Relazione esercitazione di laboratorio

Marco Zanella

31 dicembre 2015

Indice

Sommario

Il Problema del Commesso Viaggiatore (TSP) è una delle applicazioni più note del calcolo combinatorio. È classificato come problema *NP-Hard*, per questo motivo risolutori basati su metodi esatti possono impiegare molto tempo e molte risorse computazionali prima di produrre una soluzione.

In alcuni contesti, il possibile dispendio di tempo e risorse è giustificato dal vantaggio dato dall'ottenere la soluzione ottima ma, in altre applicazioni, è preferibile ottenere soluzioni subottime in tempo minore. Per questo motivo sono state sviluppate euristiche e metaeuristiche per la risoluzione di molti problemi di natura combinatoria, tra cui il TSP.

In questa relazione sono presentati e messi a confronto un metodo esatto basato su Programmazione Lineare Intera (PLI) ed un Algoritmo Genetico per la soluzione del TSP.

Capitolo 1

Assunzioni e motivazioni

1.1 Costo degli archi

Si assume che, dati due nodi, esista al più un arco che porti dal primo al secondo. Nel caso esistessero più archi possibili, essi sarebbero noti a priori e la macchina foratrice sceglierebbe comunque quello meno costoso, riconducendosi allo scenario con al più un collegamento diretto. La non esistenza di un arco tra due nodi viene per convenzione rappresentata con una distanza negativa.

Non sono assunte ulteriori proprietà riguardo i costi degli archi: la matrice dei costi non è necessariamente simmetrica, le distanze non soddisfano necessariamente la diseguaglianza triangolare e il grafo non è necessariamente connesso. Nel caso in cui il grafo non sia connesso, il problema non ha soluzione.

L'assunzione di non avere un grafo fortemente connesso mira a mettere in forte difficoltà i risolutori basati su euristiche/metaeuristiche, in quanto molti degli algoritmi documentati nella letteratura sono in grado di garantire l'ammissibilità delle soluzioni generate solo a condizione che il grafo sia fortemente connesso. Non a caso, in letteratura sono poche le fonti a trattare il TSP su grafi debolmente connessi.

L'assunzione dell'asimmetria ha motivazioni analoghe: è più difficile lavorare con matrici di distanze asimmetriche. Basti pensare all'esempio di TSP con Tabù Search visto a lezione: quando si inverte parte di un cammino (2-opt) nel TSP simmetrico, è possibile aggiornare il costo della soluzione sottraendo i costi dei due archi rimossi e sommando i costi dei due archi inseriti. Questo non è più vero quando le distanze sono asimmetriche: invertendo la direzione degli archi, tutti i costi associati cambiano ed occorre rivalutare la soluzione (o almeno la porzione interessata).

L'appendice B mostra un caso di studio semplificato basato su grafi simmetrici e fortemente connessi.

1.2 Ordinamento dei fori

Si assume che il nodo da cui cominciare il percorso sia il primo nodo presente nell'istanza del problema. Tale assunzione non fa perdere di generalità in quanto la ricerca riguarda un ciclo Hamiltoniano.

1.3 Costo della creazione di un modello (CPLEX)

Si assume che il costo della creazione del modello con CPLEX sia trascurabile rispetto al tempo di risoluzione. Questa assunzione può risultare errata per istanze particolarmente piccole ($\approx 5, 10$), ma si mostra ampliamente corretta per istanze significative. Nel caso di istanze piccole, inoltre, il tempo globale di esecuzione è comunque ridotto.

Per questi motivi, e per la sintassi oscura di CPLEX, viene utilizzata una classe *CPLEXManager* che realizza il *Facade Design Pattern*: nuove variabili e vincoli vengono aggiunti tramite opportuni metodi, uno alla volta, tramite un'interfaccia semplificata. Trattandosi di un *Facade*, non esibisce la stessa flessibilità della *CPLEX Callable Library* (o di wrapper ufficiali come *Concert Technology*) ed è ragionevolmente meno efficiente ma, sotto l'ipotesi del tempo di creazione del modello trascurabile, l'impatto sul tempo di esecuzione è nullo.

1.4 Costo della popolazione iniziale (GA)

Si assume che il costo per generare una popolazione iniziale per l'algoritmo genetico si trascurabile rispetto al tempo di risoluzione. Quest'assunzione è ragionevole poiché i criteri utilizzati nella creazione della popolazione sono il *Random Walk* ed una semplice euristica greedy.

Al contrario, essendo il tempo di risoluzione dell'algoritmo genetico dominante, molte sue componenti sono state scritte in C anziché C++ per migliorarne l'efficienza (a scapito della semplicità di lettura e dell'espressività dei linguaggi orientati ad oggetti). Per lo stesso motivo viene utilizzata la gestione diretta della memoria, anziché affidarsi ai più dispendiosi meccanismi di C++. Strumenti di analisi dinamica sono stati utilizzati per testare la corretta gestione della memoria.

Capitolo 2

Modello di PLI

Il modello di PLI proposto si basa su quello indicato nella traccia del progetto.

2.1 Rimozione dei cappi

Poiché nel TSP (così come nel più generale cammino Hamiltoniano) ogni nodo è visitato una sola volta, gli archi che congiungono un nodo con se stesso (detti *cappi*) non saranno mai percorsi. Per questo motivo è ragionevole non inserire nel modello le variabili decisionali che corrispondono a questi archi. Le variabili decisionali diventano quindi:

- x_{ij}: numero di unità di flusso trasportate dal nodo i al nodo j, $\forall (i,j) \in A, i \neq j$
- y_{ij}: 1 se l'arco (i,j) viene utilizzato, 0 altrimenti, $\forall (i,j) \in A, i \neq j$

Si osserva che quest'operazione rende il modello più semplice: vengono introdotte meno variabili e meno vincoli.

2.2 Archi mancanti

Poiché il grafo non è assunto essere fortemente connesso (sez. 1.1), alcuni archi potrebbero non essere presenti. Per come è definito il modello (in termini di A, l'insieme degli archi), questo non comporta modifiche. Si osserva tuttavia che il modello viene reso più semplice in quanto il numero di variabili e vincoli inseriti è minore od uguale a quello che si avrebbe con un grafo fortemente connesso.

Capitolo 3

Algoritmo Genetico

3.1 Algoritmo

L'algoritmo genetico proposto si serve di una ricerca locale per migliorare i cromosomi che codificano buone soluzioni ed utilizza valori adattivi per le probabilità di crossover e mutazione [1, Laoufi et al.].

L'insieme dei cromosomi che codificano soluzioni ammissibili è chiuso rispetto agli operatori di crossover e mutazione a condizione che il grafo sia fortemente connesso. Qualora così non fosse, l'approccio seguito consiste nel penalizzare le soluzioni non accettabili.

La fig. 3.1 riassume le strategie adottate.

