Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Estatística

Paula de Campos Oliveira

Função Estocástica de Tempo de Percurso

Belo Horizonte
2010

Paula de Campos Oliveira

Função Estocástica de Tempo de Percurso

Texto para Defesa de Tese apresentado ao Programa de Pós-Graduação do Departamento de Estatística do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do título de Doutor em Estatística.

Área de Concentração: Estatística e Probabilidade

Orientador: Prof. Frederico R. B. Cruz
Co-orientador: Prof. Luiz H. Duczmal

Belo Horizonte
2010

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Estatística
Programa de Pós-Graduação
Caixa Postal 702
31270-901 Belo Horizonte- MG – Brasil

Telefone (31) 3409-5923
Fax (31) 3409-5924
E-mail: pgest@ufmg.br
WEB: http://www.est.ufmg.br/posgrad/

FOLHA DE APROVAÇÃO

FUNÇÃO ESTOCÁSTICA DE TEMPO DE PERCURSO

Paula de Campos Oliveira

Tese defendida e aprovada pela banca examinadora constituída pelos Professores:

.Prof. Frederico R. B. Cruz
(Orientador/EST/UFMG);

.Prof. Luiz Henrique Duczmal
(Co-Orientador/EST/UFMG);

.Prof. Roberto da Costa Quinino
(EST/UFMG);

.Prof. Oriane Magela Neto
(DEE /UFMG);

.Prof. Anderson Ribeiro Duarte
(DEMAT/UFOP);

.Prof. Michel Ferreira da Silva
(EST/UnB).

Belo Horizonte, 27 de agosto de 2010.

Agradecimentos

Chegar neste momento não foi fácil e muito menos simples. Para chegar até aqui necessitei de apoio e auxílio de muitas pessoas, e por isso é que preciso agradecer. Segundo os dicionários da língua portuguesa, agradecer é reconhecer apoio, demonstrar o seu apreço por este e, especialmente quando as pessoas não têm obrigação alguma de lhe ajudarem, deixar claro que você sabe dar valor e considerar o que representa a pessoa ou pessoas que fazem isso por você. É muito difícil escolher as palavras certas para agradecer, mas de toda forma vou tentar demonstrar minha gratidão.

Agradeço a Deus pela oportunidade de mais uma conquista, por sempre estar presente abençoando e me guiando nesta difícil jornada.

É muito pouco somente agradecer ao meu orientador, Frederico. Serei eternamente grata por tudo o que ele é e representa para mim, afinal, além de ser um exemplo de profissionalismo e sabedoria, sempre se mostrar paciente, compreensível e disponível para me ajudar, foi meu anjo da guarda que sempre me impulsionou não me deixando desistir. Sou grata também ao Prof. Luiz H. Duczmal, por ter aceitado o desafio de co-orientar e me auxiliar com seus conhecimentos.

À Coordenadora, Glaura, e ao ex-Coordenador, Enrico, obrigada por terem me ajudado e orientado nos momentos de maiores indecisões nesses anos de curso. Gostaria de lembrar aqui e demonstrar minha gratidão, à minha companheira fiel nos estudos de Probabilidade e Inferência Avançada e amiga nos momentos difíceis, Thais, e também ao Max pela ajuda nos dando aulinhas extras de Probabilidade Avançada.

Não poderia esquecer de agradecer, aos colegas e superiores da FACISABH, pelo apoio e incentivo durante a realização deste trabalho.

Aos meus pais, obrigada por existirem, pela minha vida, pelo exemplo de dedicação e perseverança, pela compreensão, carinho, amor e pelo incentivo para concluir mais uma etapa. À minha irmã, Valéria, sou grata por sempre me incentivar a seguir em frente.

Ao meu namorado e amigo Anderson, desejo agradecer pelo apoio, companheirismo e, sobretudo, pelo amor incondicional que me reconforta e me dá forças para superar obstáculos.

Enfim, agradeço a todos que de alguma forma me apoiaram, incentivaram e ajudaram na conclusão deste trabalho.

Resumo

Aplicamos redes de filas finitas dependentes do estado para modelar o tempo de percurso em sistemas de comunicação móvel. Embora tenham sido utilizadas com sucesso no passado na modelagem de tráfego de veículos, não conhecemos aplicação dos modelos dependentes do estado aos sistemas de comunicação móveis. A novidade dos modelos de redes de filas finitas dependentes do estado é que o fenômeno do congestionamento é explicitamente considerado, ou seja, a velocidade de percurso do usuário decai quando o número de usuários do sistema aumenta. Apresentamos uma descrição detalhada do modelo de simulação para estimar as medidas de desempenho de redes de filas dependente do estado e mostramos resultados computacionais para um vasto conjunto de instâncias. Como mostramos, os modelos dependentes do estado levam a novas e interessantes conclusões como, por exemplo, que em alguns casos uma mistura de distribuições irá descrever melhor o tempo de residência em células do que as distribuições de probabilidades clássicas comumente utilizadas.

Palavras Chaves - Simulação; performace; dependente do estado; redes de filas.

Abstract

We apply finite state-dependent queueing networks to model travel time in mobile communication systems. Although they have been successfully used in the past to model vehicular traffic, state-dependent models have not been applied to mobile communication systems, to the best of our knowledge. The novelty of state-dependent stochastic mobility models is that the congestion phenomenon is explicitly considered, that is, the user speeds fall when the number of users in the system increases. We present a detailed description of the simulation model used to estimate the performance measures of the queueing networks and show computational results for a comprehensive set of instances. As we show, finite state-dependent stochastic models bring interesting new insights, for instance, that in some cases mixed bimodal distributions will better describe the cell residence time of a call than the classical probability distributions used in the past.

Keywords - Simulation; performance; state dependent; queueing networks.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1
Introdução

1.1 Preliminares

Com o aumento da população, por exemplo a população brasileira (Figura 1.1), é muito comum, hoje em dia, nos depararmos com longos tempos de espera em filas, sejam elas em bancos, em supermercados, na cantina da faculdade, em engarrafamentos e em muitos outros lugares. O tempo é um bem precioso e não gostamos de desperdiçá-lo esperando em filas. Às vezes, indignados, ficamos nos perguntando o porquê de existir filas e sempre culpamos alguém por isso. Na verdade, o que acontece é que, com o aumento da população, aumenta a demanda por serviços, como bancos, supermercados e, consequentemente, aumenta o número de veículos utilizados por esta população, como podemos observar na Figura 1.2, que nos mostra a evolução da frota circulante brasileira. Porém, na maioria das vezes, os sistemas não estão preparados para esse aumento. Assim, a demanda por serviços é maior do que a oferta de servidores, ocasionando as tão desagradáveis filas e os longos tempos de espera. Vem então a pergunta: se aumentou a demanda, por que não se aumenta também o número os servidores, para que não ocorram essas filas? Nem sempre é tão simples aumentar o número de servidores, seja pelo alto custo para alocar mais servidores ou até mesmo por impossibilidade física. Imagine um supermercado no qual existam 10 caixas (que seriam os servidores). Quando foram colocados esses 10 caixas, não havia fila. Porém hoje, com o aumento de clientes, começaram a surgir as filas, os clientes começam a ficar irritados com elas, mas não há outra opção a não ser esperar, uma vez que percebem que naquele supermercado não haveria espaço para colocar mais caixas. Um outro exemplo: quando saímos pela manhã para irmos trabalhar ou à tarde quando voltamos do trabalho, enfrentamos a *hora do rush*, e temos de enfrentar um grande engarrafamento. O que acontece é que as ruas de nossa cidade não foram projetadas para ter o fluxo de veículos que tem hoje e por isso acontecem os engarrafamentos. Então culpamos o governo por não alargar as avenidas, mas se formos analisar com frieza, não é nada fácil alargar avenidas, pois além do inconveniente de obras, na maioria das vezes há casas em ambos os lados da avenida, precisando então que, para o alargamento, sejam desapropriadas. Isso acaba gerando um alto custo. Muitas vezes, pensamos que ao aumentar o número de caixas de um supermercado ou alargar uma avenida, seria o suficiente para não ocorrer filas, mas será que isso realmente iria resolver o problema? Para tentar responder esse tipo de pergunta, surgem novas aplicações da *teoria das filas*, em que se tenta modelar várias situações reais, com a finalidade de otimizar seu funcionamento.

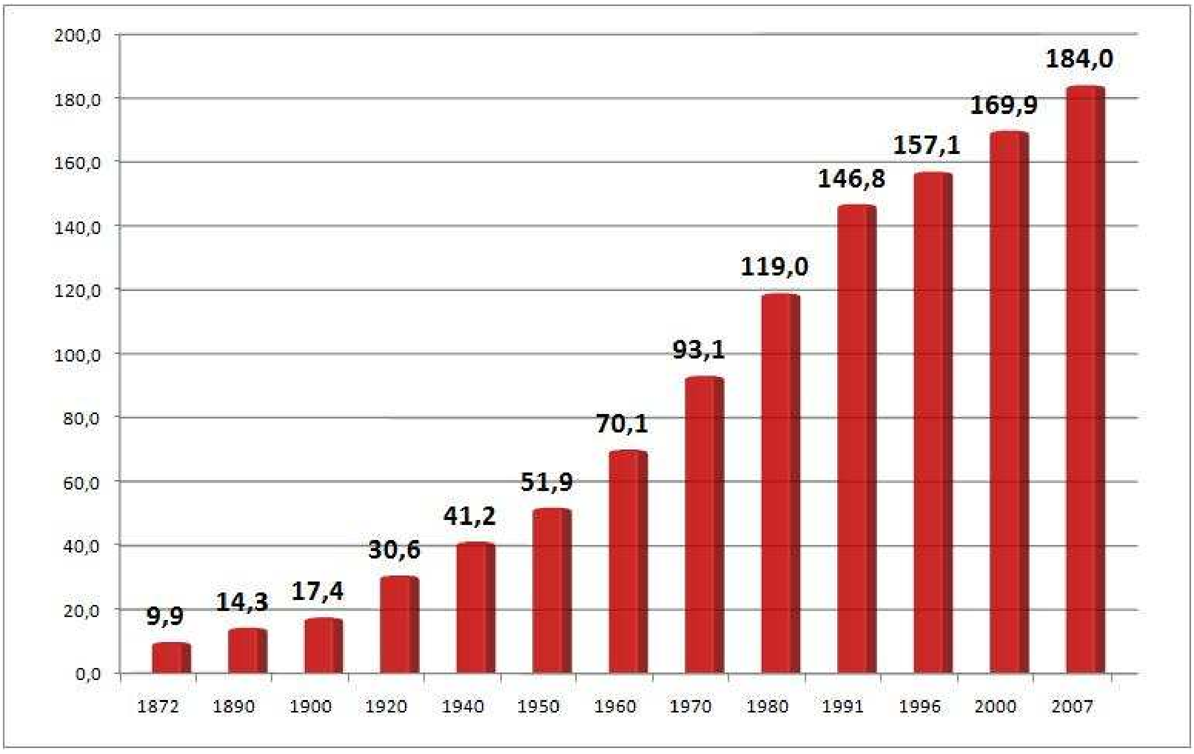
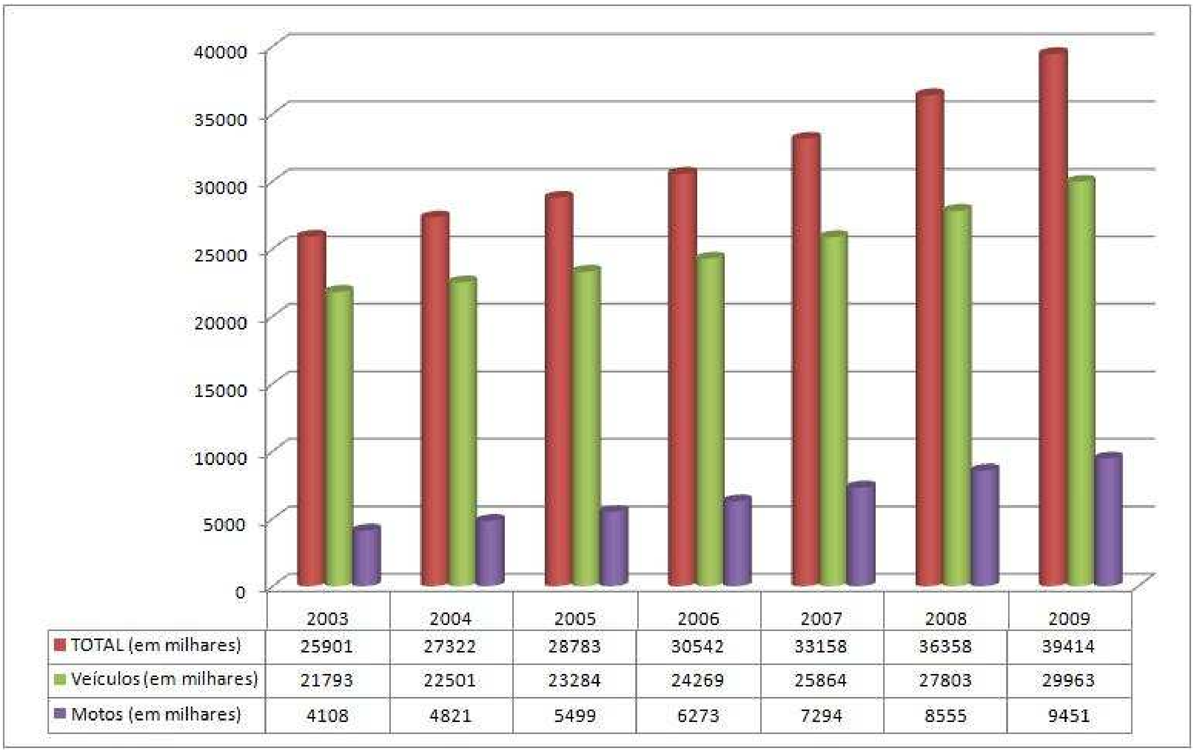


Figura 1.1: Evolução da população brasileira (IBGE, 2007)

1.2 Funções de tempo de percurso

Têm surgido na literatura diversas tentativas bem sucedidas de modelar o tempo de percurso em uma rede sujeita a congestionamento (por exemplo, veja Helbing et al., 2005, e referências). Duas linhas principais de trabalho são encontradas, a dos modelos determinísticos *versus* a dos modelos estocásticos. Entretanto, os tempos de percurso são geralmente assumidos determinísticos ou aproximadamente estocásticos.



	2003	2004	2005	2006	2007	2008	2009
TOTAL (em milhares)	25901	27322	28783	30542	33158	36358	39414
Veículos (em milhares)	21793	22501	23284	24269	25864	27803	29963
Motos (em milhares)	4108	4821	5499	6273	7294	8555	9451

Figura 1.2: Frota circulante brasileira (Sindipeças, 2009)

Tipicamente, os custos de percurso são expressos em termos de funções determinísticas do tempo de viagem (Prashker & Bekhor, 2000), embora tais tempos sejam reconhecidamente muito variáveis entre viagens, ao longo do dia ou entre dias. Os principais modelos de tempo de percurso são construídos geralmente tomando por base fórmulas clássicas que foram construídas ao longo dos últimos 40 anos, como por exemplo, a conhecida fórmula BPR (Bureau of Public Roads, 1964) usando dados empíricos do *Highway Capacity Manual*.

Kimber & Hollis (1979) desenvolveram uma outra fórmula para tempo de percurso baseada em uma aproximação para o modelo de filas $M/G/1/\infty$ dependente do tempo. Da conhecida notação de Kendall (1953), M representa um processo de chegada markoviano, G representa uma distribuição geral do tempo de serviço, 1 representa o número de servidores e, finalmente, ∞ é a capacidade total do sistema. Uma vez que expressões para o regime transitório dos modelos $M/G/1/\infty$ são intratáveis analiticamente, eles desenvolveram uma aproximação baseada em uma técnica de transformação de coordenadas para ajustar a fórmula de regime permanente aos efeitos transitórios da fila. Em sua abordagem, eles conseguem levar em conta o tráfego existente no trecho (do inglês *link*) da auto-estrada sob estudo, mas utilizam uma taxa de serviço fixa μ, uma fila de capacidade infinita e apenas um servidor para o tráfego (em outras palavras, utilizam-se de filas $M/G/1/\infty$).

Posteriormente Akçelik (1991) estendeu o trabalho de Kimber & Hollis (1979) com a dedução de expressões baseadas em técnicas de transformação de coordenadas, reconhecidamente mais eficazes para modelar o tempo de percurso, especialmente sob efeitos de severos congestionamentos, como aqueles observados durante os horários de *rush*, quando a procura excede largamente a capacidade (Ceylan & Bell, 2005). O desempenho do modelo de Akçelik é semelhante ao de Kimber & Hollis. Para tais funções típicas de desempenho, bons métodos para tratamento numérico são conhecidos.

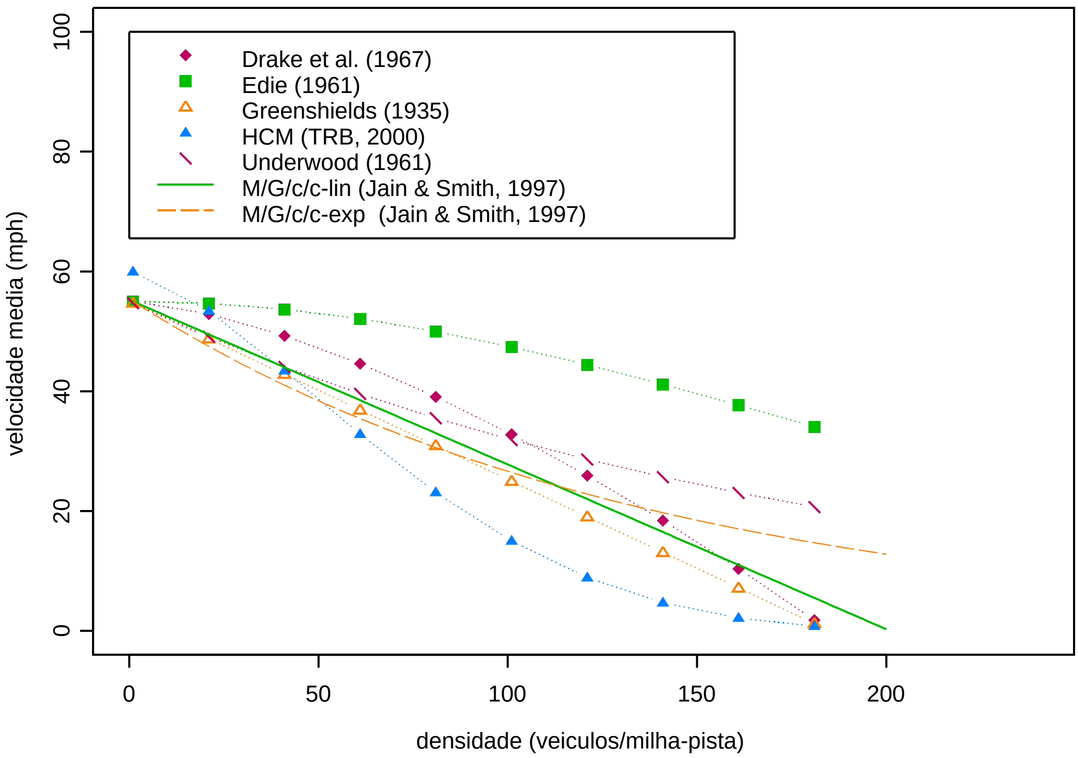


Figura 1.3: Distribuições empíricas para tráfego de veículos (Drake et al., 1967; Edie, 1961; Greenshields, 1935; Transportation Research Board, 2000; Underwood, 1961) e modelos dependentes do estado (Jain & Smith, 1997)

Pretendemos argumentar que uma abordagem estocástica é uma alternativa possível e mais aderente à realidade, quando redes de filas finitas dependentes do estado são utilizadas. De fato, os modelos dependentes do estado podem lidar com taxas de serviço bastante gerais, considerando múltiplos servidores e tanto com soluções transientes quanto com soluções de regime estacionário. Esta é uma abordagem genuinamente estocástica, sem aproximações. A Figura 1.3 apresenta resultados de vários estudos empíricos para auto-estradas norte-americanas (Drake et al., 1967; Edie, 1961; Greenshields, 1935; Transportation Research Board, 2000; Underwood, 1961) confrontados com resultados obtidos por modelos dependentes do estado Jain & Smith (1997). O congestionamento pode ser percebido como uma óbvia diminuição da velocidade média quando a densidade de veículos aumenta, o que resulta nas bem conhecidas curvas de velocidade-fluxo-densidade (ver, por exemplo, o trabalho seminal de Greenshields, 1935).

Em particular, apresentamos aqui uma versão estocástica para os custos incorridos em cada trecho de uma rede (i.e., os tempos de percurso) em que o modelo provém de redes de filas $M/G/c/c$ dependentes do estado. De fato, pela relevância ao presente estudo, descreveremos em detalhes no Capítulo 2 o desenvolvimento dos modelos de filas $M/G/c/c$ dependentes do estado no contexto de interesse. As filas $M/G/c/c$ dependentes do estado possibilitam a dedução de uma expressão estocástica para uma estimação do tempo de percurso que leve em conta os importantes efeitos de congestionamento.

Os modelos de filas $M/G/c/c$ dependentes do estado foram introduzidos por Yuhaski & Smith (1989), em estudos sobre tráfego de pedestres. O artigo de Yuhaski & Smith (1989) constitui-se então a base de vários outros modelos posteriormente desenvolvidos para determinação de tempos de percurso em diferentes contextos. Em sequência, o artigo de Cheah & Smith (1994) trouxe algumas generalizações e demonstrou que as filas dependente do estado possuem a propriedade de quase-reversibilidade (i.e., o processo de saída é igual ao processo de entrada se forem incluídas as entidades que são rejeitadas). Mais recentemente, o artigo de Jain & Smith (1997) mostrou como as filas dependentes do estado poderiam ser usadas para modelar o congestionamento de tráfego de veículos. Os modelos $M/G/c/c$ de filas estado-dependentes foram apontados por van Woensel et al. (2006) como uma alternativa válida para descrever os fluxos de tráfego e as velocidades. Além disso, os modelos $M/G/c/c$ dependentes do estado têm sido utilizados com sucesso na modelagem de fluxos de veículos (Jain & Smith, 1997) e de fluxos de pedestres (Cruz & Smith, 2007). Finalmente, Cruz, van Woensel, Smith & Lieckens (2010) mostraram recentemente que os modelos $M/G/c/c$ dependentes do estado são também bastante eficazes para modelar o tempo de percurso em trechos (*links*) simples, em comparação com expressões já bem estabelecidas, BRP e Akçelik, devido à sua capacidade de representar o congestionamento do tráfego.

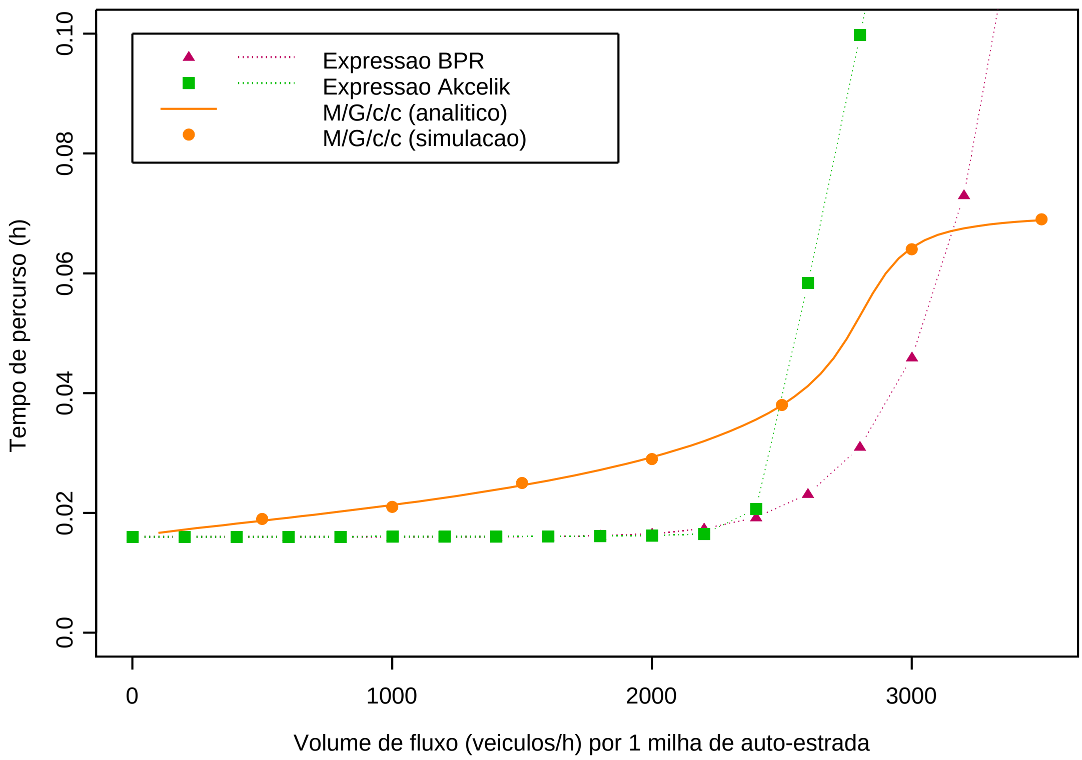


Figura 1.4: Fluxo de veículo por trecho de 1 milha

A Figura 1.4 mostra funções típicas para o tempo de percurso, BRP e Akçelik, (usadas recentemente, por exemplo, por Ghatee & Hashemi, 2009; Pursals & Garzón, 2009) em comparação com a função baseada em modelos de filas $M/G/c/c$ dependentes do estado (Jain & Smith, 1997) para um trecho de auto-estrada de 1 milha de comprimento, com velocidade máxima (velocidade de fluxo livre) de 62,5 mph ($\approx$ 100 km/h) e capacidade de 2.400 veículos/h, de acordo com o *Highway Capacity Manual* (Transportation Research Board, 2000). Além disso, a Figura 1.5 mostra como se comporta o tempo de percurso em função da taxa de chegada de vários trechos, via modelos de filas $M/G/c/c$ dependentes do estado. Note que, sob baixo tráfego, ou seja, um tráfego menor ou igual a capacidade de 2.400 veículos/h, a abordagem por filas produz resultados semelhantes aos de expressões clássicas e reconhecidamente precisas, tais como a BPR e a de Akçelik, conforme pode ser visto na Figura 1.4.

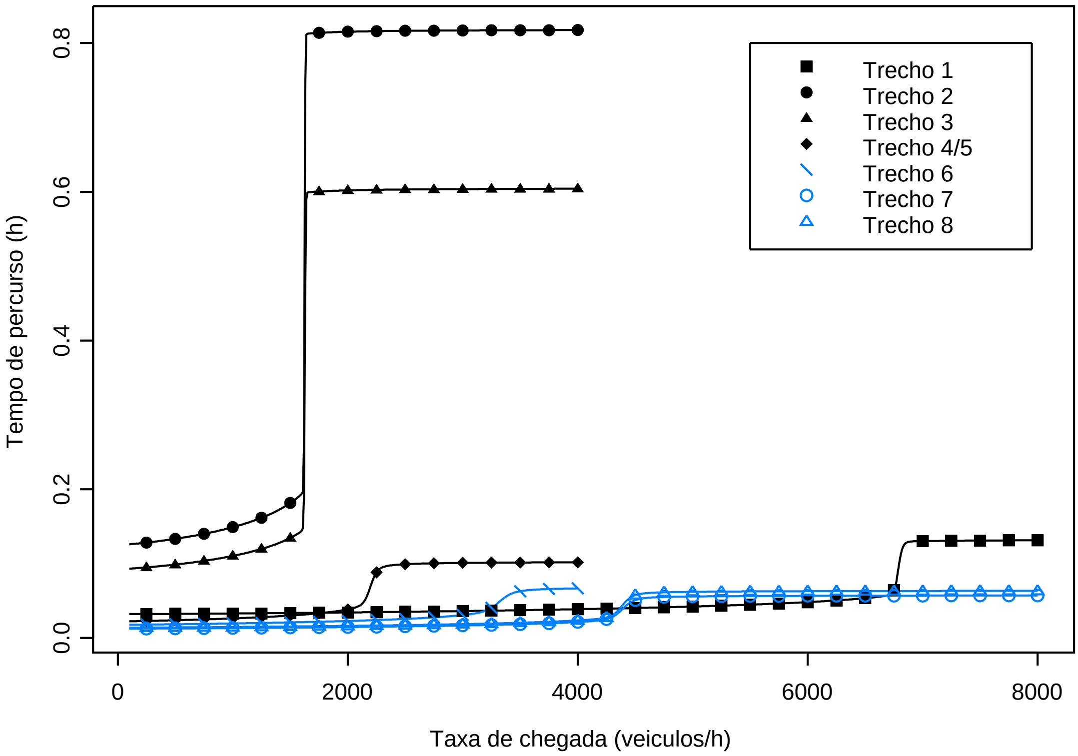


Figura 1.5: Tempo de percurso via modelo $M/G/c/c$ dependente do estado

Importante ressaltar que, pelas Figuras 1.4 e 1.5, as funções de tempo de percurso deduzidas via modelo $M/G/c/c$ não são *convexas*, mas têm uma forma de S. Esta característica poderá produzir ótimos locais quando for utilizada em modelos de atribuição de tráfego e, também, reduzir o tempo de simulação. Na realidade, isto quer dizer que a partir do momento que se excede a capacidade o tempo de percurso satura, sendo quase o mesmo para todos os veículos, isto ocorre em qualquer tipo tipo de pista, como podemos verificar na Figura 1.5 que nos mostra diferentes trechos, os trechos 2, 3, 4/5 e 6 são trechos longos e estreitos, nota-se que ocorre a saturação do tempo de percurso já com uma taxa de chegada menor e dependendo do estreitamento da pista esta saturação ocorre de uma forma mais rápida. Já para os trechos 1, 7 e 8 onde os trechos são curtos e largos, a saturação também ocorrem, mas com uma taxa de chegada alta e não tão bruscamente. Para as expressões BPR e Akçelik este tempo tende a explodir, o que na práica não acontece. Consequentemente, a introdução dos modelos estocásticos $M/G/c/c$ dependentes do estado poderão trazer importantes implicações teóricas. Para uma revisão sobre o uso de filas para modelar fluxos de tráfego e congestionamento, recomenda-se o recente artigo de van Woensel & Vandaele (2007). Para uma outra tentativa bem sucedida de refinar a estimativa de tempo de percurso, recomenda-se o artigo de García-Ródenas et al. (2006).

