

Fondamenti di Intelligenza Artificiale - Appunti e Note

Prof. Fabio Palomba

1 Introduzione all'Intelligenza Artificiale

L'Intelligenza Artificiale (IA) è un campo della scienza che mira a creare sistemi in grado di eseguire compiti che richiedono intelligenza umana. Il suo sviluppo ha portato innovazioni in numerosi settori, dall'automazione industriale alla personalizzazione di contenuti nei social media.

Esempi di applicazioni comuni includono:

- **Google Maps:** Fornisce percorsi ottimizzati tenendo conto del traffico in tempo reale.
- **Sistemi di raccomandazione:** Utilizzati da piattaforme come Netflix o YouTube per suggerire contenuti personalizzati.

L'IA è sviluppata per migliorare diversi aspetti della vita e della società, risolvendo problemi complessi che sono difficili da affrontare manualmente. Le motivazioni principali per lo sviluppo di IA includono:

- **Ottimizzazione delle risorse:** Permettere una gestione più efficiente delle risorse.
- **Velocità di apprendimento:** Creare sistemi che imparano e si adattano più velocemente rispetto agli esseri umani.
- **Risoluzione di problemi complessi:** Affrontare sfide che richiederebbero troppo tempo e risorse per essere risolte dagli esseri umani.
- **Curiosità scientifica:** Spingere i confini della conoscenza su come l'intelligenza può essere simulata o replicata artificialmente.

1.1 Definizioni di Intelligenza Artificiale

L'IA può essere definita in diversi modi, in base a ciò che si intende per "intelligenza". Esistono quattro principali approcci, ciascuno con una diversa enfasi:

- **Pensare umanamente:** L'obiettivo è simulare i processi cognitivi umani, utilizzando tecniche come la modellazione cognitiva. Si tenta di riprodurre i meccanismi del cervello umano, spesso tramite studi psicologici e neuroscienze.
- **Pensare razionalmente:** Questo approccio si concentra sul pensiero logico e deduttivo, cercando di costruire sistemi che seguano regole formali della logica per risolvere problemi. La logica formale è usata come base per la creazione di algoritmi intelligenti.

- **Agire umanamente:** L'IA viene progettata per replicare il comportamento umano. Questo approccio è alla base del **Test di Turing**, che misura l'intelligenza di una macchina sulla base della sua capacità di imitare un comportamento umano in un'interazione.
- **Agire razionalmente:** L'approccio degli **agenti razionali** si concentra sulla creazione di agenti che agiscono in modo ottimale, basandosi sulle informazioni a loro disposizione. L'obiettivo è prendere decisioni razionali per raggiungere un obiettivo specifico, indipendentemente dal fatto che tali azioni simulino o meno il pensiero umano.

1.2 Il Test di Turing

Il **Test di Turing**, proposto da Alan Turing nel 1950, è uno dei primi tentativi di definire un criterio per l'intelligenza artificiale. Il test si basa su un esperimento chiamato **gioco dell'imitazione**. Durante il test, una persona (interrogatore) interagisce con una macchina e un altro essere umano tramite una tastiera o altri mezzi di comunicazione che escludono l'elemento visivo. L'interrogatore pone domande sia alla macchina che all'essere umano e deve determinare quale dei due è la macchina. Se l'interrogatore non riesce a distinguere tra la macchina e l'umano in un numero significativo di casi, allora si dice che la macchina ha passato il Test di Turing e può essere considerata intelligente.

Il Test di Turing ha avuto un impatto profondo sulla filosofia e sullo sviluppo dell'Intelligenza Artificiale (IA). Le implicazioni del test sono molteplici e riguardano sia l'aspetto tecnico che quello etico e filosofico della definizione di intelligenza.

1. Definizione comportamentale di intelligenza Il Test di Turing si basa su una definizione di intelligenza puramente **comportamentale**. Non si interessa di come la macchina arrivi alle risposte, né della sua capacità di comprendere veramente il contenuto delle sue risposte. Se la macchina può rispondere in modo indistinguibile da un essere umano, si assume che essa sia intelligente. Questo criterio comportamentale ha implicazioni significative:

- Non richiede alla macchina di simulare esattamente i processi cognitivi umani, ma solo di imitare il comportamento umano.
- Introduce una forma pragmatica di intelligenza, in cui ciò che conta non è la natura del pensiero, ma l'effetto che esso ha sulle interazioni esterne.

2. Imitazione vs Comprensione Uno dei principali punti di discussione riguardanti il Test di Turing è la distinzione tra **imitazione** e **comprensione**. Il test valuta la capacità di una macchina di imitare un comportamento intelligente, ma non fornisce alcuna indicazione sul fatto che la macchina comprenda realmente ciò che sta facendo. Questo ha generato diversi dibattiti su temi, come da esempio quelli proposti di seguito:

- **IA forte vs IA debole:** L'IA forte sostiene che le macchine possano sviluppare una vera intelligenza, simile a quella umana, mentre l'IA debole suggerisce che le macchine possono solo imitare l'intelligenza senza comprenderla realmente.

- **Il paradosso della simulazione:** È possibile che una macchina sembri intelligente senza esserlo davvero? Se una macchina può simulare perfettamente l'intelligenza umana, ciò significa che è effettivamente intelligente o è solo un'imitazione sofisticata?

3. Limiti del Test di Turing Sebbene il Test di Turing sia stato influente, presenta alcuni rilevanti limiti, tra cui:

- **Superficialità delle risposte:** Il test misura solo l'abilità della macchina di rispondere in modo convincente, senza prendere in considerazione la profondità o la complessità del ragionamento dietro le risposte.
- **Non misura la comprensione:** Il test non valuta la capacità della macchina di comprendere concetti, emozioni o contesti. Un sistema basato su risposte predefinite potrebbe superare il test senza "pensare" in modo autonomo.
- **Critica di Searle:** John Searle, nel suo famoso *argomento della stanza cinese*, ha contestato il Test di Turing affermando che una macchina può manipolare simboli senza capire il loro significato. Questo suggerisce che la macchina non possieda una vera comprensione, anche se supera il Test di Turing.

4. Implicazioni etiche e filosofiche Il Test di Turing ha sollevato questioni etiche e filosofiche riguardanti l'intelligenza artificiale:

- **Responsabilità delle macchine:** Se una macchina può comportarsi in modo intelligente, chi è responsabile delle sue azioni? Quali sono i limiti dell'autonomia delle macchine?
- **Status delle macchine:** Se una macchina supera il Test di Turing, possiamo considerarla come avente uno status simile a quello umano? Può una macchina avere diritti o responsabilità etiche?
- **Conseguenze per la natura dell'intelligenza:** Il Test di Turing pone la domanda su cosa significhi davvero essere intelligenti. Se l'intelligenza è definita dal comportamento esterno, allora le macchine potrebbero già essere "intelligenti" in alcuni contesti. Tuttavia, se l'intelligenza implica comprensione e coscienza, il test potrebbe non essere sufficiente per definire l'intelligenza.