3.2 Parametri

L'algoritmo genetico proposto è parametrico rispetto a:

pCrossover Massima probabilità che si verifichi un crossover; la probabilità reale è calcolata in maniera adattiva: $p \in [0, pCrossover]$

pMutation Massima probabilità che si verifichi una mutazione; la probabilità reale è calcolata in maniera adattiva: $p \in [0, pMutation]$

threshold Soglia al di sotto della quale due cromosomi sono considerati simili; la distanza di Hamming viene utilizzata come misura di somiglianza: $threshold \in [0, 1]$

pAccept Probabilità di accettare un cromosoma quando ce ne è già uno simile presente nella popolazione: $pAccept \in [0, 1]$

pImprove Probabilità di cercare di migliorare un cromosoma considerato *promettente*: $pImprove \in [0, 1]$; un cromosoma è *promettente* quando il costo associato alla soluzione da esso codificata è inferiore al costo medio della popolazione meno lo scarto quadratico medio dei costi

size Dimensione della popolazione

maxIter Massimo numero di iterazioni

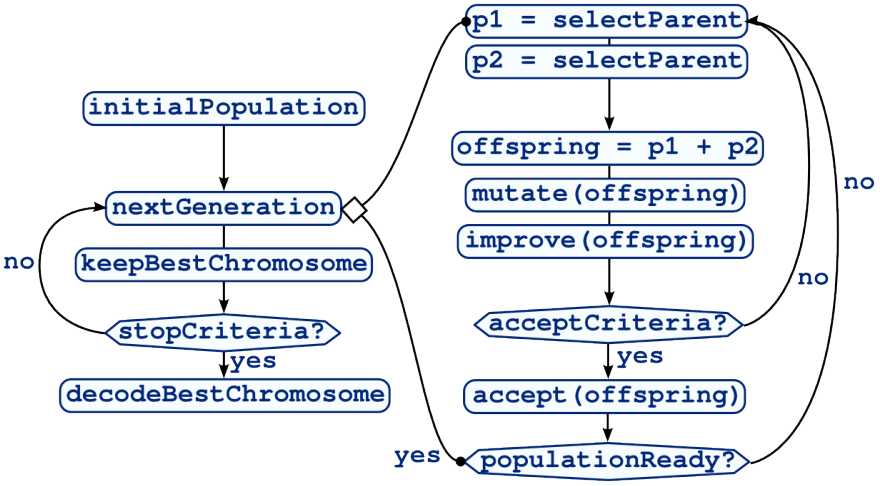

Figura 3.1: Schema dell'algoritmo genetico

maxSlack Numero massimo di iterazioni senza miglioramento; permette di identificare convergenze premature

maxTime Tempo massimo di esecuzione, in secondi

Si osserva come sia sufficiente che uno solo tra *MaxIter*, *maxSlack* e *maxTime* sia verificato per terminare la ricerca.

Dipendere da un elevato numero di parametri è uno svantaggio noto degli algoritmi genetici. Nell' Appendice A sono spiegate in dettaglio le tecniche utilizzate per la calibrazione del risolutore.

3.3 Componenti

3.3.1 Codifica e decodifica

La codifica utilizzata è il *Path Representation*: i cromosomi sono liste di interi, ad ogni intero corrisponde un nodo del grafo. Viene scelta questa rappresentazione in quanto permette di definire buoni operatori di crossover e mutazione [2, Abdoun et al.], [3, Chatterjee].

3.3.2 Popolazione iniziale

Gli individui che compongono la popolazione iniziale sono generati indipendentemente gli uni dagli altri. Ogni individuo è costruito risolvendo l'istanza del problema con un algoritmo *Random Path* o un'euristica Greedy (scegliendo non deterministicamente uno dei due approcci).

Il *Random Path* considera una permutazione casuale dei nodi, e la utilizza come ordine di visita. Il costo computazionale è praticamente nullo, ma le soluzioni hanno generalmente una bassa qualità (e potrebbero non essere ammissibili).

L'euristica Greedy parte dal nodo iniziale e si sposta, ad ogni passo, verso il nodo più vicino non visitato. Il costo computazionale è minimo, ma la qualità

delle soluzioni è poco stabile: per alcune istanze le soluzioni sono molto buone, per altre sono pessime o non ammissibili. Il problema con questo tipo di euristica risiede nel fatto che i primi archi scelti hanno costo basso ma, man mano che l'algoritmo procede, gli archi possibili tendono ad avere costi più alti.

Questa scelta di generazione della popolazione iniziale è motivata dalla necessità di avere sia soluzioni il più diverse possibili (quindi è importante il *Random Path*), sia di introdurre fin dall'inizio delle *buone caratteristiche*, o *building blocks* (ad esempio grazie all'algoritmo greedy). È anche importante che la generazione della popolazione iniziale sia estremamente veloce, a maggior ragione l'uso di permutazioni casuali e greedy risulta giustificato.

3.3.3 Selezione

La selezione degli individui avviene utilizzando il *Linear Ranking*. Altre alternative, come *Roulette Wheel* o *Tournament Selection*, producono risultati meno stabili, o funzionano bene solo per istanze relativamente piccole [4, Oladele et al.], [5, Noraini et al.].

Il *Linear Ranking* opera utilizzando l'informazione sulla posizione in classifica dei cromosomi, richiede quindi un ordinamento della popolazione ad ogni iterazione. Si è osservato sperimentalmente che l'overhead dovuto all'ordinamento è trascurabile.

3.3.4 Crossover

Va osservato che, senza l'assunzione del grafo fortemente connesso, risulta impossibile definire operatori di crossover e mutazione che garantiscano risultato ammissibile: le istanze rappresentate da un grafo sconnesso, ad esempio, non hanno alcuna soluzione ammissibile. Segue che, comunque siano definiti operatori di crossover e mutazione, essi non produrranno mai una soluzione ammissibile.

L'approccio proposto consiste nell'utilizzare comunque operatori *buoni*, ovvero che garantiscono ammissibilità a condizione di avere il grafo fortemente connesso, andando poi a penalizzare le soluzioni non ammissibili. Questa scelta è motivata dal fatto che, se il grafo è poco connesso, non ci sono forti motivazioni per utilizzare un operatore piuttosto di un altro: qualunque operatore si usi, esiste un modo di costruire un'istanza che lo sfavorisca. Per contro, tanto più il grafo è connesso, tanto meglio funzioneranno gli operatori *buoni*.

Viene utilizzato il *1-cut Ordering Crossover* tra due cromosomi. Dopo aver selezionato i due genitori, essi sono soggetti a crossover con probabilità $p = pCrossover \cdot \frac{\sigma_c}{c_w - \overline{c}} \in [0, pCrossover] \subseteq [0, 1]$, dove σ_c rappresenta la deviazione standard dei costi delle soluzioni ammissibili presenti nella popolazione, $\overline{c}$ il loro costo medio e c_w il costo della soluzione peggiore nella popolazione [1, Laoufi et al.]. Nel caso in cui il crossover non avvenga, il nuovo cromosoma è la copia di uno dei due genitori.

3.3.5 Mutazione

Valgono le osservazioni del punto precedente.

