo, entre 0 e *total_fase_time*, é realizado o seguinte procedimento:

- é sorteado um primeiro número aleatório *rand1* que pode ter valor entre 0 e 100.
- é sorteado um segundo número aleatório *rand2* que pode ter valor entre 0 e 100.
- Se *rand2* é menor que *rand2*, então o valor do gene será 1. Caso contrário será 0

```
14  elif(self.config.CREATIONMETHOD==1):
15   ones = random.randrange(0, max_green_time)
16   positions = []
17   for i in range(0, ones):
18    pos = random.randrange(0, total_fase_time)
19     while(pos in positions):
20      pos = random.randrange(0, total_fase_time)
21     positions.append(pos)
22
23     for i in range(0, total_fase_time):
24      if(i in positions):
25       cromossome += str("1")
26      else:
27       cromossome += str("0")
28
29  return cromossome
```

Segundo Método de Criação de Indivíduos

Caso a configuração estiver setada em *CREATIONMETHOD*==1, o algoritmo de inicialização do indivíduo será processado da seguinte forma:

- é sorteado um primeiro número aleatório *ones* que pode ter valor entre 0 e *max_green_time*, que é o tempo máximo de verde. Este valor irá determinar a quantidade de bits de valor 1 do cromossomo.
- é inicializada uma lista vazia *positions*. Essa lista irá armazenar as posições dos bits de valor 1, sorteadas aleatoriamente.
- Na linha 17, há um laço *for* que incrementa a variável i de 0 até o valor de *ones*. Ou seja: para cada bit de valor 1, vamos sortear aleatoriamente uma posição na variável *pos* que pode ser de 0 até *total_fase_time*.
- Na linha 19, o laço *while* repete o sorteio aleatório da posição até que essa posição ainda não tenha sido sorteada. Encontrada essa posição inédita, ela é adicionada à lista de posições.
- Na linha 23, há o laço *for* que efetivamente preenche o cromossomo. Para cada gene do cromossomo é verificado se a sua posição está entre as que possuirão bit de valor 1. Caso esteja, a posição é preenchida com valor 1, caso não, valor 0.

Apesar de ter um custo computacional mais elevado, nos testes realizados, o segundo método de criação apresentou uma cobertura maior das possíveis soluções, ao contrário do primeiro método que não apresentou uma variação de soluções mais pulverizada. Por isso este método será usado como padrão para este trabalho.

5.3.4.2 Seleção de Individuos

A seleção de indivíduos é um operador muito importante para o algoritmo genético, porque precisa oferecer uma probabilidade maior de um individuo melhor avaliado se transmitir seus genes às gerações posteriores. Neste trabalho, esta seleção será feita através do método da Roleta (Capítulo: refcap:algoritmos-geneticos, Seção 4.3.2) utilizando os valores de *fitness* e os valores de *ranking*.

Para demonstrar os métodos de seleção, considera-se a seguinte população de 10 indivíduos com seus respetivos *scores* de aptidão, conforme Tabela 7.

```
1  def selectACouple(self):
2   candidates = copy.copy(self.members)
3   couple = []
4
5   if(self.config.SELECTIONMETHOD==0):
6         for x in [0,1]:
7     roullete = selector = float(0)
8     for i in range(0, len(candidates)):
9      roullete += candidates[i].getScore()
10
11    rand = random.randrange(0, int(roullete))
12
13    if(self.config.DEBUG):
14     print "MAX VALUE FOR ROULLETE: %d"%(roullete)
15     print "RANDOM VALUE FOR ROULLETE: %d"%(rand)
16
17    for i in range(0, len(candidates)):
18     selector += candidates[i].getScore()
19     if(rand > selector):
20      continue
21     else:
22      if(self.config.DEBUG):
23       print "\nSELECTED GUY IS: "
```

Tabela 7: Individuos e valores de Fitness

Individuos e valores de Fitness	
0	3.256836
1	3.316360
2	3.462391
3	3.784341
4	3.821107
5	3.989314
6	4.044156
7	4.149112
8	4.194754
9	4.221662

```
24            print candidates[i].introduceYourself()
25
26            couple.append(candidates.pop(i))
27            break
```

Método *selectACouple()* da classe *Population*

No método da roleta utilizando os valores de *Fitness*, primeiro é somado todos os valores de *Fitness* dos indivíduos para se obter o valor máximo da roleta. Essa soma pode ser verificada nas linhas 4 e 5 do código. Em seguida, é sorteado aleatoriamente um valor em *rand* que pode ter o valor de 0 até o valor total das somas de *Fitness* de todos os indivíduos. A Figura 15 exibe a representação gráfica das probabilidade de seleção utilizando este método. Cada região do gráfico representa a probabilidade do indivíduo ser selecionado, e os valores representam seus respectivos valores de *Fitness*.

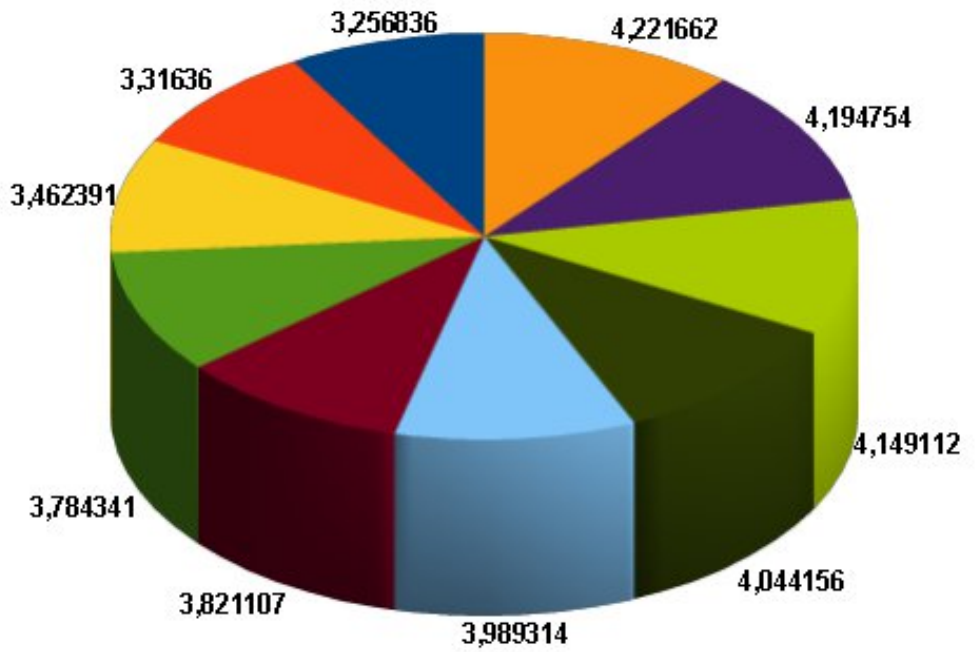

Figura 15: Representação gráfica da roleta utilizando valores de Fitness

Em seguida, novamente a lista de indivíduos é percorrida realizando a soma de valores, até que o valor sorteado em *rand* seja maior que a soma até aquele indivíduo. Quando isso ocorre, significa que chegamos a faixa de valores onde se encontra o número sorteado, e então o indivíduo é selecionado.

Notamos que quanto maior o valor de aptidão do indivíduo maior a a faixa de valores que ele ocupará na roleta, aumentando as suas chances de seleção para cruzamento.

Porém, entre indivíduos de valores de Fitness muito semelhante, como na tabela apresentada, as faixas de valores podem apresentar uma variação muito pequena, não privilegiando adequadamente os indivíduos mais adaptados.

Nestes casos, uma alterativa é o uso dos valores de *ranking* para o método da roleta. Para este método é necessário que a lista de indivíduos esteja ordenada de forma crescente. O valor utilizado para compor a roleta será um valor de *ranking*, calculado da seguinte forma (JADAAN, 2008):

$$ranking = posicao * 2$$

```
28  elif(self.config.SELECTIONMETHOD==1):
29   for x in [0,1]:
30    roullete = selector = int(0)
31    for i in range(1, len(candidates)):
32     roullete += i*2
33
34    rand = random.randrange(1, roullete)
35
36    for i in range(1, len(candidates)):
37     selector += i*2
38     if(rand > selector):
39      continue
40     else:
41      if(self.config.DEBUG):
42       print "\nSELECTED GUY IS: "
43       print candidates[i].introduceYourself()
44
45      couple.append(candidates.pop(i))
46      break
47
48  return couple
```

Segundo Método de Seleção de Indivíduos

Como é possível perceber através da figura 16, usando este método de seleção, os indivíduos com valor de *Fitness* mais elevado, possuem muito mais chances de serem

selecionados do que os primeiros indivíduos da lista, mesmo que seu valor de *Fitness* não seja tão superior aos demais. Além disso, o primeiro indivíduo da lista, considerado o menos adaptado, não terá nenhuma chance de ser selecionado.

