Progetto conclusivo Algoritmi e Strutture Dati

Giacomo Astolfi, Alessandro Cingolani, Orazio Colaneri,
Cristian Federiconi, Luca Luzi, Federica Massacci

A.A. 2016/2017

Indice

Capitolo 1

Introduzione

1.1 Generalita'

Un algoritmo genetico e' un algoritmo ispirato al principio della selezione naturale teorizzato da Charles Darwin. L'aggettivo "genetico" deriva dal fatto che gli algoritmi genetici attuano dei meccanismi concettualmente simili a quelli dei processi biochimici che a loro volta seguono il modello di Darwin.
Gli algoritmi genetici sono dunque degli algoritmi che consentono di valutare la bonta' delle soluzioni di partenza ad un dato problema e che poi ricombinando tra di loro le varie soluzione ed introducendo delle mutazioni in esse sono in grado di crearne delle migliori, ripetendo questo procedimento piu' volte si e' in grado di convergere a delle soluzioni ottime.
Questa tecniche vengono di solito utilizzate per cercare di trovare delle soluzioni a problemi di ottimizzazione per i quali non si conoscono soluzioni con complessita' lineare o polinomiale; tuttavia data la natura empirica dell'algoritmo non e' detto che sia in grado di trovare delle soluzioni accettabili al problema.

1.2 Funzionamento

Prima di spiegare il funzionamento degli algoritmi genetici e' bene introdurre la terminologia, ereditata dalla biologia, che li correda , in modo da poterne comprendere meglio il funzionamento che sara' successivamente illustrato.

- Cromosoma: una delle soluzioni ad un problema considerato. Generalmente e' codificata con un vettore di bit o di caratteri.

- Popolazione: insieme di cromosomi, dunque insieme delle soluzioni relative al problema considerato.

- Gene: parte di un cromosoma. Generalmente consiste in una o piu' parti del vettore di bit o caratteri che codificano il cromosoma.

- Fitness: grado di valutazione associato ad una soluzione. La valutazione avviene in base ad una funzione appositamente progettata detta funzione di fitness.

- Crossover: generazione di una nuova soluzione mescolando delle soluzioni esistenti.
- Mutazione: alterazione casuale di una soluzione.

Un tipico algoritmo genetico, nel corso della sua esecuzione, provvede a fare evolvere delle soluzioni secondo il seguente schema di base:

1. Generazione casuale della prima popolazione di soluzioni (cromosomi).
2. Applicazione della funzione di fitness alle soluzioni (cromosomi) appartenenti all'attuale popolazione.
3. Selezione delle soluzioni considerate migliori in base al risultato della funzione di fitness e della logica di selezione scelta.
4. Procedimento di crossover per generare delle soluzioni ibride a partire dalle soluzioni precedentemente selezionate.
5. Creazione di una nuova popolazione a partire dalle soluzioni correnti.
6. Iterazione della procedura a partire dall' applicazione della funzione di fitness ed utilizzando la popolazione appena creata.

Ad ogni iterazione le soluzioni trovate migliorano di volta in volta fino ad arrivare ad una soluzione ottimizzata per il problema preso in esame.

Il crossing over e' il meccanismo che sta alla base della ricombinazione genetica, poiche' i vari geni si mescolano tra loro in maniera casuale e' possibile che durante la riproduzione alcune delle soluzioni migliori della popolazione precedente vengano perse nella nuova.
Potrebbe inoltre accadere che l'algoritmo si soffermi su soluzioni che fanno parte di "ottimi locali" ma che non siano le migliori in assoluto.
Per evitare problemi di questo genere vengono integrate con l'algoritmo le tecniche dell' "elitarismo" e delle mutazioni casuali.
La tecnica dell' elitarismo consiste nel copiare nella nuova popolazione i migliori individui della precedente , le seconda tecnica consiste invece nell' introdurre , con un frequenza opportuna, delle mutazioni casuali nella soluzioni trovate in modo da permettere all' algoritmo di non ricadere in ottimi locali: con le mutazioni e' possibile preservare la varieta' della popolazione , tuttavia non bisogna ne' eccedere in modo da alterare significativamente le soluzioni , ne' introdurre mutazioni troppo sporadiche che quindi non garantirebbero l'eterogeneita' degli individui.

Capitolo 2

Il Progetto

2.1 Il problema del commesso viaggiatore

Il problema del commesso viaggiatore consiste , data una rete di citta' connesse tra loro tramite strade, di trovare il percorso che minimizza il tragitto e consenta di visitare tutte le citta' una ed una sola volta.

Codifica: Il metodo scelto per codificare il problema consiste nell' impiego di una struttura dati per rappresentare la posizione delle citta' sulla mappa e di un vettore di tali strutture su cui salvare le citta' generate casualmente.
Si definisce una struttura di tipo individuo che contiene un vettore di interi chiamato soluzione che codifica appunto l'ordine di visita delle citta' e altri due campi relativi alla lunghezza del percorso associato al vettore soluzione e il fitness: il grado di grado di bonta' della soluzione.
Si definisce in ultimo un array di individui che prende il nome di popolazione che contiene tutte le inizialmente generate a caso.

Fitness: La funzione di fitness valuta la bonta' delle singole soluzioni, piu' la distanza percorsa e' piccola piu' la soluzione trovata e' migliore, il fitness e' dunque inversamente proporzionale alla lunghezza dell' itinerario associato ad ogni soluzione.

Selezione: Vengono estratte le soluzioni in misura proporzionale alla loro distribuzione discreta di probabilita' associata ai rispettivi fitness. In tal modo , nella nuova popolazione, saranno presenti in maniera piu' preponderante le soluzioni migliori.

Cross Over: Nel cross over vengono ricombinate a due a due le soluzioni precedentemente estratte per crearne due nuove che poi formeranno la nuova popolazione . Tale procedimento viene poi ripetuto per tutte le coppie di soluzioni candidate.

