Relazione progetto Intelligenza Artificiale

Studente:

Fiorito Aldo 1000038099

Docente:

Pavone Francesco Mario

Corso di laurea magistrale LM-18

Risoluzione problema “MWVC”

CAPITOLO 1: Introduzione al problema

L'obiettivo principale di questo progetto è comprendere e risolvere il problema noto come "copertura dei vertici di peso minimo" attraverso l'applicazione dell'algoritmo genetico.

Il problema della "Minimum Weight Vertex Cover" (MWVC), noto anche come "copertura minima dei vertici", è uno dei problemi chiave nell'ambito dei grafi, con rilevanti applicazioni nella vita reale. L'approccio qui adottato si basa sull'utilizzo dell'algoritmo genetico, ispirato dall'osservazione del comportamento genetico negli organismi viventi.

L'obiettivo primario è identificare soluzioni valide e ottimali per il problema MWVC. Questo, viene raggiunto attraverso la manipolazione di una popolazione di soluzioni, con l'obiettivo di migliorarle nel tempo. L'approccio genetico prevede la creazione di nuove soluzioni mediante la combinazione e la modifica di soluzioni esistenti. Questo processo di evoluzione continua fino a ottenere risultati ottimali o altamente competitivi.

I grafi utilizzati per risolvere questo problema sono rappresentati in file con estensione ".txt". Ogni file contiene una serie di esempi, ognuno con un numero specifico di nodi (n) e archi (m). Durante l'esecuzione dell'algoritmo genetico, vengono sperimentati diversi parametri e tecniche al fine di ottimizzare le prestazioni e ottenere i migliori risultati possibili.

1.1 Definizione e dettagli del problema

Il problema del Minimum Weight Vertex Cover (MWVC), o "Copertura dei Vertici di Peso Minimo" in italiano, può essere definito in termini matematici come segue:

"Dato un grafo G = (V, E), dove V è l'insieme dei vertici e E è l'insieme degli archi, il MWVC consiste nel trovare un sottoinsieme S' di vertici in modo che ogni arco in E sia incidente a almeno uno dei vertici in S', e la somma dei pesi dei vertici in S' sia minimizzata"

Formalmente, il problema può essere definito come:

- S' è un sottoinsieme di V, ovvero S' ⊆ V.

- Per ogni arco (u, v) in E, almeno uno dei vertici u o v deve appartenere a S', cioè u ∈ S' o v ∈ S'.

- Si assegna un peso w(v) a ciascun vertice v ∈ V.

- L'obiettivo è minimizzare la somma dei pesi dei vertici in S.

$$\textbf{minimize} \quad \omega(S) = \sum_{v \in S} \omega(v),$$

In altre parole, stiamo cercando il sottoinsieme più piccolo di vertici (S') in modo che ogni arco nel grafo sia coperto almeno da un vertice in questo sottoinsieme e la somma dei pesi dei vertici in S' sia la più piccola possibile.

Il MWVC è un problema di ottimizzazione combinatoria e una soluzione valida S, chiamata "Vertex Cover" o "Copertura", risulta tale se ogni nodo in G ha almeno un "endpoint" in S.

1.2 Approcci per la risoluzione del problema

I principali approcci per la risoluzione di questo problema np hard sono diversi.

1. Tabu Search (TS);

2. Genetic Algorithm (GA);

3. Iterated Local Search (ILS).

Tra i 3 approcci verrà implementato e utilizzato l'algoritmo genetic GA.

L'algoritmo Genetic si discosta dagli algoritmi convenzionali per la sua randomicità e combinazione delle soluzioni. Esso crea una popolazione di punti in ciascuna iterazione, dove il punto migliore all'interno di questa popolazione rappresenta una soluzione ottimale. A differenza degli algoritmi classici, che selezionano il punto successivo nella sequenza utilizzando calcoli deterministici, l'algoritmo Genetic sfrutta una base casuale per formare la prossima popolazione a partire dalla popolazione corrente, controllando sempre che le nuove soluzioni soddisfino i vincoli stabiliti.

