UNIVERSITÀ DI SALERNO

UNIVERSITÀ DI SALERNO

DIPARTIMENTO DI INFORMATICA

WoodLot

Progetto di Fondamenti di Intelligenza Artificiale

Autore:
Alessia TURE

January 14, 2023

Contents

List of Figures

List of Tables

Symbol	Definition	Unit
V	Velocity	[m/s]
ρ	Density	[kg/m^3]
Re	Reynolds Number	
u	velocity	[m/s]
L^*	Characteristic Length	[m]
μ	Dynamic Viscosity	[Pa/s]
Pr	Prandtl Number	
Cp	Specific Heat Capacity	[J/kg K]
k	Thermal Conductivity	[W/m k]
Nu	Nusselt number	
h	Heat transfer coefficient	[W/m k]
μ_o	Dynamic viscosity of bulk fluid	[Pa/s]
μ_b	Dynamic viscosity at the wall	[Pa/s]
L	Length of heating section	[m]
E_c	Energy of the cold stream	[j]
E_h	Energy of the hot stream	[j]
η	Energy efficiency	[%]
U_o	Overall heat transfer coefficient	[W/m k]
h_i	Internal heat transfer coefficient	[W/m k]
h_o	External heat transfer coefficient	[W/m k]
k_w	External heat transfer coefficient	[W/m k]
t_c	Temperature of the cold	[℃]
t_h	Temperature of the hot	[℃]
D	Diameter	[m]
A	Area	[m^2]
Q	Heat transferred	[j]
F	Friction factor	[-]
g	Gravity	[m s$^-$2]

Table 1: Nomenclature Table

Chapter 1

Introduzione

1.1 Obiettivi

Lo scopo principale di questo progetto è stato quello di creare un' agente artificiale a supporto dell'applicativo WoodLot.

WoodLot è un e-commerce che permette l'acquisto simbolico di un albero. Gli alberi acquistati dagli utenti saranno piantati da contadini. Un albero dopo essere stato comprato, entra nello stato n̈on assegnato ̈. I contadini, che si possono registrare autonomamente alla piattaforma, possono provenire da tutti i paesi del mondo. Il responsabile degli ordini sarà responsabile di assegnare gli alberi ai contadini. Lo scopo dell' agente artificiale sarà quello di semplificare il lavoro del responsabile degli ordini, fornendo il miglior assegnamento di contadini agli alberi. Il responsabile degli ordini sarà di libero di seguire il consiglio dell'agente oppure no, soprattutto in situazioni limite

L'assegnazione degli alberi ai contadini deve rispettare alcuni vincoli:

- contadino che si trova nel luogo adatto alla crescita dell'albero
- contadino che ha in cura meno alberi di tutti
- tenere conto delle penalità dei contadini

Un contadino comunica di aver piantato un albero, caricando una foto. Il responsabile degli ordini si occupa di validare la foto. Un contadino riceve una penalità quando, dopo aver ricevuto l'assegnazione di un albero non comunica la piantumazione o dopo aver caricato una foto non valida.

1.2 Specifica PEAS

- **Performance:** la misura di performance dell'agente è la sua capacità di produrre un assegnamento di alberi ai contadini che rispetti i vincoli dell'assegnamento.
- **Environment:** Descrizione degli elementi che formano l'ambiente (descritto sotto).
- **Actuators:** Gli attuatori disponibili dell'agente per intraprendere le azioni. In questo caso gli attuatori saranno la lista degli alberi non ancora assegnati ad un contadino e la lista dei contadini.
- **Sensors:** I sensori dell'agente consistono nel bottone dell'assegnamento presente nella pagina del responsabile ordini.

1.2.1 Caratteristiche dell'ambiente

L'ambiente in cui opera l'agente è lo spazio degli utenti del sito (responsabile ordini e contadini) unito a quello degli alberi da piantare e le loro caratteristiche. L'ambiente è:

- **Sequenziale**, in quanto le azioni passate degli utenti influenzano le decisioni future dell'agente.
- **Completamente osservabile**, in quanto si ha accesso a tutte le informazioni relative agli utenti ed ai prodotti in ogni momento
- **Stocastico**, in quanto lo stato dell'agente cambia indipendentemente dalle azioni dell'agente

- **Dinamico** in quanto nel corso dell'elaborazione un altro responsabile degli ordini potrebbe effettuare un assegnazione cambiando l'insieme degli alberi da piantare
- **Discreto o continuo**
- **A singolo agente**, in quanto l'unico agente che opera in questo campo è quello in oggetto.

