UNIVERSITÀ DI SALERNO

DIPARTIMENTO DI INFORMATICA

WoodLot

Progetto di Fondamenti di Intelligenza Artificiale

Autore:
Alessia TURE

1 luglio 2023

FONDAMENTI DI INTELLIGENZA ARTIFICIALE, UNIVERSITÀ DEGLI STUDI DI SALERNO

HTTPS://GITHUB.COM/A-TURE/WOODLOT-FIA

Progetto realizzato per il corso di fondamenti di intelligenza artificiale.

Luglio 2023

Indice

Capitolo 1

Introduzione

1.1 Obiettivo

Lo scopo di questo progetto è 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 "non assegnato". I contadini, che si possono registrare autonomamente alla piattaforma, possono provenire da tutti i paesi del mondo ed hanno il compito di piantare gli alberi. Il responsabile degli ordini si occuperà di assegnare gli alberi ai contadini. Quindi, lo scopo dell' agente artificiale sarà quello di semplificare il lavoro del responsabile degli ordini, fornendo la migliore attribuzione di contadini agli alberi. Il responsabile degli ordini sarà libero di seguire il consiglio dell'agente o meno. Un contadino comunica di aver piantato un albero, caricando una foto. Il responsabile degli ordini si occupa di validare la foto. Ogni contadino ha un punteggio "penalità" che rappresenta il grado di affidabilità di un contadino. Ogni volta che un contadino non comunica la piantumazione di un albero o non carica una foto valida, il contadino riceve una penalità da parte del responsabile degli ordini.

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

- i contadini si devono trovare nel luogo adatto alla crescita dell'albero
- tutti gli alberi devono essere assegnati

Inoltre, l'assegnazione dovrebbe premiare i contadini che hanno il minimo punteggio di penalità, assegnando loro al più 4 alberi e promuovendo una distribuzione equa degli alberi ai contadini che si trovano lo stesso valore di penalità e nello stesso paese, una volta assegnati 4 alberi ai contadini con penalità minima consideriamo i nuovi contadini a penalità minima.

1.2 Specifica PEAS

Di seguito è riportata la descrizione PEAS dell'ambiente operativo.

- **Performance:** la misura di performance dell'agente è la sua capacità di produrre un'assegnazione che rispetti i vincoli e che assegni al più 4 alberi ai contadini ordinati in base alla penalità crescente.
- **Environment:** l'ambiente in cui opera l'agente è costituito dall' insieme di tutte le possibili assegnazioni.
- **Actuators:** gli attuatori disponibili dell' agente per intraprendere le azioni. In questo caso, gli attuatori saranno la lista degli alberi che si trovano nello stato non assegnato e la lista dei contadini.
- **Sensors:** il sensore dell'agente consiste nel bottone dell'assegnamento presente nella pagina del responsabile ordini.

1.2.1 Proprietà dell'ambiente

L'ambiente è:

- **Sequenziale**, i sensori dell'agente gli danno accesso allo stato completo dell'ambiente in ogni momento.
- **Completamente osservabile**, in quanto si ha accesso a tutte le informazioni relative agli utenti ed ai prodotti in ogni momento.
- **Deterministico**, in quanto lo stato dell'ambiente cambia indipendentemente dalle azioni dell'agente.
- **Dinamico** in quanto nel corso dell'elaborazione un responsabile degli ordini potrebbe effettuare un'assegnazione cambiando l'insieme degli alberi da piantare.
- **Discreto**, in quanto sono presenti un numero limitato di percezioni.
- **A singolo agente**, in quanto l'unico agente che opera in questo campo è quello in oggetto.

1.3 Analisi del problema

Il problema può essere formalizzato descrivendolo:

- **Stato iniziale:** un'assegnazione iniziale degli alberi ai contadini.
- **Descrizione delle possibili azioni:** le possibili azioni sono lo scambio di alberi tra contadini, la rimozione di alberi da un contadino e l'assegnazione di alberi a un contadino.
- **Modello di transizione:** il modello di transizione descrive come le azioni modificano lo stato corrente. Restituisce una nuova assegnazione di alberi ai contadini.
- **Test obiettivo:** il test obiettivo è verificare se ogni contadino con penalità minima abbia un numero uguale di alberi assegnati rispetto a tutti gli altri contadini a pari penalità (al più in numero di 4) e tutti gli alberi siano assegnati.
- **Costo del cammino:** il costo del cammino può essere definito come il numero di azioni necessarie per raggiungere lo stato finale desiderato.

Questo problema è stato affrontato guardandolo da diverse prospettive, in particolare sono stati utilizzati algoritmi di ricerca informata e algoritmi di ricerca locale.

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.

Prima di poter realizzare un agente intelligente capace di produrre un assegnamento ottimo è necessario implementare le classi che modellano le informazioni degli alberi e dei contadini.

Di seguito riportiamo il codice della classe Tree, che modella le informazioni di un albero.

```
public class Tree implements Cloneable {
    private int id;
    private String country;

    public Tree(int id, String country) {
        this.id = id;
        this.country = country;
    }

    public int getId() {
        return id;
    }

    public String getCountry() {
        return country;
    }

    @Override
    public String toString() {
        return "Tree [id=" + id + ", country=" + country + "]";
    }

    @Override
    public Tree clone() {
        try {
            return (Tree) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Errore durante la clonazione dell'
    oggetto Tree", e);
        }
    }
}
```

La classe ha due variabili di istanza:

1. **ID** che permette di identificare univocamente gli alberi;
2. **country** che permette di identificare il paese di piantumazione dell'albero

Per semplicità, abbiamo considerato che ciascun albero sia piantabile in un unico paese.

La classe Farmer modella le informazioni associate ad un contadino, ha due variabili di istanza:

1. **ID** che permette di identificare univocamente i contadini;
2. **country** che permette di identificare il paese in cui il contadino opera;
3. **penalties** che identifica la penalità associata ad un contadino.