1.3 Motivação

O interesse em estudar o desempenho dos sistemas de telefonia celular, por meio de modelos estocásticos, aumentou significativamente recentemente, com a crescente demanda por serviços de qualidade. Como podemos ver na Figura 1.6. No Brasil a cada 100 pessoas 90 possuem um aparelho celular e todas elas desejam um serviço de qualidade . Embora os resultados ainda sejam modestos e restritos a problemas simples, a compreensão dos técnicos da área continua a aumentar (ver, por exemplo, Alfa & Liu, 2004). É um fato bem conhecido que a mobilidade dos usuários tem gerado novos desafios para os engenheiros responsáveis pela concepção, planejamento, dimensionamento e manutenção de redes de telefonia celular. Em sistemas móveis, os usuários querem se deslocar e ainda manterem-se conectados ao sistema de celulares (ver Figura 2.1, adaptada de Zonoozi & Dassanayake (1997)). A fim de cumprir este requisito, o sistema de telefonia celular deve manter e atualizar periodicamente informações sobre todos os usuários. Por meio de modelos matemáticos, tenta-se prever o comportamento desses usuários, a fim de reduzir a quantidade de informações coletadas e armazenadas. Na verdade, foi reconhecido há cerca de uma década atrás que os modelos de mobilidade desempenham um dos papéis mais importantes na descrição e concepção dos sistemas de telefonia celular (Zonoozi & Dassanayake, 1997). Entre os parâmetros de interesse em um sistema de telefonia celular, diretamente influenciada pela mobilidade, estão o *handover* (transferência de chamadas entre centrais), o tráfego disponibilizado, o dimensionamento de redes de sinalização, a atualização da localização dos usuários, registro, *paging* (procura pelo usuário na rede) e a gestão multicamadas das redes (Zonoozi & Dassanayake, 1997).

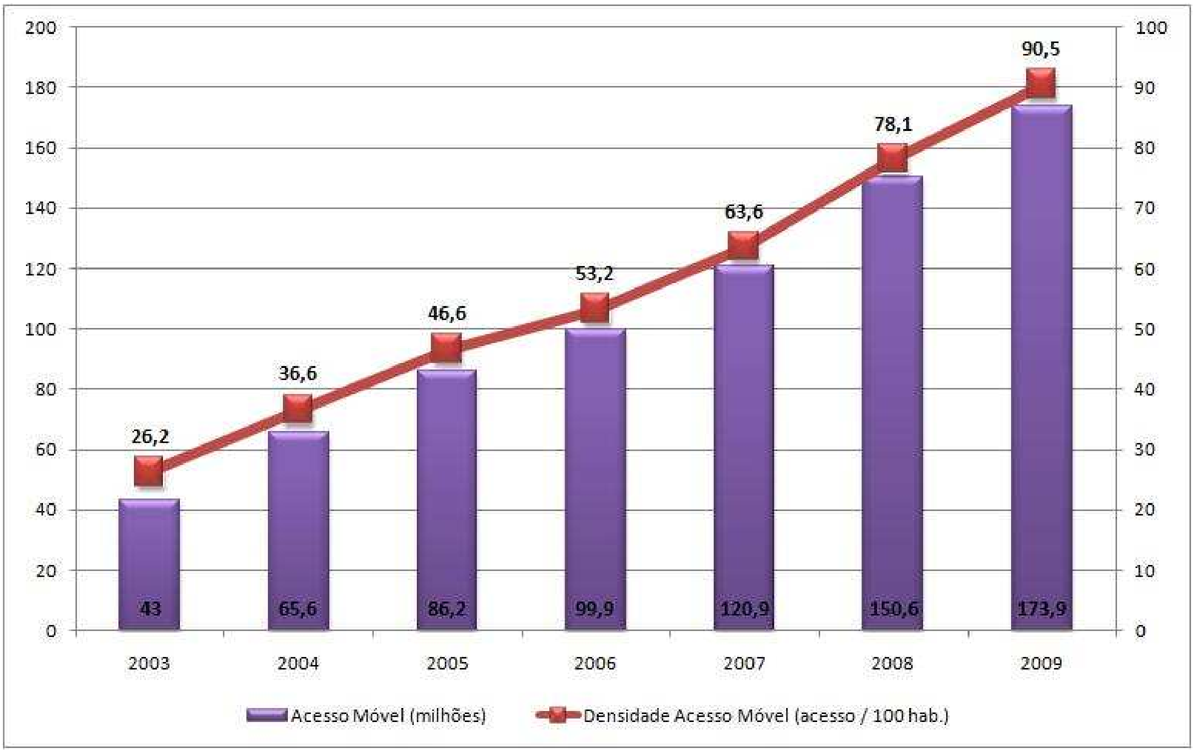


Figura 1.6: Quantidade e densidade de celulares (Anatel, 2009)

Uma revisão da literatura disponível na área mostra que muitos autores têm lidado com modelos de mobilidade para as redes de comunicação móvel, tal como apresentado na Figura 1.7. Abordagens mais recentes costumam considerar modelos estocásticos para a velocidade dos usuários. Distribuições de probabilidades uniformes e não uniformes foram usadas. Modelos de velocidade exponencialmente distribuídos e com distribuição geral também têm sido propostos. No entanto, nenhuma pesquisa pode ser localizada que considere modelos estocásticos dependentes do estado para a descrição das velocidades médias como uma função do número de usuários presentes no sistema de telefonia celular. Este é o tipo de modelo que propomos no presente trabalho, que leva em consideração que o tempo de percurso tende a saturar quando o sistema está congestionado.

Muitos dos efeitos da mobilidade dos usuários sobre o desempenho do *handover* foram investigados por Han (2002). No entanto, as velocidades dos carros foram consideradas exponencialmente distribuídas e as velocidades dos pedestres, uniformemente distribuídas. Embora essas considerações possam ser aceitáveis como uma aproximação destinada simplesmente a tornar o modelo computacionalmente mais tratável, elas podem levar à conclusão de que tanto o tempo de permanência na célula quanto o tempo de retenção

- Modelos de mobilidade
  - Velocidade uniforme
    - Gravitacional (Lam et al., 1997)
    - Modelo ‘city area’ (Markoulidakis et al., 1997)
    - Markoviano (Lam et al., 1997, 1996)
    - Hong & Rappaport (1986)
    - ...
  - Velocidade não-uniforme
    - Exponencial
      - Fluido (Tomas et al., 1988)
      - Modelo ‘metropolitan mobility’ (Lam et al., 1997, 1996)
      - McMillan (McMillan, 1993)
      - ...
    - Geral
      - Markoulidakis et al. (1998)
      - Nanda (Nanda, 1993)
      - Sadok (Alencar & Sadok, 1999; Cavalcanti et al., 2000)
      - Matemático & Tabular (Silva et al., 2002)
      - Evans (Evans, 1995)
      - ...
    - Dependente do estado
      - Cruz, Oliveira & Duczmal (2010)
      - ...

Figura 1.7: Modelos de mobilidade

do canal (tempo durante o qual uma chamada ocupa uma linha) é markoviano, o que não é verdade, de acordo com os resultados relatados por Hegde & Sohraby (2002). De fato, dados de simulação analisados por Zonoozi & Dassanayake (1997) mostram que a distribuição gama generalizada é uma melhor aproximação para a distribuição do tempo de permanência na célula e que o tempo de retenção de um canal da rede celular pode ser melhor descrito por uma distribuição exponencial negativa.

No entanto, como mostramos nas seções seguintes, não parece que uma única distribuição de probabilidade seja adequada para um modelo de rede de comunicação móvel. O tempo de residência na célula é fortemente afetado pelos efeitos do congestionamento dentro da célula. Em outras palavras, com base em hipóteses razoáveis, a velocidade média dos veículos deve ser considerada dependente do estado (ver Figura 1.5). Mostramos que às vezes é possível que uma distribuição hipoexponencial seja um modelo razoável para a variável aleatória *tempo de permanência* (ou seja, a variável aleatória segue uma distribuição de probabilidade que tem uma variabilidade menor que a de uma distribuição exponencial), em outras uma mistura de distribuições de probabilidade pode surgir, pela simples variação do nível de congestionamento do sistema sob análise. Naturalmente, os efeitos destas descobertas sobre o resultado da análise de desempenho global de um sistema de telefonia móvel não será pequeno, dada a forte ligação existente entre a mobilidade dos usuários e a qualidade do serviço nestas redes (Manner et al., 2002).

1.4 Organização do texto

Este texto está organizado da seguinte forma: o modelo de simulação dependente do estado para a mobilidade é descrito em detalhes no Capítulo 2. O Capítulo 3 apresenta os resultados experimentais obtidos para sistemas de comunicação móvel de pequena escala. O Capítulo 4 fornece as conclusões, que inclui um resumo dos principais resultados obtidos e uma discussão de algumas questões suscitadas pelo estudo de simulação realizado, questões estas ainda em aberto e sugeridas como possíveis tópicos para trabalhos futuros na área.

Capítulo 2
Um Modelo de Mobilidade Dependente do Estado

2.1 Introdução

Nos dias de hoje, em que se vê o aumento de número de veículos e, consequentemente, o aumento da quantidade de congestionamentos, vê-se também a necessidade de uma modelagem que se adeque a este novo cenário. Geralmente, este cenário é modelado com o auxílio da teoria das filas. São vários os fatores que influenciam o tráfego de veículos, como por exemplo, sua quantidade e velocidade máxima, o número de pistas do trecho da via, entre outros. Quando modelamos avenidas e ruas através de redes de filas temos várias medidas de desempenho de interesse, como por exemplo o tempo de percurso (*travel time*) em determinado trecho, tempo que o usuário, a pé ou em um veículo, gasta do início ao fim de um determinado trecho. Existem várias funções que são utilizadas para modelar este tempo. Detalharemos aqui apenas a função estocástica de tempo de percurso.

2.2 Parâmetros de mobilidade

A fim de realizar uma análise de um sistema de comunicação móvel, alguns parâmetros devem ser definidos. Na Figura 2.1, vê-se um usuário trafegar através de uma rede celular. Sua trajetória é iniciada na célula 0, em que o tempo de permanência é $T_{m,0}$, com o início de uma nova chamada. O tempo de retenção do canal nesta célula é dado por $\tau_{m,0}$. Com o passar do tempo e com a movimentação do usuário através de sua rota, o sistema muda automaticamente sua ligação da célula 0 para a célula 1, o que é comumente chamado de *handoff* (ou *handover*). Com a movimentação do usuário de célula em célula através da rede de telefonia celular, ele acabará por chegar a célula i, quando a chamada é finalizada. Um dos parâmetros mais importantes para descrever a mobilidade do usuário é a variável aleatória $T_{m,i}$, que representa o tempo que um usuário m gasta na célula i. O foco principal deste texto é descrever o desenvolvimento de um modelo que seja melhor e mais preciso para descrição da variável aleatória $T_{m,i}$.

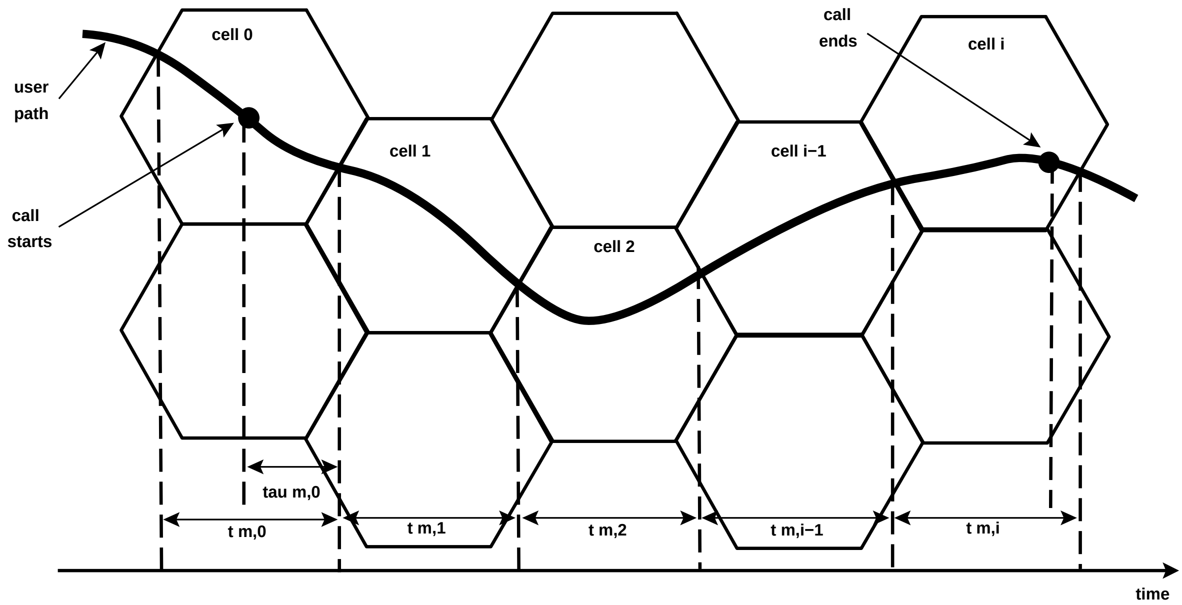


Figura 2.1: Diagramas de mobilidade temporal (Zonoozi & Dassanayake, 1997)

2.3 Modelos de congestionamento

Vamos usar redes de filas para a modelagem do tráfego. Para uma discussão, recentemente publicada sobre as conveniências e as vantagens dos modelos de redes de filas em modelagem de tráfego, recomenda-se o artigo de van Woensel & Cruz (2009). Um conjunto especial de filas finitas, conhecido como filas $M/G/c/c$ dependentes do estado, tem sido usado especificamente para a modelagem do congestionamento em redes de tráfego de veículos (Jain & Smith, 1997). Conforme a notação de Kendall (1953), M representa um processo de chegada markoviano, G representa uma distribuição geral do tempo de serviço, que aqui é dependente do estado (o tempo de serviço reduz-se com o número de usuários presentes no sistema), c representa o número de servidores em paralelo e, finalmente, c é a capacidade total do sistema *incluindo* aqueles correntemente em serviço.

A característica mais importante dos modelos $M/G/c/c$ dependentes do estado é que a sua taxa de serviço (isto é, a velocidade) diminui com o aumento no número de usuários no sistema, como mostrado na Figura 1.3 (adaptada de Jain & Smith (1997)), que apresenta curvas empíricas relacionadas a várias auto-estradas norte-americanas. Usando filas $M/G/c/c$ dependentes do estado, os trechos de rodovia podem ser visto como servidores em paralelo para seus ocupantes. O número máximo de ocupantes simultâneos iguala-se à capacidade do respectivo trecho, que também é igual à quantidade total de usuários permitidos no sistema. Esta capacidade é dada por

$$c = \lfloor k \times l \times w \rfloor,$$

em que l é o comprimento do trecho de rodovia, w é o número de pistas, c é a capacidade total, e $\lfloor x \rfloor$ é a função piso, isto é, o maior inteiro não superior ao argumento x. A constante k representa a densidade de tráfego, que é a densidade de veículos a partir da qual o fluxo se interrompe. Diferentes estimativas de k encontraram valores na faixa de 115-165 veículos/km-pista ($\approx$ 185-265 veículos/milha-pista, conforme relatado por Jain & Smith (1997); neste trabalho assumimos 124 veículos/km-pista ($\approx$ 200 veículos/milha-pista). Note-se que a discussão sobre a capacidade acima é apenas em termos do número de carros que podem caber fisicamente em um determinado trecho de rodovia, que não necessariamente correspondem à capacidade de chamadas de um sistema de telefonia celular, isto é, não corresponde ao número de chamadas que uma determinada central de telefonia móvel pode sustentar simultaneamente em um dado momento.

No modelo de congestionamento, o tráfego flui através do trecho de estrada a uma velocidade média V_n, que é uma função do número de veículos n atualmente em circulação e de sua capacidade c. Com base em dados empíricos, modelos analíticos (linear e exponencial) foram desenvolvidos por Yuhaski & Smith (1989), a serem descritos por meio das seguintes quantidades:

$V_n \rightarrow$ velocidade média para uma ocupação de n veículos;

$V_1 \rightarrow$ velocidade média para trânsito livre;

$V_a \rightarrow$ velocidade média para uma ocupação de a veículos/km-pista;

$V_b \rightarrow$ velocidade média para uma ocupação de b veículos/km-pista.

Os valores de a e b são pontos arbitrários utilizados para ajustar a curva exponencial. Ambos os modelos (linear e exponencial) geralmente se encaixam de forma satisfatória aos dados empíricos de tráfego e produzem resultados bastante satisfatórios, ver Jain & Smith (1997). Por concisão, vamos apresentar aqui apenas o modelo exponencial, que é aquele efetivamente utilizado neste trabalho:

$$V_n = V_1 \exp\left[-\left(\frac{n-1}{\beta}\right)^{\gamma}\right], \tag{2.1}$$

em que

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

e

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

Em aplicações relacionados a veículos, os valores comumente usados são de $k = 200$ veículos/milha-pista, como dito anteriormente, com $a = 20$ e $b = 140$, correspondente às densidades de veículos de 20 e 140 veículos/milha-pista, respectivamente. Olhando para as curvas apresentadas na Figura 1.3, valores razoáveis para tais pontos são $V_a = 48$ mph e $V_b = 20$ mph. A distribuição de probabilidade do número de usuários no sistema, em função do λ (taxa de chegada), é:

$$p(n) \equiv P[N = n] = \left[\frac{\Big(\lambda \times E[T_1]\Big)^n}{n! f(n) \dots f(1)} \right] \times p(0), \tag{2.2}$$

para $n = 1, 2, \dots, c$, em que

$$p(0) \equiv P[N = 0] = 1/\left\{ 1 + \sum_{i=1}^{c} \left[\frac{\Big(\lambda \times E[T_1]\Big)^i}{i! f(i) \dots f(1)} \right] \right\}$$

é a probabilidade de o sistema estar vazio, λ é a taxa de chegada, $E[T_1]$ é a esperança do tempo de serviço para um único ocupante no sistema, e $f(n) = V_n/V_1$ é a taxa de serviço para n usuários simultaneamente no sistema.

Por meio da Equação (2.2), é possível calcular várias medidas de desempenho, tais como a probabilidade de bloqueio, a taxa de atendimento (do inglês *throughput*), o número esperado de usuários no sistema (também conhecido pelo termo em inglês *work-in-process*), e tempo de serviço esperado, entre outras. A probabilidade de bloqueio é a probabilidade de um usuário adicional chegar ao sistema quando o número de usuários já presentes nele estiver na capacidade máxima c, ou seja:

$$p_{\text{bloqueio}} \equiv p(c) \equiv P\left[N = c\right].$$

A taxa de atendimento (*throughput*), também conhecida como taxa de chegada efetiva no sistema, é dada por:

$$\theta \equiv \lambda_{\text{efetiva}} \equiv \lambda[1 - p(c)].$$

O número esperado de usuários no sistema resulta diretamente da definição de esperança de uma variável aleatória:

$$L \equiv E[N] = \sum_{n=1}^{c} np(n).$$

Já o tempo esperado no sistema (isto é, o tempo de serviço esperado) pode ser calculado diretamente da Lei de Little:

$$W \equiv E[T] = \frac{L}{\theta}.$$

2.4 Modelo de simulação a eventos discretos

Descrevemos um novo modelo de simulação (Cruz, Oliveira & Duczmal, 2010) que é extensão de um algoritmo proposto anteriormente por Oliveira (2005) para redes de filas $M/G/c/c$ dependentes do estado. Essencialmente, o modelo implementa o objeto `MgccSimul`, apresentado na Fig. 2.2. Descreveremos em detalhes agora o objeto `MgccSimul` e todas as estruturas de dados envolvidas, ou seja, o número de filas $M/G/c/c$ dependentes do estado (`nOfNodes`), o tempo total de simulação (`totalTime`), a matriz origem-destino (`arcs`), um vetor de `nOfNodes` objetos do tipo `MgccResource`, e, finalmente, uma fila de eventos (`MgccEventQueue`). Os objetos `MgccResource` mantêm todas as estatísticas de interesse para cada uma das filas, ou seja, o número de bloqueios (`sumBloc`), o número de chegadas (`sumArr`), o número de partidas (`sumDep`), o tempo acumulado no sistema (`sumTime`), e o número corrente de usuários no sistema (`users`). Também faz parte de cada objeto `MgccResource` o modelo de congestionamento (`GenCM`), com métodos para acesso da capacidade da fila (c), o tempo de serviço esperado para um único ocupante no sistema ($E[T_1]$) e a velocidade média (taxa de serviço) para o número corrente de usuários no sistema (V_n).

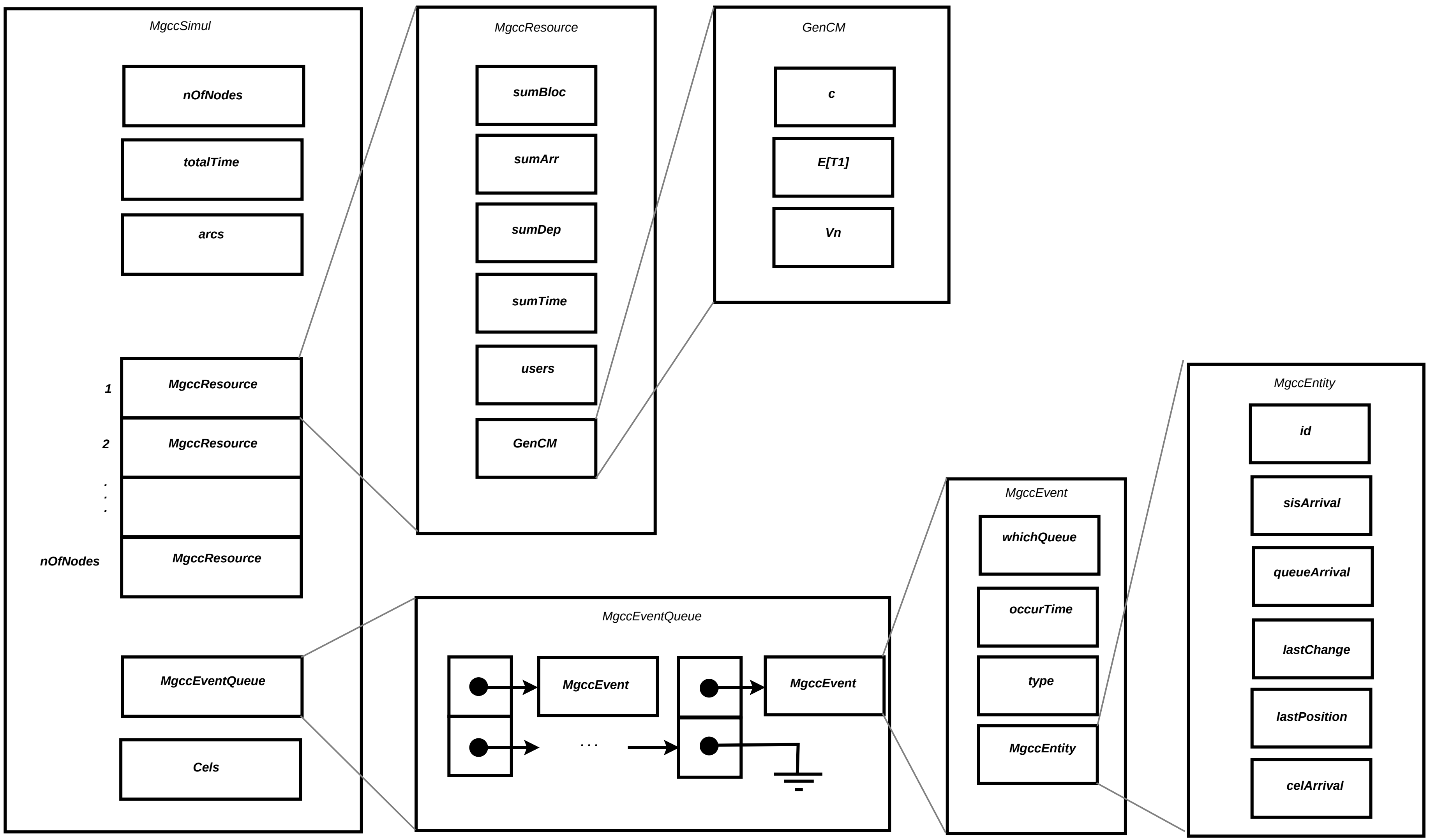


Figura 2.2: Objetos `MgccSimul`

A parte mais crítica do objeto `MgccSimul` é o objeto `MgccEventQueue`, que implementa a fila de eventos. A fila de eventos foi implementada como uma lista dinâmica, construída durante a execução da simulação, com a tarefa de registrar todos os eventos. Surpreendentemente, após muita experimentação, verificamos que é consideravelmente mais rápido manter a fila de eventos não ordenada, pelo menos para simulação de redes de filas dependentes do estado. Mesmo considerando o custo elevado de percorrer grande parte da lista para recuperar o próximo evento a ser processado, uma operação que tem complexidade $\mathcal{O}(n_{\mathtt{l}ista})$ no pior caso, é mais eficiente do que manter a lista ordenada. Isso decorre do fato de que a lista é embaralhada cada vez que uma entidade chega ou sai do sistema, uma vez que V_n, as taxas de serviço definidas na Equação (2.1), devem ser atualizadas para cada entidade que permanece no sistema.

Na fila de eventos, cada objeto `MgccEvent` tem as seguintes variáveis: `whichQueue`, que é a indicação de qual das filas $M/G/c/c$ o evento pertence; `occurTime`, que é o tempo de ocorrência do evento; `type`, que é o tipo de evento (entre os eventos possíveis `chegada`, `partida` e `fim_simulação`) e `MgccEntity`, que é a entidade a que se refere o evento. O objeto `MgccEntity` representa cada usuário (veículo) em uma rede de filas $M/G/c/c$ dependente do estado, que tem as seguintes variáveis: `id`, que é uma identificação numérica única; `sisArrival`, que é o momento em que a entidade chegou ao sistema; `queueArrival`, que é o tempo que a entidade chegou à fila atual; `lastChange`, que é o tempo de ocorrência da última mudança de estado (isto é, quando uma entidade se junta ou deixa uma fila particular, haverá uma mudança no seu estado, uma vez que a taxa de serviço é dependente do estado); `lastPosition`, que é a localização física da entidade no momento em que o estado na fila foi modificado pela última vez, e, finalmente, `celArrival`, que é o tempo de chegada da entidade na célula.

As células são definidas como um conjunto arbitrário de filas e essa informação é armazenada em uma matriz, `Cells`. Se uma fila determinada i pertence à uma célula particular j, então tem-se que

$$\mathtt{Cells}[\mathtt{i},\mathtt{j}] = \mathtt{TRUE}.$$

O algoritmo de simulação é apresentado em pseudo-código na Figura 2.3. Inicialmente, a lista de eventos `MgccEventQueue` é inicializada com o último evento (evento do tipo `fim_simulação`) e os primeiros eventos, que são as primeiras chegadas (evento do tipo `chegada`). Então, iterativamente, o evento a ocorrer primeiro é buscado na lista de eventos e processado, normalmente gerando outros eventos, que são incorporados à lista. Isto se prolonga até que o evento final (evento do tipo `fim_simulação`) seja aquele a ocorrer primeiro na lista de eventos. Os procedimentos para lidar com as chegadas, `ProcessArrival`(), e com as partidas, `ProcessDeparture`(), não serão detalhados aqui, por razões de concisão, uma vez que não são significativamente diferentes daqueles já descritos em Cruz et al. (2005).

```
algorithm Simulate
        /* initialize event queue */
    Inicitialize(MgccEventQueue);
        /* create and insert 'last' event */
    MgccEvent ← new();
    MgccEvent.occurTime ← totalTime;
    MgccEvent.type ← end_simulation;
    Insert(MgccEventQueue,MgccEvent);
        /* create and insert 'first' events */
    for ∀n| λn ≠ 0 do
        MgccEvent ← new();
        MgccEvent.whichQueue ← n;
        MgccEvent.occurTime ← 0.0;
        MgccEvent.type ← arrival;
        Insert(MgccEventQueue,MgccEvent);
    end for
        /* simulate */
    MgccEvent ← GetNext(MgccEventQueue);
    while MgccEvent.type ≠ end_simulation do
        if MgccEvent.type = arrival then
            ProcessArrival(MgccEventQueue,MgccEvent);
        else if MgccEvent.type = departure then
            ProcessDeparture(MgccEventQueue,MgccEvent);
        else
            error, unknown event
        end if
        MgccEvent ← GetNext(MgccEventQueue);
    end while
    print results
end algorithm
```

Figura 2.3: Algoritmo de simulação

2.5 Observações finais

Como observação final, lembramos que o objetivo aqui é estimar computacionalmente medidas de desempenho básicas para filas finitas configuradas em redes. Não foi assumido que todos os usuários nas rodovias terão chamadas em andamento, o que seria completamente irrealista. A relação entre a distribuição do número de clientes no sistema e a distribuição do número de usuários nas rodovias com chamadas em andamento é complexo e requer uma análise cuidadosa. Além disso, nesta formulação, os *handovers* (transferências de ligações entre centrais) não são considerados. Estas são certamente questões importantes que serão abordados em trabalhos futuros.

Capítulo 3
Experimentos Computacionais

3.1 Introdução

Apresentamos aqui os resultados dos experimentos computacionais realizados com o modelo de simulação a eventos discretos proposto. Todos os algoritmos foram codificados em C++ e estão disponíveis no Capítulo A, para fins educacionais e de pesquisa. Todos os experimentos foram executados em um mesmo PC com um processador 1.8 GHz Intel Pentium 4 e 512 MB de RAM, rodando Windows XP. As configurações foram executadas para um período de simulação de 3 horas, com descarte da primeira hora, para estabilização da simulação (este é conhecido como período de *warm-up*; detalhes podem ser vistos em Robinson, 2007).

Três topologias básicas de redes foram testadas. Elas foram escolhidas pela simplicidade, pelas conclusões que podem produzir e, principalmente, porque qualquer configuração mais complexa pode ser vista como uma combinação destas. Naturalmente, o efeito combinado de uma certa composição de topologias básicas não deverá ser uma perfeita superposição dos efeitos individuais das componentes (lembre-se que não temos um sistema linear), mas qualquer compreensão que se ganhe poderá ser útil na análise do comportamento de redes mais complexas, como veremos em breve. Como são experimentos iniciais, todos os trechos considerados tem 1 milha de comprimento e 1 pista de largura.

Uma das configurações básicas estudadas é a topologia série, apresentada na Fig. 3.1. Outra é a topologia divisão, vista na Fig. 3.6. A topologia fusão também foi testada e é mostrada na Figura 3.11.

