

Laboratorio di Ottimizzazione e Simulazione L

Una libreria per la soluzione di problemi di routing su grafi

Prof. Roberto Baldacci

DEIS - Università di Bologna

r.baldacci@unibo.it

Ing. Luca Gardelli

DEIS - Università di Bologna

luca.gardelli@unibo.it

Ultima modifica 19 Febbraio 2008

Materiale

- Tutto il materiale presentato è disponibile agli indirizzi
 - <http://or.ingce.unibo.it>
 - <http://unibo.lgardelli.com>

Obiettivi

- Sensibilizzare gli studenti verso le problematiche relative all'implementazione di algoritmi per la soluzione di problemi di ricerca operativa...
- ... attraverso un caso di studio : una libreria Java per la risoluzione di problemi di cammino minimo (spp) e visita di clienti (tsp) su grafi...
- ... e l'esperienza accumulata nello sviluppo di tale libreria.

NOTA : lo sviluppo di tale libreria ha richiesto un impegno di circa 1,5 mesi/uomo

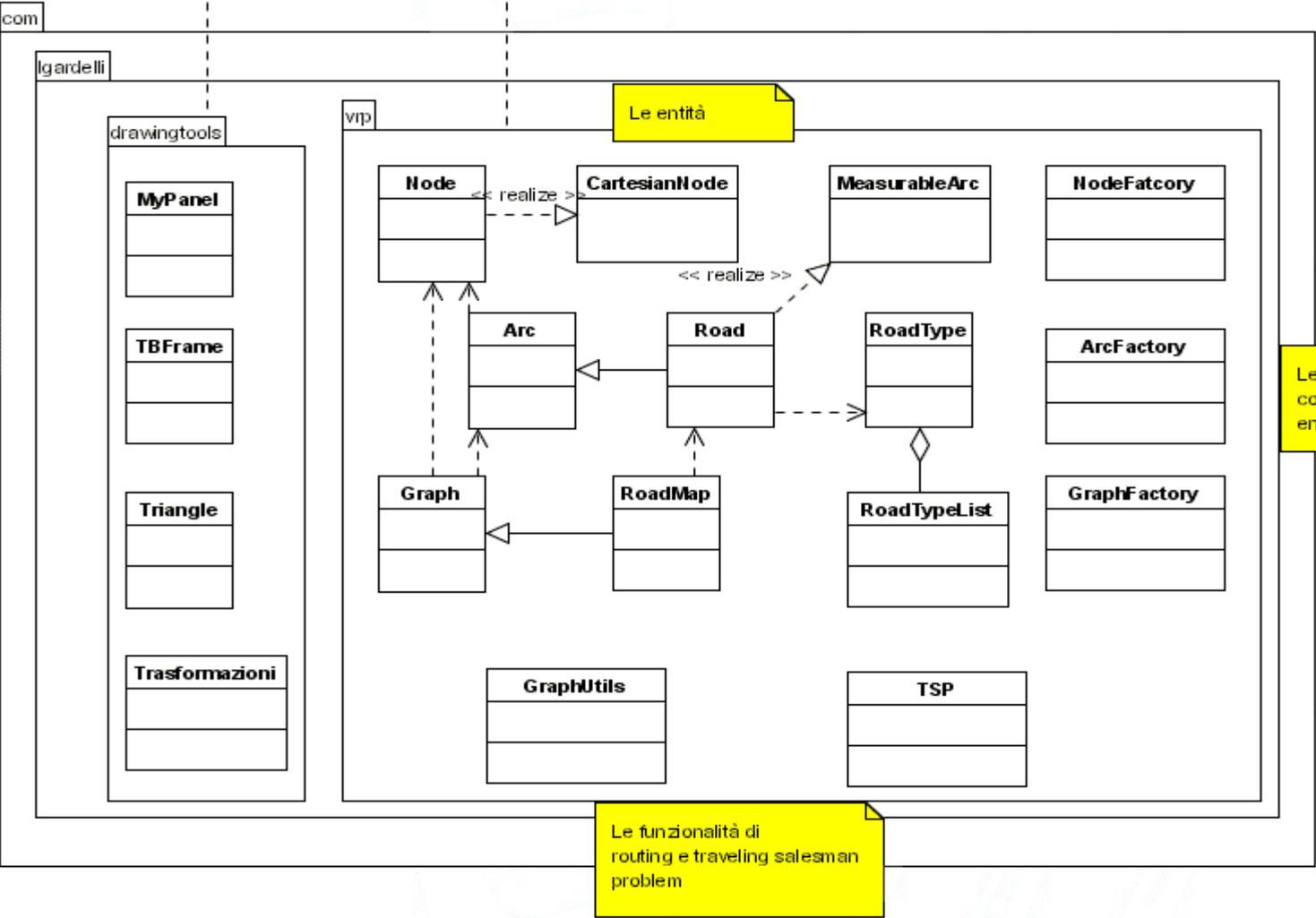
Libreria : entità e funzionalità

- *Quali entità sono necessarie per risolvere problemi sui grafi?*
 - Nodo
 - Arco
 - Grafo
 - ...
- *Quali problemi ci interessa risolvere?*
 - Il calcolo del cammino minimo tra due nodi (SPP)
 - Il calcolo del percorso di visita di N clienti (TSP)
 - ...

Libreria : architettura logica

gestisce le funzionalità per la visualizzazione

contiene le entità e le funzionalità strettamente legate al vehicle routing problem

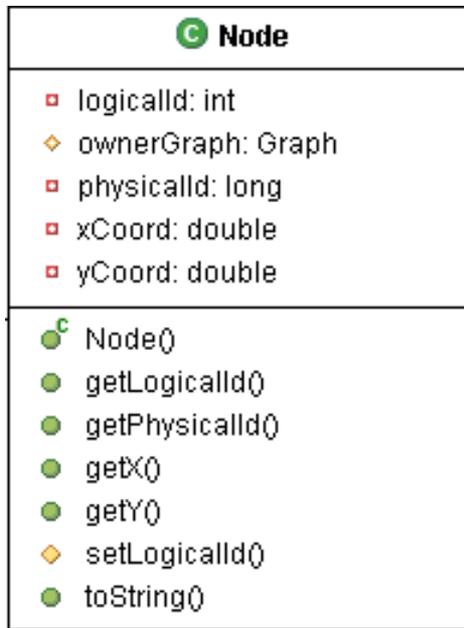
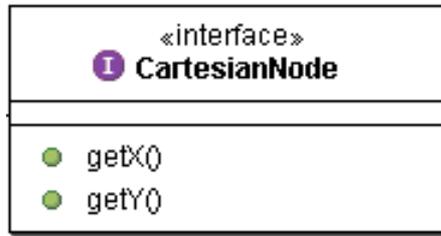