5. Influenza sulla ricerca IA Il Test di Turing ha avuto un impatto significativo sulla ricerca nel campo dell'IA, orientando l'attenzione degli scienziati verso lo sviluppo di sistemi capaci di imitare il comportamento umano. Tuttavia, con il tempo, la comunità scientifica ha iniziato a spostarsi verso altri criteri più pratici e misurabili di intelligenza, come la razionalità degli agenti e la capacità di risolvere problemi complessi in modo efficiente. Il test ha comunque aperto la strada a questioni più profonde sulla relazione tra comportamento esterno e processi interni di pensiero.

1.3 Dall'imitazione alla razionalità: perché si passa da un approccio all'altro

Nel corso degli anni, si è osservato un passaggio dall'approccio di imitare il comportamento umano (come nel Test di Turing) a quello di costruire agenti razionali che prendono decisioni ottimali. Questo passaggio è motivato da diverse ragioni:

- **Limitazioni nel replicare l'intelligenza umana:** Gli sforzi iniziali si concentravano sullo studio di come gli esseri umani pensano e agiscono. Tuttavia, l'intelligenza umana è complessa e difficile da replicare fedelmente, in quanto include emozioni, intuizioni e altri processi difficili da modellare.
- **Focalizzazione sull'efficienza e l'ottimizzazione:** Man mano che l'IA ha trovato applicazione in problemi complessi del mondo reale, come la pianificazione e l'ottimizzazione, l'enfasi si è spostata dalla semplice imitazione del comportamento umano all'efficienza e all'ottimizzazione del processo decisionale. Questo ha portato allo sviluppo degli **agenti razionali**, che agiscono in modo ottimale in base alle informazioni disponibili.
- **Risultati pratici migliori:** Costruire sistemi basati sul pensiero razionale e sulla logica formale ha prodotto risultati più concreti e utilizzabili in contesti pratici. Gli agenti razionali possono essere impiegati per risolvere problemi complessi, come il routing su reti di grandi dimensioni, con un'efficacia molto maggiore rispetto agli approcci che cercano di imitare il pensiero umano.
- **Flessibilità:** Gli agenti razionali sono più flessibili in quanto non devono simulare esattamente il comportamento umano. Possono agire in modi che gli esseri umani non considererebbero, purché il risultato sia ottimale. Questo li rende più adatti a risolvere problemi di ottimizzazione e pianificazione su larga scala.

L'evoluzione della definizione di IA riflette la transizione da un focus sull'imitazione del pensiero umano verso un'enfasi sull'azione razionale. Questo cambiamento ha portato a significativi progressi nell'efficacia e nell'efficienza degli algoritmi di IA, consentendo loro di risolvere problemi complessi in diversi settori industriali e scientifici.

2 Agenti Intelligenti

Un **agente** è un sistema che percepisce l'ambiente circostante tramite sensori e agisce su di esso tramite attuatori. Gli agenti possono essere classificati in diverse categorie a seconda delle loro capacità e del contesto in cui operano. Gli agenti possono essere:

- **Umani:** I sensori includono occhi, orecchie, ecc.; gli attuatori includono mani, gambe, tratto vocale, ecc.
- **Robotici:** I sensori includono telecamere e telemetri; gli attuatori includono motori e bracci meccanici.
- **Software:** I sensori includono input da tastiere, contenuti di file, pacchetti di rete; gli attuatori includono la visualizzazione delle informazioni, la modifica di file e l'invio di pacchetti di rete.

2.1 Comportamento di un Agente

Il comportamento di un agente è descritto da una **funzione agente**, che mappa una sequenza di percezioni in una specifica azione. In altre parole, la funzione agente determina l'azione che l'agente esegue in base a ciò che ha percepito. Questo può essere rappresentato matematicamente come:

$$\text{Funzione Agente} : P \rightarrow A$$

dove P è l'insieme delle percezioni e A è l'insieme delle azioni. La funzione agente è implementata da un **programma agente**, che riceve come input la percezione corrente e restituisce un'azione da eseguire.

2.2 Agenti Razionali

Un **agente razionale** è un agente che, per ogni sequenza di percezioni, sceglie l'azione che massimizza il valore atteso della sua misura di prestazione, date le informazioni a sua disposizione. La razionalità di un agente dipende da:

- La misura di prestazione adottata.
- La conoscenza pregressa dell'agente sull'ambiente.
- Le azioni che l'agente può compiere.
- La sequenza di percezioni ricevute fino al momento attuale.

Nota: La razionalità non implica onniscienza. Un agente razionale non conosce il risultato effettivo delle sue azioni, ma agisce sulla base delle informazioni che ha. Per migliorare la propria

razionalità, un agente può intraprendere azioni di **information gathering**, ovvero azioni finalizzate alla raccolta di nuove informazioni sull'ambiente circostante. Ad esempio, un agente può esplorare l'ambiente sconosciuto per ottenere una migliore comprensione e prendere decisioni più informate. Un agente può migliorare la propria capacità di compiere azioni razionali attraverso l'**apprendimento**. Questo consente all'agente di adattarsi a nuovi ambienti o situazioni in base alle esperienze passate, migliorando le proprie prestazioni.

2.3 Struttura di un Agente

La struttura di un agente può essere descritta come una combinazione di:

- **Architettura:** L'hardware o la piattaforma su cui l'agente opera (ad esempio, un computer con sensori e attuatori).
- **Programma agente:** Implementa la funzione agente e decide l'azione da eseguire in base alla percezione corrente.

L'obiettivo principale della progettazione di agenti intelligenti è creare programmi che producano comportamento razionale con il minor numero di regole o codice.

2.4 Tipi di Agenti

Esistono vari tipi di agenti intelligenti, ognuno con capacità diverse:

- **Agenti Reattivi Semplici:** Questi agenti intraprendono azioni basate esclusivamente sulla percezione corrente, ignorando la storia percettiva. Ad esempio, un agente può eseguire una semplice regola condizionale: "se rilevi sporco, allora pulisci".
- **Agenti Reattivi Basati su Modello:** Questi agenti hanno un modello interno dell'ambiente, che permette loro di tenere traccia dello stato del mondo e prevedere l'effetto delle proprie azioni.
- **Agenti Basati su Obiettivi:** A differenza degli agenti reattivi, questi agenti prendono decisioni basate su specifici obiettivi che desiderano raggiungere. Pianificano le azioni in base a come le loro scelte influenzeranno il raggiungimento degli obiettivi.
- **Agenti Basati sull'Utilità:** In aggiunta agli obiettivi, questi agenti utilizzano una funzione di utilità per misurare la "desiderabilità" di uno stato, permettendo loro di scegliere tra obiettivi contrastanti in modo ottimale.
- **Agenti che Apprendono:** Questi agenti migliorano continuamente le loro prestazioni grazie all'apprendimento, modificando il proprio comportamento in base al feedback ricevuto dalle loro azioni.