La mutazione proposta è il *2-opt*, ovvero l'inversione di una porzione della sequenza del cammino. Questa strategia comporta la rimozione di due archi,

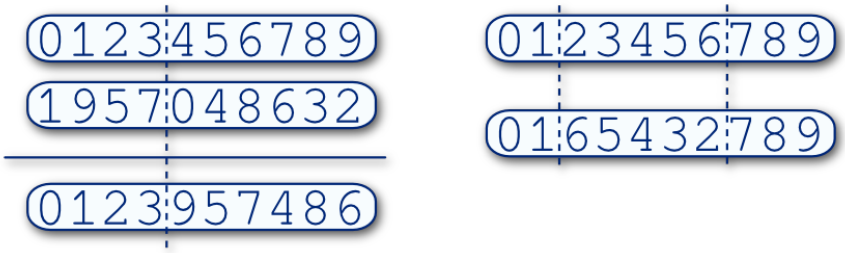

Figura 3.2: Operatori di 1-cut ordering crossover (sinistra) e mutazione 2-opt (destra)

l'aggiunta di due nuovi archi e l'inversione degli archi nella porzione di cammino interessata. Soluzioni non ammissibili possono essere generate se:

- almeno uno dei due archi da aggiungere non esiste
- non esiste l'inverso di un arco della porzione da invertire

Come per il crossover, la probabilità di mutazione è data da $p = pMutation \cdot \frac{\sigma_c}{c_w - \overline{c}} \in [0, pMutation] \subseteq [0, 1]$.

3.3.6 Miglioramento

I cromosomi *promettenti* sono soggetti ad una ulteriore fase di miglioramento. Un cromosoma C è considerato promettente se $cost(C) \leq \overline{c} - \sigma_c$, ovvero quando il costo della soluzione associata è al di sotto della media meno lo scarto quadratico medio, quindi particolarmente basso.

Poiché il miglioramento comporta comunque un costo computazionale, esso avviene con probabilità $p = pImprove \in [0, 1]$.

Il miglioramento avviene attraverso una ricerca locale, in particolare una *Hill Climbing* dove il vicinato è ottenuto considerando le mosse *2-opt*. È stato scelto questo tipo di ricerca per la sua velocità e per la qualità dei risultati osservati sperimentalmente [6, Ulder et al.].

3.3.7 Accettazione

Per mantenere diversificata la popolazione, cromosomi troppo simili a quelli già presenti non vengono accettati durante la rigenerazione della popolazione. La distanza tra due cromosomi A, B è espressa come $distance(A, B) = \frac{D_H(A,B)}{length(A)} \in [0, 1]$, dove $D_H(A, B) \in [0, length(A)]$ è la Distanza di Hamming tra A e B, e $length(A) = length(B)$.

Quando si cerca di aggiungere un nuovo cromosoma o, se nella popolazione $\nexists C.distance(o, C) < threshold$, il nuovo cromosoma viene aggiunto immediatamente. In caso contrario, viene aggiunto con probabilità $p = pAccept \in [0, 1]$.

3.3.8 Gestione della Popolazione

Viene utilizzata una popolazione a dimensione fissa con *Population Replacement*.

3.3.9 Criteri di arresto

Il ciclo sulle generazioni si arresta quando risulta verificata almeno una delle condizioni:

- l'algoritmo è in esecuzione da più di $maxTime$ secondi
- sono state compiute più di $maxIter$ iterazioni sulle generazioni
- nelle ultime $maxSlack$ generazioni non si è avuto alcun miglioramento

Il significato delle prime due condizioni è intuitivo. La terza è un tentativo di identificare una convergenza: se nelle ultime iterazioni non c'è stato miglioramento, si assume che l'algoritmo stia convergendo. Questo può essere causato da una convergenza prematura (in tal caso, probabilmente il risolutore è stato mal configurato), dal raggiungimento dell'ottimo globale o da una convergenza troppo lenta.

Capitolo 4

Analisi dei dati sperimentali

4.1 Generazione delle istanze

Un'istanza consiste in un grafo pesato di dimensione fissata. La generazione comprende tre fasi:

1. Generazione dei punti
2. Perturbazione dei punti
3. Creazione degli archi

Per la prima fase viene utilizzata una distribuzione uniforme sul piano. La perturbazione altera casualmente le coordinate dei punti usando una distribuzione normale di media e varianza fissate a priori come parametri. La creazione degli archi, infine, associa ad ogni arco una distanza utilizzando l'*Unfair distance*: $D_u(A,B) = \sqrt[p]{\sum (|A_x - B_x|^p + |A_y - B_y|^p)} + \mathcal{N}(\mu, \sigma^2)$. Se la distanza così ottenuta è superiore ad una data soglia, l'arco non viene aggiunto.

Si osserva che il primo addendo altro non è che la *Distanza di Minkowski*, la quale può essere vista come una generalizzazione della *Distanza Euclidea*: per

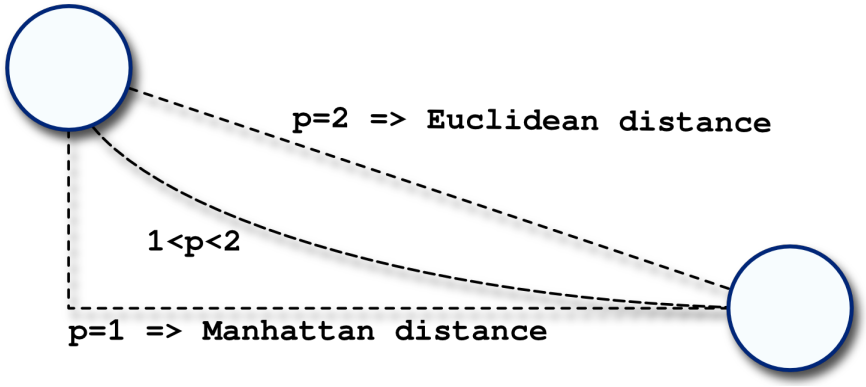

Figura 4.1: Esempi di Distanza di Minkowski

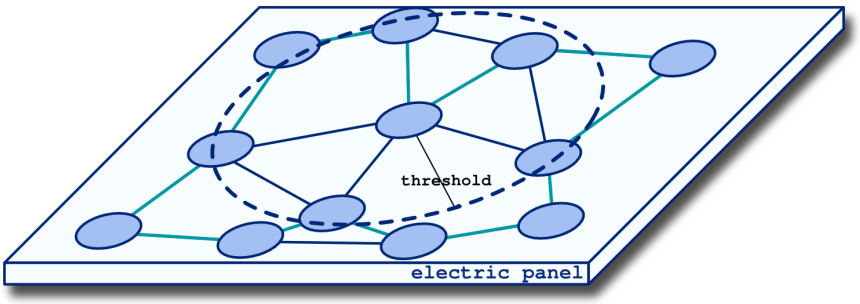

Figura 4.2: Esempio di istanza con soluzione ottima evidenziata

$p = 2$ si ottiene la distanza Euclidea, per $p = 1$ la distanza Manhattan, e così via, come esemplificato in fig. 4.1.