Finalmente, a fim de melhor demonstrar as capacidades do modelo proposto, algumas topologias mistas, complexas, foram consideradas, o que pode ser visto nas Figuras 3.16, 3.21(a), 3.21(b) e 3.21(c). Vamos agora apresentar e discutir os resultados experimentais.

O principal objetivo aqui era fazer uma análise mais profunda das variáveis tempo entre partidas e tempo de serviço nas células, mas com o objetivo de verificar a saturação do tempo de percurso também foi feito alguns experimentos aumentando o número de pistas. Os resultados obtidos foram apresentados para a topologia série nas Figuras D.1 e D.1, para a topologia divisão as Figuras D.3 e D.3, para a topologia fusão as Figuras D.5 e D.5 e por fim para a topologia mista as Figuras D.7 e D.7. Em todas as situação vemos que o tempo de serviço reduz com o aumento do número de pistas, uma vez que com menos pistas o sistema fica congestionado causando a sua saturação. Podemos observar que há uma queda maior quando aumentamos o número de pistas de 1 para 2 pistas, após 2 pistas o tempo de serviço reduz porém em menor valor.

3.2 Topologia série

Na Figura 3.1, vemos a representação de um sistema de telefonia celular simplificado composto por três células em topologia série, cada uma das quais com apenas um trecho principal de rodovia, que será modelado cada um por uma fila $M/G/c/c$ dependente do estado. Sem perda de generalidade, cada trecho tem 1 milha de comprimento e uma única pista de largura.

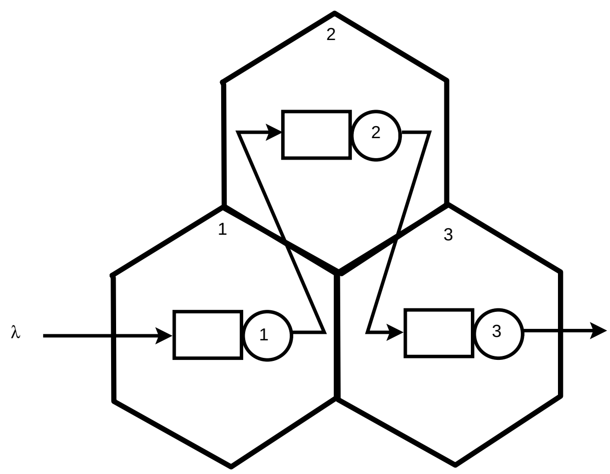


Figura 3.1: Três células em topologia série

As estatísticas descritivas da variável *tempo entre partidas* são mostradas na Tabela 3.1. É notável a equivalência dos modelos estocásticos para todas as três células, para todas as taxas de chegada λ testadas. No entanto, o efeito da dependência do estado já é perceptível, mesmo neste caso simples. Em outras palavras, até a taxa de chegada de 4.000 veículos/h, observamos uma equivalência aproximada entre as médias e os desvios-padrões para a variável *tempo entre partidas*. Acima destes valores, as médias permaneceram praticamente inalteradas, em torno de 1,85 (porque o sistema satura), mas a variabilidade cresce para 14,3. Esta nossa análise é corroborada pelas Tabelas C.2 e C.1, que foram obtidas através do teste de Kolmogorov-Smirnov, que mostram, respectivamente, os os valores-p e a decisão quando comparado ao nível de significância de 1$. Assim, os processos de chegada na células 2 e 3, que são os processos de partida das células 1 e 2, respectivamente, já não parecem ser markovianos, assemelhando-se a uma distribuição hiper-exponencial. Uma conclusão prática relevante que poderia ser tirada destes experimentos é que, se a carga no sistema é suficientemente alta, um modelo exponencial pode não mais ser adequado para descrever todos os processos de chegada, o que resulta na inaplicabilidade de filas $M/G/c/c$ dependentes do estado, para as quais o processo de entrada é markoviano.

Tabela 3.1: Descritivas da média do tempo entre partidas para a topologia série

λ	Célula	Min.	Q1	Mediana	Média	DP	Q3	Máx.
1.000	1	0,00	1,06	2,52	3,50	3,44	4,80	32,28
	2	0,00	1,06	2,49	3,50	3,43	4,90	30,24
	3	0,00	1,08	2,47	3,50	3,43	4,92	28,46
2.000	1	0,00	0,51	1,23	1,77	1,76	2,46	17,44
	2	0,00	0,51	1,22	1,78	1,77	2,47	16,44
	3	0,00	0,52	1,24	1,77	1,76	2,46	14,74
4.000	1	0,00	0,37	0,88	1,27	1,25	1,78	9,85
	2	0,00	0,37	0,86	1,27	1,25	1,77	10,26
	3	0,00	0,37	0,87	1,27	1,25	1,74	10,26
8.000	1	0,00	0,28	0,68	1,26	3,56	1,42	57,20
	2	0,00	0,28	0,69	1,28	3,30	1,45	53,45
	3	0,00	0,27	0,70	1,27	2,83	1,50	44,58
16.000	1	0,00	0,15	0,35	1,27	10,71	0,70	159,62
	2	0,00	0,14	0,35	1,27	10,55	0,71	157,28
	3	0,00	0,14	0,34	1,27	10,32	0,74	152,43

Nas Figura 3.2 e 3.3, apresentamos histogramas para a variável *tempo entre partidas*, para uma taxa de chegada de 1.000 e 4.000 veículos/h, respectivamente. A partir destas figuras e das Tabela 3.1, C.2 e C.1, a taxa de chegada de 4.000 veículos/h parece realmente ser o limite da aplicabilidade do modelo de redes de filas $M/G/c/c$ dependentes do estado. A adoção de um modelo exponencial, para uma taxa de chegada de 4.000, é visualmente razoável.

Nas Figuras 3.4 e 3.5, gráficos das séries temporais da variável *tempo de serviço nas células* são apresentados, juntamente com os respectivos histogramas, para as taxas de chegada de 1.000 e 4.000 veículos/h. Observamos que para aplicações em sistemas celulares, a variável *tempo de serviço nas células* é equivalente à variável *tempo de permanência nas células*, que é uma medida de desempenho importante em sistemas de telefonia móvel, como ressaltado anteriormente. Confirmamos aqui que os modelos exponenciais não parecem ser adequados para a modelagem de tal variável aleatória, como observado em estudos anteriores (Hegde & Sohraby, 2002; Zonoozi & Dassanayake, 1997).

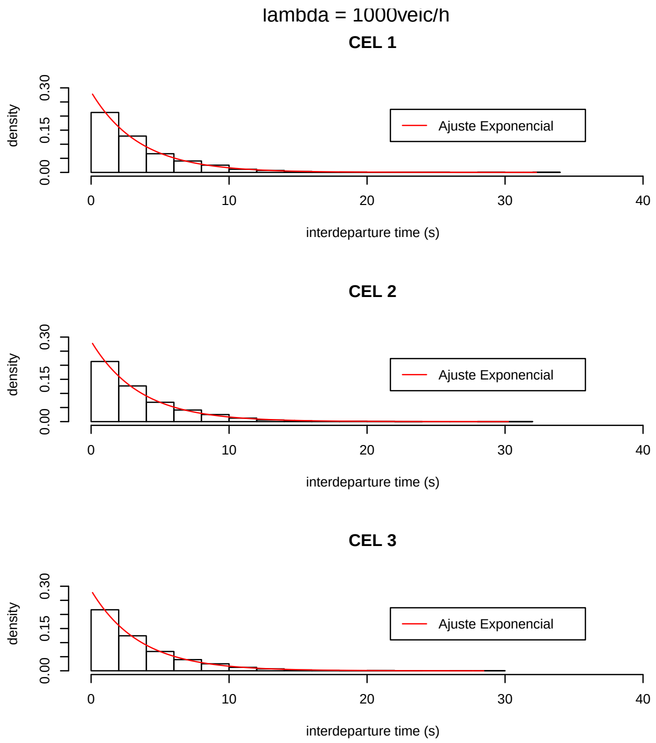


Figura 3.2: Tempo entre partidas na topologia série para $\lambda = 1.000$

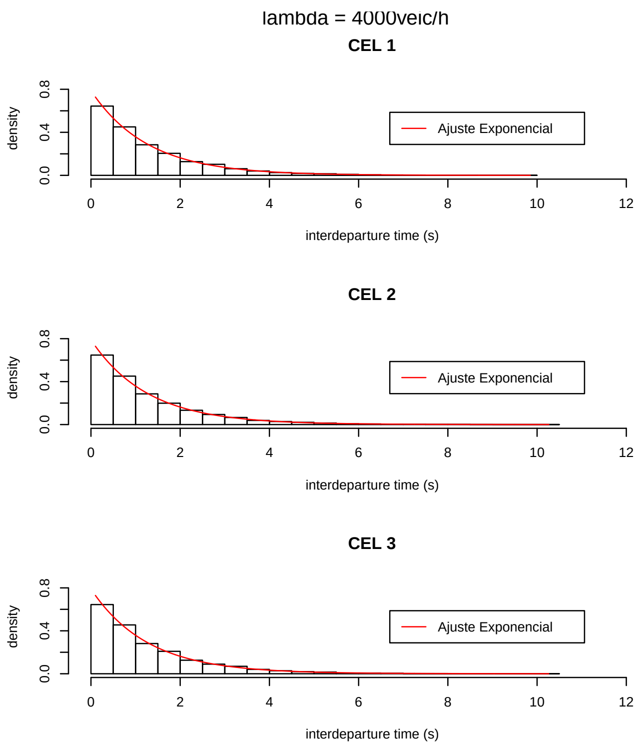


Figura 3.3: Tempo entre partidas na topologia série para $\lambda = 4.000$

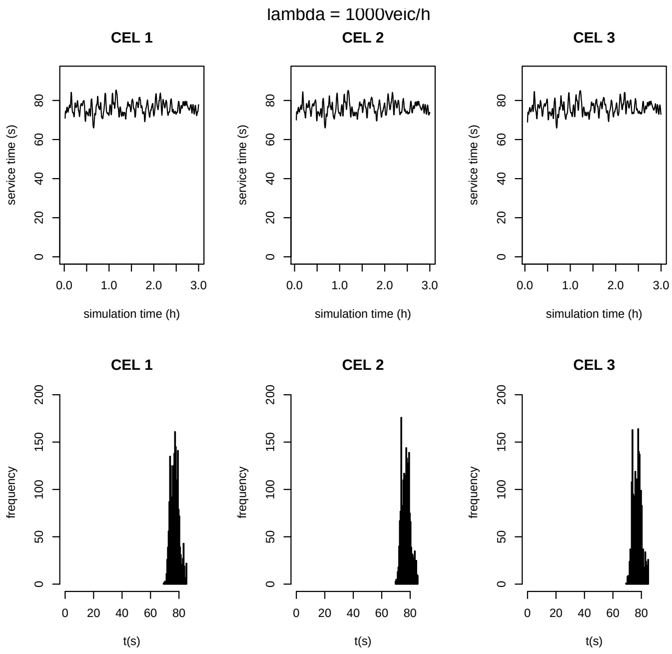


Figura 3.4: Tempos de serviço na topologia série para $\lambda = 1.000$

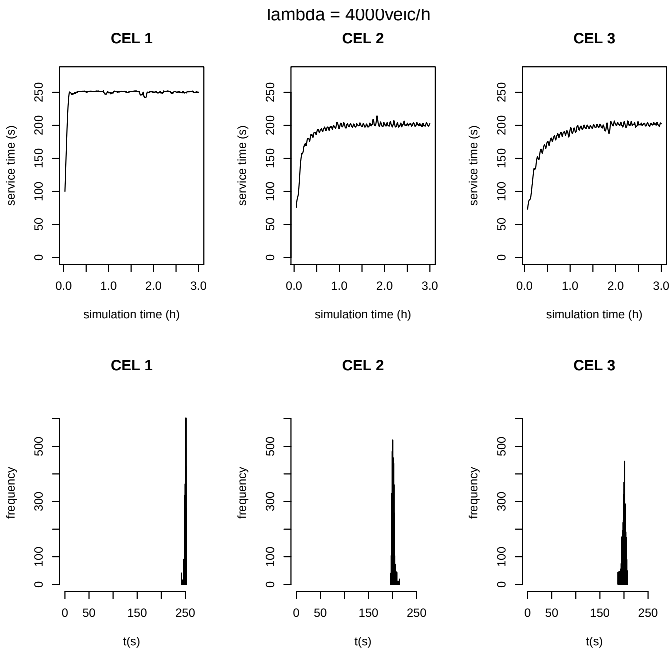


Figura 3.5: Tempos de serviço na topologia série para $\lambda = 4.000$

3.3 Topologia divisão

Na Figura 3.6, mostramos um sistema de telefonia celular em uma configuração simplificada na topologia divisão. Neste caso, cada fila modela um trecho de rodovia de 1 quilômetro de comprimento por uma pista de largura. O fluxo do trecho 1 se divide entre dois trechos, 2 e 3, na proporção de 70%–30%, respectivamente.

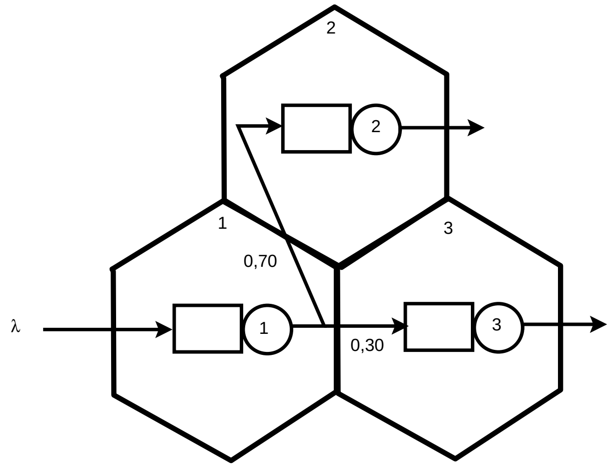


Figura 3.6: Três células em topologia divisão

As estatísticas descritivas da variável *tempo entre partidas* são mostradas na Tabela 3.2. Novamente, sob taxas de chegada de até 4.000 veículos/h, os modelos exponenciais parecem ser aplicáveis, com médias e desvios-padrões semelhantes. Para atestar visualmente estas conclusões, as Figura 3.7 e 3.8 mostram os histogramas para a variável *tempo entre partidas* com $\lambda = 1.000$ e 4.000, com os respectivos modelos exponenciais ajustados. Novamente foram feitos testes de Kolmogorov-Smirnov, apresentados nas Tabelas C.4 e C.3, que fortalecem a nossa conclusão de que os modelos exponenciais são aplicáveis sob taxas de chegadas iguais ou inferiores a 4.000 veículos/h.

As Figuras 3.9 e 3.10 apresentam o comportamento da variável *tempo de serviço nas células* (ou seja, os *tempos de permanência nas células*) e os histogramas para $\lambda = 1.000$ e 4.000 veículos/h, respectivamente. Para $\lambda = 4.000$, as saídas das redes de filas $M/G/c/c$ dependente do estado surpreendentemente indicam que a variável aleatória *tempos de permanência nas células* tem uma variabilidade muito baixa. A conclusão importante que pode ser retirada a partir destes resultados de simulação é que será necessário um cuidado

Tabela 3.2: Descritivas da média do tempo entre partidas para a topologia divisão

λ	Célula	Min.	Q1	Mediana	Média	DP	Q3	Máx.
1.000	1	0,00	1,06	2,52	3,50	3,44	4,80	32,28
	2	0,00	1,40	3,31	4,86	4,86	6,71	36,74
	3	0,01	3,66	8,20	12,47	12,19	17,60	78,89
2.000	1	0,00	0,51	1,23	1,77	1,76	2,46	17,44
	2	0,00	0,70	1,70	2,49	2,47	3,58	21,13
	3	0,00	1,72	4,12	6,17	6,18	8,70	47,77
4.000	1	0,00	0,37	0,88	1,27	1,25	1,78	9,85
	2	0,00	0,52	1,25	1,79	1,79	2,50	17,15
	3	0,01	1,26	3,00	4,29	4,27	5,86	31,08
8.000	1	0,00	0,28	0,68	1,26	3,56	1,42	57,20
	2	0,00	0,40	0,99	1,79	3,43	2,12	42,35
	3	0,00	0,97	2,44	4,34	6,97	5,16	66,74
16.000	1	0,00	0,15	0,35	1,27	10,71	0,70	159,62
	2	0,00	0,16	0,45	1,82	9,69	1,24	124,89
	3	0,00	0,43	1,14	4,25	18,16	2,59	153,37

extra no ajuste de alguma distribuição de probabilidade para esta variável aleatória. O congestionamento que uma determinada taxa de chegada pode causar no trecho de rodovia dificilmente seria determinado sem a utilização de uma ferramenta de simulação, como a utilizada aqui, ou sem o uso de algum outro modelo analítico.

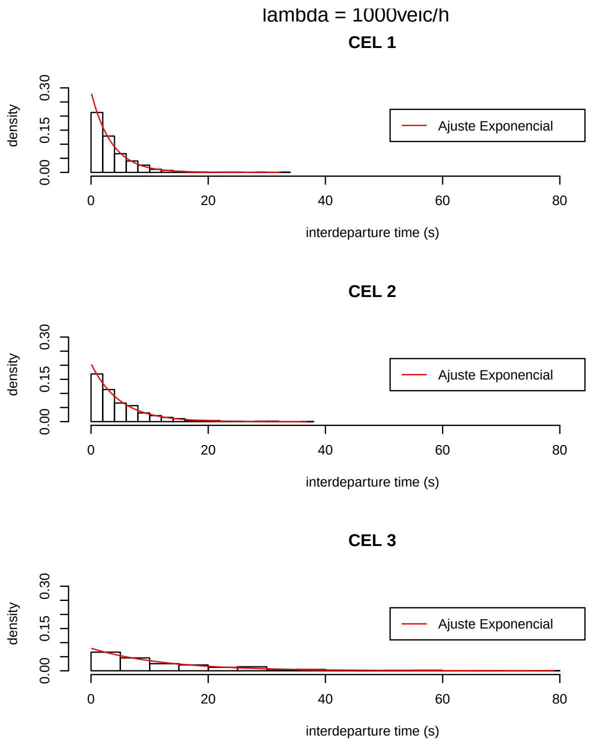


Figura 3.7: Tempo entre partidas na topologia divisão para $\lambda = 1.000$

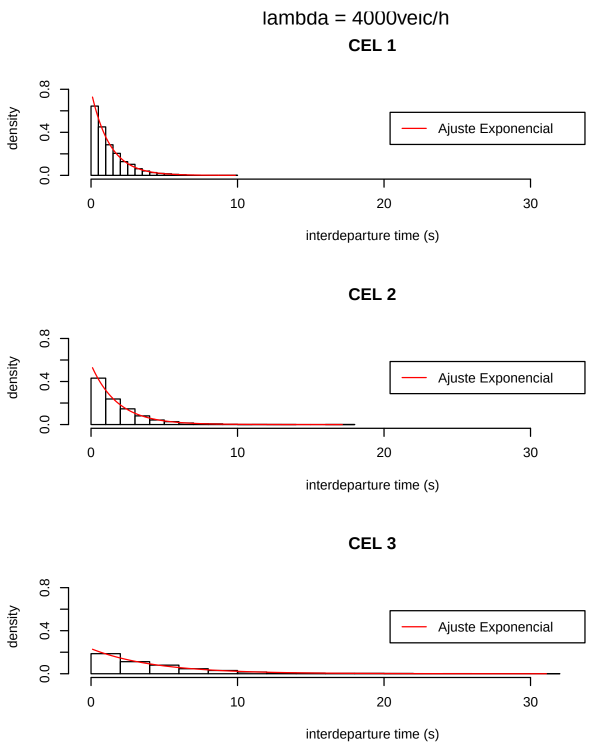


Figura 3.8: Tempo entre partidas na topologia divisão para $\lambda = 4.000$

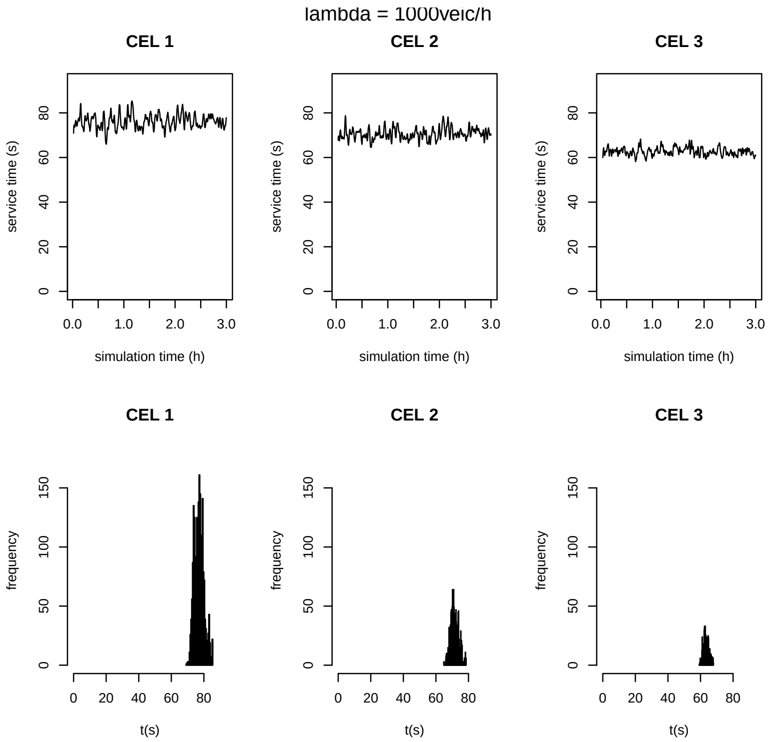


Figura 3.9: Tempos de serviço na topologia divisão para $\lambda = 1.000$

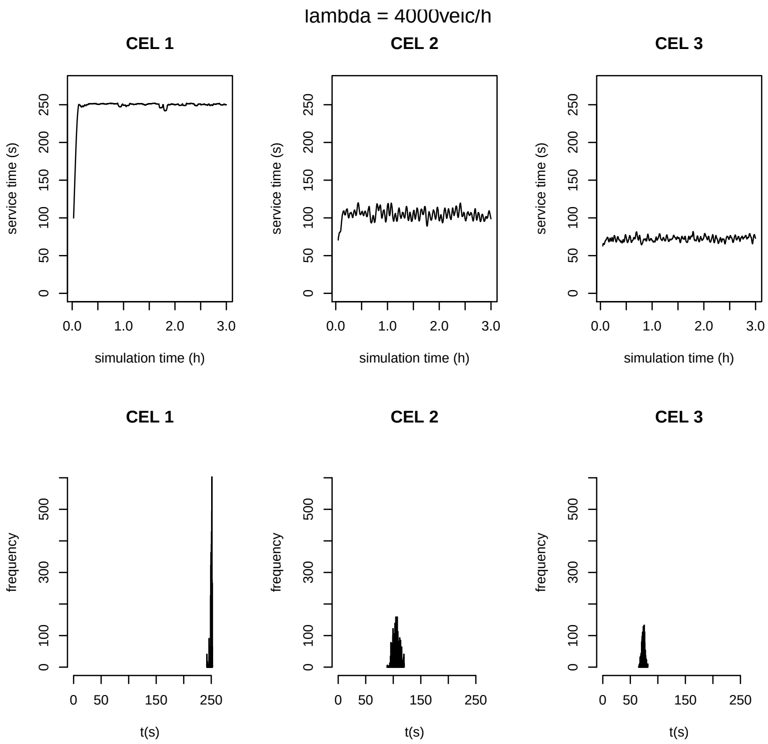


Figura 3.10: Tempos de serviço na topologia divisão para $\lambda = 4.000$

3.4 Topologia fusão

A topologia fusão, Figura 3.11, foi considerada apenas para atestar a simetria dos resultados. Na Figura 3.11, mostramos um sistema de telefonia celular em uma configuração simplificada na topologia fusão. Neste caso, cada fila modela um trecho de rodovia de 1 quilômetro de comprimento por uma pista de largura. O fluxo dos trechos 1 e 2 se fundem em um único trecho, 3, tendo uma proporção de chegada 70λ e 30λ, para os nós 1 e 2, respectivamente.

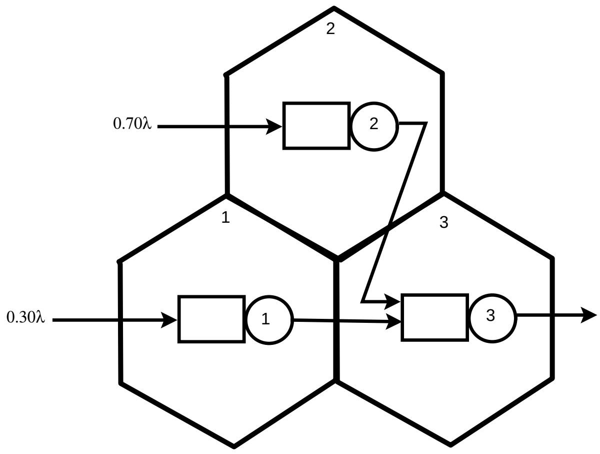


Figura 3.11: Três células em topologia fusão

Vê-se facilmente pela Tabela 3.3 e pelas Figuras 3.12 e 3.13, 3.14 e 3.15, que alguma simetria está realmente presente. Em outras palavras, as células 3 e 1 da topologia fusão comportam-se de maneira semelhante às células 1 e 3 da topologia divisão. Esse comportamento era esperado e é uma indicação de que o modelo de simulação pode estar correto. Porém, se ocorrer congestionamento nas células de entrada haverá então um hiper-congestionamento na célula 3. Outra vez, foram feitos testes de Kolmogorov-Smirnov, apresentados nas Tabelas C.6 e C.5, que fortalecem a nossa conclusão de que os modelos exponenciais são aplicáveis sob taxas de chegadas inferiores a 4.000 veículos/h.

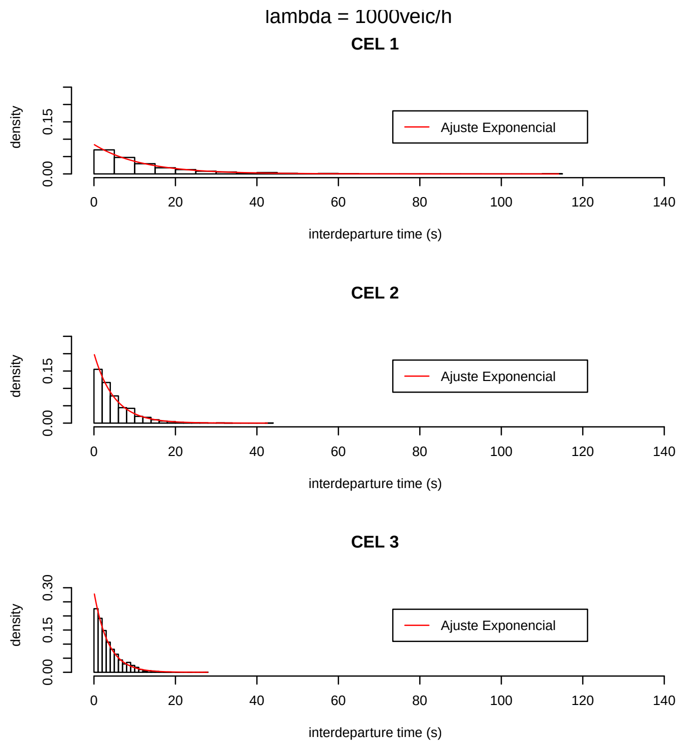


Figura 3.12: Tempo entre partidas na topologia fusão para $\lambda = 1.000$

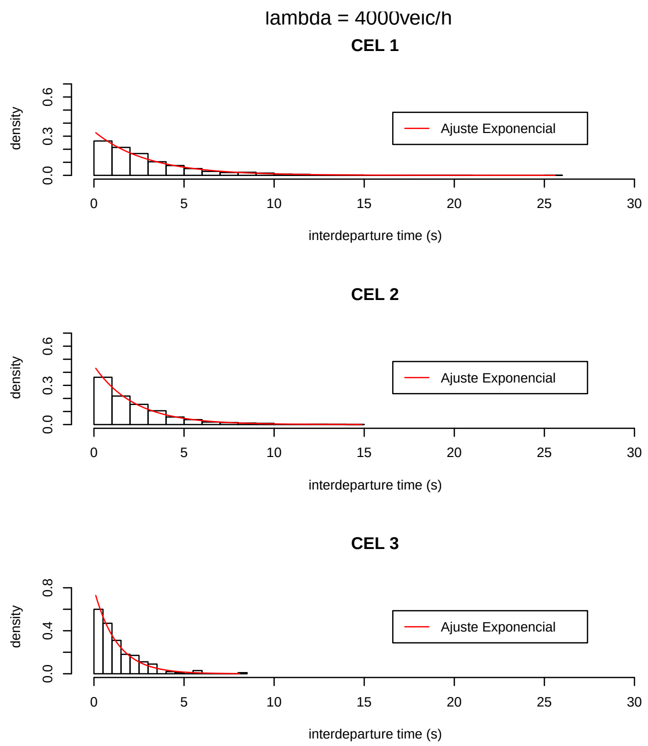


Figura 3.13: Tempo entre partidas na topologia fusão para $\lambda = 4.000$

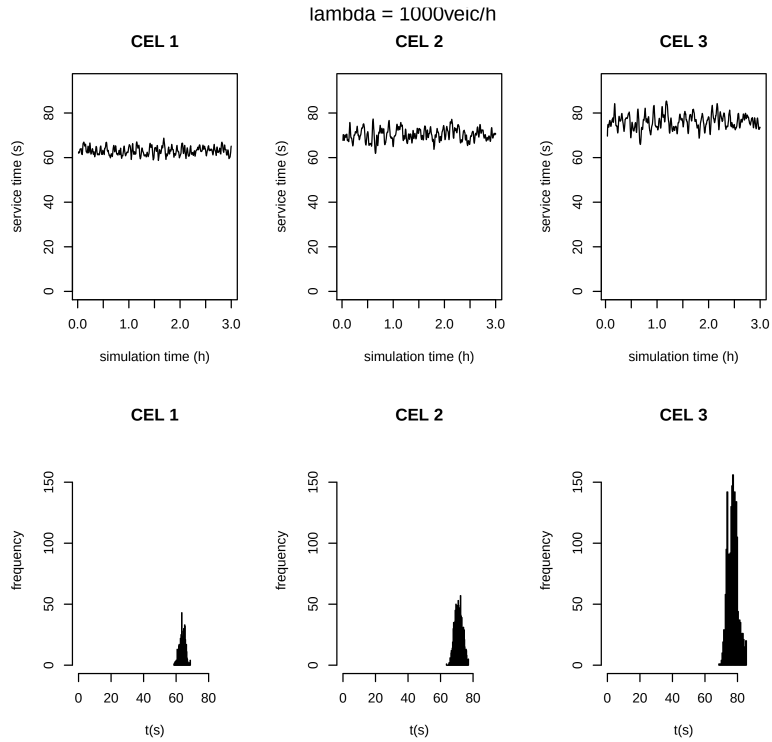


Figura 3.14: Tempos de serviço na topologia fusão para $\lambda = 1.000$

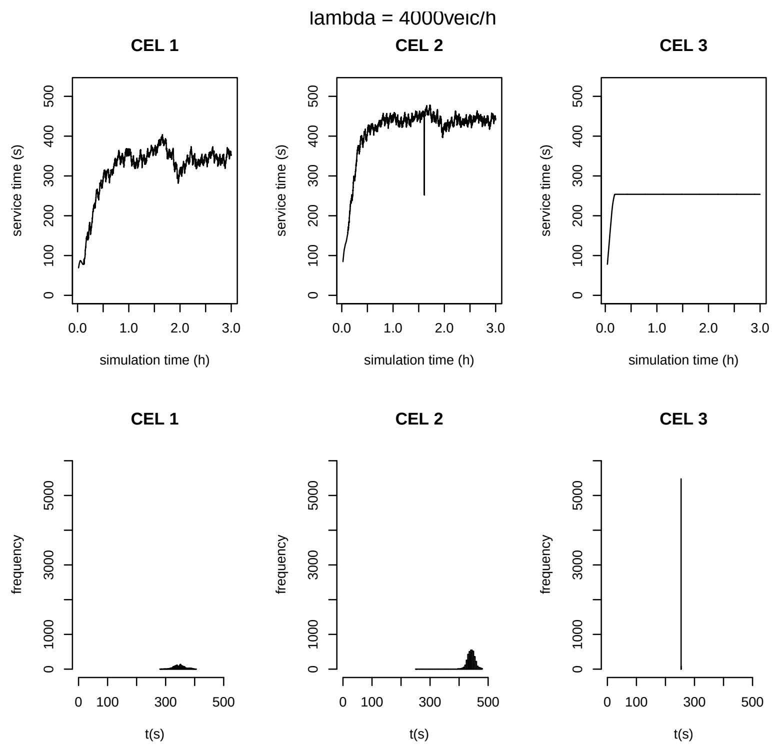


Figura 3.15: Tempos de serviço na topologia fusão para $\lambda = 4.000$

Tabela 3.3: Descritivas da média do tempo entre partidas para a topologia fusão

λ	Célula	Min.	Q1	Mediana	Média	DP	Q3	Máx.
1.000	1	0,00	3,39	8,27	11,77	12,02	16,13	114,07
	2	0,00	1,57	3,57	4,97	4,80	6,87	42,57
	3	0,00	1,12	2,53	3,50	3,24	4,95	27,91
2.000	1	0,00	1,64	3,79	5,79	6,13	7,98	59,49
	2	0,00	0,76	1,79	2,55	2,46	3,53	19,07
	3	0,00	0,52	1,24	1,78	1,75	2,43	13,01
4.000	1	0,00	0,95	2,12	2,96	2,87	4,04	25,59
	2	0,00	0,62	1,59	2,22	2,18	3,10	14,87
	3	0,00	0,38	0,94	1,27	1,23	1,83	8,08
8.000	1	0,00	0,55	1,69	2,54	2,63	3,69	18,91
	2	0,00	0,48	1,07	2,56	4,66	2,87	53,19
	3	0,00	0,33	0,78	1,27	1,39	1,72	8,31
16.000	1	0,00	0,31	0,99	2,56	5,32	2,41	62,43
	2	0,00	0,37	1,00	2,57	7,58	2,49	100,76
	3	0,00	0,26	0,62	1,28	2,79	1,24	31,62

