Esame di Metodi di Ottimizzazione per la Logistica

Federico Motta, matricola 79539, A.A. 2014 / 2015

Relazione del progetto sviluppato

Traccia / spunto: 4.HOME-WORK MOVEMENT
Data una mappa di una città, una sede di un azienda e le abitazioni dei vari dipendenti, si definisce una proposta di piano degli spostamenti casa lavoro che utilizzi:

a) il car pooling (un dipendente con la sua vettura passa a prendere anche altri dipendenti);
b) minibus (10 persone) dedicati e localizzati in un deposito presente sulla mappa, il cui costo è pari a quello di 3 auto (cioè un arco (i,j) attraversato da un auto costa c(i,j), dal minibus (3 c(i,j))

Descrizione del problema

Descrizione informale del problema

Un azienda vuole ridurre l'impatto ambientale ed energetico dei viaggi compiuti dai propri dipendenti da casa al luogo di lavoro. Ne consegue che occorre ridurre il numero di veicoli normalmente utilizzati (si suppone di partire da una situazione in cui ogni dipendente utilizzi la sua auto) per risparmiare carburante e ridurre l'inquinamento.
Le strategie da adottare sono due:

- Car pooling (ovvero un dipendente condivide la propria auto con altri 4 colleghi, passandoli a prendere prima di recarsi al lavoro
- Minibus (ovvero un mezzo con 10 posti più un autista è reso disponibile dalla azienda ai dipendenti)

Formalizzazione del problema

Ogni luogo ha un ID (indice identificativo), un tipo (azienda, deposito, incrocio o casa), un'ascissa (reale) ed un'ordinata (reale).
Tra tutti i luoghi ce ne è uno solo di tipo azienda ed uno solo di tipo deposito, gli altri sono tutti o incroci o case.
Il numero di bus presenti nel deposito è definito.
In ogni casa abita un solo dipendente con una sola auto (il numero di auto è pari al numero di case).
I collegamenti tra luoghi sono definiti, altrimenti non esistono.
Le auto hanno 5 posti (1 dipendente e 4 suoi colleghi).
I bus hanno 10 posti per i dipendenti (l'autista è considerato undicesimo, e non è preso in considerazione nella risoluzione del problema)
Ogni viaggio parte da una casa o dal deposito, e termina all'azienda (passando eventualmente per altre case e / o incroci)

Input necessari:

Ax, Ay = Coordinate azienda
Dx; Dy = Coordinate deposito
Ix; Iy = Array di coordinate degli incroci
Cx; Cy = Array di coordinate delle case
St = Array delle strade esistenti (ogni cella pari contiene l'ID del luogo di partenza, la cella successiva contiene l'ID del luogo di arrivo)
Gli ID sono così distribuiti (Ni indica il numero di incroci mentre Nc il numero di case):

azienda = 0 deposito = 1 incroci $\in$ [2 … , Ni + 1] case $\in$ [Ni + 2, … , Ni + Nc + 1]

Output fornito:

Detta route una sequenza di luoghi attraversati, per cui in ogni luogo un flag booleano indica se è stato solamente attraversato (false) o se si è raccolto il dipendente ed è stato attraversato (true).
L'output è un insieme di route con specificato il tipo di mezzo utilizzato (auto o bus), con annesso calcolo del costo totale della soluzione.

Breve analisi letteratura più rilevante

- Anno: 1983
  Paper: A Fair Carpool Scheduling Algorithm
  Autori: Ronald Fagin, John H. Williams
  Fonte: IBM Journal of research and development
  Come da titolo questo primo documento non tratta di algoritmi per risolvere il problema del car pooling come si intende in questa relazione, ma si limita a considerare un caso semplice del problema (un numero ridotto di individui da raccogliere con un solo mezzo appartenente ad uno degli stessi) per definire un modo equo e giusto per stabilire i crediti / debiti tra i partecipanti.
  In particolare si assume M = capacita del mezzo; U = minimo comune multiplo tra 1, …, M.
  Quindi si procede alla creazione di una tabella come la seguente (es. M=4, U=12):
  La tabella deve essere aggiornata giornalmente secondo le seguenti regole:
  - ogni individuo parte con 0
  - K è il numero di partecipanti di ogni giorno
  - chi guida aumenta il proprio indicatore di U*(K-1)/K
  - chi non guida cala il proprio indicatore di U / K

Data	Aldo	Bartolomeo	Carlo	Davide
	0	0	0	0
1 Gennaio	0	8	-4	-4
2 Gennaio	-3	5	-7	5

  Tale documento viene citato dunque più che altro per l'importanza storica, infatti già 30 anni fa emergeva (in piccola scala) il problema che oggi ci si propone di risolvere.