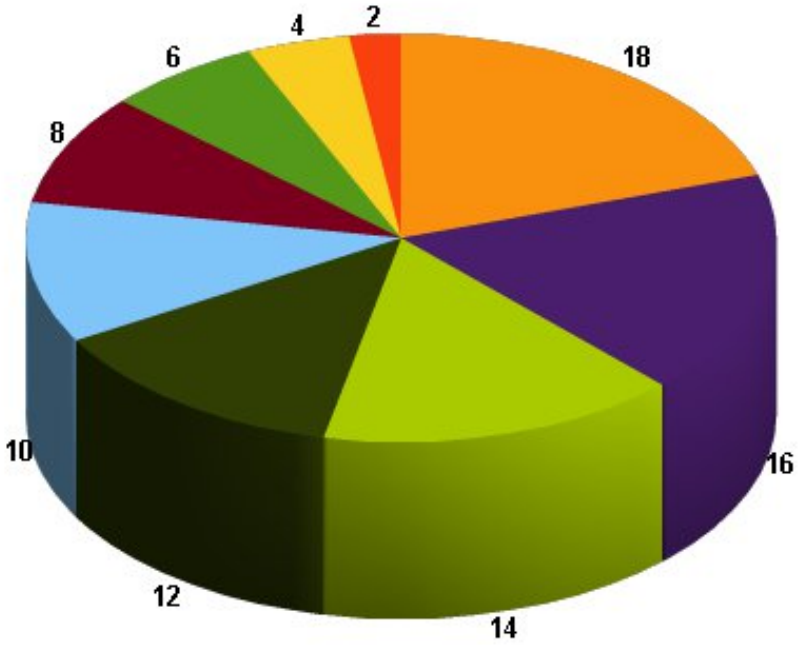

Figura 16: Representação gráfica da roleta utilizando valores de Ranking

Após a seleção do indivíduo, o mesmo é inserido na lista *couple* que é o retorno do método.

Tabela 8: Individuos e valores de ranking

Individuos e valores de ranking		
Posição	Fitness	Ranking
0	3.256836	0
1	3.316360	2
2	3.462391	4
3	3.784341	6
4	3.821107	8
5	3.989314	10
6	4.044156	12
7	4.149112	14
8	4.194754	16
9	4.221662	18

5.3.4.3 Cruzamento de Indivíduos

O objetivo do cruzamento é transmitir os genes de dois indivíduos bem adaptados para um novo indivíduo formado por genes de ambos, a fim de gerar um indivíduo novo mais adaptado que seus pais. Neste trabalho, o cruzamento dos indivíduos é chamado já no construtor da classe *Guy* que recebe opcionalmente dois objetos desta mesma classe, conforme demonstrado na Seção 5.3.4.1.

```
1 for i in range(0, len(self.config.LIGHTSID)):
2   if(randomize):
3     self.lights[i].append(self.generateCromossome())
4   else:
5     self.lights[i].append(self.crossCromossome(mother.
        getCromossome(i), father.getCromossome(i)))
```

Trecho do construtor da classe Guy

Conforme o código acima, quando o objeto Guy é instanciado sem o parâmetro *randomize*, seus cromossomos não são gerados através do método de Inicialização *generateCromossome()*, mas sim pelo método de Cruzamento *crossCromossome()*.

O método *crossCromossome()* recebe dois parâmetros, que são os cromossomos do objetos *father* e do objeto *mother*. Esses cromossomos são obtidos através do método *getCromossome()* que retorna a *String* pura do Cromossomo, do semáforo i (índice do laço que percorre a lista de semáforos *LIGHTSID*.

```
1 def getCromossome(self, i):
2  return self.lights[i][0]
```

Método getCromossome() da classe Guy

Esses cromossomos são passados para o método *crossCromossome()* que fará o cruzamento das duas cadeias de genes resultando em um novo cromossomo, com características dos dois objetos pais. Neste trabalho, o método de cruzamento utilizado é o de Ponto de Cruzamento Único (Seção 4.3.3)

```
1 def crossCromossome(self, cromofather, cromomother):
2   div = random.randrange(0, self.config.CYCLE_TIME)
3   return cromofather[:div] + cromomother[div:]
```

Método crossCromossome() da classe Guy

O método recebe os dois cromossomos dos pais, e sorteia aleatoriamente o ponto de cruzamento que pode ser de 0 até o tamanho total do cromossomo. O método retorna então um novo cromossomo com os genes das posições de 0 até o ponto de cruzamento oriundos do cromossomo *cromofather*, e do ponto de cruzamento até o fim do cromossomo, os genes são oriundos do cromossomo *cromomother*.

5.3.4.4 Mutação

O operador de Mutação tem o objetivo de alterar aleatoriamente alguns genes de indivíduos que foram criados a partir de um cruzamento, a fim de evitar que por conta da não cobertura dos genes de uma possível solução, o algoritmo fique paralisado em um máximo local, e não converja para o máximo global.

A taxa de mutação é um fator muito importante dentro deste contexto e precisa ser cuidadosamente estipulado: uma taxa de mutação muito alta pode tornar o processo evolutivo essencialmente aleatório (desvalorizando a carga genética herdada de soluções anteriores), enquanto uma taxa muito baixa pode tornar a convergência para um máximo global mais lenta.

Para este trabalho, duas constantes são utilizadas para o processo de Mutação, ambos declarados préviamente na classe Config: MUTATIONRATE (probabilidade de um indivíduo sofrer mutação) e MUTATIONCHANGE (porcentagem máxima de genes que serão aleatoriamente alterados durante a mutação).

Durante o processo de criação do objeto *Guy*, é sorteado um número aleatório para determinar se o indivíduo sofrerá mutação, conforme o código abaixo:

```
1 mutate = (float(random.randrange(0, 100))/100)
2 if(mutate <= self.config.MUTATIONRATE):
3  self.mutateMyself()
```

Trecho do construtor da classe Guy

Caso o valor sorteado estiver dentro da probabilidade definida em MUTATIONRATE, o método *mutateMyself()* é chamado diretamente pelo construtor:

```
1 def mutateMyself(self):
2  for x in range(0, len(self.config.LIGHTSID)):
3   genesToMutate = random.randrange(0, int(len(self.lights[x][0])
        * self.config.MUTATIONGENERATE))
```

```
4     for i in range(0, genesToMutate):
5       position = random.randrange(0, len(self.lights[x][0]))
6       newgene = str((int(self.lights[x][0][position]) + 1) % 2)
7       self.lights[x][0] = self.lights[x][0][:position] + newgene +
            self.lights[x][0][position+1:]
```

Método mutateMyself() da classe Guy

Para cada cromossomo do indivíduo, o processo de mutação acontece da seguinte forma:

- Na linha 5, é sorteada aleatoriamente a quantidade de genes que serão mutados, de 0 até a quantidade máxima declarada em MUTATIONGENERATE.
- Na linha 6, o valor do gene é invertido, obtendo o resto do valor do gene adicionado do valor de +1 (na base decimal).
- Em seguida o cromossomo é sobrescrito com o gene modificado.

5.3.5 Função de Avaliação

A função de avaliação é parte determinante do Algoritmo Genético, pois é a responsável por determinar o grau de adaptação do indivíduo dentro das condições impostas pelo ambiente. Neste trabalho, para determinar o grau de aptidão de um indivíduo, iremos utilizar o simulador SUMO.

As características do indivíduo serão testadas durante um tempo específico de simulação, e durante este período o ambiente é monitorado para medir a fluidez do tráfego sob essas condições.

A fórmula utilizada para medir a qualidade do indivíduo, é adaptada no algoritmo *TRANSYT* conforme visto na Seção 3.1:

A função objetivo - Índice de Performance (IP) é definida por:

$$\text{IP} = \sum_{i=1}^{n} (D_i + KS_i).$$

Sendo:

- n = número de vértices da malha representada

- Di = valor médio atraso total no vértice i
- K = fator de penalidade de parada
- Si = número médio de paradas no vértice i

Em nosso cenário de simulação, todas as vias possuem a mesma prioridade em relação ao fluxo de tráfego, portanto o fator de penalidade será descartado. Para simplificar o modelo de desempenho, usaremos como grau de avaliação de cada vértice segundo os princípios de (VILANOVA, 2005b):

- O grau de saturação de um link é 100% (x=1), quando todos os veículos que pararam durante a fase de vermelho do semáforo, conseguiram seguir na fase de verde.
- O grau de saturação de um link é 50% (x=0.5), o tempo de verde do semáforo fosse suficiente para que o dobro de veículos que pararam no semáforo seguissem durante a fase de verde.
- O grau de saturação de um link é 150% (x=1.5), quando somente metade dos veículos que pararam durante a fase de vermelho conseguiram seguir durante a fase de verde, ficando metade para o próximo ciclo.

Para facilitar a avaliação do indivíduo através da função de avaliação, neste trabalho a fórmula de (VILANOVA, 2005b) foi adaptada de forma que os indivíduos com melhor fluidez de tráfego obtivessem um valor de saturação maior, da seguinte forma:

$$\text{Sat} = (VecIn/VecOut).$$

Sendo:

- VecIn = número de veículos que pararam no semáforo durante o tempo de vermelho.
- VecOut = número de veículos que seguiram para fora do link durante o tempo de verde.

Desta forma, quanto maior o número de veículos retidos no semáforo, menor será o valor de avaliação deste link.

Unindo estes conceitos à formula de Índíce de Performance do *Transyt*, chegamos a fórmula usada na função de avaliação, aplicada a cada ciclo:

$$\mathrm{AC} = \sum_{i=1}^{n}(VecIn_i/VecOut_i).$$

Sendo:

- AC: avaliação do ciclo
- n: número de vértices da malha representada
- $VecIn_i$: número de veículos que pararam no semáforo durante o tempo de vermelho.
- $VecOut_i$: número de veículos que seguiram para fora do link durante o tempo de verde.