2.5 PEAS: Definizione degli Agenti

Un ambiente può essere descritto utilizzando la struttura **PEAS** (Performance, Environment, Actuators, Sensors):

- **P (Performance)**: La misura di prestazione adottata per valutare l'operato dell'agente.
- **E (Environment)**: L'ambiente in cui l'agente opera.
- **A (Actuators)**: Gli attuatori a disposizione dell'agente per eseguire azioni.
- **S (Sensors)**: I sensori utilizzati dall'agente per percepire l'ambiente.

2.6 Proprietà degli Ambienti

Gli ambienti in cui operano gli agenti possono essere caratterizzati da diverse proprietà:

- **Completamente osservabile vs Parzialmente osservabile**: Se l'agente ha accesso a tutte le informazioni rilevanti sull'ambiente o solo a una parte di esse.
- **Deterministico vs Stocastico**: Se le azioni dell'agente determinano completamente lo stato successivo dell'ambiente o se ci sono elementi di casualità.
- **Episodico vs Sequenziale**: Se le decisioni dell'agente sono prese indipendentemente per ciascun episodio o se ogni azione influenza gli stati futuri.
- **Statico vs Dinamico**: Se l'ambiente cambia mentre l'agente sta deliberando o se rimane invariato.
- **Discreto vs Continuo**: Se l'ambiente fornisce un numero limitato di percezioni e azioni o se le percezioni e azioni sono continue.
- **Singolo agente vs Multi-agente**: Se l'agente opera da solo o in presenza di altri agenti con cui può collaborare o competere.

Gli agenti intelligenti sono il cuore dell'Intelligenza Artificiale. Essi sono progettati per interagire con l'ambiente, prendere decisioni razionali e adattarsi a nuove situazioni grazie all'apprendimento. La varietà di agenti e di ambienti in cui operano rende l'IA una disciplina dinamica e ricca di sfide.

3 Formulazione di Problemi

Di seguito, alcune note ed osservazione rispetto la formulazione di problemi. Il materiale non è esaustivo, ma vuole essere un supporto allo studio individuale. Pertanto, la sola lettura di questo documento è sconsigliata.

3.1 Agenti Basati su Obiettivi

Un agente basato su obiettivi agisce nell'ambiente per raggiungere uno stato desiderato. Questi agenti non si limitano a rispondere a stimoli immediati, ma pianificano sequenze di azioni in base agli obiettivi da raggiungere.

- *Funzionamento.* Quando l'agente deve raggiungere un obiettivo complesso, valuta l'effetto delle proprie azioni sul mondo. La domanda che guida la sua decisione è: "Cosa accadrà se compio l'azione A? Sarò soddisfatto se lo faccio?".
- *Flessibilità:* Questo tipo di agente è flessibile perché la conoscenza può essere modificata e adattata. Gli agenti reattivi, al contrario, richiedono una riscrittura delle regole condizione-azione ogni volta che cambia il contesto.

3.2 Rappresentazione Atomica degli Stati

Nel contesto degli agenti risolutori di problemi, lo stato del mondo viene rappresentato in modo atomico. Ciò significa che ogni stato è considerato indivisibile e privo di struttura interna.

Esempio. Un agente che cerca di viaggiare da una città all'altra considera ogni città come uno stato, senza analizzare le caratteristiche interne di ciascuna città.

3.3 Formulazione di Problemi

La formulazione di un problema in AI è un passaggio cruciale per identificare le azioni che l'agente deve compiere per raggiungere un obiettivo. Il processo di formulazione comprende:

- *Stato Iniziale.* Punto di partenza dell'agente. Ad esempio, nel problema del viaggio in Romania discusso nel libro di testo e durante la lezione, lo stato iniziale potrebbe essere "in(Arad)".
- *Azioni.* Insieme di azioni applicabili in ogni stato. Ad esempio, "Go(Sibiu)", "Go(Timisoara)".
- *Modello di Transizione.* Definisce come ogni azione modifica lo stato. Ad esempio, eseguire "Go(Zerind)" nello stato "in(Arad)" porterà allo stato "In(Zerind)".

- *Test Obiettivo.* Verifica se uno stato raggiunge l'obiettivo. Ad esempio, lo stato finale può essere “In(Bucarest)”.
- *Costo di Cammino.* Funzione che calcola il costo per passare da uno stato all'altro.

3.4 Algoritmi di Ricerca

La risoluzione dei problemi tramite ricerca prevede che l'agente individui una sequenza di azioni che porti dallo stato iniziale allo stato obiettivo. Gli algoritmi di ricerca seguono tre fasi principali:

- *Formulazione del problema.* Il problema deve essere descritto formalmente, identificando le componenti che ne caratterizzeranno l'ambiente di esecuzione, ovvero quindi le componenti descritte in Sezione 3.3.
- *Ricerca di una soluzione.* Una volta formulato il problema, l'agente deve trovare una sequenza di azioni che porti dallo stato iniziale allo stato obiettivo. Esistono diversi approcci per la ricerca, tra cui, ad esempio:
 - *Ricerca non informata.* L'agente esplora lo spazio degli stati senza alcuna conoscenza a priori sull'obiettivo. Esempi sono la ricerca ad ampiezza e la ricerca in profondità.
 - *Ricerca informata.* L'agente utilizza delle euristiche per guidare la ricerca. Un esempio è l'algoritmo A*, che usa una funzione di valutazione per stimare il costo totale dal nodo attuale all'obiettivo.
- *Esecuzione della sequenza di azioni.* Una volta trovata la sequenza di azioni che conduce all'obiettivo, l'agente la esegue per raggiungere lo stato desiderato. Se la soluzione trovata è subottimale, l'agente potrebbe dover riformulare il problema o rieseguire la ricerca. Durante l'esecuzione delle azioni, l'agente ignora le sue percezioni perché ha già calcolato la sequenza completa in anticipo. Questo tipo di comportamento è detto a **ciclo aperto**, poiché l'agente non aggiorna il suo stato in base a nuove informazioni dall'ambiente. Questo approccio funziona solo in ambienti deterministici e completamente osservabili, dove ogni azione ha un unico risultato e l'agente può prevedere con esattezza gli effetti delle sue azioni. Tuttavia, se l'ambiente fosse stocastico o parzialmente osservabile, dove le azioni possono avere risultati incerti o le percezioni non riflettono completamente lo stato del mondo, l'agente avrebbe bisogno di un approccio a **ciclo chiuso**. In questo caso, l'agente aggiornerebbe costantemente la sua conoscenza dell'ambiente in base alle nuove percezioni e potrebbe ricalcolare le azioni necessarie per raggiungere l'obiettivo, adattando la sequenza di azioni durante l'esecuzione. Questo lo renderebbe più flessibile e capace di rispondere ai cambiamenti imprevisti nell'ambiente.

