4Sensing - Documentação

July 4, 2011

1 Operação do Simulador

O simulador pode ser operado a partir do Eclipse ou Ant. Um bug no plugin Greclipse com tipos enumerados (`http://netbeans.org/bugzilla/show_bug.cgi?id=189275`) não permite compilar (inicialmente) o código no IDE. Eventualmente o erro desaparece. A versão mais recente do plugin Greclipse possivelmente já não tem este problema, mas só é suportado no Eclipse 3.6 que tem bloqueios frequentes em OS X.

As próximas secções documentam as tarefas Ant e scripts para operação do simulador.

1.1 Tarefas Ant

1.1.1 deploy

Compila o código e copia binários e configurações para pasta `bin`.

1.1.2 clean

Lista as classes compiladas (pastas `antbin` e `bin`)

1.1.3 run

Executa o script run.sh para lançar simulações - ver 1.2. A tarefa é configurada por três parâmetros definidos no início em `build.xml`.

1. `runid`: string que define o nome da experiência. É utilizada para definir o path para o ficheiro de output - ver secção 2
2. `runenv`: string que define o ambiente de execução. É utilizado para definir o ficheiro `runenv_[runenv].sh` a executar para definição dos parâmetros da JVM
3. `runtimes`: número de simulações a correr para cada setup

1.1.4 pack/packsrc

Tarefas utilizadas para gerar arquivos tgz - completo ou apenas source, respectivamente - para instalação num ambiente remoto. Antes do packaging, é verificado se o repositório svn está sincronizado - i.e., se o output de `svn status` é vazio. A verificação é utilizada em conjunto com um svn hook para garantir que os resultados de simulações são marcadas com a versão do código no repositório (o hook limita-se a criar o ficheiro /src/4sensing-ver.properties com a versão actual no repositório).

NOTA: o svn hook (ficheiro `util/post-commit`) actualmente não está instalado no svn da faculdade.

1.2 Execução da Simulação - run.sh

A simulação é lançada executando `java [classe de simulação]`, colocando no classpath todos os jars definidos na pasta `lib` e classes na pasta `bin`. É conveniente configurar a JVM com pelo menos 800Mb de heap.

O script `run.sh` foi criado para facilitar a execução de uma simulação, ou várias simulações em sequência. Recebe 3 argumentos que correspondem aos três parâmetros definidos em 1.1.3: `runid`, `runenv`, `runtimes`. O script define um ou mais setups - variável `SETUPS`, definida no início. No ciclo de execução (ciclo `while` na linha 43), define-se os simuladores a correr - por exemplo pode correr-se diferentes estratégias de distribuição em sequência (ver código comentado).

Os logs gerados por cada simulação (relevantes para debugging) são colocados na pasta `logs` com o seguinte nome:

```
[run name]_[data/hora]_[nome simulador]_[FQN da classe de setup]_ \
[número de sequência da simulação].log
```

O nome do simulador é definido em `run.sh` (na chamada à função `run`) - normalmente é uma string que descreve sumariamente o simulador e.g., `SUMO_Centralized` para o simulador SUMO utilizando o modelo centralizado.

Os resultados da simulação são colocados dentro na pasta `results` numa estrutura em árvore com a configuração seguinte:

```
[FQN da classe de setup]/run_[run name]_[data/hora]/[FQN da classe de simulação]
```

O conteúdo dos resultados é definido na classe de setup.

1.3 Integração com SUMO/TAPAS Cologne

Para utilizar os dados do TAPAS Cologne é necessário correr previamente a simulação SUMO para este cenário. A simulação é corrida uma única vez para uma dada parametrização, e o output é posteriormente utilizado para as simulações 4Sensing. Para correr uma simulação SUMO:

1. Expandir o arquivo com o cenário TAPAS Cologne 0.0.3
2. Copiar conteúdo da pasta `sumo`, no projecto 4Sensing, para a pasta TAPASCologne-0.0.3
3. Editar `TAPASCologne-0.0.3/4Sensing.cfg`, tag `<end value="x"/>`, para definir a duração da simulação. Os dados disponibilizados livremente cobrem 2 horas de simulação - de 21600s a 28800s
4. Editar `TAPASCologne-0.0.3/input_additional.add.xml` para configurar os dois tipos de output utilizados pelo 4Sensing:
   (a) Na tag `meandata-edge` [1] é definido a periodicidade para as estatísticas de segmentos (actualmente 5 minutos) e o nome do ficheiro de output
   (b) Na tag `vtyperobe`[2] é definida a periodicidade da amostragem de posição/velocidades para os veículos (actualmente 5 segundos) e o nome do ficheiro de output
5. Correr simulador: `sumo -c 4Sensing.cfg`
6. Copiar resultados da simulação para a raiz do projecto 4Sensing
7. Copiar modelo da rede viária para pasta `sumo` no projecto 4Sensing:
   `TAPASCologne-0.0.3/road/no_internal/koeln_bbox.net.xml`

2 Output e Gráficos

O output da simulação é definido na classe de setup - o método `setupCharts` devolve uma lista de especializações da classe `sensing.persistence.simsim.Metric` que são responsáveis por compilar métricas, a sua visualização em ambiente gráfico e a persistência em ficheiro.