- Anno: 2003
  Paper: The car pooling problem: Heuristic algorithms based on saving functions
  Autori: Emilio Ferrari, Riccardo Manzini, Arrigo Pareschi, Alessandro Persona, Alberto Regattieri
  Fonte: Journal of Advanced Transportation
  Il documento propone una soluzione ad un problema simile (viene considerato un grafo non orientato, euclideo e completamente connesso) che sfrutta un'euristica costruttiva e successivamente un approccio combinatorio.
  La parte euristica calcola i savings per ogni coppia di luoghi $S(i,j) = C(0,i) + C(0,j) – C(i,j)$, ovvero il risparmio ottenuto col percorso “0-i-j-0” rispetto ai percorsi “0-i-0” e “0-j-0”; quindi ordina i savings in maniera decrescente. A questo punto vengono individuati dei clusters tramite tre algoritmi greedy basati sui savings, i quali differiscono tra loro per le scelte effettuate su savings inerenti luoghi già appartenenti ad un cluster o su savings non ammissibili.
  Infine, dopo aver creato i clusters, con un algoritmo esatto basato su Floyd-Warshall che ha costo $O(n^3)$ sono calcolati tutti i cammini minimi di un grafo, e viene eseguito il routing dei clusters.
  Anche questo esempio è riduttivo, principalmente perché non considera un grafo sparso ed orientato, ed anche perché su un numero elevato di luoghi diventa improponibile applicare Floyd-Warshall; tuttavia si rivela interessante l'idea dei savings.

- Anno: 2011
  Paper: PoliUniPool: a carpooling system for universities
  Autori: Maurizio Bruglieri, Diego Ciccarelli, Alberto Colornia, Alessandro Luè
  Fonte: Procedia Social and Behavioral Science
  Questo documento presenta un'istanza più complessa del problema: sono considerati tutti gli studenti dell'Università Statale e del Politecnico di Milano che non riescono ad usufruire di trasporti pubblici (46%), ed ognuno di loro ha delle necessità come degli orari o delle preferenze sui compagni di viaggio. L'algoritmo sfrutta una struttura dati dinamica chiamata Matching Matrix (MM) il cui ipotetico elemento $a_{x,y}$ indica se e come il passeggero connesso al viaggio y possa essere inserito nel pool di persone che compiono il viaggio x. La MM viene quindi aggiornata a seconda delle esigenze temporali, ed alla fine attraverso una funzione obiettivo (che tiene conto di: benefici kilometrici, soddisfacimento della domanda, livello medio dell'incremento del kilometraggio rispetto al percorso minimo, preferenze degli utenti, pool di individui che si sono già incontrati in pool precedenti) viene scelta una soluzione grazie all'utilizzo di un metodo statistico (Monte Carlo).
  La potenza di questa soluzione risiede nell'idea di introdurre un apporto statistico / stocastico per evitare moli di calcoli esponenziali. Tuttavia rispetto al problema che si intende risolvere in questo caso ci sono molti più particolari superflui, come: il concetto di compagnia ed amicizia oppure le finestre temporali (non introdotte nell'algoritmo proposto perché si suppone che in un'azienda o in un'industria gli orari siano più o meno gli stessi per tutti. Nel caso ce ne fosse bisogno, si può comunque applicare l'algoritmo più volte ad istanze diverse di utenti, dal momento che gli orari di un lavoratore non sono cosi variegati come quelli di uno studente ma seguono generalmente dei turni).

Euristiche costruttive e / o di ricerca locale / meta-euristiche

L'algoritmo proposto è orientato a superare i difetti di quelli presentati nell'analisi della letteratura più rilevante e / o trarre vantaggio dai loro punti di forza.
Il grafo considerato è dunque sparso ed orientato, la sua dimensione è medio-grande e l'approccio euristico e meta-euristico proposto sfrutta i vantaggi che apportano la casualità, il calcolo di route ottime su pochi luoghi (massimo 10), ed il concetto di saving (applicato sia a coppie di luoghi che coppie di routes).