1.2.2 Analisi del problema

Il problema può essere formalizzato descrivendolo:

- **Stato iniziale:**
- **Descrizione delle possibili azioni:**
- **Modello di transizione:**
- **Test obiettivo:**
- **Costo del cammino:**

Per la realizzazione del programma agente è stato deciso di affrontare questo problema implementando non un'unica soluzione, ma diverse soluzioni utilizzando gli algoritmi di ricerca studiati durante il corso di "Fondamenti di Intelligenza Artificiale A.A 2022/23", in modo da valutarne i diversi punti di forza e debolezza.

Chapter 2

Algoritmi di ricerca

2.1 Algortimo di ricerca best-fit greedy

Un algoritmo avido è un algoritmo che segue la strategia di fare la scelta localmente ottimale ad ogni passo con la speranza di trovare un ottimale globale. Ecco un esempio di algoritmo avido per il problema dell'assegnazione del lavoro:

1. Inizializza l'assegnazione dei lavoratori ai lavori da svuotare.
2. Mentre ci sono ancora lavoratori non assegnati:
   (a) Per ogni lavoratore non assegnato, trovare il lavoro che riduce al minimo il costo dell'assegnazione del lavoratore a quel lavoro.
   (b) Assegnare il lavoratore al lavoro selezionato.
3. Restituire l'assegnazione dei lavoratori al lavoro.

Questo algoritmo funziona selezionando ripetutamente il lavoro che minimizza il costo di assegnazione per ogni lavoratore non assegnato. Si ferma quando tutti i lavoratori sono stati assegnati a un lavoro.

Anche se l'algoritmo greedy potrebbe essere in grado di trovare una buona soluzione al problema di assegnazione del lavoro, non è garantito che trovi la soluzione ottimale. Ciò è dovuto al fatto che considera solo la scelta ottimale locale in ogni passo e non tiene conto dell'ottimizzazione globale

I best-fit greedy algorithm è un approccio che può essere utilizzato per risolvere il problema del job assignment con i vincoli che hai descritto. L'idea alla base di questo algoritmo è di assegnare il lavoro al lavoratore che è in grado di completarlo nel minor tempo possibile, tenendo conto anche delle penalità che potrebbero essere applicate se il lavoro non viene completato in tempo.

Il processo può essere suddiviso in questi passaggi:

Ordinare i lavoratori in base al tempo stimato per completare il lavoro. Assegnare il lavoro al lavoratore che può completarlo nel minor tempo possibile, tenendo conto delle penalità che potrebbero essere applicate se il lavoro non viene completato in tempo. Ripetere il processo fino a quando tutti i lavori sono stati assegnati. Questo algoritmo cerca di massimizzare il numero di lavoratori utilizzati assegnando il lavoro al lavoratore che può completarlo nel minor tempo possibile, il che significa che sarà in grado di accettare nuovi lavori in futuro. Tieni presente che questo algoritmo potrebbe non essere la soluzione ottimale in tutte le situazioni e potrebbe essere necessario modificarlo in base alle esigenze specifiche del problema.

Questo codice utilizza una classe 'Job' per rappresentare un lavoro con la sua durata e una lista di lavoratori idonei per svolgerlo, e una classe 'Worker' per rappresentare un lavoratore con il suo ID, il numero di giorni di penalità rimanenti e la durata stimata per completare il lavoro.

Il metodo 'assignJobs' prende in input la lista dei lavori e la lista dei lavoratori, e assegna ciascun lavoro al lavoratore idoneo che può completarlo nel minor tempo possibile. Prima di assegnare il lavoro, il metodo ordina la lista dei lavoratori in base alla durata stimata per completare il lavoro, quindi per ogni lavoro cerca il lavoratore idoneo che ha la durata stimata minore e che non ha giorni di penalità. Se viene trovato un lavoratore idoneo, il lavoro viene assegnato a questo lavoratore e il numero di giorni di penalità viene impostato sulla durata del lavoro. Altrimenti, viene stampato un messaggio che indica che non ci sono lavoratori idonei disponibili per il lavoro.