No caso das simulações SUMO, o output é simplesmente a compilação dos resultados da query em ficheiro, resultando num ficheiro por simulação. O ficheiro contém uma linha por cada resultado, com valores separados por `tab`. Os valores registados para cada resultado são definidos no método `outputResult`, na chamada ao método `lineOut.newLine`. Por exemplo, em `SUMOTrafficSpeedSetup`:

```
lineOut.newLine([s.count,
        s.vCount,
        s.minSpeed * 3.6, ...])
```

Isto resulta num ficheiro `results_[sim].gpd` na pasta definida em 1.2, em que `sim` é o número de sequência da simulação. Para o setup `SUMOTrafficSpeedSetup` cada linha do ficheiro contém a seguinte informação:

1. Número de amostras agregadas no resultado
2. Número de veículos que contribuem amostras para o resultado
3. Velocidade mínima em Km/h, calculada pela tabela virtual

[1] Documentação em `http://sourceforge.net/apps/mediawiki/sumo/index.php?title=SUMO_OUTPUT_EDGELANE_TRAFFIC`
[2] Documentação em `http://sourceforge.net/apps/mediawiki/sumo/index.php?title=SUMO_OUTPUT_VTYPEPROBE`

4. Velocidade máxima em Km/h, calculada pela tabela virtual
5. Velocidade média em Km/h, calculada pela tabela virtual
6. Desvio padrão da velocidade em Km/h, calculada pela tabela virtual
7. Velocidade mínima em Km/h, utilizando os traces SUMO (não é utilizado)
8. Velocidade máxima em Km/h, utilizando os traces SUMO (não é utilizado)
9. Velocidade média em Km/h, utilizando os traces SUMO (não é utilizado)
10. Desvio padrão da velocidade em Km/h, utilizando os traces SUMO (não é utilizado)
11. Número de amostras disponíveis nos traces SUMO (não é utilizado)
12. Velocidade média - obtida através das estatísticas SUMO
13. Taxa de ocupação média, obtida através das estatísticas SUMO
14. Densidade média, obtida através das estatísticas SUMO
15. Número de amostras, obtida através das estatísticas SUMO
16. Velocidade máxima para o segmento, informação estática obtida no modelo de estradas do SUMO
17. Número de faixas do segmento, informação estática obtida no modelo de estradas do SUMO
18. Comprimento do segmento, informação estática obtida no modelo de estradas do SUMO
19. Presença de semáforos? 1 - Sim, 0 - Não, informação estática inferida do modelo de estradas do SUMO
20. Segmento totalmente contido na área da query? 1 - Sim, 0 - Não, informação estática inferida do modelo de estradas do SUMO
21. Identificador do segmento
22. Timestamp do resultado

2.1 Pós-Processamento

O formato de output descrito acima pode ser utilizado directamente para gerar gráficos usando gnuplot e.g., gráficos de dispersão. Para gráficos mais elaborados é necessário pré-processar os dados, pelo que foi implementado um conjunto de scripts Groovy para esse efeito. Os scripts específicos para a versão actual do SpeedSense i.e., extracção da velocidade média por segmento, estão no package `sensing.persistence.util.sumo.segmentspeed2`.

O processo normal para a criação de gráficos após a execução de simulações será o seguinte:

1. Processamento do output para acrescentar dados de referência relativa a 100% de participação - ver 2.1.2
2. Processamento do output do passo anterior para gerar ficheiros fonte para gnuplot - ver 2.1.3 e 2.1.4
3. Gerar e executar scripts gnuplot - exemplos em `graphs/segmentspeed2` na pasta do projecto

É necessário anteceder o passo 1 com a execução do script `SegmentRatingOut` para cálculo do *comprimento dinâmico* dos segmentos. Este passo é necessário apenas uma vez após a simulação SUMO.

2.1.1 sensing.persistence.util.sumo.SegmentRatingOut

Executado pela tarefa Ant `segrate`. Determina o comprimento dinâmico dos segmentos para uma área de pesquisa e tempo de simulação, utilizando como fonte os traces SUMO (output `vtypeprobe` definido em 1.3). No exemplo abaixo, é gerado o comprimento dinâmico para os primeiros 30 minutos de simulação.

```
String mapData = "koeln_bbox_net.xml"
String vProbeData = "sumocfg3_vtypeprobe_5s_nointernal.xml"
int endTime = 23400
...
Rectangle2D qArea = new Rectangle2D.Double(minLon, minLat, width, height);
String outFile = "segmentRating_1800.tsv"
```

2.1.2 sensing.persistence.util.sumo.segmentspeed2.RefData

Executado pela tarefa Ant `ss_refdata`. O script acrescenta a cada resultado da query os dados relativos a 100% de participação. Em particular:

1. Erro da velocidade média (a diferença entre a velocidade média do resultado e o mesmo valor para 100% de participação)
2. O comprimento dinâmico do segmento
3. Número de amostras agregadas no resultado - valor obtido com 100% de participação
4. Número de veículos que contribuem amostras para o resultado - valor obtido com 100% de participação
5. Velocidade média - valor obtido com 100% de participação
6. Desvio padrão da velocidade - valor obtido com 100% de participação

No inicio do scipt é definida a localização dos dados das simulações e a pasta para output

```
String setupName = "segmentspeed2.SUMOTrafficSpeedSetup_10m"
String baseRunName = "run_segspeed2_10m08-06-2011-13-16-08"
String baseSimId = "0"
String runName = "run_segspeed2_10m08-06-2011-13-16-08"
String simId = "0"
String segRating = "segmentRating_1800.tsv"
String outDir = "results/segspeed2/rateresults"
def rates = [1,2,3,5,7,10,100]
```

- `setupName` refere a classe setup, sem o número que identifica a percentagem de participação. O valor será prefixado com: `sensing.persistence.simsim.speedsense.sumo.setup.`
- `baseRunName` é o nome do run onde se encontram os dados de referência - 100% de participação. Normalmente será igual a `runName`
- baseSimId é o número de sequência da simulação para os dados de referência
- `runName` é o nome do run onde se encontram os dados a processar
- baseSimId é o número de sequência da simulação a processar
- `outDir` é a pasta de saída
- `rates` define as percentagens de participação a considerar - usado em conjunto com o prefixo `setupName`

É criado na pasta `outDir` um ficheiro para cada percentagem de participação (rate) com o nome `[setupName]_err_[rate].gpd`.

2.1.3 sensing.persistence.util.sumo.segmentspeed2.Graph

Executado pela tarefa Ant `ss_graph`. O objectivo do script é gerar dados processados, num formato apropriado para gnuplot, para criar vários tipos de gráficos a partir da classificação dos resultados - e.g., ver figuras 1, 2 e 3. O input do script será o conjunto de ficheiros gerados pelo script `RefData`.