3.5 Topologia mista

Finalmente, foi considerada uma topologia mista. A primeira configuração testada é mostrada na Figura 3.16, juntamente com as probabilidades de roteamento e as taxas de chegada, que se divide em $0,30\lambda$ no nó 1 e $0,70\lambda$ no nó 3. Esta topologia um pouco mais complexa foi escolhida para mostrar que o modelo de simulação é capaz de lidar com casos mais gerais, e não apenas com as configurações básicas mais simples. No entanto, deve-se ter em mente que, uma vez que o modelo é baseado na simulação intensiva, o tamanho das instâncias tratáveis pode ser bastante reduzido. Os tempos de simulação podem ser proibitivos para instâncias de grande porte, mas podemos lançar mão de técnicas de decomposição e agregação para reduzir o tamanho de instâncias reais e torná-las tratáveis pelo modelo de simulação.

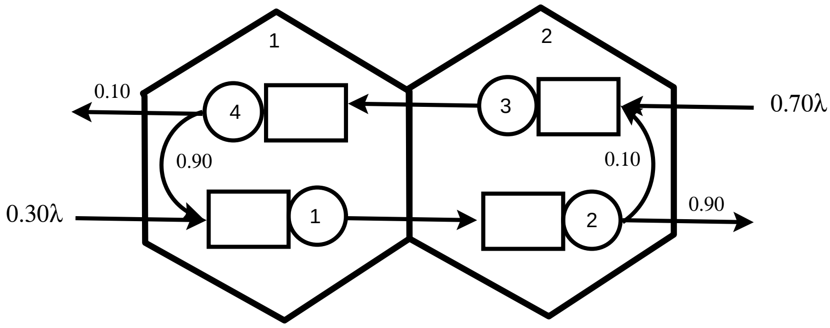


Figura 3.16: Duas células em topologia mista: topologia mista I

Nesta topologia, cada célula é composta de dois trechos de rodovia ao invés de um. Cada fila representa um trecho de 1 quilômetro de comprimento por uma pista de largura. Nesta primeira configuração nos nós onde ocorre a divisão, nó 2 e nó 4, as probabilidades, para a entidade que permanece na rede, são 0,10 e 0,90, respectivamente. A partir da Tabela 3.4 e das Figuras 3.17 e 3.18, vemos que o modelo exponencial é bastante aceitável para a variável *tempo entre partidas*, se a taxa de chegada não é tão elevada a ponto de saturar o sistema (neste caso, a saturação parece ocorrer quando $\lambda \geq 4.000$ veículos/h), conforme os testes de Kolmogorov-Smirnov realizados e apresentados nas Tabelas C.8 e C.7, que fortalecem essa nossa conclusão de que os modelos exponenciais são aplicáveis sob taxas de chegadas inferiores a 4.000 veículos/h.

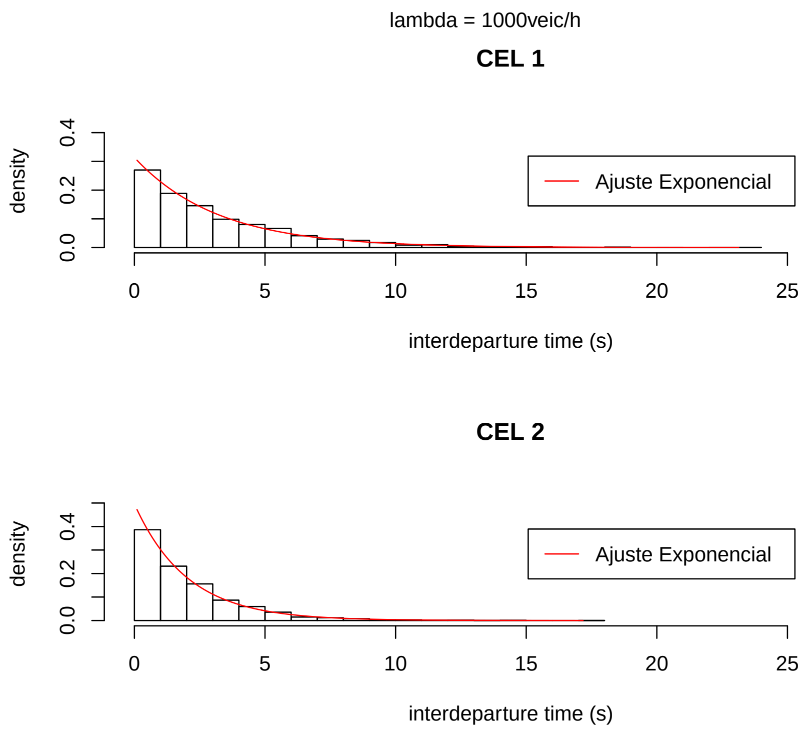


Figura 3.17: Tempo entre partidas na topologia mista I para $\lambda = 1.000$

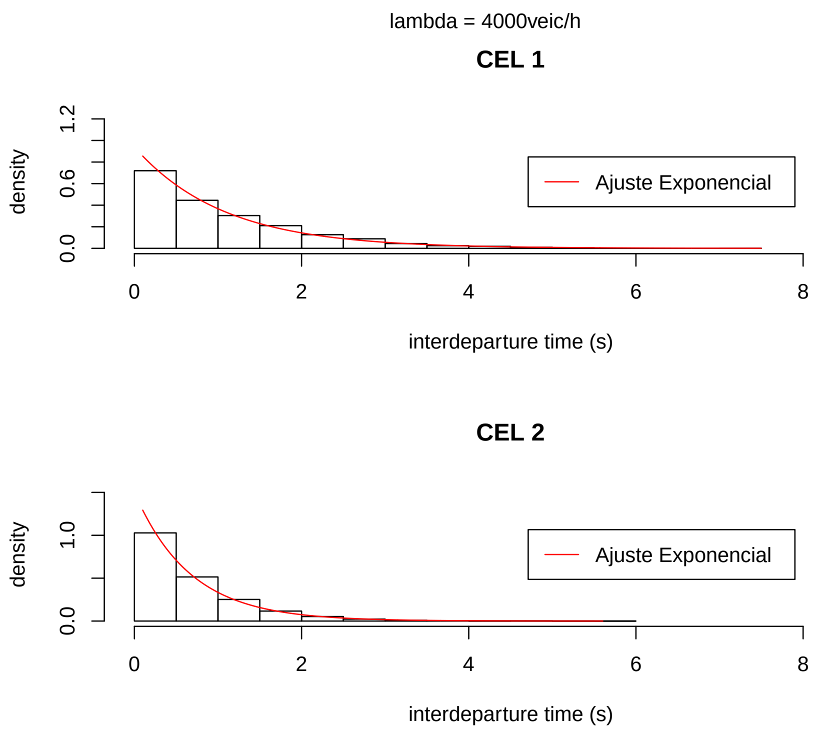


Figura 3.18: Tempo entre partidas na topologia mista I para $\lambda = 4.000$

Tabela 3.4: Descritivas da média do tempo entre partidas para a topologia mista I

λ	Célula	Min.	Q1	Mediana	Média	DP	Q3	Máx.
1.000	1	0,00	0,93	2,25	3,19	3,12	4,51	23,14
	2	0,00	0,59	1,41	2,02	1,98	2,81	17,18
2.000	1	0,00	0,48	1,13	1,61	1,59	2,21	14,56
	2	0,00	0,31	0,73	1,04	1,01	1,45	8,13
4.000	1	0,00	0,28	0,79	1,06	1,02	1,58	7,50
	2	0,00	0,20	0,48	0,67	0,63	0,94	5,60
8.000	1	0,00	0,24	0,78	1,03	1,03	1,46	6,63
	2	0,00	0,18	0,47	0,65	0,63	0,96	5,25
16.000	1	0,00	0,23	0,71	1,04	1,08	1,48	6,32
	2	0,00	0,18	0,45	0,64	0,66	0,90	6,05

É quando se analisa a variável *tempo de serviço nas células* que o resultado mais curioso aparece. Neste caso, surgem as distribuições bimodais, como visto nas Figuras 3.19 e 3.20. A partir desses resultados, é evidente que muitas vezes será necessário utilizar misturas de distribuições, como em Everitt & Hand (1981), para adequadamente descrever a variável aleatória *tempo de serviço nas células*, em lugar de usar uma distribuição única.

Para podermos tentar entender melhor esse fenômeno de bimodalidade, testamos outras configurações com a mesma topologia mista, mudando apenas a probabilidade nos nós onde ocorre a divisão, nós 2 e 4. Para a segunda configuração, que chamaremos de Mista II apresentada na Figura 3.21(a), as probabilidades de permanência na rede são 0,90 e 0,10, para os nós 2 e 4 respectivamente. Para a terceira configuração, Mista III - Figura 3.21(a), as probabilidades de permanência na rede são 0,30 e 0,10, para os nós 2 e 4 respectivamente. Já para a quarta configuração, Mista IV - Figura 3.21(c), as probabilidades de permanência na rede são 0,10 e 0,30, para os nós 2 e 4 respectivamente.

Analisando a Tabela 3.5 e as Figuras 3.22 e 3.23, observamos que, parece que o limite para a distribuição exponencial para o *tempo entre partidas* acontece para $\lambda = 2.000$, após este valor parecia que a distribuição mais adequada seria uma hiperexponencial, porém utilizando o software *EasyFit* e realizando teste de Kolmogorov-Smirnov para diversas distribuições não conseguimos fazer o ajuste a nenhuma das distribuições testadas, ver Tabelas C.10 e C.9, outro fato que nos chamou a atenção é que para $\lambda = 4.000$ e $\lambda = 8.000$, apenas para a célula 1 o *tempo entre partidas* ainda é considerado exponencial. Uma outra informação interessante que podemos obter destes resultados é que para a célula 1

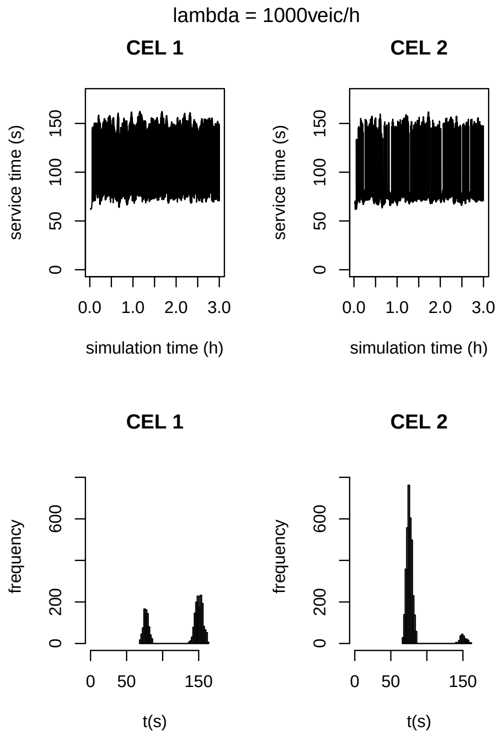


Figura 3.19: Tempos de serviço na topologia mista I para $\lambda = 1.000$

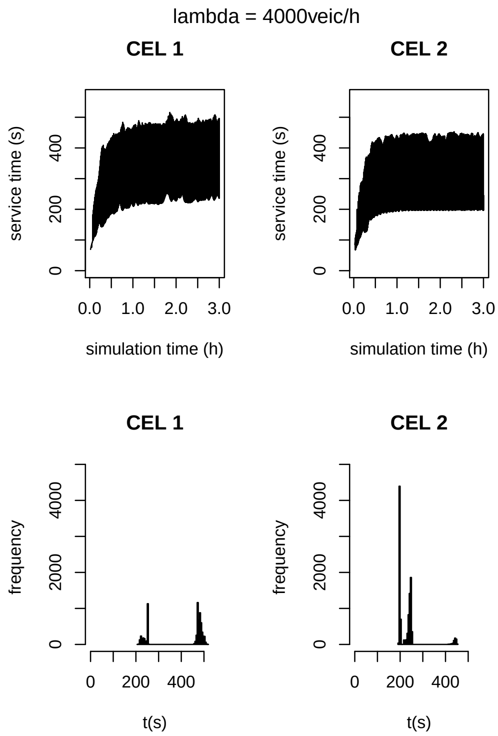
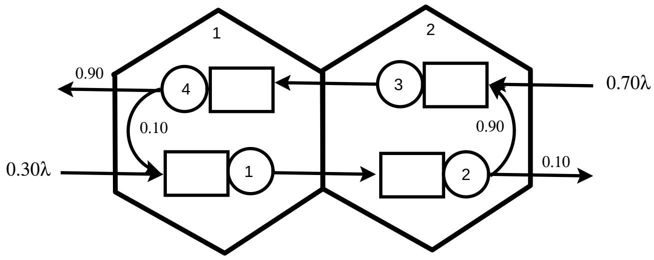
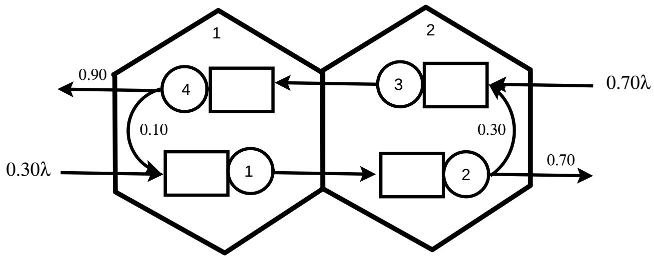


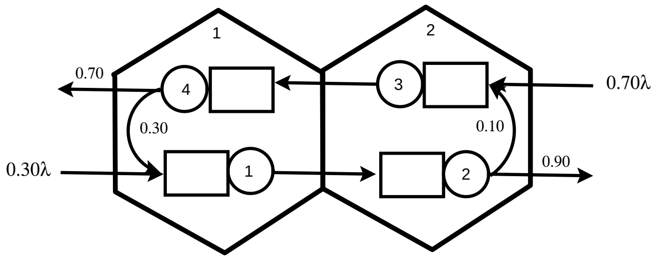
Figura 3.20: Tempos de serviço na topologia mista I para $\lambda = 4.000$



(a) Topologia mista II



(b) Topologia mista III



(c) Topologia mista IV

Figura 3.21: Mais configurações em topologia mista com duas células

a média da variável tempo entre partidas é menor que as médias obtidas para a primeira configuração da topologia mista testada, enquanto que para a célula 2 a média é maior. Este fato, ainda em suspeita, se deve a ocupação nos trechos da célula 2 ser maior do que na célula 1.

Quando analisamos as Figuras 3.24 e 3.25, que se referem ao tempo de serviço, podemos observar que para $\lambda = 1.000$ veículos/h a distribuição continua bimodal, enquanto que para $\lambda = 4.000$ já aparece uma distribuição trimodal, onde suspeitamos que a terceira moda seja as entidades que ficam circulando no sistema, que na prática poderíamos falar de um táxi ou uma lotação.

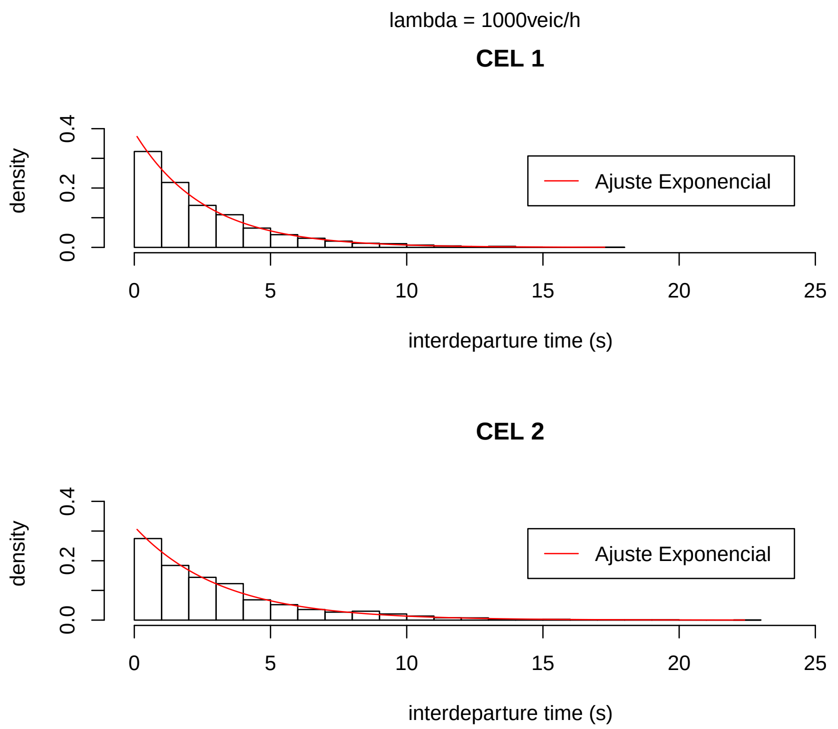


Figura 3.22: Tempo entre partidas na topologia mista II para $\lambda = 1.000$

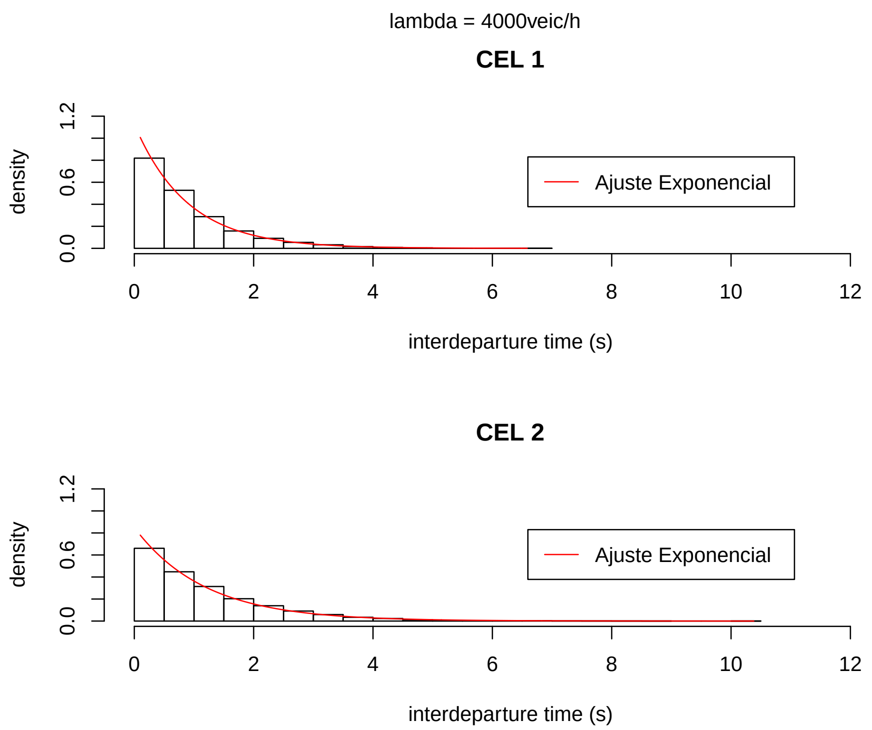


Figura 3.23: Tempo entre partidas na topologia mista II para $\lambda = 4.000$

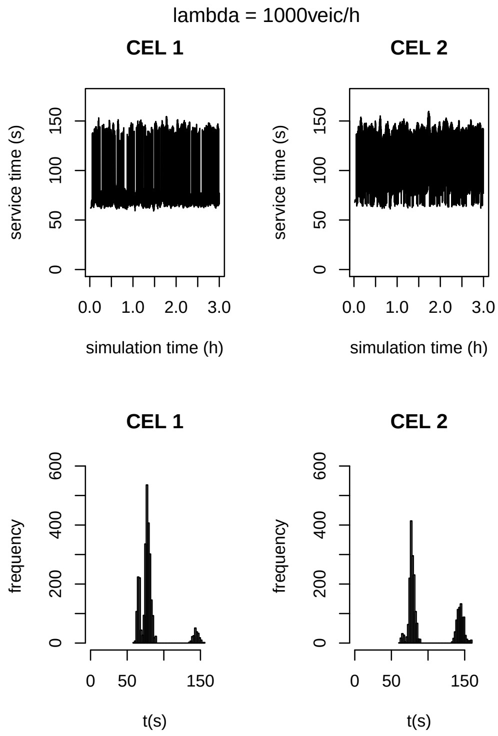


Figura 3.24: Tempos de serviço na topologia mista II para $\lambda = 1.000$

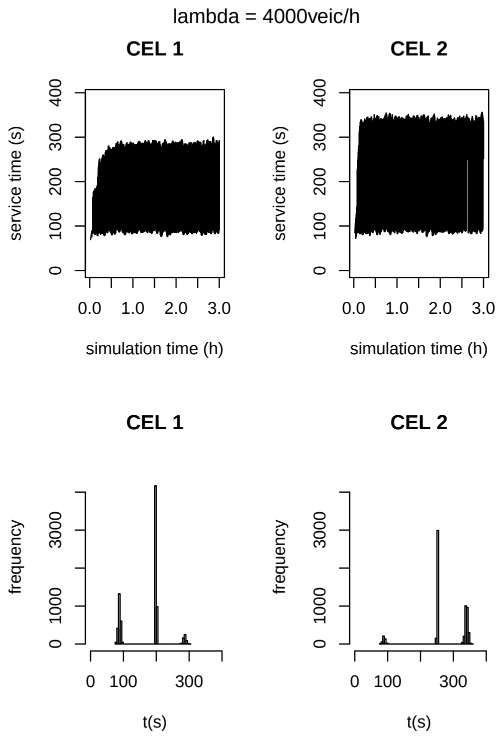


Figura 3.25: Tempos de serviço na topologia mista II para $\lambda = 4.000$

Tabela 3.5: Descritivas da média do tempo entre partidas para a topologia mista II

λ	Célula	Min.	Q1	Mediana	Média	DP	Q3	Máx.
1.000	1	0,00	0,77	1,77	2,57	2,53	3,61	17,27
	2	0,00	0,90	2,26	3,17	3,08	4,26	22,40
2.000	1	0,00	0,37	0,89	1,28	1,27	1,78	9,29
	2	0,00	0,46	1,13	1,59	1,56	2,19	17,29
4.000	1	0,00	0,27	0,65	0,89	0,84	1,23	6,58
	2	0,00	0,35	0,87	1,18	1,12	1,68	10,38
8.000	1	0,00	0,21	0,50	0,69	0,65	0,97	5,12
	2	0,00	0,26	0,76	1,01	0,95	1,52	6,00
16.000	1	0,00	0,13	0,33	0,66	3,37	0,66	72,96
	2	0,00	0,14	0,46	0,98	6,30	0,84	107,24

Partindo para a análise dos resultados obtidos para a configuração Mista III - Figura 3.21(b), vamos analisar a Tabela 3.6 e as Figuras 3.26 e 3.27, observamos que conforme as configurações anteriores, parece que o limite para a distribuição exponencial para o *tempo entre partidas* acontece para $\lambda = 4.000$. O que pode ser confirmado pelos testes de Kolmogorov-Smirnov apresentados nas Tabelas C.12 e C.11. Uma outra informação interessante que podemos obter destes resultados é que o comportamento da variável tempo entre partidas se assemelha com os resultados obtidos para a configuração Mista II. Outra observação que obtemos é que as médias são bastante parecidas entre as células 1 e 2, sendo que a média para a célula 2 é ligeiramente maior que a célula 1.

Quando analisamos as Figuras 3.28 e 3.29, que se referem ao tempo de serviço, podemos observar que para $\lambda = 1.000$ veículos/h a distribuição continua bimodal, enquanto que para $\lambda = 4.000$ já aparece uma distribuição trimodal, do mesmo modo, que ocorreu na configuração Mista II.

