Dissertação de Mestrado

Problemas de Alocação de Tráfego Sujeitos a Congestionamento

por

Felipe Figueiredo Cardoso

Novembro de 2010

Felipe Figueiredo Cardoso

Problemas de Alocação de Tráfego Sujeitos a Congestionamento

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais como requisito parcial à obtenção do título de Mestre em Engenharia Elétrica.

Orientador: Frederico R. B. Cruz
Co-orientador: Oriane Magela Neto

Universidade Federal de Minas Gerais
Belo Horizonte, novembro de 2010

"Problemas de Alocação de Tráfego Sujeitos a Congestionamento"

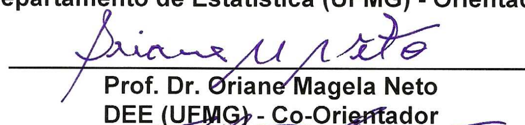
Felipe Figueiredo Cardoso

Dissertação de Mestrado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Mestre em Engenharia Elétrica.

Aprovada em 22 de novembro de 2010.

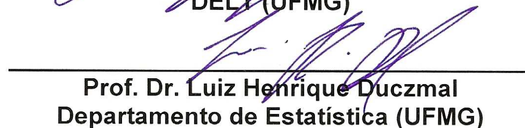
Por:

Prof. Dr. Frederico Rodrigues Borges da Cruz
Departamento de Estatística (UFMG) - Orientador



Prof. Dr. Oriane Magela Neto
DEE (UFMG) - Co-Orientador

Prof. Dr. Hani Camille Yehia
DELT (UFMG)



Prof. Dr. Luiz Henrique Duczmal
Departamento de Estatística (UFMG)

Resumo

Examinamos neste trabalho o modelo *System Optimum* de Wardrop (SO) para alocação de tráfego em uma rede de transportes. A formulação SO é equivalente a uma situação em que os usuários cooperam entre si, com o objetivo de minimizar o custo global de deslocamento na rede. Estes custos são normalmente expressos em termos de tempo para o deslocamento e são tipicamente fornecidos por fórmulas clássicas. Neste trabalho, temos como objetivo investigar uma expressão para o tempo de deslocamento que é baseada em redes de filas $M/G/c/c$ dependentes do estado. Esta nova expressão não é convexa, ao contrário das expressões usuais, possuindo uma forma de S. Como consequência, o modelo SO pode apresentar soluções ótimas locais múltiplas, o que justifica a utilização de algoritmos heurísticos, em geral, e do algoritmo *Differential Evolution* (DE), em particular. Apresentamos resultados computacionais para mostrar a eficiência e a eficácia da abordagem proposta.

Palavras-chave: tráfego, congestionamento, dependência de estado, redes.

Abstract

In this text, we examine the *Wardrop System Optimum* (SO) problem. The SO formulation is equivalent to a situation in which users cooperate with each other in order to minimize the overall travel cost. Usually, the travel costs are expressed in terms of times and are typically given from classical formulas. In this text we aim to investigate an $M/G/c/c$ state-dependent queueing network based formula, which is is not convex but S-shaped. As a consequence multiple solutions may be present for the SO, which justifies the use heuristic procedures such as a Differential Evolution (DE) algorithm. Computational results are present to show the efficacy and efficiency of the approach.

Keywords: traffic, congestion, state-dependency, networks.

Sumário

Resumo iv

Abstract v

Lista de Figuras viii

Lista de Tabelas ix

1 Introdução 1

1.1 Motivação e Definição do Problema 1

1.2 Objetivos e Escopo da Dissertação 4

1.3 Contribuições 5

1.4 Organização da Dissertação 5

2 Análise de Desempenho em Redes de Tráfego 7

2.1 Teoria de Filas 7

2.2 Modelagem de um Arco Único 9

2.3 Modelagem de Redes de Filas - Caso Série 12

2.3.1 Estágio 1 - Reconfiguração da Rede 13

2.3.2 Estágio 2 - Estimação de Parâmetros 14

2.3.3 Estágio 3 - Eliminação da Retro-alimentação e Atualização 15

2.4 Modelagem de Redes de Filas - Caso Fusão e Divisão 19
2.5 Considerações Finais 20

3 Algoritmo de Otimização **21**
3.1 Introdução 21
3.2 Características do *Differential Evolution* 23
3.3 Algoritmo *Differential Evolution* 24

4 Experimentos Computacionais **29**
4.1 Descrição dos Experimentos 29
4.2 Resultados e Discussão 30

5 Conclusões **36**
5.1 Obervações Finais 36
5.2 Tópicos para Trabalhos Futuros 37

Referências Bibliográficas **38**

A Códigos em C++ **42**

B Arquivo de Definição dos Parâmetros do DE **69**

C Arquivo de Definição dos Serviços **70**

D Arquivo de Entrada **71**

Lista de Figuras

1.1 Algoritmo para alocação de tráfego pelo modelo SO 4

2.1 Distribuições empíricas para o tráfego de veículos (Drake et al., 1967; Edie, 1961; Greenshields, 1935; TRB, 1985; Underwood, 1961) e modelos $M/G/c/c$ dependentes do estado (Jain & Smith, 1997) 10

2.2 Método da expansão generalizado (GEM) 13

2.3 Taxa de chegada externa λ_{ext} 15

2.4 Algoritmo de avaliação de desempenho: pré-avaliação 17

2.5 Algoritmo de avaliação de desempenho: avaliação 18

2.6 Fluxos de veículos em uma pista de uma milha (Cruz et al., 2010) 19

3.1 Algoritmo *Differential Evolution* simples 25

4.1 Rede de três trechos e seu correspondente modelo $M/G/c/c$ dependente do estado 30

4.2 Congestionamento exponencial (Jain & Smith, 1997) 33

Lista de Tabelas

1.1 Notação básica de redes 3

4.1 Configuração da rede de três trechos 31

4.2 Alocação ótima de tráfego para a rede de três trechos 32

4.3 Variação do parâmetro tamanho da população para a rede de três trechos 34

4.4 Variação do parâmetro fator de escala de mutação para a rede de três trechos 35

4.5 Variação do parâmetro constante de cruzamento para a rede de três trechos 35

Capítulo 1

Introdução

1.1 Motivação e Definição do Problema

Os problemas de alocação de tráfego ocorrem em várias situações encontrados na sociedade atual.

Exemplo 1.1 *Um exemplo interessante é o de um sistema para se produzir um automóvel. É necessário um grande número de estações de trabalho, como o almoxarifado para armazenamento de materiais, as diversas linhas de montagem, o centro de inspeção do controle de qualidade e o pátio de armazenagem final, entre outras. Um serviço lento em qualquer uma dessas etapas pode reduzir a utilização de outros centros de trabalho ao longo da linha de produção levando à sua ociosidade (*starvation*) enquanto espera-se pela próxima entrada. Uma maior eficiência no roteamento pode evitar os ramos congestionados e aproveitar melhor aqueles folgados, ajudando a reduzir importantes medidas de desempenho, tais como o tempo de trabalho (*sojourn time*), o trabalho em processo (*work in process*), entre outras.*

Na literatura sobre problemas de alocação de tráfego em uma rede congestionada, dois modelos são comumente empregados, quais sejam o modelo *System Optimum* de Wardrop (SO) e o modelo *User Equilibrium* (UE) (Prashker & Bekhor, 2000). Neste trabalho estaremos interessados apenas no modelo SO, uma vez que ele provê resultados bem sucedidos em redes de transporte. O modelo SO assume que todos os usuários são capazes de cooperar entre si, a fim de minimizar os custo de viagem no sistema como um todo. A notação básica de redes utilizada é apresentada na Tabela 1.1.

O modelo SO é formulado como:

$$\min z(\mathbf{x}) = \sum_a x_a c_a(x_a),$$

sujeito a:

$$\begin{aligned} \sum_k f_k^{rs} &= q^{rs}, \; \forall\, r, s, \\ x_a &= \sum_r \sum_s \sum_k f_k^{rs} \delta_k^{rs}, \; \forall\, a, \\ f_k^{rs} &\geq 0, \; \forall\, k, r, s. \end{aligned}$$

Pode-se demonstrar que a solução ótima é atingida quando são iguais os custos *marginais* de viagem, em cada caminho que transporta algum fluxo positivo, o que em outras palavras é:

$$f_k^{rs}(g_k^{rs} - g^{rs*}) = 0, \quad g_k^{rs} - g^{rs*} \geq 0, \quad \forall\, r, s,$$

onde g_k^{rs} é o custo marginal na rota k e g^{rs*} é o custo marginal ótimo, ambos entre o par r-s origem-destino.

Pesquisadores na área de atribuição de tráfego tem sido bem sucedidos na resolução do modelo SO e similares por meio de muitos algorit-

Tabela 1.1: Notação básica de redes

Variable	Description
$\mathcal{N}$	conjunto (índices) de nós
$\mathcal{A}$	conjunto (índices) de arcos
$\mathcal{R}$	conjunto de nós origem; $\mathcal{R} \subseteq \mathcal{N}$
$\mathcal{S}$	conjunto de nós destino; $\mathcal{S} \subseteq \mathcal{N}$
$\mathcal{K}_{rs}$	conjunto de caminhos que conecta o par r-s origem-destino (O-D); $r \in \mathcal{R}$, $s \in \mathcal{S}$;
x_a	fluxo no arco a; $\mathbf{x} = (\dots, x_a, \dots)$
$c_a(x_a)$	tempo de viagem no arco a; $\mathbf{c}(\mathbf{x}) = (\dots, c_a(x_a), \dots)$
f_k^{rs}	fluxo no caminho k que conecta o par O-D r-s; $\mathbf{f}^{rs} = (\dots, f_k^{rs}, \dots)$; $\mathbf{f} = (\dots, \mathbf{f}^{rs}, \dots)$
c_k^{rs}	tempo de viagem no caminho k que conecta o par r-s O-D; $\mathbf{c}^{rs} = (\dots, c_k^{rs}, \dots)$; $\mathbf{c} = (\dots, \mathbf{c}^{rs}, \dots)$
q^{rs}	demanda entre origem r e destino s; $(\mathbf{q})^{rs} = q^{rs}$
$\delta_{a,k}^{rs}$	variável indicadora: $\delta_{a,k}^{rs} = \begin{cases} 1, & \text{se o arco } a \text{ está no caminho } k \text{ entre o par } r\text{-}s\ O\text{-}D, \\ 0, & \text{caso contrário;} \end{cases}$ $(\mathbf{\Delta}^{rs})_{a,k} = \delta_{a,k}^{rs}$; $\mathbf{\Delta} = (\dots, \mathbf{\Delta}^{rs}, \dots)$

mos. Resultados animadores foram relatados tanto com algoritmos exatos duais (Hearn & Lawphongpanich, 1990), paralelos (Ho, 1990) e baseados em relaxação lagrangeana (Larsson & Patriksson, 1995), como também com heurísticas (Ceylan & Bell, 2005), incluindo algoritmos evolucionários (Cruz et al., 2010). O algoritmo da Figura 1.1 é baseado em um esquema simples e de fácil implementação. O algoritmo encontra iterativamente as probabilidades de roteamento p_a e a medida de desempenho da rede (tempo

```
algoritmo
    leia grafo, G(𝒩, 𝒜)
    leia taxas de chegadas, λ_n, ∀ n ∈ 𝒩
    leia comprimento dos arcos, l_a, ∀ a ∈ 𝒜
    repita
        /* gerar probabilidades de roteamento */
        gerar p_a, ∀ a ∈ 𝒜
        /* calcular fluxo e tempo de percurso */
        calcular x_a, c_a(x_a), ∀ a ∈ 𝒜
        /* calcular função objetivo */
        calcular ∑_a x_a c_a(x_a)
    até convergência ser atingida
    escrever p_a^(opt), ∀ a ∈ 𝒜
fim algoritmo
```

Figura 1.1: Algoritmo para alocação de tráfego pelo modelo SO

de percurso), para a minimização da função objetivo (tempo de percurso total). A heurística de otimização necessita estimar os fluxos e os tempos de viagem correspondentes, x_a e $c_a(x_a)$, respectivamente. Para encontrar estas estimativas, utilizaremos um modelo de redes de filas finitas recentemente proposto na literatura Cruz et al. (2010), conforme descrito nas próximas seções.

1.2 Objetivos e Escopo da Dissertação

O objetivo desta dissertação é o estudo de problemas de alocação de tráfego em que os custos de viagem são modelados pela teoria de filas finitas e a alocação ótima é encontrada por meio de algoritmos heurísticos. Para esta

dissertação são utilizadas as filas do tipo $M/G/c/c$ dependentes do estado, para a modelagem das redes de tráfego. De acordo com a conhecida notação de Kendall (Kendall, 1953), M representa um processo de chegadas markoviano, G, tempos de serviço com distribuição geral (e, neste caso, dependentes do estado), c é o número de servidores em parelo e o c final é a capacidade total do sistema, *incluindo* aqueles itens em serviço, o que é equivalente a dizer que não há áreas de espera (do inglês, *buffers*). O algoritmo heurístico de otimização utilizado é o conhecido *Differential Evolution* (DE), adaptado especialmente para o problema de alocação de tráfego aqui tratado.

1.3 Contribuições

Dentre as principais contribuições desta dissertação, podemos citar:

- apresentação de uma revisão bibliográfica na área, incluindo os trabalhos mais relevantes e mais recentemente publicados nesta linha de pesquisa;
- apresentação e discussão dos modelos de tempos de percurso;
- apresentação do algoritmo DE;
- condução e análise de experimentos computacionais com redes de transporte.

1.4 Organização da Dissertação

Esta dissertação está organizada da seguinte forma. No Capítulo 2 apresentamos conceitos fundamentais sobre a análise de desempenho em redes de tráfego. No Capítulo 3 apresentamos o algoritmo de otimização DE. No

Capítulo 4 apresentamos os resultados computacionais da validação do algoritmo de otimização DE. No Capítulo 5 são apresentadas as conclusões sobre a dissertação.

Capítulo 2

Análise de Desempenho em Redes de Tráfego

2.1 Teoria de Filas

Teoria das filas é um ramo do conhecimento, dentro da área de probabilidade, que lida com as filas de espera por meio de modelos matemáticos que descrevem o comportamento destes sistemas. Uma fila ocorre sempre que a procura por um determinado serviço é maior que a capacidade de o sistema prover este serviço. O modelo teórico é formado por servidores e filas de espera interligados. Os servidores provêm os serviços aos clientes que estão presentes nas filas e aqueles clientes que estão esperando pelo serviço são ditos estar na fila de espera.

Existem diversas aplicações da teoria das filas, que podem ser encontradas na literatura de probabilidade, pesquisa operacional e engenharia industrial, tais como controle de tráfego de aviões, escalonamento de processos em um sistema operacional e prestação de serviços em bancos. Os modelos de filas de espera consistem em expressões e relações matemáticas que podem ser usadas

para determinar as características de operação (ou medidas de desempenho) para uma fila de espera. Características de operação de interesse incluem a probabilidade de o sistema estar vazio, o número médio de pessoas na fila, o número médio de pessoas no sistema (a soma das pessoas em espera com aqueles em serviço), tempo médio de espera no sistema, entre outros. Engenheiros e administradores que possuem tal informação estão em melhor posição para uma tomada de decisão que balanceie o nível desejado de serviço com o custo de prover este serviço.

De acordo com a notação de Kendall (1953) um sistema de filas pode ser descrito por até seis características, de acordo com a seguinte notação $A/S/m/K/N/Q$:

A - **Processo de chegada:** Indica qual o padrão de chegada dos clientes. Pode ser determinístico, porém na maioria das aplicações é estocástico sendo representado por alguma distribuição, como a de Poisson ou a de Erlang.

S - **Distribuição do tempo de serviço:** Indica o padrão do serviço oferecido pelos servidores. Também pode ser determinístico ou estocástico, bem como pode atender clientes individualmente ou em grupos.

m - **Número de servidores:** É o total de servidores que podem atender os clientes simultaneamente.

K - **Capacidade do sistema:** É o número máximo de clientes que o sistema suporta, isto é, *incluindo* aqueles que estão em serviço.

N - **Tamanho da população:** É o número de clientes potenciais que podem chegar ao sistema. Em muito casos, como aqui, é considerado

infinito (isto é, a população é aberta, em oposição às populações fechadas).

Q **- Disciplina de atendimento:** Define a regra para o atendimento de clientes. Duas políticas comuns são *primeiro a entrar primeiro a sair* (do inglês, *fist-in-fist-out*, FIFO) e fila de prioridades. Será considerada aqui a disciplina FIFO.

2.2 Modelagem de um Arco Único

Uma fila $M/G/c/c$ dependente do estado pode ser considerado um sistema adequado para descrever uma via simples em uma rede de transporte (Yuhaski & Smith, 1989). De fato, resultados empíricos comprovam que o tempo de serviço em uma rede de tráfego depende do número de usuários presentes no sistema (vide Figura 2.1). O número de servidores pode ser considerado igual à capacidade do sistema (número de clientes que ele suporta). Logo, não existe espaço de espera.

As probabilidades limites para o número aleatório N de entidades em um modelo de fila $M/G/c/c$ dependente do estado, $p_n \equiv \Pr[N = n]$, são dadas por:

$$p_n = \left\{ \frac{\left[\lambda E[T_1]\right]^n}{n! f(n) f(n-1) \cdots f(2) f(1)} \right\} p_0,$$

em que $n = 1, 2, \ldots, c$, com c igual à capacidade do sistema, λ é a taxa de chegada, $E[T_1] = l/V_1$ é o tempo de serviço esperado para um veículo que trafega sozinho em um arco de comprimento l, considerando que V_1 é a velocidade de um veículo trafegando livre (sem redução na velocidade, causada por congestionamento), p_0 é a probabilidade de o sistema estar vazio

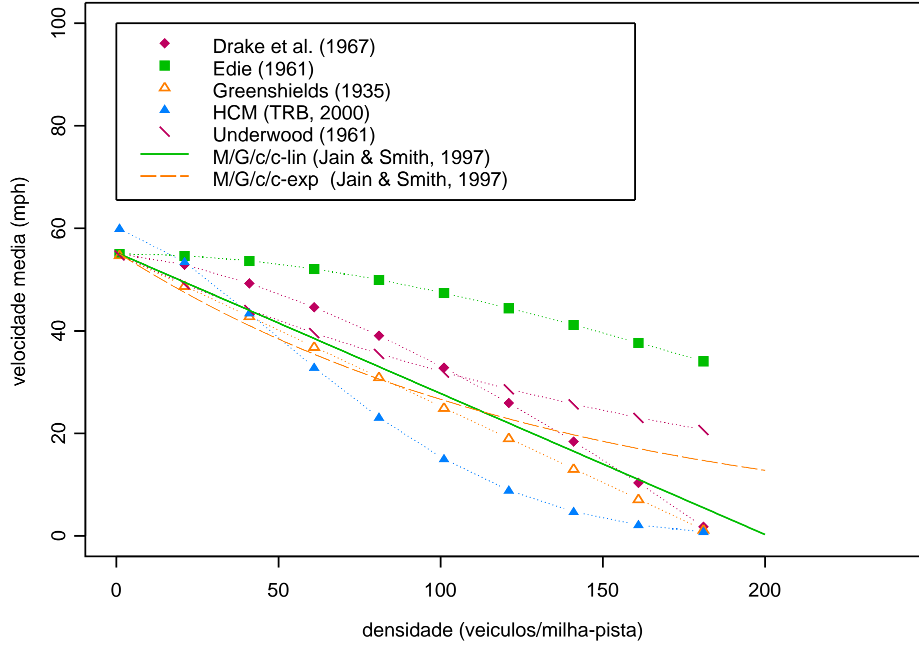


Figura 2.1: Distribuições empíricas para o tráfego de veículos (Drake et al., 1967; Edie, 1961; Greenshields, 1935; TRB, 1985; Underwood, 1961) e modelos $M/G/c/c$ dependentes do estado (Jain & Smith, 1997)

e $f(n)$ é a taxa de serviço.

