

Università degli Studi di Milano
Dipartimento di Scienze dell'Informazione

Esame di Algoritmi II
Prof. A. Bertoni

Laboratorio di Analisi di Algoritmi Genetici
applicati al problema del Commesso Viaggiatore

Jacopo Mantovani, Franco Salvetti

12 giugno 2002

Abstract

In questo documento presentiamo un applicativo per lo studio ed il confronto di diversi approcci genetici alla risoluzione del problema del Commesso Viaggiatore (trovare il percorso chiuso piú breve che passi una ed una sola volta per tutti i vertici di un grafo) nella sua versione euclidea, ovvero quella in cui le distanze tra i vertici sono calcolate come distanze euclidee, ed il grafo é completamente connesso. In considerazione della mancanza di strumenti teorici definitivi per l'analisi e la determinazione dei parametri ottimali che governano un algoritmo genetico, si é scelto di mettere a disposizione dell'utente le diverse tecniche di selezione, crossover e mutazione presenti in letteratura; egli puó combinarle, vedere e confrontare i risultati ottenuti, anche su diverse topologie di grafo.

Contents

1	Il problema del Commesso Viaggiatore (TSP)	4
1.1	Cenni sulla complessità	5
1.2	Due lower bound per il TSP	5
1.3	Un esempio di applicazione	5
2	Gli Algoritmi Genetici	6
3	Algoritmi Genetici applicati al Problema del Commesso Viaggiatore	7
3.1	Rappresentazione vettoriale	7
3.2	Rappresentazione matriciale	8
4	Euclidean TSPGenLab	9
4.1	Scelte di progetto	9
4.1.1	Input	9
4.1.2	Output	10
4.1.3	Strutture dati	11
4.1.4	L'algoritmo	12
4.1.5	Il calcolo della Fitness	13
4.1.6	Il calcolo del Kick Off	13
4.1.7	Tecniche di selezione	14
4.1.8	Tecniche di crossover	15
4.1.9	Tecniche di mutazione	17
4.1.10	Generazione di permutazioni random	18
4.1.11	L'Algoritmo di Prim per il Minimum Spanning Tree	19
4.1.12	Il thread dell'algoritmo	19
4.2	L'Interfaccia Utente	20
4.2.1	Il riscaldamento automatico dell'informazione Grafica	22
4.3	La Ricerca Locale Randomizzata	22
5	Conclusioni	23
6	Miglioramenti futuri	23

1 Il problema del Commesso Viaggiatore (TSP)

Un commesso viaggiatore deve visitare un certo numero di città una e una sola volta, tornando infine a quella di partenza, e compiendo il cammino più breve. Nella sua versione euclidea, il grafo delle città è completamente connesso, e le distanze tra le città sono quelle euclidee.

Formalmente, dato un grafo $G = \langle V, E \rangle$, si definisce *hamiltoniano* un cammino $P = (v_1, v_2, v_3, \dots, v_k)$, tale per cui $k = |V|$ e $v_i \neq v_j, \forall i, j$. In altre parole P tocca tutti i vertici del grafo una ed una sola volta. Si definisce ora *ciclo hamiltoniano* un cammino hamiltoniano in cui il vertice di arrivo coincide con quello di partenza. Il problema del Commesso Viaggiatore consiste nel trovare un ciclo hamiltoniano di costo minimo su di un grafo non orientato pesato $G = \langle V, E, F \rangle$, dove F è l'insieme che contiene le distanze tra i vertici, ovvero i pesi degli archi: $f : E \rightarrow IR$. Nel nostro caso F contiene le distanze euclidee. Visto come problema di programmazione lineare intera, è rappresentabile nel seguente modo: Sia $x_{i,j} = 1$ se l'arco (i, j) appartiene al ciclo, 0 altrimenti, avendo posto che:

$$x_{i,j} \in \{0, 1\}$$

Si hanno i seguenti vincoli di flusso:

$$\sum_{i=n}^{i=1} x_{i,j} = 1$$

$$\sum_{i=n}^{j=1} x_{i,j} = 1$$

ed i seguenti vincoli di connessione del grafo:

$$\exists V_1, V_2 : V_1 \cup V_2 = V \wedge V_1 \cap V_2 = \emptyset$$

$$\sum_{i \in V_1} \sum_{j \in V_2} x_{i,j} \geq 1$$

La funzione obiettivo è:

$$\sum_{i,j=1}^n d(v_i, v_j) \cdot x_{i,j}$$

ed il numero di vincoli è $O(n)$.

1.1 Cenni sulla complessit 

Lo spazio di ricerca per il TSP   l'insieme delle permutazioni di n citt , dove n   il numero di vertici del grafo. La complessit  del problema cresce molto rapidamente con le dimensioni del grafo: vi sono infatti $\frac{(n-1)!}{2}$ possibili soluzioni da valutare, che   un numero decisamente grande, pi  che esponenziale. Infatti:

$$\lim_{m \rightarrow \infty} \frac{a^m}{m!} = 0$$

Consideriamo, ad esempio, di avere 60 citt , che poi   proprio la configurazione di default che proponiamo nel nostro programma: il numero di possibili percorsi diversi esplose a $\frac{(60-1)!}{2} = 6,9341 \cdot 10^{79}$. Il problema del Commesso Viaggiatore   dimostrato essere NP-Completo¹, mentre la sua versione euclidea non   dimostrata tale.

1.2 Due lower bound per il TSP

Una possibile misura della *bont * di una soluzione del problema del Commesso Viaggiatore   la sua distanza dalla lunghezza dell'albero di copertura minimo (Minimum Spanning Tree). Largamente usata in letteratura, essa   sempre strettamente minore della soluzione ottima del nostro problema, quindi mai le due coincideranno. Come per  vedremo nel seguito di questo documento, questa misura di *bont * risulta particolarmente utile nel caso di algoritmi approssimati, quali quelli genetici. Un altro lower bound   costituito dalla soluzione del rilassamento della formulazione del TSP visto come problema di programmazione lineare intera (formulazione che   stata data poco sopra). Esso prende il nome di lower bound di Held-Karp, e viene calcolato in tempo polinomiale, nonostante il fatto che vi sia un numero esponenziale di vincoli.

