ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria e Scienze Informatiche

Algoritmi di ottimizzazione per la pianificazione delle attività di un satellite

Tesi di laurea in
Ricerca operativa

Relatore
Prof. Daniele Vigo

Correlatore
Dott. Luca Accorsi

Candidato
Mattia Matteini

II Sessione di Laurea
Anno Accademico 2021-2022

Sommario

L'obiettivo principale di molti problemi industriali è tipicamente massimizzare i profitti o minimizzare costi o tempi di produzione. Questi problemi sono detti "di ottimizzazione" poiché bisogna ottimizzare determinati processi o attività attraverso decisioni che portino alla soluzione ottima del problema. Il giusto utilizzo di modelli matematici può condurre, tramite l'utilizzo di algoritmi esatti, alla soluzione ottima di un problema di questo tipo. Queste tecniche sono spesso basate su l'enumerazione completa di tutte le possibili soluzioni e ciò potrebbe pertanto richiedere una quantità di calcoli talmente elevata da renderle di fatto inutilizzabili. Per risolvere problemi di grandi dimensioni vengono quindi utilizzati i cosiddetti algoritmi euristici, i quali non assicurano di trovare la soluzione ottima del problema, ma promettono di trovarne una di buona qualità.
In questa tesi vengono analizzati, sviluppati e confrontati entrambi gli approcci, attraverso l'analisi di un problema reale che richiede la pianificazione delle attività di un satellite.

Ai miei genitori che mi hanno permesso tutto,
a chi mi è stato affianco
e infine a me stesso.

Indice

Elenco dei listati

Elenco degli algoritmi

Capitolo 1

Introduzione

Struttura della tesi. Capitolo 1 introduce il contesto e le discipline trattate in questa tesi. Capitolo 2 introduce il problema analizzato e affrontato in questo elaborato. Capitolo 3 descrive il modello matematico del problema enunciato nel capitolo precedente per risolverlo in modo esatto. Capitolo 4 elenca e descrive brevemente alcuni metaeuristici tra i più popolari nel settore dell'ottimizzazione combinatoria. Capitolo 5 approfondisce nel dettaglio la struttura dell'algoritmo genetico, utilizzato successivamente per risolvere il problema in modo euristico. Capitolo 6 descrive entrambi gli algoritmi (esatto ed euristico) implementati, volti a risolvere il problema. Capitolo 7 analizza e confronta i risultati dei due approcci quando eseguiti su diverse istanze del problema. Infine, Capitolo 8 conclude la tesi traendo le conclusioni finali basate sui risultati ottenuti.

1.1 La Ricerca Operativa

La ricerca operativa (operations research, OR) è una disciplina che unisce la matematica e l'informatica per sviluppare ed applicare metodi scientifici a problemi decisionali che si presentano in strutture organizzate complesse.
Il termine "ricerca operativa" nasce poco prima della seconda guerra mondiale in Gran Bretagna, quando durante un'esercitazione militare si cercò di analizzare al meglio i dati forniti da radar e aerei, con lo scopo di salvare più piloti possibili. Il sovrintendente dell'esercitazione usò per la prima volta il termine *"operational research"* nella relazione conclusiva del progetto (1938), da qui quindi deriva l'etimologia del termine, che letteralmente vuole dire "ricerca nelle operazioni" (in quel caso, militari). Egli dunque ebbe il merito di mettere insieme in un unico gruppo di ricerca, composto da scienziati di discipline diverse tra cui matematici, ingegneri e statistici, col fine di migliorare gli aspetti operativi delle esercitazioni belliche.

Questa materia interdisciplinare si preoccupa quindi di risolvere i problemi di ottimizzazione, nei quali si vuole massimizzare o minimizzare una determinata funzione, trovando in questo modo la cosiddetta soluzione ottima. Questa materia è chiamata anche "scienza delle decisioni", perché per arrivare alla soluzione ottima è necessario prendere le giuste decisioni tra tutte quelle possibili.
La procedura di analisi e progettazione di cui fa più uso la ricerca operativa è chiamata **Programmazione Matematica**, attraverso la quale si riesce a definire il *modello matematico* di un problema ossia la sua rappresentazione in termini matematici. Un problema si dice di **Programmazione Lineare** (PL) se i suoi vincoli si riescono a esprimere tramite disequazioni lineari e se la sua funzione obiettivo è anch'essa lineare. Se la funzione da ottimizzare è quadratica, si parla di **Programmazione Quadratica**. Nel caso più generale e complesso in cui la funzione obiettivo o i vincoli possono essere di qualsiasi tipo si parla di **Programmazione Non Lineare** (PNL).
I problemi di ottimizzazione si possono suddividere ancora in **interi** e **continui**. Si parla di **Programmazione Lineare Intera** (PLI) quando il modello del problema è lineare e le soluzioni ammissibili comprendono variabili che possono assumere solo numeri interi, mentre si parla di **Programmazione Lineare Continua** (PLC) quando le variabili possono assumere anche valori non interi.
Durante questa tesi verrà analizzato e modellato un problema di PLI, descritto nel Capitolo 2, basato su di un problema reale riguardante la pianificazione delle attività di un satellite.

1.2 L'importanza di Algoritmi e Strutture Dati

Come detto precedentemente, la ricerca operativa è una materia pluridisciplinare, che coinvolge in particolar modo l'informatica e l'ingegneria.
Quando si ha di fronte un problema complesso (pochi problemi reali sono semplici), spesso non è possibile trovare la soluzione ottima a causa dell'elevata complessità computazionale richiesta dalla maggior parte degli algoritmi esatti di tipo matematico. È necessario quindi ricorrere ad **algoritmi euristici** i quali non assicurano la soluzione ottima ma ne promettono una "buona". Per realizzare un buon euristico serve analizzare bene il problema e progettare delle strutture dati e degli algoritmi *ad-hoc* per ottimizzare le prestazioni dell'algoritmo risolutivo. Infatti, quando si affrontano istanze di grandi dimensione senza utilizzare strutture dati ed algoritmi adatti, si rischia di perdere il grosso vantaggio che si guadagna usando gli euristici: la rapidità.
Si deve tenere conto però che bisogna sempre bilanciare bene memoria e prestazioni, strutture dati e algoritmi. Tipicamente infatti più si cerca di migliorare la velocità, più memoria si rischia di occupare.

1.3 Perché un euristico?

Gli euristici permettono di trovare soluzioni, non necessariamente ottime, a problemi di complessità non-polinomiale in tempi di calcolo che invece sono polinomiali. Esistono diversi tipi di euristici, ognuno di essi è più adatto ad uno specifico tipo di problema.
Generalmente non si ha modo di capire quanto la soluzione trovata tramite un euristico sia distante dalla soluzione ottima che sarebbe prodotta da un algoritmo esatto. Tuttavia l'euristico ci assicura un tempo di calcolo estremamente più ridotto rispetto ad un approccio matematico, il quale nei casi peggiori potrebbe richiedere anni di attesa.
Per esempio, l'istanza più grande del problema del commesso viaggiatore (TSP), è stata risolta su un cluster di server in un tempo che scalato per essere confrontabile con il tempo di un unico computer (nello specifico su AMD Opteron 250 2.4 GHz), corrisponde approssimativamente a **136 anni** di computazioni. Questo caso di studio, chiamato *pla85900*, consiste in un TSP da 85900 città, è stato portato a termine nel 2006 ed è descritto in Applegate [2].

Capitolo 2

Il problema PLATiNO

PLATiNo è il nome di una missione spaziale che consiste nel lancio di due satelliti (PLATiNo-1 e PLATiNO-2) che saranno messi in orbita tra la fine del 2022 e la metà del 2024. Le piattaforme per questi minisatelliti tecnologicamente avanzati e leggeri (tra i 175 kg e 350 kg) sono state finanziate da un insieme di imprese tra cui ASI, SITAEL e Thales Alenia Space.
Il problema trattato in questa tesi riguarda la pianificazione delle attività di un satellite in un determinato orizzonte di tempo.
L'insieme della attività viene chiamato *piano*.

2.1 Acquisition Requests e Data Take Opportunities

Ad inizio pianificazione è conosciuto un insieme di richieste di acquisizione, chiamate Acquisition Requests (ARs), ovvero richieste di fotografie della Terra da effettuare.
Ogni AR è caratterizzata da un'occupazione di memoria e da un indice di importanza. Siccome il satellite potrebbe passare più volte sopra la stessa area, ciascun AR può essere soddisfatto in più opportunità di acquisizione, chiamate Data Take Opportunities (DTOs).
È sufficiente che ciascun AR sia servito da un unico DTO.
Ogni DTO è caratterizzato da un tempo di inizio e fine acquisizione durante il quale il satellite effettua la fotografia dell'area richiesta.
Inoltre, sono presenti delle attività che il satellite deve svolgere, che hanno priorità su tutte le altre, queste ultime sono chiamate Platform Activities Windows (PAWs).

2.2 Downlink Opportunities e Downlink Policy

Durante il proprio tragitto, il satellite può incontrare delle stazioni sulla Terra grazie alle quali può scaricare un sottoinsieme dei dati acquisiti fino a quel momento. Le opportunità di scaricamento, dette Downlink Opportunities (DLOs), prevedono che il satellite invii le acquisizioni in un certo ordine, uno di questi potrebbe essere quello decrescente rispetto alla dimensione in modo da liberare più memoria possibile per eventuali successive acquisizioni (questa è chiamata politica di downlink o Downlink Policy).
Lo scaricamento delle acquisizioni può avvenire finché il satellite è vicino alle stazioni di terra. Ciascuna DLO è quindi caratterizzato da un istante di inizio ed uno di fine: all'interno di questo intervallo è possibile effettuare lo scaricamento. Inoltre, lo scaricamento è discreto, ovvero un'acquisizione deve essere interamente inviata sulla Terra (non può essere frazionata).

Capitolo 3

Modellazione matematica

É buona prassi quando si vuole risolvere un problema di ottimizzazione, analizzarne la complessità e creare un **modello matematico** che lo descriva formalmente. Il problema trattato può essere modellato tramite programmazione lineare intera (PLI).

3.1 Problema parziale

Un primo modello del problema ignora gli scaricamenti e tutti i vincoli legati ad essi. Partire da una versione semplificata del problema permette un analisi più chiara e attenta di quest'ultimo. Come spiegato in seguito, il problema parziale può essere modellato definendo un insieme di variabili decisionali che modellano le DTO.

3.1.1 Definizioni

Definiamo i seguenti insiemi:

- T come l'insieme delle DTOs di n elementi
- A come l'insieme delle ARs di k elementi
- P come l'insieme delle PAWs di l elementi

Modelliamo l'insieme delle DTOs con le variabili binarie:

$$x_i = \begin{cases} 1 & \text{se la DTO i è nel piano di acquisizioni} \\ 0 & \text{altrimenti} \end{cases}$$

Sia $f : T \to A$ una funzione suriettiva, che per ogni DTO restituisce l'AR che soddisfa.

Si definiscono ora le seguenti costanti:

- C la capacità di memoria del satellite
- p_i con $i = 1, \ldots, n$ la priorità della DTO i (relativa all'AR che soddisfa)
- m_i con $i = 1, \ldots, n$ l'occupazione di memoria della DTO i
- s_i con $i = 1, \ldots, n$ il timestamp d'inizio della finestra temporale della DTO i
- f_i con $i = 1, \ldots, n$ il timestamp di fine della finestra temporale della DTO i
- μ_p con $p = 1, \ldots, l$ il timestamp d'inizio della finestra temporale della PAW p
- σ_p con $p = 1, \ldots, l$ il timestamp di fine della finestra temporale della PAW p

3.1.2 Funzione obiettivo

Si vuole massimizzare l'importanza delle acquisizioni scelte per il piano.

$$\max \sum_{i=1}^{n} p_i x_i \tag{3.1}$$

3.1.3 Vincoli

Vincolo di memoria

L'occupazione di memoria complessiva delle acquisizioni effettuate non deve superare la capacità di memoria del satellite.

$$\sum_{i=1}^{n} m_i x_i \leq C \tag{3.2}$$

Vincolo di singola soddisfazione

Ogni AR deve essere soddisfatta da una DTO associata al piu' una volta.

$$\sum_{i \in \{i | f(i) = a\}} x_i \leq 1, \forall a = 1, \ldots, m \tag{3.3}$$

Vincoli di sovrapposizione

Il satellite non può effettuare delle acquisizioni durante i periodi di PAWs. Si suppone non ci siano DLO e PAW sovrapposte.

$$x_i = 0, \forall i, p \mid s_i \leq \sigma_p \wedge f_i \geq \mu_p \tag{3.4}$$

Inoltre se sono due DTOs a sovrapporsi, se ne può scegliere al più una.

$$x_i + x_j \leq 1, \forall i, j \mid s_i \leq f_j \wedge f_i \geq s_j \tag{3.5}$$

3.2 Problema completo

In questa sezione si introduce il problema completo, il quale include anche gli scaricamenti.
Per fare ciò, si estende quindi il modello del problema precedentemente introdotto ponendo ulteriori vincoli. In particolare, si vuole ottenere il piano di scaricamenti ottimo con lo stesso fine: ottimizzare l'importanza delle acquisizioni. La funzione obiettivo quindi rimarrà la stessa.
Inoltre, si suppone che le DLOs non si sovrappongano tra di esse.

3.2.1 Definizioni

Definiamo D come la lista di DLOs di m elementi ordinate temporalmente, e le modelliamo esattamente come le DTOs:

$$y_j = \begin{cases} 1 & \text{se la DLO j è nel piano di scaricamento} \\ 0 & \text{altrimenti} \end{cases}$$

Alla lista D si aggiunge anche una DLO d fittizia di durata nulla che usata in seguito per la modellazione di alcuni vincoli.

$$D = D \cup d, |D| = m + 1$$

Si definisce anche un altro insieme di variabili binarie $z_{i,j}$ in questo modo:

$$z_{i,j} = \begin{cases} 1 & \text{se la DTO } x_i \text{ viene scaricata durante la DLO } y_j \\ 0 & \text{altrimenti} \end{cases}$$

Si aggiungono poi ulteriori costanti:

- R_D il rateo di scaricamento del satellite
- δ_j con $j = 1, \ldots, m$ il timestamp d'inizio della finestra temporale della DLO j
- ϕ_j con $j = 1, \ldots, m$ il timestamp di fine della finestra temporale della DLO j

3.2.2 Come cambiano i vincoli

Il vincolo di singola soddisfazione rimane invariato, tuttavia cambiano quelli sulla sovrapposizione e sulla memoria.

Vincolo di sovrapposizione

Il satellite non può effettuare due attività contemporaneamente, quindi per qualsiasi coppia di DTO e DLO che si sovrappone, se ne deve scegliere al più una.
Si hanno di fronte due possibili strade:

1. Si aggiunge un vincolo classico di sovrapposizione e si lascia scegliere all'algoritmo quali DLOs scegliere.

$$x_i + y_j \leq 1, \forall i, j \mid s_i \leq \phi_j \wedge f_i \geq \delta_j \tag{3.6}$$

2. Si filtrano e rimuovono le DTOs che sovrappongono con le DLO, dando priorità agli scaricamenti, questo ci permette di risparmiare un vincolo e tenere il modello più semplice.

In questo caso si preferisce seguire la seconda opzione, poiché la prima, seppur più rigorosa e precisa, trasformerebbe il modello PLI in uno di programmazione quadratica.