Mutazione: Con una probabilita' adeguata, si scegli di applicare una variazione casuale del percorso che consiste nell' invertire l'ordine con cui si presentano

due citta' nella soluzione.

2.2 Come abbiamo risolto il problema

Si definiscono inizialmente due strutture dati: "city" contenente le coordinate delle citta', e la struttura dati "individual" a cui viene associato un percorso, che nel programma prende il nome di "solution", che consiste in una serie di indici, corrispondenti alle citta' da visitare, formando così un itinerario.
Ogni soluzione deve essere valutata in base alla lunghezza del percorso che le e' associato, ovvero il grado di bonta' della soluzione. Questo viene salvato nella variabile "fitness" che di fatto e' il reciproco della lunghezza. In tal modo il percorso piu' breve avra' un fitness piu' alto.
Per cominciare si inizializza il vettore "cities_map" che contiene le coordinate delle citta', disponendole a caso su una griglia. Sara' poi possibile accedere ad una citta' visitata durante il percorso grazie alla posizione che questa occupa all'interno del vettore "cities_map".

Esempio: nel percorso [2−3−1−5−0] la prima citta' ha coordinate $cities\_map[2].x$ e $cities\_map[2].y$.

La popolazione e' inizializzata, sempre casualmente, all' interno della funzione "generate_initial_population()" la quale associa ad ogni elemento della popolazione un vettore in cui compaiono, senza ripetizioni, gli indici delle citta'.
Per generare un vettore con tale permutazione si e' optato per la seguente soluzione:

```
[pseudo]

    - Per ogni j da 0 al numero di citta' (CITY_COUNT), con passo 1:
    -- Si generi un numero X a caso compreso tra 0 e j inclusi
    -- Se x e' diverso da j:
    --- l'elemento j-esimo della soluzione che stiamo considerando
       diventa l'elemento x-esimo della soluzione che stiamo considerando
    -- l'elemento x-esimo della soluzione che stiamo considerando
     diventa uguale a j (non al j-esimo, ma proprio a j!)

[/pseudo]

[codice]

void generate_initial_population(individual * population){

    int x;

    for (int i = 0; i < POPULATION_COUNT; i++){
        for (int j = 0; j < CITY_COUNT; j++){
            x = rand() % (j+1);     //Numero casuale in [0, j]
            if (x != j){
                population[i].solution[j] = population[i].solution[x];
            }
```

```
            population[i].solution[x] = j;
        }
    }

    debug("Generazione della popolazione iniziale completata");
}
```

[/codice]

In questo modo l'array che viene generato e' semplicemente una permutazione dell'array ordinato che contiene i numeri interi da 0 a CITY_COUNT-1.
Ad esempio, se CITY_COUNT fosse 5, l'algoritmo eseguirebbe (se uscissero i valori di x che useremo nell'esempio) questi passaggi:

[esempio]

```
(gli underscore indicano parti non inizializzate)
SOLUZIONE: _ _ _ _ _ ; j = 0; x = _
x può essere solamente 0, in quanto l'unico numero
tra 0 (compreso) e 1 (non compreso) e' proprio 0. Siccome x e j sono uguali,
facciamo solo l'assegnamento post-if
"l'elemento alla posizione 0 diventa 0"
SOLUZIONE: 0 _ _ _ _ ; j = 0; x = 0
Ora j e' 1, X può essere estratto come 1 o 0, supponiamo esca 1
x e j sono ancora uguali, solo l'assegnamento post if
"l'elemento alla posizione 1 diventa 1"
SOLUZIONE: 0 1 _ _ _ ; j = 1; x = 1
Ora j e' 2, X può essere 0, 1 e 2. Supponiamo esca 0.
Ora facciamo sia l'assegnamento nell'if che quello all'esterno
"l'elemento alla posizione 2 diventa l'elemento alla posizione 0"
"l'elemento alla posizione 0 diventa 2"
SOLUZIONE: 2 1 0 _ _ ; j = 2; x = 0
Ora j e' 3, x  va da 0 a 3, supponiamo esca 2, sono diversi
"l'elemento alla posizione 3 diventa l'elemento alla posizione 2"
"l'elemento alla posizione 2 diventa 3"
SOLUZIONE: 2 1 3 0 _ ; j = 3; x = 2
Alla fine, con j=4, x può andare da 0 a 4, mettiamo esca 1
"l'elemento alla posizione 4 diventa l'elemento alla posizione 1"
"l'elemento alla posizione 1 diventa 4"
SOLUZIONE: 2 4 3 0 1 ; j = 4; x = 1
Fine del ciclo.
Possiamo notare che questo array ottenuto alla fine e'
effettivamente una permutazione di "0 1 2 3 4"
```

[/esempio]

Ripetuta questa operazione per ogni elemento della popolazione se ne calcola il fitness nella funzione "evaluate_fitness()" che calcola la lunghezza associata al percorso e a questa fa corrispondere un fitness che ne e' il reciproco; tuttavia se la lunghezza del percorso fosse troppo grande al fitness corrisponderebbe un valore troppo basso che potrebbe causare instabilita' numerica, si e' pertanto

scelto di associare al fitness 100 sulla distanza.
Data la casualita' con cui vengono inizializzate le citta' e con cui vengono formate le soluzioni, i fitness associati ai vari elementi della popolazione possono assumere qualsiasi valore, non e' pertanto possibile stabilire a priori la bonta' di una soluzione senza confrontarla con le altre.
Dobbiamo quindi ripetere il procedimento, generando una nuova popolazione basandoci sulle informazioni ottenute (i fitness).
Poiche' non vogliamo estrarre e riprodurre solo gli elementi migliori, per evitare che l'algoritmo si stabilizzi su una soluzione che non e' la migliore, e' necessario riprodurre sia le soluzioni piu' ottimali che quelle meno. Un modo di risolvere il problema e' quello di assegnare ad ogni soluzione una probabilita' di riproduzione: tanto piu' il fitness e' alto tanto piu' e' alta la probabilita' di estrazione, in questo modo si riesce a mantenere variegata la popolazione, così da evitare che l'algoritmo si blocchi su dei massimi relativi.
Per estrarre le soluzioni con una probabilita' pari al fitness e' necessario rendere il fitness una distribuzione discreta di probabilita', basta dunque dividere ogni fitness per la somma degli altri e salvare la somma progressiva di tali valori in un vettore chiamato "ripartizione".