No script é usada uma função de classificação que separa os resultados em diferentes classes (bins) e para cada classe é calculado um valor, através de uma função de sumarização, que representa os dados englobados. Por exemplo para calcular o erro médio em Km/h, relativo a 100% de participação, para diferentes classes de racio de velocidades (velocidade média/velocidade máxima) - figura 1 - é usada a função de classificação `binfSpeedRate` e a função de sumarização `errorKmh`. Para o gráfico da figura 2, é usada a função de sumarização `sumoErrorKmh`.

A parametrização é definida no início do script:

```
String inDir = "results/segmentspeed2/rateresults"
String setupName = "segmentspeed2.SUMOTrafficSpeedSetup"
String metricName = "hist_speedrate_errorkmh_all"
String outDir = "results/segmentspeed2/ss_error_2"
rates = [1,2,3,5,7,10,50,100]
```

- `inDir` identifica a pasta onde estão os resultados - corresponde a `outdir` do script `RefData`

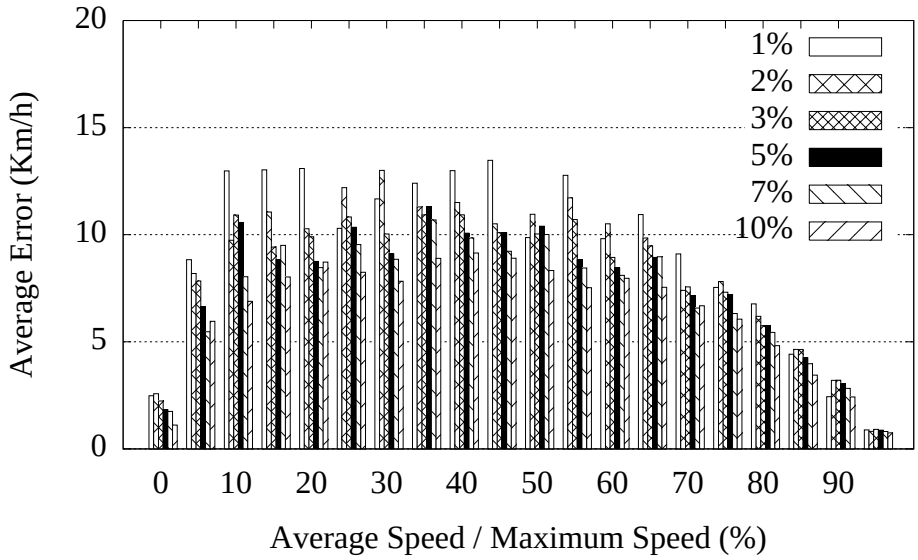

Figure 1: Relação entre o racio de velocidades e o erro médio durante uma simulação

- `outDir` define pasta onde serão guardados os resultados do script
- `setupName` o nome da classe setup, será igual ao valor `setupName` definido em `RefData`
- `metricName` é o nome da métrica - utilizado para gerar os nomes dos ficheiros de output
- `rates` define as percentagens de participação a considerar - usado em conjunto com o prefixo `setupName` para obter os ficheiros fonte

O script pré-define um conjunto de funções de classificação/sumarização que poderá ser aumentado com novas funções. Para configurar o script com as funções pretendidas, altera-se a chamada à função `run` (linha 169), substituindo os identificadores `binfSpeedRate` e `errorKmh` pelas funções pretendidas.

```
def meanerror = rates.collect{[it, run(it, {vals-> true}, binfSpeedRate, errorKmh, outfStats )] }
```

Pode também ser definida uma função de filtragem - passada como segundo argumento da função `run`. Por exemplo para utilizar apenas dados relativos a segmentos com sinais, utiliza-se a função: `{vals -> vals.edgetl.equals("1")}`. Esta é a base para criar o gráfico da figura 3.

Os dados para cada percentagem de participação (rate) são processados e os resultados guardados na pasta de output em ficheiros individuais com o nome `ss_[metricName]_[rate].gpd`. Estes ficheiros - que são a base do gráfico das figura 1 e 3 - contém uma linha por cada classe com os seguintes valores: o valor da classe (e.g., 10 representa os racios de velocidade no intervalo [10,20[), o valor de sumarização, a frequência da classe e os valores mínimos e máximos para a funcão de sumarização.

Para além dos ficheiros por percentagem de participação, é gerado um ficheiro com os dados "globais" (`ss_[metricName].gpd`) i.e., independente de classificação - a função de sumarização é aplicada à totalidade dos resultados para cada rate. Para o exemplo da figura 2, o ficheiro irá conter, para cada percentagem de participação, o erro médio global em Km/h.

Na pasta `graphs/segmentspeed2` do projecto encontram-se scripts gnuplot para gerar os gráficos usados como exemplo:

- `ss_speedrate_errorkmh.gp` para o gráfico na figura 1
- `ss_hist_sumoerror.gp` para o gráfico na figura 2
- `ss_hist_speedrate_100.gp` para o gráfico na figura 3

2.1.4 sensing.persistence.util.sumo.segmentspeed2.CoverageVsErrorGraph

Executado pela tarefa Ant `ss_coverage`. O objectivo do script é gerar os dados necessários para a criação de gráficos de cobertura com gnuplot. A configuração é definida no início do script:

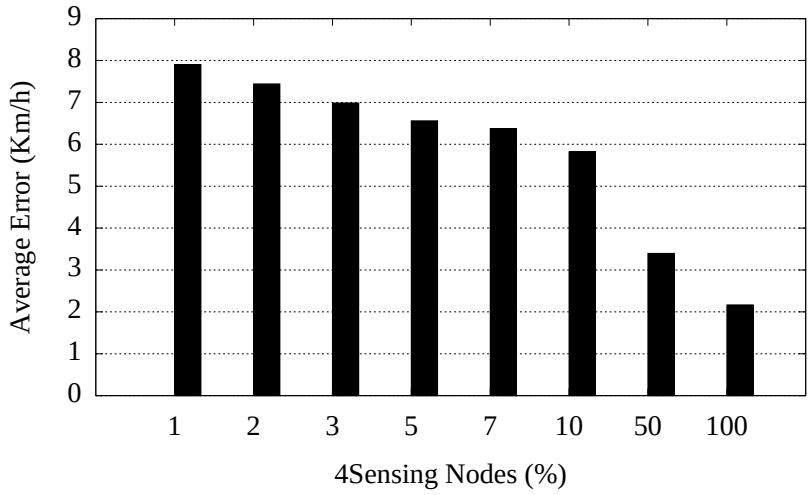

Figure 2: Erro médio relativo às estatísticas do SUMO

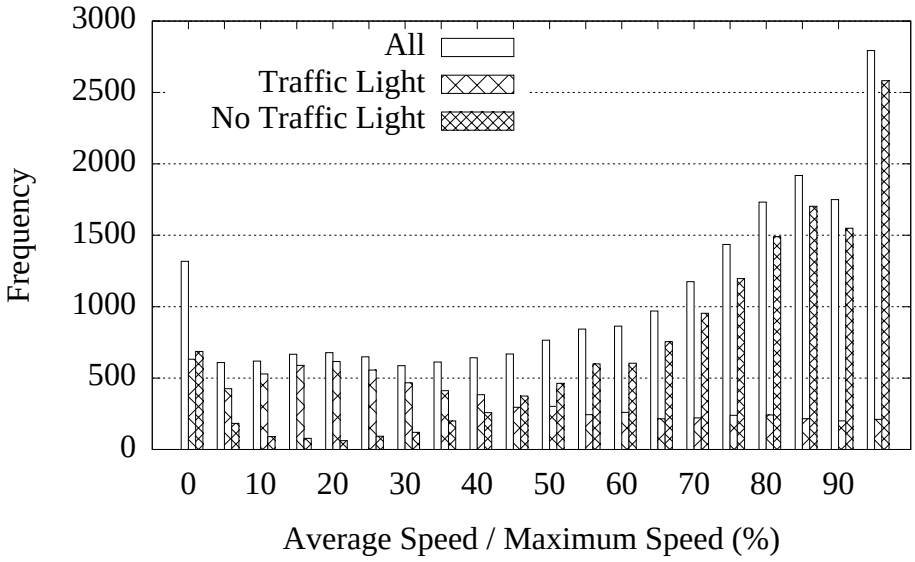

Figure 3: Ocorrência dos diferentes racios de velocidade durante uma simulação - 100% veículos 4Sensing

```
String inDir = "results/segspeed2/rateresults"
String setupName = "segmentspeed2.SUMOTrafficSpeedSetup_10m"
String metricName = "dinamic_length_coverage"
String outDir = "results/segspeed2/ss_error_2"
String segRating = "segmentRating_1800.tsv"
def rates = [1,2,3,5,7,10]
```

- `inDir` identifica a pasta onde estão os resultados - corresponde a `outdir` do script `RefData`
- `setupName` o nome da classe setup, será igual ao valor `setupName` definido em `RefData`
- `metricName` é o nome da métrica - utilizado para gerar os nomes dos ficheiros de output
- `outDir` define a pasta onde serão guardados os resultados
- `segRating` é o nome do ficheiro com os comprimentos dinâmicos por segmento, gerados pelo script `SegmentRatingOut` - ver 2.1.1
- `rates` define as percentagens de participação a considerar

Os dados para cada percentagem de participação (rate) são processados e os resultados guardados na pasta de output em ficheiros individuais com o nome `ss_[metricName]_[rate].gpd`. Em cada ficheiro, a primeira coluna representa a classe de erro em Km/h. As restantes 10 colunas contém a taxa de cobertura para o nível de erro correspondente, em que

cada coluna representa um racio de velocidades (de 10 a 100, com intervalo de 10). Estes ficheiros são a base para gerar o gráficos da figura 4. Para um exemplo de script gnuplot para criação deste gráfico, ver o ficheiro `ss_coverage_all.gp` na pasta `graphs/segmentspeed2` do projecto.

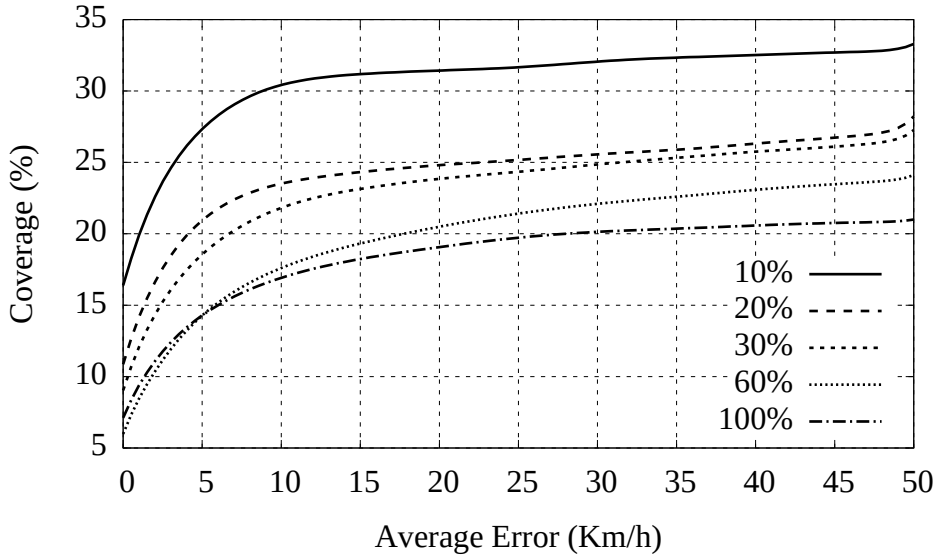

Figure 4: Erro médio vs racio de cobertura - considerando todos os resultados

Para cada percentagem de participação, é também gerado um ficheiro (`ss_[metricName]_[rate]_overall.gpd`) que contém a taxa de cobertura para cada racio de velocidades, independente do erro. Neste caso a primeira coluna representa a classe de racio de velocidades e a segunda representa a cobertura correspondente. Estes ficheiros são a base para gerar gráficos como na figura 5. Para um exemplo de script gnuplot para criação deste gráfico, ver o ficheiro `ss_speedrate_coverage.gp` na pasta `graphs/segmentspeed2` do projecto.