Vincolo di memoria acquisita

Considerando anche i DLOs, la memoria occupata del satellite durante la sua orbita non andrà solo ad aumentare, ma anche a diminuire (durante l'esecuzione di uno scaricamento).
Si definiscono allora una serie di variabili che rappresentano la quantità di memoria occupata del satellite per ogni istante di inizio delle DLOs.
La prima di queste $m+1$ variabili (considerando l'artificiale), è definita quindi come la somma delle memorie di tutte le DTOs acquisite prima della DLO iniziale:

$$\theta_0 = \sum_{i \in \{i | f_i < \delta_0\}} m_i x_i$$

Le successive m variabili dovranno tenere conto della memoria rimasta dall'ultima DLO j e delle DTOs scelte tra la DLO j e la DLO $j-1$.

$$\theta_j = \theta_{j-1} - \sum_i m_i z_{i,j-1} + \sum_{i \in \{i | \phi_{j-1} < s_i \wedge f_i < \delta_j\}} m_i x_i$$

Il valore di ognuna di queste variabili dovrà essere minore della capacità totale del satellite. Il vincolo diventa quindi il seguente:

$$\theta_j \leq C, \forall j \tag{3.7}$$

Vincolo di memoria scaricata

Tuttavia, con solamente questi vincoli definiti, l'algoritmo cercherà di scaricare per poter liberare più memoria possibile e quindi effettuare più acquisizioni, massimizzando così il risultato.
In realtà il satellite non può scaricare più di un certo numero di acquisizioni durante ogni DLO. Questo limite è definito dal seguente vincolo:

$$\sum_i m_i z_{i,j} \leq R_D(\phi_j - \delta_j), \forall j \tag{3.8}$$

Vincolo di singolo scaricamento

Ogni DTO può essere scaricata una volta soltanto durante il piano o, in altre parole, la stessa DTO può essere scaricata al più una volta durante una singola DLO.

$$\sum_j z_{i,j} \leq 1, \forall i \tag{3.9}$$

Vincolo di scaricamento post-acquisizione

Una DTO può essere scaricata solo se il satellite l'ha acquisita durante il piano.

$$z_{i,j} \leq x_i, \forall i, j \tag{3.10}$$

Dato che sia x_i che $z_{i,j}$ sono variabili binarie, possiamo rappresentare i precedenti due vincoli nel seguente:

$$\sum_j z_{i,j} \leq x_i, \forall i \tag{3.11}$$

Vincolo temporale

Una DTO non può essere scaricata prima di essere acquisita.

$$z_{i,j} = 0, \forall i, \forall j \mid \delta_j < f_i \tag{3.12}$$

Capitolo 4

Euristici e metaeuristici

Con il modello matematico descritto nel capitolo precedente, si è in grado di trovare la soluzione ottima del problema tramite l'utilizzo di un MILP (mixed integer linear programming) solver.
Tuttavia nel caso in cui l'istanza del problema sia troppo grande, l'esecuzione dell'algoritmo impiegherebbe troppo tempo poiché la complessità computazionale di algoritmi come quello del *Simplesso* o del *Branch and Bound* è esponenziale.
Si rende necessario quindi trovare un approccio alternativo di complessità polinomiale, per questo sono stati ideati gli algoritmi **euristici**.

Gli euristici sono algoritmi che non garantiscono di trovare la soluzione ottima del problema, ma promettono di trovarne una abbastanza buona. Si chiamano in questo modo perché per risolvere i problemi non seguono un percorso rigoroso, ma si affidano all'intuito, allo stato temporaneo delle circostanze e di conseguenza alla casualità. Alcuni di questi sono **greedy**, ovvero scelgono passo dopo passo la via più promettente nell'immediato effettuando delle **ricerche locali** (procedure euristiche in cui viene migliorata la soluzione attraverso semplici mosse come scambi e sostituzioni). Sono detti quindi miopi perché non riescono a valutare l'intero contesto.
In genere si definisce una *funzione di fitness*, che calcola e valuta la "bontà" di una soluzione. Si vuole quindi ottenere la soluzione che dà il valore di fitness migliore.

Purtroppo capita spesso che usando un tradizionale euristico, si trovi una soluzione che in realtà è un massimo locale (se si vuole massimizzare) nella funzione di fitness e, bloccandosi su di essa, si impedisce l'esplorazione di altre possibili soluzioni che possono essere potenzialmente migliori di quella trovata.

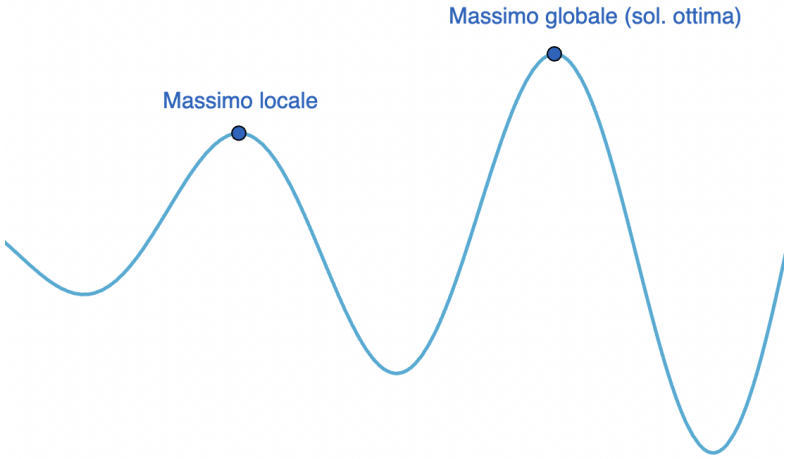

Figura 4.1: Esempio di una funzione di fitness

Per far fronte a questo problema sono stati studiati degli algoritmi che fanno uso di diverse procedure euristiche e che permettono di uscire dai massimi (o minimi) locali, visitando per esempio soluzioni non ammissibili o di peggiore qualità. Questi algoritmi sono chiamati **metaeuristici**.
Esiste una gran varietà di questo tipo di algoritmi, che hanno molti aspetti in comune, ma ognuno presenta delle differenze rispetto ad altri. In seguito vengono riportate alcune delle loro caratteristiche principali.

Nature-inspired. Alcuni metaeuristici traggono spunto dalla *natura*, da alcuni processi o sistemi naturali, o da come gli organismi riescono a evolvere, mutare e migliorare nel tempo.

Population-based vs. single point search. I metaeuristici si differenziano principalmente per la quantità di soluzioni che processano ad ogni iterazione. Algoritmi come Tabu Search (Sezione 4.2), Simulated Annealing (Sezione 4.1), Iterated Local Search e Variable Neighborhood Search hanno la proprietà comune di lavorare con una sola soluzione alla volta che viene modificata durante il corso dell'algoritmo. Essi sono anche detti *trajectory methods*.
Altri algoritmi invece processano un insieme di soluzioni ad ogni iterazione, delle quali poi verrà scelta la migliore al termine dell'esecuzione. Alcuni di essi sono Genetic algorithm (Sezione 4.5), Ant Colony algorithm (Sezione 4.3), Particle Swarm algorithm.

Dynamic vs. static objective function. Alcuni metaeuristici si differenziano per l'uso che fanno della funzione obiettivo. I più classici tengono la funzione obiettivo "così com'è" per l'intera durata dell'algoritmo, altri invece la modificano in base alle informazioni raccolte durante l'esecuzione per cercare di fuggire dai massimi locali.

One vs. various neighborhood structures. La maggior parte degli algoritmi metaeuristici utilizzano una struttura dati con una singola soluzione vicina (*neighbor*). In altre parole, la topologia del panorama della funzione di fitness non cambia nel corso dell'algoritmo. Altre metaeuristiche, come Variable Neighborhood Search (VNS), usano un insieme di soluzioni vicine che dà la possibilità di diversificare la ricerca passando per diversi panorami di miglioramento.

In generale i metaeuristici sono facili da implementare e riscontrano soluzioni migliori, tuttavia hanno un costo computazionale leggermente più alto rispetto ai tradizionali euristici. Alcuni di essi sono brevemente descritti in questo capitolo, approfondendo maggiormente l'**algoritmo genetico** (Capitolo 5) che verrà utilizzato per risolvere il problema trattato in questo elaborato.

4.1 Simulated Annealing

Il Simulated annealing (SA) è uno dei primi algoritmi che hanno di fatto proposto un metodo volto a risolvere il problema del massimo locale. Originariamente è stato presentato nel 1983 in Kirkpatrick et al. [24] nel quale viene affermato che c'è una forte connessione tra la meccanica statistica e l'ottimizzazione combinatoria. L'innovazione di questo algoritmo consiste nell'accettare delle mosse peggiorative con una probabilità che decresce durante il corso dell'algoritmo.
Si inizia generando una soluzione s (casuale o tramite un euristico) e definendo un parametro T detto *temperatura*. Ad ogni iterazione, è scelta casualmente una nuova soluzione $s' \in N(s)$, dove $N(s)$ è l'insieme di soluzioni "vicine" ad s.
La nuova soluzione s' sostituisce s se $f(s') < f(s)$ o se $f(s') \geq f(s)$, in base ad una probabilità che è in funzione di T e $f(s') - f(s)$. Questa probabilità è generalmente calcolata seguendo la distribuzione di Boltzmann, in questo caso sarà: $e^{-\frac{f(s')-f(s)}{T}}$.
La temperatura T decresce durante il processo di ricerca, ciò significa che nella fase iniziale dell'algoritmo la probabilità che faccia decisioni peggiorative (*uphill*) è molto alta, ma decresce gradualmente fino a diventare un classico algoritmo iterativo di ricerca locale. Questo permette inizialmente di uscire dai massimi locali e di esplorare maggiormente lo spazio di ricerca, andando infine a migliorare le soluzioni trovate.

Algoritmo 1 Simulated annealing

$s \leftarrow GenerateInitialSolution()$
$T \leftarrow T_0$
while termination conditions not met **do**
  $s' \leftarrow PickAtRandom(N(s))$
  if $f(s') < f(s)$ **then**
    $s \leftarrow s'$ % *downhill*
  else
    $r \leftarrow RandomValue(0, 1)$
    if $r < e^{-\frac{f(s')-f(s)}{T}}$ **then**
      $s \leftarrow s'$ % *uphill*
    end if
  end if
  $Update(T)$
end while

Questo processo è analogo a quello di annichilimento utilizzato nella metallurgia per temprare i metalli. Questi vengono inizialmente scaldati e successivamente fatti raffreddare lentamente mentre si continua a lavorarlo, con lo scopo di rimuovere particolari imperfezioni.
La scelta della temperatura T iniziale e del metodo di raffreddamento (funzione che definisce T ad ogni iterazione) è cruciale per la buona implementazione di un SA. Se per ogni iterazione k, la temperatura è definita nel seguente modo $T_{k+1} = Q(T_k, k)$ allora la funzione $Q(T_k, k)$ dovrebbe essere logaritmica per riuscire a convergere all'ottimo globale. Tuttavia a causa dell'elevato costo computazionale che richiederebbe questa scelta, non viene mai presa in considerazione. Si preferisce scegliere una $Q(T_k, k)$ che segua una legge geometrica[1].
Attualmente gli SA vengono principalmente usati come procedure in altri metaeuristici e ha dei successori tra cui Threshold Accepting e The Great Deluge Algorithm.

[1]Non è necessario che la funzione che aggiorna T sia decrescente monotona, alcuni schemi di raffreddamento elaborati possono prevedere un incremento di T

4.2 Tabu Search

L'algoritmo Tabu Search (TS) è tra i più citati e usati per i problemi di ottimizzazione combinatoria. Le idee di base del TS sono state introdotte per la prima volta in Glover (1986) [16]. Una descrizione del metodo e dei suoi concetti può essere trovata in Glover e Laguna (1998) [19].
TS utilizza in modo esplicito la cronologia delle soluzioni trovate, sia per evadere dai minimi locali sia come strategia esplorativa. Successivamente è descritta una semplice versione di TS.

Algoritmo 2 Tabu search

$s \leftarrow GenerateInitialSolution()$
$TabuList \leftarrow \emptyset$
while termination conditions not met **do**
  $s' \leftarrow ChooseBestOf(N(s) \setminus TabuList)$
  $Update(TabuList)$
end while

Questa semplice versione applica una ricerca locale migliorata grazie alla memoria utilizzata per tenere traccia delle ultime soluzioni esplorate. Questo viene implementato tramite l'utilizzo della cosiddetta *Tabu List*, di lunghezza l, che proibisce di muoversi nelle l soluzioni visitate precedentemente. Ad ogni iterazione, la migliore soluzione in $N(s)$ non tabu ($N(s) \setminus TabuList$), viene scelta come soluzione corrente, e subito dopo viene aggiunta alla lista Tabu al posto di una già presente (tipicamente in ordine First-In-First-Out).
L'algoritmo si ferma quando si verifica la condizione di terminazione. Potrebbe terminare anche nel caso in cui $N(s) \setminus TabuList = \emptyset$, ovvero quando tutte le soluzioni vicine sono proibite dalla lista tabu.
L'uso della lista tabu impedisce di tornare in soluzioni viste in precedenza e di conseguenza consente di evitare cicli infiniti, inoltre forza l'esplorazione di diverse soluzioni, anche se peggioranti (*uphill moves*). La lunghezza l della lista tabu controlla quanta memoria utilizzare durante il processo. Con una lista corta la ricerca si concentrerà in zone più piccole dello spazio di ricerca, mentre con una lista grande si forza ad allargare la ricerca e ad esplorare più regioni perché il numero delle soluzioni proibite sarà più alto.
La lunghezza l della lista tabu può essere variata nel corso dell'algoritmo, per aumentare la robustezza di quest'ultimo. Alcuni esempi possono essere trovati in Taillard (1991) [34] dove la lunghezza l viene periodicamente riassegnata casualmente in un intervallo prestabilito.
Sono stati studiati molti utilizzi di liste tabu dinamiche, per esempio la lunghezza della lista potrebbe essere incrementata se si verificano delle ripetizioni continue

nelle soluzioni esplorate (quindi è necessario diversificare), al tempo stesso potrebbe essere decrementata quando non si hanno dei miglioramenti significativi. Tecniche avanzate di questo tipo sono descritte in Glover (1990) [17].
Purtroppo la lista tabu presenta una grossa limitazione, che è proprio la memoria stessa, infatti spesso può essere oneroso tenere traccia di una lunga lista (soprattutto se di oggetti complessi) e questo fa diventare l'algoritmo altamente inefficiente. Al posto delle soluzioni, potrebbero essere memorizzati alcuni loro attributi, ma può accadere che alcuni di essi siano comuni tra più soluzioni, in questo caso potrebbe essere implementata una lista tabu per ognuna di esse. Memorizzare attributi al posto delle intere soluzioni comporta una grossa perdita di dati, una singola soluzione tabu potrebbe proibire diverse altre soluzioni esplorabili.
Questi problemi non si porrebbero nel caso in cui ogni soluzione fosse identificata univocamente da un attributo (ID).