Il secondo addendo serve da perturbatore, rendendo la matrice delle distanze non simmetrica, mentre la soglia permette di scartare degli archi.

Questo criterio per l'assegnazione delle distanze è giustificato dall'obiettivo di complicare il problema per un risolutore basato su euristiche (da cui il nome *Unfair*). Una seconda motivazione è data dalla descrizione del problema nelle specifiche del progetto: il costo varia in funzione della macchina foratrice utilizzata. Una macchina sofisticata, per esempio, è in grado di muoversi in tutte le direzioni senza vincoli (distanza Euclidea, simmetrica, fortemente connesso), mentre una macchina meno efficiente potrebbe essere in grado di spostarsi solo orizzontalmente e verticalmente (distanza Manhattan), con tempi diversi per gli spostamenti verso destra o verso sinistra causati da latenze nelle componenti o altri fattori (matrice asimmetrica), e con un limite alla distanza percorribile in un singolo passo (grafo non necessariamente fortemente connesso).

La fig. 4.2 mostra un esempio di istanza ottenuta col procedimento descritto.

Nel codice fornito sono presenti altri metodi per la generazione e perturbazione di punti, così come altri algoritmi per il calcolo delle distanze. Questi strumenti sono stati utilizzati in fase di realizzazione e di test.

4.2 Raccolta dei dati

Al fine di avere un confronto equo tra i due risolutori, ad essi viene chiesto di risolvere le stesse istanze.

Le statistiche sono prodotte su un insieme di 160 istanze, spaziando da 10 a 80 nodi per istanza, 20 istanze per ciascuna dimensione. Oltre ai risultati ottenuti con i risolutori basati su CPLEX ed algoritmo genetico, vengono mostrati anche quelli ottenuti da un risolutore random, il quale crea un percorso permutando casualmente l'ordine dei nodi.

Il risolutore random viene introdotto per dare un'idea del guadagno ottenuto utilizzando una strategia piuttosto che affidarsi ad una soluzione casuale. Questo tipo di confronto è utile poiché, in alcuni contesti, risulta più efficiente generare soluzioni casuali molto velocemente piuttosto che produrne una sola ottima in tempo esponenziale.

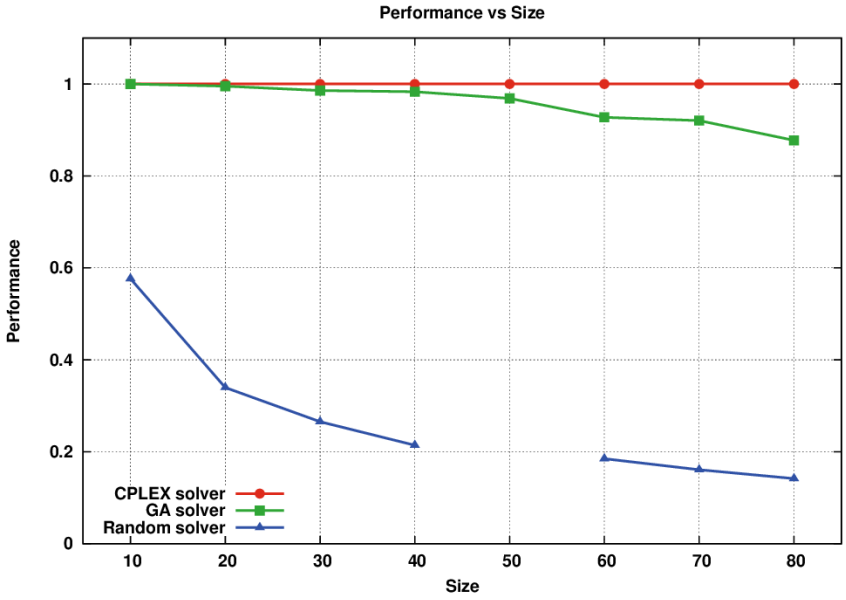

Figura 4.3: Qualità delle soluzioni ottenute

La fig. 4.3 mostra le prestazioni ottenute in funzione della dimensione dell'istanza. Per ciascuna istanza risolta, la sua prestazione è definita come il rapporto tra il costo della soluzione ottima ed il costo della soluzione calcolata. La prestazione mostrata è la media delle prestazioni delle istanze aventi la stessa dimensione. Poiché CPLEX produce sempre la soluzione ottima, la sua prestazione è 1. Si osserva inoltre che, poiché l'algoritmo genetico ha delle componenti casuali, la prestazione viene calcolata come il costo della soluzione ottima diviso la media dei costi delle soluzioni calcolate.

Il risolutore random non è stato in grado di generare soluzioni per nessuna istanza di dimensione 50: tra le permutazioni casuali considerate, nessuna è risultata ammissibile.

La fig. 4.4 mostra i tempi di esecuzione in funzione della dimensione delle istanze. Si osservano la crescita esponenziale del risolutore basato su CPLEX, l'andamento costante del risolutore random, e la limitazione superiore nel risolutore basato su algoritmo genetico (dovuta al parametro $maxTime$).

La fig. 4.5 mostra la percentuale (cumulativa) di istanze risolte entro un dato intervallo di tempo. Si osserva che il risolutore basato su algoritmo genetico è in grado di risolvere tutte le istanze entro $\approx 5s$, mentre il risolutore random fallisce sistematicamente: poiché il grafo non è strettamente connesso, permutazioni casuali possono produrre sequenze di nodi non connesse da archi.

La fig. 4.6 mostra in dettaglio le prestazioni ottenute dall'algoritmo genetico in funzione della dimensione dell'istanza, evidenziando le prestazioni minime e massime registrate e la deviazione standard dalla prestazione media (le prestazioni sono calcolate con lo stesso criterio visto per la fig. 4.3).