In ogni passo dell'algoritmo, vengono casualmente selezionati individui dalla popolazione corrente, e questi individui vengono utilizzati per generare nuove soluzioni che costituiranno la generazione successiva. Questo processo rappresenta un'evoluzione delle soluzioni rispetto alla generazione precedente.

L'algoritmo utilizza tre operatori principali e uno "bonus", per generare le nuove soluzioni:

1. **Operatore di Selezione:** Queste tecniche operano all'interno della generazione corrente, selezionando individui che siano dei genitori per la creazione delle generazioni successive. I selezionati sono utilizzati come base per la creazione di nuove soluzioni nella generazione successiva.

2. **Operatore di Crossover:** Le regole di crossover intervengono modificando i genitori selezionati al fine di creare uno o più figli, ognuno dei quali rappresenta una possibile soluzione per le generazioni future. Questo processo di combinazione dei genitori permette di mescolare le caratteristiche delle soluzioni padri per generare nuove e potenzialmente migliori soluzioni.

3. **Operatore di Mutazione:** Le regole di mutazione intervengono in modo casuale sugli individui della popolazione corrente. Questo processo di mutazione introduce piccole modifiche casuali nelle soluzioni esistenti, contribuendo a esplorare nuove aree dello spazio delle soluzioni. L'obiettivo è di trovare soluzioni ottimali o di migliorare soluzioni esistenti attraverso piccole variazioni casuali.

4. **Bonus Evolution:** Questa non è una vera e propria regola, ma un approccio evolutivo sulla scelta della successiva popolazione da esaminare e da migliorare. Per questo task è stata applicata la tecnica **μ + λ Evolution,** in cui vengono portati avanti i padri selezionati in precedenza e i figli mutati dai precedenti operatori.

1.3 Istanza del problema

- Input

  Il problema affrontato in questo lavoro coinvolge istanze con N nodi e M archi, in cui ciascun elemento della popolazione è rappresentato da una stringa binaria di lunghezza N. In questa rappresentazione, il bit "1" indica la selezione del nodo i-esimo all'interno del grafo, mentre il bit "0" indica l'esclusione del nodo i-esimo dalla soluzione in considerazione.

  In un estratto d'esempio possiamo trovare nelle righe:

  1. Il numero di nodi
  2. I nodi indicati come pesi
  3. Matrice di incidenza o degli edge, dove la presenza del bit 1 indica un collegamento tra un nodo X e Y

- Valutazione di un padre/figlio

  Al fine di poter utilizzare il GA è necessario definire un'ultima cosa. La funzione di valutazione, anche chiamata FITNESS. La scelta è la somma dei pesi dei nodi i-esimi, all'interno della soluzione, con bit pari a "1".

Per esempio, data una soluzione [1,0,0,1] ed un vettore di pesi [3,2,2,1], la funzione sarà pari alla somma 3+1 in cui 3 e 1 rappresentano relativamente i valori i-esimi, all'interno della soluzione, in cui troviamo il bit "1" e quindi accesi.

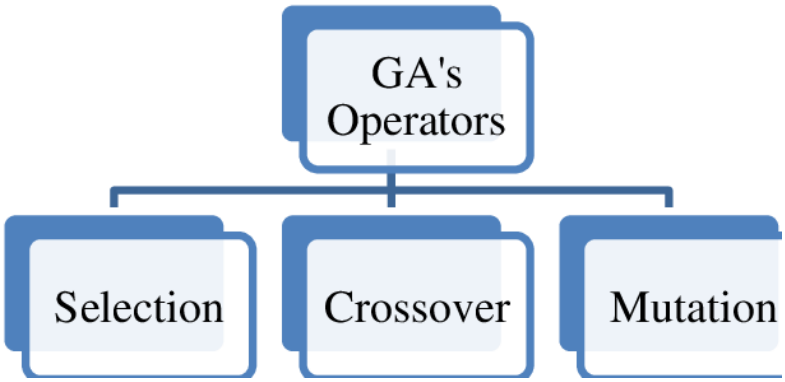
- Validità di una soluzione

  Affinché una soluzione sia valida deve rispettare necessariamente due condizioni

  1. Essere una Cover set, ovvero Un "cover set" in un grafo è un insieme di nodi (o vertici) che, quando uniti, coprono tutti gli archi del grafo. In altre parole, un cover set è un insieme di nodi tale che ogni arco del grafo ha almeno un'estremità o anche chiamato endpoint appartenente all'insieme.
  2. Essendo un problema di minimizzazione, la cover set deve essere quella di costo minore tra tutte le Cover set del grafo.