3.5 Un esempio: Formulazione del Problema delle 8 Regine

Il problema delle 8 regine richiede di piazzare 8 regine su una scacchiera in modo che nessuna possa attaccarne un'altra. Esistono due principali formulazioni di questo problema, ognuna con implicazioni diverse in termini di complessità computazionale.

Il primo modo di definire il problema è tramite una *formulazione incrementale*. In particolare, il problema sarebbe descritto come segue:

- **Descrizione.** Si aggiungono progressivamente le regine sulla scacchiera, cominciando dallo stato vuoto. Ogni stato corrisponde a una scacchiera con un numero crescente di regine.
- **Azioni:** Piazzare una regina in una casella vuota.
- **Complessità:** Questa formulazione esplora uno spazio di ricerca molto ampio, in quanto non impone vincoli immediati sul posizionamento delle regine. La ricerca dovrà esaminare circa 1.8×10^{14} possibili configurazioni per trovare una soluzione valida.
- **Svantaggi:** La mancanza di vincoli rende questa formulazione molto inefficiente per problemi con molti stati, come il problema delle 8 regine. Il numero di combinazioni da esaminare può crescere rapidamente, rendendo la ricerca complessa e lunga.

In alternativa, potremmo descrivere il problema tramite una *formulazione a stato completo*. Quindi:

- **Descrizione:** In questa formulazione, tutte le regine sono già piazzate sulla scacchiera, e il compito è spostarle in modo che nessuna minacci un'altra.
- **Azioni:** Spostare una regina all'interno della colonna in cui si trova, se minacciata.
- **Complessità:** Questa formulazione riduce drasticamente lo spazio di ricerca. Invece di esplorare tutte le possibili configurazioni, si esaminano solo configurazioni in cui ogni regina è già piazzata in una colonna separata. La complessità scende a circa 1.6×10^7 possibili configurazioni, rendendo la ricerca molto più gestibile.
- **Vantaggi:** Ridurre lo spazio di ricerca tramite vincoli (come il fatto che le regine si trovino in colonne diverse) rende questa formulazione più efficiente. Tuttavia, anche in questo caso, per problemi con N regine, la complessità può aumentare significativamente (ad esempio, con $N = 100$, lo spazio di ricerca raggiunge 10^{52} stati).

Altri esempi di problemi di ricerca sono:

- **Problema del Commesso Viaggiatore:** Ottimizzare un percorso che visiti una serie di città una sola volta minimizzando la distanza complessiva. Questo è un esempio di un problema di ottimizzazione NP-completo, in cui lo spazio di ricerca cresce esponenzialmente con il numero di città.
- **Problemi di Navigazione Robotica:** Estensione del problema di ricerca di percorsi a spazi continui e multidimensionali, come nel caso di robot dotati di braccia o gambe, che richiedono il controllo di movimenti complessi. Anche qui, la definizione corretta del problema e l'uso di astrazioni riduce la complessità della ricerca.

4 Algoritmi di Ricerca Non Informata

Nella lezione precedente abbiamo discusso la formulazione dei problemi di ricerca. In questa lezione, ci concentreremo sugli algoritmi di ricerca non informata, che non dispongono di informazioni aggiuntive sugli stati oltre a quella fornita dalla definizione del problema. Gli algoritmi di ricerca operano cercando una sequenza di azioni che porti dallo stato iniziale all'obiettivo. Durante la ricerca, viene costruito un albero di ricerca, con lo stato iniziale come radice e le azioni che espandono i rami.

4.1 Ricerca di Soluzioni

Una soluzione è una sequenza di azioni. Gli algoritmi di ricerca non informata seguono queste fasi:

- **Espansione:** Espandere uno stato significa applicare le azioni per ottenere nuovi stati.
- **Frontiera:** L'insieme di tutti i nodi foglia che possono essere espansi costituisce la frontiera.
- **Cammini ciclici:** Uno stato può essere ripetuto più volte, causando cicli, il che potrebbe portare a fallimenti in alcuni algoritmi.

Sulla base di questa definizione, un algoritmo di ricerca ad albero può essere rappresentato in termini di questo pseudo-codice:

```
function RICERCA-ALBERO(problema, strategia) returns una soluzione o fallimento
  inizializza la frontiera usando lo stato iniziale di problema
  loop do
    if la frontiera è vuota then return fallimento
    scegli un nodo foglia in base a strategia e rimuovilo dalla frontiera
    if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
    espandi il nodo scelto, aggiungendo i nodi risultanti alla frontiera
```

Un miglioramento dell'algoritmo di ricerca consiste nell'introduzione di un *insieme esplorato* per evitare la ripetizione di stati già visitati.

```
function RICERCA-GRAFO(problema, strategia) returns una soluzione o fallimento
  inizializza la frontiera usando lo stato iniziale di problema
  inizializza a vuoto l'insieme esplorato
  loop do
    if la frontiera è vuota then return fallimento
    scegli un nodo foglia e rimuovilo dalla frontiera
    if il nodo contiene uno stato obiettivo then return la soluzione corrispondente
    aggiungi il nodo all'insieme esplorato
    espandi il nodo scelto, aggiungendo i nodi risultati alla frontiera
    solo se non sono nella frontiera o nell'insieme esplorato
```

4.2 Strategie di Ricerca Non Informata

Le principali strategie di ricerca non informata includono:

- **Ricerca in ampiezza:** Espande prima i nodi di profondità minore.
- **Ricerca a costo uniforme:** Espande il nodo con il minor costo di cammino.
- **Ricerca in profondità:** Espande il nodo più profondo prima di tornare indietro.
- **Ricerca in profondità limitata:** Limita la profondità della ricerca per evitare cicli infiniti.
- **Ricerca ad approfondimento iterativo:** Combina la ricerca in ampiezza con la ricerca in profondità limitata, aumentando progressivamente il limite di profondità.
- **Ricerca bidirezionale:** Esegue una ricerca simultanea dallo stato iniziale e dall'obiettivo.

Di seguito vengono descritti i principali algoritmi di ricerca non informata trattati durante il corso. Gli appunti descrivono il funzionamento generale, lo pseudocodice, ma anche i principali vantaggi e svantaggi di ognuno di essi. Al termine, troverete anche un quiz tramite il quale potervi esercitare per la prova di esame.