1.3 Un esempio di applicazione

Nel ciclo di produzione delle schede elettroniche monostrato, esiste una fase in cui devono essere praticati i fori in cui saranno poi alloggiati (e saldati) i pin dei componenti. L'operazione di foratura necessita di un tempo proporzionale alla lunghezza del percorso eseguito dalla punta per muoversi da un foro all'altro. Minimizzare il percorso della punta (riportandola al termine nella posizione iniziale di riposo) corrisponde a risolvere il TSP sul grafo che rappresenta l'insieme dei fori, pesato con le distanze geometriche che li separano.

¹Un problema appartiene all'insieme dei problemi NP-Completi se   all'interno di NP ed   un massimo rispetto ad una relazione di riduzione su NP, dove NP   la classe dei problemi risolvibili da una macchina di Turing non deterministica in tempo polinomiale.

2 Gli Algoritmi Genetici

Gli algoritmi genetici sono un modello di calcolo che ricalca i meccanismi di evoluzione biologica presenti in natura. I processi evolutivi naturali sono riconducibili, ad una prima analisi, ad una sorta di competitività tra individui rispetto all'ambiente che li circonda: chi tra questi è dotato di caratteristiche che lo fanno resistere ad esso, ha maggiori probabilità di sopravvivere e *propagare* il proprio patrimonio genetico ai propri discendenti; chi invece non ha le caratteristiche adatte per sopravvivere all'ambiente, sicuramente ha maggiori probabilità di perire, non propagando così il proprio (con buone probabilità pessimo) patrimonio genetico alle generazioni future. Inoltre, il susseguirsi delle generazioni porta ad una sorta di *scambio* di geni tra individui, uno scambio di informazione utile alla sopravvivenza. In questo modo, si ha una certa probabilità che un individuo della generazione successiva erediti da due individui (i "genitori") della precedente determinate caratteristiche che li hanno fatti sopravvivere. Da ultimo sempre, in natura, ricorre l'elemento della casualità. Di fatto, è sempre possibile che da una generazione di individui all'altra vi siano alcuni di essi che presentino delle caratteristiche non derivate dai propri "genitori". Quella che viene definita *mutazione* interviene in maniera casuale su alcuni individui, come a voler "rinnovare" la specie, introducendo o, appunto, mutando, alcune caratteristiche.

In analogia a tutto ciò, gli algoritmi genetici procedono generalmente nel seguente modo:

- È data una popolazione iniziale di individui, costruita in modo arbitrario;
- A ciascuno di essi viene attribuito un valore, detto *fitness*, ad indicarne la bontà, ovvero la capacità di sopravvivenza;
- Viene applicato l'operatore di *Selezione*, con probabilità proporzionale alla fitness.
- A coppie, a tutti gli individui viene applicato, con una certa probabilità un operatore, detto *crossover*, che fa sì che vi sia scambio di informazione utile;
- A ciascuno degli individui così ottenuti, viene applicato, con una data probabilità, l'operatore di *mutazione*, che introduce quell'elemento di casualità nella popolazione descritto sopra;
- È ottenuta così una nuova popolazione, alla quale verranno riapplicati gli operatori appena descritti, nello stesso ordine.

3 Algoritmi Genetici applicati al Problema del Commesso Viaggiatore

Nel caso del Commesso Viaggiatore, agli algoritmi genetici classici si aggiunge una peculiarità: i cromosomi, che intendono rappresentare i possibili percorsi, hanno la proprietà di essere permutazioni. Non è dunque più possibile applicare classicamente gli operatori di crossover e mutazione, pena la possibile generazione di cromosomi non validi (che non sono più permutazioni). Due sono le soluzioni possibili a questo problema:

- applicare comunque gli operatori in modo classico, verificare che essi producano sequenze legali, e nel caso esse non lo siano, sostituirle con "le più prossime" tra quelle legali (una sorta di *generate and test*);
- applicare degli operatori di mutazione e crossover *ad hoc* che preservino la proprietà dei cromosomi di essere permutazioni.

Inoltre, vi sono numerose rappresentazioni possibili (ognuna delle quali è supportata dai propri operatori genetici) per questo problema, ciascuna con le proprie particolarità.

3.1 Rappresentazione vettoriale

1. **Adiacente:** un percorso è codificato come un vettore, in cui la città j è indicata dalla posizione i se e solo se ci si muove dalla città i alla città j . Non tutti vettori sono validi (ovvero permutazioni), possono infatti formarsi dei sottopercorsi parziali con la presenza di cicli, che devono essere risistemati con un algoritmo *riparatore*. Questa rappresentazione non è compatibile con i tradizionali crossover e mutazioni, ma si presta molto bene all'analisi degli schemi secondo Holland.
2. **Ordinale:** in questa rappresentazione ci si avvale di un vettore ordinato come riferimento. Il vettore che rappresenta il cromosoma indica passo dopo passo la posizione della successiva città nel percorso rispetto alla lista di riferimento. Mostriamone il funzionamento attraverso un esempio:

Cromosoma	3 4 1 3 1 2 2 2 1
Vettore ordinato	1 2 3 4 5 6 7 8 9
Vettore risultato	- - - - - - - - -

Il primo numero del percorso è 3, quindi si prende la terza città della lista del vettore ordinato, la si toglie da esso, e la si posiziona nel primo posto libero del vettore risultato:

Cromosoma	3 4 1 3 1 2 2 2 1
Vettore ordinato	1 2 4 5 6 7 8 9 X
Vettore risultato	3 - - - - - - - -

Si procede con il secondo numero del percorso, che é 4. Si cerca la quarta città nel vettore ordinato, la si toglie da esso, e la si posiziona nel primo posto libero del vettore risultato:

Cromosoma	3 4 1 3 1 2 2 2 1
Vettore ordinato	1 2 4 6 7 8 8 X X
Vettore risultato	3 5 _ _ _ _ _ _

Si itera il procedimento n volte, e si ottiene il percorso, che effettivamente é una permutazione di città. Questo tipo di rappresentazione permette di applicare il crossover classico al cromosoma, senza compromettere la proprietà di essere una permutazione.