Per semplicità, abbiamo considerato che la penalità sia un intero generato casualmente all'atto della costruzione di un contadino.

```
public class Farmer implements Cloneable {
    private int id;
    private String country;
    private int penalties;
    private Random random = new Random();

    public Farmer(int id, String country, int treesPlanted) {
        this.id = id;
        this.country = country;
        this.penalties = random.nextInt(10);
    }

    public int getId() {
        return id;
    }

    public String getCountry() {
        return country;
    }


    public int getPenalties() {
        return penalties;
    }

    @Override
    public String toString() {
        return "Farmer{" +
                "id=" + id +
                ", country='" + country + '\'' +
                ", penalties=" + penalties +
                '}';
    }

    @Override
    public Farmer clone() {
        try {
            Farmer cloned = (Farmer) super.clone();
            cloned.random = new Random();
            return cloned;
        } catch (CloneNotSupportedException e) {
            throw new RuntimeException("Errore durante la clonazione dell'
    oggetto Farmer", e);
        }
    }
}
```

Capitolo 2

Algoritmi di ricerca

2.1 Algoritmi di ricerca informata

Gli algoritmi di ricerca informata utilizzano informazioni specifiche del problema per guidare la ricerca verso le soluzioni più promettenti.

2.1.1 Algortimo di ricerca best-fit greedy

L'algoritmo Best Fit Greedy cercherà di trovare la combinazione migliore tra gli alberi e i contadini.

Best Fit Greedy è stato scelto come primo algoritmo perché è facile da comprendere e da implementare, non richiede strutture dati avanzate, il che semplifica la sua implementazione e utilizzo; spesso produce risultati ragionevoli in tempi relativamente brevi; nonostante non garantisca la soluzione ottimale, può fornire soluzioni accettabili o addirittura vicine all'ottimo in molte situazioni. L'algoritmo è efficiente in termini di tempo di esecuzione poiché si concentra su scelte locali ottimali in ogni passo, può evitare l'esplorazione di tutto lo spazio delle soluzioni, riducendo così la complessità computazionale.

Il codice fornito di seguito implementa l'algoritmo Best Fit Greedy per l'assegnazione degli alberi ai contadini.

```
public class TreeAssignment {
    public static void assignTrees(List<Farmer> farmers, List<Tree> trees) {
        Map<Farmer, Integer> assignedTrees = new HashMap<>(); // Mappa per
    memorizzare il numero di alberi assegnati a ciascun contadino

        for (Tree tree : trees) {
            // Creiamo una lista di contadini compatibili per ogni albero
            List<Farmer> compatibleFarmers = new ArrayList<>();
            for (Farmer farmer : farmers) {
                // Verifichiamo se il contadino è nel paese giusto e non ha
    già 4 alberi assegnati
                if (farmer.getCountry().equals(tree.getCountry()) &&
    getAssignedTreeCount(assignedTrees, farmer) < 4) {
                    compatibleFarmers.add(farmer);
                }
            }
```

```
            // Ordiniamo i contadini compatibili in base al numero di alberi
    piantati e alle penalità
            Collections.sort(compatibleFarmers, new Comparator<Farmer>() {
                public int compare(Farmer f1, Farmer f2) {
                    int penalties1 = f1.getPenalties();
                    int penalties2 = f2.getPenalties();

                    // Ordinamento in base alle penalità
                    if (penalties1 != penalties2) {
                        return penalties1 - penalties2;
                    }

                    return 0;
                }
            });

            // Assegniamo l'albero al contadino con penalità minore e meno
    alberi assegnati
            if (!compatibleFarmers.isEmpty()) {
                Farmer assignedFarmer = compatibleFarmers.get(0);
                incrementAssignedTreeCount(assignedTrees, assignedFarmer);
    // Incrementiamo il numero di alberi assegnati al contadino
                System.out.println("L'albero " + tree.getId() + " è stato
    assegnato al contadino " + assignedFarmer.getId());
            } else {
                System.out.println("Nessun contadino idoneo per l'albero " +
     tree.getId());
            }
        }
    }

    private static int getAssignedTreeCount(Map<Farmer, Integer>
    assignedTrees, Farmer farmer) {
        Integer count = assignedTrees.get(farmer);
        return count != null ? count : 0;
    }

    private static void incrementAssignedTreeCount(Map<Farmer, Integer>
    assignedTrees, Farmer farmer) {
        int count = getAssignedTreeCount(assignedTrees, farmer);
        assignedTrees.put(farmer, count + 1);
    }
}
```

Il codice inizia dichiarando una mappa *assignedTrees* per tenere traccia del numero di alberi assegnati a ciascun contadino. Successivamente, viene iterato su ogni albero presente nella lista *trees*. Per ogni albero, viene creato un elenco di contadini compatibili basati sulla loro nazionalità e sul numero di alberi già assegnati. I contadini compatibili vengono ordinati in base alle penalità. Se ci sono contadini compatibili, viene assegnato l'albero al contadino con penalità minore e viene visualizzato un messaggio di assegnazione. Se non ci sono contadini compatibili, viene visualizzato un messaggio indicando che non ci sono contadini idonei per quell'albero.

La complessità dell'algoritmo dipende dal numero di alberi e contadini presenti. In generale, per ogni albero, l'algoritmo cerca i contadini compatibili e successivamente li ordina. La complessità di

questa parte dell'algoritmo dipende dal numero di contadini compatibili per ogni albero. Nel caso peggiore, in cui ogni contadino è compatibile con ogni albero, la complessità diventa $O(n^2)$, dove n è il numero di contadini.

É importante notare che l'algoritmo Best Fit Greedy potrebbe non garantire la soluzione ottimale, in quanto considera solo la scelta ottimale locale in ogni passo e non tiene conto dell'ottimizzazione globale.

2.1.2 Algoritmo A*

L'algoritmo A* combina la ricerca in ampiezza con una funzione euristica per guidare la ricerca in direzione del percorso più promettente, utilizza due valori per ogni nodo del grafo: **g(n)**, che rappresenta il costo effettivo del percorso dal nodo iniziale fino al nodo corrente, e **h(n)**, che rappresenta una stima del costo rimanente per raggiungere il nodo di destinazione.

Nel caso specifico del problema, l'algoritmo A* può essere utilizzato per trovare l'assegnazione ottimale dei contadini agli alberi, tenendo conto delle restrizioni di compatibilità tra contadini e alberi. L'obiettivo è minimizzare il costo totale dell'assegnazione.