O indivíduo deverá ser testado no ambiente de simulação por uma quantidade C de ciclos, portanto sua avaliação final, será dada pela média das avaliações obtidas a cada ciclo:

$$\mathrm{FA} = \frac{\sum_{j=1}^{c}(AC_i)}{c}.$$

Sendo:

- FA: função de avaliação
- AC: avaliação do ciclo
- c = número de ciclos ao qual o indivíduo foi submetido

De forma geral, a cada ciclo de simulação, os detectores de entrada e saída dos links são lidos com o auxílio da *TraCI API*, e são a quantidade de veículos que passaram por detectores são acumulados em duas listas, com os valores para cada link:

```
1 for i in range(0, len(detectors[x])):
2  detecIn = traci.inductionloop.getLastStepVehicleIDs(str(
       detectors[x][i]) + "_IN")
3  detecOut = traci.inductionloop.getLastStepVehicleIDs(str(
       detectors[x][i]) + "_OUT")
4
5    for car in detecIn:
```

```
6       if(not(car in detectors_registry[x][i][0])):
7        detectors_count[x][i][0] += 1
8        detectors_jam[x][i] = 0
9        detectors_registry[x][i][0].append(car)
10      else:
11       detectors_jam[x][i] += 1
12
13      for car in detecOut:
14       if(not(car in detectors_registry[x][i][1])):
15       detectors_count[x][i][1] += 1
16       detectors_registry[x][i][1].append(car)
```

Trecho do Método evaluate() da classe Fitness

O método *getLastStepVehicleIDs()* da *TraCI API* retorna uma lista dos veículos que passaram por determinado sensor passado como parâmetro, no último passo de simulação. A cada passo, a função de avaliação obtem essa lista de todos os detectores, e caso esses veículos não estejam na lista de veículos conhecidos pelo detector, o veículo é adicionado e o contador é incrementado.

O software *SCOOT* (Capitulo 3, Seção 3.2.3.1) propõe que vias congestionadas recebam um processamento diferenciado. De fato, vias congestionadas podem afetar o valor de avaliação do link retornando um valor de saturação considerado bom, quando na verdade a via está congestionada. Isso corre porque caso a via esteja cheia, a quantidade de veículos que conseguem chegar até o link é muito pequena, e muito provavelmente o número de veículos de saída será maior, resultando em um valor alto de avaliação para o link.

Para evitar esse tipo de distorção de resultados, essa função de avaliação utiliza um mecanismo baseado nos conceitos do *SCOOT* para tratar links congestionados de forma diferenciada:

- sempre que um mesmo veículo for detectado mais de uma vez pelo mesmo detector (veículo já se encontra na lista de veículos conhecidos, Linha 10 do código acima) o fator de congestionamento de *detectors_jam* deste link é incrementado.
- sempre que um novo veículo for conhecido pelo detector, este contador de congestionamento é zerado.

Caso esse fator de congestionamento atinja um valor pré-configurado na classe Config (*JAMDETECTION*), podemos afirmar que o veículo está parado sobre o detector de entrada há uma quantidade considerável de tempo, o que caracteriza um congestionamento. Com isso, a função de avaliação aplica um fator de penalidade, da seguinte forma:

```
1 if(detectors_jam[x][i] >= self.config.JAMDETECTION):
2  detectors_count[x][i][0] += 1 * self.config.JAMPENALTY
```

Trecho do Método evaluate() da classe Fitness

A constante *JAMPENALTY* (tem como padrão valor 1) declarada na classe Config e representa o fator de penalidade aplicado ao link em caso de congestionamento. Uma vez detectado o congestionamento, o contador de veículos de entrada continua sendo incrementado com este valor, assumindo que estes veículos não estão sendo detectados pela razão da via estar cheia.

Aplicando esta penalidade, consequentemente estaremos diminuindo o valor de avaliação do link. Quanto mais vias congestionadas o indivíduo em simulação obter, menor será o seu valor de Fitness, enquanto um indivíduo sem congestionamentos terá um maior valor de avaliação.

```
1  if(new_cycle):
2         if(passed_cycles[x] > self.config.IGNORECYCLES):
3          if((detectors_count[x][i][0] > 0) and (detectors_count[x
               ][i][1]) > 0):
4           lane_saturation[x].append(float(detectors_count[x][i
                ][1]) / float(detectors_count[x][i][0]))
5          else:
6           if(detectors_count[x][i][0] <= 0):
7            lane_saturation[x].append(1)
8           else:
9            lane_saturation[x].append(0)
10
11             detectors_count[x][i][0] -= detectors_count[x][i][1]
12         detectors_count[x][i][1] = 0
13         if(detectors_count[x][i][0] < 0):
14          detectors_count[x][i][0] =0
15
16      if(new_cycle and (passed_cycles[x] > self.config.
```

```
            IGNORECYCLES) ):
17          cycles_saturation[x].append(copy.deepcopy(lane_saturation[x
                ]))
18          lane_saturation[x] = []
```

Trecho do Método evaluate() da classe Fitness

Em seguida, conforme demonstrado no código acima, a cada novo ciclo de semáforo a função de avaliação cria uma lista com o valor avaliativo de cada ciclo em *lane_saturation*. Essa lista é colocada posteriormente em uma nova lista *cycles_saturation*, para acumular os valores para cada ciclo. Temos assim uma lista de listas com os valores de avaliação para cada link a cada ciclo. Essa lista é usada pela função para fazer o cálculo médio conforme a fórmula descrita e atribuir um valor de *Fitness* para cada indivíduo.

5.3.6 Política de Elitismo

As políticas de Elitismo procuram evitar que os melhores cromossomos sejam perdidos com o passar das gerações, sendo substituidos por cromossomos mais novos com um valor de aptidão menor. Além disso, algumas técnicas são importantes para melhorar a performance do Algoritmo Genético para evitar, por ex, que o algoritmo se perca em uma baixa diversidade genética ou demore para converger na solução ótima global.

Neste trabalho, um cuidado precisou ser implementado para manter a diversidade genética do algoritmo. Filhos de Indivíduos que forem criados exatamente iguais a qualquer um dos pais, são imediatamente descartados, como pode ser visto no trecho de código abaixo:

```
1  def buildNextGeneration(self):
2    self.generation += 1
3    i = 0
4    while(i < self.config.MAXPOPULATION):
5     couple = self.selectACouple()
6     son = guy.Guy(False, False, couple[0], couple[1])
7     if(not(son.isEqual(couple[0])) and not(son.isEqual(couple[1])
          )):
8      self.nextgeneration.append(son)
9      i += 1
```

Método *buildNextGeneration()* da classe Population

Nas linhas 7 e 8 pode-se notar que o indivíduo só será adicionado a lista de membros da próxima geração, caso ele não seja igual ao objeto pai e o objeto mãe.

Para evitar que os melhores indivíduos da geração atual sejam substituidos por indivíduos piores da próxima geração, o algoritmo realiza o seguinte procedimento:

```
10  self.nextgeneration = self.evaluateYourGuys(self.
        nextgeneration)
11  self.members = self.members + self.nextgeneration
12  self.nextgeneration = []
13  self.orderYourGuys()
14  self.members = self.members[self.config.MAXPOPULATION:]
```

- Os indivíduos da próxima geração (lista *nextgeneration*) são colocados na mesma lista dos indivíduos da geração atual (lista *members*).
- Em seguida essa lista é ordenada em ordem crescente dos valores de *Fitness*.
- Depois a lista é truncada mantendo somente a quantidade de elementos declarada como quantidade máxima da população MAXPOPULATION

Este procedimento garante que somente os indivíduos melhores adaptados sobrevivam para a próxima geração, independente se ele pertence a geração atual ou da nova geração. Note que esse procedimento pode manter o mesmo indivíduo por várias gerações, até que um número expressivo de indivíduos melhores adptados surjam e esse indivíduo possa ser descartado.

5.3.7 Critérios de Parada

O Algoritmo Genético continuará evoluindo suas gerações até que um dos critérios de parada sejam cumpridos:

- O número máximo definido na classe *Config* de gerações foi atingido. Neste caso o algoritmo retorna o indivíduo melhor desenvolvido até a última geração.
- Independente do número de gerações, caso um indivíduo atinja um número pré determinado de aptidão, o algoritmo interrompe sua evolução e retorna este indivíduo.

```
1 def letsGoDarwin(self):
2  for i in range(0, self.config.MAXGENERATIONS):
3   guy = self.getBestEvolutedGuy()
4   if(guy.getScore() < self.config.VERYGOODGUY):
5    self.buildNextGeneration()
```

Método *letsGoDarwin()* da classe Population

De acordo com a fórmula de avaliação dos links, um link com saturação perfeita obteria o valor de 1. Portanto é calculado o valor da constante *VERYGOODGUY* da seguinte forma:

$$\text{VERYGOODGUY} = NLinks * PERFECTRATE.$$

Sendo:

- NLinks = número de links avaliados
- PERFECTRATE = valor de tolerância entre o valor do indivíduo e a solução ótima

A contante *PERFECTRATE* é definida na classe de configuração e para este trabalho foi utilizado com o valor padrão de 0.8.

5.4 Análise de Experimentos e Resultados

Os experimentos realizados com o software implementado neste trabalho tem como objetivo verificar a funcionalidade do algoritmo e avaliar de que forma a alteração dos parâmetros genéticos afetam seu desempenho e qualidade dos resultados.

