DIPARTIMENTO DI MATEMATICA E INFORMATICA

CORSO DI LAUREA IN INFORMATICA

Corso di "Business Processes Design Management and Optimization"

Algoritmi di ottimizzazione per problemi di Routing - Profitable Tour Problem

Docente:
M.E. Bruni

Anastasio Marco 188057
Brusco Lorenzo 187641

Anno Accademico 2017-2018

Indice

1 Introduzione

Il progetto sviluppato ha l'intento di studiare e implementare alcuni algoritmi per la risoluzione di problemi di *routing e di scheduling* di tipo deterministico. La risoluzione di questa categoria di problemi ha lo scopo di trovare il percorso migliore da eseguire, cercando di ottimizzare uno dei vari obiettivi disponibili. In particolare, la tipologia di problema preso in considerazione è quella di una variante del noto problema del **Traveling Salesman Problems (TSP)** che è uno dei problemi di ottimizzazione combinatoria più noti e complessi. Il problema in questione è noto come *Profitable Tour Problem (TPT)*, con l'obiettivo di massimizzare i profitti. Gli algoritmi studiati per la sua risoluzione sono il **Genetic Algorithm (GA)**, **Greedy Randomized Adaptive Search Procedure (GRASP)** ed il **Greedy approach**.

2 Traveling Salesman Problems

Il *problema del commesso viaggiatore* consiste nel trovare il percorso più breve che passi per tutte le città di un elenco dato e ritorni al punto di partenza. Per percorso minimo si intende quello di costo minore per ogni coppia di città. Un'altra semplificazione che si aggiunge 'e quella di considerare i costi per andare dalla città A alla B uguali ai costi per andare dalla B alla A. In questo caso si parla di problema simmetrico.

Il numero totale dei differenti percorsi possibili attraverso le n città è facile da calcolare: data una città di partenza, ci sono a disposizione n−1 scelte per la seconda città, n−2 per la terza e così via. Il totale delle possibili scelte tra le quali cercare il percorso migliore in termini di costo è dunque (n−1)!, ma dato che il problema ha simmetria, questo numero va diviso a metà. Insomma, date n città, ci sono $\frac{(n-1)!}{2}$ percorsi che le collegano. Si vede subito che questo numero cresce esponenzialmente all'aumentare di n, e questo è il motivo più citato per motivare la complessità del problema: non è possibile, infatti, controllare uno alla volta i singoli percorsi.

Tale modello viene utilizzato non solo nella ricerca di percorsi ma anche in altri ambiti:

1. **Un circuito stampato** anche conosciuto come PCB ovvero "printed circuit board" in lingua inglese , nell'elettronica, è un supporto utilizzato per interconnettere tra di loro i vari componenti elettronici di un circuito tramite piste conduttive incise su di un materiale non conduttivo.

2. **setup costs** Ridurre al minimo i costi totali di installazione (o dei tempi di installazione). Tutti i tipi di prodotti devono essere prodotti

3. **communication** Pianificazione di nuove reti per la telecomunicazione

È stato dimostrato che TSP è un problema NP-hard (più precisamente, è NP-complete) e la versione decisionale del problema ("dati i pesi e un numero x, decidere se ci sia una soluzione migliore di x") è NP-completa. Il problema rimane NP-hard anche in molte sue versioni ridotte, come nel caso in cui le città siano in un piano con distanze euclidee. Inoltre, omettere la condizione di visitare una città "una e una sola volta" non rimuove la NP-hard.

2.1 Profitable Tour Problem

Questo problema è definito su un grafo, come nel TSP, ma con l'aggiunta dei profitti su ogni nodo. E l'obiettivo non è più quello di minimizzare i tempi, ma è quello di massimizzare il guadagno sottraendo i costi di trasporto.

3 Algoritmi di risoluzione

Il "Traveling Salesman Problem" è un noto problema di ottimizzazione NP-hard; dato un grafo G=(V,E) in cui l'insieme di vertici V=1,....,N rappresentano le città ed una matrice NxN, associata all'insieme degli archi E = $\{(i,j) : i \neq j; i,j \in V\}$, dove l'elemento [i, j] è la distanza dalla città i alla città j, si richiede di trovare il ciclo Hamiltoniano[1] di minimo costo.
Considerando la sua complessità non è stato ancora trovato un algoritmo che riuscisse a risolverlo in modo deterministico in tempo polinomiale, per questo motivi sono stati utilizzati tre algoritmi che non ricercano il miglior risultato possibile ma quello relativo, infatti prendono il nome di *procedure metaheuristic*. In computer science and mathematical optimization, un metaheuristic è una procedura di alto livello progettata per trovare, generare e selezionare un euristica (algoritmo di ricerca parziale) che potrebbe fornire una soluzione sufficientemente buona ad un problema di ottimizzazione.