A expressão abaixo define p_0:

$$p_0^{-1} = 1 + \sum_{i=1}^{c} \left\{ \frac{\left[\lambda E[T_1]\right]^i}{i! f(i) f(i-1) \cdots f(2) f(1)} \right\}.$$

A capacidade do sistema, c, é dada por:

$$c = \lfloor klw \rfloor,$$

em que l é o comprimento do arco (em milhas), w é a largura (em número

de pistas), k é a capacidade deste arco (por unidade de comprimento por pista) e $\lfloor x \rfloor$ é o maior inteiro não superior a x. Considerando aplicações relacionadas ao tráfego de veículos, k representa o parâmetro de densidade de tráfego (em veículos/milha-pista), situando-se normalmente entre 185-265 veículos/milha-pista.

Finalmente, a taxa de serviço é definida como $f(n) = V_n/V_1$, isto é, a razão da velocidade média de n veículos e a velocidade de um veículo que trafega livre. O objetivo é representar o efeito de decaimento da taxa de serviço com o aumento do número de usuários no sistema. Dentre os modelos possíveis, o modelo exponencial apresenta resultados satisfatórios. Ele pode ser definido por:

$$f(n) = \exp\left[-\left(\frac{n-1}{\beta}\right)^{\gamma}\right],$$

com

$$\gamma = \log\left[\frac{\log(V_a/V_1)}{\log(V_b/V_1)}\right] / \log\left(\frac{a-1}{b-1}\right),$$

e

$$\beta = \frac{a-1}{[\log(V_1/V_a)]^{1/\gamma}} = \frac{b-1}{[\log(V_1/V_b)]^{1/\gamma}}.$$

Os valores a e b são pontos arbitrários usados para ajustar a curva exponencial. Em aplicações relacionadas ao tráfego de veículos, valores comuns são $a = 20lw$ e $b = 140lw$, que correspondem às densidades de 20 e 140 veículos/milha-pista respectivamente. Valores razoáveis para estes pontos são $V_a = 48$ mph e $V_b = 20$ mph, conforme pode ser conferido na Figura 2.1.

As medidas de desempenho são deduzidas diretamente das probabilidades limite e são:

$$\begin{cases} p_c &= \Pr[N = c], \\ \theta &= \lambda(1 - p_c), \\ L &= E[N] = \sum_{n=1}^{c} n p_n, \\ W &= E[T] = L/\theta, \end{cases}$$

onde p_c é a probabilidade de bloqueio, $\theta \equiv x_a$ é a vazão em veículos/h, L é o número esperado de veículos no arco e $W \equiv c_a(x_a)$, aqui obtida pela *Lei de Little*, é o tempo de serviço esperado (em horas).

Uma observação importante é que a Lei de Little relaciona o número médio de clientes no sistema com o tempo médio despendido nele. O número médio de clientes é igual a taxa de chegada multiplicada pelo tempo médio de resposta ($L = \theta \times W$). Por sua vez, o tempo médio no sistema é igual ao número médio de clientes no sistema dividido pela taxa de chegada ($W = L/\theta$). Esta lei se aplica sempre que a entrada de clientes (excluídos aqueles bloqueados) é equivalente à saída.

2.3 Modelagem de Redes de Filas - Caso Série

Note que o problema está resolvido apenas parcialmente. A dedução de medidas de desempenho para filas $M/G/c/c$ dependentes do estado *configuradas em redes* é uma tarefa consideravelmente mais complexa, devido às probabilidades de roteamento que irão definir a entrada em cada fila e por causa dos efeitos de bloqueio mútuo. O método da expansão generalizado (GEM, do inglês *Generalized Expansion Method*) é uma ferramenta que foi proposta por Kerbache & Smith (1987), ainda nos anos 1980, com a finalidade de estimar aproximadamente medidas de desempenho de redes de filas

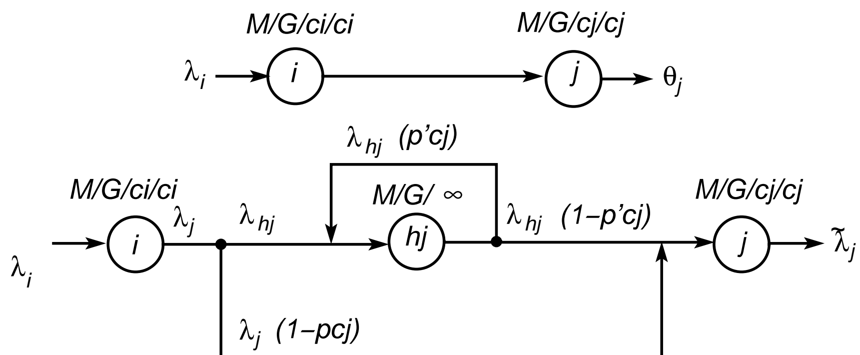


Figura 2.2: Método da expansão generalizado (GEM)

finitas. O método tem uma longa tradição nesta área e é uma combinação de métodos de tentativas repetidas e decomposição nó-a-nó, com a característica chave de adicionar um nó de espera artificial precedendo cada fila finita na rede, a fim de registrar consumidores bloqueados que tentam entrar neste nó finito, mas que são impedidos porque ele está com a sua capacidade esgotada. A Figura 2.2 mostra o exemplo de dois trechos de estrada, configurados em série, com a correspondente rede de filas. Pela adição do nó de espera, a rede de filas é expandida em uma rede de Jackson equivalente, na qual cada nó pode então ser decomposto e analisado separadamente, de maneira aproximada. Descrevemos brevemente agora o GEM, para filas em série.

O GEM consiste de três estágios que são realizados por cada nó finito na rede de filas original.

2.3.1 Estágio 1 - Reconfiguração da Rede

Para cada nó com capacidade finita, um nó artificial é adicionado precedendo-o diretamente, como mostra a Figura 2.2. Usuários que estão impe-

didos de moverem-se para o nó à frente (porque este nó está na sua capacidade máxima) são roteados para o nó artificial. A probabilidade de um usuário recém-chegado ser bloqueado pelo nó j é igual a p_{c_j}. Assim, com a probabilidade $(1 - p_{c_j})$, este usuário irá entrar no nó j. Logo, com a probabilidade p_{c_j}, ele irá entrar no nó de espera (h_j). O nó de espera é modelado como uma fila $M/G/\infty$ (logo, não existe espera para entrar neste nó).

Depois do serviço no nó h_j, o usuário poderá ser bloqueado novamente, com uma probabilidade, p'_{c_j}. Assim, com probabilidade $(1 - p'_{c_j})$, ele se dirigirá ao nó seguinte. Se bloqueado, o usuário deve refazer o caminho através do laço de retro-alimentação e voltar ao nó artificial, h_j, para um segundo período de atraso.

2.3.2 Estágio 2 - Estimação de Parâmetros

O valor de p_{c_j} pode ser determinado por resultados analíticos conhecidos. Para filas $M/G/c/c$ dependentes do estado, este valor é dado diretamente pelas medidas de desempenho, isto é, $p_{c_j} = \Pr\{N = c_j\}$.

Já o valor de p'_{c_j}, isto é, a probabilidade de um usuário completar sua primeira visita ao nó de espera (devido ao bloqueio do nó subsequente) e precisar de uma segunda visita (para um atraso adicional causado por um novo bloqueio) é determinado por métodos aproximados. Uma aproximação bastante eficaz, via técnicas de difusão (Labetoulle & Pujolle, 1980), é dada por:

$$p'_{c_j} = \left\{ \frac{\mu_j + \mu_{h_j}}{\mu_{h_j}} - \frac{\lambda\left[(r_2^{c_j} - r_1^{c_j}) - (r_2^{c_j-1} - r_1^{c_j-1})\right]}{\mu_{h_j}\left[(r_2^{c_j+1} - r_1^{c_j+1}) - (r_2^{c_j} - r_1^{c_j})\right]} \right\}^{-1}, \tag{2.1}$$

em que r_1 e r_2 são as raízes de:

$$\lambda_{\text{ext}} - (\lambda_{\text{ext}} + \mu_{h_j} + \mu_j)x + \mu_{h_j}x^2 = 0. \tag{2.2}$$

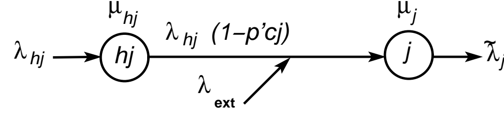


Figura 2.3: Taxa de chegada externa λ_{ext}

Definida com a ajuda da Figura 2.3, a taxa de chegada externa λ_{ext}, usada na Eq. (2.2), é dada por:

$$\begin{cases} \lambda_{\text{ext}} &= \tilde{\lambda}_j - \lambda_{h_j}(1 - p'_{c_j}), \\ \tilde{\lambda}_j &= \lambda_j(1 - p_{c_j}), \\ \lambda_{h_j} &= \lambda_j(p_{c_j}), \\ \lambda_j &= \lambda_i(1 - p_{c_i}) = \tilde{\lambda}_i. \end{cases} \tag{2.3}$$

Usando a teoria da renovação, pode ser mostrado que a taxa de serviço do nó de espera (no caso exponencial) é dada por (Kleinrock, 1975):

$$\mu_{h_j} = \frac{2\mu_j}{1 + \sigma_j^2 \mu_j^2}, \tag{2.4}$$

em que σ_j^2 é a variância do tempo de serviço. Porém, devido à taxa de serviço ser dependente do estado, uma suposição razoável é considerar o pior caso:

$$\mu_{h_j} = \mu_j \approx \frac{c_j}{E[T_1]/f(c_j)}, \tag{2.5}$$

em que c_j é o número máximo de servidores em paralelo e $E[T_1]/f(c_j)$ é o tempo de serviço para c_j ocupantes.

2.3.3 Estágio 3 - Eliminação da Retro-alimentação e Atualização

A reconfiguração do nó de espera é executada para remover as fortes dependências nos processos de chegada causadas pelas visitas repetidas (retro-

alimentação) ao nó artificial. O laço de retro-alimentação pode ser removido do nó de espera pela atualização da sua taxa de serviço, como se segue:

$$\mu'_h = (1 - p'_{c_j})\mu_{h_j}. \tag{2.6}$$

Finalmente, o tempo médio de serviço a que um usuário está submetido na fila i, precedente a uma fila finita j, pode ser corrigido pela seguinte expressão:

$$\tilde{\mu}_i^{-1} = \mu_i^{-1} + p_{c_j}(\mu'_h)^{-1}. \tag{2.7}$$

A Eq. (2.7) representa o passo final do GEM. O objetivo principal é fornecer um esquema de aproximação para atualizar as taxas de serviço em filas sucedidas por filas finitas, de modo a levar em conta todo o bloqueio causado após o serviço.

Uma implementação do GEM pode ser vista no algoritmo iterativo apresentado nas Figuras 2.4 e 2.5, recentemente proposto por Cruz & Smith (2007), quando foi usado pela primeira vez no contexto de modelagem de tráfego de veículos.

Primeiro, uma pré-avaliação é realizada, Figura 2.4. O algoritmo de avaliação de desempenho escolhe um nó arbitrário, j, do conjunto $\mathcal{N}$, mas não do conjunto Q (no qual Q é o conjunto de nó já avaliados). Assim, para todo arco $(i, j) \equiv a \in \mathcal{A}$ para o qual a extremidade i já tenha sido previamente avaliada, o nó j (a outra extremidade) tem calculada sua probabilidade de bloqueio $p_k^{(j)}$ e sua taxa de chegada, pela expressão:

$$\theta_j = \lambda_j \times \left(1 - p_k^{(j)}\right).$$

```
algoritmo
    obter as probabilidades de roteamento, p_ij, ∀ (i, j) ≡ a ∈ 𝒜
    inicializar conjunto de nós etiquetados, P ← ∅
    enquanto P ≠ 𝒩
        escolher j tal que (j ∈ 𝒩) e (j ∉ P)
        se {i| (i, j) ∈ 𝒜} ⊆ P então
            /* calcular medidas de performance */
            E[T_1]_j ← l_j/V_1
            calcular Pr[N = c_j]
            calcular θ_j
            calcular L_j, W_j
            /* informação para frente para sucessores */
            para ∀ k ∈ {k'| (j, k') ∈ 𝒜}
                λ_k ← λ_k + θ_j p_jk
            fim para
            /* etiquetar nó como pré-avaliado */
            P ← P ∪ {j}
        fim se
    fim enquanto
fim algoritmo
```

Figura 2.4: Algoritmo de avaliação de desempenho: pré-avaliação

Estas taxas de serviços são então passadas à frente, como taxas de chegada para os nós anteriores (se eles existirem), e o nó j é incluído no conjunto Q. Note que o passo de pré-avaliação é uma variante do algoritmo de caminho mínimo de Dijkstra (1959).

O algoritmo também inclui um passo de avaliação, Figura 2.5. Esta segunda parte do algoritmo procura a conservação de fluxo, definida por:

```
algoritmo
    inicializar o conjunto de nós etiquetados, P ← ∅
    inicializar máxima passagem pelos nós, θ_i^max ← ∞, ∀i ∈ 𝒩
    enquanto P ≠ V
        escolher i tal que (i ∈ 𝒩) e (i ∉ P)
        se {j| (i,j) ∈ 𝒜} ⊆ P então
            /* atualizar medidas de performance */
            E[T_1]_i^* ← min E[T_1]_i
                        s.a.: θ_i ≤ θ_i^max,
                              E[T_1]_i ≥ l_i/V_1
            calcular Pr[N = c_i], θ_i, L_i, W_i
            /* propagar de volta para predecessores */
            para ∀ k ∈ {k'| (k', i) ∈ 𝒜}
                atualizar θ_k^max
            fim para
            /* etiquetar nó como avaliado */
            P ← P ∪ {k}
        fim se
    fim enquanto
fim algoritmo
```

Figura 2.5: Algoritmo de avaliação de desempenho: avaliação

$$\theta_j \leq \lambda_j + \sum_{\forall\, i|(i,j)\equiv a\in\mathcal{A}} \theta_i p_{ij},\ \forall j \in \mathcal{N}.$$

É interessante observar que a etapa de avaliação é o algoritmo de Dijkstra (1959), trabalhando em reverso. Note-se também que o algoritmo de avaliação de desempenho deve ter disponível as probabilidades de roteamento p_a, antes que possa calcular todas as medidas de performance. Para o cálculo de p_a é usado o algoritmo *Differential Evolution*, a ser apresentado no Capítulo 3.

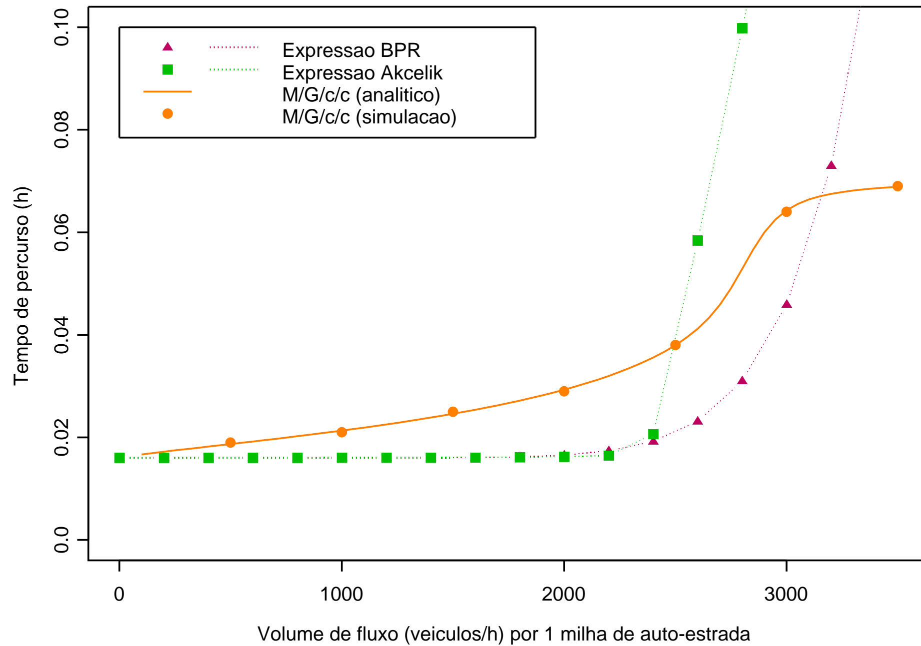


Figura 2.6: Fluxos de veículos em uma pista de uma milha (Cruz et al., 2010)

2.4 Modelagem de Redes de Filas - Caso Fusão e Divisão

Similarmente, o GEM pode ser estendido a redes em topologia fusão e divisão. Detalhes não serão dados aqui, mas podem ser facilmente encontrados na literatura (Jain & Smith, 1997). É importante ressaltar que as redes não são modificadas fisicamente. Os nós de espera são apenas nós artificiais incluídos no *software*, para possibilitar a análise aproximada do desempenho.