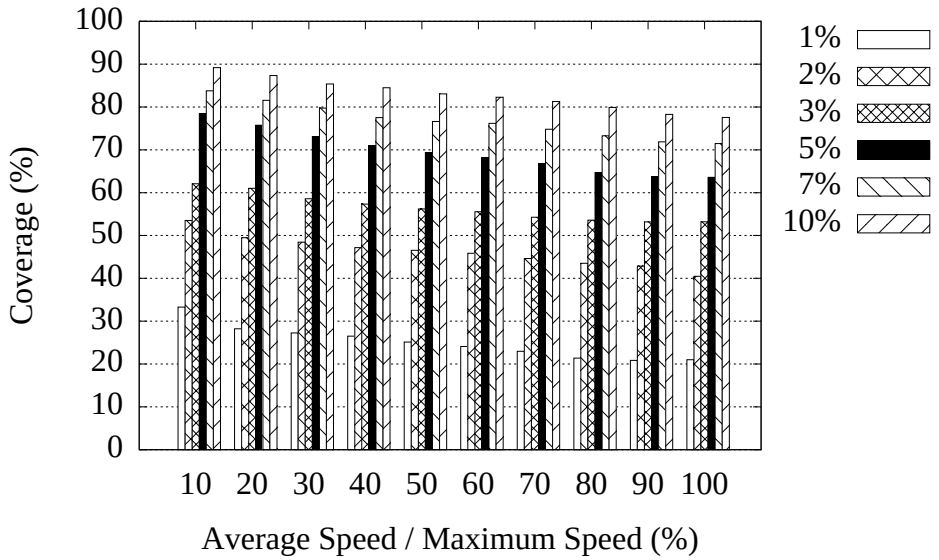

Figure 5: racio de cobertura vs racio de velocidade - considerando todos os resultados

3 Desenvolvimento de Cenário de Simulação

Para ilustrar os passos necessários para criar um novo cenário de simulação é utilizado como exemplo a detecção local de congestionamentos. O objectivo será introduzir complexidade no processamento local, de forma que os veículos forneçam informação de mais alto nível e.g., os veículos podem inferir situações de congestionamento usando a frequência de paragem numa determinada janela temporal.

1. Criar novo tipo de nó móvel

2. Criar tabela virtual e tuplos
3. Criar setups

3.1 Nó Móvel

Um novo tipo de nó móvel é definido por extensão da classe Java `sensing.persistence.simsim.MobileNode`. A classe `MobileNode` define a interface base para um nó móvel, em particular para a comunicação de leituras para a infraestrutura 4Sensing (método `sensorInput`).

Para cenários SUMO, um nó móvel estende a classe `sensing.persistence.simsim.sumo.SUMOMobileNode`, que define adicionalmente a interface para actualização da posição/velocidade a partir da reprodução de traces SUMO.

Para gerar informação de alto nível a partir de informação dos traces será necessário estender o método `update`, por exemplo colocando as leituras numa janela temporal e determinando periodicamente a frequência de paragens. A comunicação de uma leitura é feita da seguinte forma.

```
GroovyObject m = simJ.newTuple("speedsense.CongestionDetection");
m.setProperty("segmentId", sampledEdge.id);
...
sensorInput(m);
```

O metodo `sensorInput` é responsável por acrescentar o timestamp e identificador do nó móvel.

3.2 Tabela Virtual e Tuplos

A tabela virtual e os tuplos podem ser criada em qualquer package. Para a aplicação SpeedSense estas entidades estão agrupadas no package `speedsense`. Segue um exemplo para produzir tuplos `CongestionDetection` que simplesmente soma o número de detecções locais produzidas por veículos. O operador `set` na fase data-source garante que apenas uma detecção por veículo (`mNodeId`) será contabilizada. Na fase de agregação global, o operador `set` mantém a informação dos diferentes peers recebida nos últimos 5 minutos, relativa a um segmento, e desencadeia a recomputação da agregada quando é recebida nova informação de peers descendentes.

```
def wSize = SpeedSenseSim.setup.VT_WINDOW_SIZE
sensorInput(CongestionDetection)
dataSource{
        timeWindow(mode:periodic, size:wSize, slide:wSize)
        set(['mNodeId','segmentId'], mode:eos, ttl:1)
        groupBy(['segmentId]) {
                aggregate(CongestionDetection) { CongestionDetection d ->
                        count(d, 'count')
}}}
globalAggregation{
        groupBy(['segmentId]) {
                set(['peerId'], mode:change, ttl:300)
                aggregate(CongestionDetection) { CongestionDetection d ->
                        sum(d, 'count', 'count')
}}}
```

O tuplo `CongestionDetecions` define os campos `segmentId` e `count`:

```
package speedsense;
import sensing.persistence.core.pipeline.Tuple;

public class AggregateSpeed extends Tuple {
        String segmentId
        int count
}
```

3.3 Setups

O setup define o cenário de simulação:

- Configura a simulação
- Define a pesquisa a executar (ou mais de uma pesquisa)
- Define a colecção de dados de output

3.3.1 Configuração

A classe base para qualquer setup é a classe `sensing.persistence.simsim.SimSetup`. Para cenários SpeedSense, o setup irá extender a classe `sensing.persistence.simsim.speedsense.setup.SpeedSenseSetup`.

A classe `sensing.persistence.simsim.speedsense.sumo.setup.segmentspeed2.SUMOTrafficSpeedSetup` pode ser usada como referência para a criação do novo setup.