1.4 Operatori di GA

Come accettano prima, andremo ad elencare e dettagliare gli operatori implementati per il corretto approccio dell'algoritmo genetico



- **Selection operator**

  La fase di "Selezione", conosciuta anche come fase di "Selection" in inglese, rappresenta il momento in cui un algoritmo Genetico estrae genomi da una popolazione per la generazione di successivi ceppi, sfruttando gli operatori di crossover.

Nel seguente soluzione sono state implementate le tecniche di:

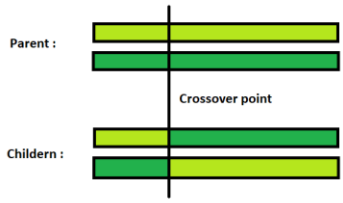
1) Roulette Wheel Selection: essa sceglie casualmente due genitori dalla generazione precedente. La scelta avviene attraverso la probabilità, direttamente in proporzione alla frazione dei valori di fitness.

2) Tournament Selection: questa modalità estrae casualmente due genitori attraverso un torneo di dimensione K, solitamente in questo torneo a vincere solo le soluzioni/popolazioni che hanno minor fitness.

3) Random Selection: questa tecnica seleziona casualmente due genitori dalla popolazione senza considerare alcuna preferenza specifica.

- **Crossover operator**

Il Crossover, è una funzione utilizzata per combinare le informazioni genetiche di due genitori, appartenenti alla stessa popolazione, per generare nuove soluzioni figlie.



Anche qui sono presenti diverse strategie per variare la combinazione, esse sono stabilite in base ai punti di crossing.

1) **1-Point Crossover,** secondo cui un punto di combinazione random combiniamo la prima parte di un genitore con l'altra del restante genitore e cosi via.

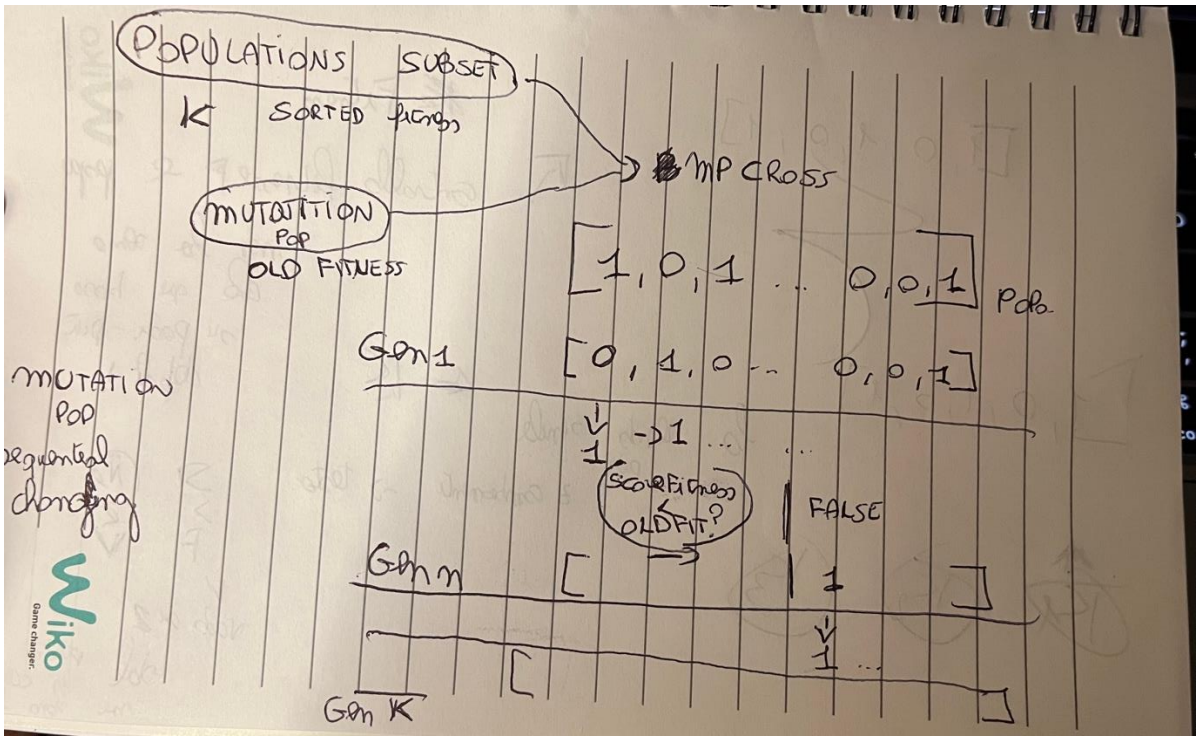
2) **Multi-Point Crossover**, evoluzione del caso baso in cui ripetiamo il concetto più volte secondo un numero arbitrario