2.5 Considerações Finais

Em conclusão, para um dado vetor de probabilidade de roteamento, $\mathbf{p}$, não é difícil estimar a correspondente medida de desempenho de interesse, $\sum_a x_a c_a(x_a)$, que é a função objetivo a ser minimizada. O problema é que o uso de tempos de viagens dependentes do estado complicam o problema de otimização. De fato, como visto na Figura 2.6, as funções típicas de tempo de percurso (funções do tipo BPR, veja Bureau of Public Roads, 1964; Akçelik, 1991) diferem consideravelmente da função de percurso deduzida dos modelos de redes de filas $M/G/c/c$ dependentes do estado, especialmente sob tráfego pesado. Quando o tráfego é leve (abaixo de 1.000 veículos/hora), a abordagem via filas finitas é até próxima das funções clássicas, mas prevê saturação nas curvas de tempo de percurso (em forma de S), o que representa sérios problemas para qualquer algoritmo de otimização.

Capítulo 3

Algoritmo de Otimização

3.1 Introdução

Na matemática, a programação não-linear (Mateus & Luna, 1986) é o processo de resolver um sistema de equações e inequações, denominadas restrições, sobre um conjunto de variáveis, com uma função objetivo a ser maximizada ou minimizada, em que uma ou mais restrições ou função-objetivo são não-lineares.

Exemplo 3.1 *Um exemplo de problema de programação não-linear é a formulação seguinte:*

$$\min f(\mathbf{x}) = x_1^2 + x_2^2 + 2x_2,$$

sujeito a:

$$\begin{aligned} h_1(\mathbf{x}) &= x_1^2 + x_2^2 - 1 = 0, \\ g_2(\mathbf{x}) &= x_1 + 2x_2 - 0.5 \leq 0, \\ g_3(\mathbf{x}) &= x_1 \leq 0, \\ g_4(\mathbf{x}) &= x_2 \leq 0. \end{aligned}$$

Neste exemplo, a função-objetivo e a restrição de igualdade são não-lineares.

Existem diversos métodos para a resolução de problemas não-lineares. Alguns destes métodos são:

- busca unidirecional;
- minimização irrestrita (gradiente, Newton, quase-Newton);
- método das penalidades;
- método de barreira;
- primais;
- decomposição;
- algoritmos genéticos.

Neste trabalho é utilizado o algoritmo de otimização *Differential Evolution* (DE), pertencente à classe dos algoritmos genéticos, para resolução do modelo SO. As características do algoritmo DE são detalhadas nas próximas seções.

3.2 Características do *Differential Evolution*

O algoritmo genético de *annealing* (Price, 1994) foi o começo do algoritmo DE. O primeiro artigo sobre o *annealing* genético foi publicado em 1994. Trata-se de um algoritmo combinacional, baseado em populações, que realiza um critério de *annealing* (busca através de perturbações nas soluções). Price foi contatado por Storn, o qual tinha interesse no algoritmo *annealing*. Price modificou o algoritmo de *annealing* para utilizar números reais e operações aritméticas (mutação diferencial), em lugar de uma cadeias de bits e de operações lógicas. Assim o algoritmo *annealing* passou de um algoritmo combinacional para um otimizador contínuo. Ambos detectaram que a mutação diferencial combinada com o cruzamento aleatório e seleção por adequabilidade não necessitava do fator de *annealing*. Então o mecanismo de *annealing* foi removido e o algoritmo obtido é o *Differential Evolution*. As seguintes características do DE, válidas para problemas de otimização de espaço contínuo (Storn & Price, 1997), o justificam como um método de solução apropriado para a resolução do modelo SO:

- é simples, rápido e robusto;
- possui capacidade de otimização global elevada;
- pode ser implementado facilmente em um ambiente de computação paralela, o que acelera o processo de otimização;
- é efetivo em otimização não linear e pode ser facilmente adaptado para otimização com parâmetros mistos;
- não requer uma função objetivo diferenciável;
- opera em superfícies planas;

- pode prover múltiplas soluções em uma única execução.

O DE é uma versão de algoritmo genético, da classe dos algoritmos evolucionários (AE), que são baseados no princípio da sobrevivência do mais apto. Isto é basicamente um algoritmo de otimização e busca computadorizada baseada em populações. O DE difere de um AE na forma de uma mutação diferencial. Em um algoritmo evolucionário a mutação é baseada na saída de uma função de distribuição pré-definida, enquanto o DE usa a diferença de vetores-objeto aleatoriamente amostrados. Ao invés de usar apenas informação local de cada vetor-objeto (indivíduos da população), o DE muta todos os vetores-objeto com a mesma probabilidade. Deste modo o espaço de busca é coberto na tentativa de encontrar um ótimo global.

O método é definido como um método de busca direta e paralela que opera em uma população P_G de tamanho constante que é associada com cada geração G e consiste de NP vetores, ou soluções candidatas, $X_{pG}, p = 1, 2, \ldots, NP$. Cada vetor X_{pG} consiste de D variáveis de decisão $X_{o,p,G}, o = 1, 2, \ldots, D$. Os detalhes do algoritmo DE são apresentados na próxima seção.

3.3 Algoritmo *Differential Evolution*

A variação dos parâmetros do DE é definida pela notação $DE/x/y/z$ (Price & Storn, 2010), em que x especifica o vetor a sofrer mutação (`rand`, vetor escolhido aleatoriamente na população, ou `best`, o vetor com menor custo da população atual), y é o número de vetores-diferença usados e z especifica o esquema de cruzamento (`bin`, cruzamento através de experimentos binomiais independentes, ou `exp`, cruzamento através de experimentos exponencias). O algoritmo DE pode ser visto na Figura 3.1.

Um algoritmo DE simples (ver Figura 3.1) pode ser descrito pelos passos

algoritmo

inicializar:

D *- dimensão do problema*

NP, F, Cr *- parâmetros de controle*

GEN *- condição de parada*

L, H *- limite das restrições*

/ inicializar população */*

$Pop_{ij} \leftarrow rand_{ij}[L, H]$

*/*avaliar adequabilidade */*

$Fit_j \leftarrow f(Pop_j)\forall j$

para $g = 1$ **até** GEN **fazer**

para $j = 1$ **até** NP **fazer**

/ escolher índices aleatórios */*

$r_{1,2,3} \in [1, \ldots, NP], r_1 \neq r_2 \neq r_3 \neq j$

/ criar indivíduo tentativa */*

$X \leftarrow S(r, F, Cr, Pop)$

/ verificar limites das restrições */*

se $(x_i \notin [L, H])$ **então**

$x_i \leftarrow rand[L, H]$

fim se

/ selecionar melhor solução */*

se $(f(X_j) > Fit_j)$ **então**

$Pop_j \leftarrow X_j$

$Fit_j \leftarrow f\left(X_j\right)$

fim se

fim para

fim para

fim algoritmo

Figura 3.1: Algoritmo *Differential Evolution* simples

a seguir (Feoktistov, 2006).

(1) Escolher uma estratégia: É escolhida uma estratégia de acordo com a notação $DE/x/y/z$.

(2) Inicializar os parâmetros-chave de configuração: Os parâmetros de controle definidos pelo usuário, os quais permanecem constante durante o processo de busca são a constante de cruzamento CR, o tamanho da população NP, o fator de escala de mutação F e o máximo número de gerações $G_{\max}$.

(3) Inicializar a população: A população inicial é escolhida aleatoriamente entre os limites dos parâmetros que são definidos pelas restrições. Deve cobrir todo o espaço de busca.

(4) Avaliar a adequabilidade: Cada vetor tem avaliada sua adequabilidade e é encontrado aquele com a maior adequabilidade.

(5) Executar mutação: O DE gera novos vetores (vetores mutantes) pela adição da diferença ponderada entre dois ou mais vetores a um terceiro vetor (vetores alvo). Esta operação é denominada mutação. A mutação pretende manter uma população robusta e procurar uma nova área.

A mutação gera uma população de vetores de acordo com a equação:

$$v_{i,G+1} = x_{r1,G} + F \times (x_{r2,G} - x_{r3,G}),$$

em que $v_{i,G+1}$ é o vetor mutante, $x_{i,G}$, para $i = 1, 2, 3, \ldots, NP$, é o vetor alvo, $r_1, r_2, r_3 \in \{1, 2, 3, \ldots, NP\}$ são os índices aleatórios e $F \in [0, 2]$ é o fator de escala de mutação.

(6) Executar cruzamento: Os parâmetros dos vetores mutantes são misturados com parâmetros de vetores predeterminados, os vetores alvo, determinando a operação de cruzamento com a criação dos vetores tentativa. A constante de cruzamento controla a probabilidade de um vetor tentativa vir de um vetor mutante ou do vetor corrente (vetor alvo) e assim varia entre 0 e 1.

O vetor tentativa gerado pelo cruzamento é definido pelas equações:

$$u_{i,G+1} = (u_{1i,G+1}, u_{2i,G+1}, ..., u_{Di,G+1}),$$

$$u_{ji,G+1} = \begin{cases} v_{ji,G+1}, & \text{if } \texttt{rand()} \quad \leq CR, \\ x_{ij,G}, & \text{if } \texttt{rand()}() \quad > CR, \end{cases}$$

em que $u_{i,G+1}$ é o vetor tentativa, `rand()` $\in [0,1]$ é um número aleatório com distribuição uniforme e $CR \in [0,1]$ é a constante de cruzamento.

(7) Checar os limites superior e inferior das variáveis: Os parâmetros dos vetores tentativa devem ser checados para as condições de limite. Se um parâmetro excede alguma restrição de limite, um caminho é selecionar um novo valor aleatório porém adequado.

(8) Executar seleção: Para selecionar os vetores da próxima geração, cada vetor tentativa deve ser avaliado pela função objetivo e comparado com o valor da função objetivo do vetor alvo. Se a adequabilidade do vetor tentativa é maior que a adequabilidade do vetor alvo, o vetor tentativa passa a ser o novo vetor alvo, caso contrário o vetor alvo é retido para próxima geração. Como resultado todos os indivíduos da próxima geração são tão bons ou melhores que suas contra partes na geração atual.

A seleção é determinada pela equação:

$$x_{i,G+1} = \begin{cases} x_{i,G}, & \text{se } f(x_{i,G}) \geq f(u_{i,G+1}), \\ u_{i,G+1}, & \text{se } f(x_{i,G}) < f(u_{i,G+1}), \end{cases}$$

em que $x_{i,G+1}$ é o vetor alvo da próxima geração, $f(x_{i,G})$ é a função de adequabilidade para o vetor alvo e $f(u_{i,G+1})$ é a função de adequabilidade para o vetor tentativa.

(9) Repetir até a parada: O ciclo evolucionário, passos **(4)** a **(8)**, são executados até que a condição de parada seja atingida.

Capítulo 4

Experimentos Computacionais

4.1 Descrição dos Experimentos

Todos os algoritmos descritos anteriormente foram codificados em C++, pela sua eficiência e flexibilidade. O programa está disponível no Apêndice A, para fins educacionais e de pesquisa. Os parâmetros do DE são definidos de acordo com o Apêndice B. Os experimentos foram conduzidos em um computador pessoal, rodando o sistema operacional Windows© Vista.

A rede utilizada nos nossos experimentos pode ser vista na Figura 4.1. Esta é uma rede bastante simples, composta de três trechos de estrada. Os pontos A e B são conectados pelo trecho a_1 e dois trechos alternativos, a_2 and a_3. Uma destas rotas é mais longa (e, consequentemente, mais lenta) do que a outra. O ajuste do modelo $M/G/c/c$ dependente do estado é apresentado na Tabela 4.1 e os respectivos arquivos de definição encontram-se no Apêndice C. O algoritmo foi rodado para a rede de três trechos, para diferentes taxas de chegada λ. Pode ser visto no Apêndice D o arquivo de entrada para o caso $\lambda = 500$.

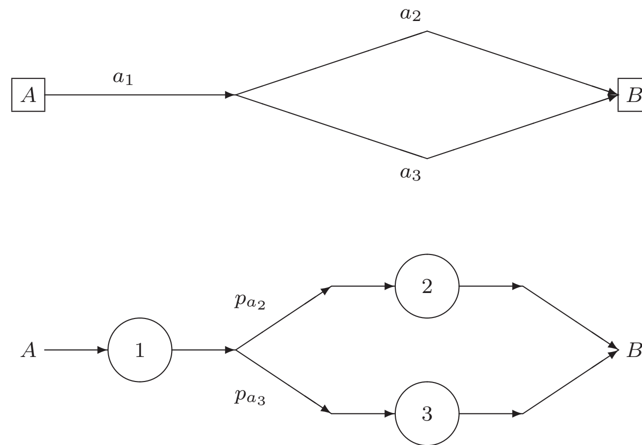


Figura 4.1: Rede de três trechos e seu correspondente modelo $M/G/c/c$ dependente do estado

4.2 Resultados e Discussão

Os resultados obtidos podem ser vistos na Tabela 4.2. Quando a taxa de chegada é nula, temos que o tempo esperado de percurso é igual ao tempo esperado para um único ocupante no sistema, que é a soma dos tempos esperados para ocupantes únicos nos respectivos trechos (em outras palavras, 0,1570 = 0,0320 + 0,1250 e 0,1245 = 0,0320 + 0,0925). A partir do valor de $\lambda = 500$, observamos um aumento no tempo esperado de percurso, que é causado pelo nível de congestionamento no sistema. Note que os tempos esperados de percurso para ambas as rotas nunca são iguais, o que significa que aqueles usuários com completo conhecimento dos tempos de percurso poderiam reduzir o próprio tempo de percurso mudando da rota lenta para a rota mais rápida. Tal melhoria seria impossível se estivéssemos diante da solução ótima do problema UE (veja Sheffi, 1985), mas o problema resolvido aqui é o SO, que busca a redução nos tempos de percurso para o sistema como um todo e não para tempos de percurso individuais.

Tabela 4.1: Configuração da rede de três trechos

Rota	Comprimento*	Largura†	$V_1^{\ddagger}$	$V_a^{\ddagger}$	$V_b^{\ddagger}$	c (vei)	$E[T_1]^{\sharp}$
a_1	0,80 (0,50)	5	25 (40)	23 (37)	10 (16)	800	0,0320 (115)
a_2	2,50 (1,55)	2	20 (32)	18 (29)	6 (10)	1.000	0,1250 (450)
a_3	1,85 (1,15)	2	20 (32)	18 (29)	6 (10)	740	0,0925 (333)

Obs.: *em milhas (km); †em # pistas; ‡em mph (km/h); ♯em h (s)

No que diz respeito à alocação ótima, observamos que o tráfego é prioritariamente direcionado para a rota mais rápida (isto é, aquela rota com o menor tempo de percurso esperado) e somente então é direcionado para a mais lenta. Esta alocação é exatamente a esperada, o que é encorajador. Para a rede testada, observamos que até o valor de taxa de chegada de 2.000 usuários por unidade de tempo (no caso aqui testado, este é o número de chegadas médio por hora), todo o tráfego é absorvido pela rede, praticamente sem nenhum bloqueio (isto é, aproximadamente 100% das chegadas são imediatamente atendidas, pois a soma das alocações em cada rota totalizam a taxa de chegada). Entretanto, acima de 2.000, a rede parece ter alcançado sua capacidade, uma vez que apenas uma fração do tráfego adicional consegue ser bem sucedido atravessando a rede.

Finalmente, gostaríamos de notar que um comportamento inesperado foi observado na rota composta pelos trechos a_1-a_3, qual seja o aumento no tempo de percurso (de 0,8964 h para 0,8970 h), apesar da redução no tráfego nela sofrida (de 1.225 usuários por hora para 1.224 usuários por hora). Este é um comportamento que seria impossível caso os tempos de percurso estivessem sendo modelados pelas tradicionais expressões do tipo BPR. Entre-

Tabela 4.2: Alocação ótima de tráfego para a rede de três trechos

λ	rota	alocação	$E[T]^*$
0	a_1-a_2	n/a	0,1570 (565)
	a_1-a_3	n/a	0,1245 (448)
500	a_1-a_2	152	0,1591 (573)
	a_1-a_3	348	0,1287 (463)
1.000	a_1-a_2	370	0,1635 (588)
	a_1-a_3	630	0,1341 (483)
2.000	a_1-a_2	890	0,1791 (645)
	a_1-a_3	1.110	0,1484 (534)
4.000	a_1-a_2	1.496	0,4742 (1.707)
	a_1-a_3	**1.225**	**0,8964 (3.227)**
8.000	a_1-a_2	1.507	0,4750 (1.710)
	a_1-a_3	**1.224**	**0,8970 (3.229)**

Obs.: *em horas (em segundos)

tanto, é um fenômeno perfeitamente razoável com a utilização dos modelos $M/G/c/c$ dependentes do estado, para modelagem dos tempos de percurso. De fato, conforme relatado em outros estudos Cardoso et al. (2010a,b), os modelos $M/G/c/c$ dependentes do estado induzem curvas *taxa de saída* versus *taxa de entrada* que alcançam um valor máximo, a partir do qual decaem antes de finalmente estabilizarem-se, conforme pode ser visto na Figura 4.2 (Jain & Smith, 1997).

As Tabelas 4.3, 4.4 e 4.5 exibem resultados para a variação dos parâmetros do algoritmo DE para a rede da Figura 4.1. São exibidas a média e o

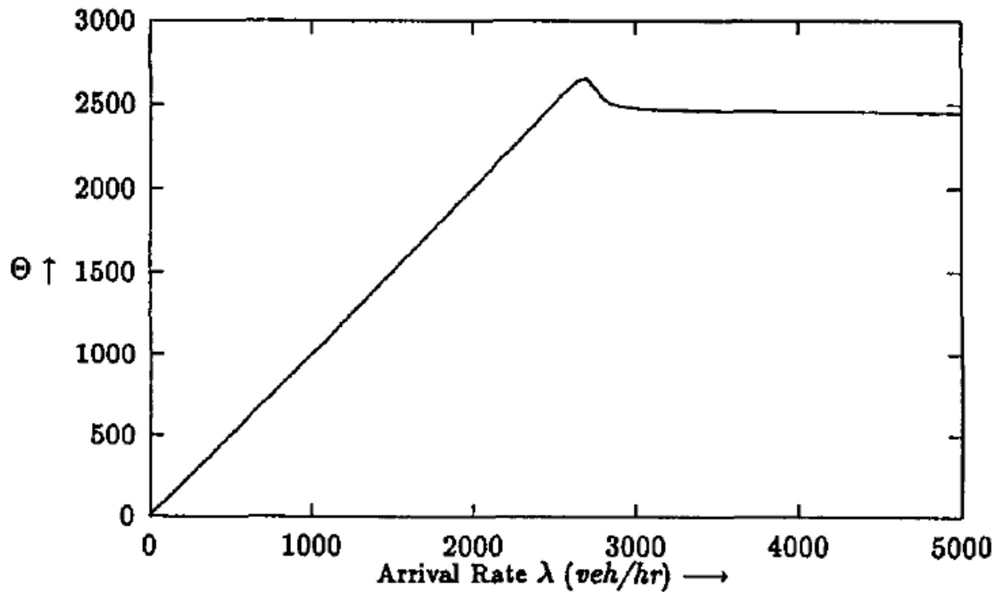


Figura 4.2: Congestionamento exponencial (Jain & Smith, 1997)

erro-padrão da média do número total de avaliações de função, para trinta execuções por variação dos parâmetros. É importante ressaltar que todas as execuções atingiram o resultado ótimo para o tempo total de percurso segundo o modelo SO para uma taxa de chegada de 1000 usuários. Os parâmetros iniciais para o algoritimo de otimização são os seguintes: tamanho da população igual a 8; fator de escala de mutação igual a 1; constante de cruzamento igual a 0,8. Na Tabela 4.3 ocorre a variação do tamanho da população, NP. Na Tabela 4.4, mostramos a variação do fator de escala de mutação, F. Finalmente, na Tabela 4.5, a variação da constante de cruzamento, CR.