Tabela 3.6: Descritivas da média do tempo entre partidas para a topologia mista III

λ	Célula	Min.	Q1	Mediana	Média	DP	Q3	Máx.
1.000	1	0,00	0,92	2,26	3,13	3,09	4,28	23,00
	2	0,00	0,95	2,34	3,25	3,18	4,46	25,93
2.000	1	0,00	0,48	1,11	1,59	1,57	2,22	12,81
	2	0,00	0,49	1,16	1,65	1,60	2,27	12,13
4.000	1	0,00	0,25	0,62	0,89	0,89	1,24	9,27
	2	0,00	0,26	0,63	0,92	0,93	1,27	7,53
8.000	1	0,00	0,20	0,48	0,69	0,68	0,95	6,12
	2	0,00	0,17	0,47	0,71	0,76	1,01	6,40
16.000	1	0,00	0,12	0,31	0,67	3,21	0,64	82,59
	2	0,00	0,10	0,30	0,68	3,18	0,67	82,06

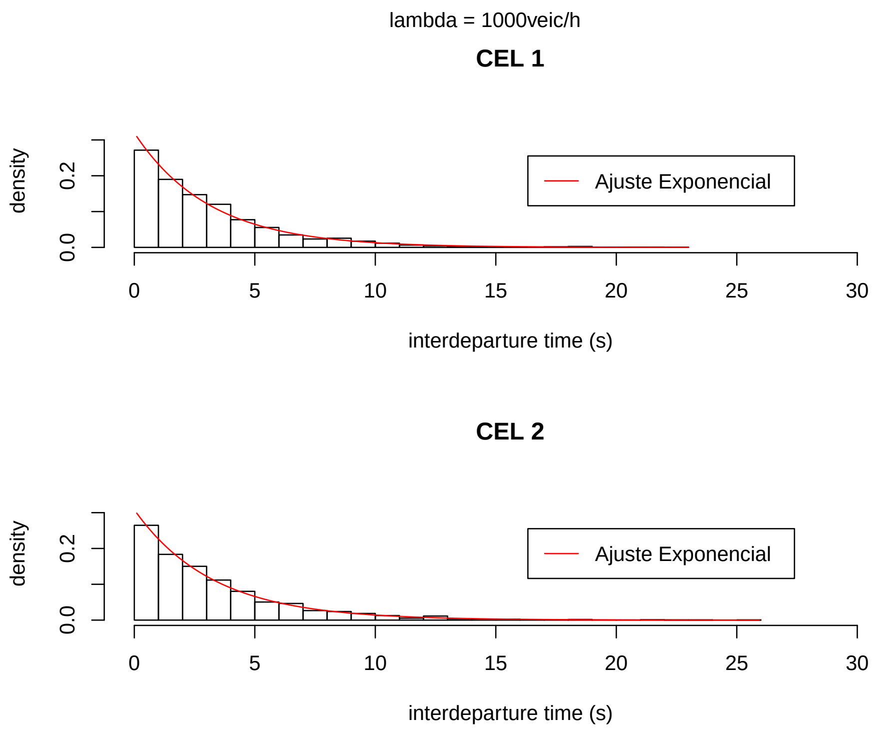


Figura 3.26: Tempo entre partidas na topologia mista III para $\lambda = 1.000$

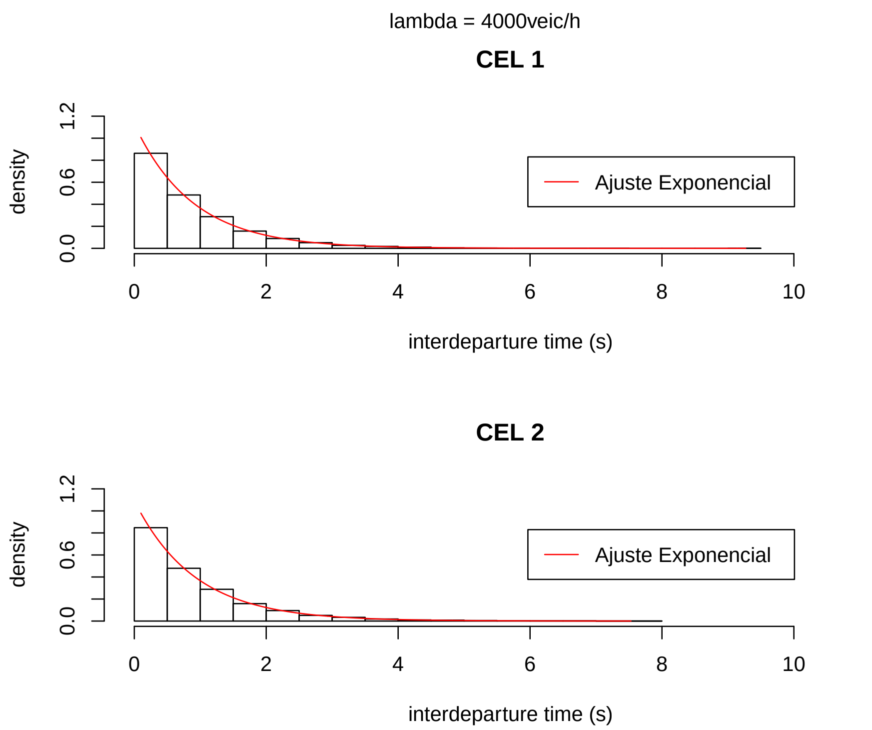


Figura 3.27: Tempo entre partidas na topologia mista III para $\lambda = 4.000$

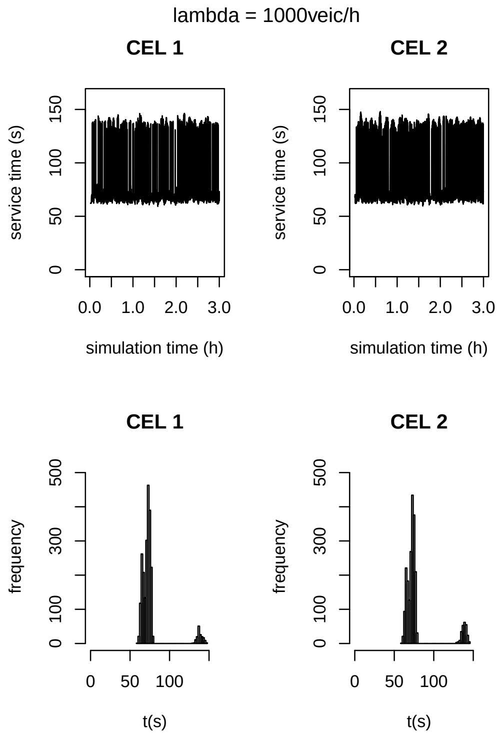


Figura 3.28: Tempos de serviço na topologia mista III para $\lambda = 1.000$

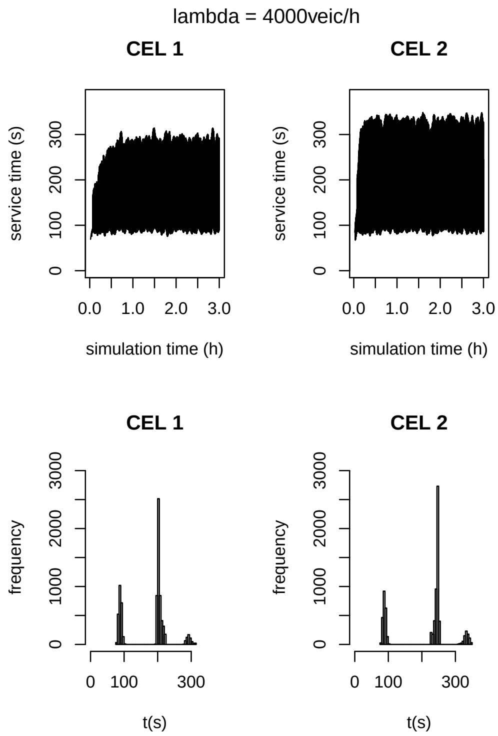


Figura 3.29: Tempos de serviço na topologia mista III para $\lambda = 4.000$

Analisando os resultados obtidos para a configuração Mista IV - Figura 3.21(c), temos a Tabela 3.7 e as Figuras 3.30 e 3.31, observamos que novamente, conforme as configurações anteriores, parece que o limite para a distribuição exponencial para o *tempo entre partidas* acontece para $\lambda = 4.000$, o que mais uma vez, pode ser corroborado com os testes de Kolmogorov-Smirnov, apresentados nas Tabelas C.14 e C.13. Uma outra informação interessante que podemos obter destes resultados é que o comportamento da variável tempo entre partidas se assemelha com os resultados obtidos para a configuração Mista I. Outra observação que obtemos é que as médias são bastante parecidas entre as células 1 e 2, sendo que a média para a célula 2 é ligeiramente menor que a célula 1.

Quando analisamos as Figuras 3.32 e 3.33, que se referem ao tempo de serviço, podemos observar que para $\lambda = 1.000$ veículos/h a distribuição continua bimodal, enquanto que para $\lambda = 4.000$ já aparece uma distribuição trimodal, do mesmo modo, que aconteceu na configuração da Figura 3.16.

Tabela 3.7: Descritivas da média do tempo entre partidas para a topologia mista IV

λ	Célula	Min.	Q1	Mediana	Média	DP	Q3	Máx.
1.000	1	0,00	0,99	2,31	3,32	3,27	4,48	26,00
	2	0,00	0,82	2,05	2,83	2,74	3,97	24,22
2.000	1	0,00	0,48	1,14	1,68	1,67	2,36	13,07
	2	0,00	0,43	1,02	1,45	1,43	2,04	12,63
4.000	1	0,00	0,25	0,61	0,89	0,89	1,22	8,31
	2	0,00	0,22	0,53	0,77	0,76	1,06	9,33
8.000	1	0,00	0,20	0,51	0,73	0,74	1,02	7,93
	2	0,00	0,17	0,45	0,65	0,67	0,91	7,01
16.000	1	0,00	0,13	0,38	0,71	1,17	0,84	19,73
	2	0,00	0,13	0,35	0,65	1,08	0,75	19,73

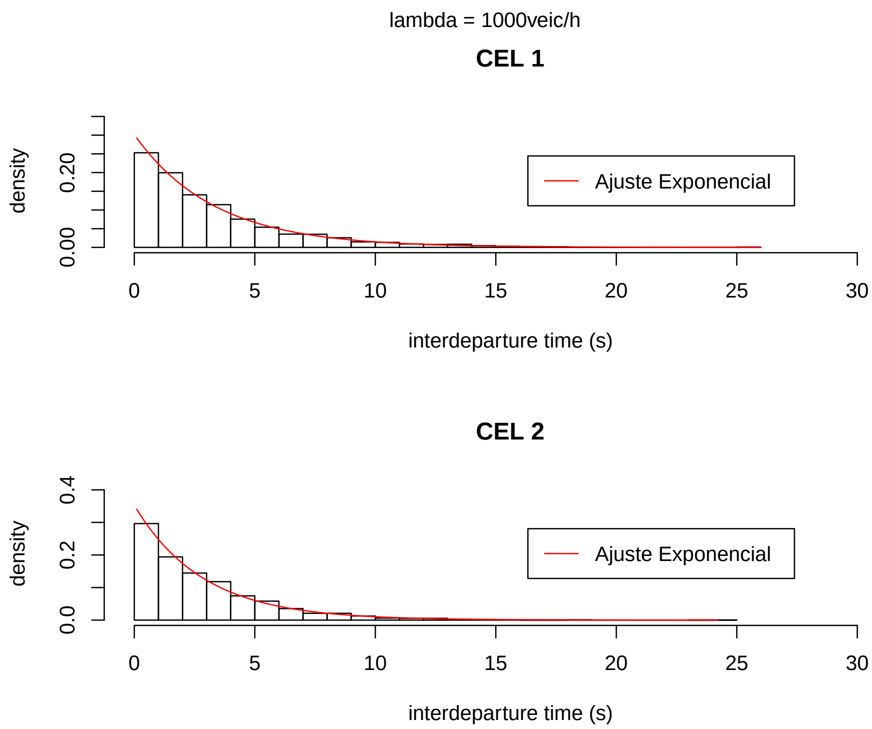


Figura 3.30: Tempo entre partidas na topologia mista IV para $\lambda = 1.000$

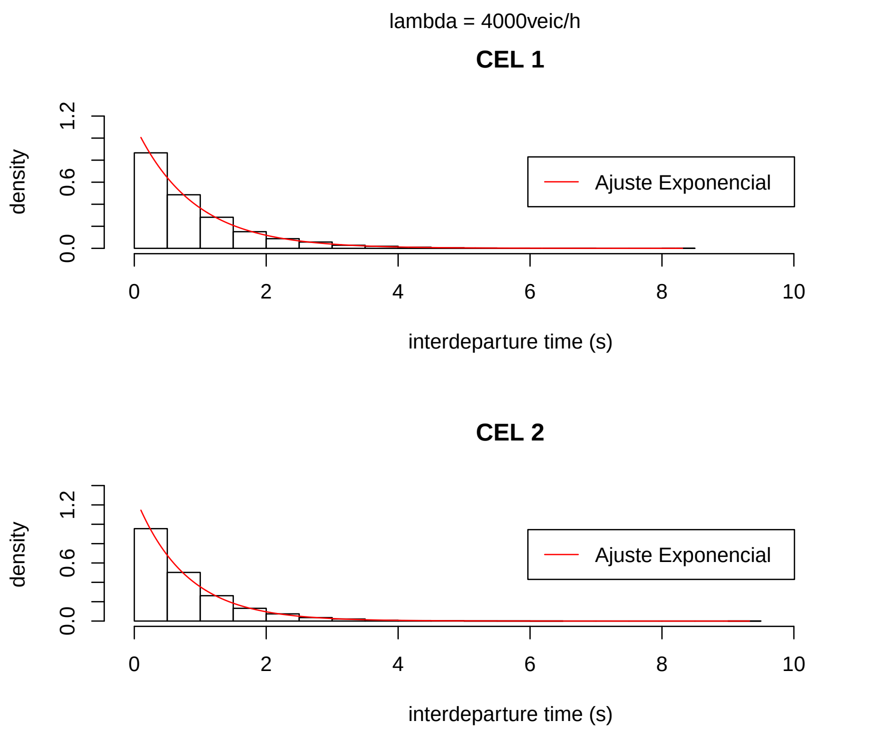


Figura 3.31: Tempo entre partidas na topologia mista IV para $\lambda = 4.000$

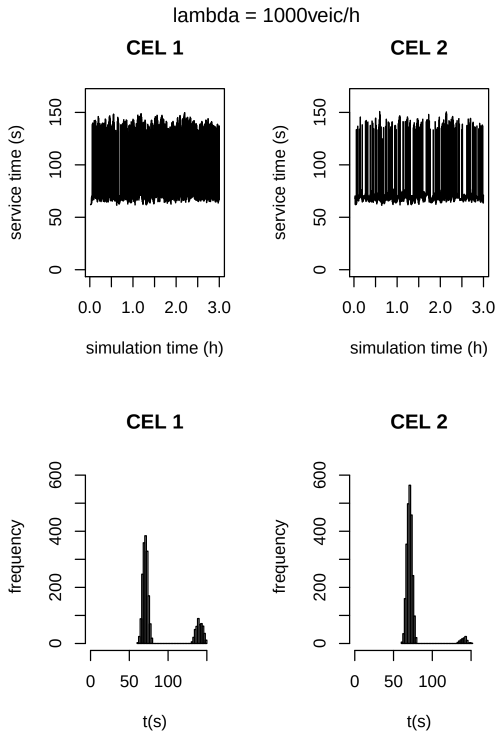


Figura 3.32: Tempos de serviço na topologia mista IV para $\lambda = 1.000$

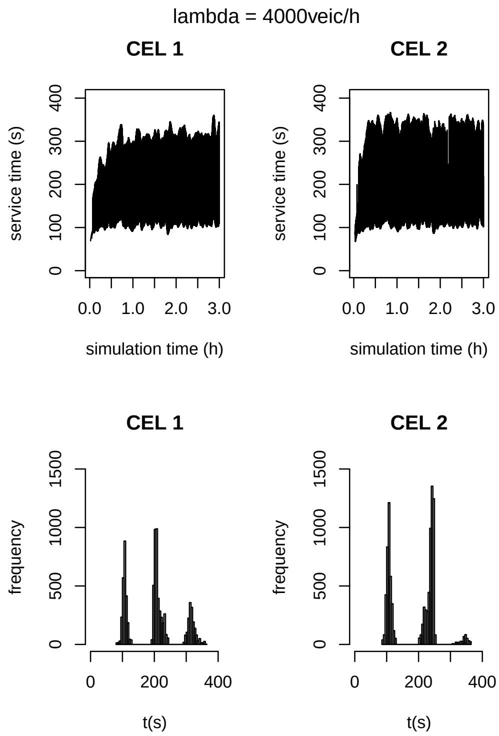


Figura 3.33: Tempos de serviço na topologia mista IV para $\lambda = 4.000$

Capítulo 4
Conclusões e Observações Finais

Uma nova abordagem foi proposta para modelar a mobilidade em redes de telefonia celular, com base em filas finitas com serviços gerais dependentes do estado configuradas em redes. Anteriormente, tal modelo estocástico foi aplicado com êxito a problemas de tráfego de veículos e de pedestres, bem como na modelagem de sistema de manufatura. Basicamente, esta nova abordagem não usa nada de conceitualmente novo em relação ao que se tem na modelagem de tráfego de veículos, uma vez que apenas considera um efeito de redução da velocidade média (taxa de serviço), com o aumento da densidade de usuários no sistema. No entanto, não é de nosso conhecimento que tal conceito chave, fortemente intuitivo, tenha sido utilizado explicitamente na modelagem de usuários em redes de telefonia celular. Com o objetivo de enfatizar o impacto da inclusão da dependência do estado sobre os modelos de mobilidade, apresentamos um modelo de simulação a eventos discretos recém-desenvolvido e destacamos algumas das medidas de desempenho.

Dentre as principais conclusões obtidas a partir das simulações, podemos constatar que, contrariamente à crença de alguns pesquisadores, o processo de chegada às células *podem não aderir* a um processo markoviano, se o sistema estiver sob carga pesada(alta taxa de chegada), como acontece muitas vezes nas grandes cidades. Além disso, os estudos de simulação confirmaram que a variável *tempo de permanência nas células* muitas vezes não é nem aproximadamente exponencial. Além disso, podemos concluir que pode ser perigoso considerar qualquer outra distribuição de probabilidade, sem uma avaliação cuidadosa do estado de congestionamento da célula. Finalmente, em topologias complexas, podemos até mesmo encontrar distribuições de probabilidade multi-modal para a variável aleatória *tempo de permanência nas células.*

4.1 Tópicos para trabalhos futuros

Algumas perguntas foram respondidas por esta pesquisa, mas os resultados apresentados dão origem a muitas outras questões. Em primeiro lugar, não se sabe qual o tamanho máximo das instâncias tratáveis pelo modelo de simulação apresentado. Não esperamos que este tamanho seja muito grande, uma vez que simulações tendem a ser muito demoradas. Muito embora alguém possa argumentar que é possível lançar mão de técnicas de decomposição e agregação, como forma de aproximar sistemas mais complexos, seria realmente de grande utilidade o desenvolvimento de modelos analíticos *ad-hoc*. Estivemos interessados aqui apenas na influência dos modelos estocásticos dependentes do estado sobre uma única variável aleatória (*tempo de permanência nas células*). Na verdade, a proposta de utilização de modelos de filas finitas dependentes do estado na modelagem de redes de telefonia móvel é apenas o começo da história. Uma vez que isto tenha sido estabelecido, é importante considerar os efeitos, por exemplo, no tráfego de chamadas, no número de *handoffs* (transferências entre centrais), na duração das chamadas e no desempenho global do sistema, os quais são completamente desconhecidos. Talvez o primeiro passo, seria tentar modelar a ocupação da célula e relacionar este número com o tráfego de chamadas. Além disso, outra área interessante da pesquisa é a de alocação de capacidade. Resultados promissores foram relatados para modelos de redes de filas finitas em sistemas de manufatura (Cruz et al., 2008), nos quais uma capacidade mínima deveria ser definida de forma que fosse assegurada uma certa qualidade de serviço (descrita em termos de baixas probabilidades de bloqueio).

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.

Alencar, B. G. S. & Sadok, D. F. H. (1999), Um modelo de mobilidade de usuários para redes móveis, *in* 'Proceedings of 'I Workshop de Comunicação Sem Fio'', Belo Horizonte, MG, Brazil, pp. 57–66.

Alfa, A. S. & Liu, B. (2004), 'Performance analysis of a mobile communication network: The tandem case', *Computer Communications* **27**(3), 208–221.

Anatel (2009), 'Frota circulante brasileira', São Paulo, SP. Disponível em <http://www.anatel.gov.br>. Acesso em: 24 jul. 2010.

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

Cavalcanti, D. A. T., Dias, K. L. & Sadok, D. (2000), Estudo dos aspectos de QoS e mobilidade no planejamento de uma rede móvel celular, *in* 'Proceedings of 'I Workshop do Projeto SIDAM'', IME-USP, São Paulo, SP, Brazil, pp. 125–140.

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.

Cheah, J. & Smith, J. M. (1994), 'Generalized $M/G/C/C$ state dependent queueing models and pedestrian traffic flows', *Queueing Systems* **15**, 365–386.

Cruz, F. R. B., Duarte, A. R. & van Woensel, T. (2008), 'Buffer allocation in general single-server queueing network', *Computers & Operations Research* **35**(11), 3581–3598.

Cruz, F. R. B., Oliveira, P. C. & Duczmal, L. (2010), 'State-dependent stochastic mobility model in mobile communication networks', *Simulation Modelling Practice and Theory* **18**(3), 348–365.

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., Smith, J. M. & Medeiros, R. O. (2005), 'An $M/G/C/C$ state dependent network simulation model', *Computers & Operations Research* **32**(4), 919–941.

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.

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.

Evans, J. S. (1995), Traffic modelling of cellular mobile communications, Master's thesis, University of Melbourne, Melbourne.

Everitt, B. S. & Hand, D. J. (1981), *Finite mixture distributions*, London: Chapman and Hall.

García-Ródenas, R., López-García, M. L., Niño-Arbelaez, A. & Verastegui-Rayo, D. (2006), 'A continuous whole-link travel time model with occupancy constraint', *European Journal of Operational Research* **175**(3), 1455–1471.

Ghatee, M. & Hashemi, S. M. (2009), 'Traffic assignment model with fuzzy level of travel demand: An efficient algorithm based on quasi-Logit formulas', *European Journal of Operational Research* **194**(2), 432–451.

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

Han, K. (2002), 'Simulation studies of the effects of user mobility on the handoff performance of mobility communications', *Simulation Modelling Practice & Theory* **10**, 497–512.

Hegde, N. & Sohraby, K. (2002), 'On the impact of soft handoff in cellular systems', *Computer Networks* **38**(2), 257–271.

Helbing, D., Schönhof, M., Stark, H.-U. & Holyst, J. A. (2005), 'How individuals learn to take turns: Emergence of alternating cooperation in a congestion game and the prisoner's dilemma', *Advances in Complex Systems* **8**(1), 87–116.

Hong, D. & Rappaport, S. S. (1986), 'Traffic model and performance analysis for cellular mobil radio telephone systems with prioritized and nonprioritized handoff procedures', *IEEE Transactions on Vehicular Technology* **VT-35**(3), 77–92.

IBGE (2007), 'Contagem da população', Rio de Janeiro, RJ. Disponível em <http://www.ibge.gov.br>. Acesso em: 29 jul. 2010.

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.

Kimber, R. M. & Hollis, E. M. (1979), Traffic queues and delays at road junctions, Technical Report Laboratory Report 909, Transport and Road Research Laboratory, Crowthorne, UK.

Lam, D., Cox, D. C. & Widom, J. (1997), 'Teletraffic modeling for personal communications services', *IEEE Communications Magazine* **35**(2), 79–87.

Lam, D., Jannink, J., Cox, D. C. & Widom, J. (1996), Modeling location management in personal communications services, *in* 'Proceedings of 'IEEE Internacional Conference on Universal Personal Communications'', Vol. 2, Cambridge, MA, pp. 596–601.

Manner, J., Toledo, A. L., Mihailovic, A., Muñoz, H. L. V., Hepworth, E. & Khouaja, Y. (2002), 'Evaluation of mobility and quality of service interaction', *Computer Networks* **38**(2), 137–163.

Markoulidakis, J. G., Lyberopoulos, G. L. & Anagnostou, M. E. (1998), 'Traffic model for third generation cellular mobile telecommunications systems', *Wireless Networks* **4**, 389–400.

Markoulidakis, J. G., Lyberopoulos, G. L., Tsirkas, D. F. & Sykas, E. D. (1997), 'Mobility modeling in third-generation mobile telecommunications systems', *IEEE Personal Communications* **4**(4), 41–56.

McMillan, D. W. (1993), Traffic Modelling and Analysis for Cellular Mobile Networks, Ph.D. thesis, University of Melbourne, Melbourne, Australia.

Nanda, S. (1993), 'Teletraffic models for urban and suburban microcells: Cell sizes and handoff rates', *IEEE Transactions on Vehicular Technology* **42**(2), 673–682.

Oliveira, P. C. (2005), Modelos de mobilidade estocásticos dependentes do estado em redes de telefonia móvel, Dissertação de mestrado, Departamento de Ciência da Computação - ICEx - UFMG, Belo Horizonte - MG.

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.

Pursals, S. C. & Garzón, F. G. (2009), 'Optimal building evacuation time considering evacuation routes', *European Journal of Operational Research* **192**(2), 692–699.

Robinson, S. (2007), 'A statistical process control approach to selecting a warm-up period for a discrete-event simulation', *European Journal of Operational Research* **176**(1), 332–346.

Silva, S. L., Rocha, M. N. & Mateus, G. R. (2002), 'Simulation and analysis of a new mobility model for mobile communication networks', *Annals of Operation Research* **116**, 57–69.

Sindipeças (2009), 'Frota circulante brasileira', São Paulo, SP. Disponível em <http://www.sindipecas.org.br>. Acesso em: 24 jul. 2010.

Tomas, R., Gilbert, H. & Mazziotto, G. (1988), Influence of the moving of the mobile stations on the performance of a radio mobile cellular network, *in* 'Proceedings of 3rd Nordic Seminar on Digital Land Mobile Radio Communication', artigo n. 9.4, Copenhagen, Denmark.

Transportation Research Board (2000), Highway capacity manual, Technical report, 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.

van Woensel, T. & Cruz, F. R. B. (2009), 'A stochastic approach to traffic congestion costs', *Computers & Operations Research* **36**(6), 1731–1739.

van Woensel, T. & Vandaele, N. (2007), 'Modelling traffic flows with queueing models: A review', *Asia-Pacific Journal of Operational Research* **24**(4), 435–461.

van Woensel, T., Wuyts, B. & Vandaele, N. (2006), 'Validating state-dependent queueing models for uninterrupted traffic flows using simulation', *4OR* **4**(2), 159–174.

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

Zonoozi, M. M. & Dassanayake, P. (1997), 'User mobility modeling and characterization of mobility patterns', *IEEE Journal on Selected Areas in Communications* **15**(7), 1239–1252.

Apêndice A
Códigos em C++

Código A.1: Mgcc.c

```
//
// Purpose:
//   to implement programs concerning MGCC queues
//
// Authors:
//   Paula de Campos Oliveira
//   Frederico R. B. Cruz
//   Departamento de Estatistica
//   Universidade Federal de Minas Gerais
//   Brazil
//   E-mail: pcampol@yahoo.com.br,fcruz@est.ufmg.br
//
//
// Version: 5.0
//
// Date:   Jul/2010
//
#include <stdlib.h>
#include <stdio.h>
#include "mgccsim.cpp"
int main(int argc, char *argv[]) {
  // check input
  if (argc < 2) {
     fprintf(stderr, "Usage: %s <operation> [script_file]\n", argv[0]);
     fprintf(stderr, "\t operation 1 -> plots a linear service rate\n");
     fprintf(stderr, "\t operation 2 -> plots an exponential service rate\n");
     fprintf(stderr, "\t operation 3 -> simulates an MGcc system\n");
     exit(0);
  }
  // perform actions
  // plot linear service rate
  if (argv[1][0]=='1') {
      CMLinUsr ModelLin;
      double length = 8.0;  // corridor length
      double width = 2.5;   // corridor width
      ModelLin.SetCorridor(length,width);
      for (int i=1; i <= 100; i++ ) {
         fprintf(stdout, "%d %f\n", i, ModelLin.Rate(i));
      }
  }
  // plot exponential service rate
  else if (argv[1][0]=='2') {
      CMExpUsr ModelExp;
      double length = 8.0;  // corridor length
      double width = 2.5;   // corridor width
      ModelExp.SetCorridor(length,width);
      for (int i=1; i <= 100; i++ ) {
         fprintf(stdout, "%d %f\n", i, ModelExp.Rate(i));
      }
  }
  // simulate system
  else if (argv[1][0]=='3') {
      if (argc < 3) {
         fprintf(stderr, "Usage: %s %s <script_file>\n", argv[0],argv[1]);
         exit(0);
      }
      int aux = 0;
      while ((argv[2][aux]!='\n')&&(argv[2][aux]!='\0')) aux++;
      argv[2][aux] = '\0';
      FILE *inputFile = fopen(argv[2],"r");
      if (inputFile == NULL) {
         fprintf(stderr, "%s: No such file\n", argv[2]);
         exit(0);
      }
      MgccSimul myMgccSimulator;
// simulation time & number of replications
//       float warmupTime = 2000.0;
//       float simTime = 1000.0 + warmupTime;
      float warmupTime = 1.0;
      float simTime = 2.0 + warmupTime;
      int i, j, k;
      int ics = 1;
      int repl = 1;
      Stats *pC, *theta, *Eq, *ETs;
//       Stats *pC2;
      rewind(inputFile);
      myMgccSimulator.ReadData(inputFile);
      myMgccSimulator.ShowNet();
```

```
        fprintf(stdout,"Warm-up time\t%f\n", warmupTime);
        fprintf(stdout,"Simulation time\t%f\n", simTime);
        fprintf(stdout,"Replications\t%d\n", repl);
        fprintf(stdout,"Monte Carlo replics\t%d\n", ics);
        pC    = new Stats [myMgccSimulator.GetNodes()];
        theta = new Stats [myMgccSimulator.GetNodes()];
        Eq    = new Stats [myMgccSimulator.GetNodes()];
        ETs   = new Stats [myMgccSimulator.GetNodes()];
//        pC2    = new Stats [myMgccSimulator.GetNodes()];
//        fprintf(stdout,"IC\tReplic\tNode\tpC\ttheta\tEq\tEts\n");
        StartCronograph();
        for (k=0; k<ics; k++) {
          for (i=0; i<repl; i++) {
             myMgccSimulator.ProcessSimul(warmupTime,simTime);
             for (j=0; j<myMgccSimulator.GetNodes(); j++) {
                 pC[j].Enter(myMgccSimulator.GetPC(j));
                 theta[j].Enter(myMgccSimulator.GetTheta(j));
                 Eq[j].Enter(myMgccSimulator.GetQ(j));
                 ETs[j].Enter(myMgccSimulator.GetTs(j));
//                 pC2[j].Enter(myMgccSimulator.GetPC2(j));
//                 fprintf(stdout,"%d\t%d\t%d\t%f\t%f\t%f\t%f\n", k+1, i+1, j+1,
//                  myMgccSimulator.GetPC(j),
//                  myMgccSimulator.GetTheta(j),
//                  myMgccSimulator.GetQ(j),
//                  myMgccSimulator.GetTs(j));
             }
//             myMgccSimulator.PrintResults();
          }
        }
#ifndef MGCCSIM_INTERDEP
#ifndef MGCCSIM_CELULAR
        ElapsedTime();
        fprintf(stdout,"Final Statistics:\n");
        const int LLENGTH = 256;
        char Line[LLENGTH];
        fgets(Line, LLENGTH, inputFile);
        while (fscanf(inputFile, "%d/n", &i) == 1 ) {
/*
              fprintf(stdout,"Node (normal CI's)\t%d\n",i);
              fprintf(stdout,"\tp(C)\t%f\t%f\t%f\n",
                     pC[i-1].Mean(),
                     pC[i-1].Mean()-1.96*pC[i-1].StD()/sqrt(repl),
                     pC[i-1].Mean()+1.96*pC[i-1].StD()/sqrt(repl));
              fprintf(stdout,"\ttheta\t%f\t%f\t%f\n",
                     theta[i-1].Mean(),
                     theta[i-1].Mean()-1.96*theta[i-1].StD()/sqrt(repl),
                     theta[i-1].Mean()+1.96*theta[i-1].StD()/sqrt(repl));
              fprintf(stdout,"\tE(q)\t%f\t%f\t%f\n",
                     Eq[i-1].Mean(),
                     Eq[i-1].Mean()-1.96*Eq[i-1].StD()/sqrt(repl),
                     Eq[i-1].Mean()+1.96*Eq[i-1].StD()/sqrt(repl));
              fprintf(stdout,"\tE(ts)\t%f\t%f\t%f\n",
                     ETs[i-1].Mean(),
                     ETs[i-1].Mean()-1.96*ETs[i-1].StD()/sqrt(repl),
                     ETs[i-1].Mean()+1.96*ETs[i-1].StD()/sqrt(repl));
//              fprintf(stdout,"\tp(C)'\t%f\t%f\t%f\n",
//                     pC2[i-1].Mean(),
//                     pC2[i-1].Mean()-1.96*pC2[i-1].StD()/sqrt(repl),
//                     pC2[i-1].Mean()+1.96*pC2[i-1].StD()/sqrt(repl));
*/
              fprintf(stdout,"Node (percent CI's)\t%d\n",i);
              fprintf(stdout,"\tp(C)\t%f\t%f\t%f\n",
                     pC[i-1].Mean(),
                     pC[i-1].Quantile(0.025),
                     pC[i-1].Quantile(0.975));
              fprintf(stdout,"\ttheta\t%f\t%f\t%f\n",
                     theta[i-1].Mean(),
                     theta[i-1].Quantile(0.025),
                     theta[i-1].Quantile(0.975));
              fprintf(stdout,"\tE(q)\t%f\t%f\t%f\n",
                     Eq[i-1].Mean(),
                     Eq[i-1].Quantile(0.025),
                     Eq[i-1].Quantile(0.975));
              fprintf(stdout,"\tE(ts)\t%f\t%f\t%f\n",
                     ETs[i-1].Mean(),
                     ETs[i-1].Quantile(0.025),
                     ETs[i-1].Quantile(0.975));
//              fprintf(stdout,"\tp(C)'\t%f\t%f\t%f\n",
//                     pC2[i-1].Mean(),
//                     pC2[i-1].Quantile(0.025),
//                     pC2[i-1].Quantile(0.975));
        }
#endif
#endif
        fclose(inputFile);
        delete[] pC;
        delete[] theta;
        delete[] Eq;
        delete[] ETs;
//        delete[] pC2;
    }
    // don't know what else to do
    else {
       fprintf(stderr, "Operation %s unknow\n", argv[1]);
    }
    return 0;
}
```