3**) Uniform Crossover**, secondo un random, scambiamo i bit da genitore a genitore secondo una probabilità uniforme.

4) **Sequential MultiParent Crossover** (bonus), in questo operatore custom è stato implementata una logica personalizzata dell'operatore crossover. Il ragionamento prevedere il cambio sequenziale dei bit della popolazione in base ai migliori K padri Questi possiedono i migliori geni in quanto, tra tutti i genitori sono i primi, essendo ordinati per fitness ottenuta.
In maniera sequenziale, scorrendo i padri, si proverà a innestare il gene del padre in quello del figlio.

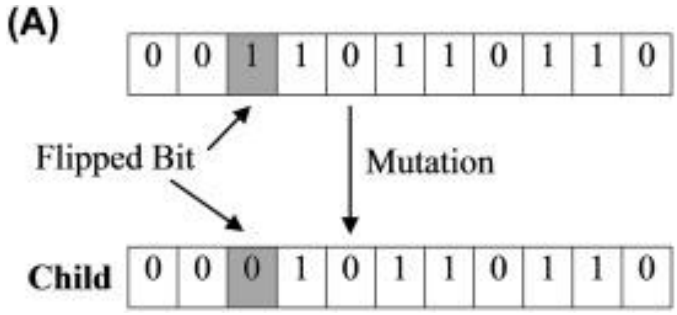
Questo avviene quando vi è una differenza di bit tra le due popolazioni, può l'inserimento del gene padre nel figlio.

Il gene viene considerato valido quando la nuova fitness ottenuta con il nuovo gene, sarà minore della fitness precedente, altrimenti si passerà al padre successivo ripristinando il gene del figlio.
Successivamente, si continuerà dal punto di interruzione, proseguendo a controllare i successivi geni con i restanti padri.
La scelta e l'ordinamento dei genitori da scansionare è importante, per l'implementazione si è scelto di partire dal miglior padre e andare verso i non ottimi.
Se nessuno dei padri apporterà dei geni validi allora la rimanente parte genetica del figlio rimarrà la stessa.



- **Mutation operator**

  La Mutazione viene applicata solo quando un valore casuale, generato ad ogni iterazione, è inferiore a una variabile denominata probabilità di mutazione (PM). Questo operatore di Mutazione genera due nuove soluzioni modificando casualmente bit all'interno delle soluzioni che sono state precedentemente generate tramite crossover.

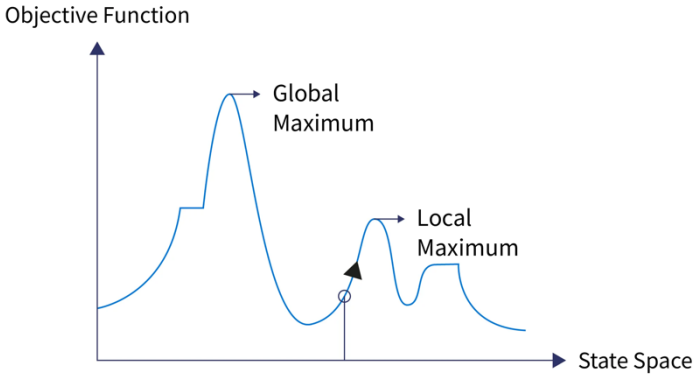


Evolution Step
Una volta apportate queste modifiche, le soluzioni risultanti sono valutate per verificarne la validità attraverso una funzione di validazione.
Se una soluzione supera con successo questa verifica e si dimostra essere valida, viene inclusa nella popolazione per la successiva iterazione dell'algoritmo, ed è qui che avviene il passo di evoluzione, in quanto la popolazione iniziale verrà unità, attraverso l'approccio **μ + λ.**
Ordinando per lo score ottenuto dalla funzione di fitness e selezionando uno slice della popolazione è possibile prendere il meglio delle parti