3. **Sentiero:** in questo caso un vettore rappresenta una lista delle città visitate. Ad esempio:

Cromosoma	3 4 1 3 1 2 2 2 1
-----------	-------------------

rappresenta il percorso fatto partendo dalla città numero 3, passando poi alla numero 4, poi alla 1, ecc.. Un crossover classico qui non é ovviamente applicabile, ma in letteratura se ne trovano numerosi che conservano le permutazioni. Holland (1975), per esempio, ha mostrato che l'operatore chiamato *Sub Tour Inversion* (di cui spiegheremo il funzionamento nel seguito), risulta utile per ricercare buoni ordini nelle sequenze di città.

3.2 Rappresentazione matriciale

Presentiamo qui di seguito due metodi di rappresentazione del TSP:

1. Un percorso é dato da una matrice binaria M i cui elementi $m_{i,j}$ contengono un 1 se e solo se la città i compare nel percorso prima della città j . Questa notazione incapsula tutte le informazioni relative alla sequenza delle città toccate dal percorso.
2. L'elemento $m_{i,j}$ contiene un 1 se e solo se, nel tour, c'è un percorso diretto che va dalla città i alla città j . Non tutte le matrici con questa caratteristica rappresentano un percorso valido, perché potrebbero presentarsi uno o più cicli che si richiudono su se stessi. In questo approccio i cicli non vengono corretti immediatamente, anzi vengono permessi, almeno in una prima fase, nella speranza che possano evidenziarsi dei sottogruppi efficienti in modo naturale.

4 Euclidean TSPGenLab

Euclidean TSPGenLab é un *laboratorio* per la sperimentazione dell'utilizzo degli algoritmi genetici per la risoluzione del problema del Commesso Viaggiatore nella versione euclidea. In considerazione della mancanza di strumenti teorici definitivi per l'analisi e la determinazione dei parametri ottimali che governano un algoritmo genetico, si é scelto di mettere a disposizione dell'utente le diverse tecniche di selezione, crossover e mutazione presenti in letteratura; l'utente puó combinarle, vedere e confrontare i risultati ottenuti, anche su diverse topologie di grafo. Questo approccio consente, oltre che di determinare una soluzione approssimata, di percepire ed apprendere come quest'ultima dipenda in modo *sensibile* dai parametri in ingresso. La libertá di azione concessa dal programma permette di esasperare i *comportamenti* di un algoritmo genetico, ad esempio applicando la sola mutazione o il solo crossover (producendo la *morte* dell'algoritmo). L'exasperazione nelle scelte dei parametri puó portare l'algoritmo genetico a degenerare in una ricerca locale randomizzata, producendo in determinati casi una soluzione che coincide con quella ottima, in un numero di generazioni relativamente basso. Da ultimo, al fine di fornire degli indicatori che guidino nella scelta dei parametri *ottimali*, sono visualizzati a video, sia graficamente che numericamente, una serie di valori quali la varianza e la lunghezza del cammino corrente, la media dei cammini, il lower bound, la distanza dal lower bound, la lunghezza del cammino migliore.

4.1 Scelte di progetto

Il progetto é stato interamente scritto in linguaggio Java. In quanto *orientato agli oggetti*, esso ci ha permesso di programmare ad un alto livello di strutturazione, consentendoci di implementare una componente di interfaccia, nonché di realizzare una Applet da poter porre sul *web*.

4.1.1 Input

Si é deciso di lasciare la piú ampia libertá all'utente, permettendogli di modificare pressoché tutti i parametri che coinvolgono l'algoritmo genetico. Egli puó variare con delle *combo box* il tipo di selezione, di crossover e di mutazione da applicare; con degli *slider* le probabilità di crossover e di selezione, nonché due parametri aggiuntivi detti *Kick off* e *Elitism*, che descriviamo qui di seguito:

- *Kick off*: attraverso questo slider é possibile impostare una percentuale calcolata sul massimo valore di fitness, per cui i cromosomi che hanno fitness che sta al di sotto di quella percentuale vengono scartati.
- *Elitism*: dati i valori di fitness di una popolazione, questi vengono elevati ad una potenza data da questo slider; in questo modo, gli individui migliori vengono premiati.

Da ultimo, l'utente può modificare la topologia del grafo, in quattro modi possibili:

- *US Cities*: vengono visualizzate sessanta città degli Stati Uniti: Seattle, S. Francisco, Eureka, Los Angeles, S. Diego, Tucson, El Paso, Sierra Blanca, Del Rio, Brownsville, Corpus Christi, Houston, Yuma, New Orleans, Pensacola, Tallahassee, Tampa, Fort Meyers, Miami, Daytona Beach, Brunswick, Charleston, Norfolk, Washington, Filadelfia, New York, Boston, Augusta, Montpelier, Buffalo, Cleveland, Toledo, Detroit, Grand Rapids, Chicago, Minneapolis, Gran Forks, Williston, Kalispel, Boulder, Atlanta, Austin, Cincinnati, Dallas, Denver, Indianapolis, Kansas City, Las Vegas, Louisville, Memphis, Nashville, New Haven, Oklahoma City, Phoenix, Pittsburgh, Portland, Salt Lake City, S. Antonio, Santa Fe, Springfield.
- *Grid*: i vertici del grafo vengono posti ai vertici di una griglia. L'utente può impostare il numero di città per lato della griglia con uno slider. Questa topologia permette all'utente di cogliere anche intuitivamente quanto l'algoritmo si avvicini alla soluzione, vista l'alta regolarità della disposizione delle città.
- *Random*: i vertici del grafo vengono posizionati casualmente, avendo però cura di non avvicinarli troppo, per evitare che a video non si riesca a distinguerli. Anche qui, uno slider permette di decidere la quantità di vertici.
- *Circle*: le città del grafo vengono poste ai vertici di un poligono regolare inscritto in una circonferenza. Il numero di vertici del poligono è dato dallo slider. Anche in questo caso, l'utente può immediatamente capire se l'algoritmo è arrivato all'ottimo, che è sempre costituito dai lati del poligono inscritto nella circonferenza. Il numero massimo di città consentito per questa configurazione è relativamente basso, ma solo per problemi di visualizzazione: un elevato numero le renderebbe indistinguibili l'una dall'altra.