O tamanho da população deve ser pequeno, como pode ser notado na Tabela 4.3. O menor número de avaliações de função ocorre quando o tamanho da população é de 8 indivíduos. O aumento do tamanho da população apenas aumenta o número de avaliações de função e, consequentemente, o

Tabela 4.3: Variação do parâmetro tamanho da população para a rede de três trechos

NP	média*	erro-padrão da média*
4	617	59
8	545	52
12	837	29
16	1.114	32
20	1.388	34
24	1.779	38
28	2.049	31
32	2.330	37
36	2.627	45
40	3.029	52

* sobre trinta avaliações de função.

tempo de execução do algoritmo. Quanto ao fator de escala de mutação, F, o melhor valor foi 0,25, porém notamos que o menor erro-padrão da média ocorre com o valor do fator um pouco maior igual a 0,50. Portanto, um fator um pouco maior, como o valor de 0,50 , piora um pouco a média de avaliações, ma fornece um resultado mais robusto. A constante de cruzamento deve possuir um valor próximo ao seu limite superior. A Tabela 4.5 mostra que o menor número de avaliações de função é obtido quando o valor da constante de cruzamento é de 0,9.

Tabela 4.4: Variação do parâmetro fator de escala de mutação para a rede de três trechos

F	média*	erro-padrão da média*
0,25	315	12
0,50	343	11
0,75	455	13
1,00	499	48
1,25	626	26
1,50	652	23
1,75	848	29
2,00	897	43

* sobre trinta avaliações de função.

Tabela 4.5: Variação do parâmetro constante de cruzamento para a rede de três trechos

CR	média*	erro-padrão da média*
0,0	1.551	139
0,1	1.209	83
0,2	968	76
0,3	911	47
0,4	719	46
0,5	738	59
0,6	648	47
0,7	547	32
0,8	540	48
0,9	452	23
1,0	453	24

* sobre trinta avaliações de função.

Capítulo 5

Conclusões

5.1 Obervações Finais

O modelo *System Optimum* de Wardrop (SO) foi tratado aqui por meio de uma expressão estocástica diferente para o tempo de percurso. Esta expressão é baseada em um modelo de filas finitas do tipo $M/G/c/c$ dependente do estado. Esta nova forma de modelar o tempo de percurso traz vantagens sobre as expressões comumente utilizadas, uma vez que ela está em melhor aderência com a realidade, por modelar os efeitos de congestionamento (isto é, a redução no tempo de percurso com o aumento do congestionamento nos trechos da rede). Por outro lado, a função resultante para o tempo de percurso tem formato S, o que traz novos desafios para os algoritmos de otimização do modelo SO, pela possibilidade de ótimos locais. Resultados computacionais aqui apresentados atestam que heurísticas do tipo *Differential Evolution* (DE) podem ser uma ferramenta efetiva na resolução de problemas SO, uma vez que os resultados obtidos fazem sentido. Adicionalmente, as soluções parecem ser robustas, como demonstrado pela análise de sensibilidade apresentada. Possivelmente o resultado mais surpreendente foi a

redução no número de usuários servidos (com um aumento no tempo de percurso), quando a taxa de entrada ao sistema cresce além da sua capacidade.

5.2 Tópicos para Trabalhos Futuros

Possíveis direções para trabalhos futuros nesta área de pesquisa incluem a análise de redes mais complexas e a aplicação dos algoritmos na modelagem de redes de pedestres, uma vez que também se aplicam ao tráfego de pedestres muitas das características da função de tempo de percurso aqui tratada.

Referências Bibliográficas

Akçelik, R. (1991). Travel time function for transport planning purposes: Davidson's function, its time dependent form and an alternative travel time function, *Australian Road Research* **21**(3): 49–59.

Bureau of Public Roads (1964). Traffic assignment manual, *Technical report*, U.S. Department of Commerce.

Cardoso, F. F., Neto, O. M. & Cruz, F. R. B. (2010a). Computational experience with a state-dependent traffic assignment problem, *Proceedings of the '4th Southern Conference on Computational Modeling - IV MCSUL'*, Rio Grande, Brazil, pp. 172–177.

Cardoso, F. F., Neto, O. M. & Cruz, F. R. B. (2010b). Traffic assignment under state-dependent travel times, *Proceedings of 'The 3rd International Conference on Power Electronics and Intelligent Transportation System (PEITS 2010)'*, Shenzhen, China, pp. 1–4.

Ceylan, H. & Bell, M. G. H. (2005). Genetic algorithm solution for the stochastic equilibrium transportation networks under congestion, *Transportation Research Part B* **39**: 169–185.

Cruz, F. R. B. & Smith, J. M. (2007). Approximate analysis of $M/G/c/c$

state-dependent queueing networks, *Computers & Operations Research* **34**(8): 2332–2344.

Cruz, F. R. B., van Woensel, T., Smith, J. M. & Lieckens, K. (2010). On the system optimum of traffic assignment in $M/G/c/c$ state-dependent queueing networks, *European Journal of Operational Research* **201**(1): 183–193.

Dijkstra, E. W. (1959). A note on two problems in connection with graphs, *Numerical Mathematics* **1**: 269–271.

Drake, J. S., Schofer, J. L. & May, A. D. (1967). A statistical analysis of speed density hypotheses, *Highway Research Record* **154**: 53–87.

Edie, L. C. (1961). Car following and steady-state theory, *Operations Research* **9**: 66–76.

Feoktistov, V. (2006). *Differential Evolution In Search of Solutions*, Springer, New York, USA.

Greenshields, B. D. (1935). A study of traffic capacity, *Highway Research Board Proceedings* **14**: 448–477.

Hearn, D. W. & Lawphongpanich, S. (1990). A dual ascent algorithm for traffic assignment problems, *Transportation Research Part B* **24**(6): 423–430.

Ho, J. K. (1990). Solving the dynamic traffic assignment problem on a hypercube multicomputer, *Transportation Research Part B* **24**(6): 443–451.

Jain, R. & Smith, J. M. (1997). Modeling vehicular traffic flow using $M/G/C/C$ state queueing models, *Transportation Science* **31**(4): 324–336.

Kendall, D. G. (1953). Stochastic processes occurring in the theory of queues and their analysis by the method of imbedded Markov chains, *Annals Mathematical Statistics* **24**: 338–354.

Kerbache, L. & Smith, J. M. (1987). The generalized expansion method for open finite queueing networks, *European Journal of Operational Research* **32**: 448–461.

Kleinrock, L. (1975). *Queueing Systems*, Vol. I: Theory, John Wiley & Sons, New York, NY, USA.

Labetoulle, J. & Pujolle, G. (1980). Isolation method in a network of queues, *IEEE Transactions on Software Engineering* **SE-6**(4): 373–381.

Larsson, T. & Patriksson, M. (1995). An augmented Lagrangean dual algorithm for link capacity side constrained traffic assignment problems, *Transportation Research Part B* **29**(6): 433–455.

Mateus, G. R. & Luna, H. P. L. (1986). *Programação Não-Linear*, V Escola de Computação, Belo Horizonte, Brazil.

Prashker, J. N. & Bekhor, S. (2000). Some observations on stochastic user equilibrium and system optimum of traffic assignment, *Transportation Research Part B* **34**: 277–291.

Price, K. (1994). Genetic annealig, *Dr. Dobb's Journal* pp. 127–132.

Price, K. & Storn, R. (2010). Differential evolution, *Website of DE as on August 2010*, International Computer Science Institute, University of California, Berkeley. **URL:** *http://www.icsi.berkeley.edu/˜storn/code.html*

Sheffi, Y. (1985). *Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods*, Prentice-Hall, Englewood Cliffs, NJ.

Storn, R. & Price, K. (1997). Differential evolution: A simple and efficient heuristic for global optimization over continuous spaces, *Journal of Global Optimization* **11**(4): 341–359.

Transportation Research Board (1985). Highway capacity manual, *Special Report 209*, National Research Council.

Underwood, R. T. (1961). Speed, volume, and density relationships: Quality and theory of traffic flow, *Yale Bureau of Highway Traffic* pp. 141–188.

Yuhaski, S. J. & Smith, J. M. (1989). Modeling circulation systems in buildings using state dependent models, *Queueing Systems* **4**: 319–338.

Apêndice A

Códigos em C++

Código A.1: mgccso-main.cpp

```
1  //
2  // Purpose:
3  //   to assign traffic in networks of MGCC queues
4  //
5  //
6  // Version:
7  //   3.0
8  //
9  // Date:
10 //   Nov/2010
11 //
12 #include <stdlib.h>
13 #include <stdio.h>
14 #include "functions.c"
15 #include "DESchemes.c"
16 #include "cmusr.cpp"
17 #include "mgccso.cpp"
18 int main(int argc, char *argv[]) {
19   // check input
20   if (argc < 2) {
21      fprintf(stderr, "mgccso-main:\t assigns traffic in MGcc networks\n");
22      fprintf(stderr, "Usage:\t mgccso-main [script_file]\n");
23      exit(0);
24   }
25   OpenDEpar();
26   int aux = 0;
27   while ((argv[1][aux]!='\n')&&(argv[1][aux]!='\0')) aux++;
28   argv[1][aux] = '\0';
29   FILE *inputFile = fopen(argv[1],'r');
30   if (inputFile == NULL) {
31         fprintf(stderr, "%s: No such file\n", argv[1]);
32      exit(0);
33   }
34   MgccNet myMgccNet;
35   myMgccNet.ReadData(inputFile);
36   myMgccNet.ShowNet();
37   const int LLENGTH = 256;
38   char Line[LLENGTH];
39   int i;
40   fgets(Line, LLENGTH, inputFile);
41   fprintf(stdout,"Final Results:\n");
42   determineDV();
43   createSpecimen(nPop);
44   PrepareSpecimen();
45   createHistory(Generations);
46   createPopulation();
```

```
47     createTMPPopulation();
48     SetScheme();
49     InitializePopulation();
50     (void) time(&t1);
51     nmut=0;
52     neval=0;
53     //initialize population
54     srand((unsigned)time(NULL));
55     srand(135790);
56     for(a1=0;a1<PNP-3;a1++) {
57            for(a2=0;a2<PD-1;a2++) {
58                 setPopulation(a2, a1,
59                      getSpecimen(0,a2)+
60                      ((double)rand()/RAND_MAX)*(getSpecimen(1,a2) - getSpecimen(0,a2))
61             );
62            }
63     }
64     for(l=0;l<Generations;l++) {
65            //evolution
66            for(a1=0;a1<PNP-3;a1++) {
67                 SetProb(a1,0);
68             myMgccNet.GetPerfMeasures(0);
69             myMgccNet.AggregatePerfMeasures();
70             setPopulation(PD-1,a1,sumd1);
71             neval++;
72             if(Scheme==1) {
73                      DER1B();
74                 } else {
75                      if(Scheme==2) {
76                           DER2B();
77                      } else {
78                           DECuRR();
79                      }
80             }
81             //best individual into new generation
82                 SetProb(PNP-3,1);
83                 int origin, dest;
84                 change=0;
85                 conNew=0;
86                 conOld=0;
87             sumNew=0;
88                 sumOld=0;
89             for(g=0;g<iKT;g++) {
90                           origin=ArcKT[g][0];
91                           dest=ArcKT[g][1];
92                           if(ArcKT[g][0]==ArcKT[g+1][0]) {
93                                sumNew+=fabs(prob[1][origin -1][dest -1]);
94                                sumOld+=fabs(prob[0][origin -1][dest -1]);
95                           } else {
96                                if(prob[1][origin -1][dest -1]!=1) {
97                                     sumNew+=fabs(prob[1][origin -1][dest -1]);
98                                     sumOld+=fabs(prob[0][origin -1][dest -1]);
99                                     sumNew--;
100                                    sumOld--;
101                                    maxNew=maximum(sumNew,0);
102                                    maxOld=maximum(sumOld,0);
103                                    if(maxNew < maxOld) {
104                                         con1++;
105                                         conNew+=maxNew;
106                                         conOld+=maxOld;
107                                    }
108                                    sumNew=0;
109                                    sumOld=0;
110                               }
111                          }
112                }
113            if(conNew > 0) {
114                          change=1;
115                } else {
116                          if(conOld>0) {
117                               change=1;
118                               myMgccNet.GetPerfMeasures(1); //New
119                               myMgccNet.AggregatePerfMeasures();
120                               setPopulation(PD-1,PNP-3,sumd1);
121                               neval++;
122                          } else {
123                               myMgccNet.GetPerfMeasures(1); //New
124                               myMgccNet.AggregatePerfMeasures();
125                               setPopulation(PD-1,PNP-3,sumd1);
126                               neval++;
127                               if(getPopulation(PD-1,a1)==0) {
128                                    myMgccNet.GetPerfMeasures(0); //Old
129                                    myMgccNet.AggregatePerfMeasures();
```

```
130                                 setPopulation(PD-1,a1,sumd1);
131                                 neval++;
132                             }
133                             if(getPopulation(PD-1,PNP-3)≤getPopulation(PD-1,a1)) {
134                                 change=1;
135                             }
136                         }
137                 }
138                 if(change==1) {
139                     nmut++;
140                     for(a3=0;a3<PD;a3++) {
141                     setTMPPopulation(a3,a1,getPopulation(a3,PNP-3));
142                     }
143                 } else {
144                 for(a3=0;a3<PD;a3++) {
145                     setTMPPopulation(a3,a1,getPopulation(a3,a1));
146                 }
147                 }
148                 setPopulation(PD-1,PNP-3,0);
149             }
150             //new population replace old population
151         for(a1=0;a1<PNP;a1++) {
152             for(a2=0;a2<PD;a2++) {
153                 setPopulation(a2,a1,getTMPPopulation(a2,a1));
154              }
155           }
156             //search for the best individual
157             WorstIndividual();
158             BestIndividual();
159             if (hh-h≥Stagnation[0]-Precision*(hh-h) &&
160                 hh-h≤Stagnation[0]+Precision*(hh-h)) {
161                 Stagnation[1]++;
162             } else {
163                 Stagnation[0]=h-hh;
164                 Stagnation[1]=0;
165             }
166             //append to History
167 /*
168             system("cls");
169             printf("Scheme = %d\n", Scheme);
170             printf("Generation = %d\n", l+1);
171             printf("Population\n");
172             for (a1=0;a1<PNP-3;a1++) {
173              for (a2=0;a2<PD;a2++) {
174                 printf("%lf\t", Population[PD * a1 + a2]);
175              }
176              printf("\n");
177           }
178             printf("Best individual = %lg\n", h);
179             printf("Worst individual = %lg\n", hh);
180 */
181             History[l]=h;
182             ldef=l;
183             if(h != 0 && hh != 0) {
184                 if( ((hh-h)/fabs(hh)<Precision)||(Stagnation[1]>10000) ) {
185                     goto stoprun1;
186                 }
187             } else {
188                 if(h == 0) {
189                     Stagnation[2]++;
190                     if(Stagnation[2]>1000000) {
191                         goto stoprun1;
192                     }
193                 }
194             }
195     }//Generations
196     stoprun1:
197     (void) time(&t2);
198     WritePOP(MutScheme);
199     SetProb(b,0);
200     WriteOut(MutScheme);
201     WriteHistory();
202     destroyTMPPopulation();
203     destroyPopulation();
204     destroyHistory();
205     destroySpecimen();
206     while (fscanf(inputFile, "%d/n", &i) == 1 ) {
207             myMgccNet.ShowPerfMeasures(i-1);
208     }
209     fclose(inputFile);
210     return 0;
211 }
```

Código A.2: functions.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <math.h>
4  #include <time.h>
5  int     ArcKT[100][2];
6  int     iKT=0;
7  float   ***prob;
8  double *Population, *TMPPopulation, *PRTV, *Specimen, *History;
9  int     PD, PNP, i, a3, a4, b, bb, s, l, ldef, n, neval, nmut, a1, a2, NP, NPi,
10         Scheme, D, smc2, V1, V2, V3, V4, V5, Generations;
11 double h, hh, ckl1, F, F1, F2, F3, CR, CR1, CR2, Kk, Kmin, Kmax, Precision;
12 FILE    *in, *pop;
13 time_t t1;//=time(NULL);
14 time_t t2;
15 char    MutScheme [10];
16 double Stagnation[3];
17 double maxNew, maxOld, conNew, conOld, sumNew, sumOld;
18 int     nPop, change;
19 int     g, sum1, con1;
20 double sumd1;
21 void OpenDEpar() {
22     if ((in = fopen("DEpar.txt", "rt")) == NULL) {
23         printf("Cannot open configuration file !!!\n");
24         exit(0);
25     }
26     const int LLENGTH = 256;
27     char Line[LLENGTH];
28     fgets(Line, LLENGTH, in); fscanf(in, "%d\n", &NPi);
29     fgets(Line, LLENGTH, in); fscanf(in, "%d\n", &Generations);
30     fgets(Line, LLENGTH, in); fscanf(in, "%lf\n", &F1);
31     fscanf(in, "%lf\n", &F2); //fgetc(in);
32     fscanf(in, "%lf\n", &F3);
33     fscanf(in, "%lf\n", &CR1);
34     fscanf(in, "%lf\n", &CR2);
35     fscanf(in, "%lf\n", &Kmin);
36     fscanf(in, "%lf\n", &Kmax);
37     fgets(Line, LLENGTH, in);
38    fscanf(in, "%d\n", &Scheme);
39     fgets(Line, LLENGTH, in);
40    fscanf(in, "%lf\n", &Precision);
41     fclose(in);
42 /*
43     printf("Npi = %d\n", NPi);
44     printf("Generations = %d\n", Generations);
45     printf("F1 = %lf\n", F1);
46     printf("F2 = %lf\n", F2);
47     printf("F3 = %lf\n", F3);
48     printf("CR1 = %lf\n", CR1);
49     printf("CR2 = %lf\n", CR2);
50     printf("Kmin = %lf\n", Kmin);
51     printf("Kmax = %lf\n", Kmax);
52     printf("Scheme = %d\n", Scheme);
53     printf("Precision = %lf\n", Precision);
54 */
55 }
56 void determineDV() {
57     for(g=0;g<iKT;g++) {
58         if(ArcKT[g][0]==ArcKT[g+1][0]) {
59         nPop++;
60       }
61    }
62    return;
63 }
64 int createSpecimen(int pd) {
65     PD = pd+1;
66     Specimen = (double*) calloc(pd * 3, sizeof(double));
67     if (Specimen==NULL) {
68         puts("Not enough memory !!!");
69         exit(0);
70     }
71     return Specimen ? 0 : 1;
72 }
73 void setSpecimen(int dm, int pd, double pop) {
74     Specimen[3 * pd + dm] = pop;
75 }
76 double maximum(double vict1, double vict2) {
77     if (vict1 >= vict2) {
78         return vict1;
79     } else {
80         return vict2;
81     }
82 }
```