4.3 Ant Colony algorithms

Ant Colony Optimization (ACO) è un paradigma di progettazione metaeuristico proposto in Dorigo 1992 [9], 1996 [14] e 1999 [13]. Esso si ispira al comportamento delle formiche nelle colonie, in grado di trovare il percorso più breve che collega la tana ad una risorsa alimentare. Camminando dal cibo alla tana e viceversa, le formiche depositano sul terreno una sostanza chiamata *feromone*. I feromoni col passare del tempo tendono a diminuire evaporando. Quando una formica deve scegliere la direzione in cui procedere, le vie che presentano una maggiore concentrazione di feromoni hanno una più alta probabilità di essere intraprese.
Gli algoritmi ACO, sono basati su modelli probabilistici parametrizzati (*pheromone model*), che vengono usati per definire la quantità di feromoni depositati si vari sentieri.
Le formiche artificiali costruiscono in maniera incrementale delle soluzioni, aggiungendo man mano delle componenti. Per fare questo, le formiche artificiali eseguono dei cammini casuali su un grafo fortemente connesso $G = (C, L)$ nel quale i vertici C sono le componenti della soluzione C, mentre L sono le connessioni. Questo grafo è comunemente chiamato *grafo di costruzione*. Quando il problema di ottimizzazione combinatoria presenta dei vincoli, essi sono direttamente considerati nella costruzione della soluzione, in modo tale che possano essere prese in considerazioni solamente soluzioni *feasible* (che rispettano tutti i vincoli). In problemi complessi tuttavia, spesso è inevitabile dover considerare anche soluzioni non ammissibili.
L'insieme delle componenti è denotato dall'insieme C, mentre le connessioni dall'insieme L. Una connessione l_{ij}, che collega la componente i alla componente j, può avere associato un valore che indica la quantità di feromone depositata su di

essa. L'insieme di feromoni e dei valori stessi vengono denotati rispettivamente con T e τ_{ij}. Inoltre, componenti e connessioni possono avere associato un valore euristico indicato con η_{ij}, che rappresenta un informazione in più tra le componenti i e j. Nel caso del problema del commesso viaggiatore (TSP), questo valore può indicare la "visibilità", cioè la distanza tra una città e l'altra. L'insieme di questi valori viene denotato con H. Questi parametri sono utilizzati dalle formiche per effettuare decisioni probabilistiche su quali movimenti fare sul grafo di costruzione. Le probabilità che inficiano sul movimento delle formiche sono dette *transition probabilities*.
Infine si ha l'insieme di formiche definito come A. Ogni formica $a \in A$, costruisce la soluzione s_a.
Successivamente viene mostrata la prima versione di un algoritmo ACO, in letteratura chiamato Ant System (AS) proposto in Dorigo et al. (1996) [14].

Algoritmo 3 Ant system

```
InitializePheromoneValues(T)
while termination conditions not met do
    for all ants a ∈ A do
        s_a ← ConstructSolution(T, H)
    end for
    ApplyOnlineDelayedPheromoneUpdate(T, {s_a | a ∈ A})
end while
```

La funzione *InitializePheromoneValues(T)* inizializza tutti i feromoni allo stesso piccolo valore $ph > 0$.
In *ConstructSolution(T, H)* la formica costruisce gradualmente una soluzione aggiungendo componenti in base a una probabilità. La scelta probabilistica della nuova componente da aggiungere dipende dalle *transition probabilities* citate precedentemente. Nell'algoritmo AS seguono la cosiddetta regola casuale di transizione proporzionale (una delle *state transition rules*):

$$p_{ij}^a = \begin{cases} \dfrac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{z \in J_i^a} [\tau_{iz}]^\alpha [\eta_{iz}]^\beta} & j \in J_i^a \\ 0 & j \notin J_i^a \end{cases}$$

Dove p_{ij}^a è la probabilità che la formica a scelga il percorso l_{ij}, J_i^a è l'insieme delle componenti (ammissibili per la soluzione) collegate alla componente i, mentre α e β sono due parametri che controllano rispettivamente l'importanza dell'intensità dei feromoni e dell'informazione euristica η.
La funzione *ApplyOnlineDelayedPheromoneUpdate*$(T, \{s_a \mid a \in A\})$, una volta

che tutte le formiche hanno costruito una soluzione, aggiorna tutti i valori dei feromoni tramite la regola *online delayed pheromone update*:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \sum_{a \in A} \Delta\tau_{ij}^{s_a}$$

Dove $\Delta\tau_{ij}^{s_a}$ è la quantità di feromone depositata dalle formiche sul tratto l_{ij}, mentre $0 < \rho < 1$ è il parametro indicante il rateo di evaporazione dei feromoni. Questa regola di aggiornamento dei feromoni mira all'aumento di essi sui componenti trovati in soluzioni di alta qualità.

Successivamente, verrà descritto l'algoritmo ACO più generale, il quale è basato sugli stessi principi dell'AS. Esso presenta miglioramenti ed estensioni al framework metaeuristico precedentemente illustrato, dopo anni di studio e ricerche. Consiste in tre principali parti raccolte nel costrutto *ScheduledActivities*, sarà il progettista a decidere come queste debbano essere schedulate e sincronizzate.

Algoritmo 4 Ant colony algorithm

```
while termination conditions not met do
    for all ants a ∈ A do
        ScheduledActivities
            AntBasedSolutionConstruction()
            PheromoneUpdate()
            DaemonActions()    % optional
        EndScheduledActivities
    end for
end while
```

AntBasedSolutionConstruction(): Una formica costruisce una soluzione del problema muovendosi di nodo in nodo nel grafo G. Le formiche si muovono seguendo una politica stocastica locale, facendo uso dei feromoni e dei valori euristici su componenti e connessioni del grafo (per esempio la *state transition rule* definita in precedenza per l'AS). Durante il percorso le formiche tengono traccia della soluzione parziale costruita in termini di percorso e connessioni.
PheromoneUpdate(): Quando una componente j viene aggiunta a una soluzione, la formica può aggiornare il valore di feromone τ_{ij} sul percorso l_{ij}, dove i era l'ultima componente aggiunta. Questo tipo di aggiornamento viene detto *online step-by-step pheromone update*. Una volta che la soluzione è costruita, la formica può effettuare il cammino a ritroso (usando la sua memoria), e aggiornare i feromoni sui sentieri delle componenti attraversate. Questa modifica è chiamata *online delayed pheromone update*. Il processo di evaporazione dei feromoni non deve essere

troppo veloce per evitare che l'algoritmo converga subito a dei minimi (o massimi) locali.
DaemonActions(): Implementa delle operazioni centralizzate che non possono eseguire le singole formiche. Alcuni esempi sono l'uso di una procedura di local search applicata alle soluzioni costruite, o il raccoglimento di informazioni globali usate per incidere sul deposito dei feromoni introducendo adeguati "bias" al processo di ricerca (per esempio per una prospettiva non locale). Un esempio pratico è l'utilizzo di un demone che osserva il percorso di ogni formica della colonia e sceglie se e dove depositare feromoni extra sui sentieri percorsi dalle formiche. L'aggiornamento di feromoni effettuati in questo modo sono detti *offline pheromone updates*. Questo paradigma ha riscosso molto successo ed è stato utilizzato per creare numerose varianti, le migliori sono Ant Colony System (ACS) e MIN-MAX Ant System.

4.4 Neural Networks

Le reti neurali (NN), nascono per risolvere problemi complessi per i quali le classiche tecniche di programmazione non bastano. Riconoscere suoni, classificare immagini e processare il linguaggio naturale sono solo alcuni di molti esempi. Le reti neurali coprono un ruolo fondamentale nel settore dell'intelligenza artificiale e in particolare nel **deep learning**, ma sono anche un grosso punto di riferimento metaeuristico, poiché di fatto, alcuni dei problemi che mirano a risolvere, sono problemi di ottimizzazione combinatoria.
Le NN artificiali si ispirano alle reti neurali biologiche, nelle quali le informazioni vengono passate di neurone in neurone, solo se il segnale che le trasporta è sufficientemente forte. L'idea generale si rappresenta tramite un **grafo di flusso** (diretto aciclico) nel quale i nodi (detti *neuroni*), sono divisi in strati (*layers*). I layers sono classificati in tre modi:

- INPUT, ricevono i dati iniziali
- HIDDEN, processano i dati di strato in strato
- OUTPUT, restituiscono la predizione sotto forma di probabilità

Le origini delle NN si hanno in McCulloch (1943) [25], nel quale viene proposto il primo modello teorico di un rudimentale neurone artificiale. Successivamente nel 1958 viene proposta da Rosenblatt [32] la prima rete neurale: **Perceptron**, che getta le basi dell'apprendimento automatico. Perceptron di Rosenblatt possiede uno strato di nodi (neuroni artificiali) di input e un nodo di output.
Il passo successivo è il Perceptron multistrato (MLP) con Werbos (1974) [35]. Al suo interno, fra i nodi di input e quello di output si trova uno strato nascosto

(*hidden*), dove avviene l'elaborazione delle informazioni provenienti dallo strato di input, che poi vengono inviate al nodo di output. È una rete **feedforward** non lineare: le connessioni in ingresso e in uscita da ogni singolo nodo sono multiple. Un altro passo fondamentale per la storia delle reti neurali è l'introduzione della tecnica di **Error Back-Propagation**, grazie a Rumelhart, Hinton e Williams (1986) [33].

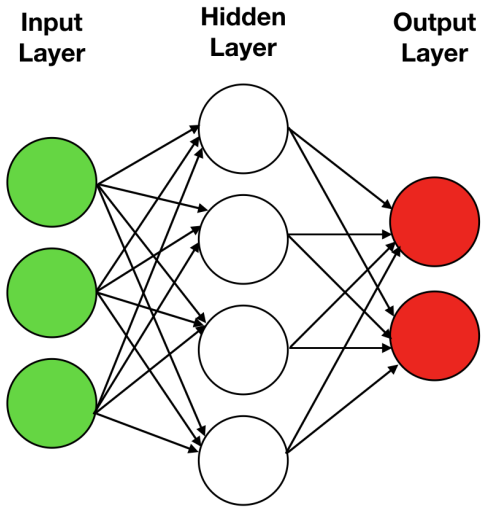

Figura 4.2: Semplice rete neurale feedforward (Medium.com, 2020)

Forward Propagation. Si tratta del processo in cui le informazioni passano da un layer all'altro, seguendo la direzione naturale di propagazione: cioè dall'input verso l'output. All'interno della rete, ogni arco ha un peso. Un buon modo di processare le informazioni è quello di mappare la somma dei pesi, in uno spazio non lineare, attraverso una *funzione di attivazione*.

Funzione di attivazione. I segnali vengono propagati da un neurone all'altro solo se riescono a superare una determinata *soglia di attivazione*. I dati quindi vengono passati in input alla cosiddetta funzione di attivazione che deciderà se il segnale è abbastanza forte o meno. Alcuni esempi sono:

- Funzione *logistica o sigmoide*: $f(x) = \frac{1}{1+e^{-x}}$
- Funzione *tangente iperbolica*: $f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- Funzione *Rectified Linear Unit (ReLU)*: $f(x) = \max(0, x)$

Back Propagation. Tecnica per l'allenamento di reti neurali artificiali *supervisionate* (si ha già cioè un dataset di risultati correttamente etichettati), attraverso cui si ripercorre il grafo all'indietro e si aggiustano i pesi sugli archi, col fine di minimizzare la differenza tra il risultato ottenuto e quello desiderato.

Esistono diversi tipi di reti neurali più complesse di quella classica appena descritta, le principali sono le RNN (NN ricorrenti), CNN, (NN convoluzionali), SNN (spiking NN).

4.5 Genetic algorithms

Gli algoritmi genetici (GA) risalgono alla fine degli anni 50, quando si hanno i primi tentativi di combinare informatica e scienza, in particolare la **biologia**. Vengono introdotti inizialmente da Bremermann (1958) [6], e successivamente resi popolari da Holland (1962) [22]. Quest'ultimo voleva studiare il fenomeno dell'adattamento naturale, ed applicarlo in qualche modo all'interno dei calcolatori. Successivamente si ha un grosso passo in avanti in Holland (1975) [23], dove si introduce lo *Schema Theorem*, il quale definisce le condizioni di lavoro ideali per il successo di un GA. In seguito sono stati studiati e utilizzati in molti contesti: da Bagley (1967) [3] per la realizzazione di un programma per il game-playing, da Rosenberg (1967) [31] per la simulazione di processi biologici e da Cavicchio (1970) [7] per problemi di pattern-recognition. Nel corso del tempo sono stati sviluppate numerose varianti più efficaci e performanti, si rimanda a Dianati et al. (2002) [12] per una storia più accurata sui genetici.
Attualmente i GA vengono impiegati con successo insieme a tecniche di ricerca locale nella risoluzione di problemi di ottimizzazione combinatoria complessi nei quali i metodi esatti sarebbero proibitivi a causa del tempo di calcolo richiesto. I GA ormai rappresentano un framework robusto e flessibile su cui basare molti algoritmi euristici adibiti a risolvere problemi di varie entità.
Successivamente è mostrato lo pseudocodice generale del paradigma GA.
Nel prossimo capitolo verrà descritto in dettaglio l'intero algoritmo.

Algoritmo 5 Genetic algorithm

$P \leftarrow GenerateInitialPopulation()$
while termination conditions not met **do**
  for all chromosomes $s \in P$ **do**
    $FitnessEvaluation(P)$
    $ParentSelection(P)$
    $Crossover(P)$
    $UpdatePopulation(P)$
    $Mutation(P)$
  end for
end while
$s \leftarrow GetBestSolution(P)$

Capitolo 5

Gli algoritmi genetici

5.1 Biologia e Informatica

L'aggettivo *genetico* nasce dall'analogia tra i componenti fondamentali e i comportamenti di questi algoritmi e la biologia vera e propria, sebbene quest'ultima sia infinitamente più complessa del modello qui discusso. In un modello biologico molto semplificato si può supporre che gli organismi siano composti da cellule contenenti un determinato insieme di *cromosomi*. Ognuno di questi codifica una stringa di DNA che racchiude informazioni essenziali dell'organismo stesso. Un cromosoma è suddiviso in *geni*, ciascuno dei quali codifica un *tratto*, ad esempio il colore dei capelli o il gruppo sanguigno. I possibili valori di un tratto, capelli biondi piuttosto che neri, vengono detti *alleli*. Ogni gene possiede una posizione o *locus* all'interno del cromosoma.

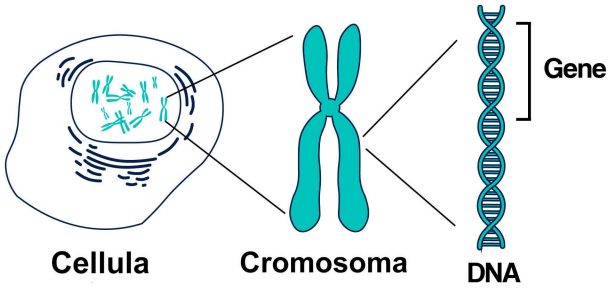

Figura 5.1: Struttura semplificata del cromosoma (dnaexpress.com)

L'insieme dei cromosomi di un organismo forma il *genoma*. Il *genotipo* rappresenta il particolare insieme di geni contenuto in un genoma. Due individui si dicono con lo stesso genotipo se hanno gli stessi genomi. Il genotipo dà origine al *fenotipo*, ovvero l'insieme delle caratteristiche fisiche visibili dell'individuo.
Il processo che avviene durante la riproduzione è detto crossing-over (in informatica parleremo di *crossover*), durante il quale coppie di cromosomi vengono combinate tra loro in modo da formare uno o due figli (*offspring*). Questi figli sono poi soggetti a mutazioni casuali (*mutation*) nelle quali alcuni bit di DNA vengono alterati a causa, ad esempio, di errori durante la copia.

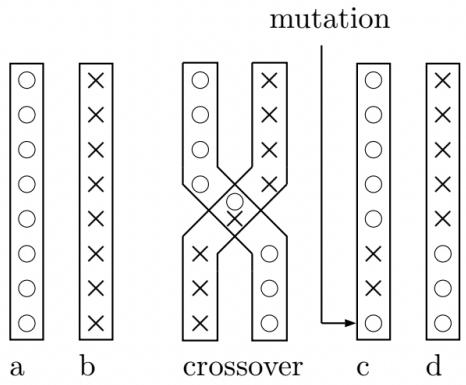

Figura 5.2: L'operazione di crossover tra i due cromosomi a e b crea i figli c e d. La freccia in c indica un errore di copia [26].