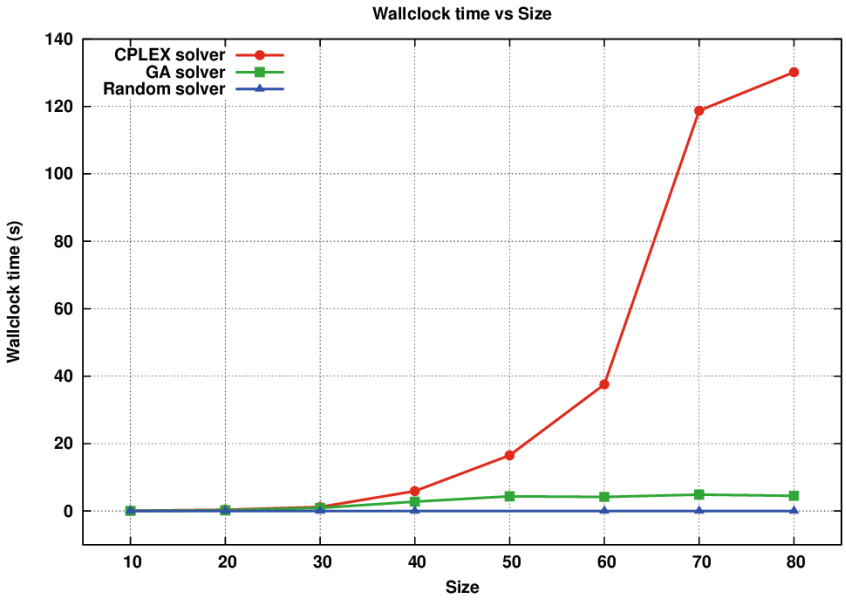

Figura 4.4: Tempo di calcolo

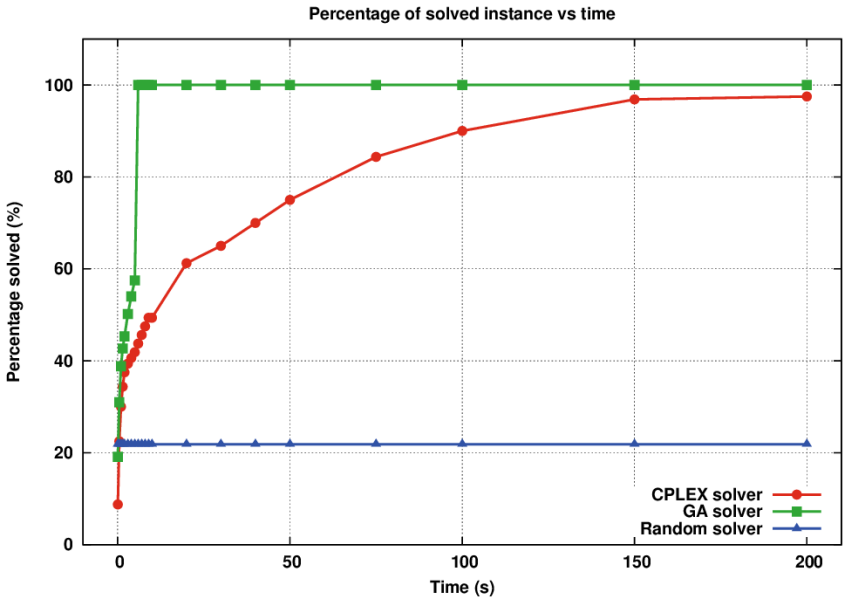

Figura 4.5: Percentuale di istanze risolte

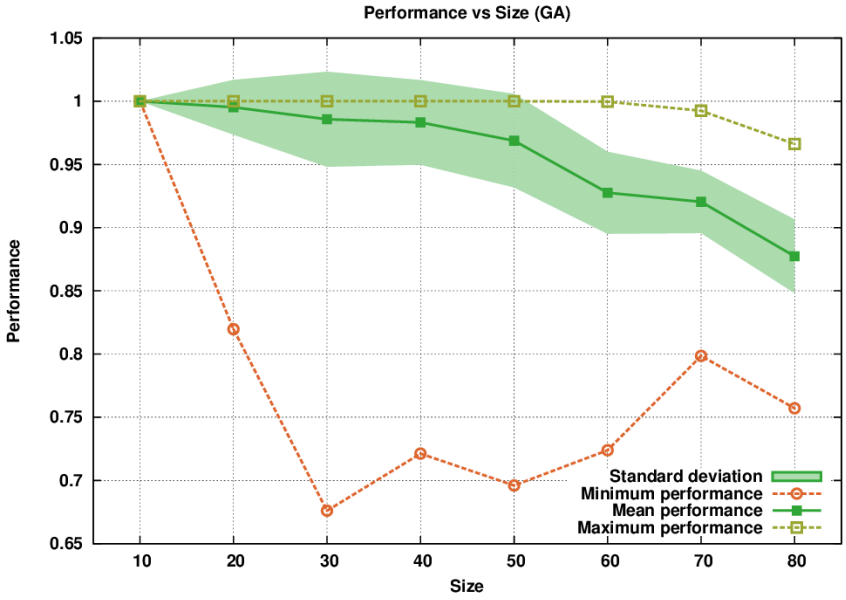

Figura 4.6: Qualità dell'algoritmo genetico

4.3 Confronto

I risolutori basati su PLI e su GA sono posti a confronto considerando:

tempo di sviluppo misura qualitativa della difficoltà di produzione del codice

qualità quanto le soluzioni sono prossime all'ottimalità

stabilità tendenza a produrre sistematicamente buone soluzioni piuttosto che alcune soluzioni molto buone ma altre pessime

tempo costo computazionale, tempo di esecuzione

4.3.1 Tempo di sviluppo

I risolutori hanno tempo di sviluppo comparabile. La PLI è generalmente più difficile da interpretare, ma ben documentata e molti problemi sono studiati e già tradotti in modelli relativamente semplici da implementare. Gli algoritmi genetici sono più immediati da comprendere, e generalmente ben documentati, ma esistono molte varianti e ciascuna delle componenti può essere implementata con caratteristiche diverse, rendendo necessari studi e calibrazioni che aumentano il tempo di sviluppo; d'altro canto, gli algoritmi genetici sono strumenti totalmente generici, le cui componenti possono essere riutilizzate facilmente, a differenza dei modelli di PLI.

4.3.2 Qualità

Il risolutore basato su PLI trova sempre la soluzione esatta (quando questa esiste), ottenendo quindi la massima qualità.

Dalle figg. 4.3 e 4.6 si osserva come l'algoritmo genetico proposto abbia una prestazione media superiore all'85%; in letteratura non sono presenti sufficienti articoli che trattano grafi non fortemente connessi per determinare se tale prestazione sia comparabile allo stato dell'arte o meno, ma resta intuitivamente una prestazione piuttosto alta. Si osserva inoltre che le prestazioni migliori registrate si mantengono entro il 5% dall'ottimo, mentre quelle peggiori promettono prestazioni al di sopra del 65%.

4.3.3 Stabilità

Il risolutore basato su PLI è perfettamente stabile, nel senso che restituisce sempre la soluzioni ottima (quando questa esiste).

L'algoritmo genetico non garantisce di trovare una soluzione, ma nei test eseguiti è sempre stato in grado di produrne una. La fig. 4.6 mostra inoltre che la qualità media è fortemente spostata verso l'alto, e la deviazione standard è $< 5\%$: questo significa che le soluzioni prodotte sono *tendenzialmente* di buona qualità, ed è improbabile osservare soluzioni significativamente al di sotto della qualità media; per istanze di dimensione ≤ 50 è molto probabile ottenere la soluzione ottima, segue che l'algoritmo genetico proposto mostra una buona stabilità.

4.3.4 Tempo

Il risolutore basato su PLI ha un andamento esponenziale, come era prevedibile. Va tuttavia osservato che i tempi assoluti sono al di sotto di un minuto per istanze di dimensione fino a 60, rendendo il problema gestibile per alcune applicazioni pratiche.