A* inizia dalla configurazione iniziale (nessun albero assegnato) e genera successivamente le assegnazioni considerando tutte le possibili combinazioni tra contadini e alberi. Utilizza la funzione euristica (calcolata tramite la somma del numero di alberi assegnati a ciascun contadino) per stimare il costo rimanente per raggiungere la soluzione ottimale. Durante la ricerca, tiene traccia dei nodi visitati e sceglie i nodi successivi in base al costo stimato più basso (g(n) + h(n)).

L'obiettivo finale è trovare l'assegnazione ottimale che minimizza il costo totale dell'assegnazione, considerando sia il numero di alberi assegnati a ciascun contadino che le penalità associate a ciascun contadino.

L'euristica (h(n)) stima il costo rimanente per raggiungere il nodo di destinazione, ovvero la configurazione finale in cui tutti gli alberi sono assegnati ai contadini in modo ottimale.

Per calcolare l'euristica si considera la configurazione corrente degli assegnamenti, in cui alcuni contadini hanno già alberi assegnati. Si somma il numero di alberi assegnati a ciascun contadino, considerando tutti i contadini presenti nella configurazione.

L'algoritmo A* termina quando trova una soluzione in cui tutti gli alberi sono stati assegnati ai contadini o quando esplora tutti i possibili nodi senza trovare una soluzione. Restituisce quindi la configurazione con l'assegnazione ottimale degli alberi ai contadini.

```
class State {
    Map<Farmer, List<Tree>> assignment; // Assegnazione degli alberi ai
    contadini

    public State(Map<Farmer, List<Tree>> assignment) {
        this.assignment = assignment;
    }

    public boolean isGoalState(List<Tree> trees) {
        // Verifica se tutti gli alberi sono stati assegnati
        for (Tree tree : trees) {
            boolean isAssigned = false;
            for (List<Tree> assignedTrees : assignment.values()) {
```

```
                if (assignedTrees.contains(tree)) {
                    isAssigned = true;
                    break;
                }
            }
            if (!isAssigned) {
                return false;
            }
        }
        return true;
    }
}

class Node implements Comparable<Node> {
    State state;
    Node parent;
    int gCost; // Costo g
    int hCost; // Costo h

    public Node(State state, int gCost, int hCost, Node parent) {
        this.state = state;
        this.gCost = gCost;
        this.hCost = hCost;
        this.parent = parent;
    }

    public State getState() {
        return state;
    }

    public Node getParent() {
        return parent;
    }

    public int getCost() {
        return gCost + hCost;
    }

    @Override
    public int compareTo(Node other) {
        return Integer.compare(getCost(), other.getCost());
    }
}

public class TreeAssignmentAStar {

    public static Map<Farmer, List<Tree>> findOptimalAssignment(List<Farmer>
     farmers, List<Tree> trees) {
        // Inizializzazione dello stato iniziale
        Map<Farmer, List<Tree>> initialState = new HashMap<>();
        for (Farmer farmer : farmers) {
            initialState.put(farmer, new ArrayList<>());
        }

        // Creazione del nodo iniziale
        Node initialNode = new Node(new State(initialState), 0,
```

```
    calculateHeuristic(initialState, trees), null);

        // Inizializzazione delle strutture dati
        PriorityQueue<Node> openList = new PriorityQueue<>();
        Set<State> closedSet = new HashSet<>();

        // Aggiunta del nodo iniziale alla coda prioritaria
        openList.add(initialNode);

        while (!openList.isEmpty()) {
            // Estrazione del nodo con il costo più basso dalla coda
    prioritaria
            Node currentNode = openList.poll();
            State currentState = currentNode.getState();

            // Verifica se lo stato corrente è una soluzione
            if (currentState.isGoalState(trees)) {
                return currentState.assignment;
            }

            // Aggiunta dello stato corrente all'insieme dei nodi visitati
            closedSet.add(currentState);

            // Generazione dei successori
            for (Tree tree : trees) {
                if (!isTreeAssigned(currentState.assignment, tree)) {
                    Map<Farmer, List<Tree>> newState = new HashMap<>(
    currentState.assignment);

                    List<Farmer> eligibleFarmers = new ArrayList<>();

                    // Trova i contadini eleggibili con penalità simili
                    int minPenalties = Integer.MAX_VALUE;
                    for (Farmer farmer : farmers) {
                        if (farmer.getCountry().equals(tree.getCountry()) &&
     newState.get(farmer).size() < 4) {
                            if (farmer.getPenalties() < minPenalties ) {
                                eligibleFarmers.clear();
                                minPenalties = farmer.getPenalties();
                            }

                            if (farmer.getPenalties() == minPenalties) {
                                eligibleFarmers.add(farmer);
                            }
                        }
                    }

                    // Assegna l'albero al contadino con meno alberi
    assegnati
                    Farmer bestFarmer = null;
                    int minAssignedTrees = Integer.MAX_VALUE;

                    for (Farmer farmer : eligibleFarmers) {
                        int assignedTrees = newState.get(farmer).size();
                        if (assignedTrees < minAssignedTrees) {
                            bestFarmer = farmer;
```

```
                        minAssignedTrees = assignedTrees;
                    }
                }

                if (bestFarmer != null && isValidAssignment(currentState
.assignment, bestFarmer, tree)) {
                    newState.get(bestFarmer).add(tree);
                    int gCost = currentNode.gCost + 1;
                    int hCost = calculateHeuristic(newState, trees);
                    Node newNode = new Node(new State(newState), gCost,
hCost, currentNode);

                    if (!closedSet.contains(newNode.getState())) {
                        openList.add(newNode);
                    }
                }
            }
        }
    }

    // Nessuna soluzione trovata
    return null;
}

private static boolean isTreeAssigned(Map<Farmer, List<Tree>> assignment
, Tree tree) {
    for (List<Tree> assignedTrees : assignment.values()) {
        if (assignedTrees.contains(tree)) {
            return true;
        }
    }
    return false;
}

private static int calculateHeuristic(Map<Farmer, List<Tree>> assignment
, List<Tree> trees) {
    int totalAssignedTrees = 0;
    int maxPenalty = 0;
    int penalty = 0;

    for (List<Tree> assignedTrees : assignment.values()) {
        totalAssignedTrees += assignedTrees.size();
    }

    for (Farmer farmer : assignment.keySet()) {
        int farmerPenalty = farmer.getPenalties();
        int assignedTreesCount = assignment.get(farmer).size();


        if (farmerPenalty > maxPenalty) {
            maxPenalty = farmerPenalty;
        }


        penalty -= farmerPenalty;
```

```
            penalty -= assignedTreesCount;
        }

        penalty -= (maxPenalty * 10);

        return totalAssignedTrees + penalty;
    }


    private static boolean isValidAssignment(Map<Farmer, List<Tree>>
    assignment, Farmer farmer, Tree tree) {
        // Verifica se il contadino si trova nel luogo adatto alla crescita
    dell'albero
        if (!farmer.getCountry().equals(tree.getCountry())) {
            return false;
        }

        // Verifica se l'albero è già stato assegnato a un contadino diverso
        for (List<Tree> assignedTrees : assignment.values()) {
            if (assignedTrees.contains(tree)) {
                return false;
            }
        }

        return true;
    }
```