[codice]

```
double * evaluate_fitness(city * cities_map, individual * population){

    int id_1, id_2;
    double total, fitness_sum = 0;

    //Allochiamo spazio per la distribuzione di probabilita'
    double * ripartizione = malloc(sizeof(double) * POPULATION_COUNT);

    for (int i = 0; i < POPULATION_COUNT; i++){

        total = 0;  //Contiene la distanza totale tra le citta'

        for (int j = 0; j < CITY_COUNT-1; j++){
            id_1 = population[i].solution[j];
            id_2 = population[i].solution[j+1];

             //Calcola la distanza tra due citta' e la somma alla totale
            total += get_distance(cities_map, id_1, id_2);
        }

   //A questo punto il fitness e' 1/distanza totale
        population[i].fitness = 1/total;
        //La somma dei fitness viene utilizzata per la "normalizzazione"
        fitness_sum += population[i].fitness;
    }

    ripartizione[0] = population[0].fitness/fitness_sum;

    for (int i = 1; i < POPULATION_COUNT; i++){
```

```
     /*Assegna al valore ripartizione il valore dell'elemento
     precedente + il fitness corrente normalizzato (diviso per la loro somma)*/
        ripartizione[i] = ripartizione[i-1] + population[i].fitness/fitness_sum;
    }

    debug("Fitness calcolati");

    return ripartizione;
}

[/codice]


[esempio]

Fitness di 5 citta' : 0.1 0.3 0.8 0.7 0.1
Somma = 0.1 + 0.3 + 0.8 + 0.7 + 0.1 = 2
Ripartizione : _ _ _ _ _

Nuovo elemento : 0.1/2 = 0.05
Ripartizione : 0.05 _ _ _ _

Nuovo elemento : 0.05 + 0.3/2 = 0.2
Ripartizione : 0.1 0.2 _ _ _

Nuovo elemento : 0.2 + 0.8/2 = 0.6
Ripartizione : 0.1 0.2 0.6 _ _

Nuovo elemento : 0.6 + 0.7/2 = 0.95
Ripartizione : 0.1 0.2 0.6 0.95 _

Nuovo elemento : 0.95 + 0.1/2 = 1
Ripartizione : 0.1 0.2 0.6 0.95 1

[/esempio]
```

Per estrarre un elemento dalla popolazione e' sufficiente estrarre un numero casuale, nominato "extraction", compreso tra 0 e 1 e cercare il primo numero maggiore di "extraction" nel vettore "ripartizione". Poiche' il vettore e' ordinato per costruzione, e' possibile applicare la ricerca binaria riducendo così la complessita' computazionale da $O(n)$ a $O(log(n))$, tenendo però conto che quello che si cerca non e' un valore uguale ad un numero dato bensì il primo numero maggiore.
Difatti, riprendendo l'esempio sopracitato, [0.10.20.60.951], consideriamo l'intervallo $[0.2, 0.6]$, lungo 0.4 e l'intervallo $[0.95, 1]$, lungo 0.05. Un numero estratto tra 0 e 1 (appunto "extraction") ha una probabilita' 8 volte superiore (0.4/0.05) di ricadere nel primo intervallo, scegliendo quindi la terza citta', piuttosto che nel secondo intervallo, scegliendo la quinta.

```
[esempio]
```

```
Ripartizione : 0.1 0.2 0.6 0.95 1
Supponiamo che venga estratto 0.7

First : 1
Last : 5
Pivot : (1+5)/2 = 3

Poiche' 0.6 e' minore di 0.7
First : 3+1 = 4

First : 4
Last : 5
Pivot : (4+5)/2 = 4

Poiche' 0.95 e' maggiore di 0.7
Last : 4

First : 4
Last : 4

Uscita dal ciclo while, e' stata estratta la citta' 4

[/esempio]

[codice]

int * selection(double * ripartizione){

    double extraction;
    int pivot, first, last;
    int * candidati = malloc(sizeof(int) * POPULATION_COUNT);

    for (int i = 0; i < POPULATION_COUNT; i++){

        first = 0;
        last = POPULATION_COUNT;

        /*rand() genera un numero in [0, RAND_MAX]
        quindi dividendo per RAND_MAX otteniamo un numero in [0, 1]*/

        extraction = (double)rand()/RAND_MAX;

  /*Eseguiamo una ricerca binaria del primo
  elemento del vettore ripartizione con probabilita'
  maggiore di quella estratta*/

        while (first != last){
            pivot = (first + last)/2;
            if (ripartizione[pivot] == extraction){
```

```
                break;
            }
            if (ripartizione[pivot] > extraction){
                last = pivot;
            } else {
                first = pivot + 1;
            }
        }

        /*candidati[i] contiene l'indice dell'individuo estratto,
        maggiore e' la sua probabilita' in ripartizione, maggiore e'
        la probabilita' di essere in candidati, anche ripetutamente*/

        candidati[i] = first;
    }

    free(ripartizione); //Non serve piu' a questo punto

    debug("Candidati estratti");

    return candidati;
}
```

[/codice]