```
83  void PrepareSpecimen() {
84      a4=0;
85      for(g=0;g<iKT;g++) {
86      if (ArcKT[g][0]==ArcKT[g+1][0]) {
87          setSpecimen(0,a4,0);
88           setSpecimen(1,a4,1);
89           setSpecimen(2,a4,0);
90           a4++;
91        }
92     }
93     D=a4;
94     D=maximum(D,4);
95      NP=NPi*D;
96     PNP=NP+3;
97     return;
98  }
99  int createHistory(int ml) {
100     History = (double*) calloc(ml, sizeof(double));
101     if (History==NULL) {
102         puts("Not enough memory !!!");
103         exit(0);
104     }
105     return Population ? 0 : 1;
106 }
107 int createPopulation() {
108     Population = (double*) calloc((NP+3) * PD, sizeof(double));
109     if (Population==NULL) {
110         puts("Not enough memory !!!");
111         exit(0);
112    }
113    return Population ? 0 : 1;
114 }
115 int createTMPPopulation() {
116     TMPPopulation = (double*) calloc((NP+3) * PD, sizeof(double));
117     if (TMPPopulation==NULL) {
118         puts("Not enough memory !!!");
119         exit(0);
120     }
121     return TMPPopulation ? 0 : 1;
122 }
123 void SetScheme() {
124     if(Scheme==1) {
125         strcpy(MutScheme,"DER1B");
126    } else if(Scheme==2) {
127         strcpy(MutScheme,"DER2B");
128     } else {
129         strcpy(MutScheme,"DECuRR");
130     }
131 }
132 void setPopulation(int pd, int pnp, double pop) {
133     Population[PD * pnp + pd] = pop;
134 }
135 void InitializePopulation() {
136     for (a1=0;a1<PD;a1++) {
137         for (a2=0;a2<NP+3;a2++) {
138             setPopulation(a1, a2, 0);
139       }
140    }
141 }
142 double getPopulation(int pd, int pnp) {
143    return Population[PD * pnp + pd];
144 }
145 double getSpecimen(int dm, int pd) {
146    return Specimen[3 * pd + dm];
147 }
148 void SetProb(int Candidate, int Individual) {
149 //  char ofilename[] = "Probs.txt";
150 //  FILE *ofp;
151 //  ofp = fopen(ofilename,"at");
152     int origin, dest;
153     a4=0;
154     sumd1=0.0;
155 //  printf("SetProb:\n");
156     for(g=0;g<iKT;g++) {
157         origin=ArcKT[g][0];
158         dest=ArcKT[g][1];
159         if (ArcKT[g][0]==ArcKT[g+1][0]) {
160             prob[Individual][origin-1][dest-1]=getPopulation(a4,Candidate);
161             sumd1+=prob[Individual][origin-1][dest-1];
162 //              printf("pop[%d][%d]=%f\t", a4, Candidate, getPopulation(a4,Candidate));
163             a4++;
164         } else {
165             if (prob[Individual][origin-1][dest-1]!=1) {
```

```
166                     prob[Individual][origin-1][dest-1]=1.0-sumd1;
167                     sumd1=0.0;
168                 }
169         }
170     }
171 /*
172     printf("pop[%d][%d]=%f\n", a4, Candidate, getPopulation(a4,Candidate));
173     for(g=0;g<iKT;g++) {
174         origin=ArcKT[g][0];
175         dest=ArcKT[g][1];
176         printf("prob[%d][%d][%d]=%f\t",
177         Individual, origin, dest, prob[Individual][origin-1][dest-1]);
178     }
179     printf("\n");
180 */
181    //    fclose(ofp);
182 }
183 void setTMPPopulation(int pd, int pnp, double pop) {
184    TMPPopulation[PD * pnp + pd] = pop;
185 }
186 double getTMPPopulation(int pd, int pnp) {
187    return TMPPopulation[PD * pnp + pd];
188 }
189 void WorstIndividual() {
190     int smc;
191    smc=0;
192     bb=smc;
193     hh=getPopulation(PD-1,smc);;
194     for (smc=1;smc<PNP-3;smc++) {
195         if (getPopulation(PD-1,smc)>hh) {
196             bb=smc;
197             hh=getPopulation(PD-1,smc);
198         }
199     }
200 }
201 void BestIndividual() {
202     int smc;
203     h=1E+38;
204     for (smc=0;smc<PNP-3;smc++) {
205         if ((getPopulation(PD-1,smc)<h)&&(getPopulation(PD-1,smc)>0.)) {
206             b=smc;
207             h=getPopulation(PD-1,smc);
208         }
209     }
210 }
211 void WritePOP(char *ret) {
212     if ((pop = fopen("Population.txt","w")) == NULL) {
213         printf("Cannot open output population file !!!\n");
214         goto END;
215     }
216     printf("Calculated by %s\n",ret);
217     printf("\n");
218     for (a1=0;a1<PNP-3;a1++) {
219         sum1=0;
220         for (a2=0;a2<PD;a2++) {
221             fprintf(pop, "%lf\t", Population[PD * a1 + a2]);
222             sum1++;
223             if (sum1==5) {
224                 sum1=0;
225                 fprintf(pop, "\n");
226          }
227       }
228         fprintf(pop, "\n");
229     }
230     fclose(pop);
231     END:;
232 }
233 void WriteOut(char *ret) {
234     //on screen
235     system("cls");
236     //return the best individual
237     printf("Calculated by %s\n",ret);
238     printf("\n");
239     printf("Evolution conditions\nNP = %d,\tD = %d,\tF = %lg,\tCR = %lg\n",
240             NP, D, F, CR);
241     printf("Generations = %d\n", ldef+1);
242     printf("Number of function evaluations = %d\n", neval);
243     printf("Number of mutations = %d\n", nmut);
244     printf("Required time = %d seconds\n", (int)(t2-t1));
245     printf("\n");
246     //profit value of the best individual
247     for(g=0;g<iKT;g++) {
248         i=ArcKT[g][0];
```

```
249           s=ArcKT[g][1];
250           printf("Best individual: Prob[%d,%d] = %lf\n", i, s, prob[0][i-1][s-1]);
251       }
252        printf("\n");
253        ckl1=getPopulation(PD-1,b);
254        printf("Best individual Expected service time = %lf\n", ckl1);
255        printf("Best Member:\n");
256        for (a2=0;a2<PD;a2++) {
257           printf("%lf\t", Population[PD * b + a2]);
258       }
259        printf("\n\n");
260        ckl1=getPopulation(PD-1,bb);
261        printf("Worst individual Expected service time = %lf\n", ckl1);
262        printf("Worst Member:\n");
263        for (a2=0;a2<PD;a2++) {
264           printf("%lf\t", Population[PD * bb + a2]);
265        }
266        printf("\n\n");
267        //to file
268       if ((in = fopen("DE-Output.txt", "at")) == NULL) {
269           printf("Cannot open output file !!!\n");
270           goto END;
271       }
272       //return the best individual
273        fprintf(in, "Calculated by %s\n",ret);
274        fprintf(in, "\n");
275        fprintf(in, "Evolution conditions:\nNP = %d,\tD = %d,\tF = %lg,\tCR = %lg\n",
276                    NP, D, F, CR);
277        fprintf(in, "Generations = %d\n", ldef+1);
278        fprintf(in, "Number of function evaluations = %d\n", neval);
279        fprintf(in, "Number of mutations = %d\n", nmut);
280        fprintf(in, "Required time = %d seconds\n", (int)(t2-t1));
281        fprintf(in, "\n");
282       //cost value of the best individual
283        for(g=0;g<iKT;g++) {
284           i=ArcKT[g][0];
285           s=ArcKT[g][1];
286           fprintf(in, "Best individual: Prob[%d,%d] = %lf\n", i, s, prob[0][i-1][s-1]);
287       }
288        fprintf(in, "\n\n");
289        ckl1=getPopulation(PD-1,b);
290        fprintf(in, "Best individual Expected service time = %lf\n", ckl1);
291        fprintf(in, "Best Member:\n");
292        for (a2=0;a2<PD;a2++) {
293           fprintf(in, "%lf\t", Population[PD * b + a2]);
294       }
295        fprintf(in, "\n\n");
296        ckl1=getPopulation(PD-1,bb);
297        fprintf(in, "Worst individual Expected service time = %lf\n", ckl1);
298        fprintf(in, "Worst Member:\n");
299        for (a2=0;a2<PD;a2++) {
300           fprintf(in, "%lf\t", Population[PD * bb + a2]);
301        }
302        fprintf(in, "\n\n");
303        fclose(in);
304        END:;
305   }
306   void WriteHistory() {
307        if ((in = fopen("History.txt", "wt")) == NULL) {
308        printf("Cannot open output file for history of evolution !!!\n");
309          goto END;
310       }
311       for (i=0;i<=ldef;i++) {
312        fprintf(in,"%d\t%lf\n", i+1, History[i]);
313       }
314       fclose(in);
315       END:;
316   }
317   int destroyTMPPopulation() {
318        free(TMPPopulation);
319        TMPPopulation = 0;
320        return 0;
321   }
322   int destroyPopulation() {
323        free(Population);
324        Population = 0;
325        return 0;
326   }
327   int destroySpecimen() {
328        free(Specimen);
329        Specimen = 0;
330        return 0;
331   }
```

```
332 int destroyHistory() {
333     free(History);
334     History = 0;
335     return 0;
336 }
```

Código A.3: DESchemes.c

```
1  void DER1B() {
2  //   printf("DER1B:\n");
3      F=F1;
4     CR=CR1;
5      //three random vectors
6     do {
7      V1=(PNP-3)*((double)rand()/RAND_MAX );
8        V2=(PNP-3)*((double)rand()/RAND_MAX );
9        V3=(PNP-3)*((double)rand()/RAND_MAX );
10    } while (V1==V2 || V1==V3 || V2==V3 || V1==a1 || V2==a1 || V3==a1);
11 //   printf("(V1,V2,V3)=(%d,%d,%d)\n", V1, V2, V3);
12     //from difference vector to trial vector
13 //   printf("trial vector: ");
14     for (a2=0;a2<PD-1;a2++) {
15         sum1=D*((double)rand()/RAND_MAX );
16       if (((double)rand()/RAND_MAX )<CR || a2==sum1) { //PD-1)
17             setPopulation(a2, PNP-3,
18             F*(getPopulation(a2,V1)-getPopulation(a2,V2))+getPopulation(a2,V3));
19         } else {
20             setPopulation(a2, PNP-3, getPopulation(a2,a1));
21       }
22 //        printf("(%f) ", getPopulation(a2, PNP-3));
23          //type of variable and boundary checking
24          //completely new random value
25  /*
26        if (getSpecimen(0,a2)>getPopulation(a2,PNP-3) || getSpecimen(1,a2)<getPopulation(
              a2,PNP-3))
27             setPopulation(a2,PNP-3,getSpecimen(0,a2)+((double) rand()/RAND_MAX )*(
                  getSpecimen(1,a2)-getSpecimen(0,a2)));
28  */
29         //new random value between current and exceeded boundary
30         if (getSpecimen(0,a2)>getPopulation(a2,PNP-3)) {
31             setPopulation(a2, PNP-3,
32             getSpecimen(0,a2)+((double)rand()/RAND_MAX )*(getPopulation(a2,a1)-
                  getSpecimen(0,a2)));
33         }
34         if (getSpecimen(1,a2)<getPopulation(a2,PNP-3)) {
35             setPopulation(a2,PNP-3,
36             getPopulation(a2,a1)+((double)rand()/RAND_MAX )*(getSpecimen(1,a2)-
                  getPopulation(a2,a1)));
37         }
38 //        printf("%f\t", getPopulation(a2, PNP-3));
39     }
40 //   printf("\n");
41 }
42 void DER2B() {
43     F=F2;
44     CR=CR2;
45     //three random vectors
46     do {
47         V1=(PNP-3)*((double) rand()/RAND_MAX );
48       V2=(PNP-3)*((double) rand()/RAND_MAX );
49       V3=(PNP-3)*((double) rand()/RAND_MAX );
50       V4=(PNP-3)*((double) rand()/RAND_MAX );
51       V5=(PNP-3)*((double) rand()/RAND_MAX );
52    } while (V1==V2 || V1==V3 || V1==V4 || V1==V5 || V2==V3 || V2==V4 || V2==V5 || V3==V4
          || V3==V5 || V4==V5 || V1==a1 || V2==a1 || V3==a1 || V4==a1 || V5==a1);
53     //from difference vector to trial vector
54     sum1=D*((double) rand()/RAND_MAX );
55     for(a2=0;a2<PD-1;a2++) {
56         if (((double) rand()/RAND_MAX )<CR || a2==sum1) { //PD-1)
57             setPopulation(a2, PNP-3,
58             F*(getPopulation(a2,V1)+getPopulation(a2,V2)-getPopulation(a2,V3)-
                  getPopulation(a2,V4))+getPopulation(a2,V5));
59         } else {
60             setPopulation(a2,PNP-3,getPopulation(a2,a1));
61       }
62         //type of variable and boundary checking
63         //completely new random value
64 /*
65         if (getSpecimen(0,a2)>getPopulation(a2,PNP-3) || getSpecimen(1,a2)<getPopulation
              (a2,PNP-3))
66             setPopulation(a2,PNP-3,getSpecimen(0,a2)+((double) rand()/RAND_MAX )*(
                  getSpecimen(1,a2)-getSpecimen(0,a2)));
67 */
68         //new random value between current and exceeded boundary
69         if(getSpecimen(0,a2)>getPopulation(a2,PNP-3)) {
70             setPopulation(a2,PNP-3,getSpecimen(0,a2)+((double) rand()/RAND_MAX )*(
                  getPopulation(a2,a1)-getSpecimen(0,a2)));
71         }
72         if(getSpecimen(1,a2)<getPopulation(a2,PNP-3)) {
```

```
73              setPopulation(a2,PNP-3,getPopulation(a2,a1)+((double) rand()/RAND_MAX )*(
                    getSpecimen(1,a2)-getPopulation(a2,a1)));
74          }
75      }
76  }
77  void DECuRR() {
78      F=F3;
79      //three random vectors
80      do {
81          V1=(PNP-3)*((double) rand()/RAND_MAX );
82          V2=(PNP-3)*((double) rand()/RAND_MAX );
83          V3=(PNP-3)*((double) rand()/RAND_MAX );
84     } while (V1==V2 || V1==V3 || V2==V3 || V1==a1 || V2==a1 || V3==a1);
85      Kk=(Kmin+(Kmax-Kmin)*((double) rand()/RAND_MAX ));
86      //from difference vector to trial vector
87      for(a2=0;a2<PD-1;a2++) {
88          setPopulation(a2, PNP-3,
89          getPopulation(a2,a1)+Kk*(getPopulation(a2,V3)-getPopulation(a2,a1))+F*(
                getPopulation(a2,V2)-getPopulation(a2,V1)));
90          //type of variable and boundary checking
91          //completely new random value
92  /*
93          if(getSpecimen(0,a2)>getPopulation(a2,PNP-3) || getSpecimen(1,a2)<getPopulation(
                a2,PNP-3))
94              setPopulation(a2,PNP-3,getSpecimen(0,a2)+((double) rand()/RAND_MAX )*(
                    getSpecimen(1,a2)-getSpecimen(0,a2)));
95  */
96          //new random value between current and exceeded boundary
97          if (getSpecimen(0,a2)>getPopulation(a2,PNP-3)) {
98              setPopulation(a2, PNP-3,
99              getSpecimen(0,a2)+((double)rand()/RAND_MAX )*(getPopulation(a2,a1)-
                    getSpecimen(0,a2)));
100         }
101         if (getSpecimen(1,a2)<getPopulation(a2,PNP-3)) {
102             setPopulation(a2, PNP-3,
103             getPopulation(a2,a1)+((double)rand()/RAND_MAX )*(getSpecimen(1,a2)-
                    getPopulation(a2,a1)));
104         }
105     }
106 }
```

Código A.4: cmusr.cpp

```
1  //
2  // Purpose:
3  //     to implement congestions models
4  //     FOR A SPECIFIC CLASS OF USERS
5  //
6  // Version:
7  //     6.00
8  //
9  // Date:
10 //     Nov/2010
11 //
12 #ifndef CMUSR_CPP
13 #define CMUSR_CPP
14 #include <stdlib.h>
15 #include <stdio.h>
16 #include <math.h>
17 #include "cm.cpp"
18 //
19 // WARNING
20 //
21 // these variables are set up for each service type XX:
22 //
23 double CMUSR_maxDens;
24 double CMUSR_maxSpeed;
25 double CMUSR_aDens;
26 double CMUSR_bDens;
27 double CMUSR_aSpeed;
28 double CMUSR_bSpeed;
29 //
30 // in the file below:
31 //
32 char CMUSR_servFileName[] = "mgcc-serv-XX.txt";
33 //
34 // function SetCorridor(double length, double width);
35 // must be called for each server to set up these variables:
36 //
37 // double length; // server length
38 // double width;  // server width
39 //
40 // generic velocity congestion model for users
41 //
42 class CMGenUsr {
43 public:
44   // default constructor
45   CMGenUsr(void) {}
46   // destructor
47   virtual ¬CMGenUsr(void) {}
48   virtual void SetCorridor(double length, double width) = 0;
49 };
50 //
51 // linear velocity congestion model for users
52 //
53 class CMLinUsr: public CMLin, public CMGenUsr {
54 public:
55   // default constructor
56   CMLinUsr(void): CMLin(), CMGenUsr() {}
57   // destructor
58   ¬CMLinUsr(void) {}
59   void SetCorridor(double length, double width);
60 };
61 //
62 // exponential velocity congestion model for users
63 //
64 class CMExpUsr: public CMExp, public CMGenUsr {
65 public:
66   // default constructor
67   CMExpUsr(void): CMExp(), CMGenUsr() {}
68   // destructor
69   ¬CMExpUsr(void) {}
70   void SetCorridor(double length, double width);
71 };
72 //
73 // implementation
74 //
75 void CMLinUsr::SetCorridor(double length, double width) {
76 #if CMUSR_DEBUG
77   fprintf(CM_OUT_FILE,"CMLinUsr::SetCorridor(double,double):\n");
78 #endif
79   int theCap = (int)floor(CMUSR_maxDens * length * width);
80   double theEts1 = length/CMUSR_maxSpeed;
81   CMGen::SetC(theCap);
82   SetEts1(theEts1);
```