4.2.1 Ricerca in Ampiezza (Breadth-First Search)

La ricerca in ampiezza esplora gli stati a partire dalla radice espandendo tutti i nodi a una determinata profondità prima di passare alla successiva. Viene utilizzata una coda FIFO (First-In, First-Out) per garantire che i nodi più vicini alla radice siano espansi per primi.

Pro:

- **Completo:** Trova sempre una soluzione se questa esiste.
- **Ottimo:** Trova la soluzione più breve in termini di numero di passaggi (quando il costo di cammino è uniforme).

Contro:

- **Complessità spaziale:** Richiede molta memoria per mantenere tutti i nodi della frontiera ($O(b^d)$, dove b è il fattore di ramificazione e d la profondità della soluzione).
- **Complessità temporale:** Il tempo necessario è esponenziale rispetto alla profondità della soluzione ($O(b^d)$).

Esempio di Utilizzo:

- **Utile:** Per problemi in cui la soluzione è vicina allo stato iniziale, come la risoluzione di puzzle con poche mosse richieste.
- **Non utile:** In problemi con un fattore di ramificazione elevato, come la ricerca di un percorso in una mappa complessa.

Pseudocodice:

```
function RICERCA-IN-AMPIEZZA(problema)
  nodo <- un nodo con stato = problema.STATO-INIZIALE
  if problema.TEST-OBIETTIVO(nodo.STATO) then return nodo
  frontiera <- una coda FIFO con nodo come unico elemento
  esplorati <- un insieme vuoto
  loop do
    if frontiera è vuota then return fallimento
    nodo <- POP(frontiera)
    aggiungi nodo.STATO a esplorati
    for each azione in problema.AZIONI(nodo.STATO) do
      figlio <- NODO-FIGLIO(problema, nodo, azione)
      if figlio.STATO non è in esplorati e non è in frontiera then
        if problema.TEST-OBIETTIVO(figlio.STATO) then return figlio
        frontiera <- INSERISCI(figlio, frontiera)
```

Spiegazione:

- Il nodo iniziale viene creato e inserito nella frontiera (una coda FIFO).
- Finché ci sono nodi nella frontiera, l'algoritmo rimuove il nodo in testa (POP).
- Se lo stato del nodo corrisponde all'obiettivo, l'algoritmo termina con successo.
- Altrimenti, il nodo viene espanso e i suoi successori (i figli) vengono aggiunti alla frontiera.
- Viene utilizzata una coda FIFO per garantire che i nodi più vicini alla radice siano espansi per primi.

4.2.2 Ricerca a Costo Uniforme (Uniform-Cost Search)

La ricerca a costo uniforme espande i nodi basandosi sul costo di cammino minimo dal nodo iniziale, piuttosto che sulla profondità. Viene utilizzata una coda a priorità, in cui i nodi con il costo di cammino più basso vengono espansi per primi.

Pro:

- **Completo:** Trova sempre una soluzione se esiste.
- **Ottimo:** Trova la soluzione con il costo di cammino minimo, indipendentemente dal numero di passi.

Contro:

- **Complessità temporale e spaziale:** Può esplorare molti nodi, soprattutto se le differenze tra i costi dei cammini sono piccole, rendendo la ricerca lenta ($O(b^{C^*/\epsilon})$, dove C^* è il costo della soluzione ottima).

Esempio di Utilizzo:

- **Utile:** Per problemi di ottimizzazione, come la pianificazione di percorsi con costi differenti per ogni passo, ad esempio nella navigazione stradale con pedaggi.
- **Non utile:** Quando tutti i cammini hanno costi simili o identici, dove la ricerca in ampiezza sarebbe più efficiente.

Pseudocodice:

```
function RICERCA-COSTO-UNIFORME(problema)
  nodo <- un nodo con stato = problema.STATO-INIZIALE
  frontiera <- una coda a priorità ordinata per COSTO-DI-CAMMINO
  esplorati <- un insieme vuoto
  loop do
    if frontiera è vuota then return fallimento
    nodo <- POP(frontiera)
    if problema.TEST-OBIETTIVO(nodo.STATO) then return nodo
    aggiungi nodo.STATO a esplorati
    for each azione in problema.AZIONI(nodo.STATO) do
      figlio <- NODO-FIGLIO(problema, nodo, azione)
      if figlio.STATO non è in esplorati e non è in frontiera then
        frontiera <- INSERISCI(figlio, frontiera)
      else if figlio.STATO è in frontiera con costo più alto then
        sostituisci quel nodo in frontiera con figlio
```

Spiegazione:

- La differenza principale rispetto alla ricerca in ampiezza è l'utilizzo di una coda a priorità per ordinare i nodi in base al loro costo di cammino.

- I nodi con il costo più basso vengono espansi per primi.
- Se si trova un nodo che raggiunge lo stato obiettivo con un costo inferiore a quelli precedenti, questo viene selezionato.

4.2.3 Ricerca in Profondità (Depth-First Search)

La ricerca in profondità esplora un cammino fino alla profondità massima, tornando indietro solo quando non ci sono più nodi da espandere lungo il cammino corrente. Utilizza una coda LIFO (Last-In, First-Out).

Pro:

- Complessità spaziale ridotta: Richiede meno memoria rispetto alla ricerca in ampiezza.
- Efficiente quando le soluzioni sono molto profonde.

Contro:

- Non completo: Può esplorare all'infinito in spazi di ricerca con profondità infinita o cicli.
- Non ottimo: Può trovare una soluzione non ottimale se c'è un cammino più corto.

Esempio di Utilizzo:

- **Utile:** Nei problemi in cui è probabile che la soluzione si trovi in profondità, come la risoluzione di puzzle profondi.
- **Non utile:** In spazi di ricerca con cicli o profondità infinita, come esplorazioni di labirinti con loop.

Pseudocodice:

```
function RICERCA-IN-PROFONDITÀ(problema)
  nodo <- un nodo con stato = problema.STATO-INIZIALE
  if problema.TEST-OBIETTIVO(nodo.STATO) then return nodo
  frontiera <- una coda LIFO con nodo come unico elemento
  loop do
    if frontiera è vuota then return fallimento
    nodo <- POP(frontiera)
    for each azione in problema.AZIONI(nodo.STATO) do
      figlio <- NODO-FIGLIO(problema, nodo, azione)
      if problema.TEST-OBIETTIVO(figlio.STATO) then return figlio
      frontiera <- INSERISCI(figlio, frontiera)
```

Spiegazione:

- La frontiera è implementata come una coda LIFO (Last-In, First-Out), il che significa che l'ultimo nodo inserito viene espanso per primo.
- L'algoritmo esplora un cammino fino alla profondità massima, e torna indietro quando non ci sono più successori da espandere lungo il cammino attuale (backtracking).
- Non tiene traccia degli stati visitati in precedenza, perciò può entrare in cicli infiniti in spazi di ricerca con cicli.