3.1 Genetic algorithm

Un algoritmo genetico è un algoritmo euristico ispirato al principio della selezione naturale ed evoluzione biologica teorizzato nel 1859 da Charles Darwin.
L'aggettivo "genetico" deriva dal fatto che il modello evolutivo darwiniano trova spiegazioni nella branca della biologia detta genetica e dal fatto che gli algoritmi genetici attuano dei meccanismi concettualmente simili a quelli dei processi biochimici scoperti da questa scienza.

In sintesi si può dire che gli algoritmi genetici consistono in algoritmi che permettono di valutare delle soluzioni di partenza e che ricombinandole ed introducendo elementi di disordine sono in grado di crearne di nuove nel tentativo di convergere a soluzioni ottime.
Queste tecniche vengono di norma utilizzate per tentare di risolvere problemi di ottimizzazione per i quali non si conoscono altri algoritmi efficienti di complessità lineare o polinomiale. Nonostante questo utilizzo, data la natura intrinseca di un algoritmo genetico, non vi è modo di sapere a priori se sarà effettivamente in grado di trovare una soluzione accettabile al problema considerato.

La struttura alla base di questo algoritmo è quella di selezionare le soluzioni migliori e di ricombinarle in qualche modo fra loro in maniera tale che esse evolvano verso un punto di ottimo. Nel linguaggio degli GA la funzione da massimizzare/minimizzare prendi il nome di **fitness** F.
Si supponga che la funzione di fitenss dipenda da n variabili: F = f(x_1, x_2,...., x_n). Un set di n valori x_1, x_2,...., x_n appartenenti ad un certo intervallo viene detta **individuo**.

[1] Cammino Hamiltoniano se esso tocca tutti i vertici del grafo una e una sola volta.

Un insieme di individui forma una **popolazione**.
Una soluzione può essere codificata in codice binario, tale sequenza (stringa) di 0 e 1 che costituisce un individuo è detta **cromosoma**.

In natura gli individui si riproducono mescolando in questo modo i propri patrimoni genetici, cioè i loro cromosomi: i nuovi individui generati avranno pertanto un patrimonio genetico derivato in parte dal padre e in parte dalla madre. La selezione naturale (cioè il riutilizzo di soluzioni buone) fa sì che riescano a sopravvivere e quindi a riprodursi solo gli individui più forti, "più adatti", cioè quelli con la fitness più elevata (più vicini all'ottimo); la fitness media della popolazione tenderà quindi ad aumentare con le generazioni, portando così la specie ad evolversi nel tempo.

3.1.1 Struttura di un GA

1. **Definizione della generazione iniziale**.
   definire (anche in maniera random) un primo insieme di possibili soluzioni al problema in esame.

2. **Valutazione di ogni soluzione e selezione delle migliori**.
   Valutare ogni soluzione, associando a ciascuna un indicatore di qualità (o fitness) in modo da poterle ordinare.

3. **Definizione di una nuova generazione**.
   Definire un nuovo gruppo di soluzioni modificando opportunamente le soluzioni con qualità elevata, in modo da favorire lo sviluppo a scapito di quelle peggiori.

4. **Conclusione dell'elaborazione**.
   Se il numero di iterazioni stabilite è stato raggiunto o la qualità della migliore soluzione disponibile è accettabile si può terminare l'elaborazione, altrimenti si ritorna al passo 2 per definire un nuovo gruppo di soluzioni.

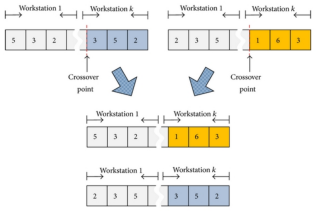
3.1.2 Selezione

La selezione di un individuo dipende dal suo valore di fitness (cioè quanto l'individuo è "buono" a risolvere il problema): un valore più alto di fitness corrisponde ad una maggiore possibilità di essere scelto come **genitore** (parent) per creare la nuova generazione.
Uno dei criteri più utilizzati è quello di Holland che attribuisce una probabilità di scelta proporzionale al fitness. Grazie al meccanismo della selezione, solo gli individui migliori hanno la possibilità di riprodursi e quindi di trasmettere il loro genoma alle generazioni successive.