4.1.2 Output

Si è scelto di mostrare all'utente il maggior numero possibile di dati, sia graficamente che numericamente. Oltre che vedere la progressione del percorso migliore raggiunto fino alla generazione corrente, egli può avvalersi di diversi grafici, che danno un'indicazione della media, così calcolata:

$$\mu = \frac{\sum PathLength[i]}{NrOfChromosomes}$$

della varianza:

$$\forall i \text{ Variance}[i] = (\mu - PathLength[i])^2$$

$$Variance = \sum_i Variance[i]$$

e del miglior individuo corrente, oltre che del miglior individuo raggiunto fino ad allora. La varianza é mostrata a video su una scala diversa, perché altrimenti le sue variazioni a video risulterebbero indistinguibili all'utente. Un ulteriore grafico rappresenta la lunghezza dell'albero di copertura minimo per il grafo scelto. In questo modo si rende possibile un costante confronto dei risultati ottenuti con un lower bound.

In corrispondenza dei grafici si é anche deciso di visualizzare alcuni risultati numerici, quali la lunghezza del percorso migliore ottenuto nella generazione corrente, e quella del percorso migliore finora ottenuto; inoltre sono visualizzati il numero della generazione computata, il numero di vertici del grafo, e la distanza relativa del percorso migliore corrente dal lower bound:

$$\frac{CurrentBestLength - LowerBound}{LowerBound} \times Fattorediscala$$

dove *Fattorediscala* é calcolato come il rapporto tra la lunghezza del primo miglior path ed il lower bound stesso. Questa moltiplicazione si rende necessaria per motivi di visualizzazione.

4.1.3 Strutture dati

Come mostrato precedentemente, vi sono diversi approcci algoritmici al problema del Commesso Viaggiatore, a cominciare dalla scelta delle strutture dati. La nostra scelta é stata di una rappresentazione vettoriale, del tipo *a sentiero*, che abbiamo ritenuto sicuramente piú rappresentativa del problema e immediatamente comprensibile. Ciascun cromosoma, dunque, non sará altro che un vettore di permutazioni di interi (la sequenza delle città), e la popolazione sará un vettore di cromosomi. Essendo un cromosoma una permutazione di città, gli operatori di crossover e mutazione devono agire sui vettori mantenendo la loro proprietá di essere permutazioni.

4.1.4 L'algoritmo

Diamo qui di seguito la struttura con cui viene eseguito l'algoritmo, rimandando alle sezioni successive il dettaglio dell'applicazione dei singoli operatori di selezione, crossover e mutazione. L'intenzione, qui, é di dare semplicemente l'idea del procedimento.

```
Generazione casuale della prima popolazione
```

```
Calcolo:
```

- percorso piu' breve
- varianza
- media
- sua distanza dal lower bound

```
While(# generazioni raggiunto) do{
```

```
    Calcolo della Fitness
```

```
    Applicazione di Selezione in base alla Fitness
```

```
    Applicazione di Crossover selezionato dall'utente,  
        con probabilita' data in input
```

```
    Applicazione di Mutazione selezionato dall'utente,  
        con probabilita' data in input
```

```
    Sulla generazione ottenuta calcolo:
```

- percorso piu' breve
- varianza
- media
- sua distanza dal lower bound

```
}
```

Come si puó notare, l'algoritmo si ferma una volta raggiunto il numero di generazioni impostato dall'utente. In questo modo risulta possibile confrontare diverse configurazioni per lo stesso problema, ma a paritá di condizioni. Prima però di fermarsi completamente, viene visualizzato su una finestra di dialogo il fatto che l'algoritmo si é avvicinato al lower bound, e si chiede all'utente se preferisce proseguire comunque la computazione, con l'intento di migliorare il risultato, oppure se preferisce fermarsi.

4.1.5 Il calcolo della Fitness

L'approccio genetico é intrinsecamente legato al calcolo della fitness. Per fitness noi intendiamo un valore reale che misura quanto un individuo/un cromosoma é adatto a sopravvivere all'ambiente. Nel caso del Commesso Viaggiatore la misura della fitness é legata alla misura della lunghezza del percorso, nel seguente modo:

$$\forall i, PathLengthInverted[i] = MaxPathLength - PathLength[i]$$

Cosí facendo, maggiore é la lunghezza del percorso i -esimo, minore é $PathLengthInverted[i]$. A questo punto si computa:

$$TotalPathLengthInverted = \sum_i PathLengthInverted[i],$$

e di seguito:

$$\forall i, Fitness[i] = \frac{PathLengthInverted[i]}{TotalPathLengthInverted}$$

Ecco, dunque, che a ciascun individuo corrisponde un numero reale compreso tra 0 e 1 (zero incluso), che esprime la sua bontá. Questa misura, che di fatto é una distribuzione di probabilitá, servirá agli operatori di selezione descritti nel seguito.