4.2.4 Ricerca in Profondità Limitata (Depth-Limited Search)

La ricerca in profondità limitata è una variante della ricerca in profondità, in cui viene imposto un limite alla profondità massima di esplorazione. Questo evita i cicli infiniti, ma introduce la possibilità di non trovare una soluzione se questa si trova oltre la profondità limite.

Pro:

- Evita i cicli infiniti imposti dalla ricerca in profondità pura.
- Migliora l'efficienza rispetto alla ricerca in profondità nei problemi con soluzioni vicine alla radice ma con ramificazioni profonde.

Contro:

- Non completo: Se la soluzione si trova oltre il limite di profondità fissato, l'algoritmo non la troverà.
- Non ottimo: Potrebbe trovare una soluzione non ottimale.

Esempio di Utilizzo:

- **Utile:** Nei casi in cui è possibile stimare la profondità massima della soluzione, come nei giochi a turni con un numero limitato di mosse.
- **Non utile:** In problemi dove la profondità della soluzione non è nota o è molto grande.

Pseudocodice:

```
function RICERCA-PROFONDITÀ-LIMITATA(problema, limite)
    return RPL-RICORSIVA(CREA-NODO(problema.STATO-INIZIALE), problema, limite)

function RPL-RICORSIVA(nodo, problema, limite)
    if problema.TEST-OBIETTIVO(nodo.STATO) then return nodo
```



```

else if limite = 0 then return taglio
else
  for each azione in problema.AZIONI(nodo.STATO) do
    figlio <- NODO-FIGLIO(problema, nodo, azione)
    risultato <- RPL-RICORSIVA(figlio, problema, limite-1)
    if risultato != fallimento then return risultato
  return fallimento

```

Spiegazione:

- L'algoritmo è simile alla ricerca in profondità standard, ma con l'aggiunta di un limite di profondità.
- Ogni volta che un nodo viene espanso, l'algoritmo verifica se ha raggiunto il limite. Se il limite è stato raggiunto, l'algoritmo interrompe l'espansione lungo quel cammino (restituisce "taglio").
- Se viene trovata una soluzione prima di raggiungere il limite, questa viene restituita immediatamente.

4.2.5 Ricerca ad Approfondimento Iterativo (Iterative Deepening Search)

La ricerca ad approfondimento iterativo combina i vantaggi della ricerca in ampiezza e della ricerca in profondità limitata. L'algoritmo esegue più iterazioni della ricerca in profondità limitata, aumentando progressivamente il limite di profondità a ogni iterazione. Questo garantisce che l'algoritmo esplori tutti i cammini con profondità crescente, trovando così la soluzione più vicina alla radice senza esaurire la memoria.

Pro:

- Completo e ottimo come la ricerca in ampiezza.
- Complessità spaziale simile alla ricerca in profondità, poiché richiede di memorizzare solo un cammino alla volta.

Contro:

- Tempo di esecuzione: Ripete più volte la ricerca a profondità crescente, il che comporta l'espansione ripetuta degli stessi nodi a profondità diverse.

Esempio di Utilizzo:

- **Utile:** Quando non si conosce la profondità della soluzione ma lo spazio di ricerca è vasto, come nella ricerca in alberi di gioco (es. scacchi).

- **Non utile:** In problemi dove il limite di profondità è noto e fisso, dove una ricerca in profondità limitata sarebbe più efficiente.

Pseudocodice:

```
function RICERCA-APPROFONDIMENTO-ITERATIVO(problema)
  for profondità = 0 to infinite do
    risultato <- RICERCA-PROFONDITÀ-LIMITATA(problema, profondità)
    if risultato != fallimento then return risultato
```

Spiegazione:

- L'algoritmo esegue più iterazioni di ricerca in profondità limitata, aumentando il limite di profondità a ogni iterazione.
- In ogni iterazione, l'algoritmo cerca soluzioni fino a un determinato limite di profondità. Se non trova una soluzione, aumenta il limite e ripete la ricerca.
- Questo approccio garantisce che venga trovata la soluzione più vicina alla radice, ma con l'efficienza spaziale della ricerca in profondità.

4.2.6 Ricerca Bidirezionale (Bidirectional Search)

La ricerca bidirezionale consiste nel cercare simultaneamente dallo stato iniziale e dallo stato obiettivo, espandendo i due alberi di ricerca finché le loro frontiere non si incontrano. Questo approccio può ridurre significativamente il numero di nodi da esplorare, poiché esplora solo metà dello spazio di ricerca da ciascuna direzione.

Pro:

- Molto efficiente: Riduce drasticamente il numero di nodi da esplorare, con complessità $O(b^{d/2})$ invece di $O(b^d)$, dove d è la profondità della soluzione.

Contro:

- Difficile da implementare: Richiede che il problema sia facilmente invertibile, cioè che sia possibile cercare "all'indietro" dallo stato obiettivo verso lo stato iniziale.

Esempio di Utilizzo:

- **Utile:** In problemi dove è facile definire uno stato obiettivo concreto e simmetrico, come nella ricerca di percorsi in una mappa.

- **Non utile:** In problemi dove l'obiettivo è difficile da definire o non simmetrico, come il problema delle N regine.

Pseudocodice:

```
function RICERCA-BIDIREZIONALE(problema)
  frontieraInizio <- una coda con lo stato iniziale
  frontieraObiettivo <- una coda con lo stato obiettivo
  esploratiInizio <- un insieme vuoto
  esploratiObiettivo <- un insieme vuoto
  loop do
    if frontieraInizio è vuota or frontieraObiettivo è vuota then return fallimento
    nodoInizio <- POP(frontieraInizio)
    nodoObiettivo <- POP(frontieraObiettivo)
    if nodoInizio è in esploratiObiettivo or nodoObiettivo è in esploratiInizio then
      return CAMMINO(nodoInizio, nodoObiettivo)
    aggiungi nodoInizio a esploratiInizio
    aggiungi nodoObiettivo a esploratiObiettivo
    espandi nodoInizio e aggiungi i suoi figli alla frontieraInizio
    espandi nodoObiettivo e aggiungi i suoi figli alla frontieraObiettivo
```

Spiegazione:

- La ricerca avviene simultaneamente dallo stato iniziale e dallo stato obiettivo, con due frontiere separate.
- Quando le due frontiere si incontrano, la soluzione viene trovata unendo i due percorsi.
- Richiede che il problema sia facilmente invertibile, cioè che sia possibile definire un cammino che parte dallo stato obiettivo e va verso lo stato iniziale.