I concetti della biologia sopra presentati in modo molto semplificato vengono mappati in quelli che sono gli elementi di un GA applicato ad un problema di ottimizzazione combinatoria. La *popolazione* (il genoma) è costituita da un insieme di soluzioni (i cromosomi), ciascuna delle quali è codificata come una stringa di bit (il DNA). A seconda della rappresentazione, che verrà successivamente illustrata, un gene può essere rappresentato da un insieme di uno o più bit di questa stringa. I possibili valori assumibili sono gli alleli e rappresentano il dominio del particolare gene all'interno del problema analizzato. Nel caso del problema trattato in questo elaborato, il cromosoma rappresenta il piano delle attività schedulate del satellite (Capitolo 2), un gene rappresenta una DTO e gli alleli tutte le possibili DTOs che si potrebbero scegliere. Gli operatori di crossover e mutation possono essere implementati in maniera molto semplice rispettivamente come la ricombinazione di porzioni di stringa di due genitori diversi e la modifica di un valore scelto in modo casuale in una posizione (locus) anch'essa scelta in modo casuale.
Infine, la popolazione evolve un certo numero di iterazioni dette *generazioni*.

5.2 Struttura dell'algoritmo

La struttura dei GA è composta prevalentemente da una inizializzazione dei cromosomi e da un ciclo che fa evolvere le soluzioni.

- INIZIALIZZAZIONE: genera un insieme di soluzioni iniziali ammissibili in maniera casuale o tramite un euristico.
- LOOP: si iterano le seguenti operazioni fino a che non viene raggiunto il limite di generazioni:
    1. *Valutazione della fitness*: valuta la bontà di ciascuna soluzione tramite una determinata funzione di fitness.
    2. *Selezione dei genitori*: si scelgono, secondo diverse strategie, le coppie di soluzioni che faranno da genitori.
    3. *Crossover*: si utilizzano i genitori scelti nella fase precedente, per generare un determinato numero di figli, normalmente una o due nuove soluzioni per ogni coppia di genitori.
    4. *Mutation*: si modificano in maniera casuale le soluzioni generate. In tal modo si cerca di simulare le variazioni aleatorie che avvengono in genetica col fine di diversificare gli individui.
    5. *Aggiornamento della popolazione*: si sostituisce tutta o una parte della precedente popolazione utilizzando le nuove soluzioni create.
    6. *Torna al loop.*

5.2.1 La rappresentazione dell'informazione

La scelta della rappresentazione dei cromosomi è fondamentale per la buona riuscita dell'algoritmo e dipende strettamente dal tipo di problema che si sta affrontando. La prima rappresentazione in assoluto adoperata da Holland è quella binaria, i geni quindi assumevano i valori 0 e 1. Questo tipo di rappresentazione può essere adeguata per alcuni problemi, per esempio per il knapsack, si potrebbe assegnare infatti 1 se l'oggetto in considerazione è stato preso, e 0 se è stato lasciato.
Tuttavia la rappresentazione binaria non funziona per tutti i problemi, serve infatti trovare una codifica apposita per il problema trattato. Alcune in generale utilizzate sono stringhe di caratteri, numeri floating-point, array di numeri e matrici.

5.2.2 La funzione di fitness

Anche la funzione di fitness è strettamente legata al problema. Essa definisce quanto è "buona" una soluzione, restituendo un determinato valore. La definizione

di questa funzione è molto importante perché i cromosomi evolveranno in base ad essa. Un esempio tratto da Mitchell (1996) [27] è la seguente funzione:

$$f(x) = y + |\, sin(23y)\,| \quad \text{dove } 0 \leq y < \pi$$

In questo esempio, l'insieme delle soluzione corrisponde a tutti i valori reali che y può assumere tra 0 e π escluso.
Un esempio per il knapsack potrebbe essere il rapporto tra la somma dei profitti (p_i) e la somma dei pesi (w_i):

$$f(x) = \frac{\sum_i p_i}{\sum_i w_i}$$

o semplicemente la somma dei profitti:

$$f(x) = \sum_i p_i$$

La funzione di fitness deve poter accettare anche soluzioni non ammissibili perché gli **operatori** dei GA (crossover 5.2.5 e mutation 5.2.6), potrebbero crearne.

5.2.3 Strategie di selezione dei genitori

La selezione dei genitori (*parent selection*) è il procedimento durante il quale si scelgono le coppie di cromosomi da fare accoppiare tramite la fase successiva: il crossover. Le coppie creeranno uno o due figli, che erediteranno caratteristiche diverse da entrambi i genitori.
Esistono varie strategie per effettuare la scelta, di seguito ne sono elencate alcune.

Roulette Wheel Selection

Il metodo funziona appunto come la ruota in una roulette nella quale le "fette" dove la pallina può fermarsi sono proporzionali in ampiezza alla fitness degli individui della popolazione. Si supponga di avere una popolazione di quattro individui $x1, x2, x3$ e $x4$. Chiamiamo $p_f(x_i)$ la percentuale di fitness dell'i-esimo individuo. Una possibile situazione potrebbe essere la seguente.

x_1	x_2	x_3	x_4
5	45	25	25

Come si vede in Figura 5.3, è probabile che la pallina si fermi sull'area di x_2 piuttosto che su quella di x_1. Si ha pertanto che la probabilità per la quale un individuo x_i è scelto tra tutta la popolazione P è pari a $p_f(x_i) = \frac{f(x_i)}{\sum_{x \in P} f(x)}$.

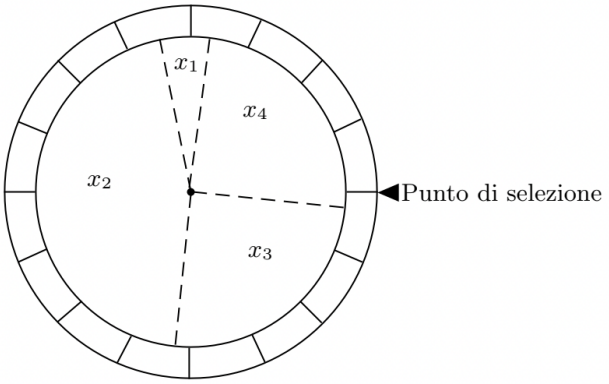

Figura 5.3: La roulette corrispondente alle percentuali in Sezione 5.2.3 (Immagine presa da [26]).

Tramite questa strategia, hanno più probabilità di essere scelti come genitori gli individui con la fitness migliore. Il fatto però che non si scelgano sempre e solo quelli (poiché si parla di probabilità) aiuta a diversificare l'*offspring* (figli generati) e di conseguenza lo spazio di ricerca. Il processo viene ripetuto finché non si genera il numero di genitori desiderato.
Una variante di questa tecnica molto comune e utilizzata, consiste nel scegliere il primo genitore con questo metodo, e il secondo in maniera casuale. Così facendo si aumenta ulteriormente la diversificazione degli individui e si favorisce una ricerca più ampia senza il rischio di generare cromosomi molto simili.

Sigma Scaling

Tipicamente la situazione iniziale in un GA è quella in cui si hanno pochi individui con buone caratteristiche e una grande varianza in termini di fitness all'interno della popolazione. Utilizzando metodi che selezionano i genitori in maniera proporzionale alla fitness, è molto probabile che nelle prime generazioni vengano scelti sempre gli stessi. Questo porta ad una poca diversificazione, cioè ad una inefficace ricerca nello spazio delle soluzioni e ad una abbastanza prevedibile immobilizzazione in ottimi locali. Questa situazione è conosciuta come convergenza prematura.
Per sopperire a questo problema sono stati proposti diversi metodi detti di *scaling* che non utilizzano direttamente la fitness dell'individuo ma considerano anche altri parametri globali come la fitness media e la varianza della popolazione. Uno di questi è il Sigma Scaling, dovuto a Forrest (1985) [15] e Goldberg (1989) [20]. La

fitness viene mappata in un *expected value* nel seguente modo:

$$E(x_i, t) = \begin{cases} 1 + \frac{f(x_i) - \overline{f(t)}}{2\sigma(t)} & \text{se } \sigma(t) \neq 0 \\ 1 & \text{se } \sigma(t) = 0 \end{cases}$$

Dove $f(x_i)$ è la fitness di x_i, $\overline{f(t)}$ è la fitness media della popolazione al tempo t e $\sigma(t)$ è la deviazione standard delle fitness della popolazione al tempo t.

Boltzmann Selection

Utilizzando il *Sigma Scaling* si mantiene la *selection pressure* (il grado per il quale individui con fitness alta sono più abilitati ad avere figli) costante durante il corso dell'algoritmo. Lo scopo di *Boltzmann Selection* è quello di variare questa pressione in base alle necessita. Ad esempio all'inizio dell'esecuzione si potrebbe voler lasciare che gli individui più *fit* vengano scelti con uno stesso rateo di quelli meno *fit* ma verso la fine del GA concentrarsi solamente sui migliori. L'approccio utilizzato qui ricorda un po' quello del *Simulated Annealing* (Sezione 4.1) nel quale inizialmente sono permesse anche soluzioni che peggiorano quella corrente ma man mano che la temperatura scende aumenta la probabilità che vengano scelte solamente quelle migliori. Un'implementazione potrebbe essere:

$$E(x_i, t) = \frac{e^{f(x_i)/T}}{\frac{\sum_{j=1}^{n} e^{f(x_j)/T}}{n}}$$

Dove T rappresenta la temperature e n la dimensione della popolazione.

Rank Selection

In questa tecnica gli individui della popolazione vengono classificati in base alla propria fitness. Il *linear ranking* inizialmente proposto da Baker (1985) [4] ordina gli individui della popolazione per fitness crescente e calcola l'*expected value* servendosi del ranking piuttosto che della fitness. Utilizzando la posizione in classifica è possibile stabilire se una soluzione e più buona di un'altra senza però conoscere l'effettivo rapporto tra le due. Questo permette di evitare una convergenza prematura ma in alcuni casi può rallentare la ricerca di soluzioni con fitness alta, proprio perché gli individui non vengono scelti proporzionalmente a quest'ultima.

5.2.4 Elitismo

I processi dei GA non garantiscono in alcun modo che i migliori individui sopravvivano alla generazione successiva. Per evitare che essi vengano persi durante il

crossover o che una mutazione possa peggiorarli, si può utilizzare una tecnica introdotta da De Jong (1975) [11] chiamata **elitismo**. Essa consiste nel copiare una piccola porzione delle migliori soluzioni direttamente nella generazione successiva senza alcun modifica. Questa scelta permette di evitare che da una generazione all'altra il valore della miglior fitness diminuisca. Diverse ricerche hanno decretato un notevole miglioramento nelle performance dei GA a fronte dell'utilizzo di questa tecnica.

5.2.5 Crossover

L'operatore di crossover (chiamato anche ricombinazione) è usato per ricombinare stocasticamente le informazioni genetiche dei genitori e per creare nuovi figli. Questa operazione emula la riproduzione sessuata degli organismi in biologia. Questo operatore genetico dipende strettamente dal tipo di rappresentazione delle informazioni che si è scelto per il problema trattato (Sezione 5.2.1). Gli esempi proposti successivamente utilizzeranno la classica rappresentazione binaria.

Single-Point Crossover

Il *single-point* crossover è la procedura più semplice di ricombinazione. Si sceglie casualmente un indice all'interno dell'array booleano (che rappresentano le variabili decisionali), e si scambiano i due sub-array definiti dall'indice scelto.
Definiamo per esempio x e y come due genitori, $i = 3$ come l'indice al quale effettuare la ricombinazione, il crossover avverrebbe nel seguente modo:

$$x = \mathbf{010}0010111$$
$$y = 101\mathbf{0110001}$$
$$z = \mathbf{0100110001}$$

Dove z è il figlio originato dall'operazione di crossover.
Così facendo però, si favoriscono alcune posizioni di geni all'interno dei cromosomi e si garantisce poca esplorazione.

Multi-Point Crossover

Per sopperire questo problema, si può utilizzare un *two-points* crossover, nel quale si definiscono due indici, per esempio con $i = 3$ e $j = 7$, l'operazione di crossover diventerebbe:

$$x = \mathbf{010}0010\mathbf{111}$$
$$y = 101\mathbf{0110}001$$
$$z = \mathbf{0100110001}$$

In maniera analoga si può sfruttare il *multi-point* crossover usando n indici.

Uniform crossover

Un'ulteriore forma di crossover consiste nel scegliere ogni gene del cromosoma figlio, in maniera casuale tra quelli dei genitori. In questo modo, il figlio ha una probabilità uniforme di ereditare le caratteristiche dai genitori, rendendo così il fenomeno ulteriormente aleatorio.

5.2.6 Mutazione

La mutazione aiuta a mantenere un ragionevole livello di diversità tra i soggetti della popolazione, questo permette di uscire dalle regioni sub-ottimali (minimi o massimi locali) dello spazio delle soluzioni e favorendo così una migliore esplorazione. Se il crossover è l'operatore più utilizzato, poichè viene eseguito praticamente ad ogni iterazione, la mutazione al contrario è sempre stata utilizzata molto meno. Alcuni studiosi hanno rilevato che la potenza di quest'ultima è stata sottostimata (vedi Muhlenbein (1992) [28]) e sarebbe opportuno sfruttarla per raggiungere zone dello spazio delle soluzioni altrimenti non facilmente raggiungibili.
La versione più semplice della mutazione è l'**inversione**, che consiste in una modifica aleatoria all'interno di un cromosoma. In particolare vengono modificati alcuni geni rimpiazzando i loro valori con altri sempre all'interno del dominio. L'inversione è molto utilizzata nei problemi rappresentabili in forma binaria poiché si tratta semplicemente di invertire lo stato della variabile binaria.

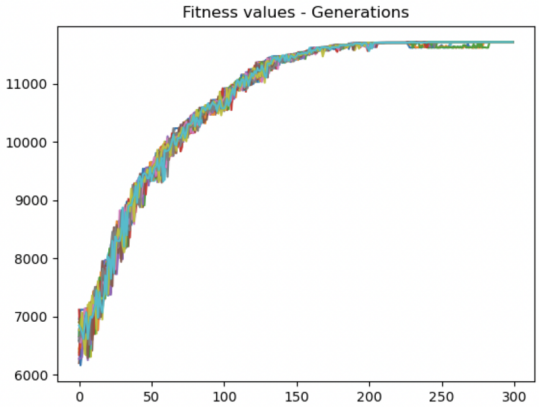

Figura 5.4: Esempio dell'andamento delle funzioni di fitness dei cromosomi senza applicare la mutazione.

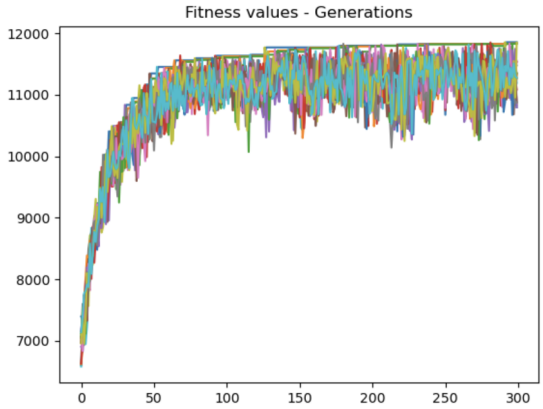

Figura 5.5: Esempio dell'andamento delle funzioni di fitness dei cromosomi applicando la mutazione.

In Figura 5.4 si nota che l'algoritmo converge molto presto ad una soluzione finale, non visitando poche possibili soluzioni. Al contrario, come si può vedere in Figura 5.5, l'algoritmo non converge prematuramente ma continua a variare le soluzioni visitate. La mutazione causa un incremento della frequenza nelle funzioni di fitness dei cromosomi, questo perché modificando i valori dei geni in modo aleatorio, si ottiene un grosso miglioramento o un grosso peggioramento, aumentando così di gran lunga lo spazio di ricerca. Si può notare anche che si arriva più facilmente ad una soluzione finale migliore.