4.1.6 Il calcolo del Kick Off

L'uso del Kick Off consente di inibire la possibilitá che gli individui *peggiori* di una generazione sopravvivano nella generazione successiva. Quanti di questi individui devono perire é delegato all'utente, attraverso uno slider. Questo slider determina una percentuale sul valore massimo di fitness, e gli individui che hanno lunghezza di percorso inferiore ad essa vengono forzati ad avere $PathLengthInverted$ nulla, e quindi anche fitness nulla. Tutto ció viene implementato nel seguente modo:

```
for (int i=0; i<NrOfChromosome; i++) {
    if(PathLengthInverted[i]>Max){
        Max=PathLengthInverted[i];
    }
}

Threshold = Max*(jSliderKickOff.getValue()/100.0);
for (int i=0; i<NrOfChromosome; i++) {
    if (PathLengthInverted[i]<Threshold){
        PathLengthInverted[i]=0;
    }
}
```

```

    }
    TotalPathLengthInverted = TotalPathLengthInverted +
                             PathLengthInverted[i];
}

```

4.1.7 Tecniche di selezione

Nell'algoritmo che implementiamo, così come in quasi tutti gli algoritmi genetici, la selezione è il primo operatore che viene applicato. Il suo input è un'intera popolazione, ed il suo output è un'altra popolazione (intermedia, nel senso che non costituisce un *cambio generazionale*), sulla quale poi agirà il crossover. Rendiamo disponibile all'utente la possibilità di scegliere tra le seguenti tecniche di fitness:

Standard: l'idea generale di questa selezione è di scegliere a caso un cromosoma, su una distribuzione proporzionale alla fitness, cosicché gli individui migliori hanno maggiore probabilità di venire selezionati. Per realizzare ciò ci si è avvalsi di una tecnica piuttosto usata quale la *roulette*, che ora andiamo a spiegare:

si genera un vettore, di dimensioni pari al numero di cromosomi, in cui il primo elemento contiene la fitness del primo cromosoma, e un generico j -esimo elemento contiene la somma delle fitness dei suoi antecedenti (quelli con indice $i=1,2,\dots,j-1$) più la propria. Essendo la fitness normalizzata ad 1, ciascun elemento del vettore è sempre minore o uguale ad 1. Così facendo, l'individuo che ha la fitness più alta ha maggiore probabilità di venire estratto:

```

public int RouletteSelection(double[] Roulette){
    //genera un numero casuale tra 0 ed 1 (escluso)
    double Rnd=Math.random();
    boolean Found=false;
    int Selected=0;

    //va alla ricerca dell'individuo
    do{
        if(Roulette[Selected]<Rnd){
            Selected++;
        } else Found=true;
    } while(!Found);
    return Selected;
}

```

Best Only: questo tipo di selezione è il più drastico. In input riceve, al solito, una popolazione, ed in output produce una popolazione che contiene l'individuo migliore della popolazione di input, duplicato tante volte quanta è la cardinalità della popolazione fissata dall'utente. Questo tipo di selezione è

concepito per far sí che la popolazione prodotta in output non venga per nulla sottoposta a crossover, bensí a mutazione con probabilitá 1.

La *Best Only* porta in pratica il nostro algoritmo genetico ad esasperare il suo comportamento in quello di una ricerca locale randomizzata; per una trattazione completa di questi argomenti rimandiamo ad una delle sezioni successive.

Biased: il comportamento di questo tipo di selezione é in ultima analisi analogo a quello della *Standard*, con la differenza che l'elevamento a potenza di $PathLengthInverted[i]$ (determinato sempre dallo slider *Elitism*) avviene gradualmente, partendo cioé dall'esponente 1, fino ad arrivare all'esponente dato da *Elitism* alla fine della computazione:

$$\alpha = Elitism \cdot \frac{Generation}{TotalNrOfGenerations}$$

$$\forall i \ PathLengthInverted[i] = (MaxPathLength - PathLength[i])^{\alpha^{3.5}}$$

4.1.8 Tecniche di crossover

L'operatore di crossover accetta in input due cromosomi (detti *genitori*), e restituisce in output altri due cromosomi (detti *figli*) che hanno nel loro *patrimonio genetico* informazione utile sia del padre che della madre. Questo é infatti l'intento: far evolvere la "specie" in modo tale che una porzione di patrimonio genetico dei genitori venga sia "mescolata" che conservata in quello dei figli. Ciò che rende peculiare il crossover nel problema del Commesso Viaggiatore é il fatto che deve essere un operatore conservativo della proprietá dei cromosomi di essere una permutazione di cittá, ovvero di essere dei percorsi. Elenchiamo qui di seguito i tipi di crossover messi a disposizione dell'utente:

Single Point: questo é uno dei crossover piú semplici tra quelli resi disponibili, ma nonostante ciò fornisce dei discreti risultati. Illustriamo il suo funzionamento con un esempio.

```
genitore  5 2 1 4 6 3
figlio    - - - - -
```

Step 1: Si seleziona a caso un elemento del genitore e lo si posiziona in un posto a caso del figlio.

```
genitore  5 2 1 4 6 3
figlio    - - - - 1 -
```

Step 2: Si posizionano i restanti elementi nello stesso ordine (per quanto possibile) in cui comparivano nel padre.

```
genitore  5 2 1 4 6 3
figlio    5 2 4 6 1 3
```

City Centered: questo crossover é l'unico ad essere imperniato non tanto sulla scelta di un indice casuale, quanto di una città. In questo modo i figli *ereditano* dai genitori la stessa sequenza di città che precedono quella selezionata.

```

padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     - - - - -      figlia - - - - -

```

Step 1: si sceglie a caso una città, e la si cerca all'interno dei cromosomi padre e madre. Fatto questo, si ricopiano le città fino a quella selezionata (il padre nel figlio, la madre nella figlia).

```

padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     5 2 - - - -      figlia 1 3 2 - - -

```

Step 2: completiamo il cromosoma figlio con gli indici che gli mancano per completare la permutazione con quelli della madre mantenendo il più possibile l'ordine che avevano nella madre, e procediamo analogamente per completare il cromosoma figlia.

```

padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     5 2 1 3 4 6      figlia 1 3 2 5 4 6