Descrizione algoritmo:

Come prima cosa si calcolano tutti i percorsi minimi tra i luoghi del seguente insieme: {azienda, deposito, case}. Ciò è possibile grazie all'algoritmo di Dijkstra, che con un costo $O(n^2)$ calcola tutti i cammini minimi da un luogo del grafo (si intende la rete costituita dai luoghi e dalle strade che li collegano) a tutti gli altri; dato che però occorrono tutti i cammini minimi tra tutti i luoghi sopra citati, bisogna applicare Dijkstra n volte. Il costo totale è allora di $O(n^3)$, dove n è il numero di case + 2.
Questo calcolo preventivo ha come scopo quello di ridurre il grafo iniziale di tutti quegli incroci e di tutti quegli archi per cui cui non passano i cammini minimi; di modo che il resto dell'algoritmo lavori su un grafo ridotto allo stretto necessario. Quindi per un numero parametrico di threads vengono create altrettante soluzioni euristiche. Ogni euristica genera una soluzione potenzialmente diversa dalle altre euristiche; le idee applicate sono le seguenti:

- Come prima cosa per ogni casa si valuta un coefficiente di “thickness” che viene calcolato come la distanza media dalle 3 o 4 case più vicine fratto la distanza dall'azienda; ovvero indica quanto una casa è parte di un “cluster” di case tra loro fitte. Successivamente per le case con “thickness” minore ad una certa soglia parametrizzata vengono create delle route ottime tra esse e le loro vicine.

◆ La seconda cosa da fare è valutare tra le case rimanenti quali sono da considerare “lontane” dall'azienda, ovvero una certa percentuale delle case rimanenti dalla prima scrematura (ordinate per distanza decrescente dall'azienda con un quick-sort randomizzato O(n·logn)). Prima di passare ad esse viene calcolato per ogni casa non ancora inclusa in una route (cioè tutte le case non “thick”) il saving rispetto ad ogni altra casa analoga; nel caso il dipendente residente nella prima si recasse al lavoro passando per la casa del collega residente nella seconda. In seguito viene stabilito un numero parametrico di colleghi che i dipendenti lontani devono raccogliere. Infine per ogni collega lontano si estrae un saving positivo scelto casualmente tra tutti i savings che lo interessano finché non viene raggiunto il numero di colleghi da raccogliere stabilito, allora viene calcolata la route ottima tra di essi.

◆ Il terzo gruppo di azioni da intraprendere prevede per le case rimaste dai primi due punti (cioè tutte quelle non “thick” e non “lontane”) la scelta di un certo pool di colleghi (scelto casualmente tra i colleghi simili rimasti), dal quale si estrae il collega con saving maggiore e lo si include nella route (che viene sempre calcolata come l'ottima); tale procedimento prende il nome di scelta per torneo e viene applicato fintanto che ci sono colleghi non inclusi in nessuna route.

L'euristica prevede dunque solamente la creazione di routes: a partire dalle case considerate fitte (o “thick”), a partire dalle case considerate lontane, ed a partire dalle case rimanenti.
Sia chiaro che l'euristica è tarata per creare routes solamente di auto e con un numero di posti liberi parametrico (un buon parametro in questo caso può essere 1 o 2), affinché la meta-euristica non debba disfare completamente le routes euristiche ma possa, almeno in un primo momento, tentare di saturare il numero di posti delle auto.

A questo punto per ogni soluzione euristica, dopo aver eliminato eventuali soluzioni doppie, viene avviato un thread meta-euristico.
Ogni thread meta-euristico migliora la soluzione assegnatagli secondo le seguenti idee di fondo:

◆ Come prima cosa per ogni gruppo di 2 o 3 routes di auto (che raccolgono un numero di dipendenti che potrebbe essere contenuto da un bus) si valuta quanto convenga prelevare i dipendenti raccolti da esse con un solo bus (seguendo una route che dal deposito arrivi al dipendente più vicino tra quelli da raccogliere, sia ottima tra le case dei dipendenti, e termini all'azienda passando per il dipendente più vicino ad essa).

◆ Successivamente occorre calcolare per ogni route ed ogni casa il saving corrispondente (ovvero quanto converrebbe spostare la casa dalla route in cui è all'altra route). Quest'operazione è onerosa perché per ogni saving può arrivare a costare O(10!) (che comunque resta un tempo affrontabile per i moderni processori, 10! ~ 3'628'800) ma è alla base del punto successivo, e del concetto di neighborhood tra routes.