Applicando gli operatori di crossover e mutazione, si possono generare soluzioni *infeasible*, ovvero soluzioni che non rispettano i vincoli che posti dal problema. Questo è molto frequente a causa dei fenomeni aleatori che sfrutta questo tipo di metaeuristico. Se non si prevede di utilizzare soluzioni infeasible, è necessario implementare delle procedure di *riparazione* che a fine generazione (iterazione), servano a rendere di nuovo ammissibili le soluzioni che non lo sono più.

5.2.7 Maturazione

Tra tutti questi fenomeni stocastici è bene infine inserire qualche procedura deterministica. La maturazione infatti consiste nel migliorare le caratteristiche di una data soluzione in maniera logica piuttosto che casuale. Questo processo viene anche chiamato **ricerca locale**, in quanto si applicano modifiche locali che non interessano la popolazione nella sua totalità. Gli algoritmi di ricerca locale sono strettamente legati al problema che si ha da risolvere e ne esistono di diversi tipi. É da tener presente che, le procedure di maturazione migliorano notevolmente le

soluzioni, ma d'altra parte tendono a uniformare la popolazione con il rischio di impedire l'esplorazione di zone diverse dello spazio di ricerca.

5.2.8 La definizione della nuova popolazione

Una volta definiti i figli è necessario sostituire totalmente o parzialmente gli individui della precedente generazione. L'idea originale di Holland consiste nella sostituzione della popolazione in blocco con i figli generati da tale popolazione. Altre versioni suggeriscono di rimpiazzare un membro casuale della popolazione non appena viene generato un figlio. De Jong (1975) [11] ha introdotto l'idea di un *gap* generazionale per evitare che genitori e figli si sovrappongano. Questo gap è stato supportato da studi empirici i quali indicano migliori performance quando diverse generazioni rimangono separate. I metodi piu utilizzati in pratica sono la sostituzione totale, la sostituzione dei k peggiori individui con i k migliori figli oppure la sostituzione di k scelti casualmente con k anch'essi scelti in modo casuale. Come già detto più volte in precedenza, una popolazione molto diversificata è preferibile in quanto permette di esplorare uno spazio delle soluzioni più vasto.

5.3 Vantaggi

I GA hanno avuto successo grazie ad alcuni punti di forza che lo caratterizzano e lo fanno spiccare tra tutti gli altri metaeuristici. In questa breve sezione ne sono descritti i principali.

Struttura semplice. I genetici vantano una struttura composta da operazioni e fasi molto semplici. Una volta analizzato bene il problema, gli operatori genetici e le altre procedure euristiche si riescono a implementare facilmente. Resta tuttavia un algoritmo che ha bisogno di diverse prove empiriche per ottenere il massimo dei risultati.

Embarassing parallel. L'algoritmo genetico presenta una struttura estremamente facile da parallelizzare, infatti la maggior parte delle operazioni da eseguire sono indipendenti e le soluzioni non dipendono mai dalle altre (se non durante il crossover). La situazione ideale è quella in cui si ha una unità di calcolo per ciascun individuo così da processare un'intera popolazione al tempo di una singola soluzione.

Problem-indipendent. I GA forniscono un paradigma flessibile che permette la creazione di algoritmi adatti ad una vasta varietà di problemi. Inoltre, è facilmente modulabile e personalizzabile in base al problema che si deve risolvere.

5.4 Varianti

Come già detto, la definizione degli algoritmi genetici ha introdotto un paradigma di design per la creazione di algoritmi metaeuristici, da esso quindi si ispirano altri algoritmi con caratteristiche simili e diverse allo stesso tempo. In seguito sono descritti brevemente due di questi.

5.4.1 Algoritmo memetico

L'algoritmo memetico (MA) è un estensione del classico GA e sfrutta la teoria filosofica di Richard Dawkins (2017) [10]. Secondo quest'ultima, la conoscenza è suddivisa in piccoli blocchi chiamati *memi*, che si propagano, per imitazione, da un individuo all'altro. Possono assumere la forma di un'idea, uno stile o un comportamento, e si diffondono tramite le relazioni interpersonali o attraverso i mezzi di comunicazione di massa. I memi, così come i geni, hanno le proprietà di replicarsi, mutare, ed essere soggetti a selezione naturale.
Nella società umana, alcuni memi non sono abbastanza interessanti e tendono a morire in un breve periodo di tempo. Altri invece sono più "forti" e si propagano maggiormente rimanendo in vita per più tempo (come ad esempio i gossip).
Questa affascinante interpretazione della cultura umana ha ispirato Moscato e Norman alla fine degli anni '80 [29], a introdurre gli algoritmi memetici. Nella loro definizione iniziale, i MA erano una modifica degli algoritmi genetici che impiegavano anche un operatore di ricerca locale per affrontare il problema del commesso viaggiatore (TSP). Altri importanti contributi agli algoritmi memetici sono stati dati da Hart (2005) [21].
Lo sviluppo di tecniche moderne che si ispirano alla diffusione culturale ma non rientrano nella definizione di MA ha introdotto il concetto di *Memetic Computing* (MC). Quest'ultimo è stato definito come "... un paradigma che utilizza la nozione di meme come unità di informazioni codificate in rappresentazioni computazionali allo scopo di risolvere problemi". In altre parole, parte della comunità scientifica ha cercato di estendere il concetto di meme per la risoluzione dei problemi a qualcosa di più ampio e innovativo.

5.4.2 Algoritmo bionomico

L'algoritmo bionomico (BA) è stato proposto da Christofides e presentato alla conferenza di Savona del 1994 [8]. Il BA è una particolare variante del GA. Esso fornisce un metodo di ottimizzazione globale che va adattato al particolare problema da risolvere.
I BA e i GA condividono la maggior parte degli step, infatti agisce su una popolazione di soluzioni che viene evoluta di generazione in generazione.

Il BA al contrario del GA necessita obbligatoriamente di una qualche tecnica di ottimizzazione locale sulle soluzioni (fase di maturazione descritta in Sezione 5.2.7). Questo approccio, ora praticato anche all'interno dei GA, è stato utilizzato raramente fino agli ultimi anni 80 mentre oggigiorno è diventato lo standard. Le fasi per le quali un BA si differenzia da un GA sono essenzialmente la selezione dei genitori (Sezione 5.2.3) e di conseguenza il crossover (Sezione 5.2.5). Data una popolazione P ad una certa generazione g, i BA definiscono un grafo di similarità tra tutte le soluzioni appartenenti a P. Viene generato un grafo i cui vertici corrispondono alle soluzioni della popolazione. I vertici di due soluzioni vengono collegate da un arco se esse sono considerate simili in base ad una qualche *funzione di similarità*. Costruito il grafo, si va ad identificare un insieme indipendente (massimale) di soluzioni (tramite un euristico), cioè si vanno a prendere tutte quelle soluzioni per le quali non si ha un arco che le collega. Queste vengono identificate come primo insieme dei genitori, esse verranno poi combinate per ottenere un certo numero di figli. I BA ammettono la possibilità di aver l'insieme delle soluzioni di dimensione variabile.
Questa strategia introduce uno speciale criterio di diversificazione che permette di esplorare zone molto distanti dello spazio delle soluzioni ed evitare ottimi locali.

Capitolo 6

Implementazione

Per risolvere il problema PLATiNO (Capitolo 2) sono stati utilizzati entrambi gli approcci, matematico ed euristico, citati in questa tesi. Per ognuno di questi due si è prima sviluppata una soluzione per il problema parziale e successivamente una per il problema completo. Tutti gli algoritmi sono stati sviluppati in **Python**, grazie al quale si riesce ad eseguire operazioni complesse in poche righe di codice. In questo capitolo è descritta la struttura, il funzionamento e l'ingegneria del software degli algoritmi implementati, mentre nel capitolo successivo (Capitolo 7), saranno descritti e confrontati i risultati computazionali ottenuti.

6.1 Approccio matematico

Si è preferito partire con l'implementazione dell'algoritmo esatto per avere un'idea più chiara e dettagliata sul problema e sui tempi di risoluzione. Per la maggior parte dei problemi che si vogliono risolvere trovando l'ottimo matematico, conviene affidarsi a un solver commerciale ottimizzato e creato appositamente per risolvere problemi di questo tipo. In questo contesto è stato utilizzato **Gurobi Optimizer**, il quale offre algoritmi per la risoluzione di diversi tipi di problemi (LP, MILP, QP, ecc...).
Gurobi offre delle implementazioni avanzate e ottimizzate degli ultimi algoritmi utilizzati in quest'ambito tra cui *Simplesso*, *Branch-and-Cut* e *Parallel Barrier*. Inoltre fornisce delle APIs (Application Programming Interfaces) di alto livello per molti linguaggi tra cui Java, Python, .NET e C++, grazie alle quali è possibile creare e popolare il modello matematico.
In Listato A.1 è riportato il codice per la risoluzione del problema parziale. Questo semplice script funziona in maniera procedurale, come prima cosa vengono filtrate le DTOs scartando quelle che sovrappongono con le PAWs, poiché queste ultime hanno priorità. Successivamente si creano le variabili decisionali, che per questa

versione del problema, corrispondono alle sole DTOs. Successivamente, seguendo il modello matematico descritto nel Capitolo 3, viene popolata l'istanza della classe *Model* di Gurobi con funzione obiettivo, variabili decisionali e vincoli. Infine, una volta pronto il modello, viene avviato l'algoritmo di risoluzione.
Il codice utilizzato per risolvere il problema completo è riportato in Listato A.2. La sostanziale differenza rispetto allo script precedente, è che sono state introdotte delle nuove variabili decisionali z_{ji} che determinano se una DTO i viene scaricata durante la DLO j. Si suppone che le DLOs fornite non si sovrappongano tra di loro, inoltre vengono scartate previamente le DTOs che sopravvengono con le DLOs. Sono anche stati aggiunti i vincoli che riguardano gli scaricamenti, ma la struttura dello script rimane la stessa.

6.2 Approccio euristico

Per risolvere il problema euristicamente si è scelto l'algoritmo genetico (Capitolo 5) il quale non è proprio adatto a problemi molto vincolati, ma permette di effettuare una grande esplorazione nello spazio delle soluzioni.
Per quanto riguarda la progettazione dell'algoritmo, in seguito (Figura 6.1) è mostrato il diagramma UML che descrive le sue principali componenti con i campi e i metodi più rilevanti. Essendo Python un linguaggio debolmente tipizzato, sono stati definiti dei tipi personalizzati (che hanno il solo scopo di aiutare la comprensione del codice) tra cui DTO, presente anche nel diagramma.

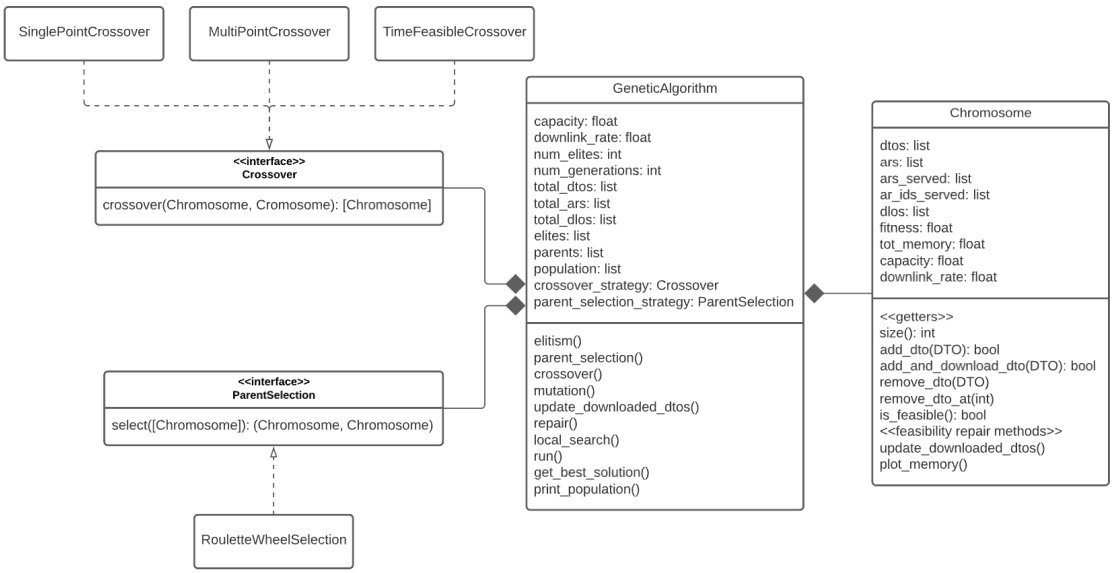

Figura 6.1: Diagramma UML di progettazione dell'algoritmo genetico

Per facilitare l'estensione del genetico a possibili varianti, si è scelto di utilizzare il design pattern **Strategy** per l'implementazione dei metodi di *parent selection* e di *crossover*. Attualmente per la selezione dei genitori è presente solo la versione **Roulette Wheel Selection**, che generalmente è una delle strategie migliori. Invece per quanto riguarda il crossover, sono stati implementati sia i metodi classici **Single Point** e **Multi Point Crossover**, sia uno personalizzato: **Ordered Crossover**, che verrà descritto successivamente.

6.2.1 Struttura dati del piano

All'interno di ogni soluzione `Chromosome` (Listato B.2 riporta le parti principali), le DTOs vengono memorizzate nella lista `dtos` che preliminarmente viene ordinata in base allo `start_time`, ovvero l'istante di inizio della DTO. Questo non solo permette di lavorare sempre con un piano temporalmente ordinato come nella realtà, ma offre anche il grosso vantaggio di effettuare l'operazione di ricerca nella lista in tempo **logaritmico** piuttosto che lineare. Infatti tenendo sempre la lista ordinata, la ricerca di una DTO all'interno del piano costa $O(logN)$. Per questa operazione è stata implementata un'apposita **ricerca binaria** riportata in Listato B.1.

Sempre all'interno di ogni `Chromosome`, oltre alla lista di `dtos` che rappresenta il piano di acquisizioni, è presente anche la lista `dlos`, per il piano degli scaricamenti. L'algoritmo implementato utilizza tutte le DLOs fornite dall'istanza del problema. Questo perché più il satellite ha opportunità di scaricare, maggiore sarà lo spazio disponibile per le acquisizioni successive, aumentando il valore di fitness finale. Come già detto, si ipotizza che le DLOs fornite non si sovrappongano tra di esse, anche perché se così fosse, l'algoritmo dovrebbe scegliere anche quali DLOs mettere nel piano e il problema sarebbe drasticamente più complesso. Inoltre, in una versione ancora più complicata del problema, si potrebbero utilizzare solo alcune partizioni di DLOs a favore di maggiori acquisizioni.
Anche la lista `dlos` è ordinata secondo lo `start_time` e all'interno di ogni suo elemento è presente una ulteriore lista: `downloaded_dtos`, contenente le DTOs scaricate da quella DLO.

Sempre all'interno di `Chromosome` è memorizzato un array chiamato `ars_served`. Questo contiene un valore booleano per ogni AR fornito dall'istanza del problema, infatti se il valore all'indice i è `True` allora l'AR con indice i è stato servito, ovvero c'è almeno una DTO all'interno del piano che la soddisfa.

6.2.2 Funzionamento dell'algoritmo

L'algoritmo vero e proprio parte invocando il metodo `run` della classe `GeneticAlgorithm` (Listato B.3). Esso itera per un numero prestabilito di volte (numero di **generazioni**) chiamando in ordine i seguenti metodi:

- `elitism`, definisce i cromosomi elites della generazione corrente.
- `parent_selection`, seleziona le coppie di genitori che verranno usate durante la fase di crossover.
- `crossover`, crea un nuovo array di cromosomi partendo dai genitori, sostituendoli ai non-elites della popolazione corrente.
- `mutation`, effettua degli scambi casuali tra DTOs all'interno del piano con altre fuori dal piano.
- `update_downloaded_dtos`, per i problemi con gli scaricamenti, popola l'array `downloaded_dtos` all'interno di ogni DLO di ogni cromosoma.
- `repair`, procedura di riparazione dei vincoli infranti nelle precedenti fasi.
- `local_search`, metodo deterministico che cerca di migliorare il valore di fitness attraverso l'aggiunta (e possibilmente lo scaricamento) di DTOs con alta priorità.