L'algoritmo genetico lavora in tempo costante (grazie all'utilizzo di $maxTime$). In particolare, nelle simulazioni eseguite, tutte le istanze sono state risolte in meno di 10 secondi.

Capitolo 5

Conclusioni

5.1 Interpretazione

In questa relazione sono stati presentati e messi a confronto due risolutori per il problema del Commesso Viaggiatore non simmetrico e non fortemente connesso, uno basato su Programmazione Lineare Intera, l'altro su Algoritmi Genetici Adattivi con Ricerca Locale.

Entrambi gli approcci hanno permesso di ottenere soluzioni di qualità significativa ed in tempi ragionevoli. Il risolutore implementato con CPLEX, in particolare, garantisce l'ottimalità al costo di un moderato carico computazionale, mentre l'algoritmo genetico lavora in tempo costante producendo soluzioni vicine all'ottimo.

Non ritengo sia possibile stabilire se uno dei risolutori sia strettamente migliore dell'altro in quanto, a parità di tempo di sviluppo e stabilità, uno eccelle nella qualità, l'altro nel tempo di calcolo, differenze che si fanno tanto più marcate quanto più elevata è la dimensione delle istanze da risolvere.

Va inoltre osservato che anche altre caratteristiche delle istanze hanno effetto su qualità e tempo di calcolo: all'aumentare della cardinalità dell'insieme degli archi nel grafo, aumenta anche il numero di variabili nel modello lineare, rendendo il calcolo con PLI meno efficiente, e diminuisce la probabilità di generare soluzioni inammissibili nell'algoritmo genetico, migliorandone la qualità. Viceversa, il risolutore basato su PLI è tanto più avvantaggiato quanto più il grafo è debolmente connesso.

In termini concreti, rispondendo alle specifiche del problema della macchina foratrice, se il numero di fori è limitato (≈ 80) e lo stesso schema di foratura deve essere rieseguito su più lastre, ritengo più conveniente utilizzare il modello PLI per calcolare, una sola volta, i movimenti del trapano da applicare su tutte le lastre. Altrimenti, se il numero di fori è molto alto, gli scemi di foratura sono troppi e si è disposti ad accettare un costo subottimo, l'algoritmo genetico è una buona scelta.

5.2 Margini di miglioramento

Sarebbe stato positivo introdurre ulteriori criteri per la generazione degli individui iniziali, ad esempio diversi tipi di greedy o euristiche costruttive in generale:

questo potrebbe introdurre *buone caratteristiche* diverse che andrebbero ad arricchire la popolazione, migliorando le prestazioni. Questo miglioramento non è stato realizzato per motivi di tempo.

È noto che la rigenerazione della popolazione può essere resa parallela [7, Mühlenbein]. Tuttavia la programmazione concorrente con pthread è più complessa, richiede maggiore tempo di sviluppo ed i miglioramenti non sono concettualmente profondi.

Analogamente si possono utilizzare algoritmi genetici distribuiti (ad es. con MPI) [8, Braun], ma anche i questo caso i miglioramenti sono di tipo architetturale e non concettuale.

Una significativa fonte di miglioramento consiste nello sperimentare diversi criteri di selezione, crossover, mutation, gestione della popolazione, miglioramento, per vedere quali danno effettivamente il risultato migliore. Una ricerca esaustiva richiederebbe però troppo tempo. Sono stati scelti i criteri che, secondo la letteratura, portano a soluzioni migliori (o comunque quelli meglio documentati).

Il processo di calibrazione può essere ulteriormente raffinato. Due miglioramenti ovvi sono una ricerca più approfondita nello spazio dei parametri e la valutazione di modelli diversi oltre al CART. Una miglioria concettualmente più profonda consiste nel cercare di sfruttare maggiormente la struttura dell'istanza da risolvere, ad esempio discriminando su quanto fortemente (o debolmente) il grafo sia connesso.

Appendice A

Calibrazione dell'Algoritmo Genetico

Per la calibrazione dell'algoritmo genetico è stato generato un insieme di istanze (*calibration set*), distinto rispetto a quello utilizzato per le statistiche. I parametri rispetto ai quali l'algoritmo è calibrato sono: $Crossover$, $Mutation$, $Threshold$, $Accept$, $Improve$ e $PopSize$

Idealmente, un test esaustivo deve coprire ogni possibile combinazione dei parametri. Lo spazio di ricerca esplode quindi in maniera combinatoria.

La fig. A.1 (sinistra) mostra un esempio di spazio di ricerca esaustivo su tre parametri. Per motivi di tempo non è stato possibile eseguire la calibrazione esaustiva. Viene quindi proposta una calibrazione ridotta (esemplificata in fig. A.1, destra) nella quale viene eseguito un numero limitato di test. I valori dei parametri ad ogni esecuzione sono generati casualmente tramite una distribuzione uniforme nello spazio di ricerca. Lo spazio delle istanze diventa così più rarefatto, perdendo tanta più precisione quando più significativa è la riduzione del numero di test.

Si osserva che, all'aumentare del numero di istanze nella calibrazione ridotta, questa tende a convergere alla calibrazione esaustiva.

I valori di maxIter, maxSlack, maxTime sono fissati. La calibrazione è svolta come segue:

1. si fissa un insieme di dimensioni di istanze (50, 60, 70, 80)
2. per ogni dimensione si generano 10 istanze
3. per ogni istanza, vengono generate 15 configurazioni in maniera uniforme
4. per ogni configurazione, la rispettiva istanza viene risolta 10 volte con quella configurazione

La ripetizione dell'ultimo punto è necessaria in quanto l'algoritmo genetico ha alcune componenti casuali.

La calibrazione proposta consiste nello sfruttare l'informazione sulla dimensione dell'istanza da risolvere per determinare i valori ottimali dei parametri per l'algoritmo genetico. Dal punto di vista analitico, si cerca di spiegare la *prestazione* dell'algoritmo su un'istanza in funzione della dimensione dell'istanza e dei parametri utilizzati. Il risultato è una relazione che si presenta come

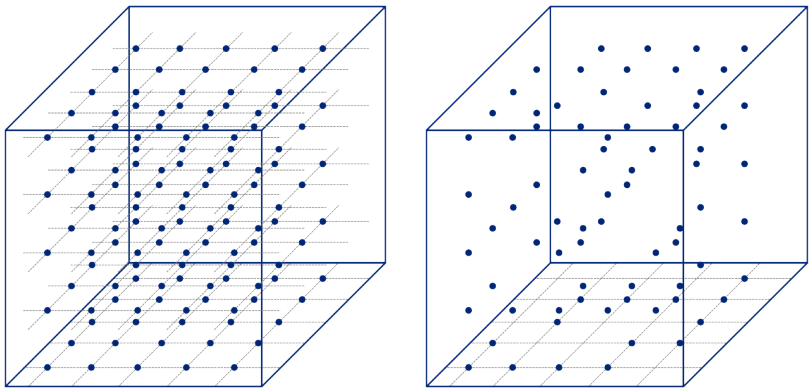

Figura A.1: Calibrazione esaustiva (sinistra) e ridotta (destra)

Parametro	Coefficiente	Pr(>|t|)
Size	-1.707e-03	< 2e-16
Crossover	1.217e-02	0.00692
Mutation	5.294e-02	< 2e-16
Threshold	-2.943e-02	6.77e-12
Accept	-1.774e-02	1.22e-05
Improvement	5.311e-04	0.89831
PopSize	-8.599e-05	0.30942

Tabella A.1: Influenza marginale dei parametri

funzione $\mathbb{R}^5 \times \mathbb{N}^2 \mapsto \mathbb{R}$, che può essere utilizzata per determinare quale combinazione di parametri massimizza le prestazioni (una volta fissata la dimensione dell'istanza).