```

Order: lo scopo di questo crossover é di conservare nei figli un certo *blocco* di informazione potenzialmente utile. Mostriamo il procedimento con un esempio:

```

padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     - - - - -      figlia - - - - -

```

Step 1: si scelgono a caso due indici. Ricopio gli elementi compresi tra i due indici nei figli.

```

padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     - - 1 4 6 -      figlia - - 2 4 6 -

```

Step 2: completiamo il cromosoma figlio con gli indici che gli mancano per completare la permutazione con quelli della madre mantenendo il più possibile l'ordine che avevano nella madre, e procediamo analogamente per il cromosoma figlia.

```

padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     3 2 1 4 6 5      figlia 5 1 2 4 6 3

```

Order Based: anche questo tipo di crossover cerca di mantenere in un certo modo sottosequenze dei genitori che con una certa probabilità contengono buona informazione. Mostriamo il suo funzionamento attraverso un esempio:

```

padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     - - - - -

```

Step 1: viene scelto a caso un certo numero di indici nella madre, ed i vertici del padre che corrispondono a quelli puntati dagli indici scelti vengono marcati.

```
padre      X 2 1 X 6 X      madre  1 3 2 4 6 5
figlio     - - - - -
```

Step 2: copiamo nel figlio gli elementi che nel padre non sono stati cancellati.

```
padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     _ 2 1 _ 6 _
```

Step 3: copiamo nel figlio nelle posizioni che sono state cancellate (quelle che contenevano elementi selezionati nella madre) gli elementi della madre nello stesso ordine in cui compaiono nella madre stessa

```
padre      5 2 1 4 6 3      madre  1 3 2 4 6 5
figlio     3 2 1 4 6 5
```

Analogamente si procede con il cromosoma figlia, scegliendo però gli elementi nel padre invece che nella madre.

4.1.9 Tecniche di mutazione

L'operatore di mutazione accetta in input un cromosoma (quindi un vettore che é una permutazione di città), e restituisce in output un cromosoma le cui componenti sono variate nei modi che elencheremo qui di seguito, pur sempre mantenendo la sua proprietà fondamentale di essere permutazione.

Simple Swap: questo tipo di mutazione semplicemente sceglie due indici a caso del cromosoma di input e scambia gli elementi da loro puntati. Eccone un esempio, in cui gli indici scelti sono 1 e 4:

```
padre      5 2 1 4 6 3
figlio     3 6 1 4 2 5
```

Simple Move: questa mutazione sceglie a caso due indici del cromosoma di input. Il primo indice indica il gene da spostare, il secondo indica la posizione in cui esso va posizionato. Gli elementi compresi tra i due indici semplicemente scalano, nel modo illustrato dal seguente esempio:

```
padre      5 2 1 4 6 3
figlio     - - - - -
```

Step 1: scelta dei due indici, e.g. 2 e 4. Se l'elemento di indice 2 (che é 1) va in posizione 4 (quella del 6).

```
padre      5 2 1 4 6 3
figlio     5 2 4 6 1 3
```

Scramble: ancora una volta vengono scelti due indici a caso, ed il settore compreso tra questi due indici viene permutato casualmente. Forniamo un esempio, in cui gli indici scelti sono 0 e 3:

```
padre      5 2 1 4 6 3
figlio     2 5 4 1 6 3
```

Sub-Tour Inversion: dati due indici, gli elementi compresi tra di essi vengono invertiti. Scelti, ad esempio, gli indici 1 e 5, abbiamo il seguente risultato:

```
padre      5 2 1 4 6 3
figlio     5 3 6 4 1 2
```

Swap Blocks: si scelgono sempre due indici casuali sul cromosoma di input. Questi indici determinano la sua suddivisione in 3 settori. Il primo ed il terzo vengono invertiti, nel seguente modo:

```
padre      A | B | C
figlio     C | B | A
```

4.1.10 Generazione di permutazioni random

Qui di seguito andiamo a descrivere la tecnica usata per la generazione random di permutazioni. L'unico vincolo di questa tecnica é la necessità di partire da una permutazione iniziale, quale può essere per esempio un semplice vettore ordinato. Di fatto, i nostri cromosomi, alla loro creazione sono già delle permutazioni costruite a partire da un vettore ordinato in modo crescente. L'algoritmo procede così:

Step 1: si parte da un vettore ordinato strettamente crescente.

```
vettore ordinato    1 2 3 4 5 6
permutazione        - - - - -
```

Step 2: viene scelto a caso un indice, ad esempio 3. Questo indice punta ad un valore (nel nostro caso 4) che viene ricopiato nella prima posizione libera della futura permutazione, e tolto dal vettore ordinato. I valori che lo seguono scalano di una posizione, riducendo così lo spazio per la prossima generazione casuale di un indice.

```
vettore ordinato    1 2 3 5 6 X
permutazione        4 - - - -
```

Step 3: ancora una volta si sceglie a caso un indice, ma tra 1 e $n-1$, se n é la dimensione del vettore. Nel nostro caso, tra 0 e 4, viene scelto 4. L'elemento puntato da questo indice va nella prima posizione libera della futura permutazione, e ancora una volta gli elementi che seguono quello scelto, nel vettore ordinato, scalano di una posizione.

vettore ordinato	1 2 3 5 X X
permutazione	4 6 - - - -

La procedura itera n volte, generando alla fine una permutazione. Essa é stata impiegata, oltre che nella generazione casuale di percorsi, anche nella mutazione denominata *Scramble*, per generare nel cromosoma di output un settore variato casualmente.