Funzione di fitness In questo caso la funzione di fitness è definita come la sommatoria delle priorità di tutte le DTOs all'interno del piano di acquisizioni. Il valore di fitness di ogni soluzione non viene calcolato ogni volta che lo si richiede, ma per motivi di costo viene aggiornato ogni volta che si effettua una modifica al piano.

Elitismo Durante la fase di elitismo (Sezione 5.2.4) viene scelto un numero di cromosomi pari a `num_elites` (Figura 6.1), i quali saranno gli *elites* della generazione corrente. Essi corrisponderanno a quelli con il valore di fitness più alto.

Selezione dei genitori Come già anticipato, per la selezione dei genitori è stato implementato il metodo **Roulette Wheel Selection** (Sezione 5.2.3, Listato 6.1), con la variante casuale sul secondo genitore. Infatti in ogni coppia il primo genitore viene scelto in base ad una probabilità basata sui valori di fitness mentre il secondo viene scelto casualmente.

Listato 6.1: RouletteWheelSelection.py

```
from random import choices, randint

from heuristic.genetic.Chromosome import Chromosome
from heuristic.genetic.parent_selection.ParentSelection import
    ParentSelection


class RouletteWheelSelection(ParentSelection):

    def __init__(self, population: [Chromosome]):
        population_tot_fitness = sum([chromosome.get_tot_fitness()
            for chromosome in population])
        self.selection_probs = [chromosome.get_tot_fitness() /
            population_tot_fitness
                                for chromosome in population]

    def select(self, population: [Chromosome]) -> (Chromosome,
        Chromosome):
        # picks the first parent based on fitness (roulette wheel
            method)
        parent1 = choices(population, weights=self.selection_probs
            , k=1)[0]

        # picks the second parent randomly within the population
        parent2 = population[randint(0, len(population) - 1)]
        return parent1, parent2
```

Crossover Per la fase di crossover inizialmente sono stati implementati i semplici *single-point* e *multi-point* (a due punti) crossover (Sezione 5.2.5). Tuttavia con questi metodi si generava un piano non ordinato che non era più coerente con il funzionamento della struttura dati utilizzata. Per sopperire a questo problema è stato creato un metodo di crossover apposito per mantenere ordinato il piano: **Ordered Crossover** (Listato 6.2). Questo metodo è un crossover a due punti nel quale il primo indice i viene scelto casualmente nel primo genitore. Mentre il secondo indice j corrisponde a quello della prima DTO, facente parte del secondo genitore, la quale inizia dopo il termine della DTO i.

Listato 6.2: OrderedCrossover.py

```
from random import choice

from heuristic.genetic.Chromosome import Chromosome
from heuristic.genetic.crossover.Crossover import Crossover


```

```
7  class OrderedCrossover(Crossover):
8
9      def crossover(self, parent1: Chromosome, parent2: Chromosome)
           -> [Chromosome]:
10         random = choice(range(0, len(parent1.dtos)))
11         stop_time = parent1.dtos[random]['stop_time']
12
13         i = 0
14         while i < len(parent2.dtos) and parent2.dtos[i]['
               start_time'] <= stop_time:
15             i += 1
16         return parent1.dtos[:random] + parent2.dtos[i:]
```

Mutation La fase di mutazione di questo algoritmo è una semplice **inversione**. Per ogni cromosoma si sostituisce il 5% delle DTOs nel piano con un altre DTOs scelte casualmente. Questa operazione serve ad introdurre nel piano alcune DTOs che probabilmente peggiorano il valore di fitness, ma aiutano ad esplorare soluzioni diverse, facilitando così la discesa dai massimi locali e aumentando le probabilità che l'algoritmo converga a un risultato migliore.

Fase di scaricamento Una volta applicati i principali operatori genetici sui cromosomi contenenti le DTOs ordinate, è il momento di assegnare alle DLOs quali DTOs all'interno del piano deve scaricare. Questo viene fatto tramite il metodo `update_downloaded_dtos`, seguendo la **policy di scaricamento** che dà la precedenza alle DTOs che occupano più memoria.

Fase di riparazione Questa fase serve a ripristinare i vincoli che sono stati infranti nelle fasi precedenti, infatti a fine generazione si devono avere tutte soluzioni *feasible*. Sono state implementate quattro procedure di riparazione per i seguenti vincoli: memoria, sovrapposizione, singola soddisfazione delle ARs, DTOs duplicati. Ognuna di queste va a rimuovere dal piano le DTOs che rendono il piano *infeasible*.

Local search Per il problema parziale si è sviluppato un metodo che prova a inserire le DTOs con priorità più alta. In particolare, all'inizio dell'algoritmo vengono ordinate tutte le DTOs in ordine descrescente rispetto all'importanza, poi durante la local search si cercano di inserire nel piano.
Il procedimento per il problema completo è analogo ma, oltre ad inserire la DTO, si tenta anche di scaricarla, quindi di inserirla in una delle liste `downloaded_dtos` all'interno di una DLO. Questo viene fatto tramite una simulazione, infatti inizialmente si controlla che la DTO che si tenta di aggiungere non soddisfi un AR

già servito e che non sovrapponga con altre DTOs già presenti nel piano. Successivamente la DTO viene inserita e poi viene ricalcolato il piano di scaricamenti, considerandola. Se il ricalcolo del piano termina con successo, ossia non vengono violati i vincoli, allora si va avanti con la modifica effettuata, altrimenti la DTO appena inserita viene rimossa.

Capitolo 7

Risultati

7.1 Istanze del problema

I risultati mostrati in questo capitolo fanno riferimento alle seguenti due istanze del problema, una per la versione parziale e l'altra per la versione completa.
La prima, mostrata in Figura 7.1, ha 3890 DTOs e il suo modello matematico presenta 202863 vincoli e 3715 variabili decisionali. La capacità del satellite è di 1481 MB.

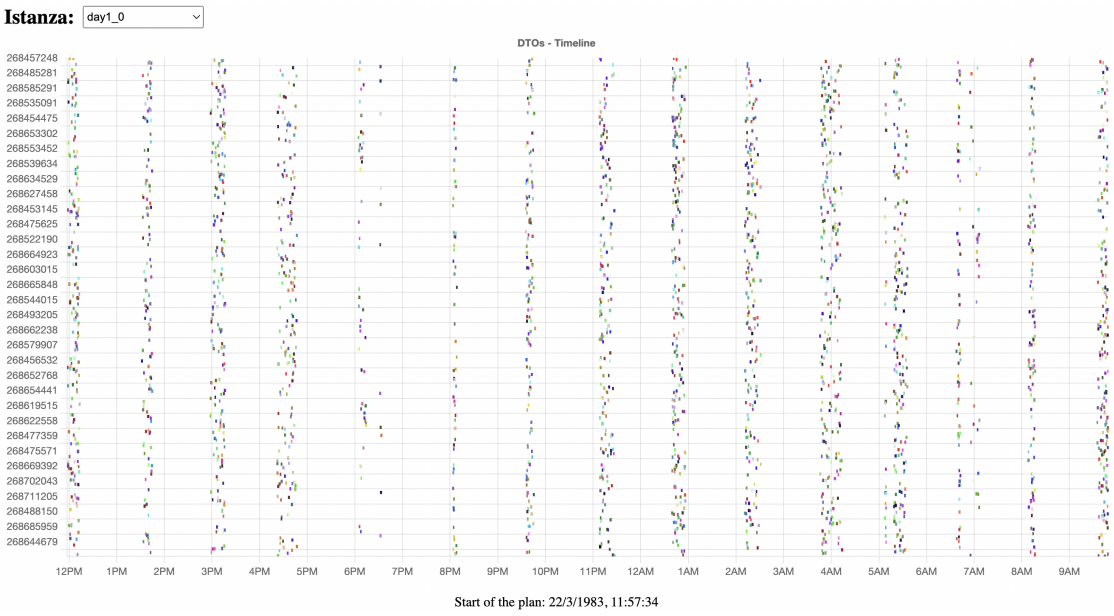

Figura 7.1: Istanza utilizzata per il problema parziale

La seconda istanza è mostrata in Figura 7.2, ha 1562 DTOs, 16 DLOs ed è

rappresentata con un modello di 25713 vincoli e 28116 variabili decisionali. Il satellite ha una capacità di memoria di 37 MB e una velocità di scaricamento di 0.064 MB/s.

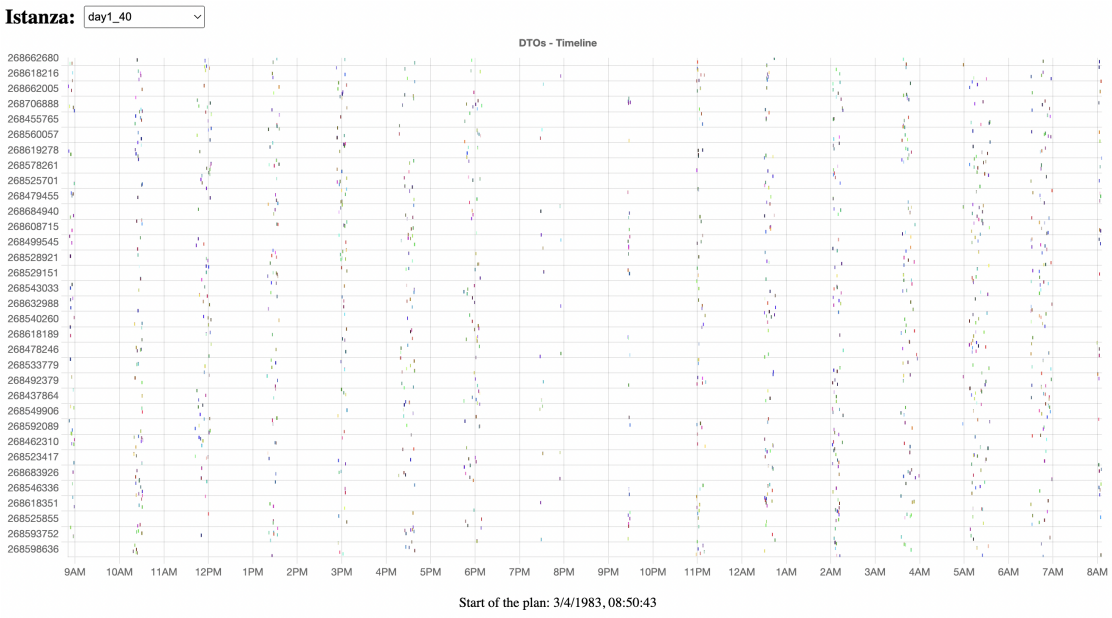

Figura 7.2: Istanza utilizzata per il problema completo

7.2 Risultati esatti

La risoluzione del modello tramite Gurobi riporta come risultato una priorità totale rispettivamente di 12438 per il problema parziale e di 12593 per il problema completo.
Il primo termina in un paio secondi. Nonostante il suo modello matematico abbia molti vincoli, questi non sono eccessivamente complessi e non inficiano molto anche grazie alle alte prestazioni del solver.
Il secondo termina in una quarantina di secondi, nonostante il numero di DTOs sia minore. Questo perché il modello presenta molte più variabili a causa degli scaricamenti. Infatti, se n è il numero di DTOs e m è il numero di DLOs, il numero totale di variabili decisionali del modello è uguale a $n + nm$.
Successivamente, in Figura 7.3 viene mostrato l'andamento della memoria occupata del satellite durante il piano delle attività determinato dal risultato dell'istanza del problema completo (Figura 7.2).

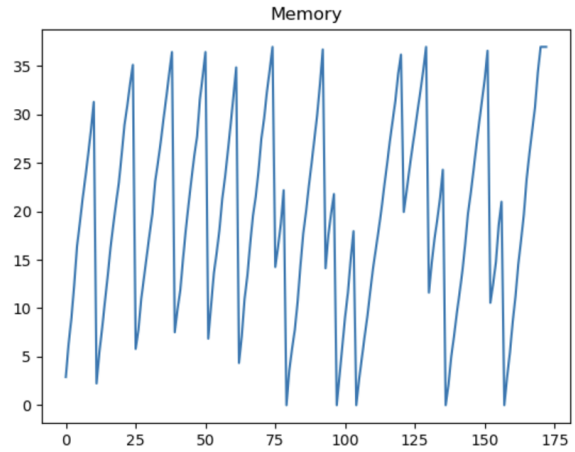

Figura 7.3: Andamento della memoria occupata del satellite.

7.3 Risultati euristici

Per analizzare le prestazioni e i risultati forniti dal genetico, l'algoritmo è stato eseguito 10 volte sia per il problema parziale sia per quello completo.
Si premette che l'algoritmo è stato configurato con 300 generazioni (iterazioni) e con una popolazione di 20 cromosomi, di cui 3 elites.
Successivamente è riportato un grafico che riporta i risultati di entrambi (Figura 7.4).

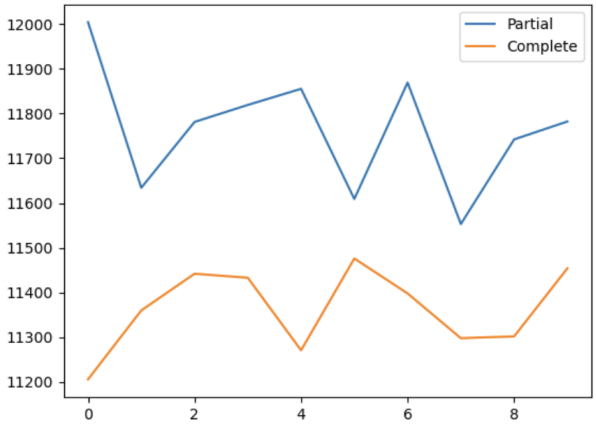

Figura 7.4: Risultati dell'algoritmo genetico ottenuti durante le diverse esecuzioni.

Per il problema parziale si ha che:

- Il miglior risultato ha una fitness di 12004.
- Il minor tempo di esecuzione equivale a 69 secondi, che è proprio quello impiegato dal test migliore.
- La media dei risultati ottenuta è 11765.
- In media l'algoritmo impiega 72 secondi.

Invece per il problema completo si hanno i seguenti risultati:

- Il miglior risultato ha una fitness di 11476.
- Il minor tempo di esecuzione equivale a 855 secondi, e anche in questo caso è dato dal test che dà il risultato migliore.
- La media dei risultati ottenuta è 11364.
- In media l'algoritmo impiega 884 secondi (15 minuti circa).

7.4 Confronto

Il genetico per il problema parziale ha dato un ottimo riscontro poiché il risultato migliore ottenuto dista davvero poco dal risultato esatto. Più precisamente, se calcoliamo il gap come $100 \cdot (z^* - z)/z^*$ dove z^* è la soluzione ottima del problema e z è la migliore soluzione ottenuta dall'algoritmo genetico, si ottiene un valore del 3.49%. Inoltre impiega pochissimo tempo di esecuzione, seppur più dell'algoritmo matematico.
Anche il genetico per l'istanza del problema completo restituisce un risultato non troppo distante dall'esatto. In particolare, si ha un gap del 8.87%. Tuttavia i tempi di esecuzione sono molto più lunghi rispetto al problema parziale a causa della costruzione del piano di scaricamenti. Infatti, il punto cruciale che causa l'innalzamento di questo tempo è la *local search*, che cerca di migliorare il piano simulando un'acquisizione e uno scaricamento, scorrendo tutto il piano, e controllando di non violare nessun vincolo.
Si precisa che questo è dovuto anche al fatto che l'algoritmo implementa la Downlink Policy che scarica prima le DTOs con memoria maggiore. Se si fosse più flessibili sulla policy, l'algoritmo impiegherebbe molto meno tempo.