Le entità

Le funzionalità di costruzione delle entità

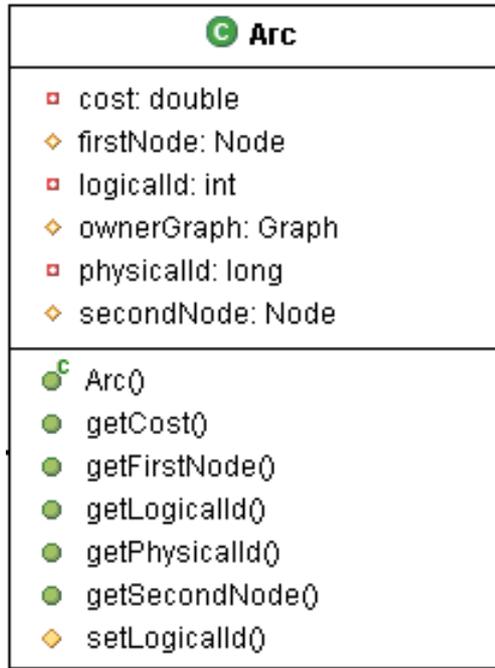
Le funzionalità di routing e traveling salesman problem

Le entità della libreria : *Node*



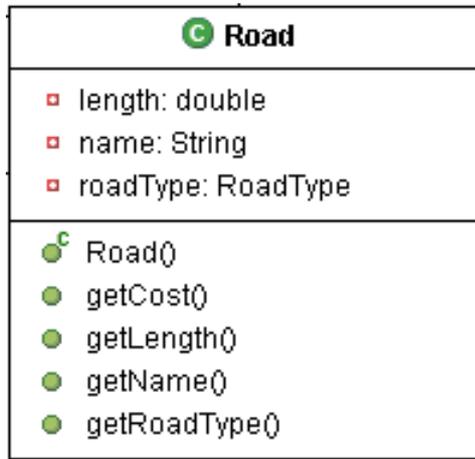
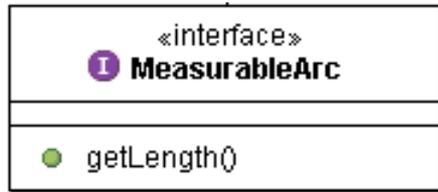
- *Che cosa caratterizza un nodo?*
- Un oggetto di tipo *Node* è un puro valore, cioè l'utente non può modificarne lo stato una volta creato
- Sicuramente deve possedere un ID. Nel nostro caso ne possiede due:
 - uno passatogli al momento della costruzione (`physicalID`)
 - uno per uso interno (`logicalID`)
- Nel nostro caso un oggetto di tipo *Node* implementa l'interfaccia (definita da noi) *CartesianNode*, cioè il nodo è rappresentabile in un spazio di coordinate cartesiane

Le entità della libreria : *Arc*



- *Che cosa caratterizza un arco?*
- Un oggetto di tipo *Arc* è un puro valore, cioè l'utente non può modificarne lo stato una volta creato
- Sicuramente deve possedere un ID, nel nostro caso ne possiede due
 - uno passatogli al momento della costruzione (`physicalID`)
 - uno per uso interno (`logicalID`)
- Un arco è orientato e parte da un nodo (`FirstNode`) e arriva ad un secondo nodo (`SecondNode`). Questa transizione implica un costo che non necessariamente è la lunghezza dell'arco

Le entità della libreria : *Road*



- *Che cosa caratterizza una strada?*
- Un oggetto di tipo *Road* è un puro valore, cioè l'utente non può modificarne lo stato una volta creato
- Sicuramente deve possedere tutte le proprietà di un arco. In più possiede
 - un nome
 - una lunghezza
 - una tipo di strada (extraurbana...)
- Allora la classe *Road* estende la classe *Arc* ed implementa l'interfaccia *MeasurableArc*

Le entità della libreria : *RoadType* e *RoadTypeList*

C RoadType	
▣	speed: double
▣	type: String
●	RoadType()
●	getSpeed()
●	getType()

C RoadTypeList	
▣	roadTypeList: Hashtable
●	RoadTypeList()
●	addRoadType()
●	getRoadType()

- *Che cosa caratterizza una tipo di strada?*
- Un oggetto di tipo *RoadType* è un puro valore, cioè l'utente non può modificarne lo stato una volta creato
- E' identificato da un nome di tipo e dalla velocità media di percorrenza
- Un oggetto di tipo *RoadTypeList* non è altro che una lista di oggetti di tipo *RoadType*

C Graph	
▣	arcCount: int
◇	arcList: Vector
◇	bsTable: Hashtable
◇	fsTable: Hashtable
▣	nodeCount: int
◇	nodeList: Vector
◇	physicalArcId: Vector
◇	physicalNodeId: Vector
●	Graph()
●	addArc()
◇	addArcToBwdStar()
◇	addArcToFwdStar()
●	addNode()
●	containsArc()
●	containsNode()
●	getArc()
●	getArc()
●	getArcList()
●	getArcsCount()
●	getArcsInBwdStar()
●	getArcsInFwdStar()
●	getBwdStar()
●	getFwdStar()
●	getNode()
●	getNodeList()
●	getNodesCount()
●	resolveArcId()
●	resolveNodeId()

Le entità della libreria : *Graph*

- *Che cosa caratterizza un grafo?*
- Un oggetto di tipo Graph NON è un puro valore. Sarebbe troppo scomodo ed inefficiente (un po' come la lista...)
- Un grafo è definito come un insieme di nodi ed un insieme di archi fra i nodi
- Quali sono le funzionalità di cui dispone?
- Ottenere un arco, un nodo, la lista di archi, la lista di nodi, la fwd star, la bwd star, controllare se un arco o un nodo sono contenuti, il numero di archi e nodi, aggiungere un arco e un nodo...

Le entità della libreria : *RoadMap*

 RoadMap
▣ parameter: double
 RoadMap()
 getParameter()
 setParameter()

- *Che cosa caratterizza una rete stradale?*
- Un oggetto di tipo RoadMap eredita le proprietà di un grafo. In più il calcolo della funzione costo di ogni singolo oggetto Road che contiene dipende da un parametro globale