La classe *State* rappresenta uno stato nel contesto del problema, contiene l'assegnazione corrente degli alberi ai contadini, rappresentata una mappa in cui le chiavi sono i contadini e i valori sono le liste degli alberi assegnati a ciascun contadino.

La classe *Node* rappresenta un nodo, contiene un riferimento allo stato corrente, un riferimento al nodo genitore (per tracciare il percorso)e i costi g e h (gCost e hCost) e fornisce metodi per accedere a queste informazioni. Il costo g (gCost) rappresenta il costo effettivo del percorso dal nodo iniziale fino al nodo corrente. Il costo h (hCost) rappresenta una stima del costo rimanente per raggiungere il nodo di destinazione, calcolato tramite un'euristica.

Il processo di assegnazione avviene attraverso la generazione di uno spazio degli stati, dove ogni stato rappresenta un'assegnazione parziale di alberi ai contadini. L'algoritmo esamina uno stato alla volta, generando successori che corrispondono ad assegnazioni aggiuntive di alberi. La scelta dei successori avviene in base a due criteri principali: penalità del contadino e numero di alberi già assegnati.

Durante l'esecuzione dell'algoritmo, viene utilizzata una coda prioritaria (*openList*) per mantenere i nodi da esaminare in ordine di costo ($gCost + hCost$), dove $gCost$ rappresenta il costo effettivo dell'assegnazione corrente e $hCost$ rappresenta una stima del costo rimanente basata su una funzione euristica. L'algoritmo continua ad esaminare i nodi finché non trova uno stato obiettivo in cui tutti gli alberi sono stati assegnati correttamente a ciascun contadino o finché non esaurisce tutti i nodi da esaminare senza trovare una soluzione. Alla fine dell'esecuzione, viene restituita l'assegnazione ottimale rappresentata da una mappa che associa a ciascun contadino la lista degli alberi assegnati.

L'algoritmo A* è preferibile a un algoritmo "Best Fit Greedy" nel contesto dell'assegnazione degli alberi ai contadini per diversi motivi:

1. **Ottimalità:** A* è un algoritmo completo e ottimo, il che significa che è garantito trovare la soluzione ottimale se esiste; al contrario, un algoritmo greedy potrebbe fornire una soluzione subottimale, poiché si basa su scelte localmente ottimali senza considerare l'intero spazio di ricerca.
2. **Euristiche:** A* utilizza una funzione euristica per guidare la ricerca verso le soluzioni più promettenti. Le euristiche forniscono una stima del costo rimanente per raggiungere la soluzione ottimale, consentendo ad A* di fare scelte più intelligenti nella ricerca. Un algoritmo "best fit" greedy, d'altra parte, si basa solo sulla disponibilità immediata e cerca di assegnare gli alberi al contadino con meno punti di penalità, senza considerare il costo complessivo dell'assegnazione.
3. **Complessità:** A* ha una complessità temporale maggiore rispetto all'algoritmo "Best Fit Greedy", poiché esplora un numero potenzialmente maggiore di stati nel grafo di ricerca. Tuttavia, la sua capacità di guidare la ricerca in direzione del percorso ottimale compensa questa maggiore complessità.

2.2 Algoritmi di ricerca locale

Gli algoritmi di ricerca locale cercano di migliorare iterativamente una soluzione corrente, spostandosi in direzione di soluzioni migliori nel vicinato della soluzione corrente. Tuttavia, va notato che gli algoritmi di ricerca locale non garantiscono di trovare la soluzione ottimale globale. Possono rimanere intrappolati in ottimi locali, che sono soluzioni che sembrano buone nel vicinato ma non sono ottimali rispetto alla soluzione globale. Pertanto, l'efficacia di un algoritmo di ricerca locale dipende fortemente dalla natura del problema e dalle caratteristiche dello spazio di ricerca.

Nel caso del problema, un algoritmo di ricerca locale è una buona opzione poiché ci basta anche ottenere una soluzione soddisfacente in tempi ragionevoli senza la necessità di trovare la soluzione ottimale assoluta. Questo tipo di ricerca può comunque rivelarsi utile poiché il nostro obiettivo è proporre un suggerimento al responsabile degli ordini in tempi ragionevoli per migliorare la sua esperienza utente.

2.2.1 Algoritmi genetici

Gli algoritmi genetici sono ispirati al processo di selezione naturale e utilizzano concetti come la selezione, la mutazione e il crossover per trovare soluzioni ottimali a problemi di ottimizzazione.

Rappresentazione degli individui

Inizialmente potremmo pensare di rappresentare un individuo come una matrice n per m, dove n rappresenta il numero di contadini e m è il numero di alberi da allocare. Assegniamo il valore 1 alla cella [i,j] per indicare che al contadino i è stato assegnato l'albero j, al contrario il valore 0 indica che il contadino i non ha ricevuto in carico l'albero j. 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, perché un albero verrà assegnato ad un unico contadino, quindi per ogni colonna avremo un unico valore pari ad 1 e il restante a pari a 0. Quindi, possiamo individuare una rappresentazione più compatta degli individui che sprechi meno spazio di memoria.