Os seguintes parâmetros foram definidos para para os experimentos realizados:

- Qualquer execução para um determinado cenário e determinada configuração do algoritmo, foi realizada 3 vezes e o seus resultados comparados. Se o algoritmo estiver correto, apesar da natureza aleatória da geração de indivíduos, espera-se que as 3 execuções encontrem resultados semelhantes, próximos ao máximo global da otimização.
- O algoritmo foi executado variando os seguintes parâmetros genéticos: Método de Seleção de Indivíduos e Taxa de Mutação.
- Além da variação dos parâmetros genéticos, o algoritmo teve sua eficiência testada em duas variações do ambiente de simulação: uma com o fluxo de veículos balanceado entre as duas vias das intersecções semaforizadas, e outro onde o fluxo de uma das vias das intersecções semaforizadas foi reduzido pela metade. Se o algoritmo estiver correto, espera-se que o algoritmo converja para uma solução onde a via com maior fluxo seja priorizada.
- A cada teste, serão apresentados os seguintes dados: sumário da população inicial, sumário da população final e indivíduo mais adaptado encontrado.

5.4.1 Experimento 1 - Variação do Método de Seleção

Este experimento foi realizado variando o método de seleção dos indivíduos para cruzamento. A primeira execução utilizará o método da roleta utilizando valores de *Fitness* e a segunda execução utilizará o método da roleta utilizando valores de *Ranking* como visto na seção 5.3.4.2 deste capítulo.

5.4.1.1 Método da Roleta com Ranking

Cenário de experimentação:

- População Máxima: 20 indivíduos
- Número máximo de gerações: 10 gerações
- Taxa de Mutação: 10%
- Percentual Máximo de Mutação: 5% (max. 6 genes)
- Método de Seleção: Roleta (Ranking)
- Fluxo: balanceado

Tabela 9: Sumário das Primeiras Gerações do Experimento 1 - Roleta com Ranking

Execução 1	
Indivíduo	Fitness
0	4.043756
1	4.270020
2	4.328340
3	4.360924
4	4.431764
5	4.579854
6	4.607562
7	4.733320
8	4.787812
9	4.813591
10	4.891310
11	4.902615
12	4.954249
13	5.159219
14	5.287972
15	5.330776
16	5.461483
17	5.775865
18	5.942228
19	5.951133

Execução 2	
Indivíduo	Fitness
0	4.208570
1	4.248644
2	4.248644
3	4.317492
4	4.321546
5	4.630166
6	4.716878
7	4.808033
8	4.822155
9	4.842066
10	4.843037
11	4.987507
12	5.030614
13	5.057926
14	5.074150
15	5.141630
16	5.747924
17	5.826847
18	6.158644
19	6.182112

Execução 3	
Indivíduo	Fitness
0	3.970789
1	4.178861
2	4.289173
3	4.651401
4	4.692780
5	4.738440
6	4.803315
7	4.836642
8	4.871466
9	4.953493
10	4.963625
11	5.074150
12	5.133419
13	5.168672
14	5.257023
15	5.438394
16	5.756270
17	5.769155
18	5.828588
19	5.946440

Tabela 10: Sumário das Últimas Gerações do Experimento 1 - Roleta com Ranking

Execução 1	
Indivíduo	Fitness
0	6.304245
1	6.304245
2	6.304245
3	6.313985
4	6.313985
5	6.313985
6	6.313985
7	6.313985
8	6.327240
9	6.327240
10	6.327240
11	6.328860
12	6.369977
13	6.369977
14	6.369977
15	6.386893
16	6.399917
17	6.399917
18	6.399917
19	6.399917

Execução 2	
Indivíduo	Fitness
0	6.278695
1	6.278695
2	6.298965
3	6.298965
4	6.298965
5	6.298965
6	6.298965
7	6.298965
8	6.304245
9	6.304245
10	6.304245
11	6.369977
12	6.369977
13	6.369977
14	6.386893
15	6.386893
16	6.386893
17	6.399917
18	6.399917
19	6.399917

Execução 3	
Indivíduo	Fitness
0	6.201503
1	6.211469
2	6.221217
3	6.239678
4	6.239678
5	6.253342
6	6.253342
7	6.272263
8	6.272263
9	6.272263
10	6.272263
11	6.298965
12	6.298965
13	6.304245
14	6.313985
15	6.328860
16	6.386893
17	6.386893
18	6.386893
19	6.386893

Melhores Indivíduos:

Execução 1
Cromossomo 1: 00000001001111010110011000000101110000000000010001101011
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 10001010010001010011111111111011111110011010101100011111
Tempo de Verde: 35s
Tempo de Vermelho: 25s
Fitness 6.399917

Execução 2
Cromossomo 1: 111100110000100100000100001010000100100000110110110001 00
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 000111111100110101011101101010101111000101011111111101001
Tempo de Verde: 35s
Tempo de Vermelho: 25s
Fitness 6.399917

Execução 3
Cromossomo 1: 000000000110000100010001101100110100100011001010110010 01
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 001111011001110111001101001001001101001001111010111101 10
Tempo de Verde: 32s
Tempo de Vermelho: 28s
Fitness 6.386893

5.4.1.2 Método da Roleta com Fitness

Cenário de experimentação:

- População Máxima: 20 indivíduos
- Número máximo de gerações: 10 gerações
- Taxa de Mutação: 10%
- Percentual Máximo de Mutação: 5% (max. 6 genes)
- Método de Seleção: Roleta (Fitness)
- Fluxo: balanceado

Tabela 11: Sumário das Primeiras Gerações do Experimento 1 Roleta com Fitness

Execução 1	
Indivíduo	Fitness
0	4.043596
1	4.235821
2	4.401088
3	4.410726
4	4.450673
5	4.505145
6	4.544396
7	4.570382
8	4.575557
9	4.852004
10	4.862553
11	4.987507
12	5.105480
13	5.116170
14	5.159219
15	5.169960
16	5.348678
17	5.492476
18	5.612933
19	5.970548

Execução 2	
Indivíduo	Fitness
0	3.816564
1	3.996372
2	4.036241
3	4.181077
4	4.273466
5	4.478235
6	4.643149
7	4.744154
8	4.911194
9	4.948328
10	4.983602
11	4.992042
12	5.003012
13	5.042878
14	5.099016
15	5.421220
16	5.542911
17	5.728588
18	5.763555
19	5.843076

Execução 3	
Indivíduo	Fitness
0	3.872419
1	3.953490
2	4.057531
3	4.086655
4	4.553279
5	4.559606
6	4.645218
7	4.652059
8	4.843037
9	4.863287
10	4.943824
11	5.200845
12	5.226071
13	5.350496
14	5.617208
15	5.656558
16	5.783124
17	5.997741
18	6.000150
19	6.300680

Tabela 12: Sumário das Últimas Gerações do Experimento 1 - Roleta com Fitness

Execução 1	
Indivíduo	Fitness
0	6.003779
1	6.039348
2	6.056525
3	6.061779
4	6.076707
5	6.076707
6	6.103395
7	6.103395
8	6.103395
9	6.105905
10	6.117871
11	6.124109
12	6.135328
13	6.148218
14	6.148218
15	6.148218
16	6.186670
17	6.186670
18	6.186670
19	6.186670

Execução 2	
Indivíduo	Fitness
0	6.160760
1	6.160760
2	6.166064
3	6.166064
4	6.172277
5	6.180223
6	6.182112
7	6.186670
8	6.186670
9	6.187072
10	6.253342
11	6.254624
12	6.254624
13	6.292624
14	6.292624
15	6.292624
16	6.304245
17	6.313985
18	6.327240
19	6.369977

Execução 3	
Indivíduo	Fitness
0	6.314825
1	6.314825
2	6.328860
3	6.328860
4	6.328860
5	6.328860
6	6.369977
7	6.369977
8	6.369977
9	6.369977
10	6.369977
11	6.386893
12	6.386893
13	6.386893
14	6.386893
15	6.386893
16	6.386893
17	6.399917
18	6.399917
19	6.399917

Melhores Indivíduos:

Execução 1
Cromossomo 1: 00011001011001110101111110111011100000100001000000000000
Tempo de Verde: 23s
Tempo de Vermelho: 37s
Cromossomo 2: 01011110110000111100000100110101000101001001110100111000
Tempo de Verde: 26s
Tempo de Vermelho: 34s
Fitness 6.187072

Execução 2
Cromossomo 1: 01010110000011101100000011111001011000010001001000000000
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 11110111110010010001111111000111111011110110000101010100
Tempo de Verde: 34s
Tempo de Vermelho: 26s
Fitness 6.369977

Execução 3
Cromossomo 1: 11100011000010111001110100000000000010110001010000001010
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 11110001110101001100111001010011101111111101111101110000
Tempo de Verde: 35s
Tempo de Vermelho: 25s
Fitness 6.399917

5.4.1.3 Considerações

Pode-se notar que independentemente do método de seleção de indivíduos adotado, todas as execuções do algoritmo convergiram para uma solução próxima do máximo global, mesmo que todas elas tenham convergido para um indivíduo diferente ao final do algoritmo. Isso ocorre porque indivíduos com cromossomos diferentes podem conter o mesmo valor de Fitness e serem igualmente adaptados, por possuirem a mesma quantidade de bits 0 e 1 em seus cromossomos, configurando os mesmos tempos de verde para os semáforos.