A configuração é definida no construtor da classe, definindo os parâmetros da seguinte forma:

```
config.VT_WINDOW_SIZE = 300
```

Neste caso é configurado o parâmetro `VT_WINDOW_SIZE` utilizado na definição da tabela virtual. Podem ser definidos parâmetros arbitrários a usar nas tabelas virtuais ou nós móveis. Existe um conjunto de parâmetros pré-definidios para um cenário SUMO:

Parâmetro	Função	Default
`RUN_TIME`	Tempo de execução da simulação em, segundos	0
`IDLE_TIME`	Intervalo de tempo entre o início da simulação e a execução da query	0
`SIM_SEED`	Seed para números aleatórios - equivale a `Sim_RandomSeed`, da classe `Globals` do SimSim	0L
`NET_SEED`	Seed para números aleatórios - equivale a `Net_RandomSeed`, da classe `Globals` do SimSim	0L
`SIM_TIME_WARP`	Determina velocidade de execução do simulador relativamente a tempo real	1E9
`TOTAL_NODES`	Número de nós na infraestrutura fixa 4Sensing	500
`MIN_NODES_PER_QUAD`	Parametro específico para a estratégia de distribuição QTree	2
`MIN_NODE_DISTANCE`	Distância mínima entre nós fixos, em metros	250
`SAMPLING_PERIOD`	Período de amostragem para um nó móvel 4Sensing, em segundo. O significado deste período depende da implementação do nó móvel	-
`SUMO_MNODE_RATE`	Define o racio de participação i.e., o racio de veículos SUMO que correspondem a nós 4Sensing	-
`SUMO_MNODE_CLASS_NAME`	FQN da classe que implementa o nó móvel	-
`SUMO_VPROBE_SAMPLING_PERIOD`	O período de amostragem para os traces SUMO, terá que corresponder à parametrização `vtypeproble` - ver secção 1.3	-
`SUMO_EDGESTATS_PERIOD`	O período de actualização das estatísticas SUMO, terá que corresponder à parametrização `meandata-edge` - ver secção 1.3	-

3.3.2 Pesquisa

O início da pesquisa é desencadeado pelo método `startQuery`. Para criar uma query centrada na área simulação e com 1/4 da área desta:

```
protected void startQuery() {
        String pipelineName = "speedsense.CongestionDetectionVT";
        double centerLat = sim.world.center().y;
        double centerLon = sim.world.center().x;
        double width = 0.25 * sim.world.width;
        double height = 0.25 * sim.world.height;
        Query q = createQuery(pipelineName, centerLat, centerLon, width, height);
        runQuery(q) { CongestionDetection d -> outputResult(d)}
}
```

3.3.3 Output

No caso mais simples, o resultado da simulação será o registo em ficheiro dos resultados da query. Para isso instancia-se a classe `LineOutputMetric` no método `setupCharts`:

```
public setupCharts() {
        lineOut = new LineOutputMetric("results.gpd", "");
        return [lineOut];
}
```

A classe `LineOutputMetric` é uma especialização de `sensing.persistence.simsim.Metric` que permite produzr output por linha - um ficheiro por cada simulação. Em geral as especializações de `Metric` são usadas compilar métricas arbitrárias, representar graficamente e persistir os resultados. O método `setupCharts` devolve um conjunto de especializações de `Metric` que vão receber notificações relativamente ao ciclo de vida da simulação - em particular, o inicio de simulação, um update periódico e o fim da simulação.

O método `outputResults` irá compilar a linha de output para cada resultado da seguinte forma:

```
protected void outputResult(CongestionDetection d) {
        def edge = mapModel.getEdge(d.segmentId);
        lineOut.newLine([ sim.currentTime(),
                d.segmentId,
                d.count,
                edge.pAvgSpeed * 3.6]);
}
```

Os dados registados poderão incluir dados estáticos ou dinâmicos relativos ao segmento (edge) - consultar na classe `sensing.persistence.simsim.map.sumoSUMOMapModel.Edge` os valores disponíveis.

4 Multiplas Janelas

A utilização de múltiplas janelas não é directamente suportada pela actual implementação dos pipelines, isto porque seria necessário descrever caminhos alternativos para os dados sensoriais, um para cada janela. Uma alternativa será usar uma única janela e implementar um processador responsável por desmultiplicar os tuplos, de acordo com as sub-janelas pretendidas.

Para computar o racio da velocidade nos últimos 5 minutos e a velocidade no último minuto podemos utilizar uma única janela de 5 minutos que avança a cada minuto - ver listagem abaixo. Na fase de data-sourcing, o processador que sucede à janela desdobra os tuplos onde as sub-janelas se sobrepõem e marca os tuplos de acordo com a janela a que pertencem (`window` toma o valor 1 para a janela de 5 minutos, 2 para a janela de 1 minuto).

O operador `groupBy` cria um sub-stream para cada `segmentIdwindow` para agregar parcialmente as velocidades para cada segmento e para cada janela. Como `window` é um dos "invariantes" do operador `groupBy`, as agregadas serão também marcadas com o identificador da janela.

Assume-se que os próprios veículos definem a `boundingBox` do tuplo `MappedSpeed`, em opção esta operação pode ser feita também na fase de data-source recorrendo a' informação sobre o segmento disponibilizada em mapModel.

```
1  sensorInput(MappedSpeed)
2  dataSource{
3          timeWindow(size:300, slide: 60)
4          process{ MappedSpeed m ->
5                  if(m.timestamp <= 60) {
6                          m2 = new  MappedSpeed(m)
7                          m2.window = 2
8                          forward m2
9                  }
10                 m.window = 1
11                 forward m
12         }
13         groupBy(['segmentId','window']) {
14                 aggregate(AggregateSpeed) { MappedSpeed m ->
15                         sum(m, 'speed', 'sumSpeed)
16                         count(m, 'count')
17                 }
18         }
19 }
```

Na fase de agregação global, uma janela de pequena dimensão desencadeia a recomputação 10 segundos após recepção da primeira actualização. A ideia desta janela é simplesmente aguardar que todos os peers tenham tempo de comunicar as suas actualizações - assume-se que os peers estarão dessincronizados no máximo 10 segundos.