Iterando l'estrazione e la ricerca tante volte quanti sono gli individui della popolazione si salvano in un vettore di interi "candidati" gli indici delle citta' estratte.
Si può adesso procedere a ricombinare tra di loro gli individui. Con il cross over si intende generare una popolazione diversa dalla precedente ma che al contempo ne erediti in parte delle caratteristiche, nel caso in esame un cross over valido non deve solo preservare parti di soluzione passando da una generazione ad un'altra ma deve anche generare soluzioni in cui non ci siano citta' che si ripetono. Il cross over scelto per affrontare questo problema e' il "cycle cross over" che si adatta ai vincoli posti dal problema in esame.
Il cycle crossover genera, a partire da due genitori, cioe' due soluzioni della popolazione corrente, due figli, soluzioni della popolazione che verra' costruita pian piano. Gli elementi precedentemente selezionati vengono ricombinati a due a due; e' interessante osservare che due soluzioni con un fitness alto, e dunque buone, avendo un'alta probabilita' di estrazione, potrebbero essere selezionate due o piu' volte consecutivamente, con questo tipo di cross over da due genitori uguali vengono generati due elementi uguali a quelli di partenza, rispecchiando dunque l'idea di partenza di tramandare senza alterazioni di generazione in generazione le soluzioni migliori.
Una volta ricombinati tra di loro tutti gli elementi e messi nella nuova popolazione, restituiamo il puntatore a questa, "new gen", e cancelliamo la memoria occupata dalla popolazione precedente.
Di seguito e' riportata una spiegazione del cycle crossover.

[pseudo]

```
    -Per ogni i da 1 a CITY_COUNT & fino a che non ho visitato tutte le colonne,
     con passo 1:
    --Se sono gia' passato in questa colonna:
    ---Passa all'iterazione successiva nel ciclo
    --Con j=i, fai:
    ---incrementare di 1 le colonne visitate
    ---Se il ciclo e' pari:
    ----Copiare l'elemento del genitore 1 di indice i nel figlio 1 all'indice i
    ----Copiare l'elemento del genitore 2 di indice i nel figlio 2 all'indice i
    ---Se il ciclo e' dispari:
    ----Copiare l'elemento del genitore 2 di indice i nel figlio 1 all'indice i
    ----Copiare l'elemento del genitore 1 di indice i nel figlio 2 all'indice i
    ---Se l'elemento di indice j nel genitore 2 e' diverso dall'elemento di
        indice i nel genitore 1:
    ----Porre j uguale all'indice dell'elemento nel genitore 1 che ha lo
        stesso valore dell'elemento di indice j nel genitore 2
    --scambiamo i genitori per rispettare l'alternanza dei cicli
    --Finche' non ho visitato tutte le colonne dei genitori

[/pseudo]

[codice]

individual * crossover(individual * population, int * candidati){

    individual * new_gen = malloc(sizeof(individual) * POPULATION_COUNT);

    for (int i = 0; i < POPULATION_COUNT; i+=2){

     /*Eseguiamo il crossover a due a due scegliendo come
     genitori due candidati scelti tramite l'indice, salviamo i figli in new_gen*/

        cycle_crossover(population, candidati[i], candidati[i+1],
        new_gen[i].solution, new_gen[i+1].solution);
    }

    free(candidati);    //Non servono piu'

    debug("Riproduzione effettuata");

    return new_gen;
}

---------------

void cycle_crossover(individual * population, int id_1, int id_2, int * child_1,
int * child_2){

    //Copiamo gli indici dei genitori
    int parent_1 = id_1, parent_2 = id_2;
```

```
    //Colonna corrente
    int j = 0;
    //Vettore di indirizzamento
    int lookup[CITY_COUNT];
    //Vettore delle colonne visitate
    int flags[CITY_COUNT] = {0};
     //Numero di colonne visitate
    int visited = 0;
    //primo e ultimo elemento del ciclo
    int first = population[id_1].solution[0], last;

    for (int i = 0; i < CITY_COUNT; i++){

     /*Questo vettore viene utilizzato per ottenere la
      posizione di un elemento del vettore id_2 nel vettore id_1,
       per farlo salviamo in lookup[i] la posizione dell'elemento da cercare*/

        lookup[population[id_1].solution[i]] = i;
    }

    while (visited < CITY_COUNT){

        child_1[j] = population[parent_1].solution[j];  //Copiamo le citta' nei figli
        child_2[j] = population[parent_2].solution[j];
        last = population[id_2].solution[j];

        flags[j] = 1;       //Impostiamo la colonna j come visitata
        visited++;
        j = lookup[last];   //Cerchiamo last in id_1

        if (first == last){

            /*Una volta terminato il ciclo scambiamo
             i genitori in modo da copiare le citta' al contrario*/

            scambia(&parent_1, &parent_2);
             //Cerchiamo una colonna libera
            for (j = 0; flags[j]; j++);
            //Reinizializziamo il primo elemento del ciclo
            first = population[id_1].solution[j];
        }
    }
}
```

[/codice]

Di seguito vi e' un esempio in cui viene applicato questo algoritmo, serve a mostrare la "potenza" dello stesso e a illustrarne il funzionamento in modo intuitivo.