$$\text{Costo} = p * L_{ij} + (1-p) * T_{ij}$$

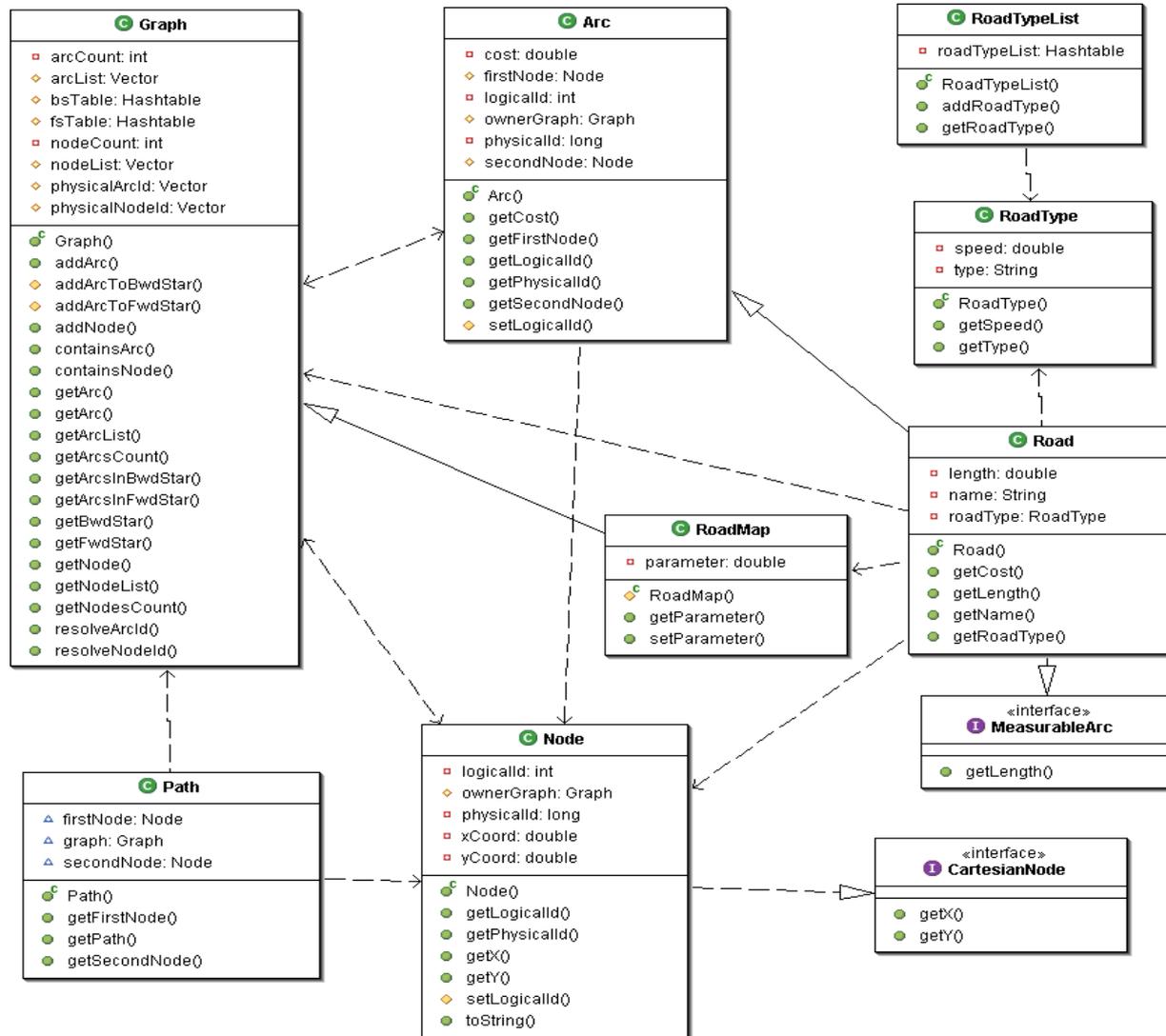
- L_{ij} è la lunghezza dell'arco dal nodo i al nodo j
- T_{ij} è il tempo medio di attraversamento dell'arco

Le entità della libreria : *Path*

Path	
△	firstNode: Node
△	graph: Graph
△	secondNode: Node
● ^c	Path()
●	getFirstNode()
●	getPath()
●	getSecondNode()

- *Che cosa caratterizza un cammino?*
- Un oggetto di tipo Path rappresenta un cammino dal nodo di partenza (firstNode) al nodo di arrivo (secondNode). Il cammino è dato sotto forma di grafo

Le entità della libreria



Libreria : costruire un grafo

- Creare un grafo non è un compito difficile, ma richiede molte istruzioni: diventa quindi una potenziale fonte di errori
- Per un grafo completamente connesso con 6 nodi
 - 1 istruzione per creare il grafo
 - 6 istruzioni per creare i nodi
 - 6 istruzioni per aggiungere i nodi al grafo
 - $N*(N-1) = 30$ istruzioni per creare gli archi (essendo orientati sono il doppio)
 - 30 istruzioni per aggiungere gli archi al grafo
- Totale 73 istruzioni per creare un grafo "piuttosto semplice"!

Libreria : il pattern Factory

- Il metodo di creazione manuale è inefficiente
- Si è scelto di utilizzare quindi il pattern Factory: si richiede la creazione dell'istanza di un grafo ad una specifica classe
- Vantaggi
 - il codice è più facile da collaudare
 - il grafo è indipendente dalla sorgente fisica dei dati
 - si possono aggiungere più sorgenti dati

Libreria : NodeFactory

C NodeFactory	
▣	con: Connection
▣	nodeDBMS: String
SF	nodeIdTag: String
▣	nodeList: ResultSet
▣	nodeTable: String
SF	nodeXCoordTag: String
SF	nodeYCoordTag: String
▣	pswd: String
▣	userId: String
C	NodeFactory()
●	scan()

- Questa Factory è stata progettata in modo specifico per la lettura dei Nodi dal database della rete stradale di Bologna
- Non va utilizzata direttamente!
- Occorre avere il database della rete stradale di Bologna... che non è di pubblico dominio !

Libreria : ArcFactory

C ArcFactory	
SF	arcCostTag: String
▣	arcDBMS: String
SF	arcIdTag: String
▣	arcList: ResultSet
SF	arcNameTag: String
SF	arcSinkTag: String
SF	arcSourceTag: String
▣	arcTable: String
SF	arcTypeTag: String
▣	con: Connection
▣	currentReverseIndex: int
SF	oneWayTag: String
▣	pswd: String
▣	types: RoadTypeList
▣	userId: String
C	ArcFactory()
●	scan()

- Questa Factory è stata progettata in modo specifico per la lettura degli Archi dal database della rete stradale di Bologna
- Non va utilizzata direttamente!
- Occorre avere il database della rete stradale di Bologna... che non è di pubblico dominio !

Libreria : ArcFactory

C GraphFactory	
C	GraphFactory()
S	getRoadMapInstance()
S	getTestGraphInstance()
S	getTestInstance()
S	getTestInstance()

- Questa Factory crea istanze di grafi.
- `getRoadMapInstance()` caricamento della rete stradale di Bologna (richiede il DB)
- `getTestInstance()` crea un grafo di tipo "Manhattan", il primo con valori di default (10x10), il secondo in modo parametrico (righe x colonne)
- `getTestGraphInstance()` carica i dati dalla TSPLIB: in questo modo si potrà risolvere il problema del commesso viaggiatore (TSP) conoscendo la soluzione ottima

Libreria : GraphUtils

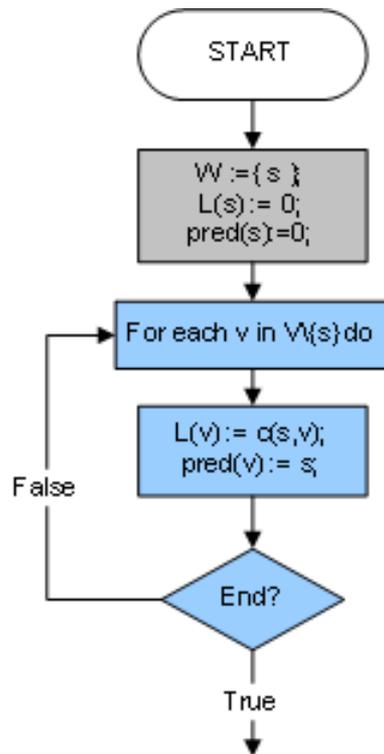
GraphUtils
<ul style="list-style-type: none">● Dijkstra()●^c GraphUtils()● getEquivalentGraph()● getExpandedGraph()■ getPathTotalCost()● nClientsShortestPath()● pathUnion()● twoPhaseSP()
NodeLabel
<ul style="list-style-type: none">● compareTo()● getCostFromS()● getPrevNode()● setCostFromS()● setPrevNode()● toString()