```
83     SetV1(CMUSR_maxSpeed);
84   }
85   void CMExpUsr::SetCorridor(double length, double width) {
86   #if CMUSR_DEBUG
87     fprintf(CM_OUT_FILE,"CMExpUsr::SetCorridor(double,double):\n");
88   #endif
89     CMExpUsr::SetShapeForm(CMUSR_maxDens, CMUSR_aDens, CMUSR_aSpeed,
90                            CMUSR_bDens, CMUSR_bSpeed);
91     int theCap = (int)floor(CMUSR_maxDens * length * width);
92     double theEts1 = length/CMUSR_maxSpeed;
93     SetC(theCap);
94     SetEts1(theEts1);
95     SetV1(CMUSR_maxSpeed);
96   }
97   #endif
```

Código A.5: cm.cpp

```
//
// Purpose:
//    to implement linear and exponential congestion models
//
// Version: 5.0
//
// Date: Nov/2010
//
#ifndef CM_CPP
#define CM_CPP
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
//
// these are general settings
//
#define CM_IN_FILE stdin    // input file
#define CM_OUT_FILE stdout  // output file
#define CM_ERR_FILE stderr  // error file
#define CM_EVALUATED 1      // flag
//
// WARNING
//
// these variables must be eventually set up for each server:
//
// int     cap;            // server capacity
// double  expecST;        // expected service time for lone occupant
// double  maxSpeed;       // lone occupant speed
//
// double  maxDens;        // maximum density per unit of area
// double  aDens;          // density A
// double  aSpeed;         // speed at density A
// double  bDens;          // density B
// double  bSpeed;         // speed at density B
//
// generic congestion model
//
class CMGen {
protected:
  int cap;          // server capacity
  double Ets1;      // expected service time for lone occupant
  double maxSpeed;  // lone occupant speed
  int status;
public:
  // default constructor
  CMGen(void): cap(0), Ets1(0), maxSpeed(0), status(!CM_EVALUATED) {}
  // destructor
  virtual ¬CMGen(void) {}
  void SetC(int C)              {cap=C; status=!CM_EVALUATED;}
  void SetEts1(double theEts1)  {Ets1=theEts1; status=!CM_EVALUATED;}
  void SetV1(double V1)         {maxSpeed=V1; status=!CM_EVALUATED;}
  int GetC(void)                {return cap;}
  double GetEts1(void)          {return Ets1;}
  double GetV1(void)            {return maxSpeed;}
  virtual double Rate(int customers) = 0;
};
//
// linear velocity congestion model
//
class CMLin: public CMGen {
public:
  // default constructor
  CMLin(void): CMGen() {}
  // destructor
  ¬CMLin(void) {}
  double Rate(int customers);
};
//
// exponential velocity congestion model
//
class CMExp: public CMGen {
  int    statConsts; // status of constants
  double maxDens;    // maximum density
  double aDens;      // density A
  double aSpeed;     // speed at density A
  double bDens;      // density B
  double bSpeed;     // speed at density B
  double beta;       // shape and form parameters
  double gamma;      // shape and form parameters
public:
  // default constructor
  CMExp(void): CMGen(), statConsts(!CM_EVALUATED) {}
```

```
83      // destructor
84      ¬CMExp(void) {}
85      void SetShapeForm(double maxDens, double aDens, double aSpeed,
86                        double bDens, double bSpeed);
87      double Rate(int customers);
88    };
89    //
90    // implementation
91    //
92    double CMLin::Rate(int customers) {
93      if ((cap≤0)||(customers<0)||(customers>cap)) {
94        fprintf(CM_ERR_FILE, "CMLin::Rate(int): ERROR: parameter out of range\n");
95        exit(1);
96      }
97    #if CM_DEBUG
98      fprintf(CM_OUT_FILE, "CMLin::Rate(int):\t%20.18f\n",
99              ((double)(cap+1−customers)/cap));
100   #endif
101     return((double)(cap+1−customers)/cap);
102   }
103   void CMExp::SetShapeForm(double theMaxDens, double theADens, double theASpeed,
104                            double theBDens, double theBSpeed) {
105   #if CM_DEBUG
106      fprintf(CM_OUT_FILE,"CMExp::SetShapeForm:\n");
107   #endif
108      maxDens = theMaxDens;
109      aDens = theADens;
110      aSpeed = theASpeed;
111      bDens = theBDens;
112      bSpeed = theBSpeed;
113      status = !CM_EVALUATED;
114   }
115   double CMExp::Rate(int customers) {
116     if (status != CM_EVALUATED) {
117           double a=aDens*cap/maxDens;
118         double b=bDens*cap/maxDens;
119         gamma=log( log(aSpeed/maxSpeed)/log(bSpeed/maxSpeed)) / log((a−1)/(b−1));
120   //        if (errno) {
121   //            fprintf(CM_OUT_FILE, "CMExp::Rate(%d):\tgamma\tlog(log(%f/%f)/log(%f/%f))/log
          ((%f−1)/(%f−1)\t%f\n",
122   //                  customers, aSpeed, maxSpeed, bSpeed, maxSpeed, a, b, gamma);
123   //            exit(1);
124   //        }
125         beta=(a−1)/pow(log(maxSpeed/aSpeed),(1/gamma));
126   //        if (errno) {
127   //            fprintf(CM_OUT_FILE, "CMExp::Rate(%d):\tbeta\t(%f−1)/pow(log(%f/%f),(1/%f))\t
          %f\n",
128   //                  customers, a, maxSpeed, aSpeed, gamma, beta);
129   //            exit(1);
130   //        }
131         status = CM_EVALUATED;
132   #if CM_DEBUG
133         fprintf(CM_OUT_FILE, "CMExp::Rate(%d):\n\tgamma\t%20.18f\tbeta\t%20.18f\n",
134                 customers, gamma, beta);
135         fprintf(CM_OUT_FILE, "CMExp::Rate(%d):\texp(−((%d−1)/%f)^%f\t%f\n",
136                 customers, customers, beta, gamma, exp(−pow((customers−1)/beta,gamma)));
137   #endif
138     }
139     if ((cap≤0)||(customers<0)||(customers>cap)) {
140       fprintf(CM_ERR_FILE, "CMExp::Rate(%d):\tERROR: parameter out of range\n",
141               customers);
142       exit(1);
143     }
144     return(exp(−pow((customers−1)/beta,gamma)));
145   }
146   #endif
```

Código A.6: mgccso.cpp

```
//
// Given:
//   - velocity congestion model and
//   - input lambda,
// this library determines performance measures:
//   - balking probability,
//   - throughput rate,
//   - expected number of occupants in the queue, and
//   - waiting time.
//
// Version: 3.00
//
// Date: Nov/2010
//
#ifndef MGCCANL_CPP
#define MGCCANL_CPP
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cmusr.cpp"
// #define or #undef below for either improved or traditional implementation
#define MGCCANL_IMPROVED
//#undef MGCCANL_IMPROVED
//
// these are general settings
//
#define MGCCANL_IN_FILE stdin    // input file
#define MGCCANL_OUT_FILE stdout // output file
#define MGCCANL_ERR_FILE stderr // error file
#define MGCCANL_NOT_EVALUATED 0 // flag
#define MGCCANL_PRE_EVALUATED 1 // flag
#define MGCCANL_EVALUATED 2      // flag
#define MGCCANL_EPSILON 1E-06    // precision (iterative process)
#define MGCCANL_MAX_IT 50        // max. # iterations (iterative proc.)
#define MGCCANL_INFINITY 1E+38   // infinity
//
// MGCC Class
//
class MgccNet;
class Mgcc {
  friend class MgccNet;
// input
  CMGen *service;  // velocity congestion model
  double lambda0i; // external input
  double lambda_i; // total input
  double eLambda;  // effective lambda
  int status;      // node status (not-evaluated, pre-evaluated or evaluated)
// output
  double pC;       // blocking probability
  double theta;    // throughput
  double Eq;       // expected number of customers
  double Ets;      // expected service time
  double mu_h;     // service rate at previous holding node
  double rho;      // node utilization of holding node
public:
// default constructor
  Mgcc(void);
// destructor
  ¬Mgcc(void);
// methods
  void   Reset(void);
  void   SetService(CMGen *service);
  void   SetExtLambda(double lambda);
  double GetLambda(void);
  int    GetC(void);
  double GetEts1(void);
  void   GetPerfMeasures(double theEts1);
  void   GetPerfMeasuresMMcc(double theEts1);
  double GetPC(void)     {return pC;}
  double GetTheta(void) {return theta;}
  double GetWIP(void)    {return Eq;}
  double GetEts(void)    {return Ets;}
  void   ShowInput(void);
  void   ShowPerfMeasures(void);
  void   ShowAll(void);
  void   AggregatePerfMeasures(void);
};
//
// mgcc node type
//
typedef struct mgccNode {
  int index;
```

```
83     double prob;
84     mgccNode *next;
85   } mgccNodeType;
86   //
87   // network of MGCC Class
88   //
89   class MgccNet: public Mgcc {
90     int          nOfNodes;
91     Mgcc         *node;
92     mgccNodeType **prevNode;
93   public:
94     // default constructor
95     MgccNet(void);
96     // destructor
97     ¬MgccNet(void);
98     int  ReadData(FILE *inputFile);
99     void Reset(void);
100    void SetNOfNodes(int nNodes);
101    void SetArc(int origin, int dest, float theProb, int Individual);
102    int  IsArc(int origin, int dest,int Individual) {return(prob[Individual][origin][dest
           ]>MGCCANL_EPSILON);}
103    void SetService(int nIndex, CMGen *serv);
104    void SetExtLambda(int nIndex, double lambda);
105    void GetPerfMeasures(int Individual);
106    void GetPerfMeasuresMMcc(void);
107    void PreEvaluate(int nIndex,int Individual);
108    void Evaluate(int nIndex,int Individual);
109    void ShowInput(int nIndex);
110    void ShowPerfMeasures(void);
111    void ShowPerfMeasures(int nIndex);
112    void ShowNet(void);
113    void AggregatePerfMeasures(void);
114    double GetPC(int nIndex)     {return node[nIndex].GetPC();}
115    double GetTheta(int nIndex) {return node[nIndex].GetTheta();}
116    double GetWIP(int nIndex)    {return node[nIndex].GetWIP();}
117    double GetEts(int nIndex)    {return node[nIndex].GetEts();}
118  };
119  //
120  // implementation
121  //
122  Mgcc::Mgcc(void): service(NULL), lambda0i(0.0), lambda_i(0.0),
123                    eLambda(MGCCANL_INFINITY), status(MGCCANL_NOT_EVALUATED) {
124  #if MGCCSO_DEBUG
125    fprintf(MGCCANL_OUT_FILE, "Mgcc::Mgcc()\n" );
126  #endif
127  };
128  Mgcc::¬Mgcc(void) {
129  #if MGCCSO_DEBUG
130    fprintf(MGCCANL_OUT_FILE, "Mgcc::¬Mgcc()\n" );
131  #endif
132  };
133  void Mgcc::Reset(void) {
134  #if MGCCSO_DEBUG
135    fprintf(MGCCANL_OUT_FILE, "Mgcc::Reset\n" );
136  #endif
137    eLambda = MGCCANL_INFINITY;
138    status = MGCCANL_NOT_EVALUATED;
139  }
140  void Mgcc::SetService(CMGen *theService) {
141    service = theService;
142    Reset();
143  }
144  int Mgcc::GetC(void) {
145    if (service==NULL) {
146       fprintf(MGCCANL_OUT_FILE, "Mgcc::GetC: error service not set");
147       exit(1);
148    }
149    return (service->GetC());
150  }
151  double Mgcc::GetEts1(void) {
152    if (service==NULL) {
153       fprintf(MGCCANL_OUT_FILE, "Mgcc::GetEts: error service not set");
154       exit(1);
155    }
156    return (service->GetEts1());
157  }
158  void Mgcc::SetExtLambda(double theLambda) {
159    lambda0i = theLambda;
160    lambda_i = lambda0i;
161    Reset();
162  }
163  double Mgcc::GetLambda(void) {
164    return (lambda0i);
```

```
165 }
166 void Mgcc::GetPerfMeasures(double Ets1) {
167 #if MGCCSO_DEBUG
168   fprintf(MGCCANL_OUT_FILE, "Mgcc::GetPerfMeasures(Ets1=%f):\n", Ets1);
169 #endif
170   double *pn, maxPn, ΔPn, pCPrime, pCPrimeOld;
171   double lambda, lambda_h, z, r1, r2, xn, xd;
172   double Emu_i;
173   int C, Cmax, k, it;
174 //
175 //  compute blocking probability
176 //
177 //    initializations
178   C = service->GetC();
179   pn = new double[C+1];
180 //    calculate empty system probability p0 for node i
181   pn[1] = log(exp(-80)+lambda_i*Ets1/(1*service->Rate(1)));
182 //  pn[0] = 1.0 + exp(pn[1]);
183 //  fprintf(MGCCANL_OUT_FILE, "pn[1]\t%20.18f\n",pn[1]);
184   for (k=2; k≤C; k++) {
185      pn[k] = pn[k-1] + log(exp(-80)+lambda_i*Ets1/(k*service->Rate(k)));
186 //      pn[0] += exp(pn[k]);
187 //      fprintf(MGCCANL_OUT_FILE, "pn[%d]\t%20.18f\n",k,pn[k]);
188   }
189 //  pn[0] = 1/pn[0];
190 //  fprintf(MGCCANL_OUT_FILE, "pn[%d]\t%e\n",0,pn[0]);
191   maxPn = -1;
192   for (k=1; k≤C; k++) {
193      if (pn[k]>maxPn) {
194         maxPn=pn[k];
195      }
196   }
197   if (maxPn>80) {
198      ΔPn = (maxPn-80);
199 //      fprintf(MGCCANL_OUT_FILE, "(%e-80)\t%e\n",maxPn,ΔPn);
200      for (k=1; k≤C; k++) {
201         pn[k]-=ΔPn;
202      }
203   }
204   pn[0] = 1.0;
205   for (k=1; k≤C; k++) {
206      pn[0] += exp(pn[k]);
207   }
208   pn[0] = 1/pn[0];
209 //  fprintf(MGCCANL_OUT_FILE, "pn[%d]'\t%e\n",0,pn[0]);
210 //    calculate non-zero state probabilities
211   pn[0] = log(pn[0]);
212   for (k=1; k≤C; k++) {
213      pn[k] += pn[0];
214      pn[k] = exp(pn[k]);
215 //      fprintf(MGCCANL_OUT_FILE, "pn[%d]\t%20.18f\n",k,pn[k]);
216   }
217   pn[0] = exp(pn[0]);
218 //    compute blocking probability
219   pC = pn[C];
220 //
221 //  compute theta
222 //
223 //    method one
224 //  theta = pn[1]*service->Rate(1);
225 //  for (k=2; k≤C; k++) {
226 //      theta += k*pn[k]*service->Rate(k);
227 //  }
228 //  theta /= Ets1;
229 //  fprintf(MGCCANL_OUT_FILE, "\ttheta\t%20.18f\n", theta);
230 //
231 //    method two
232   theta = lambda_i*(1-pC);
233 //
234 //  compute expected number of customers
235 //
236   Eq = pn[1];
237   for (k=2; k≤C; k++) {
238      Eq += k*pn[k];
239   }
240 //
241 //  compute waiting time
242 //
243 //    method one
244 //  Ets = pn[1]/service->Rate(1);
245 //  for (k=2; k≤C; k++) {
246 //      Ets += pn[k]/service->Rate(k);
247 //  }
```