Comparando os dois métodos de Seleção de Indivíduos, pode-se notar claramente que o Método de Seleção utilizando valores de Ranking convergiu para o máximo global em duas das três execuções, sendo que na terceira solução encontra-se muito próxima do máximo global. Pode-se notar também que neste método, nas últimas gerações encontramos muitos indivíduos com valor de aptidão semelhantes, isso pode ocorrer pela grande probabilidade dos indivíduos com valor de Fitness mais elevado serem selecionados a maior parte das vezes, diminuindo consideravelmente a diversidade genética com o passar das gerações, o que ocorre com menor frequência no método de Seleção utilizando os valores de Fitness. Apesar da diversidade genética mais alta, este método só convergiu para o máximo global em uma das três execuções, sendo que em uma delas, o valor encontrado ficou mais distante do máximo global do que todas as outras execuções.

Considera-se portanto neste teste que, apesar de limitar a diversidade genética com o passar das gerações, o método de Seleção da Roleta Utilizando valores de Ranking é mais efetivo para encontrar a solução mais próxima do máximo global.

5.4.2 Experimento 2 - Aumento da Taxa de Mutação

O objetivo deste experimento é analisar quais são os efeitos da elevação da taxa de probabilidade de mutação dos indivíduos de 10% para 30%. Os resultados serão comparados com o experimento anterior realizado utilizando o Método de Seleção da Roleto com valores de Ranking e taxa de Mutação de 10%.

Cenário de experimentação:

- População Máxima: 20 indivíduos
- Número máximo de gerações: 10 gerações
- Taxa de Mutação: 30%
- Percentual Máximo de Mutação: 5% (max. 6 genes)
- Método de Seleção: Roleta (Ranking)
- Fluxo: balanceado

Tabela 13: Sumário das Primeiras Gerações do Experimento 2 - Taxa de Mutação

Execução 1	
Indivíduo	Fitness
0	3.821849
1	4.096326
2	4.138197
3	4.332159
4	4.498482
5	4.579854
6	4.625107
7	4.649051
8	4.745303
9	4.866709
10	4.973485
11	5.032566
12	5.116787
13	5.309272
14	5.411326
15	5.501518
16	5.758049
17	5.760777
18	5.769119
19	5.821025

Execução 2	
Indivíduo	Fitness
0	3.952902
1	3.970972
2	4.000152
3	4.119445
4	4.279266
5	4.371263
6	4.460000
7	4.494508
8	4.507215
9	4.553279
10	4.578154
11	4.693559
12	5.020460
13	5.053682
14	5.345803
15	5.505761
16	5.772819
17	5.944638
18	6.061779
19	6.153370

Execução 3	
Indivíduo	Fitness
0	4.054392
1	4.313301
2	4.317954
3	4.409711
4	4.440062
5	4.493500
6	4.561173
7	4.570404
8	4.599846
9	4.836170
10	4.932026
11	5.001177
12	5.037899
13	5.122359
14	5.309181
15	5.309181
16	5.333797
17	5.849170
18	5.970548
19	6.000150

Tabela 14: Sumário das Últimas Gerações do Experimento 2 - Taxa de Mutação

Execução 1	
Indivíduo	Fitness
0	6.300680
1	6.314825
2	6.328860
3	6.328860
4	6.328860
5	6.328860
6	6.328860
7	6.328860
8	6.369977
9	6.369977
10	6.369977
11	6.369977
12	6.386893
13	6.399917
14	6.399917
15	6.399917
16	6.399917
17	6.399917
18	6.399917
19	6.399917

Execução 2	
Indivíduo	Fitness
0	6.369977
1	6.369977
2	6.369977
3	6.369977
4	6.369977
5	6.369977
6	6.369977
7	6.386893
8	6.386893
9	6.386893
10	6.386893
11	6.386893
12	6.386893
13	6.386893
14	6.386893
15	6.386893
16	6.399917
17	6.399917
18	6.399917
19	6.399917

Execução 3	
Indivíduo	Fitness
0	6.369977
1	6.369977
2	6.369977
3	6.369977
4	6.369977
5	6.369977
6	6.369977
7	6.386893
8	6.386893
9	6.386893
10	6.386893
11	6.386893
12	6.386893
13	6.386893
14	6.386893
15	6.386893
16	6.399917
17	6.399917
18	6.399917
19	6.399917

Melhores Indivíduos:

Execução 1
Cromossomo 1: 00011000001010000111001000000100001010001000111011001011
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 00110001101100111011000010001101111111111111111101110010
Tempo de Verde: 35s
Tempo de Vermelho: 25s
Fitness 6.399917

Execução 2
Cromossomo 1: 10101000111001010001101110110101010010000000000000100000
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 00110100111011011110100011011101110101001111010111011101
Tempo de Verde: 35s
Tempo de Vermelho: 25s
Fitness 6.399917

Execução 3
Cromossomo 1: 01100010100001010100100000000010111000100011100100110100
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 11010001010101001110010110011110111111101101101111001011
Tempo de Verde: 35s
Tempo de Vermelho: 25s
Fitness 6.399917

5.4.2.1 Considerações

Analisando os resultados obtidos neste experimento e comparando-os com os resultados obtidos no experimento anterior com iguais condições, porém com taxa de mutação em 10%, notamos que ambas as execuções do algoritmo convergiram para soluções muito próximas do máximo global. Com taxa de mutação em 10%, duas das execuções convergiram para a solução otimizada, diferentemente do experimento atual quando as três execuções convergiram para a solução otimizada.

Podemos notar que a Execução 1 do algoritmo com a taxa de mutação elevada produziu na primeira geração indivíduos com os menores valores de Fitness em média. Porém, apesar desta caraciterística o algoritmo convergiu para a solução otimizada antes das outras duas execuções.

Desta forma podemos considerar que para este cenário de simulação, uma taxa de mutação mais elevada pode tornar a velocidade de convergência para o máximo global mais efetiva.

5.4.3 Experimento 3 - Desbalanceamento do Fluxo

O objetivo deste experimento é analisar a capacidade do algoritmo de encontrar um máximo global para um fluxo de veículos desbalanceado entre as vias que chegam ao semáforos. Para isso, vamos utilizar uma configuração de fluxo que reduz pela metade a quantidade de veículos em uma das vias que chegam ao semáforo. As configurações para o experimento são feitas com base nos melhores resultados dos experimentos anteriores.

Cenário de experimentação:

- População Máxima: 20 indivíduos
- Número máximo de gerações: 10 gerações
- Taxa de Mutação: 30%
- Percentual Máximo de Mutação: 5% (max. 6 genes)
- Método de Seleção: Roleta (Ranking)
- Fluxo: não-balanceado

Tabela 15: Sumário das Primeiras Gerações do Experimento 3 - Desbalanceamento de Fluxo

Execução 1	
Indivíduo	Fitness
0	4.442317
1	4.723094
2	4.934575
3	5.084898
4	5.118542
5	5.152799
6	5.181400
7	5.678713
8	5.981979
9	6.090268
10	6.104568
11	6.167817
12	6.257156
13	6.433765
14	6.468574
15	6.546240
16	6.568517
17	6.694039
18	6.837154
19	7.188771

Execução 2	
Indivíduo	Fitness
0	4.702823
1	5.061345
2	5.313586
3	5.346961
4	5.386917
5	5.516674
6	5.554961
7	5.564412
8	5.743162
9	5.817271
10	5.984439
11	5.996951
12	6.163058
13	6.256004
14	6.526806
15	6.599151
16	6.676432
17	6.676673
18	6.743225
19	6.853157

Execução 3	
Indivíduo	Fitness
0	4.263937
1	4.792869
2	4.877325
3	5.003927
4	5.047378
5	5.417818
6	5.527338
7	5.570724
8	5.665131
9	5.713591
10	6.002419
11	6.023879
12	6.073920
13	6.205651
14	6.244926
15	6.252333
16	6.271485
17	6.336107
18	6.424989
19	6.630392

Tabela 16: Sumário das Últimas Gerações do Experimento 3 - Desbalanceamento de Fluxo

Execução 1	
Indivíduo	Fitness
0	7.112638
1	7.112638
2	7.112638
3	7.177756
4	7.188771
5	7.188771
6	7.188771
7	7.188771
8	7.188771
9	7.188771
10	7.188771
11	7.188771
12	7.200733
13	7.200733
14	7.200733
15	7.200733
16	7.200733
17	7.200733
18	7.200733
19	7.200733

Execução 2	
Indivíduo	Fitness
0	7.112638
1	7.177756
2	7.188771
3	7.188771
4	7.188771
5	7.188771
6	7.188771
7	7.188771
8	7.188771
9	7.188771
10	7.188771
11	7.188771
12	7.188771
13	7.200733
14	7.200733
15	7.200733
16	7.200733
17	7.200733
18	7.200733
19	7.200733

Execução 3	
Indivíduo	Fitness
0	7.104751
1	7.104751
2	7.104751
3	7.104751
4	7.104751
5	7.110055
6	7.110055
7	7.110055
8	7.110055
9	7.112638
10	7.112638
11	7.177756
12	7.177756
13	7.177756
14	7.177756
15	7.188771
16	7.200733
17	7.200733
18	7.200733
19	7.200733

Melhores Indivíduos:

Execução 1
Cromossomo 1: 00001101001101000000000110011001010110000111100001000001
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 01100000101101100000010001000010101000000001000000101110
Tempo de Verde: 17s
Tempo de Vermelho: 43s
Fitness 7.200733

Execução 2
Cromossomo 1: 10011001100101001010100010010000000000110001001001001101
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 10000110110000010001000000010000000011100100000011101010
Tempo de Verde: 17s
Tempo de Vermelho: 43s
Fitness 7.200733

Execução 3
Cromossomo 1: 11100110010011111111101100000100000010000000000001000000
Tempo de Verde: 20s
Tempo de Vermelho: 40s
Cromossomo 2: 00100000100000010001010111111000100001000001001000001100
Tempo de Verde: 17s
Tempo de Vermelho: 43s
Fitness 7.200733

5.4.3.1 Considerações

De acordo com o resultado das três execuções com a quantidade de fluxo modificada em 50% para uma das vias das intersecções semaforizadas, pode-se considerar que o algoritmo corretamente determinou um tempo de verde proporcionalmente menor (cerca de 48% menor) para a via com menor fluxo de tráfego. Pode-se estimar que o algoritmo possui a capacidade de convergir para uma solução que priorize as vias de tráfego com maior demanda, além de convergir para um valor próximo ao máximo global.