O stream é de novo separado por segmento e janela; a agregação final irá resultar da fusão das janelas dos peers dependentes. Por fim é aplicado um classificador para cada segmento.

```
1  globalAggregation{
2          timeWindow(size:10, slide:10 mode:triggered)
```

```
        groupBy(['segmentId','window']) {
                aggregate(AggregateSpeed) { AggregateSpeed a ->
                        avg(a, 'sumSpeed', 'count', 'avgSpeed')
                }
        }
        groupBy(['segmentId']) {
                classify new SpeedRateClassifier()
        }
}

```

O classificador irá produzir um tuplo `SpeedChange` sempre que o racio de velocidades indicar uma diferença de velocidades significativa. Para isso mantém a velocidade para as duas janelas, e quando recebe a marcação de fim de stream (`EOS`) por parte da `timeWindow`, calcula o novo racio. Caso o racio revele uma diferença de velocidades relevante é emitido o tuplo `SpeedChange`.

```
classe SpeedRateClassifier extends Classifier {
        AggregateSpeed speed1, speed2 = null

        process(AggregateSpeed a) {
                if(a.window == 1) {speed1 = a} else {speed2 = a}
        }

        process(EOS eos) {
                if(speed1 && speed2) {
                        double rate = speed2.avgSpeed / speed1.avgSpeed
                        if(rate < 0.5 || rate > 1.5) {
                                forward new SpeedChange(rate:rate,
                                        speed1: speed1.avgSpeed, speed2: speed2.avgSpeed)
                        }
                }
        }
}
```

5 Atrasos Observados

Pretende-se dar uma estimativa de atraso para cada troço com base nos atrasos observados para os veículos que já saíram do troço, e usando também os atrasos dos que já saíram para calcular um atraso estimado para os carros que se encontram bloqueados no troço.

Para o fazer podemos calcular um atraso médio observado $\overline{atr_o}$ com base nos veículos que já saíram do troço. O atraso médio calculado com base em todos os veículos (os que saíram do troço e os bloqueados) tem como base um atraso total atr_t, calculado de acordo com a equação 1.

$$atr_t = \sum_{i=0}^{n_o} atro_i + \sum_{i=0}^{n_b} atrb_i + \frac{\overline{atr_o}}{length} \times \sum_{i=0}^{n_b} rem_i \tag{1}$$

O atraso total é dado pela soma do atraso total dos carros que atravessaram o troço (atrasos observados - $atro_i$), com o atraso actual dos veículos bloqueados ($atrb_i$) e o atraso adicional previsto calculado usado como referência o atraso dos veículos observados - multiplicando o atraso médio em s/m pelo total de metros que falta percorrer (somatório de rem_i).

Como um atraso é a diferença entre o tempo decorrido e o tempo esperado para a distância percorrida, a equação anterior pode ser decomposta na equação 2.

$$atr_t = (\sum_{i=0}^{n_o} tto_i + \sum_{i=0}^{n_b} ttb_i) - (\sum_{i=0}^{n_o} etto_i + \sum_{i=0}^{n_b} ettb_i) + \frac{\sum_{i=0}^{n_o} tto_i - \sum_{i=0}^{n_o} etto_i}{length} \times \sum_{i=0}^{n_o+n_b} rem_i \tag{2}$$

Para calcular este valor é suficiente os veículos comunicarem o tempo decorrido e o número de metros que falta percorrer, enviando leituras `TravelTime`:

```
class TravelTime extends Tuple {
        String segmentId
        double travelTime // tempo decorrido
        double remLen // metros por percorrer
}
```

Estas leituras serão agregadas em tuplos `AggregateTT`:

```
class AggregateTT extends Tuple {
        String segmentId
        double sumTTo // tempo decorrido - travessias completas
        double sumETTo // tempo esperado - travessias completas
        int countTTo // numero de travessias completas
        double sumTTb // tempo decorrido - veículos bloqueados
        double sumETTb // tempo esperado - veículos bloqueados
        double sumRemLen // metros que falta percorrer - veículos bloqueados
        int countTTb // numero de veículos bloqueados
}
```

A tabela virtua irá produzir, quando existe informação suficente, tuplos `Delay` com o atraso médio para o troço:

```
class Delay extends Tuple {
        String segmentId
        double avgDelay
}
```

Na fase de data-source a tabela define uma janela temporal. Quando ocorre o disparo periódico da janela, os tuplos `TravelTime` são transformados em tuplos `AggregateTT`, inicializando os contadores e acumuladores apropriados de acordo com o tipo de situação (travessia completa ou veículo bloqueado). Assume-se que os próprios veículos definem a `boundingBox` do tuplo, em opção esta operação pode ser feita também na fase de data-source recorrendo à informação sobre o segmento disponibilizada em `mapModel`.

```
sensorInput(TravelTime)
dataSource{
        timeWindow(size:300, slide: 60)
        process{ TravelTime t ->
                def edge = mapModel.getEdge(t.segmentId)
                if(t.remaining > 0) {
                        return new AggregateTT(segmentId: t.segmentId,
                                sumTTo: t.travelTime,
                                sumETTo: edge.length/edge.avgSpeed,
                                countTTo: 1
                        )
                } else {
                        return new AggregateTT(segmentId: t.segmentId,
                                sumTTp: t.travelTime,
                                sumETTb: (edge.length-t.remLen)/edge.avgSpeed,
                                countTTb: 1,
                                sumRemLen: t.remLen
                        )
                }
        }
}
```