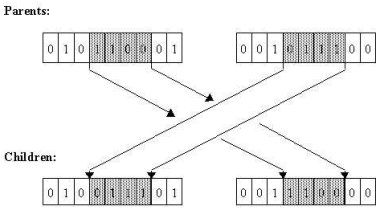
3.1.3 Crossover e Mutazione

Per la creazione di nuovi individui ad ogni generazione si utilizzano due operazioni ispirate dall'evoluzionismo biologico: il **crossover** e la **mutazione**. Una volta individuati una coppia di genitori con il meccanismo della selezione, i loro genomi possono andare incontro a ciascuna delle due operazioni con probabilità rispettivamente $\mathbf{P_{cross}}$ (probabilità di crossover per coppia di individui, detta crossover rate) e $\mathbf{P_{mut}}$ (probabilità di mutazione per gene, detta mutation rate).

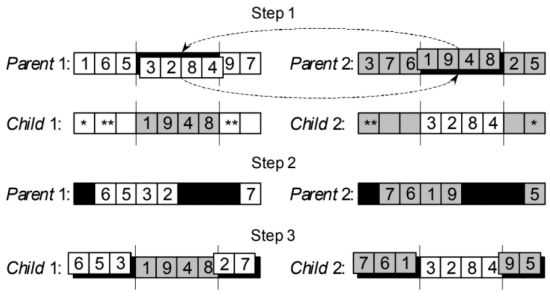
1. **One Point Crossover** È la tecnica più semplice, date due stringhe viene individuato un punto che separa ciascuna di esse in due parti, in seguito si ricombinano come mostrato in figura.
   La probabilità che si verifichi il cross over è in genere abbastanza alta (in caso contrario è possibile avere figli che sono uguali ai genitori).



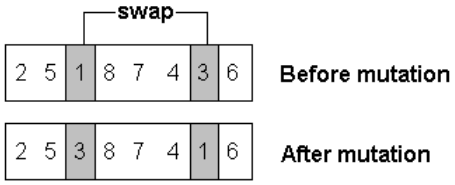
2. **Standard Crossover** Simile al one point crossover ma si selezionano due punti di taglio suddividendo la stringa in tre sottostringhe.



3. **Order Crossover** Come lo standard crossover lavora su coppie di cromosomi. I due punti di taglio sono scelti a caso. Ciascum figlio è costruito prendendo la crossing section (la parte centrale del taglio) di un genitore e riempiendo gli spazi rimanendi con i simboli mancanti presi nell'ordine in cui compaiono nell'altro genitore.



Mutation data la probabilità di mutazione, per ciascuna cifra di ogni individuo si estrae un numero casuale q: se q $\leq \mathbf{P_{mut}}$, la cifra binaria viene scambiata (1 diventa 0 e 0 diventa 1), nel caso in cui la sequenza non è binaria si effettua uno **swap** con un altro elemento dello stesso individuo.



3.2 Greedy Randomized Adaptive Search Procedure

The greedy randomized adaptive search procedure(conosciuto anche col nome GRASP) è un metaheuristic algorithm comunemente applicato in problemi combinatori di ottimizzazione. GRASP consiste tipicamente di iterazioni composte da successive costruzioni di una soluzione randomizzata e greedy e di successivi miglioramenti iterativi attraverso una ricerca locale.

Una volta definita la funzione neighborhood N pi'u adatta al problema da affrontare, esso genera ad ogni iterazione una soluzione ammissibile x diversa ed applica ad essa una procedura di ricerca locale al fine di individuare in N (x) una soluzione ammissibile migliorativa. Ciascuna iterazione GRASP è formata da due fasi:

1. una fase di costruzione.
2. una fase di ricerca locale.

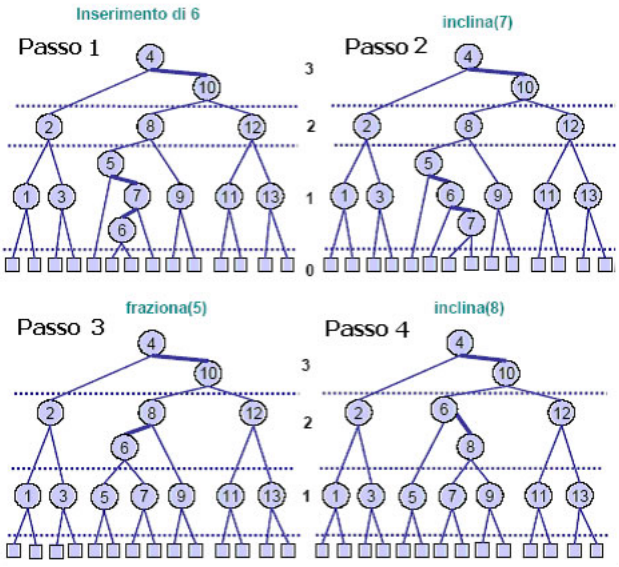
```
procedure ConstructGreedyRandomizedSolution(α)
1   x = ∅;
2   ordina i candidati in accordo ad un prestabilito criterio
    greedy
3   while (x non completa) do
4       RCL = MakeRCL(α);
5       v=Select(RCL);
6       x = x ∪ {v}
7       riordina i candidati coerentemente alla scelta fatta
8   endwhile
9   return (x)
end ConstructGreedyRandomizedSolution
```