4.1.11 L'Algoritmo di Prim per il Minimum Spanning Tree

Come già detto, l'albero di copertura minimo é un lower bound largamente usato in letteratura per il problema del Commesso Viaggiatore. Si é scelto di implementarlo seguendo l'algoritmo di Prim, senza però costruire l'albero, ma tenendo conto solo della sua lunghezza, che é il valore che ci interessa per poter dare una valutazione di bontá del nostro algoritmo genetico. L'algoritmo di Prim procede nel seguente modo:

- Inizialmente la lunghezza dell'albero e' 0, e viene predisposta una matrice nxn che contiene le distanze da ogni nodo a tutti gli altri.
- Si parte dal primo vertice del grafo $G(V,E)$, $v[0]$
- $v[0]$ sempre nell'insieme IN dei nodi che stanno nell'albero di copertura minimo
- Gli altri vertici stanno in OUT: $OUT = G-v[0]$
- si calcolano tutte le distanze da $v[0]$ ai suoi nodi vicini. Il nodo piu' vicino, $v[k]$, viene inserito nell'insieme IN: $IN = \{v[0], v[k]\}$
- la lunghezza dell'albero viene incrementata con la lunghezza dell'arco $(v[0],v[k])$
- .
- .
- .
- Si calcolano tutte le distanze da ogni nodo di IN a tutti i nodi di $G-IN$.
- Il nodo piu' vicino viene aggiunto a IN e la lunghezza dell'albero viene incrementata con la lunghezza dell'arco che lo collega.
- Al termine della computazione la lunghezza ottenuta e' quella dell'albero di copertura minimo.

4.1.12 Il thread dell'algoritmo

Durante l'esecuzione, é sempre possibile fermare momentaneamente (con il bottone *Pause*) o definitivamente (con il bottone *Stop*) l'algoritmo. Per rendere disponibili queste funzionalitá si é reso necessario disaccoppiare l'algoritmo

dall'interfaccia, che altrimenti sarebbe stata inibita dalla mole di calcolo. Ciò é stato implementato con un *thread* (o anche *light weight process*) che si genera ad ogni pressione del bottone *Run*. La pressione del bottone *Pause* una prima volta porta l'algoritmo in uno stato di attesa (istruzione *wait*), mentre una successiva pressione ne provoca il *risveglio* (istruzione *notify*). La pressione del bottone *Stop* semplicemente fa uscire il programma dal ciclo di calcolo.

4.2 L'Interfaccia Utente

Si é cercato di rendere l'interfaccia utente il piú semplice possibile, compatibilmente con l'intenzione di porre in un'unica finestra la possibilitá di manipolare tutti i parametri che regolano l'algoritmo genetico, ed anche i risultati della computazione. Descriviamo brevemente qui di seguito le varie componenti dell'interfaccia e le loro funzionalitá, attraverso delle "*domande guida*".

- **Come esco dal programma?**
Esco dal programma andando su *File* a sinistra della barra comandi, e scegliendo *Exit*.
- **Posso ripristinare le impostazioni dei parametri genetici iniziali?**
Sì, andando su *Options* sulla barra comandi, e scegliendo *Restore Genetic Parameters*.
- **Posso ripristinare le impostazioni iniziali del programma?**
Sì, andando su *Options* sulla barra comandi, e scegliendo *Restore All*.
- **C'è uno help in linea?**
Sì, basta andare su *Help* sulla barra comandi, e scegliere *Tutorial*.
- **Come posso avere informazioni sul programma e sugli autori?**
Sì, basta andare su *Help* sulla barra comandi, e scegliere *About*.
- **Come faccio partire il programma?**
Basta premere il bottone *Run*, che sta tra i bottoni subito sotto il grafo.
- **Come faccio far fermare il programma?**
Basta premere il bottone *Pause* se lo si vuole arrestare momentaneamente, oppure *Stop* se lo si vuole fermare definitivamente. In questo caso un'ulteriore finestra di dialogo ne chiederá conferma.
- **Come faccio far ripartire il programma dopo aver premuto *Pause*?**
Basta ripremere lo stesso bottone, che però ora é etichettato *Restart*.
- **É possibile cambiare il tipo di grafo su cui lavorare?**
É sufficiente premere il bottone *New*, che si trova tra i bottoni sotto il grafo, e si aprirá un'altra finestra, attraverso la quale sará possibile cambiare topologia di grafo scegliendo tra le 60 città piú importanti degli Stati

Uniti, una topologia in cui i vertici vengono posizionati agli incroci di una griglia, una in cui i vertici sono posizionati in maniera randomica sullo schermo (con una distanza minima imposta, per ragioni di visibilità), ed infine una in cui le città sono poste su di una circonferenza (anche in questo caso, il numero di città non é eccessivamente alto, per ragioni di visualizzazione).

- **Cosa rappresentano i grafici in alto a destra della finestra?**

La linea retta di colore verde indica il valore del Lower Bound; la spezzata di colore rosso indica la varianza della popolazione corrente. La spezzata di colore azzurro rappresenta il valore del percorso migliore raggiunto dall'inizio dell'algoritmo, mentre quella di colore giallo rappresenta il valore medio della lunghezza del percorso, come già mostrato.

- **Come posso modificare le probabilità di mutazione e crossover?**

É sufficiente muovere i due slider posizionati in basso a destra della finestra. Essendo valori di probabilità, l'intervallo in cui si possono muovere é $[0, 1] \subset \mathbb{R}$.

- **Posso aumentare il numero di cromosomi della popolazione?**

Sì, vi é per l'appunto uno slider denominato *Nr. of Chromosomes*, che permette di portare il numero di cromosomi fino a 999. Ovviamente, piú alto é il numero di cromosomi, piú viene rallentato l'algoritmo, visto che gli operatori di fitness, crossover e mutazione devono comunque scorrere l'intero vettore della popolazione per calcolare la generazione successiva. Da notare, infine, che lo slider permette di scegliere soltanto valori pari: la popolazione infatti deve necessariamente essere costituita da un numero pari di cromosomi, poiché il crossover avviene solo tra coppie contigue di individui. Un numero dispari di essi lascerebbe l'ultimo spaaiato, impedendo il crossover.