Per ogni soluzione i, la prestazione è calcolata come $\frac{min(c_{istance})}{c_i}$, ovvero il minimo dei costi ottenuti per l'istanza risolta dalla soluzione i diviso il costo della soluzione i (prestazione relativa).

A.1 Effetto marginale dei parametri

Per avere un'idea dell'effetto marginale dei parametri sulla prestazione, viene costruito un modello lineare col criterio dei minimi quadrati. La tab. A.1 mostra i coefficienti ottenuti per ciascun parametro, assieme alla probabilità di osservare un *t-value* maggiore sotto l'ipotesi nulla. Quest'ultimo dato è una misura della *significatività* del parametro: quanto più esso è prossimo allo zero, tanto più la sua influenza sulla performance è significativa.

I parametri $Size, Crossover, Mutation, Threshold$ sono risultati essere fortemente significativi, mentre $Improvement, PopSize$ sono risultati scarsamente significativi.

La scarsa significatività di $Improvement$ è motivata dal fatto che esso è strettamente correlato ad *Accept*, mentre per $PopSize$ la motivazione risiede nel fatto che la dimensione della popolazione effettivamente non influenza significativamente la qualità dell'algoritmo, risultato confermato in letteratura.

Si osserva inoltre che $Crossover, Mutation, Improvement$ hanno coefficiente positivo: l'aumento delle probabilità di crossover, mutazione e miglioramento tramite ricerca locale porta *tendenzialmente* ad un miglioramento delle prestazioni, come era facilmente intuibile.

Al contrario $Size, Threshold$, *Accept* hanno un'influenza negativa: istanze di dimensioni maggiori sono più difficili da risolvere, soglie molto alte comportano il rifiuto di molti nuovi individui, ed un'alta probabilità di accettare cromosomi duplicati comporta scarsa diversificazione nella popolazione.

A.2 Distribuzione delle prestazioni

La fig. A.2 mostra la distribuzione delle prestazioni relative durante la calibrazione. Dalla figura si osserva come i quartili della distribuzione siano spostati verso l'alto. Si osserva inoltre che la prestazione minima è attorno al 70%. Questi risultati suggeriscono che l'algoritmo proposto è *stabile*, nel senso che la qualità delle soluzioni fornite si mantiene alta anche in caso di configurazioni non ottimali, verosimilmente grazie all'utilizzo di parametri adattivi e della ricerca locale.

A.3 Relazione tra parametri e prestazioni

La relazione tra i parametri e la prestazione è ottenuta attraverso l'uso di un CART (*Classification And Regression Tree*). Idealmente, sono necessari studi preliminari per la scelta del modello, dei suoi parametri di configurazione, dello spazio campione e della suddivisione in *training* e *validation set*. La scelta è stata limitata al CART come strumento *black box*: la libreria *tree* del software *R*

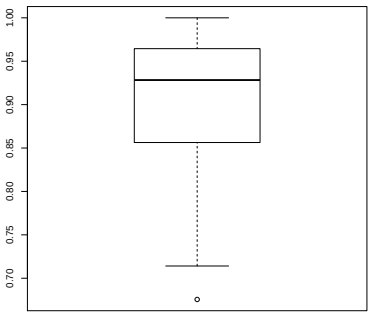

Figura A.2: Distribuzione delle prestazioni

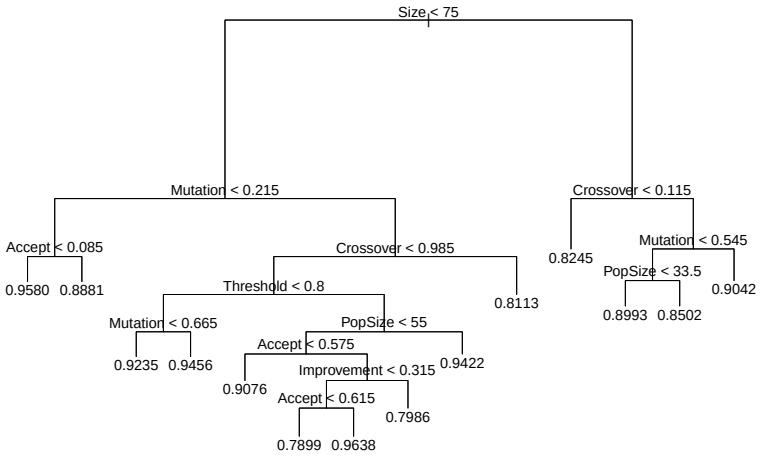

Figura A.3: Albero CART

produce un CART tramite *k-fold cross validation* (suddividendo autonomamente le istanze in insiemi di allenamento e verifica), e determina automaticamente i propri parametri di configurazione (come la potatura dell'albero). La scelta del CART è inoltre motivata dal fatto che gli alberi prodotti sono più semplici da interpretare rispetto ad altri modelli (ad es. reti neurali o GAM), e richiedono inoltre un costo computazionale minimo per essere generati. Gli svantaggi riguardano il tipo di funzione prodotta (*funzione "a gradini"*), generalmente meno precisa delle relazioni ricavate da altri modelli, ed il fatto che un eventuale aggiornamento con nuove osservazioni comporta il ricalcolo completo del CART stesso. La fig. A.3 mostra il CART ottenuto durante la calibrazione.

A.4 Funzione di calibrazione

Sfruttando le informazioni ottenute dal CART e dal modello lineare è stato creato un risolutore *proxy* che determina i parametri ottimali in base alla dimensione dell'istanza e richiama il risolutore genetico proposto. Per determinare i parametri viene utilizzata la seguente strategia:

1. si inizializza un insieme di vincoli $C = \emptyset$
2. per ogni parametro p_i che rappresenta una probabilità, si aggiungono a C i vincoli $p_i \geq 0, p_i \leq 1$; per la dimensione della popolazione si aggiunge un vincolo *ragionevole*, ad esempio $PopSize \geq 16, PopSize \leq 64$
3. si parte dalla radice dell'albero
4. se il nodo corrente discrimina sulla dimensione dell'istanza, si segue il ramo che corrisponde alla dimensione dell'instanza in input (ritorno a 4, considerando il nodo figlio)
5. se il nodo corrente discrimina su di un parametro, si percorre l'arco che porta alla foglia con prestazione più alta, e si aggiunge a C il vincolo che soddisfa l'arco scelto (ritorno a 4, considerando il nodo figlio)
6. se il nodo è una foglia, la scansione termina e da C si ottengono delle limitazioni inferiori e superiori sui parametri
7. per ciascun parametro, se il suo coefficiente nel modello lineare è positivo viene scelto il suo limite superiore, altrimenti viene scelto il suo limite inferiore

Il risultato di questa applicazione è una funzione nella forma $\mathbb{N} \mapsto \mathbb{R}^5 \times \mathbb{N}$, ovvero una funzione che data la dimensione di un'istanza, restituisce la configurazione ottimale dei sei parametri (cinque reali ed uno naturale) dell'algoritmo genetico.