```
248 //   Ets *= Ets1;
249 //   fprintf(MGCCANL_OUT_FILE, "\tE(ts)\t%20.18f\n", Ets);
250 //
251 //      method two
252    if (theta>MGCCANL_EPSILON) {
253          Ets = Eq/theta;
254    } else {
255          Ets = 0.0;
256    }
257 //
258 //   compute lambda_h
259 //
260    lambda_h = lambda_i*pC;
261 //   fprintf(MGCCANL_OUT_FILE, "\tlambda_h\t%20.18f\n", lambda_h);
262 //
263 //   compute service rate at holding node
264 //
265 //      approximate expected service rate at holding node
266 //
267 //            method one
268 //   Emu_i = pn[1]*service->Rate(1);
269 //   for (k=2; k<=C; k++) {
270 //       Emu_i += k*pn[k]*service->Rate(k);
271 //   }
272 //   Emu_i /= Ets1;
273 //   double VARTs_i = 1/(Emu_i*Emu_i);
274 //   mu_h = 2*Emu_i/(1+Emu_i*Emu_i*VARTs_i);
275 //   fprintf(MGCCANL_OUT_FILE, "\tEmu\t%20.18f\n", Emu_i);
276 //   fprintf(MGCCANL_OUT_FILE, "\tmu_h\t%20.18f\n", mu_h);
277 //
278 //            method two
279    Emu_i = C/(Ets1/service->Rate(C));
280    mu_h = Emu_i;
281 //
282 //      determine holding node utilization
283 //
284    rho = lambda_i/mu_h;
285 //
286 //      compute pCPrime
287 //
288    Cmax=C;
289    if (Cmax>50) Cmax=50;
290    pCPrime = 0.5;
291    it = 0;
292 //   fprintf(MGCCANL_OUT_FILE, "it\t");
293 //   fprintf(MGCCANL_OUT_FILE, "lambda  \t");
294 //   fprintf(MGCCANL_OUT_FILE, "z       \t");
295 //   fprintf(MGCCANL_OUT_FILE, "r1      \t");
296 //   fprintf(MGCCANL_OUT_FILE, "r2      \t");
297 //   fprintf(MGCCANL_OUT_FILE, "xn      \t");
298 //   fprintf(MGCCANL_OUT_FILE, "xd      \t");
299 //   fprintf(MGCCANL_OUT_FILE, "p(C)'\n");
300 ///*
301    do {
302       pCPrimeOld=pCPrime;
303 //        update lambda
304       lambda = theta - lambda_h*(1-pCPrime);
305 //        update pjCjPrime
306       z = pow((lambda + Emu_i + mu_h),2)-(4*lambda*mu_h);
307       r1 = ((lambda + Emu_i + mu_h)-sqrt(z))/(2*mu_h);
308       r2 = ((lambda + Emu_i + mu_h)+sqrt(z))/(2*mu_h);
309       xn = (pow(r2,Cmax)-pow(r1,Cmax))-(pow(r2,Cmax-1)-pow(r1,Cmax-1));
310       xd = (pow(r2,Cmax+1)-pow(r1,Cmax+1))-(pow(r2,Cmax)-pow(r1,Cmax));
311       pCPrime=1/((Emu_i+mu_h)/mu_h-(lambda*xn)/(mu_h*xd));
312       it++;
313 //       fprintf(MGCCANL_OUT_FILE, "%d\t",it);
314 //       fprintf(MGCCANL_OUT_FILE, "%f\t", lambda);
315 //       fprintf(MGCCANL_OUT_FILE, "%f\t",z);
316 //       fprintf(MGCCANL_OUT_FILE, "%f\t", r1);
317 //       fprintf(MGCCANL_OUT_FILE, "%f\t", r2);
318 //       fprintf(MGCCANL_OUT_FILE, "%f\t", xn);
319 //       fprintf(MGCCANL_OUT_FILE, "%f\t", xd);
320 //       fprintf(MGCCANL_OUT_FILE, "%f\n", pCPrime);
321    } while ((fabs(pCPrime-pCPrimeOld)>MGCCANL_EPSILON)&&(it<MGCCANL_MAX_IT));
322 //*/
323 //   fprintf(MGCCANL_OUT_FILE, "\tp(C)'\t%f\n", pCPrime);
324 //      update mu_h
325    mu_h = mu_h*(1-pCPrime);
326 //   fprintf(MGCCANL_OUT_FILE, "\tmu_h'\t%20.18f\n", mu_h);
327 //      print final results
328 //   fprintf(MGCCANL_OUT_FILE, "\tp(C)\t%20.18f\n", pC);
329 //   fprintf(MGCCANL_OUT_FILE, "\ttheta\t%20.18f\n", theta);
330 //   fprintf(MGCCANL_OUT_FILE, "\tE(q)\t%20.18f\n", Eq);
```

```
331 //   fprintf(MGCCANL_OUT_FILE, "\tE(ts)\t%20.18f\n", Ets);
332   status = MGCCANL_PRE_EVALUATED;
333 //   free variables
334   delete[] pn;
335 }
336 void Mgcc::GetPerfMeasuresMMcc(double Ets1) {
337 #if MGCCSO_DEBUG
338   fprintf(MGCCANL_OUT_FILE, "Mgcc::GetPerfMeasuresMMcc(Ets1=%f):\n", Ets1);
339 #endif
340   double *pn, maxPn, ΔPn;
341   double gamma, beta, muk, V1, Va, Vb, l, w, a, b;
342   int C, k;
343 //
344 //   compute blocking probability
345 //
346 //      initializations
347   C = service->GetC();
348   pn = new double[C+1];
349 //      calculate empty system probability p0 for node i
350   V1=1.5; l=Ets1*V1; w=C/(5*l); a=2*l*w; Va=0.64; b=4*l*w; Vb=0.25;
351   gamma=log(log(Va/V1)/log(Vb/V1))/log((a-1)/(b-1));
352   beta=(a-1)/pow(log(V1/Va),(1/gamma));
353   k=1;
354   muk = k*(V1/l)*(exp(-pow((k-1)/beta,gamma)));
355 //   fprintf(MGCCANL_OUT_FILE, "gamma\t%f\tbeta\t%f\tmui\t%f\n",gamma, beta, mui);
356   pn[1] = log(lambda_i/muk);
357 //   fprintf(MGCCANL_OUT_FILE, "pn[1]\t%20.18f\n",pn[1]);
358   for (k=2; k≤C; k++) {
359       muk = k*(V1/l)*(exp(-pow((k-1)/beta,gamma)));
360       pn[k] = pn[k-1] + log(lambda_i/muk);
361 //       fprintf(MGCCANL_OUT_FILE, "pn[%d]\t%20.18f\n",k,pn[k]);
362   }
363   maxPn = -1;
364   for (k=1; k≤C; k++) {
365       if (pn[k]>maxPn) {
366           maxPn=pn[k];
367       }
368   }
369   if (maxPn>80) {
370       ΔPn = (maxPn-80);
371 //       fprintf(MGCCANL_OUT_FILE, "(%e-80)\t%e\n",maxPn,ΔPn);
372       for (k=1; k≤C; k++) {
373           pn[k]-=ΔPn;
374       }
375   }
376   pn[0] = 1.0;
377   for (k=1; k≤C; k++) {
378       pn[0] += exp(pn[k]);
379   }
380   pn[0] = 1/pn[0];
381   pn[0] = log(pn[0]);
382   for (k=1; k≤C; k++) {
383       pn[k] += pn[0];
384       pn[k] = exp(pn[k]);
385   }
386   pn[0] = exp(pn[0]);
387 //      compute blocking probability
388   pC = pn[C];
389 //   compute theta
390   theta = lambda_i*(1-pC);
391 //   compute expected number of customers
392   Eq = pn[1];
393   for (k=2; k≤C; k++) {
394       Eq += k*pn[k];
395   }
396 //   compute waiting time
397   Ets = Eq/theta;
398   status = MGCCANL_PRE_EVALUATED;
399 //   free variables
400   delete[] pn;
401 }
402 void Mgcc::ShowInput(void) {
403 #if MGCCSO_DEBUG
404   fprintf(MGCCANL_OUT_FILE,"Mgcc::ShowInput():\n" );
405   fprintf(MGCCANL_OUT_FILE,"\tExpected service time for lone occupant\t%f\n",
406           service->GetEts());
407   fprintf(MGCCANL_OUT_FILE,"\tCapacity\t%d\n", service->GetC() );
408   fprintf(MGCCANL_OUT_FILE,"\tLambda\t%f\n", lambda0i );
409 #endif
410   fprintf(MGCCANL_OUT_FILE,"\tE(ts)\tC\tLambda\n");
411   fprintf(MGCCANL_OUT_FILE,"\t%.3f\t%d\t%.3f\n",
412           service->GetEts1(), service->GetC(), lambda0i);
413 }
```

```
414  void Mgcc::ShowPerfMeasures(void) {
415  #if MGCCSO_DEBUG
416    fprintf(MGCCANL_OUT_FILE,"Mgcc::ShowPerfMeasur:\n" );
417    fprintf(MGCCANL_OUT_FILE,"\tBlocking probability\t%f\n", pC );
418    fprintf(MGCCANL_OUT_FILE,"\tThroughput\t%f\n", theta );
419    fprintf(MGCCANL_OUT_FILE,"\tExpected number of customers\t%f\n", Eq );
420    fprintf(MGCCANL_OUT_FILE,"\tExpected service time\t%f\n", Ets );
421    fprintf(MGCCANL_OUT_FILE,"\tHolding node utilization\t%f\n", rho );
422    fprintf(MGCCANL_OUT_FILE,"\trho\tPn\tcustomers\ttheta\tEts\n");
423    fprintf(MGCCANL_OUT_FILE,"%f\t%f\t%f\t%f\t%f\n",
424            rho, pC, Eq, theta, Ets);
425    fprintf(MGCCANL_OUT_FILE,"p(C)\ttheta\tE(q)\tE(ts)\trho\n");
426    fprintf(MGCCANL_OUT_FILE,"%f\t%f\t%f\t%f\t%f\n",pC,theta,Eq,Ets,rho);
427  #endif
428    fprintf(MGCCANL_OUT_FILE,"\tp(C)\t%20.18f\n", pC);
429    fprintf(MGCCANL_OUT_FILE,"\ttheta\t%20.18f\n", theta);
430    fprintf(MGCCANL_OUT_FILE,"\tE(q)\t%20.18f\n", Eq);
431    fprintf(MGCCANL_OUT_FILE,"\tE(ts)\t%20.18f\n", Ets);
432  //  fprintf(MGCCANL_OUT_FILE,"\trho\t%f\n", rho);
433  //  fprintf(MGCCANL_OUT_FILE,"%f\t%f\t%f\t%f\t%f\t",pC,theta,Eq,Ets,rho);
434  }
435  void Mgcc::ShowAll(void) {
436  #if MGCCSO_DEBUG
437    fprintf(MGCCANL_OUT_FILE,
438    "E(ts)\tcap\tlambda\tp(C)\ttheta\tE(q)\tE(ts)\trho\n");
439  #endif
440    fprintf(MGCCANL_OUT_FILE,"%f %8d %f ", service->GetEts1(), service->GetC(), lambda0i);
441    fprintf(MGCCANL_OUT_FILE,"%f %f %f %f %f\n",pC, theta, Eq, Ets, rho);
442  }
443  MgccNet::MgccNet(void): nOfNodes(0), node(NULL), prevNode(NULL) {
444  #if MGCCSO_DEBUG
445     fprintf(MGCCANL_OUT_FILE, "MgccNet:Constructor()\n" );
446  #endif
447  }
448  MgccNet::¬MgccNet(void) {
449  #if MGCCSO_DEBUG
450    fprintf(MGCCANL_OUT_FILE, "MgccNet::¬MgccNet():\n" );
451    fprintf(MGCCANL_OUT_FILE, "\tFreeing memory\n" );
452  #endif
453    mgccNodeType *auxNode;
454    int i;
455    delete[] node;
456    for (i=0; i < nOfNodes; i++) {
457      while ( prevNode[i] != NULL ) {
458        auxNode = prevNode[i];
459        prevNode[i] = prevNode[i]->next;
460        delete[] auxNode;
461      }
462    }
463    delete[] prevNode;
464  };
465  int MgccNet::ReadData(FILE *inputFile) {
466  #if MGCCSO_DEBUG
467    fprintf(MGCCANL_OUT_FILE, "MgccNet::ReadData(FILE)\n" );
468  #endif
469    const int LLENGTH = 256;
470    char Line[LLENGTH];
471    int nNodes, idx, orig, dest;
472    float prob;
473    int i, serv;
474    float length, width, theLambda;
475    char servFileName[LLENGTH];
476    FILE *servFile;
477    CMLinUsr *lServ;
478    CMExpUsr *eServ;
479    /* read and set number of nodes */
480    fgets(Line, LLENGTH, inputFile);
481    fscanf(inputFile, "%d\n", &nNodes );
482  #if MGCCSO_DEBUG
483    fprintf(MGCCANL_OUT_FILE, "%d\n", nNodes);
484  #endif
485    SetNOfNodes(nNodes);
486    /* read arc related data */
487    fgets(Line, LLENGTH, inputFile);
488    while (fscanf(inputFile, "%d %d %d %f\n",
489                  &idx, &orig, &dest, &prob)==4) {
490  #if MGCCSO_DEBUG
491      fprintf(MGCCANL_OUT_FILE, "%d %d %d %f\n",
492                  idx, orig, dest, prob);
493  #endif
494      SetArc(orig, dest, prob, 0);
495      SetArc(orig, dest, prob, 1);
496      if (prob≤1.) {
```

```
497             ArcKT[iKT][0]=orig;
498           ArcKT[iKT][1]=dest;
499           iKT++;
500         }
501       }
502       /* read node number, service, length, width, and lambda */
503       fgets(Line, LLENGTH, inputFile);
504       for (i=0; i < nNodes; i++) {
505         fscanf(inputFile,
506            "%d %d %f %f %f\n", &idx,&serv,&length,&width,&theLambda);
507   #if MGCCSO_DEBUG
508         fprintf(MGCCANL_OUT_FILE,
509            "%d %d %f %f %f\n",idx,serv,length,width,theLambda);
510   #endif
511         //  choose service
512         //  get file name
513         int aux=0;
514         while ((servFileName[aux]=CMUSR_servFileName[aux])!='\0') aux++;
515         servFileName[10] = '0'+(serv/10)%10;
516         servFileName[11] = '0'+serv%10;
517         servFile = fopen(servFileName,"r");
518         if (servFile == NULL) {
519                 fprintf(MGCCANL_ERR_FILE, "Error: service %d unknown\n", serv);
520                 exit(1);
521         }
522         //  get information
523   #if MGCCSO_DEBUG
524         fprintf(MGCCANL_OUT_FILE, "%s: current service file\n", servFileName);
525   #endif
526         fgets(Line, LLENGTH, servFile);
527         fscanf(servFile, "%lf\n", &CMUSR_maxDens);
528         fgets(Line, LLENGTH, servFile);
529         fscanf(servFile, "%lf\n", &CMUSR_maxSpeed);
530         if (fgets(Line, LLENGTH, servFile)==0) {
531   //            linear service time
532             lServ = new CMLinUsr;
533             lServ->SetCorridor(length,width);
534             SetService(idx-1, lServ);
535   #if MGCCSO_DEBUG
536         fprintf(MGCCANL_OUT_FILE, "Linear queue: (%lf)\n", CMUSR_maxSpeed);
537   #endif
538         } else {
539   //            exponential service time
540                 fscanf(servFile, "%lf\n", &CMUSR_aDens);
541                 fgets(Line, LLENGTH, servFile);
542                 fscanf(servFile, "%lf\n", &CMUSR_bDens);
543                 fgets(Line, LLENGTH, servFile);
544                 fscanf(servFile, "%lf\n", &CMUSR_aSpeed);
545                 fgets(Line, LLENGTH, servFile);
546                 fscanf(servFile, "%lf\n", &CMUSR_bSpeed);
547             eServ = new CMExpUsr;
548             eServ->SetCorridor(length,width);
549             SetService(idx-1, eServ);
550   #if MGCCSO_DEBUG
551         fprintf(MGCCANL_OUT_FILE, "Exponential type %d queue: (%lf;%lf;%lf)\n",
552                 serv, CMUSR_maxSpeed, CMUSR_aSpeed, CMUSR_bSpeed);
553   #endif
554         }
555         fclose(servFile);
556         SetExtLambda(idx-1,theLambda);
557   #if MGCCSO_DEBUG
558         ShowInput(idx-1);
559         fprintf(MGCCANL_OUT_FILE, "\n");
560   #endif
561       }
562       return 0;
563   }
564   void MgccNet::Reset(void) {
565   #if MGCCSO_DEBUG
566     fprintf(MGCCANL_OUT_FILE, "MgccNet::Reset():\n" );
567   #endif
568     int i;
569     for (i=0; i<nOfNodes; i++) {
570       node[i].Reset();
571     }
572   }
573   void MgccNet::SetNOfNodes(int nNodes) {
574     int i, j,k;
575     /* get number of nodes and allocate them */
576     nOfNodes = nNodes;
577     /* create new nodes and mark all of them as not evaluated */
578     node = new Mgcc[nOfNodes];
579     prevNode = new mgccNodeType*[nOfNodes];
```

```
580     for (i=0; i<nOfNodes; i++) {
581       prevNode[i] = NULL;
582     }
583     /*for (i=0; i<nOfNodes; i++) {
584       prob[i] = new float[nOfNodes];
585       for (j=0; j<nOfNodes; j++) {
586           prob[i][j] = 0.0;
587       }
588     }*/
589     prob = new float**[2];
590     for (i=0; i<2; i++)
591     {
592         prob[i] = new float*[nOfNodes];
593         for (j=0; j<nOfNodes; j++)
594         {
595             prob[i][j] = new float[nOfNodes];
596             for (k=0; k<nOfNodes; k++)
597             {
598                 prob[i][j][k] = 0.0;
599             }
600         }
601     }
602     Reset();
603   }
604   void MgccNet::SetArc(int origin, int dest, float theProb, int Individual) {
605   #if MGCCSO_DEBUG
606     fprintf(MGCCANL_OUT_FILE, "MgccNet::SetArc(int,int,float,int):\n");
607   #endif
608     mgccNodeType *auxNode;
609     auxNode = prevNode[dest-1];
610     while ((auxNode!=NULL )&&(auxNode->index!=(origin-1))) {
611           auxNode = auxNode->next;
612     }
613     if (auxNode!=NULL) {
614           auxNode->prob = theProb;
615     } else {
616           auxNode = new mgccNodeType;
617         auxNode->index = origin-1;
618         auxNode->prob = theProb;
619         auxNode->next = prevNode[dest-1];
620         prevNode[dest-1] = auxNode;
621     }
622     prob[Individual][origin-1][dest-1] = theProb;
623     Reset();
624   }
625   void MgccNet::SetExtLambda(int nIndex, double theLambda) {
626     node[nIndex].SetExtLambda(theLambda);
627     Reset();
628   }
629   void MgccNet::SetService(int nIndex, CMGen *serv) {
630     node[nIndex].SetService(serv);
631     Reset();
632   }
633   void MgccNet::GetPerfMeasuresMMcc(void) {
634   #if MGCCSO_DEBUG
635     fprintf(MGCCANL_OUT_FILE, "MgccNet::GetPerfMeasuresMMcc(void):\n");
636   #endif
637     node[0].GetPerfMeasuresMMcc(node[0].service->GetEts1());
638   }
639   #ifndef MGCCANL_IMPROVED
640   void MgccNet::GetPerfMeasures(int Individual) {
641   #if MGCCSO_DEBUG
642       fprintf(MGCCANL_OUT_FILE, "MgccNet::GetPerfMeasures(%d):\n",Individual+1);
643   #endif
644       int i;
645       Reset();
646   //
647   //  pre-evaluate everybody
648       for (i=0; i<nOfNodes; i++) {
649           if (node[i].status != MGCCANL_PRE_EVALUATED) {
650               PreEvaluate(i,Individual);
651         }
652      }
653   //
654   //  evaluate everybody
655       for (i=0; i<nOfNodes; i++) {
656       if (node[i].status != MGCCANL_EVALUATED) {
657           Evaluate(i,Individual);
658         }
659      }
660   }
661   void MgccNet::PreEvaluate(int nIndex,int Individual) {
662     int i;
```