[esempio]

```
Supponiamo di avere 7 citta' (gli ID vanno da 1 a 7), prendiamo
 2 genitori e due array da inizializzare per inserire i figli.
        Ad esempio [G1 e G2 sono i genitori, F1 e F2 sono i figli]:
    ciclo pari
    G1 -> 1 2 3 4 5 6 7
    G2 -> 3 5 4 1 6 2 7
    F1 -> _ _ _ _ _ _ _
    F2 -> _ _ _ _ _ _ _
    Partiamo dalla prima colonna, che non e' stata ancora fatta:
    Siccome il ciclo e' pari, copiamo "1" nella posizione 1 del
     figlio 1 e copiamo "3" nella posizione 1 del figlio 2
    G1 -> 1 2 3 4 5 6 7
    G2 -> 3 5 4 1 6 2 7
    F1 -> 1 _ _ _ _ _ _
    F2 -> 3 _ _ _ _ _ _
    Cerchiamo ora "3" nel genitore 1. Lo troviamo all'indice 3.
    Ripetiamo ora lo stesso passaggio precedente:
    Siccome il ciclo e' pari, copiamo "3" nella posizione 3
    del figlio 1 e copiamo "4" nella posizione 3 del figlio 2
    G1 -> 1 2 3 4 5 6 7
    G2 -> 3 5 4 1 6 2 7
    F1 -> 1 _ 3 _ _ _ _
    F2 -> 3 _ 4 _ _ _ _
    Cerchiamo ora "4" nel genitore 1. Lo troviamo all'indice 3.
    Ripetiamo ora il solito passaggio:
    Siccome il ciclo e' pari, copiamo "4" nella posizione 4
    del figlio 1 e copiamo "1" nella posizione 4 del figlio 2
    G1 -> 1 2 3 4 5 6 7
    G2 -> 3 5 4 1 6 2 7
    F1 -> 1 _ 3 4 _ _ _
    F2 -> 3 _ 4 1 _ _ _
    Ora attenzione: dovremmo cercare "1" nel genitore 1,
    ma e' l'elemento da cui siamo partiti. Interrompiamo quindi questo ciclo.
    Torniamo quindi a ripetere le stesse iterazioni partendo
    dalla seconda colonna (cioe' la prima che non e' stata visitata).
    Ora siamo in un ciclo dispari, quindi dobbiamo copiare "2"
    nella posizione 2 del figlio 2 e copiare "5" nella posizione 2 del figlio 1
    G1 -> 1 2 3 4 5 6 7
    F1 -> 1 5 3 4 _ _ _
    Cerchiamo "5" nel genitore 1. Lo troviamo alla posizione 5.
    Dato che il ciclo e' dispari, si copi "5" nella posizione 5
    del figlio 2 e si copi "6" nella posizione 5 del figlio 1
    G1 -> 1 2 3 4 5 6 7
    G2 -> 3 5 4 1 6 2 7
    F2 -> 3 2 4 1 5 _ _
    Cerchiamo 6 nel genitore 1, troviamolo nella colonna 6,
     effettuiamo le copie nei figli
    G1 -> 1 2 3 4 5 6 7
    G2 -> 3 5 4 1 6 2 7
```

```
    F1 -> 1 5 3 4 6 2 _
    F2 -> 3 2 4 1 5 6 _
    Siccome 2 e' il numero da cui siamo partiti,
    blocchiamo il ciclo corrente.
    Ora, copiamo i 7 nei figli (e' la prima colonna libera).
    G2 -> 3 5 4 1 6 2 7
    F1 -> 1 5 3 4 6 2 7
    F2 -> 3 2 4 1 5 6 7
    Interrompiamo il ciclo con il 7 perche' 7 stesso e'
    l'elemento da cui partiamo.
    Abbiamo visitato tutte le colonne, fine della riproduzione,
    ritorniamo i figli e prendiamo altri due genitori.

[/esempio]
```

L'ultima procedura che si pone a conclusione del processo e' l'inserimento di mutazioni casuali per introdurre instabilita', evitando che, prossimi alla soluzione, i vettori della popolazione siamo tutti molto simili tra loro e dunque la popolazione peccherebbe di variabilita' tra le soluzioni. Infatti potrebbe verificarsi il caso limite di una popolazione formata da elementi tutti uguali, tanto che con il cross over i figli sarebbero identici ai genitori e quindi si la popolazione non evolverebbe. La funzione di mutazione e' progettata per effettuare all'interno di una soluzione, con una probabilita' del 25% lo scambio tra la posizione di due elementi, ovvero lo scambio tra due citta'.

```
void mutation(individual * population){

    int x, y, p = RAND_MAX * MUTATION_LIKELIHOOD;

    for (int i = 0; i < POPULATION_COUNT; i++){

        if (rand() < p){
            do {
                x = rand() % CITY_COUNT;
                y = rand() % CITY_COUNT;
            } while (x == y);   //evitiamo che x sia uguale a y

            scambia(&population[i].solution[x], &population[i].solution[y]);
        }
    }

    debug("Mutazioni eseguite");
}
```

Una volta inserite le mutazioni casuali, e' stata definitivamente creata la nuova generazione su cui ripetere in maniera iterativa le operazioni sopra viste ripartendo dalla valutazione del fitness. Ripetendo per piu' volte l'intero processo si arriva ad una soluzione finale che non e' necessariamente la migliore in assoluto, ma e' sufficientemente buona, considerando che il numero di iterazioni effettuato e' comunque sensibilmente minore rispetto al numero di iterazioni necessarie per una ricerca esaustiva, dunque esatta.

Abbiamo testato infatti che l'algoritmo genetico trova la soluzione corretta per 10 citta' (che sono poche) in meno di 126.000 combinazioni totali tra tutta la popolazione, contro le 10! permutazioni (circa 3 milioni e mezzo) per dimostrare che e' effettivamente quella la soluzione corretta.