Il costruttore viene chiamato quando viene creato un nuovo oggetto di tipo Worker e serve a inizializzare i campi di istanza id, penaltyDays e estimatedDuration con i valori passati come argomenti. Ad esempio, nel codice che ti ho fornito precedentemente, il costruttore viene chiamato come segue per creare un nuovo oggetto di tipo Worker

2.2 Algoritmo BFS

è possibile risolvere il problema di assegnazione del lavoro utilizzando un algoritmo di ricerca non informata. Gli algoritmi di ricerca non informata (noti anche come algoritmi di ricerca in ampiezza o di larghezza) esplorano l'intero spazio delle soluzioni senza utilizzare alcuna informazione specifica del problema.

Un esempio di algoritmo di ricerca non informata che potrebbe essere utilizzato per risolvere il problema di assegnazione del lavoro è l'algoritmo di ricerca in ampiezza (BFS, dall'inglese Breadth-First Search).

L'algoritmo di ricerca in ampiezza funziona esplorando tutte le soluzioni a distanza k dalla soluzione iniziale prima di passare alle soluzioni a distanza k + 1. La distanza di una soluzione dalla soluzione iniziale è il numero di passi di modifica (ad esempio, assegnamenti di lavoro) che sono necessari per arrivarci a partire dalla soluzione iniziale.

Per risolvere il problema di assegnazione del lavoro utilizzando l'algoritmo di ricerca in ampiezza, dovremmo rappresentare le soluzioni come nodi in un grafo e utilizzare l'algoritmo di ricerca in ampiezza per esplorare il grafo alla ricerca della soluzione ottimale.

Tieni presente che gli algoritmi di ricerca non informata, come l'algoritmo di ricerca in ampiezza, possono essere molto lenti per risolvere problemi di grandi dimensioni

L'algoritmo BFS (Breadth-First Search) è un algoritmo di ricerca che esplora un grafo o un albero in modo da visitare tutti i nodi ad una certa distanza prima di passare a quelli ad una distanza maggiore. In altre parole, l'algoritmo BFS esplora tutti i vicini di un nodo prima di esplorare i suoi figli.

L'algoritmo BFS è utilizzato comunemente per trovare la soluzione ottimale a problemi di ricerca in cui la soluzione ottimale è quella più breve (ad esempio, il percorso minimo in un grafo). Tieni presente che l'algoritmo BFS può essere molto inefficiente in termini di tempo e spazio se il grafo o l'albero da esplorare è molto grande o ha una profondità molto elevata.

Inoltre, l'algoritmo BFS richiede che tutti i nodi del grafo o dell'albero siano memorizzati in memoria, il che può essere un problema se il numero di nodi è molto elevato. Per questo motivo, può essere preferibile utilizzare l'algoritmo DFS (Depth-First Search) in questi casi, che esplora solo un ramo alla volta e non richiede di mantenere in memoria tutti i nodi del grafo o dell'albero.

L'algoritmo di assegnamento dei lavori che ho proposto utilizzando l'algoritmo BFS (Breadth-First Search) cerca di trovare una soluzione ottimale assegnando i lavori ai lavoratori in modo che il numero di lavoratori utilizzati sia il massimo possibile e che i lavori siano completati nel minor tempo possibile.

L'algoritmo utilizza una coda per mantenere traccia dei lavoratori da esaminare e una mappa per evitare di esaminare gli stati già visitati. Ad ogni passo, il lavoratore in cima alla coda viene estratto e si cerca il prossimo lavoro da assegnare a questo lavoratore. Se il lavoratore è idoneo per il lavoro, il lavoro viene assegnato e un nuovo stato per il lavoratore viene aggiunto alla coda con i giorni di penalità aggiornati. Se il lavoratore non è idoneo per il lavoro, viene semplicemente saltato e si passa al prossimo lavoratore nella coda.

L'algoritmo BFS è spesso utilizzato per trovare la soluzione ottimale a problemi di ricerca in cui la soluzione ottimale è quella più breve (ad esempio, il percorso minimo in un grafo). Tieni presente che l'algoritmo BFS può essere molto inefficiente in termini di tempo e spazio se il grafo o l'albero da esplorare è molto grande

L'algoritmo di assegnamento dei lavori che ho proposto utilizzando l'algoritmo BFS (Breadth-First Search) cerca di trovare una soluzione ottimale assegnando i lavori ai lavoratori in modo che il numero di lavoratori utilizzati sia il massimo possibile e che i lavori siano completati nel minor tempo possibile.