a. **Alternative strategy: LocalSearch**

   Una ulteriore strategia per migliorare le soluzioni esistenti e/o riuscire ad uscire da un possibile punto di minimo locale è quella della local search. In questa versione è stato scelto di combinare la strategia di simulated annealing con la struttura di supporto del Tabu search, per ampliare lo spazio delle possibili soluzioni da visitare. L'idea di base è quella di generare delle perturbazioni alle soluzioni uscente dal genetic algoritm, in maniera analoga al passo di mutazione spiegato prima.
   Ricordiamo che siamo interessati a decrementare la fitness, per questo la scelta dello SA.
   L'algortimo di search è controllato da 3 parametri, uno di max_iteration, uno di temperature e uno di cooling. Questi ultimi due regolano la scelta di accettare o meno una soluzione modificata. A volte è davvero necessario riuscire ad accettare soluzioni ben diverse dalla esistente per provare ad

esplorare uno spazio bel diverso da quello analizzato prima. L'accettazione di questa pertubazione, seppur con la premessa di essere valida, viene gestita secondo la temperatura iniziale. Questa, iterazione dopo iterazione, dimuisce a seconda del parametro di cooling (impostato a 0.95), quindi la soluzione nuova avrà probabilità di essere accettata diminuire.
A completare il tutto, viene popolata una struttura di soluzioni già esplorate chiamata tabu search, nel momento di generazione della nuova pertubazione andiamo prima a controllare che questa non si trovi nelle ultime 10 soluzioni usate.



A one-dimensional state-space landscape in which elevation corresponds to the objective function

CAPITOLO 2 – Applicazione algoritmo ed esecuzione

Una volta presi in input i dati necessari per risolvere il problema, l'esecuzione dell'algoritmo parte. Per le diverse esecuzioni su diverse istanze dei problemi forniti in file ".txt", abbiamo utilizzato i seguenti valori come parametri.

NB, Durante l'esecuzione del programma sono stati inserite delle istruzioni di **copy.deepcopy()** necessarie per intervenire durante l'operazione di assegnamento, rispetto al funzionamento di python. Infatti, esso, con l'operatore "=" genera un puntatore all'oggetto di destra. Il nostro intento è quello di copiare l'intera struttura come ex novo ogni iterazione e assegnamento.

Limite di Valutazione della Fitness: Abbiamo impostato un limite utilizzando una variabile contatore chiamata "FE" (Fitness Evaluation), che rappresenta il numero massimo di volte che la funzione fitness può essere valutata. Quando il contatore FE raggiunge o supera il valore limite, l'esecuzione viene interrotta. Questo valore, da specifiche è impostato a *2*(10**4)*.

Convergenza anticipata fitness: Per fare in modo che l'algoritmo converga più velocemente è possibile impostare un valore massimo di "replicabilità" della soluzione migliore. Questo farà in modo di bloccare l'algoritmo quando la migliore fitness trovata sarà uguale per le successive X iterazioni. Questo X è possibile trattarlo come una sorta di hyperparametro. È consigliabile aumentarne il valore quando si analizzano istanze con un grande numero di nodi e di archi.

Probabilità operatori: Per gli operatori di *crossover* e *mutation* è stato scelto di applicare una probabilità di "evento" per rendere l'esecuzione ancora più genetica.

Questa probabilità è possibile modificarla attraverso il file *settings.py*

Popolazione: per la scelta della popolazione iniziale, è stato impostato un valore di 100, così come per le successive iterazioni di evoluzioni. Questa è prodotta in maniera casuale una volta creato l'oggetto *Mutation* e inizializzato il problema dall'input. Da qui possiamo fin da subito generare soluzioni non ottime ma candidate ad essere ottime attraverso la validazione. Infatti, una prima scrematura è possibile farla controllando se il singolo elemento è un cover set per il grafo.