- **Posso incrementare il numero di generazioni?**

Sì, attraverso lo slider denominato *Nr. of Generations*. Aumentando il numero di generazioni viene incrementata la probabilità di ottenere buone soluzioni al problema, poiché selezione, crossover e mutazione vengono applicati piú volte.

- **Cosa indica la barra di scorrimento?**

Dalla barra di scorrimento si possono ottenere indicazioni su quanto manca al termine e su a che punto sia arrivata la computazione. Essa progredisce fino a ché il numero di generazioni da calcolare (impostato dall'utente) é stato raggiunto.

- **Perché gli slider, le combo box e alcuni bottoni diventano grigi durante l'esecuzione?**

Questo accade per evitare che accidentalmente o volontariamente l'utente

cambi i parametri genetici che l'algoritmo deve seguire. Inibendo slider e combo box i parametri restano fissati dall'inizio al termine della computazione.

4.2.1 Il riscaldamento automatico dell'informazione Grafica

Per sfruttare al meglio tutta la finestra, il grafo delle città é stato concepito in modo tale da riscalarsi, per far sí che occupi sempre tutto lo spazio a sua disposizione, per una maggiore visibilitá dell'utente, ma anche per il fatto che la finestra del programma puó venire ingrandita o ridimensionata ad ogni momento. Inoltre, i grafici sono interamente ricalcolati ad ogni generazione, in modo da fornire all'utente tutta l'informazione disponibile nel maggiore spazio possibile.

4.3 La Ricerca Locale Randomizzata

Portando agli estremi alcuni parametri dell'algoritmo genetico é possibile simulare una ricerca locale randomizzata. É infatti possibile, optando per la selezione *Best Only*, impostare automaticamente la probabilitá di crossover a zero, e quella di mutazione a uno. La selezione effettuata non procede piú scegliendo casualmente degli individui basandosi sulla loro fitness, bensí al termine di ogni generazione, viene individuato il cromosoma migliore (con la lunghezza di percorso piú breve) e viene duplicato tante volte quanta é la cardinalitá della popolazione stessa. Cosí facendo, all'inizio di ogni epoca, la popolazione corrente risulta costituita da tante copie dello stesso individuo. A questo punto la mutazione, avendo probabilitá 1 di essere applicata, muta tutti gli individui in base alla tecnica di mutazione selezionata. Ricordando che la tecnica di ricerca locale si basa su una esplorazione esaustiva dell'intorno dell'individuo corrente, ed osservando che ogni tecnica di mutazione implicitamente definisce un intorno, possiamo concludere che vi sia forte analogia tra l'approccio genetico e la ricerca locale. La differenza consiste nel fatto che la nostra tecnica non prevede piú un'esplorazione esaustiva, limitata all'interno sí dell'intorno definito dalla mutazione, ma in cardinalitá limitato dal numero di individui presenti nella popolazione.

Da questo tipo di osservazione segue, in conformitá alle osservazioni fatte sperimentalmente, che la tecnica di selezione *Best Only* produce buoni risultati, paragonabili a quelli della ricerca locale, qualora la cardinalitá della popolazione sia impostata a valori elevati. Tutto ció in contrapposizione all'algoritmo genetico applicato nella sua interezza, che riesce a produrre risultati accettabili anche con popolazioni di cardinalitá ridotta.

Abbiamo detto che ogni mutazione definisce implicitamente un intorno. Cambiando mutazione cambierá l'intorno, sia in termini di cardinalitá (numero dei vicini), che in termini di raggio, nel senso che a paritá di individui nell'intorno, determinate mutazioni implicano una diversificazione molto maggiore nei vicini.

Ad esempio, la *Sub Tour Inversion* ha un numero di vicini paragonabile alla *Simple Swap*, essendo entrambe caratterizzate dalla scelta di due città all'interno del percorso. Ma, mentre la prima produce una completa inversione del percorso tra queste due città, la seconda inverte solo gli estremi dell'intervallo. Sperimentalmente si è potuto verificare che un'alta cardinalità della popolazione, un'alta cardinalità dei vicini e una forte diversificazione tra i vicini determinano risultati migliori.

5 Conclusioni

Con questa relazione speriamo di aver dato un'idea di quanto diversificato possa essere il modo di costruire un algoritmo genetico, in particolare per il problema del Commesso Viaggiatore. Tante sono le strutture dati possibili da implementare, tanti i diversi operatori, ciascuno dei quali con i suoi vantaggi e svantaggi, dal punto di vista dell'analisi e dal punto di vista del risultato. La ricerca di un programma evolutivo per il TSP che includa la miglior rappresentazione e operatori genetici adatti è tuttora in corso.

Oltre che un algoritmo genetico puro, abbiamo reso possibile la ricerca locale randomizzata, che si è dimostrata essere piuttosto efficiente se non, in diversi casi, migliore. Da quanto si evince consultando la letteratura sull'argomento, in ogni caso, un buon programma evolutivo per il TSP dovrebbe incorporare sia operatori per il miglioramento locale, sia operatori per l'interscambio di informazione utile (come il crossover).

6 Miglioramenti futuri

In considerazione dei risultati ottenuti e delle informazioni acquisite durante lo svolgimento del progetto, riteniamo che siano possibili dei miglioramenti all'algoritmo, in due direzioni. Una direzione è quella del raffinamento di quanto già implementato, con ulteriori operatori e con diverse rappresentazioni a livello implementativo; una seconda direzione è quella dell'introduzione di una maggiore flessibilità del controllo dei parametri di input da parte dell'utente. Ad esempio potrebbe risultare più efficace e più utile di poter combinare assieme diverse tecniche di mutazione e crossover, con una data probabilità. Questo introdurrebbe una maggiore variabilità alla popolazione durante l'esecuzione dell'algoritmo, consentendo di esplorare un intorno maggiore specialmente quando si è vicini ad un ottimo locale.