◆ Infine viene applicata una vera e propria ricerca locale; ovvero per ogni saving positivo rimasto si modificano le routes corrispondenti e si aggiornano solamente i savings che tale modifica può aver cambiato. I savings vengono estratti in ordine di risparmio decrescente, quindi la ricerca locale non esce dai minimi locali.
Dato che l'operazione di ricerca locale può durare un tempo considerevole (non è detto che il numero di savings rimasti cali: potrebbe darsi che applicare un saving, comporti in fase di aggiornamento dei possibili savings cambiati, il passaggio di diversi savings che prima erano sconvenienti a convenienti), è possibile interrompere l'algoritmo tramite un segnale ed avere così una soluzione parziale.

La percentuale visualizzata di avanzamento è quindi calcolata sul numero di savings applicati fino all'iterazione corrente fratto il numero di savings rimanenti all'iterazione corrente.
Un'altra considerazione rilevante è la condizione di uscita aggiuntiva (dalla ricerca locale), che termina la meta-euristica se dopo un intero percentuale di progresso (es. dal 50% si passa al 51%) il gradiente di miglioramento del costo della soluzione è in valore assoluto minore a 10^{-4}.

Pseudocodice

```
function allMinDist
        for all {firm, depot, houses} do                    /* Calcola tutti i cammini minimi
                distance=Dijkstra(currentItem)               * e li salva in una matrice. */
        end-for
end-function

function Heuristc
        for all {houses} do                                 /* Calcola i coefficienti di
                calculateThickCoeff(currentItem)             * thickness di tutte le case
        end-for                                              * e li salva in un array. */
        for all (thickCoeff < thickThreshold) do            /* Crea le route a partire dai
                createRoute(clusterOfCurrentItem)            * clusters di case. */
        end-for

        for all {houses not in any route} do                /* Calcola i savings tra tutte le
                for all {houses not in any route} do         * coppie di case rimaste, se
                        calculateSaving(item1,item2)         * l'item1 e l'item2 fossero in una
                end-for                                      * stessa route rispetto al caso in
        end-for                                              * cui fossero in routes diverse.*/

        for all {houses not in any route} do                /* Calcola le lunghezze dei
                L=getLength(Distance(currentItem,firm))      * cammini minimi calcolati
        end-for                                              * in precedenza. */
        R=reverseRandomizedQuickSort(L)                     /* Le ordina decrescentemente */
        for all {item in R*farThreshold} do                 /* Per una certa percentuale di
                createEmptyRoute                             * dipendenti lontani rimasti,
                while (route has less than K employee)       * crea delle route ottime con
                        extractRandomPositiveSaving          * colleghi aventi saving
                        addToRoute(houseExtracted)           * positivo scelti casualmente
                end-while                                    * finché ognuna ha 5-K posti
        end-for                                              * liberi. */

        for all {houses not in any route} do                /* Per ogni casa rimanente crea
                createEmptyRoute                             * una route vuota, e finché non
                while(route has less than W emplyee)         * ha 5-W posti liberi o non ci
                        getRandomSet(remaining houses)       * sono più case senza route,
                        getGreaterPositiveSaving(randSet)    * si estrae la casa con saving
                        addToRoute(collegue extracted)       * maggiore e non nullo da
                end-while                                    * un insieme ottenuto
        end-for                                              * scegliendo casualmente le case
        return solution                                      * tra quelle rimaste. */
end-function
```

```
function RemoveDuplicatedHeuristicSolution                  /* Tra tutte le soluzioni euristiche
end-function                                                 * create vengono tolte eventuali
                                                             * doppie. */
function Meta-heuristic
        for each set of 2 or 3 routes                        /* Tra tutti i gruppi di 2 - 3 routes
                if (pickedEmployees are less than 10) than   * di auto con meno di 10 posti
                        evaluateBusRoute(pickedEmplyees)     * occupati, si calcola la route di
                        if (busRouteCost is less than        * bus ottima sostitutiva, e se ha
                                        carRouteCost) than   * costo minore delle routes di
                                keepBusRoute                 * auto la si mantiene, altrimenti
                                removeCarRoutes              * vengono mantenute le routes
                        end-if                               * delle auto. */
                end-if
        end-for

                                                             /* Per ogni coppia route-casa
        for all {routes} do                                  * si calcola il saving se la casa
                for all {houses} do                          * venisse spostata dalla route
                        calculateSaving(item1, item2)        * in cui è, a quella item1. Le
                end-for                                      * routes calcolate sono sempre
        end-for                                              * le ottime per le case incluse.*/

        while (positiveSavingNumber is greater than 0)       /* Finché rimangono savings
                if (signal stop) than                        * positivi, se si riceve il segnale
                        break                                * stop si esce dalla ricerca locale
                end-if                                       * altrimenti si estrae il saving
                applyGreaterSaving                           * maggiore e lo si applica.
                refreshInfluencedSavings                     * Quindi si aggiornano i savings
                if (percentageProgress is greater by 1 and   * inerenti le routes / case
                         currentSaving is less than 10^-4) than  * modificate. Se la percentuale
                        break                                * di progresso è aumentata di 1
                end-if                                       * ed il saving corrente è minore
        end-while                                            * di 10^-4 si esce dalla ricerca
                                                             * locale, altrimenti si reitera. */
        if (checkSolutionAdmissibility) than
                return solution                              /* Si verifica che la soluzione sia
        else                                                 * ammissibile, altrimenti la si
                makeSolutionAdmissible                       * rende tale e la si ritorna. */
                return solution
        end-if
end-function

function getBestSolution                                     /* Tra tutte le soluzioni trovate
        for all {solutions} do                               * si estrae quella con costo
                if (currentItem cost is less than others) than  * minore e la si ritorna */
                        return currentItem
                end-if
        end-for
end-function
```