Código A.2: Mgccsim.c

```
//
// Given:
//   - velocity congestion model and
//   - input lambda,
// this library determines, by simulation, the performance measures:
//   - blocking probability,
//   - throughput rate,
//   - number of customers in the sistem (WIP),
//   - waiting time.
//
// Authors:
//    Paula de Campos Oliveira
//    Frederico R. B. Cruz
//    Departamento de Estatistica
//    Universidade Federal de Minas Gerais
//    Brazil
//    E-mail: pcampol@yahoo.com.br,fcruz@est.ufmg.br
//
//
// Version:
//    5.0
//
// Date:
//    Jul/2010
//
#ifndef MGCCSIM_CPP
#define MGCCSIM_CPP
//
#include <stdio.h>
#include <math.h>
#include "cmusr.cpp"
#include "randx.c"
//
//*************************************************************************
// introduced by Cruz & Araujo (2004)
//*************************************************************************
// #define or #undef below for interdeparture studies
//#define MGCCSIM_INTERDEP
#undef MGCCSIM_INTERDEP
//*************************************************************************
// end of introduced by Cruz & Araujo (2004)
//*************************************************************************
//*************************************************************************
// introduced by Cruz & Oliveira (2004)
//*************************************************************************
// #define or #undef below for modeling of mobile systems
#define MGCCSIM_CELULAR
//#undef MGCCSIM_CELULAR
//*************************************************************************
// end of introduced by Cruz & Oliveira (2004)
//*************************************************************************
//
// these are general settings
//
#define MGCCSIM_IN_FILE stdin     // input file
#define MGCCSIM_OUT_FILE stdout   // output file
#define MGCCSIM_ERR_FILE stderr   // error file
#define MGCCSIM_EPSILON 1E-06     // precision
//
// events
//
#define MGCCSIM_ARRIVAL 0    // arrival event
#define MGCCSIM_DEPARTURE 1  // departure event
#define MGCCSIM_WARMUP 2     // end of warm-up period
#define MGCCSIM_END 3        // end of simulation event
#define MGCCSIM_UNK -1       // unknown event
#define MGCCSIM_NO 0         // boolean value no
#define MGCCSIM_YES 1        // boolean value yes
//
// entity (pedestrian) definition
//
class MgccEntity {
protected:
public:
  static int last;
  int id;                 // entity identification
  float sistArrival;      // system arrival time
  float queueArrival;     // queue arrival time
#ifdef MGCCSIM_CELULAR
  float cellArrival;           // cell arrival time
#endif
  float timeLastChange; // time when occured last change in Vn
  float lastPosition;    // last position since last change in Vn
  int   blocked;         // is entity currently blocked?
  float timeBlocked;     // time when blocked
  int   blockedAgain;    // is it blocked again?
  MgccEntity(void);
  MgccEntity(MgccEntity &myEnt);
  ¬MgccEntity(void);
  MgccEntity &operator = (MgccEntity &myEnt);
  void Print(void);
};
int MgccEntity::last=0;
//
// event definition
//
class MgccEvent {
protected:
```

```
public:
  int whichQueue;    // where it occurs
  float occurTime;   // time of occurrence
  int type;          // type of event (arrival,departure or final)
  MgccEntity *myMgccEntity; // entity
  MgccEvent(void);
  MgccEvent(const MgccEvent &myEvent);
  ~MgccEvent(void);
  MgccEvent &operator = (const MgccEvent &myEvent);
  int operator < (const MgccEvent &myEvent);
  int operator <= (const MgccEvent &myEvent);
  void Print(void);
};
//
// mgccEventQueue node type
//
typedef struct EQNode {
     MgccEvent *Node;
     struct EQNode *Next;
} EQNodeType;
//
// list of events
//
class MgccEventQueue {
protected:
  EQNodeType *head;
  EQNodeType *tail;
  EQNodeType *Current;
public:
  MgccEventQueue(void);
  ~MgccEventQueue(void);
  int Reset(void);
  int Insert(const MgccEvent &myEvent); // insert event in list
  MgccEvent *GetEarliest(void);         // get earliest event from list
  MgccEvent *GetFirst(void);            // get first event from list
  MgccEvent *GetNext(void);             // get next event from list
  MgccEvent *ShowFirst(void);           // show first event from list
  MgccEvent *ShowNext(void);            // show next from list
  int ReSort(void);                     // sort list
  void Print(void);                     // print all event list
};
//
// resource (corridors) settings
//
class MgccResource  {
private:
  int sumBloc;        // total number of blocked customers
  int sumArr;         // total number of arrivals
  int sumDep;         // total number of departures
  double sumTime;     // total time in queue
  int users;          // current number of users
protected:
  double lambda;      // external input
public:
  CMGen *service;     // velocity congestion model
  int sumBloc2;       // total number of twice-blocked customers
  MgccResource(void);
  ~MgccResource(void);
  void   SetService(CMGen *theServ) {service=theServ;}
  void   SetExtLambda(double theLambda) {lambda=theLambda;}
  void   ResetSum(void);
  void   Reset(void);
  void   AddBlocked(void)   {sumBloc++;}
  void   AddBlocked2(void)  {sumBloc2++;}
  void   AddArrival(void)   {sumArr++;}
  void   AddDepart(void)    {sumDep++;}
  void   AddTime(float time){sumTime+=time;}
  void   AddUser(void)      {users++;}
  void   DelUser(void)      {users--;}
  int    GetC(void)         {return service->GetC();}
  double GetEts1(void)      {return service->GetEts1();}
  double GetLambda(void)    {return lambda;}
  double GetPC(void)        {return (double)sumBloc/(sumBloc+sumDep);}
  double GetPC2(void)       {return (double)sumBloc2/(sumBloc2+sumDep);}
  double GetDepart(void)    {return (double)sumDep;}
  double GetTs(void)        {return sumTime/sumDep;}
  int    GetUsers(void)     {return users;}
  void   Print(void);
};
//
// queue simulation
//
class MgccSimul {
private:
  static int seed1;
  static int seed2;
  static int cont;
  int nOfNodes;
  MgccResource *myMgccResource;
  MgccEventQueue myMgccEventQueue;
  float **arcs;
  float warmupTime;
  float totalTime;
#ifdef MGCCSIM_INTERDEP
  static int MGCCSIM_INTERDEP_FIRST;
#endif
#ifdef MGCCSIM_CELULAR
  static int MGCCSIM_CELULAR_FIRST;
  static int first;
```

```
  int **cels;
#endif
public:
  MgccSimul(void);                                  // default constructor
  ~MgccSimul(void);                                 // destructor
  int ReadData(FILE *inputFile);                    // read data
  int ShowNet(void);
  int GetNodes(void)         {return nOfNodes;}
  double GetPC(int i)        {return myMgccResource[i].GetPC();}
  double GetPC2(int i)       {return myMgccResource[i].GetPC2();}
  double GetTheta(int i)     {return myMgccResource[i].GetDepart()/(totalTime-warmupTime);}
  double GetQ(int i)         {return GetTheta(i)*GetTs(i);}
  double GetTs(int i)        {return myMgccResource[i].GetTs();}
  int PrintResults(void);
  int ProcessSimul(float warmup, float finalTime);
  int ProcessEvent(MgccEvent *myEvent);
  int ProcessArrival(MgccEvent *myArr);
  int ProcessDepart(MgccEvent *myDep);
  int DelayST(int queue, double now);
  int AdvanceST(int queue, double now);
};
int MgccSimul::seed1 = 13579;
int MgccSimul::seed2 = 24680;
int MgccSimul::cont = 0;
#ifdef MGCCSIM_INTERDEP
  int MgccSimul::MGCCSIM_INTERDEP_FIRST = 1;
#endif
#ifdef MGCCSIM_CELULAR
  int MgccSimul::MGCCSIM_CELULAR_FIRST = 1;
#endif
//
// implementation
//
MgccEntity::MgccEntity(void):
  id(last++),
  sistArrival(0.0),
  queueArrival(0.0),
#ifdef MGCCSIM_CELULAR
  cellArrival(0.0),
#endif
  timeLastChange(0.0),
  lastPosition(0.0),
  blocked(MGCCSIM_NO),
  timeBlocked(-1.0),
  blockedAgain(MGCCSIM_NO) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE,"MgccEntity::MgccEntity():\n");
#endif
}
MgccEntity::MgccEntity(MgccEntity &myEnt) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE,"MgccEntity::MgccEntity(MgccEntity):\n");
#endif
  id=myEnt.id;
  sistArrival=myEnt.sistArrival;
  queueArrival=myEnt.queueArrival;
#ifdef MGCCSIM_CELULAR
  cellArrival=myEnt.cellArrival;
#endif
  timeLastChange=myEnt.timeLastChange;
  lastPosition=myEnt.lastPosition;
  blocked=myEnt.blocked;
  timeBlocked=myEnt.timeBlocked;
  blockedAgain=myEnt.blockedAgain;
}
MgccEntity::~MgccEntity(void) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE,"MgccEntity::~MgccEntity():\n");
#endif
}
MgccEntity &MgccEntity::operator = (MgccEntity &myEnt) {
  id=myEnt.id;
  sistArrival=myEnt.sistArrival;
  queueArrival=myEnt.queueArrival;
#ifdef MGCCSIM_CELULAR
  cellArrival=myEnt.cellArrival;
#endif
  timeLastChange=myEnt.timeLastChange;
  lastPosition=myEnt.lastPosition;
  blocked=myEnt.blocked;
  timeBlocked=myEnt.timeBlocked;
  blockedAgain=myEnt.blockedAgain;
  return *this;
}
void MgccEntity::Print(void) {
  fprintf(MGCCSIM_OUT_FILE,"MgccEntity::Print():\n");
  fprintf(MGCCSIM_OUT_FILE,"\tid\t%d\n",id+1);
  fprintf(MGCCSIM_OUT_FILE,"\tsistArrival\t%f\n",sistArrival);
  fprintf(MGCCSIM_OUT_FILE,"\tqueueArrival\t%f\n", queueArrival);
#ifdef MGCCSIM_CELULAR
  fprintf(MGCCSIM_OUT_FILE,"\tcellArrival\t%f\n", cellArrival);
#endif
  fprintf(MGCCSIM_OUT_FILE,"\ttimeLastChange\t%f\n", timeLastChange);
  fprintf(MGCCSIM_OUT_FILE,"\tlastPosition\t%f\n", lastPosition);
  fprintf(MGCCSIM_OUT_FILE,"\tblocked\t%d\n", blocked);
  fprintf(MGCCSIM_OUT_FILE,"\ttimeBlocked\t%f\n", timeBlocked);
  fprintf(MGCCSIM_OUT_FILE,"\tblockedAgain\t%d\n", blockedAgain);
}
MgccEvent::MgccEvent(void):
```

```
  whichQueue(0),
  occurTime(0),
  type(MGCCSIM_UNK),
  myMgccEntity(NULL)  {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE, "MgccEvent::MgccEvent():\n");
#endif
}
MgccEvent::¬MgccEvent(void) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE, "MgccEvent::¬MgccEvent():\n");
#endif
}
MgccEvent::MgccEvent(const MgccEvent &myEvent) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE, "MgccEvent::MgccEvent(MgccEvent):\n");
#endif
  whichQueue=myEvent.whichQueue;
  occurTime=myEvent.occurTime;
  type=myEvent.type;
  myMgccEntity=myEvent.myMgccEntity;
}
MgccEvent &MgccEvent::operator = (const MgccEvent &myEvent) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE, "MgccEvent::operator = (MgccEvent):\n");
#endif
  whichQueue=myEvent.whichQueue;
  occurTime=myEvent.occurTime;
  type=myEvent.type;
  myMgccEntity=myEvent.myMgccEntity;
  return *this;
}
int MgccEvent::operator < (const MgccEvent &myEvent) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE, "MgccEvent::operator < (MgccEvent):\n");
#endif
  if (occurTime < myEvent.occurTime) {
        return (1);
//  } else if (occurTime == myEvent.occurTime) {
//          return (myMgccEntity->id < myEvent.myMgccEntity->id);
  }
  return (0);
}
int MgccEvent::operator ≤ (const MgccEvent &myEvent) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE, "MgccEvent::operator ≤ (MgccEvent):\n");
#endif
  if (occurTime < myEvent.occurTime) {
        return (1);
  } else if (occurTime == myEvent.occurTime) {
        return (myMgccEntity->id ≤ myEvent.myMgccEntity->id);
  }
  return (0);
}
void MgccEvent::Print(void) {
  fprintf(MGCCSIM_OUT_FILE, "MgccEvent::Print():\n");
  fprintf(MGCCSIM_OUT_FILE, "\twhichQueue\t%d\n", whichQueue);
  fprintf(MGCCSIM_OUT_FILE, "\toccurTime\t%f\n", occurTime);
  if (type==MGCCSIM_ARRIVAL)
    fprintf(MGCCSIM_OUT_FILE, "\ttype\tMGCCSIM_ARRIVAL\n");
  else if (type==MGCCSIM_DEPARTURE)
    fprintf(MGCCSIM_OUT_FILE, "\ttype\tMGCCSIM_DEPARTURE\n");
  else if (type==MGCCSIM_WARMUP)
    fprintf(MGCCSIM_OUT_FILE, "\ttype\tMGCCSIM_WARMUP\n");
  else if (type==MGCCSIM_END)
    fprintf(MGCCSIM_OUT_FILE, "\ttype\tMGCCSIM_END\n");
  else
    fprintf(MGCCSIM_OUT_FILE, "\ttype\tUNKNOWN\n");
  myMgccEntity->Print();
}
MgccEventQueue::MgccEventQueue(void) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE, "MgccEventQueue::MgccEventQueue():\n");
#endif
  head = new EQNodeType;
  head->Node=NULL;
  head->Next=NULL;
  tail = head;
  Current = head->Next;
}
MgccEventQueue::¬MgccEventQueue(void) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE, "MgccEventQueue::¬MgccEventQueue():\n");
#endif
  EQNodeType *Aux1, *Aux2;
  Aux1 = head->Next;
  while (Aux1 != NULL) {
    Aux2 = Aux1;
    Aux1 = Aux1->Next;
#if MGCCSIM_DEBUG
    fprintf(MGCCSIM_OUT_FILE, "&(Aux2->Node)\t%p\n",Aux2->Node);
#endif
    delete Aux2->Node;
#if MGCCSIM_DEBUG
    fprintf(MGCCSIM_OUT_FILE, "&(Aux2)\t%p\n",Aux2);
#endif
    delete Aux2;
  }
#if MGCCSIM_DEBUG
```

```
  fprintf(MGCCSIM_OUT_FILE,"&(Head)\t%p\n",head);
#endif
  delete head;
}
int MgccEventQueue::Reset(void) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::Reset():\n");
#endif
  EQNodeType *Aux1, *Aux2;
  Aux1 = head->Next;
  while (Aux1 != NULL) {
    Aux2 = Aux1;
    Aux1 = Aux1->Next;
#if MGCCSIM_DEBUG
    fprintf(MGCCSIM_OUT_FILE,"&(Aux2->Node)\t%p\n",Aux2->Node);
#endif
    delete Aux2->Node;
#if MGCCSIM_DEBUG
    fprintf(MGCCSIM_OUT_FILE,"&(Aux2)\t%p\n",Aux2);
#endif
    delete Aux2;
  }
  delete head;
  head = new EQNodeType;
  head->Node=NULL;
  head->Next=NULL;
  tail = head;
  Current = head->Next;
  return 0;
}
int MgccEventQueue::Insert(const MgccEvent &myEvent) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::Insert(MgccEvent)\n");
#endif
//  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::Insert: before\n");
//  Print();
//  EQNodeType *Aux;
//  Aux = Queue;
  // find position
//  while ( (Aux->Next != NULL)&&(*(Aux->Next->Node) <= myEvent) ) {
//     Aux=Aux->Next;
//  }
  // insert
  tail->Next = new EQNodeType;
  if (tail->Next==NULL) {
     fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::Insert: Error, no memory\n");
     exit(1);
  }
  tail->Next->Node = new MgccEvent(myEvent);
  tail->Next->Next = NULL;
  tail = tail->Next;
//  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::Insert: after\n");
//  Print();
  return 0;
}
MgccEvent *MgccEventQueue::GetEarliest(void) {
#if MGCCSIM_DEBUG
    fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::GetEarliest():\n");
#endif
  EQNodeType *eqNode, *earliest;
  MgccEvent *earliestEvent;
  eqNode = head;
  earliest = head;
  if (earliest->Next==NULL) {
     fprintf(MGCCSIM_ERR_FILE,
     "MgccEventQueue::GetEarliest: Error, empty queue\n");
     exit(1);
  }
  // find earliest
  earliestEvent = earliest->Next->Node;
  while (eqNode->Next!=NULL) {
      if ( (*eqNode->Next->Node) < (*earliestEvent) ) {
         earliestEvent = eqNode->Next->Node;
         earliest = eqNode;
//          fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::GetEarliest: found\n");
//          earliestEvent->Print();
      }
      eqNode = eqNode->Next;
  }
  // delete it from list and update tail
  eqNode = earliest->Next;
  if (eqNode==tail) {
     tail = earliest;
  }
  earliest->Next = earliest->Next->Next;
  delete eqNode;
  return(earliestEvent);
}
MgccEvent *MgccEventQueue::GetFirst(void) {
#if MGCCSIM_DEBUG
//  fprintf(OUT_FILE,"MgccEventQueue::GetFirst():\n");
#endif
  return(GetNext());
}
MgccEvent *MgccEventQueue::GetNext(void) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::GetNext():\n");
#endif
  EQNodeType *Aux;
```

```
  MgccEvent *event;
  Aux = head->Next;
  if (Aux == NULL) {
     event=NULL;
  } else {
     event = Aux->Node;
     head->Next = Aux->Next;
  }
  return(event);
}
MgccEvent *MgccEventQueue::ShowFirst(void) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::ShowFirst()\n");
#endif
  Current = head->Next;
  return(ShowNext());
}
MgccEvent *MgccEventQueue::ShowNext(void) {
#if MGCCSIM_DEBUG
//  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::ShowNext():\n");
#endif
  MgccEvent *event;
  if (Current==NULL) {
     event=NULL;
  } else {
     event=Current->Node;
     Current=Current->Next;
  }
  return(event);
}
int MgccEventQueue::ReSort(void) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::ReSort():\n");
#endif
  MgccEventQueue *newQueue = new MgccEventQueue;
  MgccEvent *event;
  for (event=this->GetNext(); event!=NULL; event=this->GetNext()) {
     newQueue->Insert((*event));
  }
  delete this->head;
  this->head=newQueue->head;
  return 0;
}
void MgccEventQueue::Print(void) {
  fprintf(MGCCSIM_OUT_FILE,"MgccEventQueue::Print():\n");
  MgccEvent *event;
  for (event=ShowFirst(); event!=NULL; event=ShowNext()) {
     event->Print();
  }
}
MgccResource::MgccResource(void):
 sumBloc(0),
 sumArr(0),
 sumDep(0),
 sumTime(0.0),
 users(0),
 lambda(0),
 service(NULL),
 sumBloc2(0) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccResource::MgccResource():\n");
#endif
}
MgccResource::~MgccResource(void) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccResource::~MgccResource()\n");
#endif
}
void MgccResource::ResetSum(void) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccResource::ResetSum():\n");
#endif
  sumBloc=0;
  sumArr=0;
  sumDep=0;
  sumTime=0.0;
  sumBloc2=0;
}
void MgccResource::Reset(void) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccResource::Reset():\n");
#endif
  ResetSum();
  users=0;
}
void MgccResource::Print(void) {
  fprintf(MGCCSIM_OUT_FILE,"MgccResource::Print():\n");
  fprintf(MGCCSIM_OUT_FILE,"\t sumBloc\t%d\n",sumBloc);
  fprintf(MGCCSIM_OUT_FILE,"\t sumArr\t%d\n",sumArr);
  fprintf(MGCCSIM_OUT_FILE,"\t sumDep\t%d\n",sumDep);
  fprintf(MGCCSIM_OUT_FILE,"\t sumTime\t%f\n",sumTime);
  fprintf(MGCCSIM_OUT_FILE,"\t users\t%d\n",users);
  fprintf(MGCCSIM_OUT_FILE,"\t lambda\t%f\n",lambda);
  fprintf(MGCCSIM_OUT_FILE,"\t ES\t%f\n",service->GetEts1());
  fprintf(MGCCSIM_OUT_FILE,"\t C\t%d\n",service->GetC());
  fprintf(MGCCSIM_OUT_FILE,"\t sumBloc2\t%d\n",sumBloc2);
}
MgccSimul::MgccSimul(void):
 nOfNodes(0),
```

```
  myMgccResource(NULL),
  myMgccEventQueue(),
  arcs(NULL),
  warmupTime(0.0),
  totalTime(0.0) {
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE,"MgccSimul::MgccSimul():\n");
#endif
 }
 MgccSimul::~MgccSimul(void) {
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE,"MgccSimul::~MgccSimul():\n");
#endif
   delete[] myMgccResource;
 }
 int MgccSimul::ReadData(FILE *inputFile) {
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ReadData(FILE):\n");
#endif
   const int LLENGTH = 256;
   char Line[LLENGTH];
   int index, orig, dest;
   float prob;
   int i, j, serv;
   float length, width, lambda;
   CMLinUsr *servLin;
   CMExpUsr *servExp;
   // read and set number of nodes
   fgets(Line, LLENGTH, inputFile);
   fscanf(inputFile, "%d\n", &nOfNodes );
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE, "%d\n", nOfNodes);
#endif
   myMgccResource = new MgccResource[nOfNodes];
   arcs = new float *[nOfNodes];
   for (i=0; i < nOfNodes; i++) {
       arcs[i] = new float[nOfNodes];
       for (j=0; j < nOfNodes; j++) {
           arcs[i][j] = 0.0;
       }
   }
   // read arc related data
   fgets(Line, LLENGTH, inputFile);
   while (fscanf(inputFile, "%d %d %d %f\n",
                 &index, &orig, &dest, &prob)==4) {
        arcs[orig-1][dest-1] = prob;
#if MGCCSIM_DEBUG
        fprintf(MGCCSIM_OUT_FILE,"%d %d %d %f\n",index,orig,dest,prob);
#endif
   }
   // read node number, service, length, width, and lambda
   fgets(Line, LLENGTH, inputFile);
   cont++;
   for (i=0; i < nOfNodes; i++) {
      fscanf(inputFile, "%d %d %f %f %f\n",
             &index,&serv,&length,&width,&lambda );
#if MGCCSIM_DEBUG
      fprintf(MGCCSIM_OUT_FILE, "%d %d %f %f %f\n",
             index,serv,length,width,lambda );
#endif
      //  choose service
      if ( serv==1 ) {
           servLin = new CMLinUsr;
           servLin->SetCorridor(length,width);
           myMgccResource[index-1].SetService(servLin);
      } else if ( serv==2 ) {
           servExp = new CMExpUsr;
           servExp->SetCorridor(length,width);
           myMgccResource[index-1].SetService(servExp);
      } else {
         fprintf(MGCCSIM_ERR_FILE,
         "Usage: service should be 1 (LINEAR) or 2 (EXPONENTIAL)\n");
         exit(1);
      }
      myMgccResource[index-1].SetExtLambda(cont*lambda);
   }
#ifdef MGCCSIM_CELULAR
//**********************************************************************
// introduced by Cruz & Oliveira (2004)
//**********************************************************************
   cels = new int *[nOfNodes];
   for (i=0; i < nOfNodes; i++) {
       cels[i] = new int[nOfNodes];
       for (j=0; j < nOfNodes; j++) {
           cels[i][j] = 0;
       }
   }
   long int current_pos = ftell(inputFile);
   fgets(Line, LLENGTH, inputFile);
   while (fscanf(inputFile, "%d/n", &i) == 1 ) {
//          fprintf(MGCCSIM_OUT_FILE, "node %d\n", i);
   }
   fgets(Line, LLENGTH, inputFile);
   while (fscanf(inputFile, "%d %d/n", &i, &j) == 2 ) {
         cels[i-1][j-1] = 1;
//          fprintf(MGCCSIM_OUT_FILE, "cel %d node %d\n", i, j);
   }
   fseek(inputFile, current_pos, SEEK_SET);
//**********************************************************************
```

```
// end of introduced by Cruz & Oliveira (2004)
//**************************************************************************
#endif
  return 0;
}
int MgccSimul::ShowNet(void) {
  fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ShowNet():\n");
  int i, j, aux;
  fprintf(MGCCSIM_OUT_FILE, "Nodes\n%d\n", nOfNodes );
  fprintf(MGCCSIM_OUT_FILE, "Arc\tOrig\tDest\tProb\n");
  aux = 0;
  for (i=0; i < nOfNodes; i++) {
      for (j=0; j < nOfNodes; j++) {
          if (arcs[i][j]>0.0) {
//             fprintf(MGCCSIM_OUT_FILE, "   %d        %d              %d %f\n",
             fprintf(MGCCSIM_OUT_FILE, "%d\t%d\t%d\t%.3f\n",
                     ++aux, i+1, j+1, arcs[i][j]);
          }
    }
  }
  for (i=0; i < nOfNodes; i++) {
    fprintf(MGCCSIM_OUT_FILE, "Node\t%d\n", i+1);
    fprintf(MGCCSIM_OUT_FILE,"\tE(ts)\tC\tLambda\n");
    fprintf(MGCCSIM_OUT_FILE,"\t%.3f\t%d\t%.3f\n",
            myMgccResource[i].GetEts1(),
            myMgccResource[i].GetC(),
            myMgccResource[i].GetLambda());
//          myMgccResource[i].Print());
  }
  return 0;
}
int MgccSimul::ProcessSimul(float warmup, float finalTime) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessSimul(%f;%f):\n", warmup, finalTime);
#endif
  MgccEvent event;
  MgccEvent *currentEvent;
  int i;
  warmupTime=warmup;
  totalTime=finalTime;
  // general initialization
  myMgccEventQueue.Reset();
  for (i=0; i<nOfNodes; i++) {
      myMgccResource[i].Reset();
  }
  // insert "last" event
  event.whichQueue = 0;
  event.occurTime = finalTime;
  event.type = MGCCSIM_END;
  event.myMgccEntity = new MgccEntity;
  myMgccEventQueue.Insert(event);
  // insert "warm-up" event
  event.whichQueue = 0;
  event.occurTime = warmupTime;
  event.type = MGCCSIM_WARMUP;
  event.myMgccEntity = new MgccEntity;
  myMgccEventQueue.Insert(event);
  // insert "first" events
  for (i=0; i<nOfNodes; i++) {
      if (myMgccResource[i].GetLambda() > MGCCSIM_EPSILON) {
          event.whichQueue = i;
          event.occurTime = 0.0;
          event.type = MGCCSIM_ARRIVAL;
          event.myMgccEntity = new MgccEntity;
          myMgccEventQueue.Insert(event);
      }
  }
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessSimul: initial event queue\n");
  myMgccEventQueue.Print();
#endif
  // simulation per se
  currentEvent = myMgccEventQueue.GetEarliest();
  while (currentEvent->type != MGCCSIM_END) {
#if MGCCSIM_DEBUG
      fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessSimul: current event\n");
      currentEvent->Print();
      fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessSimul: event queue\n");
      myMgccEventQueue.Print();
#endif
      ProcessEvent(currentEvent);
      currentEvent = myMgccEventQueue.GetEarliest();
  }
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessSimul: current event\n");
  currentEvent->Print();
  fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessSimul: final event queue\n");
  myMgccEventQueue.Print();
#endif
  return 0;
}
int MgccSimul::PrintResults(void) {
  fprintf(MGCCSIM_OUT_FILE,"MgccSimul::PrintResults():\n");
  for (int i=0; i < nOfNodes; i++) {
#if MGCCSIM_DEBUG
    myMgccResource[i].Print();
#endif
    fprintf(MGCCSIM_OUT_FILE,"Node\t%d\n", i+1);
    fprintf(MGCCSIM_OUT_FILE,"\tp(C)\t%f\n", GetPC(i));
```

```
      fprintf(MGCCSIM_OUT_FILE,"\ttheta\t%f\n", GetTheta(i));
      fprintf(MGCCSIM_OUT_FILE,"\tE(q)\t%f\n", GetQ(i));
      fprintf(MGCCSIM_OUT_FILE,"\tE(ts)\t%f\n", GetTs(i));
   }
   return 0;
}
int MgccSimul::ProcessEvent(MgccEvent *myEvent) {
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessEvent(MgccEvent):\n");
#endif
   int i;
   if (myEvent->type==MGCCSIM_ARRIVAL) {
      ProcessArrival(myEvent);
   } else if (myEvent->type==MGCCSIM_DEPARTURE) {
      ProcessDepart(myEvent);
   } else if (myEvent->type==MGCCSIM_WARMUP) {
      for (i=0; i<nOfNodes; i++) {
         myMgccResource[i].ResetSum();
      }
      delete myEvent->myMgccEntity;
      delete myEvent;
   } else {
      fprintf(MGCCSIM_OUT_FILE, "MgccSimul::ProcessEvent: error unknow event");
      exit(1);
   }
   return 0;
}
int MgccSimul::ProcessArrival(MgccEvent *myArr) {
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessArrival(MgccEvent):\n");
#endif
   MgccEvent event;
   int queue, n;
   double now, Vn, L;
   queue = myArr->whichQueue;
   now = myArr->occurTime;
   // create new arrival
   event.whichQueue = queue;
   event.occurTime = now+rExpo(&seed1,myMgccResource[queue].GetLambda());
   event.type = MGCCSIM_ARRIVAL;
   event.myMgccEntity = new MgccEntity;
   event.myMgccEntity->sistArrival = event.occurTime;
   event.myMgccEntity->queueArrival = event.occurTime;
#ifdef MGCCSIM_CELULAR
   event.myMgccEntity->cellArrival = event.occurTime;
#endif
   event.myMgccEntity->timeLastChange = event.occurTime;
   event.myMgccEntity->lastPosition = 0.0;
   event.myMgccEntity->blocked=MGCCSIM_NO;
   event.myMgccEntity->timeBlocked=-1.0;
   event.myMgccEntity->blockedAgain=MGCCSIM_NO;
   myMgccEventQueue.Insert(event);
   // if queue's blocked, just reject arrival
   if (myMgccResource[queue].GetUsers() ≥ myMgccResource[queue].GetC()) {
#if MGCCSIM_DEBUG
      fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessArrival: arrival rejected\n");
#endif
      // update statistics
      myMgccResource[queue].AddBlocked();
      // free entity
      delete myArr->myMgccEntity;
   // if there is enough room, admit arrival
   } else {
#if MGCCSIM_DEBUG
      fprintf(MGCCSIM_OUT_FILE,
              "MgccSimul::ProcessArrival: arrival accepted\n");
#endif
      // update statistics
      myMgccResource[queue].AddArrival();
      myMgccResource[queue].AddUser();
      // schedule new departure
      event.whichQueue = queue;
      n = myMgccResource[queue].GetUsers();
      Vn = myMgccResource[queue].service->GetV1()*
           myMgccResource[queue].service->Rate(n);
      L = myMgccResource[queue].service->GetEts1()*
          myMgccResource[queue].service->GetV1();
      event.occurTime = now + L/Vn;
      event.type = MGCCSIM_DEPARTURE;
      event.myMgccEntity = myArr->myMgccEntity;
      // delay everybody
      DelayST(queue,now);
      // insert new departure
      myMgccEventQueue.Insert(event);
   }
   delete myArr;
   return 0;
}
int MgccSimul::ProcessDepart(MgccEvent *myDep) {
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessDepart(MgccEvent):\n");
   fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessDepart:() now\t%f\n",
           myDep->occurTime);
#endif
   MgccEvent event;
   MgccEvent *currEvent;
   int queue, orig, dest, n;
   double Vn, L;
   double now, prob, sumProb, dTime, earliest;
```

```
    queue = myDep->whichQueue;
    now = myDep->occurTime;
    // select forwarding queue
    prob = rUnif(&seed2);
    orig = queue;
    dest = 0;
    sumProb = arcs[orig][dest];
    while ((dest < (nOfNodes-1))&&(sumProb < prob)) {
            dest++;
            sumProb += arcs[orig][dest];
    }
    // if there is not a forwarding queue ...
    if (sumProb < prob) {
#if MGCCSIM_DEBUG
      fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessDepart: final destination\n");
#endif
      // update statistics
      myMgccResource[queue].AddDepart();
      dTime = now - myDep->myMgccEntity->queueArrival;
      myMgccResource[queue].AddTime(dTime);
      myMgccResource[queue].DelUser();
#ifdef MGCCSIM_INTERDEP
//*****************************************************************
// introduced by Cruz & Araujo (2004)
//*****************************************************************
      // print departure & service times
      if (MGCCSIM_INTERDEP_FIRST) {
           MGCCSIM_INTERDEP_FIRST = 0;
           fprintf(MGCCSIM_OUT_FILE, "Node\tEntity\tdT.serv\tT.depa\n");
      }
      dTime = now - myDep->myMgccEntity->queueArrival;
      fprintf(MGCCSIM_OUT_FILE, "%d\t%d\t%12.8f\t%12.8f\n",
           queue, myDep->myMgccEntity->id+1, dTime, now);
//*****************************************************************
// end of introduced by Araujo & Cruz (2004)
//*****************************************************************
#endif
#ifdef MGCCSIM_CELULAR
//*****************************************************************
// introduced by Cruz & Oliveira (2004)
//*****************************************************************
      // print departure & service times
      if (MGCCSIM_CELULAR_FIRST) {
           MGCCSIM_CELULAR_FIRST = 0;
           fprintf(MGCCSIM_OUT_FILE, "Cel\tEntity\tdT.serv\tT.depa\n");
      }
      int celOrig=0;
      while ((celOrig<(nOfNodes-1))&&(cels[celOrig][orig]!=1)) celOrig++;
      dTime = now - myDep->myMgccEntity->cellArrival;
      fprintf(MGCCSIM_OUT_FILE, "%d\t%d\t%12.8f\t%12.8f\n",
           celOrig, myDep->myMgccEntity->id+1, dTime, now);
//*****************************************************************
// end of introduced by Oliveira & Cruz (2004)
//*****************************************************************
#endif
//      fprintf(MGCCSIM_OUT_FILE, "t \t%d\t%d\t%12.8f\n",
//           queue, myDep->myMgccEntity->id+1, now);
//      fprintf(MGCCSIM_OUT_FILE, "dt\t%d\t%d\t%12.8f\n",
//           queue, myDep->myMgccEntity->id+1, dTime);
      // advance everybody
      AdvanceST(queue,now);
#if MGCCSIM_DEBUG
      myDep->myMgccEntity->Print();
#endif
      // just free the entity
      delete myDep->myMgccEntity;
    // if there is a forwarding queue
    } else {
#if MGCCSIM_DEBUG
      fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessDepart: destination %d\n", dest);
#endif
      // if destination queue is full, keep entity in its original queue
      if (myMgccResource[dest].GetUsers() >= myMgccResource[dest].GetC()) {
#if MGCCSIM_DEBUG
         fprintf(MGCCSIM_OUT_FILE, "MgccSimul::ProcessDepart: delayed\n");
#endif
         // estimate time until next availability
         earliest = totalTime;
         for (currEvent=myMgccEventQueue.ShowFirst(); currEvent!=NULL;
                currEvent=myMgccEventQueue.ShowNext()) {
             if ( (currEvent->whichQueue==dest)&&
                  (currEvent->type==MGCCSIM_DEPARTURE)&&
                  (currEvent->occurTime<earliest) ) {
                earliest = currEvent->occurTime;
             }
         }
         // update everything
         event.whichQueue = queue;
         event.occurTime = earliest;
         event.type = MGCCSIM_DEPARTURE;
         event.myMgccEntity = myDep->myMgccEntity;
//
//         // if necessary update status, and statistics
//         L = myMgccResource[queue].service->GetEts1()*
//             myMgccResource[queue].service->GetV1();
//         if (event.myMgccEntity->lastPosition<L) {
//             event.myMgccEntity->lastPosition = L;
//            myMgccResource[dest].AddBlocked();
//            event.myMgccEntity->timeBlocked = now;
```

```
//
        // if it was not blocked before, update status & statistics
        if (event.myMgccEntity->blocked==MGCCSIM_NO) {
#if MGCCSIM_DEBUG
          fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessDepart: blocked\n");
#endif
             L = myMgccResource[queue].service->GetEts1()*
             myMgccResource[queue].service->GetV1();
         event.myMgccEntity->lastPosition = L;
          event.myMgccEntity->timeLastChange = now;
         event.myMgccEntity->blocked=MGCCSIM_YES;
          event.myMgccEntity->timeBlocked = now;
          myMgccResource[dest].AddBlocked();
        // if this is the second blocking, update status & statistics
        } else if (event.myMgccEntity->blockedAgain==MGCCSIM_NO) {
#if MGCCSIM_DEBUG
          fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessDepart: double-blocked\n");
#endif
          event.myMgccEntity->blockedAgain=MGCCSIM_YES;
              myMgccResource[queue].AddBlocked2();
        }
#if MGCCSIM_DEBUG
        event.myMgccEntity->Print();
#endif
        // reinsert departure at once
        myMgccEventQueue.Insert(event);
      // if there is enough room, admit departure into the next queue
      } else {
#if MGCCSIM_DEBUG
        fprintf(MGCCSIM_OUT_FILE,"MgccSimul::ProcessDepart: accepted\n");
#endif
        // update statistics of current queue
        myMgccResource[queue].AddDepart();
        dTime = now - myDep->myMgccEntity->queueArrival;
        myMgccResource[queue].AddTime(dTime);
        myMgccResource[queue].DelUser();
#ifdef MGCCSIM_INTERDEP
//***********************************************************************
// introduced by Cruz & Araujo (2004)
//***********************************************************************
        // print service & departure
        if (MGCCSIM_INTERDEP_FIRST) {
         MGCCSIM_INTERDEP_FIRST = 0;
          fprintf(MGCCSIM_OUT_FILE, "Node\tEntity\tdT.serv\tT.depa\n");
        }
        dTime = now - myDep->myMgccEntity->queueArrival;
        fprintf(MGCCSIM_OUT_FILE, "%d\t%d\t%12.8f\t%12.8f\n",
          queue, myDep->myMgccEntity->id+1, dTime, now);
//***********************************************************************
// end of introduced by Cruz & Araujo (2004)
//***********************************************************************
#endif
#ifdef MGCCSIM_CELULAR
//***********************************************************************
// introduced by Cruz & Oliveira (2004)
//***********************************************************************
        // print departure & service times
        if (MGCCSIM_CELULAR_FIRST) {
         MGCCSIM_CELULAR_FIRST = 0;
          fprintf(MGCCSIM_OUT_FILE, "Cel\tEntity\tdT.serv\tT.depa\n");
        }
        int celOrig=0;
        while ((celOrig<(nOfNodes-1))&&(cels[celOrig][orig]!=1)) celOrig++;
        int celDest=0;
        while ((celDest<(nOfNodes-1))&&(cels[celDest][dest]!=1)) celDest++;
        if (celOrig!=celDest) {
         dTime = now - myDep->myMgccEntity->cellArrival;
          fprintf(MGCCSIM_OUT_FILE, "%d\t%d\t%12.8f\t%12.8f\n",
          celOrig, myDep->myMgccEntity->id+1, dTime, now);
        }
//***********************************************************************
// end of introduced by Cruz & Oliveira (2004)
//***********************************************************************
#endif
/*
//          if (myDep->myMgccEntity->timeBlocked < 0.0) {
        if (myDep->myMgccEntity->blocked==MGCCSIM_NO) {
           // entity never blocked
           dTime = now - myDep->myMgccEntity->queueArrival;
           fprintf(MGCCSIM_OUT_FILE, "dt\t%d\t%d\t%12.8f\n", queue, myDep->myMgccEntity->id+1, dTime);
           fprintf(MGCCSIM_OUT_FILE, "t \t%d\t%d\t%12.8f\n", queue, myDep->myMgccEntity->id+1, now);
           // entity skiped holding node
           dTime = 0.0;
           fprintf(MGCCSIM_OUT_FILE, "dt\th%d\t%d\t%12.8f\n", queue, myDep->myMgccEntity->id+1, dTime);
           fprintf(MGCCSIM_OUT_FILE, "t \th%d\t%d\t%12.8f\n", queue, myDep->myMgccEntity->id+1, now);
        } else {
           // entity blocked
           dTime = myDep->myMgccEntity->timeBlocked - myDep->myMgccEntity->queueArrival;
           fprintf(MGCCSIM_OUT_FILE, "dt\t%d\t%d\t%12.8f\n", queue, myDep->myMgccEntity->id+1, dTime);
           fprintf(MGCCSIM_OUT_FILE, "t \t%d\t%d\t%12.8f\n", queue, myDep->myMgccEntity->id+1, myDep->
                myMgccEntity->timeBlocked);
           // entity stayed at holding node
           dTime=now - myDep->myMgccEntity->timeBlocked;
           fprintf(MGCCSIM_OUT_FILE, "t \th%d\t%d\t%12.8f\n", queue, myDep->myMgccEntity->id+1, now);
           fprintf(MGCCSIM_OUT_FILE, "dt\th%d\t%d\t%12.8f\n", queue, myDep->myMgccEntity->id+1, dTime);
        }
*/
        // advance everybody in current queue
        AdvanceST(queue,now);
```

```
        // update statistics of destination queue
        myMgccResource[dest].AddArrival();
        myMgccResource[dest].AddUser();
        // schedule new departure
        event.whichQueue = dest;
        n = myMgccResource[dest].GetUsers();
        Vn = myMgccResource[dest].service->GetV1()*
             myMgccResource[dest].service->Rate(n);
        L = myMgccResource[dest].service->GetEts1()*
            myMgccResource[dest].service->GetV1();
        event.occurTime = now + L/Vn;
        event.type = MGCCSIM_DEPARTURE;
        event.myMgccEntity = myDep->myMgccEntity;
        event.myMgccEntity->queueArrival = now;
#ifdef MGCCSIM_CELULAR
        if (celOrig!=celDest) {
         event.myMgccEntity->cellArrival = now;
        }
#endif
        event.myMgccEntity->timeLastChange = now;
        event.myMgccEntity->lastPosition = 0.0;
        event.myMgccEntity->blocked=MGCCSIM_NO;
        event.myMgccEntity->timeBlocked=-1.0;
        event.myMgccEntity->blockedAgain=MGCCSIM_NO;
        // delay everybody in destination queue
        DelayST(dest,now);
#if MGCCSIM_DEBUG
        event.myMgccEntity->Print();
#endif
        // insert departure in destination queue
        myMgccEventQueue.Insert(event);
     }
   }
   delete myDep;
   return 0;
}
int MgccSimul::DelayST(int queue, double now) {
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE,"MgccSimul::DelayST(queue,now):\n");
#endif
   MgccEvent *currEvent;
   double Vn_1, Vn, L, ΔT, ΔSToGo;
   int n;
   // compute queue constants
   n = myMgccResource[queue].GetUsers();
   if (n>1)
      Vn_1 = myMgccResource[queue].service->GetV1()*
             myMgccResource[queue].service->Rate(n-1);
   else
      Vn_1 = myMgccResource[queue].service->GetV1();
   Vn = myMgccResource[queue].service->GetV1()*
        myMgccResource[queue].service->Rate(n);
   L = myMgccResource[queue].service->GetEts1()*
       myMgccResource[queue].service->GetV1();
#if MGCCSIM_DEBUG
   fprintf(MGCCSIM_OUT_FILE,"V(%d)\t%20.16f\tV(%d)\t%20.16f\tL\t%20.16f\n",
           n-1,Vn_1,n,Vn,L);
#endif
   // update queue events
   for (currEvent=myMgccEventQueue.ShowFirst(); currEvent!=NULL;
        currEvent=myMgccEventQueue.ShowNext()) {
      if ((currEvent->whichQueue==queue)&&
          (currEvent->type==MGCCSIM_DEPARTURE)&&
          (currEvent->myMgccEntity->blocked==MGCCSIM_NO)) {
#if MGCCSIM_DEBUG
         fprintf(MGCCSIM_OUT_FILE,"before\n");
         currEvent->Print();
#endif
         ΔT = now - currEvent->myMgccEntity->timeLastChange;
         // update time since last change
         currEvent->myMgccEntity->timeLastChange = now;
         // update position since last change & occurence time
         currEvent->myMgccEntity->lastPosition += Vn_1*ΔT;
         if (currEvent->myMgccEntity->lastPosition > L) {
            currEvent->myMgccEntity->lastPosition = L;
#if MGCCSIM_DEBUG
            fprintf(MGCCSIM_OUT_FILE,"warning: last position beyond L\n");
#endif
         }
         ΔSToGo = L - currEvent->myMgccEntity->lastPosition;
#if MGCCSIM_DEBUG
         if (currEvent->occurTime > now+ΔSToGo/Vn) {
            fprintf(MGCCSIM_OUT_FILE,"warning: entity acually not delayed:\n");
            fprintf(MGCCSIM_OUT_FILE,"\toccurTime\t%20.16f\toccurTime'\t%20.16f\n",
                    currEvent->occurTime, now+ΔSToGo/Vn);
            fprintf(MGCCSIM_OUT_FILE,"\tV(%d)\t%20.16f\tV(%d)\t%20.16f\tL\t%20.16f\n",
                    n-1,Vn_1,n,Vn_1,L);
            fprintf(MGCCSIM_OUT_FILE,"\tΔT\t%20.16f\tΔS\t%20.16f\n",
                    ΔT,Vn_1*ΔT);
            fprintf(MGCCSIM_OUT_FILE,"\tΔTToGo\t%20.16f\tΔSToGo\t%20.16f\n",
                    ΔSToGo/Vn,ΔSToGo);
            currEvent->Print();
//            exit(1);
         }
#endif
         currEvent->occurTime = now + ΔSToGo/Vn;
#if MGCCSIM_DEBUG
         fprintf(MGCCSIM_OUT_FILE, "after\n");
         currEvent->Print();
```

```
#endif
      } // end if
  } // end for
  // sort event queue
//  myMgccEventQueue.ReSort();
  return 0;
}
int MgccSimul::AdvanceST(int queue, double now) {
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"MgccSimul::AdvanceST(queue,now):\n");
#endif
  MgccEvent *currEvent;
  double Vn_1, Vn, L, ΔT, ΔSToGo;
  int n;
  // compute queue constants
  n = myMgccResource[queue].GetUsers();
  Vn_1 = myMgccResource[queue].service->GetV1()*
         myMgccResource[queue].service->Rate(n+1);
  if (n>0)
     Vn = myMgccResource[queue].service->GetV1()*
          myMgccResource[queue].service->Rate(n);
  else
     Vn = myMgccResource[queue].service->GetV1();
  L = myMgccResource[queue].service->GetEts1()*
      myMgccResource[queue].service->GetV1();
#if MGCCSIM_DEBUG
  fprintf(MGCCSIM_OUT_FILE,"V(%d)\t%20.16f\tV(%d)\t%20.16f\tL\t%20.16f\n",
          n+1,Vn_1,n,Vn,L);
#endif
  for (currEvent=myMgccEventQueue.ShowFirst(); currEvent!=NULL;
       currEvent=myMgccEventQueue.ShowNext()) {
      if ((currEvent->whichQueue==queue)&&
          (currEvent->type==MGCCSIM_DEPARTURE)&&
          (currEvent->myMgccEntity->blocked==MGCCSIM_NO)) {
#if MGCCSIM_DEBUG
        fprintf(MGCCSIM_OUT_FILE,"before\n");
        currEvent->Print();
#endif
        ΔT = now - currEvent->myMgccEntity->timeLastChange;
        // update time since last change
        currEvent->myMgccEntity->timeLastChange = now;
        // update position since last change & occurence time
        currEvent->myMgccEntity->lastPosition += Vn_1*ΔT;
        if (currEvent->myMgccEntity->lastPosition > L) {
           currEvent->myMgccEntity->lastPosition = L;
#if MGCCSIM_DEBUG
           fprintf(MGCCSIM_OUT_FILE,"warning: last position beyond L\n");
#endif
        }
        ΔSToGo = L - currEvent->myMgccEntity->lastPosition;
#if MGCCSIM_DEBUG
        if (currEvent->occurTime < now+ΔSToGo/Vn) {
           fprintf(MGCCSIM_OUT_FILE,"warning: entity acually not advanced:\n");
           fprintf(MGCCSIM_OUT_FILE,"\toccurTime\t%20.16f\toccurTime'\t%20.16f\n",
                   currEvent->occurTime, now+ΔSToGo/Vn);
           fprintf(MGCCSIM_OUT_FILE,"\tV(%d)\t%20.16f\tV(%d)\t%20.16f\tL\t%20.16f\n",
                   n+1,Vn_1,n,Vn,L);
           fprintf(MGCCSIM_OUT_FILE,"\tΔT\t%20.16f\tΔS\t%20.16f\n",
                   ΔT, Vn_1*ΔT);
           fprintf(MGCCSIM_OUT_FILE,"\tΔTToGo\t%20.16f\tΔSToGo\t%20.16f\n",
                   ΔSToGo/Vn,ΔSToGo);
           currEvent->Print();
        }
#endif
        currEvent->occurTime = now + ΔSToGo/Vn;
#if MGCCSIM_DEBUG
        fprintf(MGCCSIM_OUT_FILE, "after\n");
        currEvent->Print();
#endif
      } // end if
  } // end for
  // sort event queue
//  myMgccEventQueue.ReSort();
  return 0;
}
#endif
```