- GraphUtils contiene delle funzioni che operano sui grafi
- I metodi *Dijkstra()* e *twoPhaseShortestPath()* implementano due diversi algoritmi per il calcolo del cammino minimo
- Il metodo *nClientShortestPath()*, dato un vettore di clienti ed un grafo, restituisce il vettore di cammini che uniscono i clienti presi a coppie
- Il metodo *getEquivalentGraph()* restituisce un grafo equivalente partendo dal risultato di *nClientShortestPath()*. Ogni cammino viene sostituito da un arco equivalente.
- Il metodo *pathUnion()* costruisce un grafo dato un vettore di cammini
- Il metodo *getEquivalentGraph()* "espande" un grafo equivalente sul grafo originale

Dijkstra : l'algoritmo

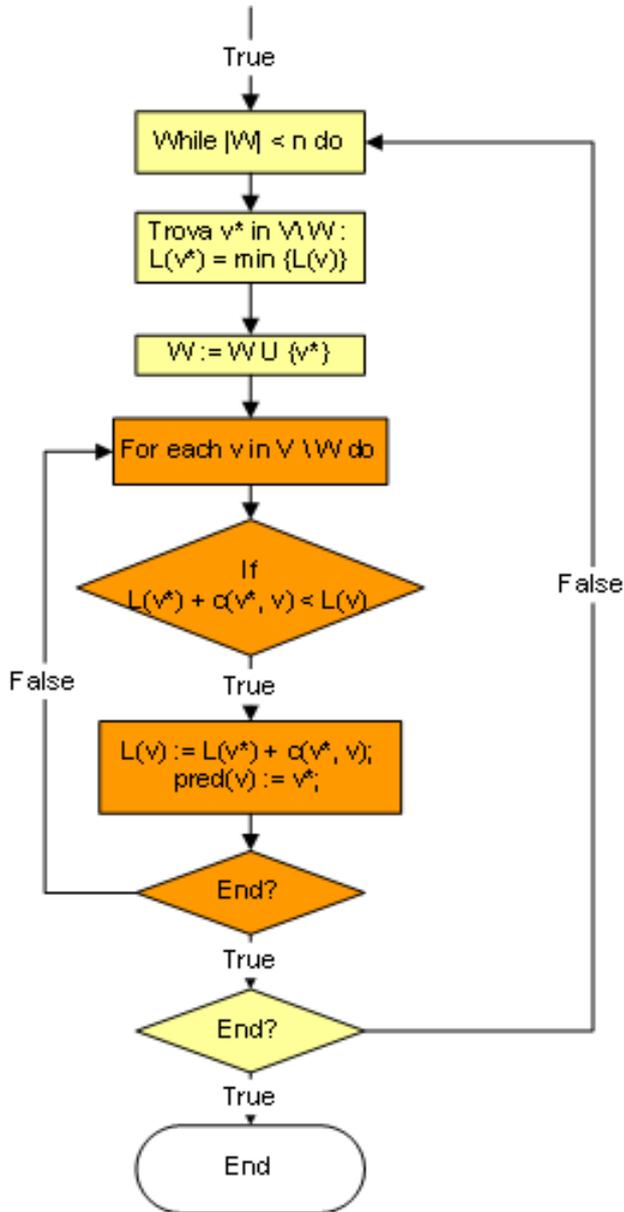
- Dati un grafo Γ e una coppia di vertici (s, t) tale algoritmo permette di calcolare il percorso minimo da s a t su Γ , dove gli archi di Γ hanno tutti costo positivo
- L'algoritmo opera sulle seguenti strutture dati
 - W = insieme dei vertici raggiunti in modo permanente da s
 - $L(v)$ = costo del cammino min. da s a v attraverso $j \in W$
 - $pred(v)$ = vertice che precede v nel cammino da s a v

Dijkstra : l'algoritmo



- A lato è riportato il diagramma di flusso relativo alla fase di inizializzazione dell'algoritmo di Dijkstra
- Il primo blocco (in grigio) si occupa di inizializzare l'etichetta del nodo sorgente, che viene inserito nell'insieme dei nodi visitati
- Il secondo blocco (in azzurro) inizializza le etichette di tutti i nodi del grafo. $L(v)$ viene impostato al costo dell'arco da s a v (infinito se v non è nella forward star di s)