Un individuo è rappresentato da un array di dimensione n, dove n è il numero di alberi da assegnare. L'i-esima cella dell'array conterà l'ID che identifica il contadino a cui viene assegnato l' i-esimo albero. Ad esempio, una possibile soluzione potrebbe essere '[1, 2, 2, 3, 1]', che indica che il primo albero è assegnato al contadino 1, il secondo e il terzo albero sono assegnati al contadino 2 e così via. In questo caso la popolazione sarà formata da array di dimensione n.

Tuttavia, per ottenere l'elenco degli alberi assegnati a ciascun contadino, la rappresentazione con una mappa è più adatta. Nella rappresentazione con la mappa, ogni chiave rappresenta un contadino e il valore associato è una lista di alberi assegnati a quel contadino. Ad esempio, una possibile soluzione potrebbe essere:

{ Contadino 1: [Albero 1, Albero 5], Contadino 2: [Albero 2, Albero 3, Albero 4], Contadino 3: [Albero 6] }

Questa rappresentazione può essere vantaggiosa per accedere facilmente agli alberi assegnati a ciascun contadino. Notiamo che nel problema che stiamo affrontando la size degli individui non è fissata a priori ma varia in base al numero di contadini da assegnare.

Inizializzazione

Descriviamo ora con quale criterio si crea la prima generazione di individui. Potremmo pensare di creare una popolazione iniziale di individui completamente casuali, ma questo approccio risulta errato per il nostro problema poiché potrebbe portare alla generazione di individui inammissibili.

Dobbiamo perciò imporre delle regole per la generazione degli individui, in particolare dobbiamo verificare che siano rispettati i seguenti vincoli:

1. Garantire che ogni albero venga assegnato solo a un contadino nello stesso paese;
2. Garantire che ogni albero venga assegnato a un solo contadino;
3. Garantire che nessun contadino riceva più di 4 alberi;

Se un individuo rispetta entrambe le condizioni viene detto valido. La generazione degli individui viene quindi effettuata in maniera semi-casuale, in quanto si sceglie in modo casuale tra le coppie contadino-albero valide.

Di seguito l'implementazione del metodo che permette di generare gli individui pseudo-casuali:

```
private static Chromosome generateRandomChromosome(List<Farmer> farmers,
    List<Tree> trees) {
    List<Tree> availableTrees = new ArrayList<>(trees);
    Map<Farmer, List<Tree>> assignment = new HashMap<>();

    for (Farmer farmer : farmers) {
        assignment.put(farmer, new ArrayList<>());
    }

    Random random = new Random();
    int assignedTreesCount = 0;

    while (!availableTrees.isEmpty() && assignedTreesCount < trees.size()) {
        Tree selectedTree = null;
        Farmer selectedFarmer = null;

        // Cerca un albero disponibile che non sia stato ancora assegnato
```

```
        for (Tree tree : availableTrees) {
            boolean isAssigned = false;

            // Controlla se l'albero è già stato assegnato a un contadino
            for (List<Tree> assignedTrees : assignment.values()) {
                if (assignedTrees.contains(tree)) {
                    isAssigned = true;
                    break;
                }
            }

            if (!isAssigned) {
                selectedTree = tree;
                break;
            }
        }

        // Cerca un contadino disponibile con lo stesso paese dell'albero
    selezionato
        if (selectedTree != null) {
            List<Farmer> availableFarmers = new ArrayList<>();
            for (Farmer farmer : farmers) {
                if (farmer.getCountry().equals(selectedTree.getCountry()) &&
     assignment.get(farmer).size() < 4) {
                    availableFarmers.add(farmer);
                }
            }

            if (!availableFarmers.isEmpty()) {
                int randomIndex = random.nextInt(availableFarmers.size());
                selectedFarmer = availableFarmers.get(randomIndex);
            }
        }

        if (selectedTree != null && selectedFarmer != null) {
            assignment.get(selectedFarmer).add(selectedTree);
            availableTrees.remove(selectedTree);
            assignedTreesCount++;
        }
    }

    return new Chromosome(assignment);
}
```

Il metodo prende in input una lista di contadini e una lista di alberi disponibili. Inizialmente, vengono creati un elenco di alberi disponibili e una mappa di assegnazione che tiene traccia di quali alberi sono stati assegnati a ciascun contadino. Successivamente, viene utilizzato un ciclo per assegnare gli alberi ai contadini fino a quando tutti gli alberi disponibili sono stati assegnati. All'interno del ciclo, viene selezionato un albero disponibile che non è ancora stato assegnato controllando la mappa di assegnazione. Successivamente, viene cercato un contadino disponibile che abbia lo stesso paese dell'albero selezionato. Se viene trovato un contadino idoneo, l'albero viene assegnato a quel contadino, viene aggiornata la mappa di assegnazione e l'albero viene rimosso dall'elenco degli alberi disponibili. Infine, viene restituito il cromosoma creato con l'assegnazione degli alberi ai contadini.

Funzione obiettivo

L'obiettivo è distribuire gli alberi in modo equo, al più 4 alberi, tra i contadini, dando la precedenza a coloro che hanno il minor penalità. Il valore della fitness viene quindi cosi calcolato:

- Se il contadino ha meno di 4 alberi assegnati, viene incrementato il valore della fitness. Questo indica che assegnare meno di 4 alberi ai contadini con penalità minima viene considerato positivamente.
- Se il contadino ha esattamente 4 alberi assegnati, viene incrementato il valore della fitness di 5. Questo premia i contadini con penalità minima che hanno ricevuto esattamente 4 alberi.
- Se il contadino ha più di 4 alberi assegnati, viene decrementato il valore della fitness. Questo indica che assegnare più di 4 alberi ai contadini con penalità minima viene considerato negativamente.

La soluzione ottima è data dalla soluzione che assegna al più 4 alberi ai contadini ordinati in base alla penalità, quindi rappresenta il cromosoma con valore di fitness massimo.