Selection: Per selezionare gli individui da utilizzare nelle operazioni di crossover, abbiamo calcolato il valore di fitness per ciascun individuo come il reciproco della sua fitness iniziale. In questo modo, abbiamo selezionato valori inversamente proporzionali alla fitness. Selezione tramite Roulette Wheel:
Per la selezione dei genitori da utilizzare nel crossover, si è utilizzato abbiamo utilizzato un algoritmo noto come "Tournament Selection", che ha dimostrato di produrre risultati migliori rispetto ad altri approcci.

Crossover: Abbiamo eseguito l'operazione di crossover solo se un numero generato casualmente era minore o uguale a un valore, impostato a 0.9. Abbiamo utilizzato il metodo del "1-point crossover" preceduto dal sequential multi parent crossover, con l'inclusione di 3 padri (K) per determinate quanti padri considerare. Mentre, per altre è stato scelto di utilizzare semplicemente il multipoint crossover. L'uso del crossover uniforme non ha portato sempre grandi risultati.

Mutation: Durante la fase di mutazione, abbiamo ottenuto risultati migliori modificando due bit (differenti) in modo casuale. La mutazione è stata eseguita solo se un numero generato casualmente era minore o uguale a un valore, impostato a 0.1. Prima di essere inserite nella popolazione, tutte le soluzioni mutate sono state validate.

LocalSearch: Durante l'esecuzione dell'algoritmo genetic è possibile saltare o meno la parte di mutazione e rimpiazzare quest'ultima con la local search. Anche qui, a seconda di un flag impostabile dall'utente, possiamo decidere se eseguire o meno questo modulo. Per quanto riguarda i parametri dell'oggeto LocalSearch si è scelto di utilizzare un max_iter di 5000, un temperature di 2500 e un cooling rate del 5% di decremento.

Valori dei Parametri: Per le diverse esecuzioni su diverse istanze dei problemi forniti in file con estensione ".txt", abbiamo utilizzato i seguenti valori come parametri.

CAPITOLO 3: Pseudocodici

Fitness

Fitness (Popolazione, Pesi):

```
score ← 0;
Per ogni i nell'intervallo da 0 a len(Popolazione) – 1 do:
        if Popolazione[i] == 1 do :
        score = score + Pesi [i];
        return score;
```

Check soluzione valida (inverse logic)

isValid(Popolazione,Vertici):

```
Per ogni i nell'intervallo da 0 a len(Popolazione) – 1 do:
   Se elementoPopolazione_i è uguale a 0 do:
      tmp ← Vertici[i]
      tmp_neigh  ← Vicini(elementoVertice_i)

      Per ogni j nell'intervallo da 0 a len(tmp_neigh) – 1: # loop tra i suoi vicini
         tmp_vertex ← Nome(elementoVertice)

         Se Popolazione[vertex] è uguale a 0: # non connesso
            Ritorna True

Ritorna False
```

Selection operator

Selection(metodo,Popolazione, fitness):

Se metodo è uguale a 0 do:
a,b = Random(Popolazione,2,fitness) #fitness proporzionale
Se metodo è uguale a 1 do:
a,b = Random(Minimo(Popolazione,2,fitness)) #roulette
Se metodo è uguale a 2 do:
a,b = Random(Popolazione,2) #random

Crossover operator

*Crossover***(***metodo***,** *parent1, parent2, vertexNumber):*

x = Random(vertexNumber)
xAll = Random(vertexNumber,K)
Se metodo è uguale a 0 do: #SinglePoint
parent1 = Unisci(parent1 [0:x], parent2[x;]
parent2 = Unisci(parent2 [0:x], parent1[x;]
Se metodo è uguale a 1 do: #MultiplePoint
Per ogni i in xAll do:
parent1 = Unisci(parent1 [0:i], parent2[i;])
parent2 = Unisci(parent2 [0:i], parent1[i;])
Se metodo è uguale a 2 do:
parent1,parent2 = UniformExchange (parent1,parent2,2) #Uniform