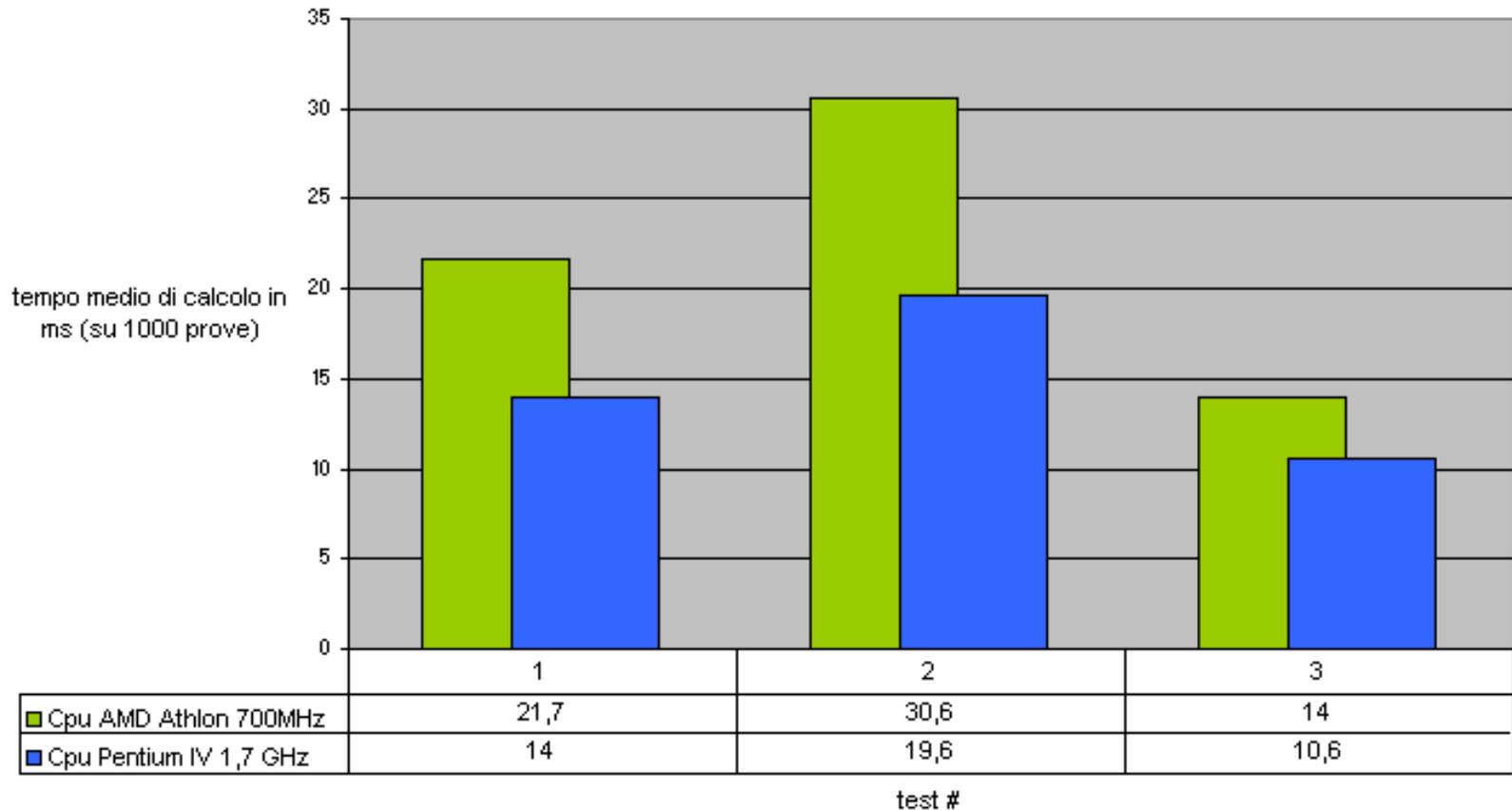
Dijkstra : l'algoritmo



- Il terzo blocco (in giallo) è il ciclo principale dell'algoritmo e viene eseguito fino a che tutti i vertici non sono stati raggiunti in maniera permanente, cioè fino a che non sono stati etichettati con il costo minimo
- Per prima cosa viene ricercato un nodo per cui il costo da s sia minimo; esso viene inserito nell'insieme dei nodi raggiunti
- Il quarto blocco (in arancio) è interno al terzo blocco. Partendo dal nodo appena trovato si controlla se è possibile aggiornare il campo $L(v)$ relativo alle etichette dei nodi non ancora raggiunti

Dijkstra() : prestazioni

Confronto fra tempi medi di calcolo su diverse piattaforme



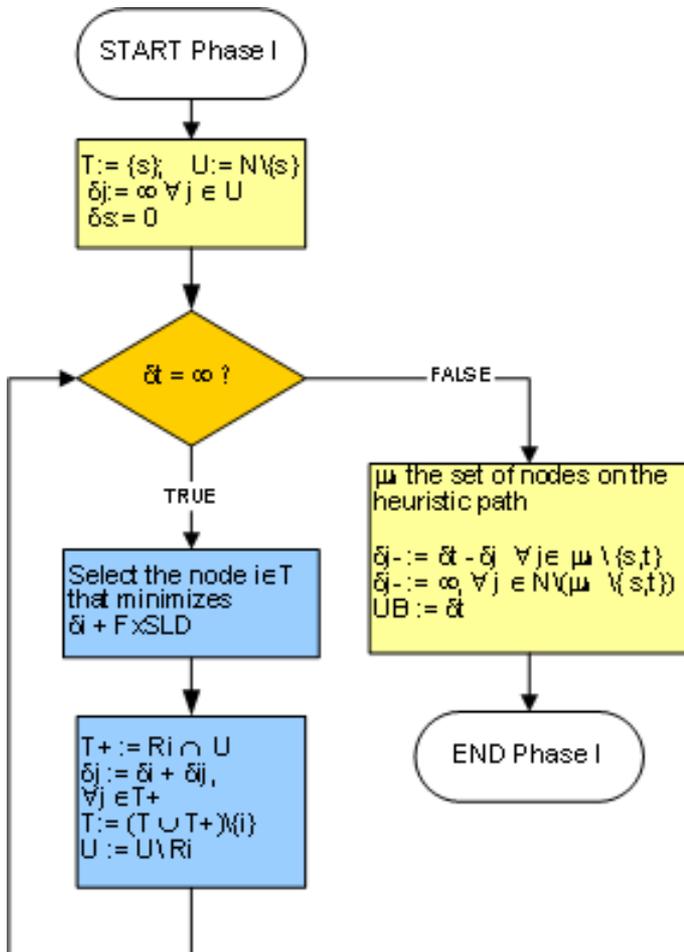
Algoritmo a due fasi

- Dati un grafo Γ e una coppia di vertici (s, t) tale algoritmo permette di calcolare il percorso minimo da s a t su Γ , dove i nodi sono rappresentabili su un piano cartesiano ed ad ogni coppia di nodi è possibile associare una distanza euclidea, espressa dalla seguente formula

- $$SLD_{IJ} = \sqrt{(x_I - x_J)^2 + (y_I - y_J)^2}$$

- Nella Fase I viene calcolato un percorso fra il nodo s e t utilizzando un criterio euristico.
- Nella Fase II la lunghezza del cammino euristico viene utilizzata per ridurre l'insieme dei nodi da visitare per la costruzione del cammino minimo tra s e t .