6 Considerações finais e trabalhos futuros

Sabemos que a computação natural vem sendo objeto de estudo nas últimas décadas, com a tentativa de buscar soluções inspiradas na natureza para problemas computacionais que não podem ser resolvidos de forma eficiente com os modelos computacionais. Os algoritmos genéticos, baseados na teoria evolutiva de Darwin, aparecem como uma possibilidade real de busca de soluções otimizadas para problemas em espaços de todos os tamanhos.

Neste trabalho foi demonstrado a possibilidade de se modelar um problema comum de busca de otimização em função dos conceitos utilizados pelos algoritmos genéticos e aplicá-los para obtenção desta solução, criando soluções satisfatórias para os cenários onde o algoritmo foi testado.

É importante ressaltar a importância da função de avaliação na capacidade do algoritmo culminar em uma solução próxima do valor máximo para o problema de otimização. Neste trabalho, o desenvolvimento com o simulador SUMO foi essencial para que o algoritmo avaliasse corretamente os indivíduos, verificando as consequências da aplicação da solução representada pelo indivíduo em um ambiente que simule o comportamento do tráfego real.

O desempenho do sistema pode ser um fator crucial para a aplicabilidade dos algoritmos genéticos no problema da otimização de tempos para programação de semáforos.

O uso do simulador como heurística para avaliação dos indivíduos torna a execução do algoritmo consideravelmente lenta, levando de 1 a 2 segundos para avaliação de cada indivíduo. Considerando uma população relativamente pequena, e uma baixa quantidade de gerações, o algoritmo pode chegar a alguns minutos de execução.

Este baixo desempenho na execução do algoritmo pode tornar a sua aplicação inviável considerando um cenário de simulação plenamente amplo, como uma cidade.

Algumas observações, porém podem ser feitas considerando um ambiente real de aplicação do algoritmo na programação semáforica:

- Considerando que o algoritmo seria executado em pequenos intervalos de tempo para adaptar a configuração do semáforo à nova demanda de tráfego, não há a necessidade de se gerar uma população inicial aleatória e aguardar a convergência da solução em número grande de gerações. A última população evoluída da execução do algoritmo anterior pode ser armazenada e aplicada ao novo cenário de tráfego. Partindo de uma população parcialmente evoluída, espera-se reduzir consideravelmente a quantidade de gerações necessárias para alcançar a solução dada como ótima.

- Considerando que a otimização é feita pelo algoritmo utilizando somente as vias de entrada do semáforo, pode-se imaginar um cenário de aplicação reduzindo, com apenas uma intersecção semaforizada, diminuindo assim o esforço computacional necessário. O processamento dessas intersecções únicas pode ser feito paralelamente, encontrando soluções otimizadas individuais para cada semáforo.

Apesar do baixo desempenho de execução, os experimentos realizados neste trabalho demonstraram que o algoritmo é eficaz na busca de soluções otimizadas para a proporção de tempos entre verde e vermelho de semáforos, considerando a demanda de fluxo para entre as vias semaforizadas, inclusive mostrando eficácia na adaptação das soluções às variações de demanda de tráfego.

Como trabalhos futuros complementares a esta pesquisa, pode-se apontar:

- Processamento Distribuído: baseado no problema de desempenho demonstrado nestre trabalho, pode-se modelar uma quantidade de cenários individuais com apenas um semáforo, aplicar o algoritmo de otimização de forma distribuída e individual para cada semáforo, em busca de solução ótima global.

- Desenvolvimento de outras funções de avaliação: desenvolvimento de outras funções de avaliação, utilizando outros simuladores encontrados no mercado, a fim de comparar os resultados de performance.

Referências

ARTERO, A. O. *Inteligência Artificil - Teórica e Prática*. [S.l.]: Editoria Livraria da Física, 2009.

BEHRISCH, M. et al. Sumo - simulation of urban mobility: An overview. In: *SIMUL 2011, The Third International Conference on Advances in System Simulation*. Barcelona, Spain: [s.n.], 2011. p. 63–68.

BOZOLA, C. F. et al. *NT 089/83 - Controle de Tráfego em Área - Sistema SEMCO*. [S.l.], 1983.

BRETHERTON, D. Advice leaflet 1: The scoot urban traffic control system. 2011. URL: http://www.scoot-utc.com, Last visited: 06/06/2011.

CAPELLI, A. *Semáforos Inteligentes*. 2009.

COPPIN, B. *Inteligência Artificil*. [S.l.]: Editoria LTC, 2010.

FOUNDATION, T. P. *About Python*. 2011. URL: http://python.org/about/. Last visited: 10/11/2011.

JADAAN, O. A. Improved selection operator for ga. 2008. URL: http://www.jatit.org/volumes/research-papers/Vol4No4/3vol4no4.pdf, Last visited: 15/11/2011.

MUNHOZ, E. A. M. *NT 005/78 - Transyt/6 Programa Computacional Para Coordenação E Sincronismo de Semáforos.* [S.l.], 1978.

NETO, J. C. *Engenharia de Tráfego e Transportes Urbanos*. 2011. URL: http://meusite.mackenzie.com.br/professor_cucci/. Last visited: 03/06/2011.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. [S.l.]: Campus, 2003.

VILANOVA, L. M. *O controle de semáforos em tempo real: a experiência de São Paulo*. 2005.

VILANOVA, L. M. Programação semafórica usando método da saturação. 2005. URL: http://www.scoot-utc.com, Last visited: 15/11/2011.

VILANOVA, L. M. Fundamentos da programaÇÃo semafÓrica. 2011. URL: http://www.sinaldetransito.com.br/artigos/programacao.pdf, Last visited: 19/11/2011.

APÊNDICE A – Códigos fonte

•Arquivo Inicial do Sistema (tgiv1.py)

```
1  import os, subprocess, sys, socket, time, struct, random, traci,
       random
2  import fitness, guy, population
3
4  population = population.Population()
5  population.letsGoDarwin()
6  guy = population.getBestEvolutedGuy()
```

•Classe Config (config.py)

```
1  import os, subprocess, sys, socket, time, struct, random, traci,
       random, datetime
2  class Config:
3
4   def __init__(self):
5    #TCP PORT for TRACI
6    self.PORT = 8813
7
8    #Application
9    self.DEBUG = False
10   self.LOG = True
11   self.DEEPLOG = False
12   self.SPEEDLOG = False
13   self.GENERATIONLOG = True
14
```

```
15    #Logfiles
16    self.LOGFILE = "log.txt"
17    self.DEEPLOGFILE = "deeplog.txt"
18    self.SPEEDLOGFILE = "speed.txt"
19    self.GENERATIONLOGFILE = "generations.txt"
20
21    #Bussiness rules
22    self.JAMDETECTION = 25
23    self.JAMPENALTY = 2
24    self.YELLOW_TIME = 2
25    self.MINFASETIME = 2
26
27    #Times
28    self.SIMULATION_TIME = 720
29    #self.SIMULATION_TIME = 360
30    self.CYCLE_TIME = 60
31    self.OFFSET = 10;
32    self.IGNORECYCLES = 2
33
34    #TRACI Lights Config
35    self.LIGHTSID = ["A", "B"]
36    #self.DETECTORSID = [["AT_A","A_AT","AB_A","A_AB", "AL_A","
          A_AL","B_A"],["BT_B","B_BT","BB_B","B_BB","BR_B","B_BR","
          A_B"]]
37    self.DETECTORSID = [["AT_A","AB_A","AL_A","B_A"],["BT_B","BB_B
          ","BR_B","A_B"]] #Vias de Entrada
38    #self.DETECTORSID = [["AT_A"],[]] #Vias de Entrada
39
40    self.GREEN_PHASE = "GGrrGGrr"
41    self.YELLOW_PHASEA = "rryyrryy"
42    self.YELLOW_PHASEB = "yyrryyrr"
43    self.RED_PHASE = "rrGGrrGG"
44
45    #Sumo Config
46    self.SUMOEXEGUI = "sumo-gui"
```

```
47    #self.SUMOEXE = "sumo-gui"
48    self.SUMOEXE = "sumo"
49    self.SUMOCONFIG = "tgiv1.sumo.cfg"
50
51    #Genetic Algorithms
52    self.MAXPOPULATION = 20
53    self.MAXGENERATIONS = 10
54    self.MUTATIONRATE = 0.3
55    self.MUTATIONGENERATE = 0.05
56    self.CREATIONMETHOD = 1
57    self.SELECTIONMETHOD = 0
58    self.PERFECTRATE = 0.8
59    self.VERYGOODGUY = (len(self.DETECTORSID[0]) + len(self.
          DETECTORSID[1])) * self.PERFECTRATE
60
61  def configSumo(self, GUI):
62    if "SUMO" in os.environ:
63      self.SUMOEXE = os.path.join(os.environ["SUMO"], "sumo-gui")
64    if(GUI):
65      self.SUMOPROCESS = subprocess.Popen("%s -c %s" % (self.
           SUMOEXEGUI, self.SUMOCONFIG), shell=True, stdout=sys
66      .stdout)
67    else:
68      self.SUMOPROCESS = subprocess.Popen("%s -c %s" % (self.
           SUMOEXE, self.SUMOCONFIG), shell=True, stdout=sys
69      .stdout)
```