Código A.3: Cmusr.c

```
//
// Purpose:
//     to implement congestions models
//     FOR A SPECIFIC CLASS OF USERS
//
// Authors:
//    Paula de Campos Oliveira
//    Frederico R. B. Cruz
//    Departamento de Estatistica
//    Universidade Federal de Minas Gerais
//    Brazil
//    E-mail: pcampol@yahoo.com.br,fcruz@est.ufmg.br
//
//
// Version:
//    5.0
//
// Date:
//    Jul/2010
//
#ifndef CMUSR_CPP
#define CMUSR_CPP
//
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cm.cpp"
//
// values specially for pedestrian flows
//
//double CMUSR_maxDens   = 5.0;
//double CMUSR_maxSpeed = 1.5;
//double CMUSR_aDens     = 2.0;
//double CMUSR_aSpeed    = 0.64;
//double CMUSR_bDens     = 4.0;
//double CMUSR_bSpeed    = 0.25;
//
// but there are other possibilities ...
//
//double CMUSR_maxDens = 5.0;
//double CMUSR_maxSpeed = 0.71;
//double CMUSR_aDens = 2.0;
//double CMUSR_aSpeed = 0.31;
//double CMUSR_bDens = 4.0;
//double CMUSR_bSpeed = 0.12;
//
// values specially for vehicular flows (Antonio Carlos)
//
//double CMUSR_maxDens   = 118.0;
//double CMUSR_maxSpeed = 49.0;
//double CMUSR_aDens     = 20.0;
//double CMUSR_aSpeed    = 40.0;
//double CMUSR_bDens     = 40.0;
//double CMUSR_bSpeed    = 33.0;
//
// but there are other possibilities ...
//
//double CMUSR_maxDens   = 118.0;
//double CMUSR_maxSpeed = 49.0;
//double CMUSR_aDens     = 20.0;
//double CMUSR_aSpeed    = 40.0;
//double CMUSR_bDens     = 40.0;
//double CMUSR_bSpeed    = 33.0;
//
double CMUSR_maxDens   = 200.0;
double CMUSR_maxSpeed = 62.5;
double CMUSR_aDens     = 20.0;
double CMUSR_aSpeed    = 48.0;
double CMUSR_bDens     = 140.0;
double CMUSR_bSpeed    = 20.0;
//
// WARNING
//
// these variables must be eventually set up for each server:
//
// double length; // server length
// double width;  // server width
//
// generic velocity congestion model for users
//
class CMGenUsr {
public:
  // default constructor
  CMGenUsr(void) {}
  // destructor
  virtual ¬CMGenUsr(void) {}
  virtual void SetCorridor(double length, double width) = 0;
};
//
// linear velocity congestion model for users
//
class CMLinUsr: public CMLin, public CMGenUsr {
public:
  // default constructor
  CMLinUsr(void): CMLin(), CMGenUsr() {}
  // destructor
  ¬CMLinUsr(void) {}
  void SetCorridor(double length, double width);
```

```
};
//
// exponential velocity congestion model for users
//
class CMExpUsr: public CMExp, public CMGenUsr {
public:
  // default constructor
  CMExpUsr(void): CMExp(), CMGenUsr() {}
  // destructor
  ¬CMExpUsr(void) {}
  void SetCorridor(double length, double width);
};
//
// implementation
//
void CMLinUsr::SetCorridor(double length, double width) {
#if CMUSR_DEBUG
  fprintf(CM_OUT_FILE, "CMLinUsr::SetCorridor(double,double):\n");
#endif
  int theCap = (int)floor(CMUSR_maxDens * length * width);
  double theEts1 = length/CMUSR_maxSpeed;
  CMGen::SetC(theCap);
  SetEts1(theEts1);
  SetV1(CMUSR_maxSpeed);
}
void CMExpUsr::SetCorridor(double length, double width) {
#if CMUSR_DEBUG
  fprintf(CM_OUT_FILE, "CMExpUsr::SetCorridor(double,double):\n");
#endif
  CMExpUsr::SetShapeForm(CMUSR_maxDens, CMUSR_aDens, CMUSR_aSpeed,
                         CMUSR_bDens, CMUSR_bSpeed);
  int theCap = (int)floor(CMUSR_maxDens * length * width);
  double theEts1 = length/CMUSR_maxSpeed;
  SetC(theCap);
  SetEts1(theEts1);
  SetV1(CMUSR_maxSpeed);
}
#endif
```

Código A.4: Cm.c

```
//
// Purpose:
//    to implement linear and exponential congestion models
//
// Authors:
//    Paula de Campos Oliveira
//    Frederico R. B. Cruz
//    Departamento de Estatistica
//    Universidade Federal de Minas Gerais
//    Brazil
//    E-mail: pcampol@yahoo.com.br,fcruz@est.ufmg.br
//
//
// Version:
//    5.0
//
// Date:
//    Jul/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) {}
  // destructor
  ¬CMExp(void) {}
  void SetShapeForm(double maxDens, double aDens, double aSpeed,
                    double bDens, double bSpeed);
```

```
  double Rate(int customers);
};
//
// implementation
//
double CMLin::Rate(int customers) {
  if ((cap≤0)||(customers<0)||(customers>cap)) {
     fprintf(CM_ERR_FILE, "CMLin::Rate(int): ERROR: parameter out of range\n");
     exit(1);
  }
#if CM_DEBUG
  fprintf(CM_OUT_FILE, "CMLin::Rate(int):\t%20.18f\n",
         ((double)(cap+1-customers)/cap));
#endif
  return((double)(cap+1-customers)/cap);
}
void CMExp::SetShapeForm(double theMaxDens, double theADens, double theASpeed,
                         double theBDens, double theBSpeed) {
#if CM_DEBUG
   fprintf(CM_OUT_FILE,"CMExp::SetShapeForm:\n");
#endif
   maxDens = theMaxDens;
   aDens = theADens;
   aSpeed = theASpeed;
   bDens = theBDens;
   bSpeed = theBSpeed;
   status = !CM_EVALUATED;
}
double CMExp::Rate(int customers) {
  if (status != CM_EVALUATED) {
          double a=aDens*cap/maxDens;
        double b=bDens*cap/maxDens;
        gamma=log( log(aSpeed/maxSpeed)/log(bSpeed/maxSpeed)) / log((a-1)/(b-1));
//       if (errno) {
//          fprintf(CM_OUT_FILE, "CMExp::Rate(%d):\tgamma\tlog(log(%f/%f)/log(%f/%f))/log((%f-1)/(%f-1)\
   t%f\n",
//               customers, aSpeed, maxSpeed, bSpeed, maxSpeed, a, b, gamma);
//          exit(1);
//       }
        beta=(a-1)/pow(log(maxSpeed/aSpeed),(1/gamma));
//       if (errno) {
//          fprintf(CM_OUT_FILE, "CMExp::Rate(%d):\tbeta\t(%f-1)/pow(log(%f/%f),(1/%f))\t%f\n",
//               customers, a, maxSpeed, aSpeed, gamma, beta);
//          exit(1);
//       }
        status = CM_EVALUATED;
#if CM_DEBUG
        fprintf(CM_OUT_FILE, "CMExp::Rate(%d):\n\tgamma\t%20.18f\tbeta\t%20.18f\n",
                customers, gamma, beta);
        fprintf(CM_OUT_FILE, "CMExp::Rate(%d):\texp(-((%d-1)/%f)^%f\t%f\n",
                customers, customers, beta, gamma, exp(-pow((customers-1)/beta,gamma)));
#endif
  }
  if ((cap≤0)||(customers<0)||(customers>cap)) {
     fprintf(CM_ERR_FILE, "CMExp::Rate(%d):\tERROR: parameter out of range\n",
            customers);
     exit(1);
  }
  return(exp(-pow((customers-1)/beta,gamma)));
}
#endif
```