Algoritmo a due fasi



T = insieme dei nodi temporaneamente

etichettati

T^+ = insieme dei nodi da aggiungere ad ogni iterazione

U = insieme dei nodi non etichettati

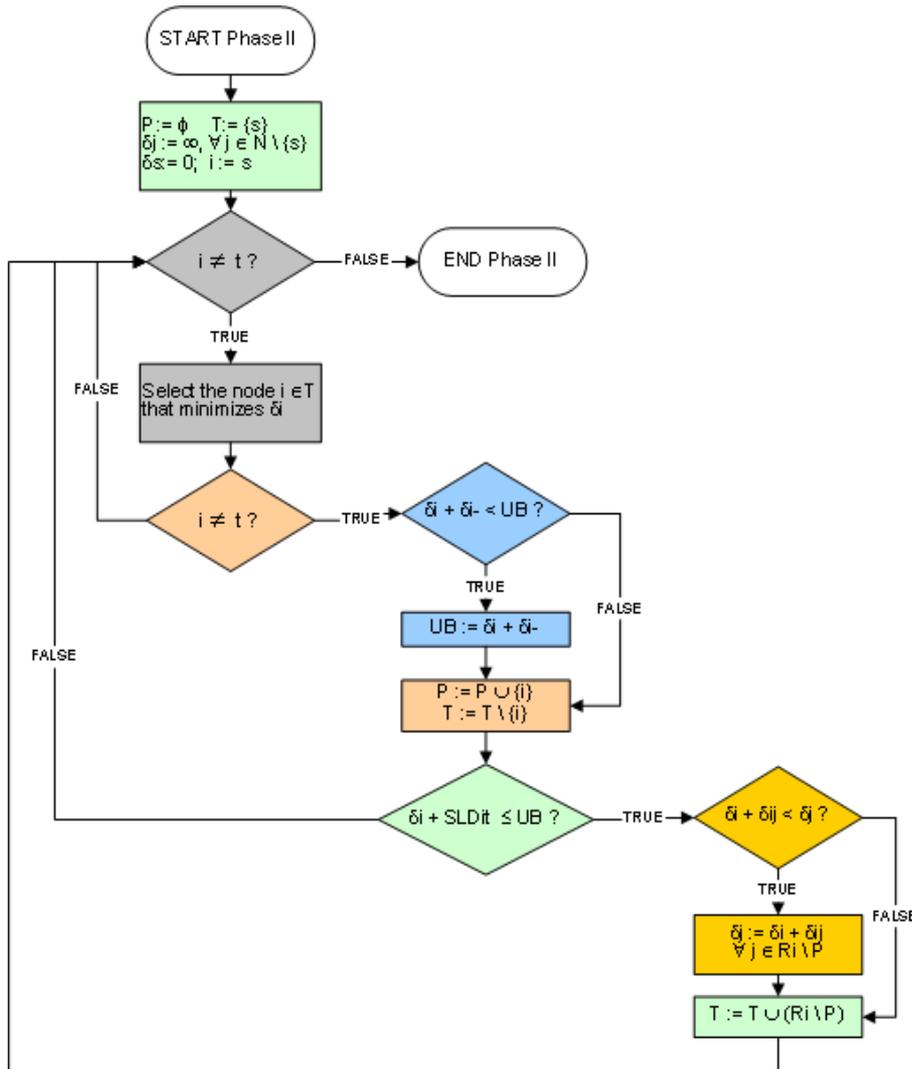
F = peso associato a SLD

R_i = forward star del nodo i

- La Fase I termina quando t viene etichettato. Ad ogni ciclo si seleziona un nodo che minimizza la somma fra il costo associato al nodo e SLD pesata dal fattore F . Del nodo selezionato viene aggiornata la forward star.

- Terminato il ciclo, ad UB si assegna la lunghezza del cammino trovato. Si inizializza la struttura dati contenente le etichette dei nodi appartenenti al cammino euristico.