•Classe Guy (guy.py)

```
1  import os, subprocess, sys, socket, time, struct, random, traci,
       random
2  import config, fitness
3
4  class Guy:
5
```

```
6   def __init__(self, autoFitness=True, randomize=True, mother=
        False, father=False):
7    self.config = config.Config()
8    self.lights = [[],[]]
9
10   for i in range(0, len(self.config.LIGHTSID)):
11    if(randomize):
12     self.lights[i].append(self.generateCromossome())
13    else:
14     self.lights[i].append(self.crossCromossome(mother.
           getCromossome(i), father.getCromossome(i)))
15
16   mutate = (float(random.randrange(0, 100))/100)
17
18   if(mutate <= self.config.MUTATIONRATE):
19    self.mutateMyself()
20
21   for i in range(0, len(self.config.LIGHTSID)):
22    self.lights[i].append(self.calculateTime(self.lights[i][0]))
23    self.lights[i].append(self.config.CYCLE_TIME - self.lights[i
          ][1])
24
25   self.fitness = fitness.Fitness(self)
26   self.score = 0.0
27
28   if(autoFitness):
29    self.evaluateYourself()
30    self.lights.append(self.score)
31
32  def getScore(self):
33   return self.score
34
35  def getCromossome(self, i):
36   return self.lights[i][0]
37
```

```
38  def generateCromossome(self):
39   cromossome = str()
40   max_green_time = self.config.CYCLE_TIME - (self.config.
        YELLOW_TIME * 2) - (self.config.MINFASETIME)
41   total_fase_time = self.config.CYCLE_TIME - (self.config.
        YELLOW_TIME * 2)
42
43   if(self.config.CREATIONMETHOD==0):
44    for i in range(0, max_green_time):
45     rand1 = random.randrange(0, 100)
46     rand2 = random.randrange(0, 100)
47     if(rand2 < rand1):
48      cromossome += str("1")
49     else:
50      cromossome += str("0")
51
52   elif(self.config.CREATIONMETHOD==1):
53    ones = random.randrange(0, max_green_time)
54    positions = []
55    for i in range(0, ones):
56     pos = random.randrange(0, total_fase_time)
57     while(pos in positions):
58      pos = random.randrange(0, total_fase_time)
59     positions.append(pos)
60
61    for i in range(0, total_fase_time):
62     if(i in positions):
63      cromossome += str("1")
64     else:
65      cromossome += str("0")
66
67   return cromossome
68
69  def crossCromossome(self, cromofather, cromomother):
70   div = random.randrange(0, self.config.CYCLE_TIME)
```

```
71
72      if(self.config.DEBUG):
73          print "DIV %d"%(div)
74          print "DAD: %s"%(cromofather)
75          print "MON: %s"%(cromomother)
76          print "SON: %s"%(cromofather[:div] + cromomother[div:])
77
78      return cromofather[:div] + cromomother[div:]
79
80  def mutateMyself(self):
81
82      for x in range(0, len(self.config.LIGHTSID)):
83          genesToMutate = random.randrange(0, int(len(self.lights[x
                ][0]) * self.config.MUTATIONGENERATE))
84          for i in range(0, genesToMutate):
85              position = random.randrange(0, len(self.lights[x][0]))
86              newgene = str((int(self.lights[x][0][position]) + 1) % 2)
87              self.lights[x][0] = self.lights[x][0][:position] + newgene +
                    self.lights[x][0][position+1:]
88
89  def calculateTime(self, cromossome):
90      greenTime = int(0)
91      for c in cromossome:
92          if(int(c) > 0):
93              greenTime += 1
94      return greenTime
95
96  def isEqual(self, otherguy):
97      if((self.lights[0][0] == otherguy.lights[0][0]) and (self.
                lights[1][0] == otherguy.lights[1][0])):
98          return True
99
100
101 def introduceYourself(self):
102     print "Hello, I'm a Guy"
```

```
103     print "My Cromossome is for Light %s is %s."%(self.config.
            LIGHTSID[0], self.lights[0][0])
104     print "My Cromossome is for Light %s is %s."%(self.config.
            LIGHTSID[1], self.lights[1][0])
105     print "My Green Time for Light %s is %d."%(self.config.
            LIGHTSID[0], self.lights[0][1])
106     print "My Green Time for Light %s is %d."%(self.config.
            LIGHTSID[1], self.lights[1][1])
107     print "My Red Time for Light %s is %d."%(self.config.LIGHTSID
            [0], self.lights[0][2])
108     print "My Red Time for Light %s is %d."%(self.config.LIGHTSID
            [1], self.lights[1][2])
109     print "And my Score is %f.\n"%(self.score)
110
111   def evaluateYourself(self):
112     self.score = self.fitness.evaluate()
113
114   def showGUI(self):
115     self.score = self.fitness.evaluate(True)
```

•Classe Population (population.py)

```
1  import os, subprocess, sys, socket, time, struct, random, traci,
       random, datetime, copy
2  import guy, config
3
4  class Population:
5
6    def __init__(self, brandnew=True):
7      self.config = config.Config()
8
9      self.members = []
10     self.nextgeneration = []
11     self.generation = int(0)
12
```

```
13    if(self.config.GENERATIONLOG):
14     self.generationlogfile = open(self.config.GENERATIONLOGFILE+
           str(random.randrange(1000,9999)), "w")
15
16    if(brandnew):
17     for i in range(0, self.config.MAXPOPULATION):
18      self.members.append(guy.Guy())
19
20    self.orderYourGuys()
21
22  def introduceYourGuys(self):
23   for guy in self.members:
24    guy.introduceYourself()
25
26  def evaluateYourGuys(self, members):
27   if(self.config.SPEEDLOG):
28    speedlogfile = open(self.config.SPEEDLOGFILE, "w")
29    print >> speedlogfile, "POPULATION EVALUATION STARTED AT %s"
         % (datetime.datetime.now())
30
31   for i in range(0, len(members)):
32    if(self.config.SPEEDLOG):
33     print >> speedlogfile, "\tEVALUATION OF GUY NUM %d STATED AT
          %s" % (i, datetime.datetime.now())
34
35    members[i].evaluateYourself()
36    print " GUY %i SCORE: %f"%(i, members[i].getScore())
37
38    if(self.config.SPEEDLOG):
39     print >> speedlogfile, "\tEVALUATION OF GUY NUM %d FINISHED
          AT %s" % (i, datetime.datetime.now())
40
41   if(self.config.SPEEDLOG):
42    print >> speedlogfile, "POPULATION EVALUATION FINISHED AT %s"
         % (datetime.datetime.now())
```

```
43
44    return members
45
46
47  def orderYourGuys(self):
48    if(self.config.DEBUG):
49      print "MEMBERS LIST BEFORE QUICKSORT"
50      for guy in self.members:
51        print guy.getScore()
52
53    self.members = self._quicksort_(self.members)
54
55    if(self.config.DEBUG):
56      print "MEMBERS LIST AFTER QUICKSORT"
57      for guy in self.members:
58        print guy.getScore()
59
60  def selectACouple(self):
61    candidates = copy.copy(self.members)
62    couple = []
63
64    if(self.config.SELECTIONMETHOD==0):
65      for x in [0,1]:
66        roullete = selector = int(0)
67        for i in range(1, len(candidates)):
68          roullete += i*2
69
70        rand = random.randrange(1, roullete)
71
72        if(self.config.DEBUG):
73          print "MAX VALUE FOR ROULLETE: %d"%(roullete)
74          print "RANDOM VALUE FOR ROULLETE: %d"%(rand)
75
76        for i in range(1, len(candidates)):
77          selector += i*2
```

```
78          if(rand > selector):
79           continue
80          else:
81           if(self.config.DEBUG):
82            print "\nSELECTED GUY IS: "
83            print candidates[i].introduceYourself()
84
85           couple.append(candidates.pop(i))
86           break
87
88      elif(self.config.SELECTIONMETHOD==1):
89       for x in [0,1]:
90        roullete = selector = float(0)
91        for i in range(0, len(candidates)):
92         roullete += candidates[i].getScore()
93
94        rand = random.randrange(0, int(roullete))
95
96        if(self.config.DEBUG):
97         print "MAX VALUE FOR ROULLETE: %d"%(roullete)
98         print "RANDOM VALUE FOR ROULLETE: %d"%(rand)
99
100       for i in range(0, len(candidates)):
101        selector += candidates[i].getScore()
102        if(rand > selector):
103         continue
104        else:
105         if(self.config.DEBUG):
106          print "\nSELECTED GUY IS: "
107          print candidates[i].introduceYourself()
108
109         couple.append(candidates.pop(i))
110         break
111
112     return couple
```

```
113
114  def buildNextGeneration(self):
115
116   if(self.config.GENERATIONLOG):
117    print >> self.generationlogfile, "GENERATION %d SUMMARY:"%(
         self.generation)
118    for i in range(0, len(self.members)):
119     print >> self.generationlogfile, "GUY %i SCORE %f "%(i, self
          .members[i].getScore())
120
121   self.generation += 1
122
123   i = 0
124   while(i < self.config.MAXPOPULATION):
125    couple = self.selectACouple()
126    son = guy.Guy(False, False, couple[0], couple[1])
127    if(not(son.isEqual(couple[0])) and not(son.isEqual(couple[1])
         )):
128     self.nextgeneration.append(son)
129     i += 1
130
131   self.nextgeneration = self.evaluateYourGuys(self.
       nextgeneration)
132   self.members = self.members + self.nextgeneration
133   self.nextgeneration = []
134
135   self.orderYourGuys()
136
137   self.members = self.members[self.config.MAXPOPULATION:]
138
139   print "GENERATION %d SUMMARY"%(self.generation)
140   for i in range(0, len(self.members)):
141    g = self.members[i]
142    print "i: %d s: %f"%(i, g.getScore())
143
```

```
144   def letsGoDarwin(self):
145     for i in range(0, self.config.MAXGENERATIONS):
146       guy = self.getBestEvolutedGuy()
147       if(guy.getScore() < self.config.VERYGOODGUY):
148         self.buildNextGeneration()
149
150   def getBestEvolutedGuy(self):
151     return self.members[len(self.members)-1]
152
153   def _quicksort_(self, members):
154
155     if len(members) <= 1:
156       return members
157
158     less, equal, greater = [], [], []
159     pivot = members[0]
160
161     for guy in members:
162       if(guy.getScore() < pivot.getScore()):
163         less.append(guy)
164       elif(guy.getScore() == pivot.getScore()):
165         equal.append(guy)
166       else:
167         greater.append(guy)
168
169     return self._quicksort_(less) + equal + self._quicksort_(
          greater)
```