```
663     node[nIndex].lambda_i = node[nIndex].lambda0i;
664     for (i=0; i<nOfNodes; i++ ) {
665         if (IsArc(i,nIndex,Individual)) {
666             if (node[i].status != MGCCANL_PRE_EVALUATED) {
667                 PreEvaluate(i,Individual);
668             }
669             node[nIndex].lambda_i += prob[Individual][i][nIndex]*node[i].theta;
670         }
671     }
672 #if MGCCSO_DEBUG
673     fprintf(MGCCANL_OUT_FILE, "MgccNet::PreEvaluate(%d,%d):\n", nIndex+1,Individual+1);
674 #endif
675     node[nIndex].GetPerfMeasures(node[nIndex].service->GetEts1());
676     node[nIndex].status=MGCCANL_PRE_EVALUATED;
677 //   node[nIndex].ShowPerfMeasures();
678 }
679 void MgccNet::Evaluate(int nIndex,int Individual) {
680     double Ets1, Ets1Inf, Ets1Sup, unassignedFlow;
681     int i, j, numPred, predLeft;
682     for (j=0; j<nOfNodes; j++ ) {
683         if (IsArc(nIndex,j,Individual)) {
684             if (node[j].status != MGCCANL_EVALUATED) {
685                 Evaluate(j,Individual);
686             }
687         }
688     }
689 #if MGCCSO_DEBUG
690     fprintf(MGCCANL_OUT_FILE, "MgccNet::Evaluate(%d,%d):\n", nIndex+1,Individual+1);
691 #endif
692 //   fprintf(MGCCANL_OUT_FILE, "\teLambda\t%20.18f\n", node[nIndex].eLambda);
693     if ((node[nIndex].theta-node[nIndex].eLambda) > MGCCANL_EPSILON) {
694 //
695 //        if solution is infeasible then adjust theta
696 //
697 //        Bisection Method
698 //
699         Ets1Inf=node[nIndex].GetEts1();
700         Ets1Sup=Ets1Inf;
701         do {
702             Ets1Sup *= 2;
703             node[nIndex].GetPerfMeasures(Ets1Sup);
704 //              fprintf(MGCCANL_OUT_FILE, "theta(Ets1Sup=%f)\t%f\n",
705 //                          Ets1Sup, node[nIndex].theta);
706         } while ((node[nIndex].theta-node[nIndex].eLambda) > MGCCANL_EPSILON);
707         Ets1Inf=Ets1Sup/2;
708 //          fprintf(MGCCANL_OUT_FILE, "[%f ; %f]\n", Ets1Inf, Ets1Sup);
709         do {
710             Ets1=(Ets1Inf+Ets1Sup)/2;
711             node[nIndex].GetPerfMeasures(Ets1);
712             if ((node[nIndex].theta-node[nIndex].eLambda) > MGCCANL_EPSILON) {
713               Ets1Inf=Ets1;
714             } else {
715               Ets1Sup=Ets1;
716             }
717 //              fprintf(MGCCANL_OUT_FILE, "[%f ; %f]\t", Ets1Inf, Ets1Sup);
718 //              fprintf(MGCCANL_OUT_FILE, "theta(%f)\t%f\n",
719 //                          Ets1, node[nIndex].theta);
720         } while (fabs((node[nIndex].theta-node[nIndex].eLambda))>MGCCANL_EPSILON);
721 //          fprintf(MGCCANL_OUT_FILE, "E(ts1)\t%20.18f\n", Ets1);
722     }
723     node[nIndex].status = MGCCANL_EVALUATED;
724 //   node[nIndex].ShowPerfMeasures();
725 //
726 //      propagate backwards information about new theta
727 //
728     numPred = 0;
729     for (i=0; i<nOfNodes; i++ ) {
730         if (prob[Individual][i][nIndex] > MGCCANL_EPSILON) {
731             numPred++;
732         }
733     }
734     if (numPred > 0) {
735         unassignedFlow=node[nIndex].theta;
736         while (unassignedFlow > MGCCANL_EPSILON) {
737               predLeft = numPred;
738                   for (i=0; i<nOfNodes; i++ ) {
739                   if (prob[Individual][i][nIndex] > MGCCANL_EPSILON) {
740                         if (node[i].eLambda == MGCCANL_INFINITY) {
741                         if (unassignedFlow/predLeft > node[i].theta) {
742                               node[i].eLambda = node[i].theta;
743                                unassignedFlow -= node[i].theta;
744                            } else {
745                               node[i].eLambda = unassignedFlow/predLeft;
```

```
746                         unassignedFlow -= unassignedFlow/predLeft;
747                      }
748                  } else {
749                   if ((node[i].eLambda+unassignedFlow/predLeft) > node[i].theta) {
750                         unassignedFlow -= (node[i].theta-node[i].eLambda);
751                       node[i].eLambda = node[i].theta;
752                      } else {
753                       node[i].eLambda += unassignedFlow/predLeft;
754                         unassignedFlow -= unassignedFlow/predLeft;
755                      }
756                  } // if (node[i].eLambda == MGCCANL_INFINITY) {
757                  predLeft--;
758             } // end if (prob[Individual][i][nIndex] > MGCCANL_EPSILON) {
759          } // end for
760       } // while (unassignedFlow > MGCCANL_EPSILON) {
761    } // if (numPred > 0) {
762 }
763 #endif MGCCANL_IMPROVED
764 #ifdef MGCCANL_IMPROVED
765 void MgccNet::GetPerfMeasures(int Individual) {
766 #if MGCCSO_DEBUG
767     fprintf(MGCCANL_OUT_FILE, "MgccNet::GetPerfMeasures(%d):\n",Individual);
768 #endif
769     int uneval, i, j, predec, succ;
770     Reset();
771 //
772 //  pre-evaluate everybody
773     uneval=nOfNodes;
774     while (uneval>0) {
775         // for all nodes
776         for (j=0; j<nOfNodes; j++ ) {
777             if (node[j].status!=MGCCANL_PRE_EVALUATED) {
778                 predec=MGCCANL_PRE_EVALUATED;
779                 i=0;
780                 // confirm all predecessors were pre-evaluated
781                 while ((i<nOfNodes)&&(predec==MGCCANL_PRE_EVALUATED)) {
782                     if (IsArc(i,j,Individual))
783                         if (node[i].status!=MGCCANL_PRE_EVALUATED)
784                             predec=MGCCANL_NOT_EVALUATED;
785                         i++;
786             }
787             if (predec==MGCCANL_PRE_EVALUATED) {
788                 PreEvaluate(j,Individual);
789                node[j].status=MGCCANL_PRE_EVALUATED;
790                uneval--;
791             }
792          }
793       }
794    }
795 //
796 //  evaluate everybody
797     uneval=nOfNodes;
798     while (uneval>0) {
799         // for all nodes
800       for (i=0; i<nOfNodes; i++ ) {
801         if (node[i].status!=MGCCANL_EVALUATED) {
802             succ=MGCCANL_EVALUATED;
803             j=0;
804             // confirm all sucessors were evaluated
805             while ((j<nOfNodes)&&(succ==MGCCANL_EVALUATED)) {
806                 if (IsArc(i,j,Individual))
807                 if (node[j].status!=MGCCANL_EVALUATED)
808                     succ=MGCCANL_NOT_EVALUATED;
809                j++;
810             }
811             if (succ==MGCCANL_EVALUATED) {
812                 Evaluate(i,Individual);
813                node[i].status = MGCCANL_EVALUATED;
814                uneval--;
815             }
816          }
817       }
818    }
819 #if MGCCSO_DEBUG
820   int g, origin, dest;
821   for(g=0;g<iKT;g++) {
822         origin=ArcKT[g][0];
823         dest=ArcKT[g][1];
824         printf("prob[%d][%d][%d]=%f\t", Individual, origin, dest,
825             prob[Individual][origin-1][dest-1]);
826   }
827   printf("\n");
828 #endif
```

```
829 }
830 void MgccNet::PreEvaluate(int nIndex,int Individual) {
831 #if MGCCSO_DEBUG
832     fprintf(MGCCANL_OUT_FILE, "MgccNet::PreEvaluate(%d,%d):\n", nIndex+1,Individual+1);
833 #endif
834     int i;
835     node[nIndex].lambda_i = node[nIndex].lambda0i;
836     for (i=0; i<nOfNodes; i++ )
837         if (IsArc(i,nIndex,Individual))
838             node[nIndex].lambda_i += prob[Individual][i][nIndex]*node[i].theta;
839     node[nIndex].GetPerfMeasures(node[nIndex].service->GetEts1());
840 //  node[nIndex].ShowPerfMeasures();
841 }
842 void MgccNet::Evaluate(int nIndex,int Individual) {
843 #if MGCCSO_DEBUG
844     fprintf(MGCCANL_OUT_FILE, "MgccNet::Evaluate(%d,%d):\n", nIndex+1,Individual+1);
845 #endif
846     double Ets1, Ets1Inf, Ets1Sup, unassignedFlow;
847     int i, j, *idxPred, numPred, smallIdx, smallTheta, idxAux, unassigPred;
848 //
849 //  if solution is infeasible then adjust theta
850     if ((node[nIndex].theta-node[nIndex].eLambda)>MGCCANL_EPSILON) {
851 //      Bisection Method
852         Ets1Inf=node[nIndex].GetEts1();
853       Ets1Sup=Ets1Inf;
854       do {
855             Ets1Sup *= 2;
856             node[nIndex].GetPerfMeasures(Ets1Sup);
857 //          fprintf(MGCCANL_OUT_FILE, "theta(Ets1Sup=%f)\t%f\n",
858 //                      Ets1Sup, node[nIndex].theta);
859         } while((node[nIndex].theta-node[nIndex].eLambda)>MGCCANL_EPSILON);
860         Ets1Inf=Ets1Sup/2;
861 //      fprintf(MGCCANL_OUT_FILE, "[%f ; %f]\n", Ets1Inf, Ets1Sup);
862         do {
863             Ets1=(Ets1Inf+Ets1Sup)/2;
864             node[nIndex].GetPerfMeasures(Ets1);
865             if ((node[nIndex].theta-node[nIndex].eLambda) > MGCCANL_EPSILON) {
866                 Ets1Inf=Ets1;
867             } else {
868                 Ets1Sup=Ets1;
869             }
870 //          fprintf(MGCCANL_OUT_FILE, "[%f ; %f]\t", Ets1Inf, Ets1Sup);
871 //          fprintf(MGCCANL_OUT_FILE, "theta(%f)\t%f\n",
872 //                      Ets1, node[nIndex].theta);
873         } while (fabs((node[nIndex].theta-node[nIndex].eLambda))>MGCCANL_EPSILON);
874 //      fprintf(MGCCANL_OUT_FILE, "E(ts1)\t%20.18f\n", Ets1);
875     } // end if
876 //  node[nIndex].ShowPerfMeasures();
877 //
878 //  backpropagate information about new theta to all predecessors
879    unassignedFlow=node[nIndex].theta;
880    //   find number of predecessors
881    idxPred=new int[nOfNodes];
882     numPred=0;
883     for (i=0; i<nOfNodes; i++)
884         if (IsArc(i,nIndex,Individual)) {
885         idxPred[numPred]=i;
886             numPred++;
887       }
888 //  fprintf(MGCCANL_OUT_FILE, "Predecessors\t");
889 //  for (i=0; i<numPred; i++) fprintf(MGCCANL_OUT_FILE, "%d ", idxPred[i]+1);
890 //  fprintf(MGCCANL_OUT_FILE, "\n");
891    //   creat list of predecessors sorted by throughput
892    for (i=0; i<numPred; i++) {
893     //  find smaller throughput
894         smallTheta=node[idxPred[i]].theta;
895         smallIdx=i;
896       for (j=i+1; j<numPred; j++) {
897         if (smallTheta>node[idxPred[j]].theta) {
898                 smallTheta=node[idxPred[j]].theta;
899                 smallIdx=j;
900          }
901       }
902       // exchange elements
903         idxAux=idxPred[i];
904         idxPred[i]=idxPred[smallIdx];
905         idxPred[smallIdx]=idxAux;
906    }
907 //  fprintf(MGCCANL_OUT_FILE, "Predecessors sorted\t");
908 //  for (i=0; i<numPred; i++) fprintf(MGCCANL_OUT_FILE, "%d ", idxPred[i]+1);
909 //  fprintf(MGCCANL_OUT_FILE, "\n");
910    // update eLambdas
911    unassigPred=numPred;
```

```
912      for (i=0; i<numPred; i++) {
913       j=idxPred[i];
914            if (node[j].eLambda==MGCCANL_INFINITY) {
915            if (node[j].theta*prob[Individual][j][nIndex]<unassignedFlow/unassigPred) {
916                    node[j].eLambda = node[j].theta;
917                    unassignedFlow -= node[j].eLambda*prob[Individual][j][nIndex];
918             } else {
919                node[j].eLambda = unassignedFlow/unassigPred/prob[Individual][j][nIndex];
920                    unassignedFlow -= node[j].eLambda*prob[Individual][j][nIndex];
921             }
922         } else if (node[j].eLambda*prob[Individual][j][nIndex]>unassignedFlow/unassigPred)
              {
923            node[j].eLambda -= (node[j].eLambda*prob[Individual][j][nIndex]-
924                                            unassignedFlow/unassigPred);
925                unassignedFlow -= (node[j].eLambda*prob[Individual][j][nIndex]-
926                                            unassignedFlow/unassigPred);
927         } else {
928                unassignedFlow -= node[j].eLambda*prob[Individual][j][nIndex];
929           } // if (node[i].eLambda == MGCCANL_INFINITY) {
930         unassigPred--;
931 //         fprintf(MGCCANL_OUT_FILE, "Assigned to node %d\t%20.18f\n",
932 //                     j+1, node[j].eLambda);
933      } // end for (i=0; i<numPred; i++) {
934 }
935 #endif MGCCANL_IMPROVED
936 void MgccNet::ShowInput(int nIndex) {
937 #if MGCCSO_DEBUG
938    fprintf(MGCCANL_OUT_FILE, "MgccNet::ShowInput(%d):\n", nIndex+1);
939 #endif
940    fprintf(MGCCANL_OUT_FILE, "Node\t%d\n", nIndex+1 );
941    node[nIndex].ShowInput();
942 }
943 void MgccNet::ShowPerfMeasures(void) {
944 #if MGCCSO_DEBUG
945    fprintf(MGCCANL_OUT_FILE, "MgccNet::ShowPerfMeasures():\n");
946 #endif
947    for (int i=0; i<nOfNodes; i++) {
948        fprintf(MGCCANL_OUT_FILE, "Node\t%d\n", i+1 );
949        node[i].ShowPerfMeasures();
950    }
951 }
952 void MgccNet::ShowPerfMeasures(int nIndex) {
953 #if MGCCSO_DEBUG
954    fprintf(MGCCANL_OUT_FILE, "MgccNet::ShowPerfMeasures(%d):\n", nIndex+1);
955 #endif
956    fprintf(MGCCANL_OUT_FILE, "Node\t%d\n", nIndex+1 );
957    node[nIndex].ShowPerfMeasures();
958 }
959 void MgccNet::ShowNet(void) {
960    fprintf(MGCCANL_OUT_FILE, "MgccNet::ShowNet():\n");
961    int i, aux;
962    mgccNodeType *auxNode;
963    fprintf(MGCCANL_OUT_FILE, "Nodes\n%d\n", nOfNodes );
964    fprintf(MGCCANL_OUT_FILE, "Arc\tOrig\tDest\tProb\n");
965    aux = 0;
966    for (i=0; i < nOfNodes; i++) {
967      auxNode = prevNode[i];
968      while (auxNode != NULL ) {
969        fprintf(MGCCANL_OUT_FILE, "%d\t%d\t%d\t%.3f\n",
970               ++aux, auxNode->index+1, i+1, auxNode->prob);
971        auxNode = auxNode->next;
972      }
973    }
974    for (i=0; i < nOfNodes; i++) {
975      fprintf(MGCCANL_OUT_FILE, "Node\t%d\n", i+1);
976      node[i].ShowInput();
977    }
978 }
979 void Mgcc::AggregatePerfMeasures(void) {
980 #if MGCCSO_DEBUG
981    fprintf(MGCCANL_OUT_FILE, "Mgcc::AggregatePerfMeasures():\n");
982 #endif
983    sumd1+=(theta*Ets);
984 };
985 void MgccNet::AggregatePerfMeasures() {
986 #if MGCCSO_DEBUG
987    fprintf(MGCCANL_OUT_FILE, "MgccNet::AggregatePerfMeasures():\n");
988 #endif
989    sumd1=0;
990    for (i=0; i<nOfNodes; i++) {
991          node[i].AggregatePerfMeasures();
992    }
993 #if MGCCSO_DEBUG
```

```
994     fprintf(MGCCANL_OUT_FILE, "sumd1=%f\n",sumd1);
995  #endif
996  }
997  #endif
```

Apêndice B

Arquivo de Definição dos Parâmetros do DE

Arquivo de Definição dos Parâmetros do DE B.1: DEpar.txt

```
NPi => determines the population size, good values or 2,10 or 20.
2
Generations => how long the DE algorithm can run at most
1000
F1, F2, F3, CR1, CR2, Kmin, Kmax => DE parameters
1
0.6
0.4
0.8
0.99
0.99
0
Scheme => DE scheme (values 1,2 or 3)
1
Precision => the lower this value, the longer the algorithm will search and
    the higher the quality of the solution
0.0000001
```

Apêndice C

Arquivo de Definição dos Serviços

Arquivo de Definição dos Serviços C.1: mgcc-serv-06.txt

```
1  CMUSR_maxDens
2  200.0
3  CMUSR_maxSpeed
4  25.0
5  CMUSR_aDens
6  20.0
7  CMUSR_bDens
8  140.0
9  CMUSR_aSpeed
10 23.0
11 CMUSR_bSpeed
12 10.0
```

Arquivo de Definição dos Serviços C.2: mgcc-serv-07.txt

```
1  CMUSR_maxDens
2  200.0
3  CMUSR_maxSpeed
4  20.0
5  CMUSR_aDens
6  20.0
7  CMUSR_bDens
8  140.0
9  CMUSR_aSpeed
10 18.0
11 CMUSR_bSpeed
12 6.0
```

Apêndice D

Arquivo de Entrada

Arquivo de entrada D.1: 3node-L0500.txt

```
1  Nodes
2   3
3  Arc Origin Destination Probability
4  1    1    2    0.5
5  2    1    3    0.5
6  Node Service Length Width Lambda
7  1    6    0.80      5.0 500.0
8  2    7    2.50      2.0 0.000
9  3    7    1.85      2.0 0.000
10 Exit Nodes
11 1
12 2
13 3
```