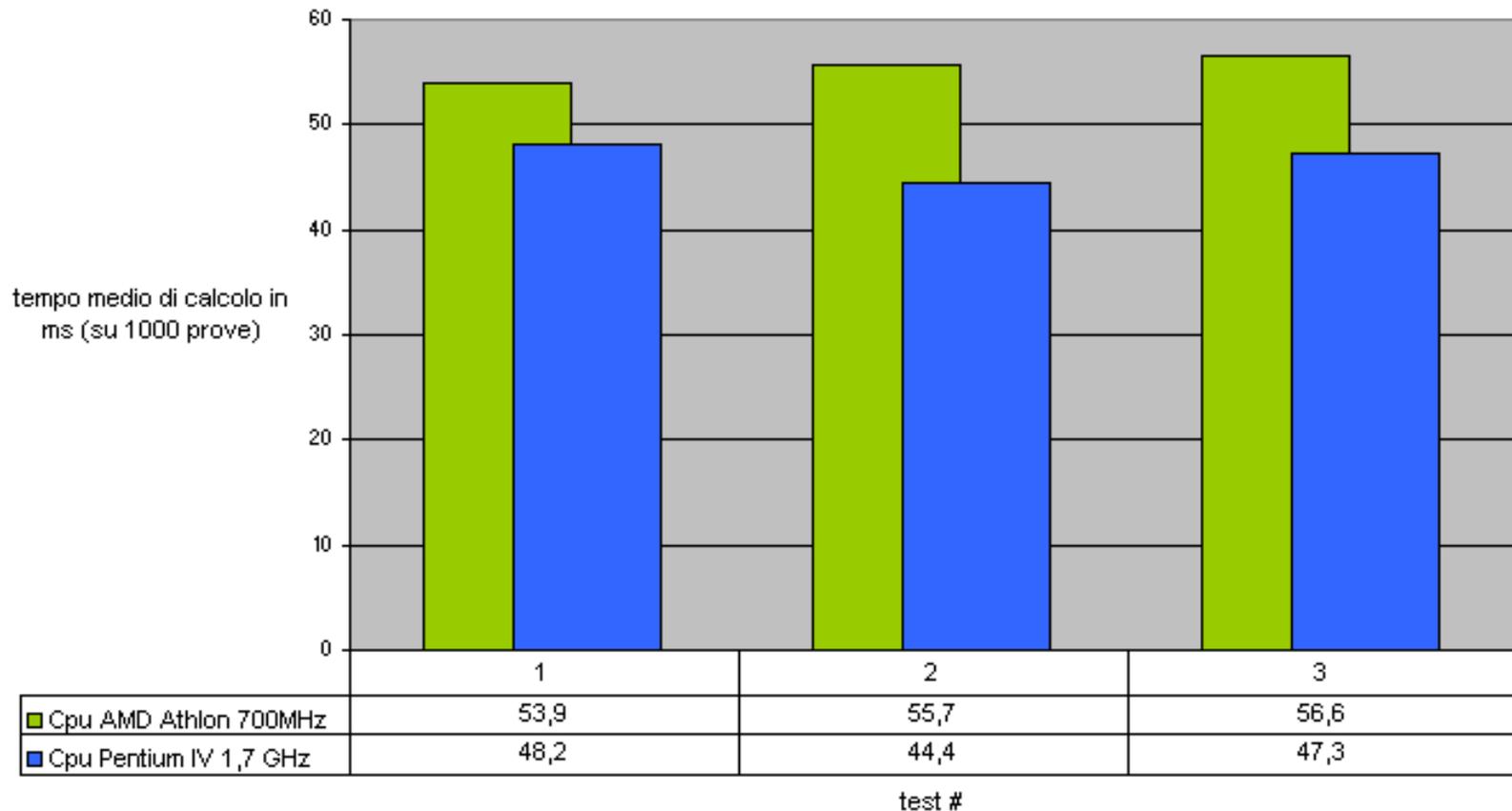
Algoritmo a due fasi

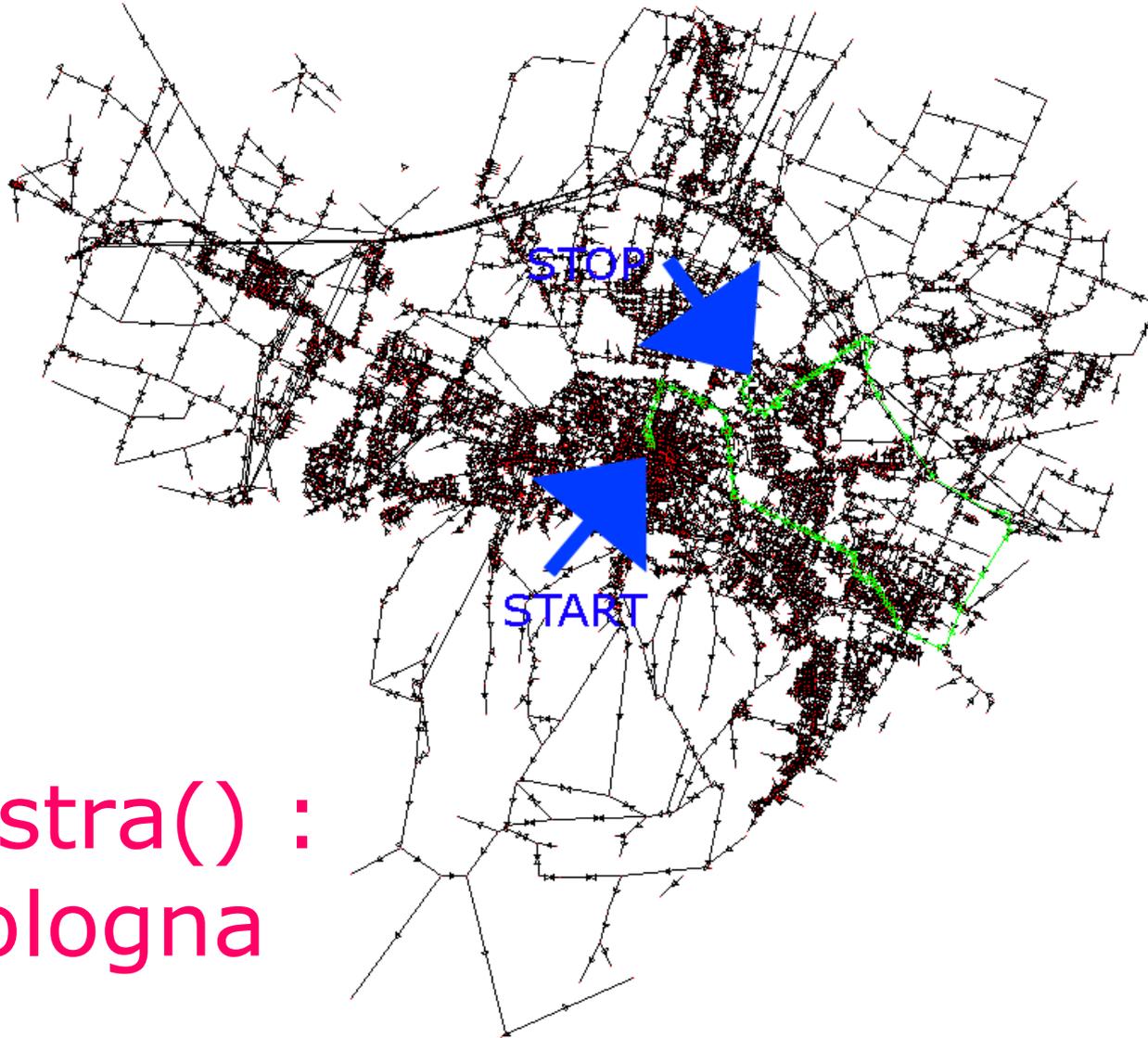


- La fase II termina quando il nodo corrente è il nodo t
- Ad ogni iterazione si seleziona il nodo che minimizza la distanza dal nodo sorgente
- Si valuta se l'upper bound deve essere aggiornato, quindi si aggiunge il nodo corrente all'insieme dei nodi visitati rimuovendolo da quelli etichettati
- Se la distanza cartesiana da questo nodo al nodo t è minore di UB allora si aggiornano i vertici nella forward star del nodo corrente

Algoritmo a due fasi : prestazioni

Confronto fra tempi medi di calcolo su diverse piattaforme





Dijkstra() : Bologna



Dijkstra() :
ecco perché...

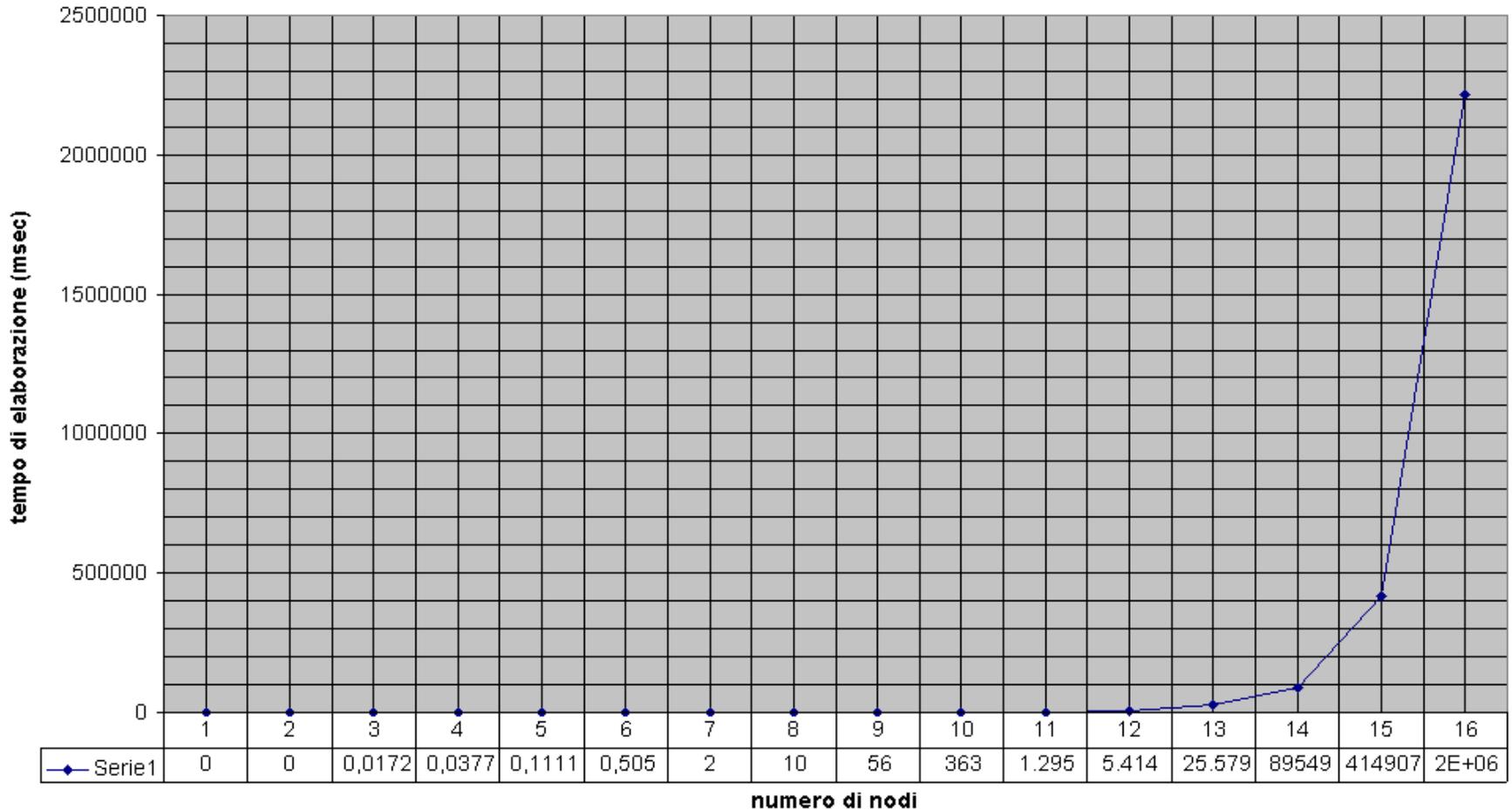
Libreria : TSP

C TSP	
▣	best[0..*]: Node
▣	cost: double
▣	found: boolean
▣	graph: Graph
▣	journey[0..*]: Node
▣	minCost: double
▣	nodeNumber: int
▣	result: Graph
▣	source: Node
▣	step: int
▣	visited: Vector
◆	tsp()
●	tspBranchBound()