Di seguito vediamo l'implementazione del calcolo della fitness:

```
private static void evaluateFitness(List<Chromosome> population, List<Tree>
    trees) {
    for (Chromosome chromosome : population) {
        Map<Farmer, List<Tree>> assignment = chromosome.getAssignment();
        double fitness = 0.0;

        // Calcola la penalità minima per ogni paese
        Map<String, Integer> minPenaltiesByCountry = new HashMap<>();
        for (Farmer farmer : assignment.keySet()) {
            String country = farmer.getCountry();
            int farmerPenalties = farmer.getPenalties();

            if (!minPenaltiesByCountry.containsKey(country) ||
    farmerPenalties < minPenaltiesByCountry.get(country)) {
                minPenaltiesByCountry.put(country, farmerPenalties);
            }
        }

        // Calcola il numero di contadini per ogni paese con la penalità
    minima
        Map<String, Integer> farmersWithMinPenaltyByCountry = new HashMap
    <>();
        for (Farmer farmer : assignment.keySet()) {
            String country = farmer.getCountry();
            int farmerPenalties = farmer.getPenalties();
            int assignedTrees = assignment.get(farmer).size();

            if (farmerPenalties == minPenaltiesByCountry.get(country)) {
                if (Math.abs(assignedTrees) < 4) {
                    fitness++;
                } else if (assignedTrees == 4) {
                    fitness += 5;
                    farmersWithMinPenaltyByCountry.put(country,
    farmersWithMinPenaltyByCountry.getOrDefault(country, farmerPenalties));
                } else if (assignedTrees > 4) {
                    fitness--;
                }
```

```
            }
        }

        chromosome.setFitness(fitness);
    }
}
```

Per ogni cromosoma, si accede all'assegnazione dei contadini e si inizializza il punteggio di fitness a 0.0. Successivamente, vengono eseguiti due cicli separati. Nel primo ciclo, si calcola la penalità minima per ogni paese tenendo conto dei contadini presenti nell'assegnazione del cromosoma. Queste informazioni vengono memorizzate nella mappa *minPenaltiesByCountry*, in cui la chiave è il paese e il valore è la penalità minima. Nel secondo ciclo, si confronta la penalità di ciascun contadino con la penalità minima del rispettivo paese e si calcola la fitness secodno le regole specificate prima.

Algoritmo di selezione

La fase di selezione prevede di selezionare una parte della popolazione attuale come genitori per la generazione successiva. L'algoritmo di selezione scelto è **roulette wheel** che seleziona i genitori in base alla loro fitness. Gli individui con una fitness più alta avranno una maggiore probabilità di essere selezionati, ma non escluderà completamente gli individui con fitness più bassa. Questo permette di mantenere una certa diversità genetica nella popolazione e di evitare di rimanere bloccati in minimi locali. Inoltre, l'algoritmo della roulette wheel è relativamente semplice da implementare e comprendere. Non richiede complessi calcoli o strutture dati aggiuntive, rendendo l'implementazione più facile e il codice più leggibile. Infine, può essere implementato in modo efficiente senza richiedere risorse computazionali eccessive. Questo aspetto è molto importante, in quanto l'algoritmo viene eseguito all'interno di un sito e-commerce per cui è importante avere una bassa latenza per migliorare l'esperienza utente.

L'unico svantaggio dall'uso di questo algoritmo è la convergenza prematura, ma la diversità genetica nella popolazione è mantenuta attraverso l'utilizzo di operatori genetici come la mutazione e il crossover, le soluzioni con lo stesso valore di fitness possono presentare una variazione significativa nella loro composizione genetica. Questo permette una continua esplorazione dello spazio delle soluzioni e riduce il rischio di convergenza prematura verso un singolo ottimo locale.

Osserviamo però che nel problema in esame, la convergenza prematura non è un problema significativo. Ciò è dovuto al fatto che, una volta raggiunto un determinato valore di fitness massimo per diverse soluzioni, le differenze tra di loro sono principalmente nel modo in cui gli alberi sono assegnati ai contadini. Tuttavia, è importante notare che la convergenza prematura potrebbe comunque verificarsi se l'algoritmo di selezione non è in grado di mantenere una diversità genetica sufficiente o se gli operatori genetici non sono adeguatamente configurati.

Di seguito riportiamo l'implementazione dell'algoritmo

```
private static List<Chromosome> selectParents(List<Chromosome> population) {
    List<Chromosome> parents = new ArrayList<>();

    double totalFitness = population.stream().mapToDouble(Chromosome::
    getFitness).sum();

    for (int i = 0; i < POPULATION_SIZE / 2; i++) {
        double randomValue = Math.random() * totalFitness;
        double partialSum = 0;
```

```
        int j = 0;

        while (partialSum < randomValue && j < population.size()) {
            partialSum += population.get(j).getFitness();
            j++;
        }

        if (j > 0) {
            parents.add(population.get(j - 1));
        }
    }

    return parents;
}
```

Crossover

La strategia di crossover è **single point**. In questa strategia, viene selezionato casualmente un punto di taglio lungo la sequenza genetica dei cromosomi genitori. I geni prima del punto di taglio vengono presi dal primo genitore, mentre i geni dopo il punto di taglio vengono presi dal secondo genitore.

Il crossover a un punto consente una certa esplorazione dello spazio delle soluzioni, poiché i punti di taglio possono variare tra diverse iterazioni dell'algoritmo. Ciò significa che le soluzioni generate attraverso il crossover a un punto possono differire in modo significativo, consentendo una maggiore diversità genetica nella popolazione e una migliore esplorazione delle possibili soluzioni.

Inoltre, come nel caso dell'algoritmo di selezione, garantisce efficienza computazionale: è relativamente efficiente dal punto di vista computazionale, poiché richiede un numero limitato di operazioni per generare i cromosomi figli.

```
private static Chromosome crossover(Chromosome parent1, Chromosome parent2)
    {
    Map<Farmer, List<Tree>> assignment1 = parent1.getAssignment();
    Map<Farmer, List<Tree>> assignment2 = parent2.getAssignment();
    Map<Farmer, List<Tree>> childAssignment = new HashMap<>();
    Set<Tree> assignedTrees = new HashSet<>();

    for (Farmer farmer : assignment1.keySet()) {
        List<Tree> trees1 = assignment1.get(farmer);
        List<Tree> trees2 = assignment2.get(farmer);
        List<Tree> childTrees = new ArrayList<>();

        for (Tree tree : trees1) {
            if (!childTrees.contains(tree) && !assignedTrees.contains(tree))
    {
                childTrees.add(tree);
                assignedTrees.add(tree);
            }
        }

        for (Tree tree : trees2) {
            if (!childTrees.contains(tree) && !assignedTrees.contains(tree))
    {
                childTrees.add(tree);
                assignedTrees.add(tree);
```

```
            }
        }

        childAssignment.put(farmer, childTrees);
    }

    return new Chromosome(childAssignment);
}
```

Il processo di crossover avviene combinando le caratteristiche dei genitori per creare una nuova assegnazione di alberi ai contadini. Il nuovo cromosoma figlio ha una mappa di assegnazione *childAssignment* che rappresenta l'assegnazione degli alberi ai contadini.