Dal momento che sperimentalmente si è mostrato che migliore è la soluzione ammissibile x migliore è la soluzione neighbor $\overline{x}$ e, dunque, la soluzione output x_b, la fase di costruzione di un algoritmo GRASP è tesa proprio a trovare una soluzione iniziale (per la ricerca locale) x abbastanza buona e per raggiungere questo scopo adotta una strategia di tipo greedy randomizzata ed adattiva. A partire da una soluzione x vuota, essa procede iterativamente e ad ogni iterazione aggiunge un nuovo elemento ad x fino ad ottenere una soluzione ammissibile completa. La scelta di tale nuovo elemento dipende dall'ordine con cui i potenziali candidati sono memorizzati in una lista L. Tale ordine riflette un prefissato criterio greedy basato sul "beneficio" legato alla selezione dell'elemento. Detto beneficio, tuttavia, non è costante, ma adattivo, nel senso che è aggiornato ad ogni iterazione della fase.

3.3 Greedy approach

Un algoritmo greedy è un algoritmo che cerca di ottenere una soluzione ottima da un punto di vista globale attraverso la scelta della soluzione più golosa (aggressiva o avida, a seconda della traduzione preferita del termine greedy dall'inglese) ad ogni passo locale. Questa tecnica consente, dove applicabile (infatti non sempre si arriva ad una soluzione ottima), di trovare soluzioni ottimali per determinati problemi in un tempo polinomiale (cfr. Problemi NP-Completi, cioè problemi di soluzione non deterministica polinomiale).

Tecniche greedy sono particolarmente utilizzate nel caso di problemi di ottimizzazione "su insiemi", cioè problemi P in cui, data un'istanza $x \in I$, una soluzione y è un sottoinsieme di x che soddisfa una determinata proprietà ed ha un determinato valore m(x,y). Data un'istanza $x = \{e_1, \ldots, e_n\}$ ogni elemento ha un "valore" e la soluzione viene costruita utilizzando prima gli elementi di maggior valore.



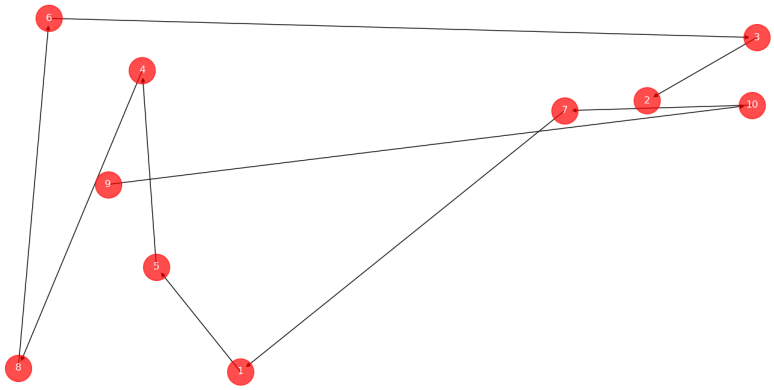
4 Formulazione

Dopo aver analizzato a fondo il problema ed aver letto diversi papers siamo riusciti a formulare un modello matematico per il **TPT**.
Sia G = (V, E) un grafo indiretto completo, V = {0,.....,n } l'insieme dei nodi e E = {(i,j) : i,j ∈ V, i ≠ j }. Una possibile soluzione è un'insieme V' = {0,....,n} con ‖V'‖ = ‖V‖. Sia $d_{i,j}$ la distanza, in termini di costo, dal nodo i al nodo j, c_j il costo per soddisfare l'ordine del nodo j e g_j il guadagno ottenuto arrivando al nodo j-esimo, allora possiamo definire la nostra funzione obiettivo:

$$max \sum g_{\mathrm{i}} - (d_{\mathrm{i,j}} + c_{\mathrm{i}}), i = 0, ...n$$

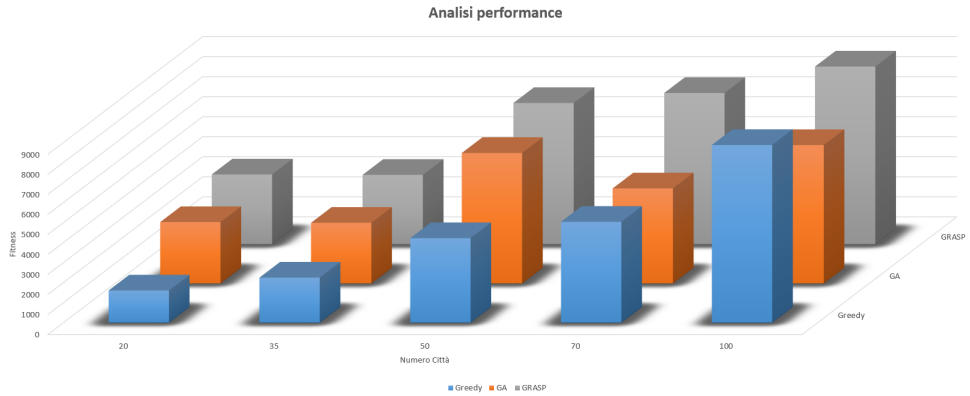
5 Implementazione

I tre algoritmi illustrati in precedenza sono stati implementati in un applicazione Python che stampa su console tutti i passi eseguiti dagli algoritmi e mostra, con un interfaccia il risultato sotto forma di grafo. Per l'implementazione dell'algoritmo Greedy è stata utilizzato lo stesso principio utilizzato nella risoluzione dello *Knapsack problem*, ovvero etichettare i nodi già sceltie ordinare i profitti in modo decrescente e prendere il migliore, fra quelli non ancora etichettati, per quanto riguarda il *genetic algorithm*: è stato utilizzando un metodo random per la generazione iniziale della popolazione uno *swap value* per la mutazione ed un *order crossover*. Mentre l'ultimo, il *grasp* è stato implementato nel maniera sopracitata.

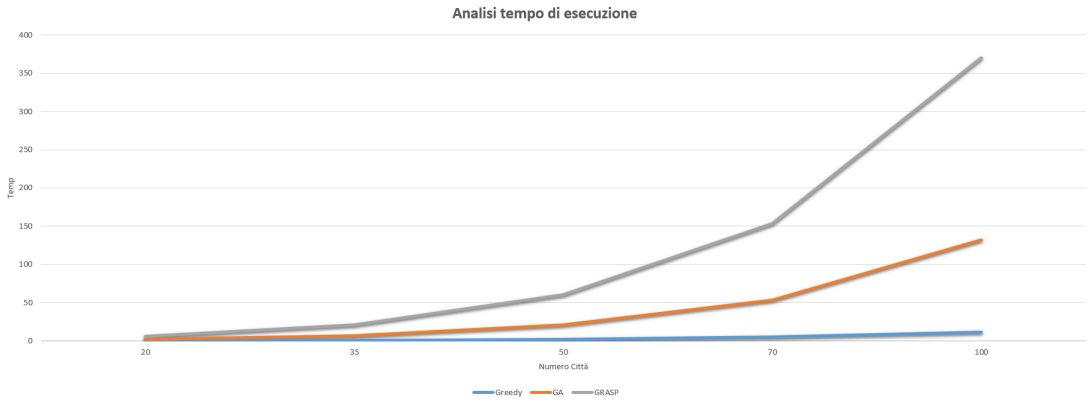


6 Risultati

Dai risultati ottenuti si evince che **GRASP** è il migliore, in tutti i casi, per quanto riguarda la massimizzazione della funzione obiettivo, a seguire **Genetic Algorithm**, che però soffre un po' in caso di un elevato numero di nodi, a differenza del **Greedy approach** che migliora con un elevato numero di nodi.



Ma la vera differenza la si può notare nella tempistiche, che sia in **GRASP** e sia in **GA** aumentano in modo esponenziale all'aumentare dei nodi, mentre il **Greedy approach** si mantiene decisamente più polinomiale.



7 Considerazioni Finali

L'applicazione potrebbe essere arricchita migliorando le funzioni di input del problema, che attualmente è inseribile solo attraverso un file excel e non attraverso un interfaccia grafica. Si potrebbe inoltre procedere all'ottimizzazione di alcune parti di codice per migliorare le performance e ridurre i tempi di esecuzione.

Riferimenti bibliografici

[Rahma Lahyani] [Mahdi Khemakhem]Khemakhem[Frédéric Semet]Semet *A unified matheuristic for solving multi-constrained traveling salesman problems with profits*, 2016.

[Burak Suyunu] *Final project- tabu search on travelling salesperson problems with profits*

[Dominique Feillet] [Pierre Dejax] Dejax[Michel Gendreau]Gendreau *Travelling salesman problems with profits*