Capitolo 8

Conclusioni

Lo scopo di questa tesi progettuale era di confrontare, in termini di risultati e prestazioni, algoritmi esatti ed euristici, cercando di valorizzare l'utilità di questi ultimi in determinati scenari complessi. Il problema preso in considerazione, essendo molto vincolato, ha messo a dura prova l'algoritmo genetico, ma non il solver utilizzato per l'algoritmo esatto. Infatti quest'ultimo, con le istanze del problema descritte, ci permette di trovare l'ottimo in pochi secondi.
Per questo tipo di problema è davvero difficile superare le prestazioni di un solver commerciale così performante. Fornendo istanze di maggiori dimensioni, il solver non riesce a convergere all'esatto in tempi ragionevoli a causa dell'esplosione del costo computazionale, ma riesce comunque in poco tempo a fornire un buon risultato estremamente vicino al miglior upper bound trovato.
Tuttavia i risultati ottenuti dal genetico sono abbastanza buoni e relativamente vicini alla soluzione ottima.
Bisogna poi spezzare una lancia a favore dell'algoritmo genetico. Difatti si è dovuto scontrare con un solver che ha utilizzato in parallelo tutti e 10 i core disponibili sulla macchina dove sono stati effettuati i test. Il genetico invece non è stato parallelizzato, di conseguenza ha avuto un notevole svantaggio in termini di performance.
Inoltre, gli algoritmi euristici rimangono al 100% personalizzabili in base al problema e ad ogni sua istanza. Per esempio, se le esigenze dovessero richiedere tempi ancora minori si potrebbero adoperare tecniche di *pruning* e di accelerazione, volte a semplificare e velocizzare le procedure computazionalmente più onerose dell'algoritmo, ad esempio la ricerca locale.
I test effettuati con questo algoritmo fanno capo ad una singola configurazione, ma nulla vieta che, attraverso prove empiriche e piccole modifiche, si possano ottenere risultati migliori e tempi di esecuzione minori (ad esempio parallelizzando).
Il progetto contenente gli algoritmi descritti in questo elaborato si può trovare su Github.com/Mala1180/satellites-optimization-algorithms.

Appendice A

Algoritmo esatto

A.1 Problema parziale

Listato A.1: Algoritmo esatto per il problema parziale

```
# Read and solve model
model = gp.Model()

print("Prepare variables and constraints...")
start = time.time()

# add the decision variables to the model
dtos_variables = list(model.addMVar((DTOS_NUMBER,), vtype=GRB.
    BINARY, name="DTOs"))

grouped_dtos = dict()

# for each couple of dtos which overlap add a constraint
for i1, dto1 in enumerate(dtos):
    for i2, dto2 in enumerate(dtos):
        if overlap(dto1, dto2) and dto1 != dto2:
            # add overlapping constraints
            model.addConstr(dtos_variables[i1] + dtos_variables[i2
                ] <= 1,
                            f"Overlapping constraint for DTOs {
                                dto1['id']} and {dto2['id']}")

    if dto1['ar_id'] not in grouped_dtos.keys():
        grouped_dtos[dto1['ar_id']] = [dto1]
    else:
        grouped_dtos[dto1['ar_id']].append(dto1)

# add the single satisfaction constraints
for ar_id in grouped_dtos.keys():
```

```
    model.addConstr(gp.quicksum([dtos_variables[dtos.index(dto_)]
                                 for dto_ in grouped_dtos[ar_id]])
                                     <= 1,
                    f"Single satisfaction constraint for {ar_id}")

# add the memory constraint
model.addConstr(gp.quicksum([memories[i] * dtos_variables[i]
                             for i in range(DTOS_NUMBER)]) <=
                                CAPACITY,
                "Memory constraint")
# set objective function to maximize dtos priority
model.setObjective(gp.quicksum([priorities[i] * dtos_variables[i]
                                for i in range(DTOS_NUMBER)]), GRB
                                    .MAXIMIZE)

end = time.time()
print("Preparation terminated in ", end - start)

# solve model
print("Solve model...")
start = time.time()

model.optimize()
```

A.2 Problema completo

Listato A.2: Algoritmo esatto per il problema completo

```
# Read and solve model
model = gp.Model()

print("Prepare variables and constraints...")
start = time.time()

# add the decision variables to the model
dtos_variables = list(model.addMVar((DTOS_NUMBER,), vtype=GRB.
    BINARY, name="DTOs"))
dlos_variables = list(model.addMVar((DLOS_NUMBER,), vtype=GRB.
    BINARY, name="DLOs"))

z_ji = []
for index in range(DLOS_NUMBER):
    z_ji.append(list(model.addMVar((DTOS_NUMBER,),
                                   vtype=GRB.BINARY,
                                   name=f"DTOs downloaded in DLO {
                                       index}")))

```

```
grouped_dtos = dict()

for i1, dto1 in enumerate(dtos):
    # add overlapping constraints between dtos
    for i2, dto2 in enumerate(dtos):
        if overlap(dto1, dto2) and dto1 != dto2:
            model.addConstr(dtos_variables[i1] + dtos_variables[i2] <= 1,
                            f"Overlapping constraint for DTOs {dto1['id']} and {dto2['id']}")

    # add overlapping constraints between dtos and dlos
    # for dlo_index, dlo in enumerate(dlos):
    #     if overlap(dto1, dlo):
    #         model.addConstr(dtos_variables[i1] + dlos_variables[dlo_index] <= 1,
    #                         f"Overlapping_constraint_between_DTO_{dto1['id']}_and_DLO_{dlo_index}")

    if dto1['ar_id'] not in grouped_dtos.keys():
        grouped_dtos[dto1['ar_id']] = [dto1]
    else:
        grouped_dtos[dto1['ar_id']].append(dto1)

# add the single satisfaction constraints
for ar_id in grouped_dtos.keys():
    model.addConstr(gp.quicksum([dtos_variables[dtos.index(dto_)] for dto_ in grouped_dtos[ar_id]]) <= 1,
                    f"Single satisfaction constraint for AR {ar_id}")

# add the taken memory constraints
satellite_memories = [gp.quicksum([memories[i] * dtos_variables[i] for i, dto in enumerate(dtos)
                                   if dto['stop_time'] < dlos[0]['start_time']])]
model.addConstr(satellite_memories[0] <= CAPACITY,
                f'Memory constraint DLO 0')

for j in range(1, DLOS_NUMBER):
    satellite_memories.append(
        satellite_memories[j - 1]
        - gp.quicksum([memories[i] * z_ji[j - 1][i] for i in range(DTOS_NUMBER)])
        + gp.quicksum([memories[i] * dtos_variables[i] for i, dto in enumerate(dtos)
                       if dto['start_time'] > dlos[j - 1]['stop_time']
```

```
                        and dto['stop_time'] < dlos[j]['start_time']])
    )

    model.addConstr(satellite_memories[j] <= CAPACITY,
                    f'Memory constraint DLO {j}')

# # add DTO selected in plan constraint
# for j in range(DLOS_NUMBER):
#     for i in range(DTOS_NUMBER):
#         model.addConstr(z_ji[j][i] <= dtos_variables[i],
#                         f'DTO_selected_in_plan_constraint_DLO:{j}_DTO:{i}')
#
# # add single downlink constraint
# for i in range(DTOS_NUMBER):
#     model.addConstr(gp.quicksum([z_ji[j][i] for j in range(DLOS_NUMBER)]) <= 1,
#                     f'Single_downlink_constraint_DTO:{i}')

# last two commented constraints can be reduced to the next one
for i in range(DTOS_NUMBER):
    model.addConstr(gp.quicksum([z_ji[j][i] for j in range(DLOS_NUMBER)]) <= dtos_variables[i],
                    f'Single downlink constraint and post acquisition for DTO {i}')

# add downloaded memory constraint
for j in range(DLOS_NUMBER):
    model.addConstr(gp.quicksum([memories[i] * z_ji[j][i] for i in range(DTOS_NUMBER)]) <=
                    DOWNLINK_RATE * (dlos[j]['stop_time'] - dlos[j]['start_time']),
                    f'Downloaded memory constraint DLO {j}')

# add time constraint
for j in range(DLOS_NUMBER):
    for i in range(DTOS_NUMBER):
        if dlos[j]['start_time'] < dtos[i]['stop_time']:
            model.addConstr(z_ji[j][i] == 0, f'Time constraint for DTO {i} DLO {j}')

# set objective function to maximize dtos priority
model.setObjective(gp.quicksum([priorities[i] * dtos_variables[i] for i in range(DTOS_NUMBER)]), GRB.MAXIMIZE)

end = time.time()
print("Preparation terminated in ", end - start)

```

```
94 # solve model
95 print("Solve model...")
96 start = time.time()
97
98 model.optimize()
```

Appendice B

Algoritmo euristico

B.1 Ricerca binaria

Listato B.1: Ricerca binaria sul piano di DTOs

```
def binary_search(dto, plan) -> int:
    """ Iterative implementation of binary search.
        Returns index of dto in the given plan, -1 if not found
           """
    low = 0
    high = len(plan) - 1
    while high >= low:
        mid = (high + low) // 2

        # If element is present at the middle itself
        if plan[mid] == dto:
            return mid

        # If element is smaller than mid, then it can only be
           present in left sub-array
        if dto['start_time'] > plan[mid]['start_time']:
            low = mid + 1

        # Else the element can only be present in right sub-array
        else:
            high = mid - 1

    # Element is not present in the array
    return -1
```

B.2 Chromosome.py

Listato B.2: Chromosome.py

```
class Chromosome:
    """ A class that represents a possible solution of
        GeneticAlgorithm class """

    def __init__(self, capacity: float, ars: [AR], dtos: [DTO] =
        None,
                 tot_dlos: [DLO] = None, downlink_rate: float =
                     None) -> None:
        """ If no argument is given, creates an empty solution,
            otherwise creates a solution with given DTOs """
        if dtos is None:
            dtos = []
        if tot_dlos is None:
            tot_dlos = []
        # Loads DTOs
        self.dtos: [DTO] = deepcopy(dtos)
        self.dtos = sorted(self.dtos, key=lambda dto_: dto_['
            start_time'])
        # Loads ARs
        self.ars: [AR] = deepcopy(ars)
        # Loads DLOs
        self.dlos: [DLO] = []
        if tot_dlos is not None:
            self.dlos = deepcopy(tot_dlos)

        self.dlos = sorted(self.dlos, key=lambda dlo_: dlo_['
            start_time'])

        if len(self.dtos) == 0:
            self.ar_ids_served: [int] = []
            self.ars_served = np.full(len(ars), False)
        else:
            self.ar_ids_served: [int] = [dto['ar_id'] for dto in
                self.dtos]
            self.ars_served = np.isin(list(map(lambda ar: ar['id'
                ], self.ars)),
                                      self.ar_ids_served)

        self.fitness: float = sum(self.get_priorities())
        self.tot_memory: float = sum(self.get_memories())

        self.capacity: float = capacity
        self.downlink_rate: float = downlink_rate

    def add_dto(self, dto: DTO) -> bool:
        """ Adds a DTO to the solution in start time order,
            updates total memory, fitness and ARs served.
```

```
39              Returns True if the insertion """
40          if self.ars_served[dto['ar_index']]:
41              return False
42
43          index = find_insertion_point(dto, self.dtos)
44          self.dtos.insert(index, dto)
45          self.tot_memory += dto['memory']
46          self.fitness += dto['priority']
47          self.ars_served[dto['ar_index']] = True
48          self.ar_ids_served.append(dto['ar_id'])
49          return True
50
51      def add_and_download_dto(self, dto: DTO) -> bool:
52          """ Tries to add and download a DTO to the solution,
53          returns True if the insertion is successful """
54          if len(self.dlos) == 0:
55              raise Exception("This method works only with downlink
                    problems")
56
57          # Checks if AR of the DTO is already served, and if DTO is
                already in the plan
58          if self.ars_served[dto['ar_index']]:
59              return False
60
61          # Checks if the DTO would overlap with another DTO
62          insertion_index = find_insertion_point(dto, self.dtos)
63          if insertion_index == 0:
64              if overlap(dto, self.dtos[0]):
65                  return False
66          elif insertion_index == len(self.dtos):
67              if overlap(dto, self.dtos[-1]):
68                  return False
69          else:
70              if overlap(dto, self.dtos[insertion_index - 1]) or
                    overlap(dto, self.dtos[insertion_index]):
71                  return False
72
73          backup_dlos = deepcopy(self.dlos)
74          self.add_dto(dto)
75
76          memory: float = 0
77          added, downloaded, term_condition = False, False, True
78          success: bool = True
79          dtos_in_memory: [DTO] = []
80          i: int = 0
81          j: int = 0
82          while i < len(self.dtos) and success:  # and
                term_condition:
83              # if the DTO comes before the DLO j, sum its memory
```

```
84              if self.dtos[i]['stop_time'] < self.dlos[j]['
                    start_time']:
85                  memory = memory + self.dtos[i]['memory']
86                  dtos_in_memory.append(self.dtos[i].copy())
87                  if dto == self.dtos[i]:
88                      added = True
89                  # if memory exceed because the new DTO is added,
                        stop iterating and return False
90                  if memory > self.capacity:
91                      success = False
92                  i += 1
93              else:
94                  # the DLO j downloads all DTOs that were already
                        there
95                  memory_downloaded: float = 0
96                  if not added:
97                      for dto_ in self.dlos[j]['downloaded_dtos']:
98                          memory -= dto_['memory']
99                          memory_downloaded += dto_['memory']
100                         dtos_in_memory.remove(dto_)
101                 else:
102                     self.dlos[j]['downloaded_dtos'] = []
103                     dtos_in_memory.sort(key=lambda dto__: dto__['
                            memory'], reverse=True)
104                     z: int = 0
105                     while z < len(dtos_in_memory):
106                         if self.is_dto_downloadable(dtos_in_memory
                                [z], self.dlos[j], memory_downloaded):
107                             self.dlos[j]['downloaded_dtos'].append
                                    (dtos_in_memory[z].copy())
108                             memory -= dtos_in_memory[z]['memory']
109                             memory_downloaded += dtos_in_memory[z
                                    ]['memory']
110                             dtos_in_memory.remove(dtos_in_memory[z
                                    ].copy())
111                             z -= 1
112                         z += 1
113                 j += 1
114
115         if not success:
116             self.remove_dto(dto)
117             self.dlos = backup_dlos
118
119         if DEBUG and not self.is_feasible():
120             raise Exception("Plan is not feasible")
121
122         return success
123
124     def remove_dto(self, dto: DTO) -> bool:
```

```
125         """ Removes a DTO from the solution """
126         index = binary_search(dto, self.dtos)
127         if index == -1:
128             return False
129         return self.remove_dto_at(index)
130 
131     def remove_dto_at(self, index: int):
132         """ Removes the DTO at the given index, raises an
                exception if the index is out of bounds """
133         if index < 0 or index >= len(self.dtos):
134             print(f'Index:{index}, len(self.dtos):{len(self.dtos)}
                    ')
135             raise IndexError("Index out of range")
136         dto = self.dtos[index]
137         self.ar_ids_served.remove(dto['ar_id'])
138         self.dtos.pop(index)
139         self.tot_memory -= dto['memory']
140         self.fitness -= dto['priority']
141 
142         if dto['ar_id'] not in self.ar_ids_served:
143             self.ars_served[dto['ar_index']] = False
144 
145         for dlo in self.dlos:
146             if dto in dlo['downloaded_dtos']:
147                 dlo['downloaded_dtos'].remove(dto)
148 
149     def is_constraint_respected(self, constraint: Constraint) ->
            bool:
150         """ Returns true if the solution respects the given
                constraint, false otherwise """
151         if constraint == Constraint.MEMORY:
152             if len(self.dlos) == 0:  # if problem is relaxed
153                 return self.get_tot_memory() < self.capacity
154             else:  # if problem includes down-links
155                 memory: float = 0
156                 i: int = 0
157                 j: int = 0
158 
159                 while i < len(self.dtos):
160                     # if the DTO comes before the DLO j, sum its
                            memory
161                     if self.dtos[i]['stop_time'] < self.dlos[j]['
                            start_time']:
162                         memory = memory + self.dtos[i]['memory']
163                         # if memory exceed because the new DTO is
                                added, stop iterating and return False
164                         if memory > self.capacity:
165                             return False
166                         i += 1
```

```
167                     else:
168                         for dto in self.dlos[j]['downloaded_dtos']:
169                             memory = memory - dto['memory']
170
171                         if memory < 0:
172                             return False
173                         j += 1
174
175                 return True
176
177         elif constraint == Constraint.OVERLAP:
178             for index in range(self.size() - 1):
179                 if overlap(self.dtos[index], self.dtos[index + 1]):
180                     return False
181             return True
182
183         elif constraint == Constraint.SINGLE_SATISFACTION:
184             return len(self.get_ars_served()) == len(set(self.get_ars_served()))
185
186         elif constraint == Constraint.DUPLICATES:
187             return len(self.get_dto_ids()) == len(set(self.get_dto_ids()))
188
189     def is_dto_downloaded(self, dto: DTO) -> bool:
190         """ Returns true if the given DTO is downloaded in the solution """
191         for dlo in self.dlos:
192             index = binary_search(dto, dlo['downloaded_dtos'])
193             if index != -1:
194                 return True
195         return False
196
197     def is_dto_downloadable(self, dto: DTO, dlo: DLO, memory_downloaded: float) -> bool:
198         """ Returns true if the given DTO is downloaded in the solution """
199         if dto['stop_time'] >= dlo['start_time']:
200             raise Exception('The DTO comes after the DLO')
201         return memory_downloaded + dto['memory'] <= self.downlink_rate * (dlo['stop_time'] - dlo['start_time'])
202
203     def update_downloaded_dtos(self):
204         memory: float = 0
205         downloadable_dtos: [DTO] = []
206
207         for dlo in self.dlos:
```

```
208             dlo['downloaded_dtos'] = []
209
210         for j, dlo in enumerate(self.dlos):
211             if j == 0:
212                 dtos_between_dlos: [DTO] = self.
                        get_dtos_between_dates(0, dlo['start_time'])
213             else:
214                 dtos_between_dlos: [DTO] = self.
                        get_dtos_between_dates(self.dlos[j - 1]['
                        stop_time'], dlo['start_time'])
215
216             memory = memory + sum(list(map(lambda dto_: dto_['
                    memory'], dtos_between_dlos)))
217             downloadable_dtos += dtos_between_dlos
218             downloadable_dtos.sort(key=lambda dto_: dto_['memory'
                    ], reverse=True)
219             memory_downloaded: float = 0
220             i: int = 0
221             while i < len(downloadable_dtos):
222                 dto = downloadable_dtos[i]
223                 if DEBUG and self.is_dto_downloaded(dto):
224                     raise Exception('The DTO is already downloaded
                            ')
225
226                 if self.is_dto_downloadable(dto, dlo,
                        memory_downloaded):
227                     dlo['downloaded_dtos'].append(dto.copy())
228                     memory = memory - dto['memory']
229                     memory_downloaded += dto['memory']
230                     downloadable_dtos.remove(dto)
231                     i -= 1
232                 i += 1
```