Appendice B

Grafo simmetrico e fortemente connesso

Nella letteratura il problema del TSP su grafo simmetrico e fortemente connesso è ampliamente trattato. L'assunzione del grafo fortemente connesso rappresenta una semplificazione importante sulla natura del problema, rendendo ogni permutazione una soluzione ammissibile.

Quest'appendice propone un caso di studio su grafi simmetrici e fortemente connessi, generati con le modalità descritte nella sez. 4.1. I grafici presentati sono costruiti in maniera analoga a quelli visti nella sez. 4.2.

In questo caso di studio sono state considerate solo le istanze più *difficili*, quelle aventi dimensioni $60, 70, 80$. Per ciascuna dimensione sono state generate 5 diverse istanze. Ogni istanza è stata risolta una sola volta col risolutore basato su PLI, e 10 volte con il risolutore basato su algoritmo genetico.

La fig. B.1 conferma i risultati presentati nella sez. 4.2: l'algoritmo genetico lavora in tempo costante, contro il tempo esponenziale impiegato dal modello PLI.

La fig. B.2 mostra le prestazioni media, minima e massima dell'algoritmo genetico, evidenziando la deviazione standard dalla media. Si osserva che la qualità media delle soluzioni è entro lo 0.5% dall'ottimo, con la prestazione massima pari a 1. Questo risultato indica che l'algoritmo genetico è stato in grado di ottenere la soluzione ottima almeno una volta per ciascuna delle istanze

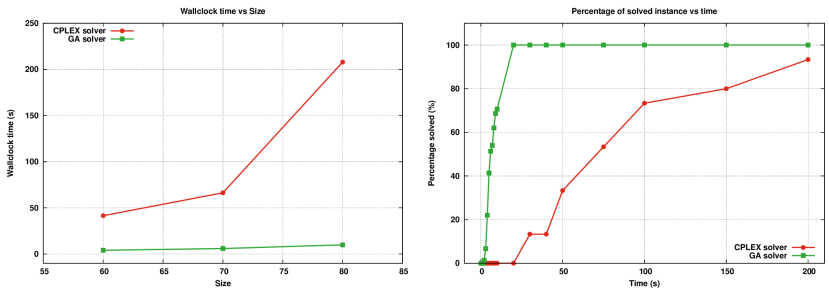

Figura B.1: Tempo di calcolo (sinistra) e percentuale di istanze risolte (destra)

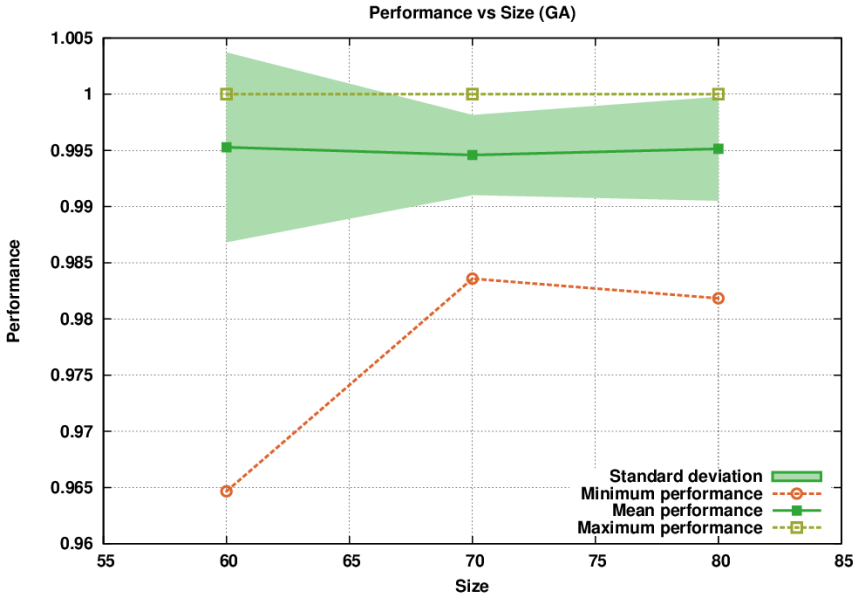

Figura B.2: Qualità dell'algoritmo genetico

proposte. Si osserva inoltre che la prestazione peggiore è del 96.5%, ovvero che la soluzione peggiore ottenuta è entro il 3.5% dall'ottimo.

Questi risultati preliminari riguardano un insieme ristretto di istanze, ma sono incoraggianti in quanto comparabili con lo *stato dell'arte*, nel quale le soluzioni ottenute sono mediamente entro il 3% dall'ottimo: in [5, Noraini et al.], ad esempio, vengono mostrati risultati mediamente entro lo 0.9% dall'ottimo, su un insieme di sole 8 istanze.

Bibliografia

[1] A. Laoufi, S. Hadjeri, and A. Hazzab. Adaptive probabilities of crossover and mutation in genetic algorithms for power economic dispatch.

[2] Otman Abdoun, Jaafar Abouchabaka, and Chakir Tajani. Analyzing the performance of mutation operators to solve the travelling salesman problem. *arXiv preprint arXiv:1203.3099*, 2012.

[3] Sangit Chatterjee, Cecilia Carrera, and Lucy A Lynch. Genetic algorithms and traveling salesman problems. *European journal of operational research*, 93(3):490–510, 1996.

[4] RO Oladele and JS Sadiku. Genetic algorithm performance with different selection methods in solving multi-objective network design problem. *International Journal of Computer Applications*, 70(12):5–9, 2013.

[5] Mohd Razali Noraini and John Geraghty. Genetic algorithm performance with different selection strategies in solving tsp. 2011.

[6] NicoL.J. Ulder, EmileH.L. Aarts, Hans-Jürgen Bandelt, PeterJ.M. van Laarhoven, and Erwin Pesch. Genetic local search algorithms for the traveling salesman problem. 496:109–116, 1991.

[7] H. Mühlenbein. Parallel genetic algorithms, population genetics and combinatorial optimization. 565:398–406, 1991.

[8] Heinrich Braun. On solving travelling salesman problems by genetic algorithms. 496:129–133, 1991.

[9] Patrick R. Nicolas. *Scala for Machine Learning.* Packt Publishing, 2014.