Predisposizione istanze di test

Per testare l'algoritmo precedentemente descritto e la sua implementazione in Java allegata alla relazione, occorre dunque un insieme di problemi di Car Pooling da risolvere.
Per ovviare questo problema sono percorribili due approcci differenti, di seguito descritti e successivamente applicati:

- Il primo modo di generare detti problemi è abbastanza artigianale-manuale, si trova una cartina o mappa stradale; la si apre con un editor grafico che supporti i layer (es. GIMP), si creano due layer trasparenti e manualmente si segnano con dei punti neri (es. pennello tondo di raggio 9 pixel) tutti gli incroci che si desidera importare. Quindi terminata la fase dei punti, si cambia layer e con lo strumento linea retta ed un colore chiaro (es. giallo) si collegano i punti / incroci con linee, sfruttando sempre la trasparenza per osservare la sottostante cartina. A questo punto grazie ad Imagemagick, un programma di grafica utilizzabile da BASH è possibile con opportune pipeline (che si basano sulle regioni contigue di colore e sulle componenti connesse di un'immagine) estrarre dal layer dei punti un elenco di coordinate x, y; ed altrettanto per il layer di linee. Successivamente sempre grazie a delle pipeline BASH (tra i comandi rilevatisi utili troviamo: awk, cat, column, cut, paste, sed, sort, tr, xargs) si riescono a trasformare tali elenchi di coordinate negli array di input al problema, l'unica cosa da decidere rimangono il numero di bus, quali coordinate destinare all'azienda, quali al deposito e quante assegnarne agli incroci piuttosto che alle case. (In allegato alla relazione, il file RE.dat è un esempio di tale approccio applicato a Reggio Nell'Emilia (44°42'34"56 N; 10°37'13"80 E))
- Il secondo approccio, di certo più informatico-automatizzato, è l'utilizzo di un programma dedicato. In allegato alla relazione si trova appunto l'archivio compresso JAR, del programma Java HWM_test_generator. Tramite tale programma è possibile generare problemi anche di grosse dimensioni in poco tempo. Tra i parametri impostabili si trovano:
    - il numero di luoghi
    - la posizione dell'azienda (al centro, casuale o manuale)
    - la posizione del deposito (vicina all'azienda, lontano da essa, casuale o manuale)
    - la disposizione dei luoghi (gaussiana intorno all'azienda o casuale-uniforme)
    - la percentuale dei luoghi che si desidera impostare come case
    - il numero di strade
    - la percentuale di strade da rendere a senso unico
    - il numero di bus nel deposito

  (In allegato alla relazione, diversi file “Sample”.dat sono stati creati con tale programma)