Nel dettaglio, per ogni contadino presente nei genitori, vengono considerati gli insiemi di alberi assegnati da entrambi i genitori. I geni corrispondenti a tali insiemi vengono combinati per formare gli alberi assegnati al contadino nel cromosoma figlio.

Durante la combinazione, vengono rispettate due condizioni:

- Un albero non può essere assegnato due volte nello stesso cromosoma figlio.
- Ogni contadino può avere al massimo 4 alberi assegnati.

L'algoritmo scorre gli alberi assegnati da entrambi i genitori, li aggiunge al cromosoma figlio solo se non sono già presenti nell'insieme degli alberi del contadino e se il numero massimo di alberi per contadino non è stato raggiunto.

Alla fine del processo, il cromosoma figlio con l'assegnazione combinata viene restituito come risultato.

Mutazione

La mutazione avviene tramite **swap**. Lo swap tra gli alberi assegnati a due contadini diversi compatibili (ossia dello stesso paese, per evitare di generare individui non validi) consente di introdurre una variazione nella configurazione delle soluzioni. Questo aiuta a esplorare nuove regioni dello spazio delle soluzioni e ad evitare la convergenza prematura verso un ottimo locale. La variazione è fondamentale per garantire una diversità genetica nella popolazione e aumentare le possibilità di scoprire soluzioni migliori. L'operatore di mutazione swap è relativamente semplice da implementare. Richiede solo la selezione di alberi da scambiare e la scelta di un altro contadino come destinazione dello scambio. La sua semplicità lo rende facilmente integrabile all'interno dell'algoritmo genetico e richiede un basso costo computazionale. La scelta dell'operatore di mutazione è stata guidata dal fatto che altri tipi di mutazione potrebbero introdurre individui non validi nel contesto del problema. Ad esempio, se consideriamo una mutazione che modifica casualmente l'assegnazione di un albero da un contadino a un altro, potrebbe verificarsi il caso in cui l'assegnazione risultante violi una o più restrizioni del problema.

```
private static void mutate(Chromosome chromosome) {
    Map<Farmer, List<Tree>> assignment = chromosome.getAssignment();

    for (Farmer farmer : assignment.keySet()) {
        List<Tree> trees = assignment.get(farmer);

        for (int i = 0; i < trees.size(); i++) {
            if (Math.random() < MUTATION_RATE) {
```

```
                Tree tree1 = trees.get(i);

                Farmer randomFarmer = getRandomFarmerExcept(farmer,
    assignment.keySet());
                List<Tree> randomFarmerTrees = assignment.get(randomFarmer);

                if (!randomFarmerTrees.contains(tree1) && !randomFarmerTrees
    .isEmpty() && trees.size() < 4 && randomFarmerTrees.size() < 4) {
                    trees.remove(tree1);
                    randomFarmerTrees.add(tree1);

                    if (randomFarmerTrees.size() >= 2) {
                        int randomIndex = (int) (Math.random() *
    randomFarmerTrees.size());
                        Tree tree2 = randomFarmerTrees.get(randomIndex);

                        randomFarmerTrees.remove(tree2);
                        randomFarmerTrees.add(tree1);

                        if (i < trees.size()) {
                            trees.set(i, tree2);
                        } else {
                            trees.add(tree2);
                        }
                    } else {
                        randomFarmerTrees.remove(tree1);
                        trees.add(tree1);
                    }
                }
            }
        }
    }
}
```

Inizialmente, si itera attraverso gli alberi assegnati a ciascun contadino nel cromosoma in esame. Per ogni albero, si genera un numero casuale compreso tra 0 e 1 e si confronta con una probabilità di mutazione prestabilita MUTATION-RATE. Se il numero casuale è inferiore alla probabilità di mutazione, viene eseguita la mutazione sull'albero corrente.

Durante la mutazione, l'albero viene rimosso dall'assegnazione del contadino corrente. Successivamente, viene selezionato casualmente un altro contadino (diverso dal contadino corrente) all'interno del cromosoma. L'albero mutato viene quindi aggiunto all'assegnazione del contadino selezionato casualmente.

Per garantire la validità della soluzione, vengono applicate alcune condizioni. Prima di tutto, l'albero mutato non deve essere già presente nell'assegnazione del contadino selezionato casualmente. Inoltre, entrambi i contadini coinvolti nella mutazione devono avere un numero di alberi inferiore al limite massimo consentito (4 alberi).

Inoltre, viene eseguito uno scambio aggiuntivo tra gli alberi del contadino corrente e il contadino selezionato casualmente. Se il contadino selezionato casualmente ha almeno due alberi nella sua assegnazione, viene scelto casualmente uno di questi alberi e viene effettuato uno scambio con l'albero mutato. Questo scambio aiuta a diversificare ulteriormente la soluzione. Tuttavia, se il

contadino selezionato casualmente ha solo un albero, l'albero mutato viene restituito all'assegnazione del contadino corrente.

Stopping condition

La stopping condition (condizione di arresto) è la condizione che determina quando fermare l'esecuzione dell'algoritmo genetico. La stopping condition scelta è rappresentata dal numero massimo di generazioni (MAX GENERATION). L'algoritmo viene eseguito per un massimo di MAX GENERATION iterazioni, quindi viene interrotto e viene restituito il risultato corrente. Limitare il numero di iterazioni aiuta a ottimizzare l'uso delle risorse computazionali. Inoltre come abbiamo già ampiamente descritto il nostro l'algoritmo genetico viene utilizzato in un contesto in cui è richiesta una risposta tempestiva ed impostare un numero massimo di iterazioni può garantire che l'algoritmo produca un risultato entro un tempo ragionevole.