---

Mutation Operator

*Mutation***(***Popolazione, vertexNumber):*

x = Random(vertexNumber)
Popolazione[x] = 1 - Popolazione[x]

MultiParent Crossover operator

MultiParentCrossover **(***baseGene***,** *parent, allPopulation, K):*

```
subItems ← SelezionaTop(allPopulation,k)
index = 0

Per ogni item  in subItems do:
        Se parent è diverso da item do:
                Fintantoche range è vera:
                        Se parent[index] diverso da item[index] do:
                            tmp  = parent[index]
                            parent[index] = item[index]
                            fitness ← Calcolafitness(parent)
                                Se fitness è più grande da oldFitness do:
                                    range = False
                                    parent[index] = tmp
                                    break
                                Altrimenti do:
                                    oldFitness = fitness
                        index = index + 1
```

CAPITOLO 4 – Plotting e risultati

Spiegazione parametri e operatori

Durante l'esecuzione dell'algoritmo, per ogni operatore (selection e crossover) è possibile specificare una metodologia da applicare attraverso l'uso di un contatore.

Ad esempio, nel caso del Selection

- 0 indica la metodologia "Roulette"
- 1 indica Tournament selection
- 2 indica random selection

Allo stesso modo, nel caso dell'operatore Crossover

- 0 indica un single point crossover

- 1 indica un multipoint crossover
- 2 indica un uniform crossover

Il flag per la gestione del multiparent crossover è manipolabile a parte attraverso la variabile "doMultiCross".

La denominazione mp-x, nella colonna crossover indica che è stata applicata una strategia multipla a cascata, composta dalla crossover multipla tra genitori e una strategia base.

Sono state effettuate diverse run dell'algoritmo, con i diversi metodi e probabilità su diverse istanze. Di default, è impostata una popolazione di cento elementi, canditati ad essere cover set.

Per una comprensione migliore chiameremo istanze "piccole" quelle che presentano un numero di archi minore o uguale a 2000 o un numero di nodi minore a 200.

Mentre chiameremo istanze "grandi" l'opposto ovvero quelle che presentano un numero maggiore di archi e/o un numero maggiore di nodi.

La notazione usata in tabella prevede l'affiancamento della lettera (S) o (L), small o large.

In questa suddivisione non rientra l'istanza più grande, quella composta da 10000 nodi, essa verrà trattata a parte.

Nel seguente file allegato è possibile trovare i confronti tra le performance ottenute

1. Con il multiparent eseguito o meno.
2. Con il possibile miglioramentro local search al posto del passo di mutation
3. Confronto tra stessi operatori su istanze diverse

iA_RESULTS.xlsx

Si sono eseguite tutte le 71 istanze, registrando lo score per ognuno di loro.

Non si sono ottenuti grandi miglioramenti con il rimpiazzo della localsearch per portare avanti la migliore soluzione del passo genetic. Solo in 4 casi su 71 si è ottenuto un valore minore dello score.

Per quanto riguarda l'operatore MP si è registrato un miglioramento nel 22% dei casi, rispetto alla singola crossover.

Il confronto tra gli operatori non ha portato un distacco completo di una configurazione rispetto ad un'altra, come è possibile notare dalle caselle verdi. Non vi sono abbastanza evidenze per preferire una configurazione valida per tutte le tipologie di istanze

Per quanto riguarda l'istanza composta da 800 nodi non vi è stato un sostanziale miglioramento, nonostante i diversi parametri e la maggiore popolazione., tuttavia la migliore configurazione si è rivelata la **(0,0)** ovvero la strategia roulette selection e il single point crossover.

Andando a vedere le convergenze delle istanze durante l'esecuzione degli algoritmi. L'algoritmo, come è possibile vedere va a convergenza tra la 3000° iterazione e la 4000°.