Capitolo 3

Codice

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define DEBUG 0
#define X_MAX 700                   //Dimensione griglia
#define Y_MAX 700
#define CITY_COUNT 25               //Numero di città
#define POPULATION_COUNT 256        //Numero di individui
#define ITERATION_COUNT 10000       //Numero di generazioni
#define MUTATION_LIKELIHOOD 0.25    //probabilità di mutazione in [0,1]

/*-------------------------------------------------------------*/

typedef struct{                 //Coordinate città
    unsigned x, y;
} city;

typedef struct{                 //Struttura individuo
    int solution[CITY_COUNT];   //Vettore solution, ovvero l'ordine di visita delle città
    double fitness;             //Bonta' della soluzione
} individual;

/*-------------------------------------------------------------*/

void debug(char *);
void setup_cities(city *);
void generate_initial_population(individual *);
double get_distance(city *, int, int);
double * evaluate_fitness(city *, individual *);
int * selection(double *);
individual * crossover(individual *, int *);
void mutation(individual *);
void cycle_crossover(individual *, int, int, int *, int *);
```

```
void scambia(int *, int *);
int highest_fitness(individual * population);

/*-------------------------------------------------------------*/

int main(){

    srand(time(NULL));           //Inizializzazione del generatore
      di numeri pseudo casuali

    city cities_map[CITY_COUNT]; //Mappa delle città

    individual * population, * new_gen;

    //populatio contiene POPULATION_COUNT individui
    population = malloc(sizeof(individual) * POPULATION_COUNT);

    setup_cities(cities_map);       //Generazione iniziale della mappa

    FILE * fd;

    //Questo file contiene le informazioni da utilizzare per la grafica
    fd = fopen("data.txt", "w");
    if (fd == NULL){
        printf("Errore apertura file\n");
        return -1;
    }

    fprintf(fd, "%d\n", CITY_COUNT);        //Salviamo il numero di città

    for (int i = 0; i < CITY_COUNT; i++){   //E le loro coordinate
        fprintf(fd, "%d,%d\n", cities_map[i].x, cities_map[i].y);
    }

     /*Genera la popolazione iniziale,
     partiamo da soluzioni casuali per ogni individuo*/

    generate_initial_population(population);

    int best;
    int * candidati;
    double length;
    double * ripartizione;

    for (int i = 0; i < ITERATION_COUNT; i++){

   /*Otteniamo la distribuzione di probabilità (ripartizione)
    associata alla popolazione corrente*/

        ripartizione = evaluate_fitness(cities_map, population);
```

```
        if (i%100 == 0){

  //Troviamo l'indice dell'individuo con la soluzione migliore
           best = highest_fitness(population);

          /*la lunghezza del percorso è 1/fitness
          essendo fitness calcolato come 1/lunghezza*/

           length = 1/population[best].fitness;

          //Salviamo la lunghezza del percorso
           fprintf(fd, "%d %f ", i, length);

           for (int j = 0; j < CITY_COUNT; j++){
                //E l'ordine delle città
               fprintf(fd, "%d ", population[best].solution[j]);
           }
           fprintf(fd, "\n");

           printf("Generazione attuale: %d | Distanza migliore: %f\n", i, length);
        }

    /*canditati è un vettore di interi contenente
    gli indici degli individui scelti secondo
     la distribuzione ripartizione*/

        candidati = selection(ripartizione);

       //Incrociamo gli individui ottenendo la nuova popolazione new_gen
        new_gen = crossover(population, candidati);
        free(population);

        population = new_gen;

  //Applichiamo la mutazione sulla popolazione
        mutation(population);
    }

    fclose(fd);

    return 0;
}

/*-------------------------------------------------------------*/

void debug(char *message){
    if (DEBUG) printf("[D] %s\n", message);
}
```

```
void scambia(int * a, int * b){
    int t = *a;
    *a = *b;
    *b = t;
}

int highest_fitness(individual * population){

    int index = 0;
    double initial = population[0].fitness;
    for (int j = 0; j < POPULATION_COUNT; j++){
     //Troviamo il fitness massimo e restituiamo l'indice
        if (population[j].fitness > initial){
            initial = population[j].fitness;
            index = j;
        }
    }
    return index;
}

void setup_cities(city * cities_map){

    for (int i = 0; i < CITY_COUNT; i++){
      //Genera le coordinate per ogni città in [0, X_MAX) e [0, Y_MAX)
        cities_map[i].x = rand() % X_MAX;
        cities_map[i].y = rand() % Y_MAX;
    }

    debug("Generazione della mappa completata");
}

void generate_initial_population(individual * population){

    int x;

    for (int i = 0; i < POPULATION_COUNT; i++){
        for (int j = 0; j < CITY_COUNT; j++){
            x = rand() % (j+1);     //Numero casuale in [0, j]

            /*All'elemento alla posizione j viene assegnato
             l'elemento alla posizione x (se x strettamente minore
              di j altrimenti assegneremmo a solution[j]
               un valore non inizializzato)*/

            if (x != j){
                population[i].solution[j] = population[i].solution[x];
            }
            population[i].solution[x] = j;
        }
    }
```

```
    debug("Generazione della popolazione iniziale completata");
}

double get_distance(city * cities_map, int id_1, int id_2){

    int x1 = cities_map[id_1].x;
    int y1 = cities_map[id_1].y;
    int x2 = cities_map[id_2].x;
    int y2 = cities_map[id_2].y;

    double dist = sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
    return dist;
}

double * evaluate_fitness(city * cities_map, individual * population){

    int id_1, id_2;
    double total, fitness_sum = 0;

     //Allochiamo spazio per la distribuzione di probabilità
    double * ripartizione = malloc(sizeof(double) * POPULATION_COUNT);
    for (int i = 0; i < POPULATION_COUNT; i++){

        total = 0;  //Contiene la distanza totale tra le città

        for (int j = 0; j < CITY_COUNT-1; j++){
            id_1 = population[i].solution[j];
            id_2 = population[i].solution[j+1];
            //Calcola la distanza tra due città e la somma alla totale
            total += get_distance(cities_map, id_1, id_2);
        }
    //A questo punto il fitness è 1/distanza totale
        population[i].fitness = 1/total;
        //La somma dei fitness viene utilizzata per la "normalizzazione"
        fitness_sum += population[i].fitness;
    }

    ripartizione[0] = population[0].fitness/fitness_sum;

    for (int i = 1; i < POPULATION_COUNT; i++){

     /*Assegna al valore ripartizione
      il valore dell'elemento precedente + il fitness
       corrente normalizzato (diviso per la loro somma)*/

        ripartizione[i] = ripartizione[i-1] + population[i].fitness/fitness_sum;
    }

    debug("Fitness calcolati");
```

```
    return ripartizione;
}

int * selection(double * ripartizione){

    double extraction;
    int pivot, first, last;
    int * candidati = malloc(sizeof(int) * POPULATION_COUNT);

    for (int i = 0; i < POPULATION_COUNT; i++){

        first = 0;
        last = POPULATION_COUNT;