L'algoritmo utilizza una coda per mantenere traccia dei lavoratori da esaminare e una mappa per evitare di esaminare gli stati già visitati. Ad ogni passo, il lavoratore in cima alla coda viene estratto e si cerca il prossimo lavoro da assegnare a questo lavoratore. Se il lavoratore è idoneo per il lavoro, il lavoro viene assegnato e un nuovo stato per il lavoratore viene aggiunto alla coda con i giorni di penalità aggiornati. Se il lavoratore non è idoneo per il lavoro, viene semplicemente saltato e si passa al prossimo lavoratore nella coda.

L'algoritmo BFS è spesso utilizzato per trovare la soluzione ottimale a problemi di ricerca in cui la soluzione ottimale è quella più breve (ad esempio, il percorso minimo in un grafo). Tieni presente che l'algoritmo BFS può essere molto inefficiente in termini di tempo e spazio se il grafo o l'albero da esplorare è molto grande aggiunto alla coda con i giorni di penalità aggiornati. In questo modo, l'algoritmo tiene traccia di quanti lavori sono stati completati dal lavoratore e di quanti giorni di penalità dovrà affrontare se il lavoro non viene completato in tempo.

Se il lavoratore non è idoneo per il lavoro o non ci sono altri lavori da assegnare, il lavoratore viene rimosso dalla coda e si passa al prossimo lavoratore. L'algoritmo continua finché non viene trovata una soluzione valida o finché non vengono esaminati tutti i lavoratori e tutti i lavori.

Se l'algoritmo trova una soluzione valida, restituisce una mappa che assegna a ogni lavoro il lavoratore che lo eseguirà. Se non viene trovata alcuna soluzione valida, il metodo restituisce null.

Spero che questo ti possa essere d'aiuto. Se hai altre domande o hai bisogno di ulteriore aiuto, non esitare a chiedere.

L'algoritmo di assegnamento dei lavori che ho proposto utilizzando l'algoritmo BFS (Breadth-First Search) cerca di trovare una soluzione ottimale assegnando i lavori ai lavoratori in modo che il numero di lavoratori

utilizzati sia il massimo possibile e che i lavori siano completati nel minor tempo possibile.

L'algoritmo utilizza una coda per mantenere traccia dei lavoratori da esaminare e una mappa per evitare di esaminare gli stati già visitati. Ad ogni passo, il lavoratore in cima alla coda viene estratto e si cerca il prossimo lavoro da assegnare a questo lavoratore. Se il lavoratore è idoneo per il lavoro, il lavoro viene assegnato e un nuovo stato per il lavoratore viene aggiunto alla coda con i giorni di penalità aggiornati. Se il lavoratore non è idoneo per il lavoro, viene semplicemente saltato e si passa al prossimo lavoratore nella coda.

L'algoritmo BFS è spesso utilizzato per trovare la soluzione ottimale a problemi di ricerca in cui la soluzione ottimale è quella più breve (ad esempio, il percorso minimo in un grafo). Tieni presente che l'algoritmo BFS può essere molto inefficiente in termini di tempo e spazio se il grafo o l'albero da esplorare è molto grande aggiunto alla coda con i giorni di penalità aggiornati. In questo modo, l'algoritmo tiene traccia di quanti lavori sono stati completati dal lavoratore e di quanti giorni di penalità dovrà affrontare se il lavoro non viene completato in tempo.

Se il lavoratore non è idoneo per il lavoro o non ci sono altri lavori da assegnare, il lavoratore viene rimosso dalla coda e si passa al prossimo lavoratore. L'algoritmo continua finché non viene trovata una soluzione valida o finché non vengono esaminati tutti i lavoratori e tutti i lavori.

Se l'algoritmo trova una soluzione valida, restituisce una mappa che assegna a ogni lavoro il lavoratore che lo eseguirà. Se non viene trovata alcuna soluzione valida, il metodo restituisce null.

2.3 Algoritmo A*

è anche possibile risolvere il problema di assegnazione del lavoro utilizzando un algoritmo di ricerca informata (noto anche come algoritmo di ricerca in profondità o di profondità). Gli algoritmi di ricerca informata utilizzano informazioni specifiche del problema per guidare la ricerca verso le soluzioni più promettenti.