•Classe Fitness (fitness.py)

```
1  import os, subprocess, sys, socket, time, struct, random, traci,
       random, copy
2  import config
3
4  class Fitness:
```

```
5
6   def __init__(self, guy):
7     self.guy = guy
8     self.config = config.Config()
9
10  def evaluate(self, GUI=False):
11
12    global_step = int(1)
13    lights_config = self.guy.lights
14    cycle= [[],[]]
15
16    lights_id = self.config.LIGHTSID
17    passed_cycles = [int(0),int(0)]
18    detectors = self.config.DETECTORSID
19
20    detectors_registry = [[],[]]
21    detectors_jam = [[],[]]
22    detectors_count = [[],[]]
23    cycles_saturation = [[],[]]
24    lane_saturation = [[],[]]
25    medium_values = [[],[]]
26
27    for x in range(0, len(self.config.LIGHTSID)):
28      cycle[x] = dict([(lights_config[x][1] + lights_config[x][2] +
            self.config.YELLOW_TIME, self.config.YELLOW_PHASEA), (
          lights_config[x][1] + self.config.YELLOW_TIME, self.config
          .RED_PHASE), (lights_config[x][1], self.config.
          YELLOW_PHASEB), (0, self.config.GREEN_PHASE)])
29
30    if(self.config.LOG):
31      logfile = open(self.config.LOGFILE, "w")
32
33    if(self.config.DEEPLOG):
34      deeplogfile = open(self.config.DEEPLOGFILE, "w")
35
```

```
36    for x in range(len(detectors)):
37     for i in range(len(detectors[x])):
38      detectors_registry[x].append([[],[]])
39      detectors_count[x].append([0, 0])
40      detectors_jam[x].append(0)
41      medium_values[x].append(0)
42
43    self.config.configSumo(GUI)
44
45    traci.init(self.config.PORT)
46
47    while(global_step <= self.config.SIMULATION_TIME):
48
49     step_lights = [int(global_step % self.config.CYCLE_TIME), int
          ((global_step + self.config.OFFSET) % self.config.
          CYCLE_TIME)]
50
51     traci.simulationStep(global_step)
52
53     for x in range(0, len(lights_id)):
54
55      for time in sorted(cycle[x].keys(), reverse=True):
56
57       if(step_lights[x] >= time):
58        traci.trafficlights.setRedYellowGreenState(lights_id[x],
             cycle[x][time])
59        break
60
61      if(self.config.DEEPLOG):
62       print >> deeplogfile, "\nGLOBAL STEP %d: LIGHT STEP %d
           LIGHT: %s" % (global_step, step_lights[x], lights_id[x])
63
64      if(step_lights[x] == 0):
65       new_cycle = True
66       passed_cycles[x] += 1
```

```
67      else:
68       new_cycle = False
69
70      if(self.config.LOG and new_cycle and (passed_cycles[x] >
           self.config.IGNORECYCLES)):
71       print >> logfile, "CYCLE %d LIGHT: %s"%(passed_cycles[x],
           lights_id[x])
72
73      for i in range(0, len(detectors[x])):
74
75       '''
76       print str(detectors[x][i]) + "_IN"
77       print traci.inductionloop.getLastStepVehicleNumber(str(
           detectors[x][i]) + "_IN")
78       print str(detectors[x][i]) + "_OUT"
79       print traci.inductionloop.getLastStepVehicleNumber(str(
           detectors[x][i]) + "_OUT")
80       print traci.inductionloop.getLastStepVehicleIDs(str(
           detectors[x][i]) + "_OUT")
81       '''
82
83       detecIn = traci.inductionloop.getLastStepVehicleIDs(str(
           detectors[x][i]) + "_IN")
84       detecOut = traci.inductionloop.getLastStepVehicleIDs(str(
           detectors[x][i]) + "_OUT")
85
86       for car in detecIn:
87        if(not(car in detectors_registry[x][i][0])):
88         detectors_count[x][i][0] += 1
89         detectors_jam[x][i] = 0
90         detectors_registry[x][i][0].append(car)
91        else:
92         detectors_jam[x][i] += 1
93
94       for car in detecOut:
```

```
95          if(not(car in detectors_registry[x][i][1])):
96           detectors_count[x][i][1] += 1
97           detectors_registry[x][i][1].append(car)
98
99         if(detectors_jam[x][i] >= self.config.JAMDETECTION):
100         if(self.config.DEBUG):
101          print "JAM DETECTED ON LANE: %s"%(detectors[x][i])
102         detectors_count[x][i][0] += 1 * self.config.JAMPENALTY
103
104        if(self.config.DEBUG):
105         print detectors_count[x]
106
107        if(self.config.DEEPLOG):
108         print >> deeplogfile, "LANE: %s COUNT IN: %d OUT: %d" % (
               detectors[x][i], detectors_count[x][i][0],
               detectors_count[x][i][1])
109
110        if(new_cycle):
111
112         if(passed_cycles[x] > self.config.IGNORECYCLES):
113
114          if((detectors_count[x][i][0] > 0) and (detectors_count[x
                ][i][1]) > 0):
115           lane_saturation[x].append(float(detectors_count[x][i
                ][1]) / float(detectors_count[x][i][0]))
116          else:
117           if(detectors_count[x][i][0] <= 0):
118            lane_saturation[x].append(1)
119           else:
120            lane_saturation[x].append(0)
121
122          if(self.config.DEBUG):
123           print "LANE: %s OUT: %f IN: %f SAT: %f"%(detectors[x][i
                ], float(detectors_count[x][i][1]), float(
                detectors_count[x][i][0]), lane_saturation[x][i])
```

```
124
125         if(self.config.LOG):
126          print >> logfile, "\tLANE %s\n\tIN: %d\n\tOUT:%d\n\
                tSATURATION %f\n" % (detectors[x][i], detectors_count
                [x][i][0], detectors_count[x][i][1], lane_saturation[
                x][i])
127
128        detectors_count[x][i][0] -= detectors_count[x][i][1]
129        detectors_count[x][i][1] = 0
130        if(detectors_count[x][i][0] < 0):
131         detectors_count[x][i][0] =0
132
133      if(new_cycle and (passed_cycles[x] > self.config.
           IGNORECYCLES)):
134       if(self.config.DEBUG):
135        print "Saturacao x: %d i: %d"%(x,i)
136        print lane_saturation[x]
137       cycles_saturation[x].append(copy.deepcopy(lane_saturation[x
             ]))
138       lane_saturation[x] = []
139
140     global_step += 1
141
142   traci.close()
143
144   cycles_logged = [int(0),int(0)]
145
146   for light in range(0, len(cycles_saturation)):
147    for cycle in range(0, len(cycles_saturation[light])):
148     cycles_logged[light] += 1
149     for lane in range(0, len(cycles_saturation[light][cycle])):
150      medium_values[light][lane] += cycles_saturation[light][
           cycle][lane]
151      if(self.config.DEBUG):
152       print "L: %d C: %d L: %d V: %f \n"%(light, cycle, lane,
```

```
            cycles_saturation[light][cycle][lane])
153
154     for light in range(0, len(medium_values)):
155      for lane in range(0, len(medium_values[light])):
156       medium_values[light][lane] = medium_values[light][lane] /
              cycles_logged[light]
157
158     medium_values = copy.deepcopy(medium_values[0] + medium_values
            [1])
159     score = 0.0
160
161     for value in range(0, len(medium_values)):
162      score += medium_values[value]
163
164     return score
```