        /*rand() genera un numero in [0, RAND_MAX]
         quindi dividendo per RAND_MAX otteniamo un numero in [0, 1]*/

        extraction = (double)rand()/RAND_MAX;

   /*Eseguiamo una ricerca binaria
    del primo elemento del vettore
     ripartizione con probabilità maggiore di quella estratta*/

        while (first != last){
            pivot = (first + last)/2;
            if (ripartizione[pivot] == extraction){
                break;
            }
            if (ripartizione[pivot] > extraction){
                last = pivot;
            } else {
                first = pivot + 1;
            }
        }
         /*candidati[i] contiene l'indice dell'individuo
          estratto, maggiore è la sua probabilità in ripartizione,
           maggiore è la probabilità di essere in candidati,
           anche ripetutamente*/

        candidati[i] = first;
    }

    free(ripartizione); //Non serve più a questo punto

    debug("Candidati estratti");

    return candidati;
}
```

```
individual * crossover(individual * population, int * candidati){

    individual * new_gen = malloc(sizeof(individual) * POPULATION_COUNT);

    for (int i = 0; i < POPULATION_COUNT; i+=2){

     /*Eseguiamo il crossover a due a due scegliendo come genitori due candidati
      scelti tramite l'indice, salviamo i figli in new_gen*/

        cycle_crossover(population, candidati[i], candidati[i+1], new_gen[i].solution,
         new_gen[i+1].solution);
    }

    free(candidati);    //Non servono più

    debug("Riproduzione effettuata");

    return new_gen;
}

void mutation(individual * population){

    int x, y, p = RAND_MAX * MUTATION_LIKELIHOOD;

    for (int i = 0; i < POPULATION_COUNT; i++){

  /*In questo caso p = RAND_MAX*0.25 = RAND_MAX/4,
  poiché rand() genera un numero in [0, RAND_MAX]
  la condizione rand() < p ha 1 probabilità su 4 di verificarsi*/

        if (rand() < p){
            do {
                x = rand() % CITY_COUNT;
                y = rand() % CITY_COUNT;
            } while (x == y);   //evitiamo che x sia uguale a y

   //Scambiamo le 2 città
            scambia(&population[i].solution[x], &population[i].solution[y]);
        }
    }

    debug("Mutazioni eseguite");
}

void cycle_crossover(individual * population, int id_1, int id_2, int * child_1,
int * child_2){

    int parent_1 = id_1, parent_2 = id_2;           //Copiamo gli indici dei genitori
    int j = 0;                                      //Colonna corrente
    int lookup[CITY_COUNT];                         //Vettore di indirizzamento
```

```
    int flags[CITY_COUNT] = {0};                    //Vettore delle colonne visitate
    int visited = 0;                                //Numero di colonne visitate
    int first = population[id_1].solution[0], last; //primo e ultimo elemento
     // del ciclo

    for (int i = 0; i < CITY_COUNT; i++){

    /*Questo vettore viene utilizzato per ottenere la
     posizione di un elemento del vettore id_2 nel vettore id_1,
     per farlo salviamo in lookup[i] la posizione dell'elemento da cercare*/

        lookup[population[id_1].solution[i]] = i;
    }

    while (visited < CITY_COUNT){
  //Copiamo le città nei figli
        child_1[j] = population[parent_1].solution[j];
        child_2[j] = population[parent_2].solution[j];
        last = population[id_2].solution[j];

        flags[j] = 1;       //Impostiamo la colonna j come visitata
        visited++;
        j = lookup[last];   //Cerchiamo last in id_1

        if (first == last){

          /*Una volta terminato il ciclo scambiamo
           i genitori in modo da copiare le città al contrario*/