Un esempio di algoritmo di ricerca informata che potrebbe essere utilizzato per risolvere il problema di assegnazione del lavoro è l'algoritmo A* (pronunciato "A-star"). L'algoritmo A* è un algoritmo di ricerca in profondità che utilizza una funzione di valutazione (nota anche come funzione euristica) per stimare il costo di arrivare alla soluzione ottimale a partire da una soluzione data. La funzione di valutazione tiene conto sia del costo attuale del percorso sia del costo stimato per arrivare alla soluzione ottimale a partire da quel punto. L'algoritmo A* esplora le soluzioni in ordine decrescente di valutazione, il che significa che esplora prima le soluzioni che sembrano più promettenti.

Per risolvere il problema di assegnazione del lavoro utilizzando l'algoritmo A*, dovremmo rappresentare le soluzioni come nodi in un grafo e utilizzare l'algoritmo A* per esplorare il grafo alla ricerca della soluzione ottimale. La funzione di valutazione dovrebbe essere scelta in modo tale da stimare il costo di assegnare ogni lavoratore a un lavoro in modo da arrivare alla soluzione ottimale.

Gli algoritmi di ricerca informata, come l'algoritmo A*, sono generalmente più efficaci dei metodi di ricerca non informata per risolvere problemi di grandi dimensioni, poiché utilizzano informazioni specifiche del problema per orientare la ricerca verso le soluzioni più promettenti. Tuttavia, possono essere più complessi da implementare e richiedono una buona scelta della funzione di valutazione

L'algoritmo A* è un algoritmo di ricerca che utilizza una funzione di valutazione per valutare l'efficacia di ogni nodo nell'albero di ricerca e seleziona il nodo con il valore di valutazione più basso per essere esplorato per primo. La funzione di valutazione utilizzata dall'algoritmo A* viene spesso chiamata "fattore di valutazione" ed è una combinazione della distanza euristica del nodo dall'obiettivo (nota anche come "costo stimato") e della distanza del nodo dalla radice dell'albero di ricerca (nota anche come "costo effettivo").

L'algoritmo A* è spesso utilizzato per trovare la soluzione ottimale a problemi di ricerca in cui la soluzione ottimale è quella più breve (ad esempio, il percorso minimo in un grafo). Tieni presente che l'algoritmo A* può essere molto efficiente in termini di tempo e spazio se la funzione di valutazione scelta è accurata e conduce a una buona stima del costo effettivo dei nodi.

2.4 Simulated Annealing

è anche possibile risolvere il problema di assegnazione del lavoro utilizzando un algoritmo di ricerca locale. Gli algoritmi di ricerca locale partono da una soluzione iniziale e cercano di migliorarla iterativamente utilizzando operazioni di modifica locali (ad esempio, assegnare un lavoratore a un lavoro diverso).

Un esempio di algoritmo di ricerca locale che potrebbe essere utilizzato per risolvere il problema di assegnazione del lavoro è l'algoritmo di ricerca simulata (Simulated Annealing, SA). L'algoritmo SA utilizza un processo di raffreddamento simulato per evitare di rimanere bloccato in una soluzione ottima locale. A temperature più alte, SA accetta soluzioni peggiori con maggiore probabilità, il che permette di esplorare l'intero spazio delle soluzioni e di evitare di rimanere bloccato in una soluzione ottima locale. A temperature più basse,

SA diventa meno probabile che accetti soluzioni peggiori, il che significa che si concentra su soluzioni sempre migliori.

Per risolvere il problema di assegnazione del lavoro utilizzando l'algoritmo SA, dovremmo inizializzare una soluzione iniziale casuale e quindi applicare iterativamente operazioni di modifica locali (ad esempio, assegnare un lavoratore a un lavoro diverso) per cercare di migliorare la soluzione corrente. L'algoritmo SA dovrebbe utilizzare un processo di raffreddamento simulato per decidere se accettare o meno soluzioni peggiori.

Gli algoritmi di ricerca locale, come l'algoritmo SA, sono generalmente meno efficienti dei metodi di ricerca informata per risolvere problemi di grandi dimensioni, poiché si concentrano solo su soluzioni vicine alla soluzione corrente. Tuttavia, possono essere facili da implementare e possono fornire buoni risultati per molti problemi di ottimizzazione.

È importante notare che gli algoritmi di ricerca locale possono essere bloccati in soluzioni ottime locali e non possono garantire di trovare la soluzione ottimale globale. Pertanto, potrebbe essere necessario eseguire più volte l'algoritmo di ricerca locale con diverse soluzioni iniziali o utilizzare una combinazione di algoritmi di ricerca per ottenere risultati migliori.