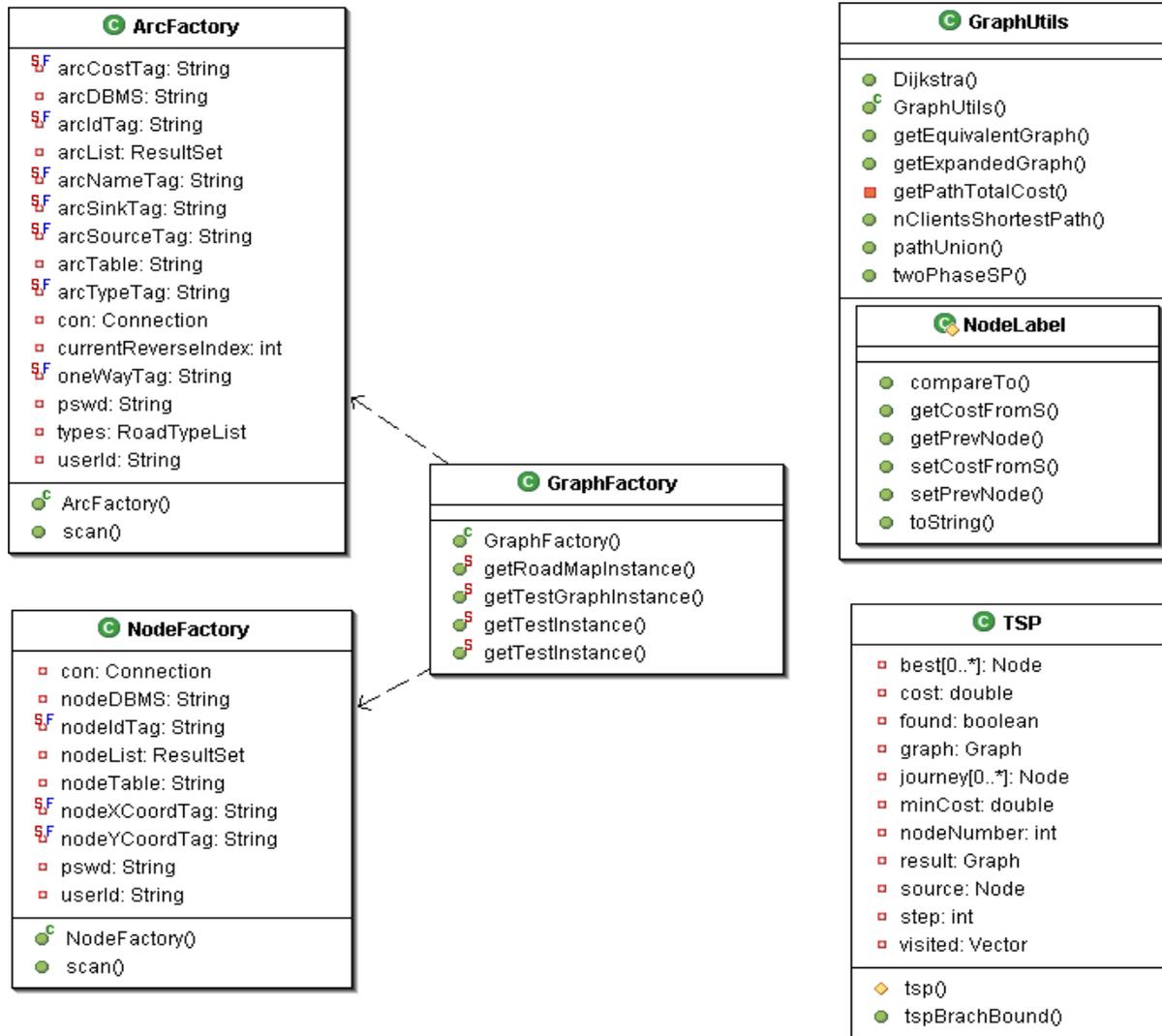
- La classe TSP risolve il problema del commesso viaggiatore (TSP)
- Dati un nodo ed un grafo il metodo `tspBranchBound()` restituisce il percorso di visita di costo minimo sotto forma di grafo
- La complessità di questo metodo è esponenziale, quindi questo approccio è inadatto a trattare problemi con un numero di nodi grande
- In particolare già con 15 nodi il tempo di calcolo sale a circa 7 minuti!
- Con 16 nodi impiega circa 39 minuti!

TSP() : prestazioni

Prestazioni TSP (soluzione ottima)



Libreria : funzionalità



Visualizzare i risultati

- Risulta interessante poter visualizzare i risultati delle funzionalità descritte fino ad ora
- Vantaggi
 - Debugging: più facile verificare/analizzare il comportamento degli algoritmi
 - Produttività: si possono anche ottenere delle immagini utili per articoli, materiale didattico, relazioni
 - Commerciale: il committente è più contento ed è più facile attestare il rispetto dei requisiti
 - Web: da remoto come Applet... perché no!

Libreria : drawing tools

 TBFram
<ul style="list-style-type: none">screenSize: DimensionxPos: intyPos: int
 TBFram()

 Trasformazioni
<ul style="list-style-type: none"> ruotaPunto() traslaPunto()

 Triangle
<ul style="list-style-type: none">arrowHeadDim: intp1: Pointp2: Pointp3: Point
<ul style="list-style-type: none"> Triangle()reset()rotate()translate()

- Ogni applicazione grafica necessita di una finestra. *TBFram* è una semplice estensione di JFrame
- Spesso è necessario eseguire delle trasformazioni di coordinate di punti. Di questo si occupa *Trasformazioni*
- La cosa più complessa da disegnare è il triangolo che rappresenta la direzione di una freccia. *Triangle* realizza questa astrazione. E' importante che la freccia sia orientata come l'arco...

MyPanel	
▣	arcColor: Color
▣	arcCostColor: Color
▣	arrowDimension: int
▣	aspectRatio: double
▣	backgroundColor: Color
▣	colorArray[0..*]: Color
▣	currentGraph: Graph
▣	drawArcCost: boolean
▣	graphVector: Vector
▣	lineWidth: int
▣	nodeColor: Color
▣	nodeDiameter: int
▣	oldColor: Color
▣	originX: double
▣	originY: double
▣	panelHeight: int
▣	panelWidth: int
▣	textColor: Color
▣	textFont: Font
●	MyPanel()
●	clear()
◆	drawArc()
●	drawArcCost()
◆	drawArrow()
◆	drawCost()
◆	drawGraph()
◆	drawGraphs()
◆	drawNode()
●	paint()
◆	resolveCoords()
●	setArcColor()
●	setArcCostColor()
●	setArrowDimension()
●	setBackgroundColor()
◆	setDrawingParams()
●	setFont()
●	setGraph()
●	setGraphs()
●	setNodeColor()
●	setNodeDiameter()
●	setTextColor()

Libreria : drawing tools

- Le funzioni di disegno sono state implementate in *MyPanel*, il pannello di visualizzazione, il quale estende *JPanel*
- Usando i metodi `setGraph()` e `setGraphs()` è possibile impostare uno o più grafi da disegnare. Automaticamente il pannello calcola il fattore di scala per visualizzare tutti i grafi in un'unica schermata
- E' possibile agire sui parametri di visualizzazione in termini di colore, dimensioni degli elementi e font
- Scenderemo più nei dettagli con i casi d'uso

Domande?