            scambia(&parent_1, &parent_2);
             //Cerchiamo una colonna libera
            for (j = 0; flags[j]; j++);
             //Reinizializziamo il primo elemento del ciclo
            first = population[id_1].solution[j];
        }
    }
}
```

Capitolo 4

Complessità Computazionale

Per calcolare la complessità computazionale del programma, siamo partiti dalle funzioni più elementari e poi via via fino a quelle più complesse, che inglobano le prime.
In maniera molto semplice possiamo subito vedere che le funzioni quali "debug", "scambia" e "get_distance" hanno complessità uguale ad 1 sia nel caso medio, che in quello peggiore (non hanno cicli al loro interno e svolgono un numero finito di operazioni elementari indipendenti da n).
Contrariamente funzioni quali "highest_fitness", "setup_cities" e "mutation" hanno complessità n in quanto sono funzioni che scorrono almeno una volta (non ci sono cicli annidati) il vettore che gli viene passato, e sul quale eseguono operazioni elementari.

Decisamente più complesse ed interessanti sono le altre funzioni:

- *generate_initial_population*: tale funzione ha un doppio ciclo (annidato) che comporta un incremento della complessità computazionale che diventa quadratica. Il caso medio differisce dal caso peggiore per un termine costante (che in tabella non è stato indicato), ma per n che tende ad infinito il limite asintotico corrisponde allo stesso del caso peggiore.

- *evaluate_fitness*: anche qui abbiamo più cicli, ma due in particolare sono annidati e fanno si che la complessità sia quadratica.

- *selection*: in questa funzione abbiamo due cicli annidati, di cui uno ha complessità n, mentre l'altro grazie all'uso dell'algoritmo di ricerca binaria, ha complessità $log(n)$. Ciò fa si che la complessità totale della funzione sia il prodotto delle due: $nlog(n)$.

- *cycle_crossover*: in questa funzione abbiamo due cicli annidati in cui nel primo il numero di iterazioni è fisso e comporta una complessità n, mentre nel secondo abbiamo che la complessità può variare in base alla disposizione del vettore da riprodurre. Abbiamo un caso medio in cui consideriamo (per convenzione) che il ciclo più interno compia un numero di iterazioni pari a $n/2$ (con n lunghezza vettore), ed un caso peggiore in cui compia

FUNZIONE	C.C. CASO MEDIO	C.C. CASO PEGGIORE
debug	$\theta(1)$	$O(1)$
scambia	$\theta(1)$	$O(1)$
highest_fitness	$\theta(n)$	$O(n)$
setup_cities	$\theta(n)$	$O(n)$
generate_initial_population	$\theta(n^2)$	$O(n^2)$
get_distance	$\theta(1)$	$O(1)$
evaluate_fitness	$\theta(n^2)$	$O(n^2)$
selection	$\theta(nlog(n))$	$O(nlog(n))$
crossover	$\theta(n^3)$	$O(n^3)$
mutation	$\theta(n)$	$O(n)$
cycle_crossover	$\theta(n^2)$	$O(n^2)$
main	$\theta(n^4)$	$O(n^4)$

Tabella 4.1: Complessità computazionale

n iterazioni. Dunque nei due casi la complessità differisce per un fattore costante (che nella tabella non riportiamo) che con il limite asintotico sparisce. Sicuramente però il *tempo effettivo di esecuzione* risentirà di tale costante.

- *crossover*: tale funzione è solo una comodità usata per spezzare la funzione completa del "cross over" in una funzione in cui fosse contenuto l'algoritmo vero e proprio e una chiamante. In particolare questa (che è la chiamante) non fa altro che eseguire in un ciclo *for* la precedente funzione (quella contenente l'algoritmo vero e proprio) "cycle_crossover", per n volte. Ciò porta ad avere una complessità cubica.

- main: infine nel "main" abbiamo le chiamate alle precedenti funzioni, ma in particolare ciò che fa salire la complessità computazionale è la chiamata alla funzione "crossover" (appena esaminata) inserita nel ciclo *for* con complessità n. Ciò provoca una complessità totale n^4.

Possiamo affermare dunque che tale algoritmo è decisamente "goloso".

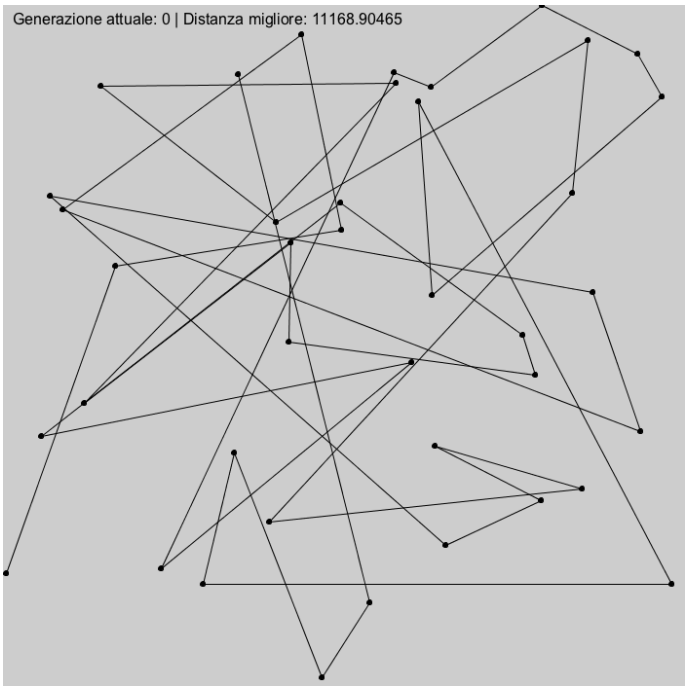

Figura 4.1: "percorso di partenza"

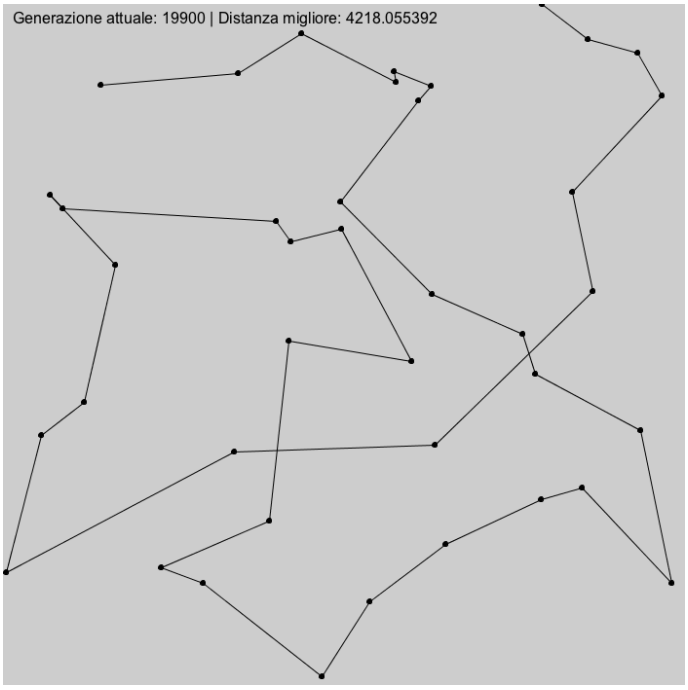

Figura 4.2: "percorso finale"