4.3 Conclusioni

Gli algoritmi di ricerca non informata offrono diverse strategie per esplorare lo spazio degli stati. A seconda del problema, è possibile scegliere un algoritmo adatto in base a criteri di completezza, ottimalità, complessità temporale e spaziale.

4.4 Quiz Time

Domanda 1: Quale algoritmo di ricerca esplora prima tutti i nodi di una profondità prima di passare ai nodi della profondità successiva?

- A) Ricerca in profondità

- B) Ricerca a costo uniforme
- C) Ricerca bidirezionale
- D) Ricerca in ampiezza

Risposta corretta: D) Ricerca in ampiezza

Domanda 2: Qual è la principale differenza tra la ricerca a costo uniforme e la ricerca in ampiezza?

- A) La ricerca in ampiezza utilizza una coda LIFO, mentre la ricerca a costo uniforme utilizza una coda FIFO
- B) La ricerca a costo uniforme espande il nodo con il costo di cammino più basso, mentre la ricerca in ampiezza espande il nodo più vicino alla radice
- C) La ricerca in ampiezza è ottima, mentre la ricerca a costo uniforme non lo è
- D) La ricerca a costo uniforme è incompleta, mentre la ricerca in ampiezza è completa

Risposta corretta: B) La ricerca a costo uniforme espande il nodo con il costo di cammino più basso, mentre la ricerca in ampiezza espande il nodo più vicino alla radice

Domanda 3: Quale algoritmo di ricerca è particolarmente adatto quando non si conosce la profondità della soluzione, ma si vuole comunque garantire una ricerca completa ed efficiente in termini di memoria?

- A) Ricerca in profondità
- B) Ricerca a costo uniforme
- C) Ricerca ad approfondimento iterativo
- D) Ricerca bidirezionale

Risposta corretta: C) Ricerca ad approfondimento iterativo

Domanda 4: Quale tra i seguenti algoritmi di ricerca non garantisce di trovare una soluzione se questa si trova oltre un certo limite di profondità?

- A) Ricerca in ampiezza
- B) Ricerca a costo uniforme
- C) Ricerca in profondità limitata
- D) Ricerca bidirezionale

Risposta corretta: C) Ricerca in profondità limitata

Domanda 5: Quale strategia di ricerca è più efficiente in termini di memoria ma non garantisce l'ottimalità della soluzione trovata?

- A) Ricerca in ampiezza

- B) Ricerca in profondità
- C) Ricerca a costo uniforme
- D) Ricerca ad approfondimento iterativo

Risposta corretta: B) Ricerca in profondità

Domanda 6: Quale algoritmo espande simultaneamente dallo stato iniziale e dallo stato obiettivo, cercando di far incontrare le due ricerche?

- A) Ricerca in profondità
- B) Ricerca a costo uniforme
- C) Ricerca bidirezionale
- D) Ricerca ad approfondimento iterativo

Risposta corretta: C) Ricerca bidirezionale

Domanda 7: Quale tra i seguenti algoritmi è inefficiente dal punto di vista della memoria, soprattutto per problemi con grandi spazi di ricerca?

- A) Ricerca in profondità
- B) Ricerca in ampiezza
- C) Ricerca ad approfondimento iterativo
- D) Ricerca bidirezionale

Risposta corretta: B) Ricerca in ampiezza

Domanda 8: Qual è un vantaggio principale della ricerca ad approfondimento iterativo rispetto alla ricerca in profondità?

- A) Richiede meno memoria
- B) Trova sempre una soluzione ottima
- C) Evita di esplorare nodi ciclici
- D) Può essere applicata solo a spazi di ricerca limitati

Risposta corretta: B) Trova sempre una soluzione ottima

Domanda 9: In quale caso la ricerca in profondità può fallire nel trovare una soluzione?

- A) Quando la profondità della soluzione è troppo grande
- B) Quando il problema ha un numero finito di stati
- C) Quando viene usata una coda FIFO
- D) Quando il fattore di ramificazione è piccolo

Risposta corretta: A) Quando la profondità della soluzione è troppo grande

Domanda 10: Quale tra i seguenti algoritmi è in grado di trovare la soluzione ottimale in termini di costo di cammino?

- A) Ricerca in profondità
- B) Ricerca in ampiezza
- C) Ricerca a costo uniforme
- D) Ricerca ad approfondimento iterativo

Risposta corretta: C) Ricerca a costo uniforme

5 Algoritmi di Ricerca Informata

Gli algoritmi di ricerca informata utilizzano informazioni aggiuntive, come funzioni euristiche, per migliorare l'efficienza della ricerca rispetto agli algoritmi non informati. In questi algoritmi, il nodo da espandere viene scelto sulla base di una funzione di valutazione $f(n)$, che combina il costo accumulato e una stima del costo futuro.

5.1 Strategie di Ricerca Informata

Le principali strategie di ricerca informata includono:

- **Ricerca Greedy Best-First:** Espande il nodo che sembra essere il più vicino all'obiettivo, basandosi esclusivamente sulla funzione euristica $h(n)$, senza considerare il costo accumulato del cammino.
- **Algoritmo A*:** Combina il costo accumulato *begin : math : text* $g(n)$ *end : math : text* con una stima euristica *begin : math : text* $h(n)$ *end : math : text* del costo per raggiungere l'obiettivo, espandendo i nodi in base alla funzione *begin : math : text* $f(n) = g(n) + h(n)$ *end : math : text*. Questo garantisce sia completezza che ottimalità, a patto che l euristica sia ammissibile.
- **Beam Search:** Limita il numero di nodi esplorati mantenendo solo i *begin : math : text* k *end : math : text* nodi più promettenti a ogni livello, riducendo così il consumo di memoria ma sacrificando completezza e ottimalità.
- **Iterative Deepening A* (IDA*):** Combina l'approfondimento iterativo con A*, esplorando progressivamente lo spazio di ricerca con limiti crescenti sulla funzione *begin : math : text* $f(n)$ *end : math : text*. Questo approccio riduce il consumo di memoria rispetto ad A*, ma può riesplorare più volte gli stessi stati.
- **Ricerca Best-First Ricorsiva (RBFS):** Espande ricorsivamente i nodi più promettenti in termini di *begin : math : text* $f(n)$ *end : math : text*, utilizzando una quantità di memoria lineare rispetto alla profondità della soluzione. Mantiene in memoria solo il percorso attuale, ma può riesplorare cammini già visitati.

Di seguito vengono descritti i principali algoritmi di ricerca informata trattati durante il corso. Gli appunti descrivono il funzionamento generale, lo pseudocodice, ma anche i principali vantaggi e svantaggi di ognuno di essi. Al termine, troverete anche un quiz tramite il quale potervi esercitare per la prova di esame.