B.3 GeneticAlgorithm.py

Listato B.3: GeneticAlgorithm.py

```
1 class GeneticAlgorithm:
2     """ Implements the structure and methods of a genetic
          algorithm to solve satellite optimization problem """
3
4     def __init__(self, capacity, total_dtos, total_ars, total_dlos
          =None, downlink_rate=None,
5                  num_generations=300, num_chromosomes=20,
                       num_elites=3,
6                  parent_selection_strategy='roulette',
                       crossover_strategy='ordered'):
```

```
        """ Creates a random initial population and prepares data
            for the algorithm """
        if crossover_strategy == 'single':
            self.crossover_strategy: Crossover =
                SinglePointCrossover()
        elif crossover_strategy == 'multi':
            self.crossover_strategy: Crossover =
                MultiPointCrossover()
        elif crossover_strategy == 'ordered':
            self.crossover_strategy: Crossover = OrderedCrossover
                ()
        else:
            raise ValueError(f'Invalid crossover strategy: {
                crossover_strategy}, choose from "single" or "multi
                "')

        self.capacity = capacity
        self.downlink_rate = downlink_rate
        self.num_elites = num_elites
        print(f'Capacity: {capacity}')
        self.total_dtos: [DTO] = total_dtos.copy()
        for dto in self.total_dtos:
            dto['memory']: int = round(dto['memory'])
        self.total_ars: [DTO] = total_ars.copy()
        self.total_dlos: [DLO] = []
        if total_dlos is not None:
            self.total_dlos = total_dlos.copy()
            for dlo in self.total_dlos:
                dlo['downloaded_dtos'] = []

        self.ordered_dtos = sorted(total_dtos, key=lambda dto_:
            dto_['priority'], reverse=True)
        self.num_generations: int = num_generations
        self.elites: [Chromosome] = []
        self.parents: [(Chromosome, Chromosome)] = []
        self.fitness_history: [float] = []
        self.population: [Chromosome] = []

        for i in range(num_chromosomes):
            chromosome = Chromosome(self.capacity, total_ars.copy
                (),
                                    tot_dlos=self.total_dlos,
                                    downlink_rate=self.
                                        downlink_rate)
            shuffled_dtos: [DTO] = sample(self.total_dtos, len(
                self.total_dtos))

            for dto in shuffled_dtos:
```

```
45                  if chromosome.size() == 0 or chromosome.
                        keeps_feasibility(dto):
46                      chromosome.add_dto(dto)
47
48              self.population.append(chromosome)
49
50          if parent_selection_strategy == 'roulette':
51              self.parent_selection_strategy: ParentSelection =
                    RouletteWheelSelection(self.population)
52          else:
53              raise ValueError(f'Invalid parent selection strategy:
                    {parent_selection_strategy}, the only implemented '
54                               f'is roulette wheel')
55
56      def elitism(self):
57          """ Updates the elites for the current generation """
58          self.elites = sorted(self.population,
59                               key=lambda chromosome: chromosome.
                                    get_fitness(),
60                               reverse=True)[:self.num_elites]
61
62      def parent_selection(self):
63          """ Chooses and returns the chromosomes to make crossover
                with roulette wheel selection method """
64          self.parents = []
65          self.parent_selection_strategy = RouletteWheelSelection(
                self.population)
66          # finds number of couples equals to population length -
                elites length
67          for i in range(len(self.population) - len(self.elites)):
68              parents: (Chromosome, Chromosome) = self.
                    parent_selection_strategy.select(self.population)
69              self.parents.append(parents)
70
71      def crossover(self):
72          """ Makes crossover between each couple of parents, and
                replaces the entire population except
73              elites with the new offspring """
74          sons: [Chromosome] = []
75          for parent1, parent2 in self.parents:
76              son_dtos = self.crossover_strategy.crossover(parent1,
                    parent2)
77              son = Chromosome(self.capacity, self.total_ars.copy(),
                     son_dtos.copy(),
78                               self.total_dlos.copy(), self.
                                    downlink_rate)
79              sons.append(son)
80              if DEBUG and not son.is_constraint_respected(
                    Constraint.DUPLICATES):
```

```
81                  raise Exception('The solution contains duplicates'
                        )
82
83          self.population = self.elites + sons
84
85      def mutation(self):
86          """ Mutates randomly the 10% of each chromosome in the
                population """
87          for chromosome in list(set(self.population) - set(self.
                elites)):
88              # Replaces 5% of DTOs in the plan with new random DTOs
89              for _ in range(len(chromosome.dtos) // 20):
90                  new_dto = choice(self.total_dtos)
91                  chromosome.add_dto(new_dto)
92                  chromosome.remove_dto_at(randint(0, len(chromosome
                        .dtos) - 1))
93
94      def update_downloaded_dtos(self):
95          for chromosome in list(set(self.population) - set(self.
                elites)):
96              chromosome.update_downloaded_dtos()
97
98      def repair(self):
99          """ Repairs the population if some chromosomes are not
                feasible """
100         for chromosome in self.population:
101             if not chromosome.is_feasible(Constraint.OVERLAP):
102                 chromosome.repair_overlap()
103             if not chromosome.is_feasible(Constraint.
                    SINGLE_SATISFACTION):
104                 chromosome.repair_satisfaction()
105             if not chromosome.is_feasible(Constraint.DUPLICATES):
106                 chromosome.repair_duplicates()
107             if not chromosome.is_feasible(Constraint.MEMORY):
108                 while not chromosome.repair_memory():
109                     chromosome.update_downloaded_dtos()
110
111     def local_search(self):
112         """ Performs local search on the population. Tries to
                insert new DTOs in the plan. """
113         for chromosome in list(set(self.population) - set(self.
                elites)):
114
115             if len(self.total_dlos) == 0:
116                 for dto in self.ordered_dtos:
117                     if chromosome.keeps_feasibility(dto):
118                         chromosome.add_dto(dto)
119             else:
```

```
120                 dtos_to_insert = [dto for dto in self.ordered_dtos
                        [:len(self.ordered_dtos) // 2]
121                                     if dto not in chromosome.dtos]
122                 for dto in dtos_to_insert:
123                     # for dto in self.ordered_dtos:
124                     chromosome.add_and_download_dto(dto)
125
126     def run(self):
127         """ Starts the algorithm itself """
128         for i in range(self.num_generations):
129             print(f'Generation {i + 1}')
130             self.elitism()
131             self.parent_selection()
132             self.crossover()
133             self.mutation()
134             if len(self.total_dlos) > 0:
135                 self.update_downloaded_dtos()
136             self.repair()
137             self.local_search()
138             chromosome_fitness = [chromosome.get_fitness() for
                    chromosome in self.population]
139             self.fitness_history.append(chromosome_fitness)
140             print(f'Fitness: {self.fitness_history[i]}')
141
142     def get_best_solution(self) -> Chromosome:
143         """ Returns the best solution in the population after
                running of the algorithm """
144         return max(self.population, key=lambda chromosome:
                chromosome.get_fitness())
```

Bibliografia

[1] Luca Accorsi, Francesco Cavaliere, Michele Monaci, and Daniele Vigo. Daily planning of acquisitions and scheduling of dynamic downlinks for the platino satellite. Technical report, Department of Electrical, Electronic and Information Engineering "G. Marconi", University of Bologna CIRI ICT, University of Bologna, 2022.

[2] David L Applegate, Robert E Bixby, Vašek Chvátal, and William J Cook. The traveling salesman problem. In *The Traveling Salesman Problem*. Princeton university press, 2011.

[3] John Daniel Bagley. *The behavior of adaptive systems which employ genetic and correlation algorithms*. University of Michigan, 1967.

[4] James Edward Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and their applications*, volume 1. Hillsdale, New Jersey, 1985.

[5] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Comput. Surv.*, 35(3):268–308, sep 2003.

[6] Hans J Bremermann. *The evolution of intelligence: The nervous system as a model of its environment*. University of Washington, Department of Mathematics, 1958.

[7] Daniel Joseph Cavicchio. Adaptive search using simulated evolution. Technical report, University of Michigan., 1970.

[8] N Christofides. The bionomic algorithm. In *Proc of the associazione Italiana di ricerca operativa conference, Savona, Italy*, 1994.

[9] Alberto Colorni, Marco Dorigo, Vittorio Maniezzo, et al. An investigation of some properties of an" ant algorithm". In *Ppsn*, volume 92, 1992.

[10] Richard Dawkins and Nicola Davis. *The selfish gene*. Macat Library, 2017.

[11] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* University of Michigan, 1975.

[12] Mehrdad Dianati, Insop Song, and Mark Treiber. An introduction to genetic algorithms and evolution strategies. Technical report, Citeseer, 2002.

[13] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, volume 2, pages 1470–1477. IEEE, 1999.

[14] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[15] Stephanie Forrest. Scaling fitnesses in the genetic algorithm. *Documentation for Prisoners Dilemma and Norms Programs That Use the Genetic Algorithm. Unpublished manuscript*, 1985.

[16] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[17] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[18] Fred Glover. Tabu search and adaptive memory programming—advances, applications and challenges. *Interfaces in computer science and operations research*, pages 1–75, 1997.

[19] Fred Glover and Manuel Laguna. Tabu search. In *Handbook of combinatorial optimization*, pages 2093–2229. Springer, 1998.

[20] DE Goldberg. Genetic algorythms in search, optimization and machine learning.-addison-wesley pc. 1989.

[21] William Eugene Hart, Natalio Krasnogor, and James E Smith. Memetic evolutionary algorithms. In *Recent advances in memetic algorithms*, pages 3–27. Springer, 2005.

[22] John H Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM (JACM)*, 9(3):297–314, 1962.

[23] H Holland John. Adaptation in natural and artificial systems. *Ann Arbor: University of Michigan Press*, 1975.

[24] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[25] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[26] Aristide Mingozzi and Luca Accorsi. L'algoritmo bionomico per il traveling salesman problem. 2015.

[27] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[28] Heinz Muhlenbein. How genetic algorithms really work: I. mutation and hillclimbing. In *Proc. 2nd Int. Conf. on Parallel Problem Solving from Nature, 1992*. Elsevier, 1992.

[29] Michael G Norman, Pablo Moscato, et al. A competitive and cooperative approach to complex combinatorial search. In *Proceedings of the 20th Informatics and Operations Research Meeting*, pages 3–15. Citeseer, 1991.

[30] Antonin Ponsich, Catherine Azzaro-Pantel, Serge Domenech, and Luc Pibouleau. Constraint handling strategies in genetic algorithms application to optimal batch plant design. *Chemical Engineering and Processing: Process Intensification*, 47(3):420–434, March 2008.

[31] RS Rosenberg. Simulation of genetic populations with biochemical properties (ph. d. thesis). university of michigan. *Ann Arbor, Michigan*, 1967.

[32] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[33] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[34] Éric Taillard. Robust taboo search for the quadratic assignment problem. *Parallel computing*, 17(4-5):443–455, 1991.

[35] Paul Werbos. Beyond regression:" new tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

Ringraziamenti

É finalmente giunto al termine questo primo percorso fatto a Cesena. Percorso che mi ha formato, cresciuto, ma soprattutto stupito in positivo.
Ringrazio i miei genitori, che mi hanno permesso di studiare e di fare questa esperienza che mi ha sicuramente cambiato la vita.
Questo corso mi ha regalato tante soddisfazioni, ma anche alcuni momenti di demoralizzazione, che fortunatamente grazie ai miei compagni sono riuscito a superare.
Ringrazio soprattutto *All*, *Buiz* e *Kel* con cui ho passato i momenti migliori. Ringrazio anche tutte quelle persone con cui in questi anni ho scherzato, riso, studiato, mangiato e bevuto un caffè.
Non posso non ringraziare gli amici che da sempre mi accompagnano in serate stupende e spensierate. Una menzione particolare va a *Filo* e alla *Linda* che mi hanno accompagnato e supportato nei momenti più difficili.
Tra i docenti invece desidero ringraziare il prof. Vigo, nonché mio relatore, per l'opportunità che mi ha dato, e il prof. Ghini per la gentilezza, la bontà e l'empatia dimostrate agli studenti.
Infine un enorme ringraziamento a Luca che è sempre stato disponibile durante il percorso di questo elaborato, anche se non era tenuto a farlo. Oltre a questo lo ringrazio anche per tutte le cose che ho imparato e che mi ha fatto scoprire.