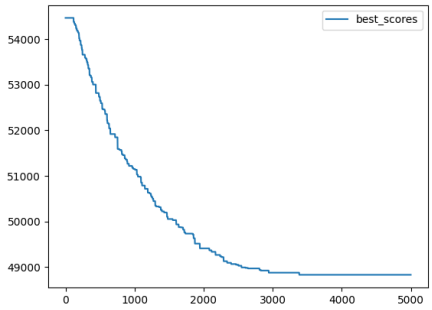


Figura 1 - 800 nodi

Convergenza lenta rispetto al resto delle istanze dove la convergenza avviene in maniera più rapida. Si è notato inoltre che questa sia legata e dovuta maggiormente al numero di nodi rispetto al numero di archi presenti.

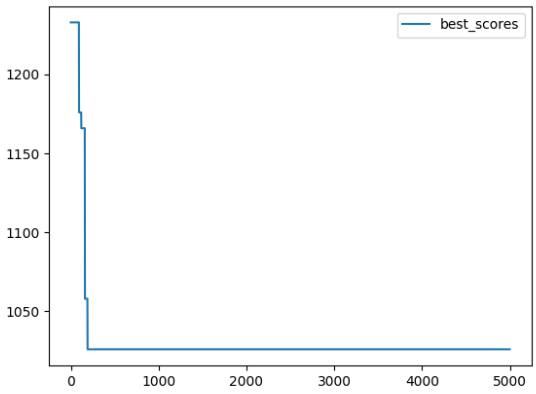


Figura 2 - 20 nodi

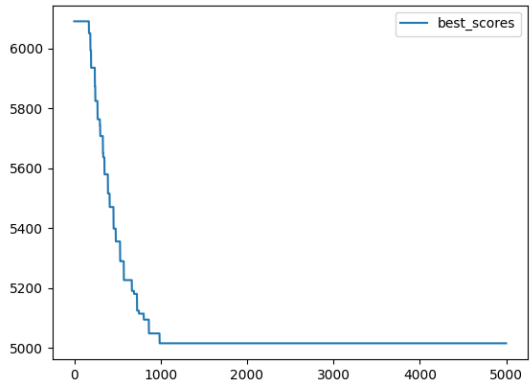


Figura 3 - 100 nodi

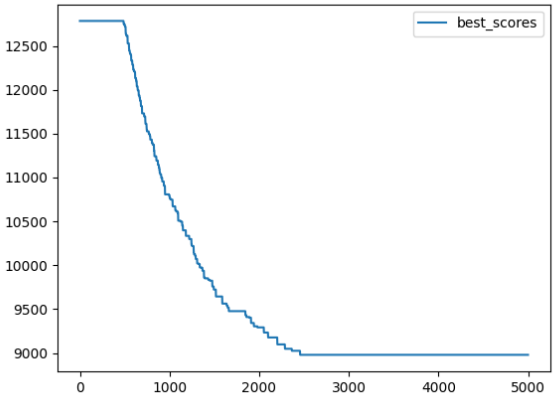


Figura 4 - 200 nodi

CAPITOLO 5 – Conclusioni e miglioramenti

I risultati potrebbero essere migliorati mediante l'incremento della popolazione e l'aumento della variabile limite FE. Un aumento della popolazione permetterebbe di trovare e quindi di valutare più soluzioni. Tuttavia, il limite della FE e una popolazione di 100 elementi, anche per il problema più complesso hanno portato alla convergenza dell'algoritmo.

Un altro miglioramento possibile potrebbe essere quello di estendere il concetto di crossOver non a due genitori ma ad N genitori. Valutare nell'insieme più elementi, prendendo da ognuno il meglio potrebbe essere un buon approccio per la generazione di un "super" figlio. È

possibile applicare questo principio ricorsivamente e cercare la miglior combinazione genetica tra i padre.

Per poter risparmiare tempo macchina è possibile inserire un accorgimento dopo lo step della mutazione. Se il cambio di uno o più bit può portare a un peggioramento della soluzione, forse è meglio non imboccare quella strada. È anche vero che, a volte, per poter sfuggire da punti di minimo locali è necessario esplorare nuovi elementi, mutando numerosi bit.

Parlando invece della fase di selection, la ricerca suggerisce altri metodi utili per risolvere il problema, alcuni si discostano dal concetto di algoritmo genetic, altri lo affiancano.
Tra gli operatori non implementati che vale la pena esplorare vi sono:

- Steady state selection
- Elitist selection
- Boltzmann selection