Campagna di test

Come gia anticipato in precedenza sono stati utilizzati due diversi approcci di test.
L'unico test su una città reale è stato eseguito su Reggio Nell'Emilia, ed è stato eseguito svariate volte su un PC fisso dual-core sempre con il massimo numero di threads impostabili per tale processore (per vincoli volontariamente introdotti nel programma HWM_solver.jar che limitano il numero di threads in base al numero di cores disponibili; perché la Java Virtual Machine su tale PC supporterebbe in teoria fino a 5000 threads circa), ovvero 64 threads.
Tutti i risultati delle diverse esecuzioni (durate tutte tra l'ora e l'ora e mezza) si sono aggirati intorno alle 525'000-535'000 unità di costo, con un minimo assoluto di 523'327, con 60 route di auto (le case sono 250); quindi il numero medio di dipendenti per automobile è 4,2.
È interessante osservare come neanche una route sia di bus; né in questo che in tutti gli altri test condotti; ciò è probabilmente dovuto al costo eccessivo di percorrimento dei bus rispetto alle auto, infatti provando ad abbassare il coefficiente da 3 a 2 od a valori tra 2 e 3 in certi problemi emergono

route di bus. Per coerenza con la traccia indicata tali esperimenti sono solamente citati in questa relazione. I test condotti successivamente sono stati eseguiti meno volte (data la mole di luoghi ed il tempo impiegato per risolverli) ma hanno condotto risultati altrettanto interessanti.

Sample1:

- 1'000 luoghi
- azienda centrale
- deposito lontano
- disposizione gaussiana
- case = 20% dei luoghi (=200)
- 10'000 strade
- sensi unici = 20% delle strade (=2'000)
- 200 bus

- Costo soluzione migliore: 43'090
- Numero di auto utilizzate: 42
- Numero medio di dipendenti per auto: 4,8
- Numero di threads utilizzato: 32
- Tempo impiegato: 60-90 minuti

Sample2:

- 1'000 luoghi
- azienda centrale
- deposito lontano
- disposizione casuale-uniforme
- case = 20% dei luoghi (=200)
- 10'000 strade
- sensi unici = 20% delle strade (=2'000)
- 200 bus

- Costo soluzione migliore: 23'922
- Numero di auto utilizzate: 42
- Numero medio di dipendenti per auto: 4,8
- Numero di threads utilizzato: 32
- Tempo impiegato: 60-90 minuti

È interessante a questo punto osservare come la disposizione gaussiana sia simile ad una grande città, sviluppata “a ragnatela”, mentre quella uniforme sia approssimabile ad una conurbazione di tante piccole città poste discretamente vicine le une alle altre.
Inoltre i due primi Sample differiscono solo per la distribuzione, ma i costi sono notevolmente diversi; questo è probabilmente dovuto alla topologia della rete. Infatti la distribuzione casuale-uniforme ha prodotto grafi di reti simili a quelle dette “scale-free”, dove diversi hub formati da pochi nodi strettamente connessi tra loro, hanno archi lunghi che li connettono ad altri hub distanti; così come ciò favorisce il passaggio di informazioni in maniera veloce in tali reti (Internet è una rete “scale-free”), probabilmente in questo caso consente con pochi archi di raggiungere il centro.
Infine il numero di nodi dei primi due Sample è paragonabile a quello di Reggio Nell'Emilia, tuttavia i costi sono profondamente diversi, sicuramente per la diversa scala, e per il diverso numero di strade e sensi unici, che costringono la soluzione a cammini tortuosi.

Sample3:

- 5'000 luoghi
- azienda disposta casualmente
- deposito lontano dall'azienda
- disposizione gaussiana
- case = 40% dei luoghi (=2'000)
- 50'000 strade
- sensi unici = 30% delle strade (=15'000)
- 500 bus

- Costo soluzione migliore: 1'718'107
- Numero di auto utilizzate: 408
- Numero medio di dipendenti per auto: 4,9
- Numero di threads utilizzato: 16
- Tempo impiegato: 8 ore

Sample4:

- 5'000 luoghi
- azienda disposta casualmente
- deposito lontano dall'azienda
- disposizione casuale-uniforme
- case = 40% dei luoghi (=2'000)
- 50'000 strade
- sensi unici = 30% delle strade (=15'000)
- 500 bus

♦ Costo soluzione migliore:	1'130'839
♦ Numero di auto utilizzate:	402
♦ Numero medio di dipendenti per auto:	4,98
♦ Numero di threads utilizzato:	16
♦Tempo impiegato:	2 giorni 19 ore 15 minuti

Anche questi ultimi due test confermano quanto ipotizzato precedentemente, come dimostrato dal costo della soluzione ed il numero medio di dipendenti per auto (il più alto in assoluto).

Per i sample 5 e 6 le risorse a disposizione ma soprattutto il tempo rimasto non risultano sufficienti per condurre test su dimensioni ancora maggiori, sono tuttavia resi disponibili i dati di input, al lettore interessato, in allegato alla relazione.

Federico Motta, matricola 79539, 14 agosto 2015