Na fase de agregação global, a tabela usa uma janela em modo *triggered* para obter os tuplos `AggregateTT` de todos os peers dependentes e agrega os acumuladores e contadores. Finalmente, é usado um classificador para gerar um tuplo `Delay` quando existe informação suficiente - neste caso são suficientes duas travessias completas. Se se utilizar o modo de classificação completo a classificação só será aplicada quando houver informação completa sobre o troço (o modo de classificação é controlado na classe de configurações dos serviços `sensing.persistence.coreServicesConfig`.

```
globalAggregation{
        timeWindow(size:10, slide:10. mode:triggered)
        groupBy(['segmentId]) {
                aggregate(AggregateTT) { AggregateTT a ->
                        sum(a, 'sumTTo', 'sumTTo')
                        sum(a, 'sumETTo', 'sumETTo'),
                        sum(a, 'sumTTp', 'sumTTp')
                        sum(a, 'sumETTb', 'sumETTb'),
```

```
                        sum(a, 'countTTo', 'countTTo'),
                        sum(a, 'countTTb', 'countTTb'),
                        sum(a, 'sumRemLen', 'sumRemLen')
                }
        }
        classify{ AggregateTT a ->
                if(a.countTTo > 2) {
                        double totalDelay = a.sumTTo+a.sumTTb - (a.sumETTo+a,sumETTb) +
                                (a.sumTTo-a.sumETTo)/a.countTTo*a.sumRemLen
                        return new Delay(segmentId: a.segmentId,
                                avgDelay: totalDelay/(a.countTTo+a.countTTb)
                }
        }
}
```

6 Análise com/sem Semáforos

Para a análise dos dados das simulações com e sem semáforos não são necessárias simulações independentes, os resultados são persistidos juntamente com a indicação da presença de semáforo no troço em questão (ver secção 2). Nota: a presença de semáforo significa que pelo menos uma das faixas do troço tem semáforo.

Os scripts descritos na secção 2.1 podem ser utilizados para fazer a análise diferenciada. Na secção 2.1.3 é descrito como se pode utilizar uma função de filtragem no script `Graph` para obter gráficos diferenciados para troços com e sem sinais. O mesmo é possível com o script `CoverageVsErrorGraph` (secção 2.1.4) - a função `run`, aceita também uma função de filtragem como argumento opcional.

Como exemplo podemos construir um histograma da taxa de ocupação dos segmentos com e sem semáforos (figura 6) - para 100% de participação. Para isso é necessário correr duas vezes o script `Graph` especificando as funções de filtragem. Para obter dados para segmentos com sinais configura-se o script da seguinte forma

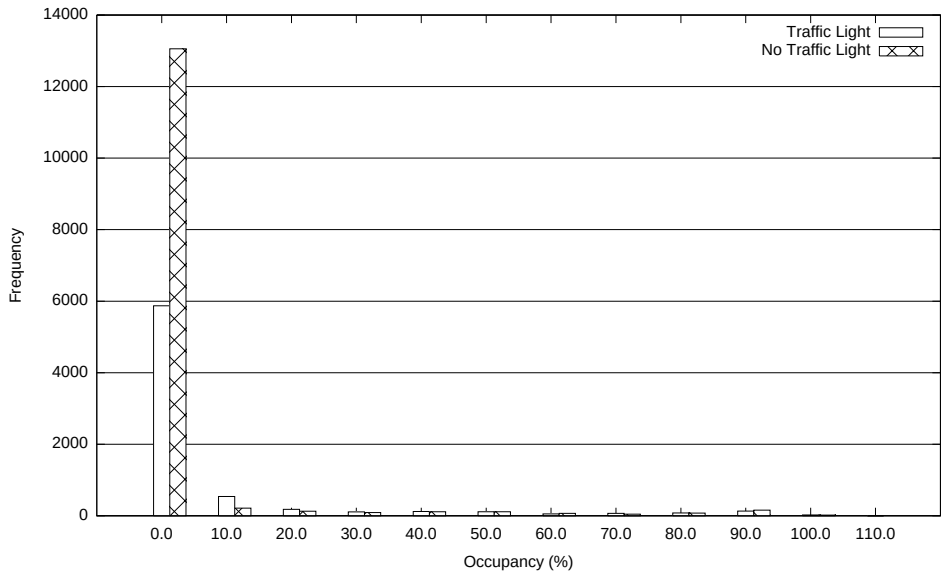

Figure 6: Histograma da Taxa de Ocupação com e sem sinais

```
String setupName = "segmentspeed2.SUMOTrafficSpeedSetup"
String inDir = "results/segmentspeed2/rateresults"
String outDir = "results/segmentspeed2/ss_occupancy"
rates = [100]
String metricName = "hist_occup_tl"
...
def meanerror = rates.collect{[it,
        run(it, {vals-> vals.edgetl.equals("1")}, binfOccup, {1}, outfStats )]
}
```

Na linha 8, a chamada à função `run` especifica no 2º argumento o filtro para segmentos com semáforo. O 3º argumento é a função de classificação por taxa de ocupação. Segue-se a função de sumarização que neste caso não é relevante - é uma função que devolve sempre o valor 1 - estamos interessado apenas na frequência de ocorrência.

Para obter os dados para segmentos sem sinais, altera-se apenas o nome da métrica (define o nome dos ficheiros de output) e a função de filtragem como se segue:

```
1 ...
2 String metricName = "hist_occup_notl"
3 ...
4 def meanerror = rates.collect{[it,
5         run(it, {vals-> vals.edgetl.equals("0")}, binfOccup, {1}, outfStats )]
6 }
```

A frequência de cada classe será a 4ª coluna dos ficheiro de output. O seguinte script gnuplot constroi o gráfico da figura 6:

```
1 set term pdf monochrome
2 set output "ss_hist_occup.pdf"
3 set style histogram clustered
4 set style fill pattern
5 set grid ytics
6 set xlabel "Occupancy (%)"
7 set ylabel "Frequency"
8 plot "ss_hist_occup_tl_100.gpd" u 4:xtic(1) w histogram t "Traffic Light",\
9         "ss_hist_occup_notl_100.gpd" u 4 w histogram t "No Traffic Light"
```