L'algoritmo di Simulated Annealing (SA) è un algoritmo di ottimizzazione globale basato sulla ricerca di una soluzione ottimale attraverso l'esplorazione di uno spazio di soluzioni. Si può utilizzare per risolvere il problema dell'assegnamento dei lavori cercando di trovare una soluzione che massimizzi il numero di lavoratori utilizzati e rispetti i vincoli del problema (ad esempio, i lavoratori devono avere le competenze necessarie per completare il lavoro e non possono avere troppi giorni di penalità).

Il metodo generateRandomSolution genera una soluzione casuale assegnando a ogni lavoro un lavoratore casuale idoneo. Il metodo generateRandomNeighbor genera una soluzione adiacente alla soluzione corrente cambiando l'assegnamento di un solo lavoro. Il metodo calculateCost calcola il costo di una soluzione, ovvero il numero di lavoratori utilizzati e il numero di giorni di penalità. Il metodo calculateCost prende in input la soluzione, la lista dei lavori e la lista dei lavoratori. Inizializza il costo a zero e crea una mappa per memorizzare il numero di giorni di penalità di ogni lavoratore. Per ogni lavoro nella soluzione, aggiunge il lavoratore alla mappa e aumenta il suo numero di giorni di penalità di conseguenza. Infine, aggiunge il numero totale di giorni di penalità alla costo e restituisce il risultato.

Il metodo generateRandomNeighbor crea una copia della soluzione corrente e seleziona a caso un lavoro da cambiare. Utilizza il metodo findQualifiedWorker per trovare

2.5 Algoritmi genetici

Nell'esempio di algoritmo genetico che ti ho fornito, viene utilizzata una metaheuristica di tipo "algoritmo genetico" per risolvere il problema di assegnamento dei lavori. L'algoritmo funziona creando una "popolazione" di soluzioni candidate, facendo "evolvere" queste soluzioni attraverso un processo di selezione, crossover e mutazione, e restituendo la soluzione di migliore qualità trovata alla fine del processo.

Formulato il problema, era chiara la necessità di un algoritmo di ottimizzazione. Infine l'opzione più promettente è sembrata essere quella di utilizzare un algoritmo genetico in quanto le sue caratteristiche sono simili a quelle del problema.

2.5.1 Rappresentazione degli individui

Un individuo è rappresentato da una matrice n per m dove n rappresenta il numero di contadini presenti nel database e m è il numero di alberi da allocare. Questo implica che la popolazione sarà formata da un insieme di matrici di dimensione n per m. Questa rappresentazione degli individui però spreca spazio di memoria, perchè un albero verrà assegnato ad un unico contadino, quindi per ogni colonna avremo un unico valore. Quindi, si è scelta una rappresentazione più compatta degli individui. Un individuo è rappresentato da un array di dimensione n, dove n è il numero di alberi da assegnare. Ogni cella dell'array conterà un ID che identifica il contadino a cui viene assegnato l'albero. In questo caso la popolazione sarà formata da array di dimensione n.

La popolazione verrà inizializzata casualmente.

2.5.2 Soluzione ottima

Durante la progettazione dell'algoritmo genetico si è cercato un modo per stabilire la bontà di una configurazione di parametri. La soluzione ottima è composta da un individuo che massimizzi il numero di contadini.

2.5.3 Dati e variabili

2.5.4 Definizione dei vincoli

1. Un albero può essere assegnato solo ai contadini che si trovano nel luogo adatto alla pinatumazione del albero.
2. Un contadino che non ha piantato un albero che ha ricevuto in precedenza non deve ricevere nuovi alberi (deve essere messo in fondo alla lista ??)
3. f(x) = massimizzare il numero di alberi assegnati.

2.5.5 Funzioni obiettivo

L'obiettivo è ottimizzare l'assegnazione degli alberi rispettando i requisiti. Pertanto dobbiamo innanzitutto definire i requisiti del problema.

1. f(x) = massimizzare il numero di contadini utilizzati. (questo perchè lo scopo del progetto è auitare quanti più contadini possibili in modo da auitare quante più persone possibili)
2. (non so se va bene) f(x) =

Quindi l'algoritmo dovrebbe massimizzare il numero di contadini usati tenendo assicurando che tutti gli alberi siano effettivamente allocati

2.5.6 Operatori genetici

Selezione

L'algoritmo di selezione scelto per gli individui è: *Truncation* dove vengono scelti i primi M individui con valore di fit minore. Modificare la selezione dei genitori: puoi modificare la funzione di selezione dei genitori in modo che selezioni solo i cromosomi che soddisfano i vincoli. Ad esempio, potresti utilizzare la selezione per ruoli o ruoli di ruolo per garantire che solo i cromosomi che assegnano i lavoratori ai lavori appropriati vengano selezionati per il crossover.