Código A.5: Randx.c

```
/***********************************************************************
*                                                                      *
* Purpose:                                                             *
*     randX is a library for random number generation.                 *
*     The integer randXseed should be initialized to an arbitrary      *
* integer prior to the first call to the desired function. The calling *
* program should not alter the value of randXseed between subsequent   *
* calls.                                                               *
*                                                                      *
* Author:                                                              *
*     Frederico R. B. Cruz                                             *
*     Departamento de Estatistica                                      *
*     Universidade Federal de Minas Gerais                             *
*     Brazil                                                           *
*     E-mail: fcruz@est.ufmg.br                                        *
*                                                                      *
* Version:                                                             *
*     1.00                                                             *
*                                                                      *
* Date:                                                                *
*     March/2002                                                       *
*                                                                      *
***********************************************************************/
#ifndef RANDX_C
#define RANDX_C
#include <math.h>
#define RANDX_PI 3.14159265358979323846
#define RANDX_EE 2.71828182845904523536
/* uniform [0,1]                                                       */
float  rUnif(int *randXseed);
float  rUnif2(unsigned long *randXseed1, unsigned long *randXseed2);
/* normal(mu=0,sd=1)                                                   */
float  rNorm(int *randXseed);
/* exponetial(lambda)                                                  */
float  rExpo(int *randXseed, float lambda);
/* gamma(alpha)                                                        */
double rGamma(int *randXseed, double alpha);
/* beta(alpha,beta)                                                    */
double rBeta(int *randXseed, double alpha, double beta);
/***********************************************************************
*                                                                      *
*        The rUnif function is a uniform random number generator based *
*  on theory and suggestions given in Forsythe, G.E., Malcolm, M.A., & *
*  Moter, C.B. Computer Methods For Mathematical Computations,         *
*  Prentice-Hall, 1977.                                                *
*        The integer randXseed should be initialized to an arbitrary   *
* integer prior to the first call to rUnif. The calling program should *
* not alter the value of randXseed between subsequent calls to rUnif.  *
* Values of rUnif will be returned in the interval (0,1).              *
*                                                                      *
***********************************************************************/
float rUnif(int *randXseed) {
    /* initialized data */
    static int m2 = 0;
    static int two = 2;
    /* local variables */
    static int m;
    static float s;
    static float halfm;
    static int a, c, mc;
    /* if first entry then ... */
    if (m2 == 0) {
        /* compute machine integer word length */
    m = 1;
    do {
            m2 = m;
            m = two * m2;
    } while( m > m2 );
    halfm = (float) m2;
        /* compute multiplier and increment for linear   */
        /* congruential method                           */
        a = ( (int) ( halfm * atan(1.) / 8.) << 3 ) + 5;
        c = ( (int) ( halfm * ( 0.5 - sqrt(3.) / 6.)) << 1 ) + 1;
        mc = m2 - c + m2;
        /*   s is the scale factor for converting to floating point */
    s = 0.5 / halfm;
    }
    /* compute next random number */
    *randXseed *= a;
    /* the following statement is for computers which do not */
    /* allow integer overflow on addition                    */
    if (*randXseed > mc)
        *randXseed = *randXseed - m2 - m2;
    *randXseed += c;
    /* the following statement is for computers where the word */
    /* length for addition is greater than for multiplication  */
    if (*randXseed / 2 > m2)
        *randXseed = *randXseed - m2 - m2;
    /* the following statement is for computers where integer    */
    /* overflow affects the sign bit                             */
    if (*randXseed < 0)
        *randXseed = *randXseed + m2 + m2;
    return ( (float) (*randXseed) * s );
}
/***********************************************************************
*                                                                      *
*        George Marsaglia's uniform random number generator            *
*        The integers randXseed1 and randXseed2 should be initialized  *
```

```
* to an arbitrary integer prior to the first call to rUnif2. The          *
* calling program should not alter these values between subsequent        *
* calls to rUnif.                                                          *
*                                                                          *
**************************************************************************/
float  rUnif2(unsigned long *randXseed1, unsigned long *randXseed2) {
  static unsigned long s1new, s2new;
  s1new = ((*randXseed1=36969*(*randXseed1&65535)+(*randXseed1>>16))<<16);
  s2new = ((*randXseed2=18000*(*randXseed2&65535)+(*randXseed2>>16))&65535);
  return ((s1new+s2new)*2.32830643708E-10);
}
/**************************************************************************
#include <stdio.h>
#include "uni.c"
int main() {
  int replic = 1000;
  int randXseed = 123456;
  unsigned long randXseed1=362436069, randXseed2=521288629;
  int i;
  for (i=0; i<replic; i++){
      fprintf(stdout, "%20.18f\t%20.18f\t%20.18f\n",
              rUnif(&randXseed), rUnif2(&randXseed1,&randXseed2), UNI);
  }
  return 0;
}
**************************************************************************/
/**************************************************************************
*                                                                          *
*         The rNorm function is a normal random number generator.          *
*         The integer randXseed should be initialized to an arbitrary      *
* integer prior to the first call to rNorm. The calling program should     *
* not alter the value of randXseed between subsequent calls to rNorm.      *
* Values of rNorm will be returned following N(0,1).                       *
*                                                                          *
**************************************************************************/
float rNorm(int *randXseed) {
  static float y1, y2;
  while(1) {
      y1=-log(rUnif(randXseed));
      y2=-log(rUnif(randXseed));
      if (y2-(y1-1)*(y1-1)/2 ≥ 0) {
         if (rUnif(randXseed) > 0.5)
            return -y1;
         else
            return y1;
      }
  }
}
/**************************************************************************
#include <stdio.h>
int main() {
 int replic = 100000;
 float mu = 0;
 float sigma = 1;
 int seed = 123456;
 float numb;
 float sum, sum2;
 int i;
 sum = 0.0;
 sum2 = 0.0;
 for (i=0; i<replic; i++){
   numb = mu + rNorm(&seed)*sigma;
   sum += numb;
   sum2 += (numb*numb);
 }
 printf("X ¬ N(%f,%f) had E(x)=%f and Var(x)=%f over %d replics.\n",
         mu, sigma*sigma, sum/replic , (sum2-(sum*sum)/replic)/replic , replic);
 return 0;
}
**************************************************************************/
/**************************************************************************
*                                                                          *
*         The rExpo function is an exponential random number generator.    *
*         The integer randXseed should be initialized to an arbitrary      *
* integer prior to the first call to erand. The calling program should     *
* not alter the value of randXseed between subsequent calls to erand.      *
* Values of erand will be returned following E(lambda).                    *
*                                                                          *
**************************************************************************/
float rExpo(int *randXseed, float lambda) {
  return (-log(rUnif(randXseed))/lambda);
}
/**************************************************************************
#include <stdio.h>
int main() {
 int replic = 10000;
 float lambda = 2;
 int seed = 123456;
 float numb;
 float sum = 0.0;
 float sum2 = 0.0;
 int i;
 int cont;
 for (i=0; i<replic; i++){
   numb  = rExpo(&seed,lambda);
   sum += numb;
   sum2 += (numb*numb);
 }
 printf("X ¬ E(%f) had E(x)=%f^-1 and Var(x)=%f^-2 over %d replics.\n",
```

```
        lambda, 1/(sum/replic), 1/sqrt((sum2-(sum*sum)/replic)/replic), replic);
 sum = 0.0;
 cont = 0;
 while (sum ≤ replic){
   numb  = rExpo(&seed,lambda);
   sum += numb;
   cont++;
 }
 printf("There were %d events over %d time units.\n", cont, replic);
 return 0;
}
*************************************************************************/
/*************************************************************************
*                                                                       *
*          The rGamma function is a gamma random number generator.      *
*          The integer randXseed should be initialized to an arbitrary  *
* integer prior to the first call to erand. The calling program should  *
* not alter the value of randXseed between subsequent calls to erand.   *
* Values of erand will be returned following G(alpha).                  *
*                                                                       *
*************************************************************************/
double rGamma(int *randXseed, double alpha) {
   static double r1,r2,aa,x,w,c1,c2,c3,c4,c5;
   if (alpha≤0.)
      return 0.;
   if (alpha == 1.)
      return rExpo(randXseed,1.);
   if (alpha<1) {
      aa=(alpha+RANDX_EE)/RANDX_EE;
      do {
         r1=rUnif(randXseed);
         r2=rUnif(randXseed);
         if(r1>1./aa) {
            x = -log(aa*(1.-r1)/alpha);
            if (r2<pow(x,(alpha-1.)))
               return x;
         }
         else {
            x = pow((aa*r1),(1./alpha));
            if (r2<exp(-x))
               return x;
         }
      } while(1);
   }
   else {
      c1=alpha-1;
      c2=(alpha-1./(6.*alpha))/c1;
      c3=2./c1;
      c4=c3+2.;
      c5=1./sqrt(alpha);
      do {
         do {
            r1=rUnif(randXseed);
            r2=rUnif(randXseed);
            if (alpha>2.5)
               r1=r2+c5*(1.-1.86*r1);
         } while(r1≤0 || r1 ≥ 1);
         w=c2*r2/r1;
         if (c3*r1+w+1/w ≤ c4)
            return c1*w;
         if (c3*log(r1)-log(w)+w<1)
            return c1*w;
      } while (1);
   }
}
/*************************************************************************
#include <stdio.h>
int main() {
 int replic = 10000;
 int seed = 123456;
 int i;
 double alpha = 1.5;
 for (i=0; i<replic; i++){
   fprintf(stdout, "%f\n", rGamma(&seed,alpha));
 }
 return 0;
}
*************************************************************************/
/*************************************************************************
*                                                                       *
*          The rBeta function is a beta random number generator.        *
*          The integer randXseed should be initialized to an arbitrary  *
* integer prior to the first call to erand. The calling program should  *
* not alter the value of randXseed between subsequent calls to erand.   *
* Values of erand will be returned following B(alpha,beta).             *
*                                                                       *
*************************************************************************/
double rBeta(int *randXseed, double alpha, double beta) {
   double r1;
   if (alpha ≤0. || beta ≤ 0.)
      return 0.;
   r1=rGamma(randXseed,alpha);
   return r1/(r1+rGamma(randXseed,beta));
}
/*************************************************************************
#include <stdio.h>
int main() {
 int replic = 10000;
 int seed = 123456;
```

```
 int i;
 double alpha = 2.0;
 double beta = 3.0;
 for (i=0; i<replic; i++){
   fprintf(stdout, "%f\n", rBeta(&seed,alpha,beta));
 }
 return 0;
}
********************************************************************/
#endif
```

Apêndice B
Arquivo de Entrada

Arquivo de entrada B.1: Topologia Série

```
Nodes
3
Arc  Orig       Dest       Prob
1      1           2           1.00
2      2           3           1.00
Node       Serv       Length   Width      Lambda
1              2            1.0       1.0       1000.0
2              2            1.0       1.0       0.000
3              2            1.0       1.0       0.000
Exit  Nodes
1
2
3
Cel  Node
1     1
2     2
3     3
```

Apêndice C
Testes de Kolmogorov-Smirnov

Testes de Kolmogorov-Smirnov realizados para a variável *tempo entre partidas* para a topologia série, divisão, fusão e as quatro configurações mistas apresentadas.

As Tabelas C.1, C.3, C.5, C.7, C.9, C.11 e C.13, apresentam os valores-p para as topologias série, divisão, fusão, mistaI, mista II, mista III e mista IV, respectivamente, relativos ao teste de Kolmogorov-Smirnov em relação à distribuição Exponencial, Exponencial com 2 parâmetros, Gamma, Gamma com 3 parâmetros, Gamma Generalizada e Gamma Generaliza com 4 parâmetros.

As Tabelas C.2, C.4, C.6, C.8, C.10, C.12 e C.14, apresentam a conclusão para as topologias série, divisão, fusão, mistaI, mista II, mista III e mista IV, respectivamente, relativos ao teste de Kolmogorov-Smirnov, utilizando um nível de significância de 1%, em relação à distribuição Exponencial, Exponencial com 2 parâmetros, Gamma, Gamma com 3 parâmetros, Gamma Generalizada e Gamma Generaliza com 4 parâmetros.

C.1 Topologia Série

Tabela C.1: Valores-p para topologia série

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	0,71580	0,72508	0,99564	0,96007	0,96807	0,94765
	2	0,56125	0,57019	0,99871	0,87052	0,94437	0,94158
	3	1,48688	0,49517	0,99090	0,67441	0,79889	0,83866
2.000	1	0,78502	0,79892	0,94929	0,94715	0,94682	0,90198
	2	0,96845	0,97338	0,98231	0,98477	0,97069	0,98351
	3	0,93051	0,93914	0,99701	0,99438	0,99073	0,97610
4.000	1	0,73771	0,80643	0,73740	0,91438	0,90766	0,97218
	2	0,89495	0,87676	0,51012	0,79332	0,78612	0,73147
	3	0,56241	0,63689	0,61260	0,63577	0,87448	0,74314
8.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00001
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00013
	3	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
16.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00401
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00035
	3	0,00000	0,00000	0,00000	0,00000	0,00000	0,00001

Tabela C.2: Comparação com nível de significância de 1% para topologia série

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
	3	não	não	não	não	não	não
2.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
	3	não	não	não	não	não	não
4.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
	3	não	não	não	não	não	não
8.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	3	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
16.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	3	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**

C.2 Topologia Divisão

Tabela C.3: Valores-p para topologia divisão

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	0,71580	0,72508	0,99564	0,96007	0,96807	0,94765
	2	0,92047	0,91917	0,92844	0,93507	0,84383	0,89663
	3	0,93955	0,93809	0,73003	0,78402	0,74783	0,86186
2.000	1	0,78502	0,79892	0,94929	0,94715	0,94682	0,90198
	2	0,57331	0,57168	0,55798	0,56425	0,57274	0,54950
	3	0,68076	0,64727	0,71376	0,51633	0,72650	0,72405
4.000	1	0,73771	0,80643	0,73740	0,91438	0,90766	0,87218
	2	0,97294	0,95539	0,97905	0,98455	0,99301	0,99513
	3	0,67648	0,76984	0,81740	0,90794	0,67443	0,94155
8.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00001
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00035
	3	0,00000	0,00000	0,00000	0,00020	0,00000	0,19135
16.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00401
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,03862
	3	0,00000	0,00000	0,00000	0,00000	0,00000	0,00274

Tabela C.4: Comparação com nível de significância de 1% para topologia divisão

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
	3	não	não	não	não	não	não
2.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
	3	não	não	não	não	não	não
4.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
	3	não	não	não	não	não	não
8.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	3	**sim**	**sim**	**sim**	**sim**	**sim**	não
16.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	não
	3	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**

C.3 Topologia Fusão

Tabela C.5: Valores-p para topologia fusão

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	0,89073	0,88558	0,69050	0,90039	0,76394	0,87134
	2	0,33954	0,34356	0,88029	0,60051	0,66393	0,76949
	3	0,07752	0,07891	0,11495	0,40713	0,36073	0,67900
2.000	1	0,27128	0,26943	0,22272	0,39659	0,18285	0,92327
	2	0,62044	0,62706	0,59189	0,88269	0,70634	0,89213
	3	0,63484	0,93484	0,94280	0,99599	0,99451	0,98693
4.000	1	0,00219	0,00272	0,10831	0,05701	0,12153	0,00640
	2	0,20077	0,16600	0,01949	0,00546	0,17008	0,01352
	3	0,00002	0,00003	0,00014	0,00000	0,00002	0,00005
8.000	1	0,00000	0,00000	0,00002	0,00012	0,00002	0,00015
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00120
	3	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
16.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00038
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00049
	3	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000

Tabela C.6: Comparação com nível de significância de 1% para topologia fusão

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
	3	não	não	não	não	não	não
2.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
	3	não	não	não	não	não	não
4.000	1	**sim**	**sim**	não	não	não	**sim**
	2	não	não	não	**sim**	não	não
	3	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
8.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	3	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
16.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	3	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**

C.4 Topologia Mista I

Tabela C.7: Valores-p para topologia mista I

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	0,77584	0,77584	0,57267	0,79430	0,67270	0,94015
	2	0,60345	0,60345	0,30994	0,58898	0,40889	0,96550
2.000	1	0,67552	0,68758	0,99231	0,94055	0,96638	0,91292
	2	0,34896	0,34896	0,44280	0,40696	0,26957	0,73926
4.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2	0,00051	0,00051	0,00015	0,19052	0,05366	0,24216
8.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
16.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000

Tabela C.8: Comparação com nível de significância de 1% para topologia mista I

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
2.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
4.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	não	não	não
8.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
16.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**

C.5 Topologia Mista II

Tabela C.9: Valores-p para topologia mista II

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	0,76195	0,78044	0,67124	0,85306	0,89843	0,89056
	2	0,69202	0,69334	0,27677	0,52644	0,40621	0,50081
2.000	1	0,75118	0,75118	0,56569	0,90791	0,89453	0,84277
	2	0,43590	0,43590	0,15432	0,51653	0,38902	0,78390
4.000	1	0,01961	0,02166	0,11009	0,25347	0,24001	0,58850
	2	0,00001	0,00010	0,00000	0,00007	0,00001	0,00172
8.000	1	0,02458	0,02581	0,09699	0,16945	0,11901	0,83488
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
16.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000

Tabela C.10: Comparação com nível de significância de 1% para topologia mista II

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
2.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
4.000	1	não	não	não	não	não	não
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
8.000	1	não	não	não	não	não	não
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
16.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**

C.6 Topologia Mista III

Tabela C.11: Valores-p para topologia mista III

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	0,27113	0,27113	0,40912	0,49546	0,51701	0,51978
	2	0,59423	0,59811	0,82823	0,54474	0,54845	0,89634
2.000	1	0,64310	0,64310	0,27984	0,59869	0,64559	0,49506
	2	0,25412	0,25848	0,79229	0,74681	0,72112	0,96709
4.000	1	0,67097	0,67278	0,74841	0,63719	0,64844	0,89831
	2	0,31446	0,31446	0,41845	0,03035	0,03239	0,02989
8.000	1	0,43968	0,45722	0,79165	0,90724	0,97501	0,90357
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
16.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000

Tabela C.12: Comparação com nível de significância de 1% para topologia mista III

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
2.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
4.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
8.000	1	não	não	não	não	não	não
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
16.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**

C.7 Topologia Mista IV

Tabela C.13: Valores-p para topologia mista IV

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	0,94045	0,94139	0,94096	0,81948	0,97746	0,78753
	2	0,28937	0,28937	0,39320	0,35723	0,20066	0,90846
2.000	1	0,85848	0,85493	0,70934	0,88689	0,82098	0,93495
	2	0,71483	0,71483	0,86653	0,87374	0,88952	0,94473
4.000	1	0,98519	0,98519	0,99242	0,98937	0,98291	0,99129
	2	0,69673	0,69673	0,72869	0,90791	0,91608	0,92737
8.000	1	0,06505	0,06505	0,08874	0,00680	0,00769	0,01073
	2	0,00027	0,00027	0,00379	0,00124	0,00139	0,00153
16.000	1	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
	2	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000

Tabela C.14: Comparação com nível de significância de 1% para topologia mista IV

λ	Célula	Exp	Exp(2p)	Gam	Gam(3p)	Gam.Gen	Gam.Gen(4p)
1.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
2.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
4.000	1	não	não	não	não	não	não
	2	não	não	não	não	não	não
8.000	1	não	não	não	**sim**	**sim**	não
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
16.000	1	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**
	2	**sim**	**sim**	**sim**	**sim**	**sim**	**sim**

Apêndice D
Comparação do Número de Pistas

Comparação da variável tempo de serviço nas células em relação ao número de pistas do trecho. As Figuras D.1 e D.2 apresentam os resultados para a topologia série. As Figuras D.3 e D.4 apresentam os resultados para a topologia divisão. Já as Figuras D.5 e D.6 apresentam os resultados para a topologia fusão. E, por fim, as Figuras D.7 e D.8 apresentam os resultados para a topologia mista I.

D.1 Topologia Série

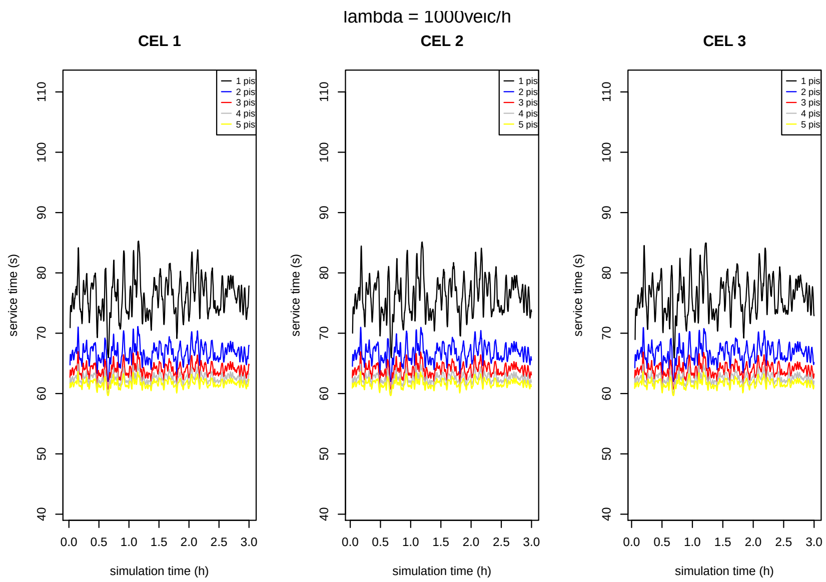


Figura D.1: Comparação do número de pistas para topologia série - $\lambda = 1000$

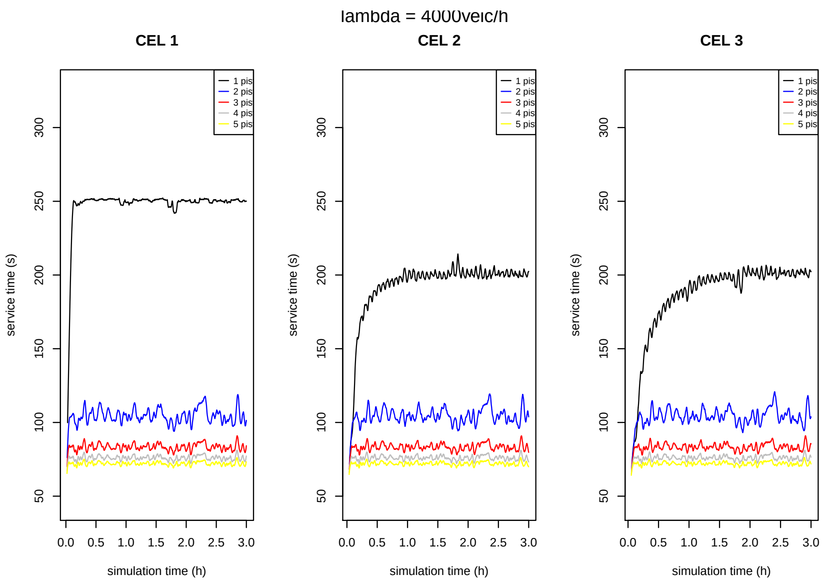


Figura D.2: Comparação do número de pistas para topologia série - $\lambda = 4000$

D.2 Topologia Divisão

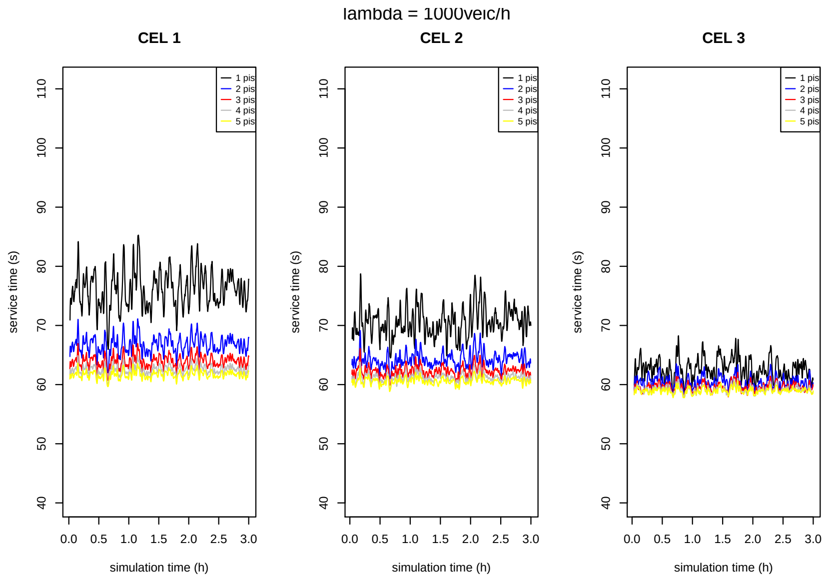


Figura D.3: Comparação do número de pistas para topologia divisão - $\lambda = 1000$

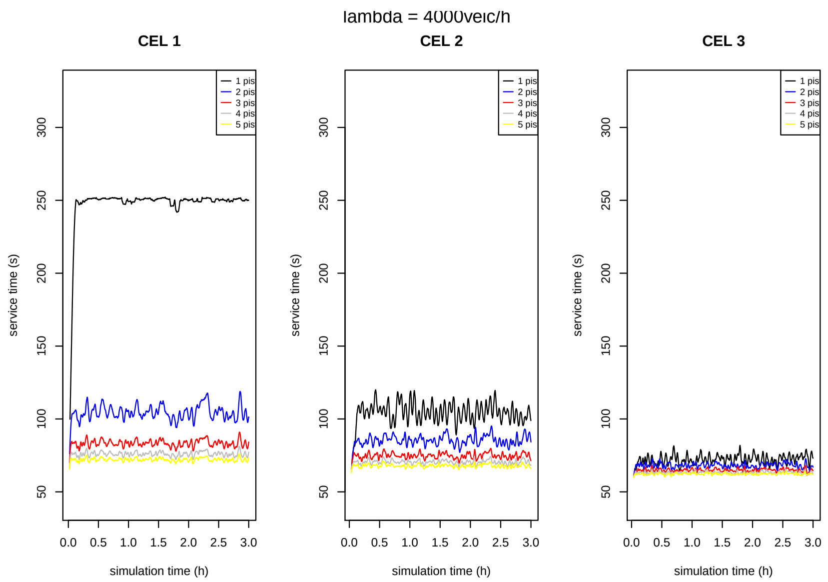


Figura D.4: Comparação do número de pistas para topologia divisão - $\lambda = 4000$

D.3 Topologia Fusão

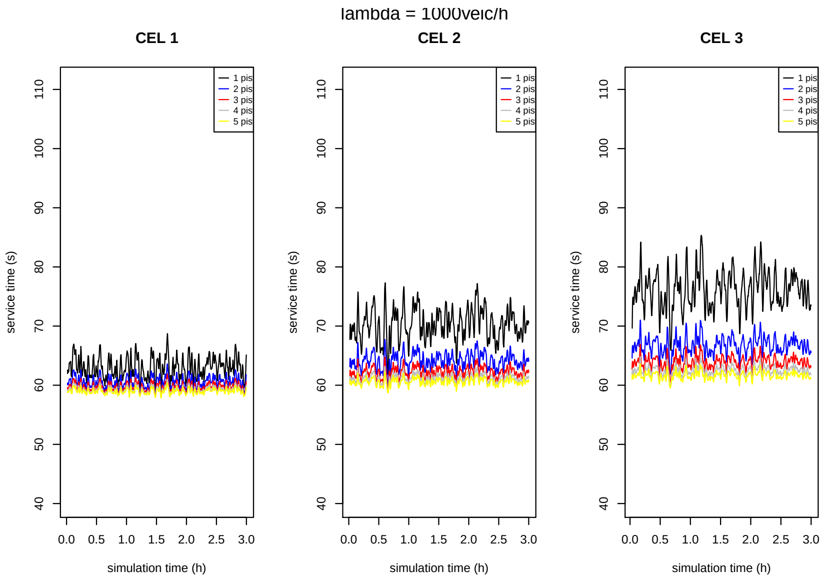


Figura D.5: Comparação do número de pistas para topologia fusão - $\lambda = 1000$

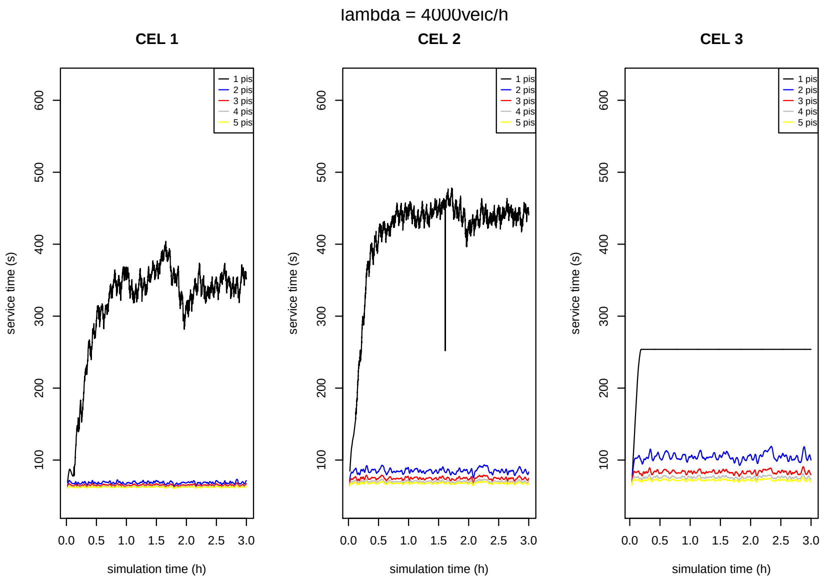


Figura D.6: Comparação do número de pistas para topologia fusão - $\lambda = 4000$

D.4 Topologia Mista I

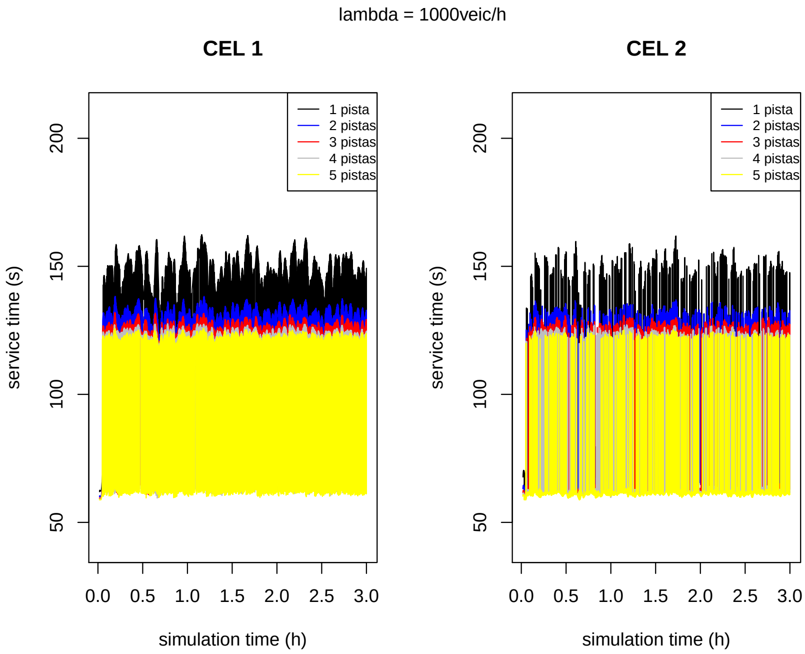


Figura D.7: Comparação do número de pistas para topologia mista I - $\lambda = 1000$

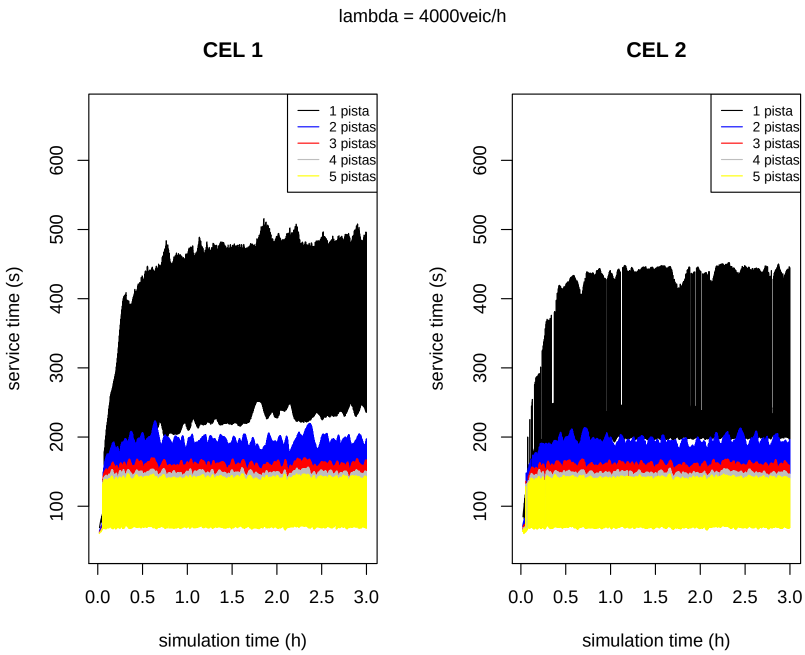


Figura D.8: Comparação do número de pistas para topologia mista I - $\lambda = 4000$