Parametri dell'algoritmo

La scelta dei parametri è fondamentale per il corretto funzionamento e l'efficacia dell'algoritmo genetico. Vediamo perché ciascuno di essi è importante:

1. MAX GENERATION (Massimo numero di generazioni): Questo parametro determina il numero massimo di iterazioni o generazioni che l'algoritmo genetico esegue prima di terminare. Una scelta appropriata per MAX GENERATION dipende dalla complessità del problema e dalla velocità di convergenza dell'algoritmo. Se il valore di MAX GENERATION è troppo basso, l'algoritmo potrebbe terminare prematuramente senza raggiungere una soluzione ottimale. Al contrario, se è troppo alto, l'algoritmo potrebbe richiedere un tempo e una computazione eccessivi senza apportare miglioramenti significativi. Pertanto, è importante scegliere un valore che consenta all'algoritmo di convergere verso una buona soluzione entro un tempo ragionevole.
2. MUTATION RATE (Tasso di mutazione): La mutazione è un'operazione che introduce una variazione casuale nei geni dei cromosomi. Il tasso di mutazione rappresenta la probabilità che un gene di un cromosoma subisca una mutazione. Una mutazione occasionalmente introduce nuove soluzioni nel pool genetico, consentendo all'algoritmo di esplorare lo spazio delle soluzioni in modo più ampio. Un tasso di mutazione troppo basso potrebbe limitare la diversità genetica e rallentare la ricerca di soluzioni migliori. D'altra parte, un tasso di mutazione troppo alto potrebbe causare una rapida perdita delle soluzioni ottime trovate in precedenza. È importante trovare un equilibrio tra l'esplorazione e lo sfruttamento delle soluzioni, adattando il tasso di mutazione alla specificità del problema e alla fase di evoluzione dell'algoritmo.
3. POPULATION SIZE (Dimensione della popolazione): La dimensione della popolazione rappresenta il numero di cromosomi (soluzioni) che compongono una generazione iniziale. Una popolazione più grande aumenta la diversità genetica e fornisce una base più solida per la selezione naturale e l'evoluzione. Una popolazione più piccola potrebbe ridurre la capacità di esplorazione e rallentare la convergenza verso una soluzione ottimale. D'altra parte, una popolazione troppo grande può richiedere maggiori risorse computazionali e aumentare il tempo di esecuzione dell'algoritmo. La scelta della dimensione della popolazione dipende dalla complessità del problema e dalle risorse disponibili. È importante bilanciare la diversità genetica e l'efficienza computazionale selezionando una dimensione della popolazione adeguata.

In virtù, di queste considerazioni i valori dei parametri scelti sono: POPULATION SIZE = 20, MUTATION RATE = 0.9, MAX GENERATION = 150, che permettono di esplorare adeguatamente lo spazio delle soluzioni ma mentendo un basso tempo di esecuzione. Prima di arrivare a tali valori per i parametri sono stati sperimentati vari combinazioni (ciascuna combinazione è stata testa 20 volte):

- POPULATION SIZE = 10, MUTATION RATE = 0.5, MAX GENERATION = 100, ha portato al valore massimo di fitness pari a 16, mentre gli altri cromosomi avevano un valori di fitness compreso tra 3 e 12
- POPULATION SIZE = 15, MUTATION RATE = 0.6, MAX GENERATION = 120, ha portato, nuovamente, al valore massimo di fitness pari a 16, mentre gli altri cromosomi avevano un valori di fitness compreso tra 12 e 16
- POPULATION SIZE =20, MUTATION RATE = 0.9, MAX GENERATION = 150, ha portato, nuovamente, al valore massimo di fitness pari a 16, mentre gli altri cromosomi avevano un valori di fitness compreso tra 3 e 16

Risultati ottenuti

L'algoritmo genetico creato, in alcune iterazioni, si blocca in un ottimo locale per questo motivo potrebbe essere ulteriormente migliorato esplorando variazioni degli operatori genetici o dei parametri. Osserviamo che il problema che stiamo analizzando prevede l'interazione con l'essere umano per questo motivo una variante possibile per l'algoritmo genetico è un Interactive GA.

Capitolo 3

Conclusioni

3.1 Integrazione con il sistema

L'algoritmo scelto per essere integrato con l'applicativo WoodLot è l'algoritmo A*, in quanto produce sempre una soluzione ottima in un lasso di tempo accettabile per i requisiti dell'applicativo.

L'algoritmo genetico, come già ampiamente discusso, non porta sempre ad una soluzione ottima ma potrebbe bloccarsi in un ottimo locale e porta con se un overhead superiore rispetto ad A*.

3.2 Possibili miglioramenti

L'algoritmo risulta soddisfacente per i risultati ottenuti e per il suo scopo, nonostante l'applicazione di alcune semplificazioni. Queste semplificazioni, in una fase successiva di lavoro, potrebbero essere eliminate per rendere l'algoritmo più aderente alla realtà. Innanzitutto, si potrebbero considerare ulteriori vincoli sull'assegnazione degli alberi: oltre alla penalità potremmo discriminare i contadini anche sulla base del numero di alberi che già attualmente gestiscono, sulla data dell' ultima assegnazione ricevuta e molto altro ancora. Tali vincoli aggiuntivi avrebbe complicato notevolmente il lavoro dovendo considerare funzioni multi-obiettivo.

3.3 Link Utili

Di seguito riportiamo tutti i link usati per la creazione di questo documento:

- Il codice mostrato in questo documento è reperibile alla seguente repository Git-Hub: WoodLot-Fia
- Maggiori dettagli sul funzionamento dell'applicativo WoddLot sono reperibili alla seguente repository Git-Hub: WoodLot
- Il progetto WoodLot trae ispirazione dal sito : Treedom
- Il templeate usato per realizzare questo documento è ispirato dal seguento progretto `https://it.overleaf.com/articles/clustering-the-interstellar-medium/mtthgyyfrdkn`