Elitismo

La selezione in base alla funzione di fit in maniera non decrescente racchiude implicitamente l'elitismo, solo i migliori (un individuo x è migliore di un individuo y se: fit(x) ¡ fit(y))

Crossover

Per semplicità i primi tentativi sono stati effettuati con un crossover di tipo *Single Point*. Modificare il crossover: puoi modificare il crossover in modo che rispetti i vincoli. Ad esempio, potresti utilizzare un crossover di tipo "uniforme" in cui ciascun gene del figlio viene scelto in modo casuale dai genitori, in modo da evitare di trasmettere genes che violano i vincoli.

Mutazione

La mutazione avviene tramite *Random Resetting*, Modificare la mutazione: puoi modificare la mutazione in modo che rispetti i vincoli. Ad esempio, potresti utilizzare una mutazione di tipo "swap" che scambia due lavoratori tra loro, in modo da garantire che i lavoratori siano sempre assegnati a lavori appropriati.

2.5.7 Stopping condition

La "stopping condition" (condizione di arresto) è la condizione che determina quando fermare l'esecuzione dell'algoritmo genetico. In genere, questa condizione viene definita in modo da fermare l'algoritmo quando si raggiunge un risultato soddisfacente o quando si supera un certo numero di iterazioni.

Nell'algoritmo genetico fornito, la stopping condition è rappresentata dal numero massimo di generazioni (maxGenerations). L'algoritmo viene eseguito per un massimo di maxGenerations iterazioni, quindi viene interrotto e viene restituito il risultato corrente.

2.5.8 Metaeuristica

2.5.9 Stopping Condition

Dato il numero molto limitato di entry nel dataset (al più circa un migliaio di ordini), l'algoritmo arriva molto velocemente a convergenza.

2.5.10 Algoritmo

1. Costruire una popolazione iniziale contente tutti gli individui contenuti ne DB

2. Inizializzare la popololazione

3. Selezionare un sottoinsieme di individui della popololazione iniziale per ammertterli nel Mating Pool tramite Truncation

4. Gli individui selezionati vengono fatti accoppiare tramire Single Po

2.5.11 Preferenced Sorting

Il "preference sorting" potrebbe essere una buona idea da considerare per un algoritmo genetico che risolve il problema di assegnazione del lavoro. In generale, il preference sorting consiste nel classificare gli elementi di un insieme in base a preferenze o priorità.

In questo specifico contesto, potremmo utilizzare il preference sorting per classificare i lavoratori in base alle loro preferenze per i diversi lavori. Ad esempio, potremmo assegnare un valore di preferenza più alto ai lavoratori che hanno maggiori competenze o esperienze per un determinato lavoro.

Il preference sorting potrebbe essere utilizzato come parte di un algoritmo genetico per il problema di assegnazione del lavoro in diversi modi. Ad esempio, potremmo utilizzare il preference sorting per selezionare i genitori durante il crossover, in modo che i lavoratori con preferenze più alte per determinati lavori abbiano maggiori probabilità di trasmettere le loro caratteristiche ai loro figli. Potremmo anche utilizzare il preference sorting per valutare la fitness dei cromosomi, assegnando una maggiore fitness ai cromosomi che assegnano i lavoratori ai lavori in base alle loro preferenze.

2.6 Appendix

2.6.1 Appendix A

Item List:

- Item 1
- Item 2
- Item 3
- Item 4
- Item 5

2.6.2 Appendix C

```
import unittest


class TestSum(unittest.TestCase):

   def test_sum(self):
      self.assertEqual(sum([1, 2, 3]), 6, "Should be 6")

   def test_sum_tuple(self):
      self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")

if __name__ == '__main__':
   unittest.main()

$ python test_sum_unittest.py
.F
======================================================================
FAIL: test_sum_tuple (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)

$ pip install nose2
$ python -m nose2
.F
======================================================================
FAIL: test_sum_tuple (__main__.TestSum)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_sum_unittest.py", line 9, in test_sum_tuple
    self.assertEqual(sum((1, 2, 2)), 6, "Should be 6")
AssertionError: Should be 6

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```