5.1.1 Ricerca Greedy Best-First

La ricerca Greedy Best-First espande il nodo che sembra essere il più vicino all'obiettivo, basandosi sulla funzione euristica $h(n)$. Questo algoritmo è “goloso” nel senso che cerca di avvicinarsi il più rapidamente possibile all'obiettivo, ma senza considerare il costo totale del cammino.

Pro:

- Veloce: Spesso trova rapidamente una soluzione, specialmente quando l'euristica è ben progettata.
- Semplice da implementare rispetto a molti altri algoritmi di ricerca informata.

Contro:

- Non ottimale: Non garantisce che la soluzione trovata sia la più economica in termini di costo complessivo del cammino.
- Non completo: L'algoritmo potrebbe non trovare una soluzione se entra in un ciclo o se si trova bloccato in un vicolo cieco.

Esempio di Utilizzo:

- **Utile:** Problemi in cui la velocità di trovare una soluzione è più importante della sua ottimalità, come la ricerca in grandi spazi di stati in cui non è necessario il cammino più breve.
- **Non utile:** In contesti dove l'ottimalità della soluzione è fondamentale, come nei problemi di pianificazione di percorsi con costi variabili.

Pseudocodice:

```
function RICERCA-GREEDY(problema)
nodo <- CREA-NODO(problema.STATO-INIZIALE)
frontiera <- una coda ordinata per h(n), con nodo come unico elemento
esplorati <- insieme vuoto
loop do
if frontiera è vuota then return fallimento
nodo <- POP(frontiera)
if problema.TEST-OBIETTIVO(nodo.STATO) then return SOLUZIONE(nodo)
aggiungi nodo.STATO a esplorati
for each azione in problema.AZIONI(nodo.STATO) do
figlio <- NODO-FIGLIO(problema, nodo, azione)
if figlio.STATO non è in esplorati e non è in frontiera then
INSERISCI(figlio, frontiera)
```

Spiegazione:

- Il nodo iniziale viene inserito nella frontiera, una coda ordinata per il valore dell'euristica $h(n)$.
- L'algoritmo espande sempre il nodo con il valore di $h(n)$ più basso, supponendo che questo lo avvicini all'obiettivo.

- Se lo stato del nodo espanso soddisfa il test obiettivo, la soluzione viene restituita.
- Altrimenti, i successori del nodo vengono generati e aggiunti alla frontiera, sempre ordinata per $h(n)$.

5.1.2 Algoritmo A*

L'algoritmo A* combina il costo accumulato $g(n)$ con una stima del costo per raggiungere l'obiettivo $h(n)$, valutando i nodi con la funzione $f(n) = g(n) + h(n)$. Questo garantisce che vengano esplorati prima i nodi che sembrano portare alla soluzione più economica.

Pro:

- Completo: Trova sempre una soluzione se una soluzione esiste.
- Ottimale: Se l'euristica è ammissibile, A* trova la soluzione con il costo minimo.

Contro:

- Complessità spaziale: Richiede molta memoria per mantenere tutti i nodi esplorati e quelli nella frontiera.
- Complessità temporale: Può diventare inefficiente in spazi di ricerca molto grandi.

Esempio di Utilizzo:

- **Utile:** Per la pianificazione di percorsi in ambienti con costi variabili, come la navigazione stradale o la ricerca di percorsi ottimali in grafi.
- **Non utile:** In contesti dove lo spazio di ricerca è troppo grande per essere gestito in memoria.

Pseudocodice:

```
function RICERCA-A*(problema)
nodo <- CREA-NODO(problema.STATO-INIZIALE)
frontiera <- coda a priorità ordinata per  $f(n) = g(n) + h(n)$ 
esplorati <- insieme vuoto
loop do
if frontiera è vuota then return fallimento
nodo <- POP(frontiera)
if problema.TEST-OBIETTIVO(nodo.STATO) then return SOLUZIONE(nodo)
aggiungi nodo.STATO a esplorati
for each azione in problema.AZIONI(nodo.STATO) do
figlio <- NODO-FIGLIO(problema, nodo, azione)
if figlio.STATO non è in esplorati e non è in frontiera then
INSERISCI(figlio, frontiera)
```

Spiegazione:

- Il nodo iniziale viene inserito nella frontiera, una coda a priorità ordinata per $f(n) = g(n) + h(n)$.
- L'algoritmo espande il nodo con il valore più basso di $f(n)$, garantendo che si esplori prima il cammino più promettente.
- Se viene raggiunto un nodo che soddisfa il test obiettivo, la soluzione viene restituita.
- L'algoritmo garantisce l'ottimalità solo se l'euristica $h(n)$ è ammissibile (non sovrastima mai il costo rimanente).

5.1.3 Beam Search

La Beam Search è una variante della ricerca Best-First in cui viene limitato il numero di nodi espansi a ogni livello, mantenendo solo i k nodi più promettenti. Questo approccio riduce l'uso della memoria rispetto ad algoritmi come A^* .

Pro:

- Minor uso della memoria: Rispetto ad A^* , Beam Search mantiene in memoria solo un numero limitato di nodi a ogni livello.

Contro:

- Non ottimale: Beam Search può ignorare cammini che porterebbero a soluzioni più ottimali.
- Non completo: Potrebbe non trovare una soluzione, soprattutto se il numero di nodi mantenuti è troppo limitato.

Esempio di Utilizzo:

- **Utile:** Problemi in cui la memoria è limitata e la velocità di ricerca è più importante dell'ottimalità della soluzione.
- **Non utile:** In problemi dove è necessario garantire l'ottimalità o la completezza, come nella pianificazione di percorsi ottimali.

Pseudocodice:

```
function BEAM-SEARCH(problema, k)
nodo <- CREA-NODO(problema.STATO-INIZIALE)
frontiera <- una coda con massimo k elementi, ordinati per h(n)
esplorati <- insieme vuoto
loop do
if frontiera è vuota then return fallimento
nodo <- POP(frontiera)
if problema.TEST-OBIETTIVO(nodo.STATO) then return SOLUZIONE(nodo)
aggiungi nodo.STATO a esplorati
for each azione in problema.AZIONI(nodo.STATO) do
figlio <- NODO-FIGLIO(problema, nodo, azione)
INSERISCI(figlio, frontiera)
if dimensione(frontiera) > k then RIMUOVI(fronte)
```

Spiegazione:

- Beam Search mantiene un massimo di k nodi a ogni livello dell'albero di ricerca.
- L'algoritmo ordina i nodi in base all'euristica $h(n)$, conservando solo i k nodi più promettenti.
- Sebbene utilizzi meno memoria rispetto ad A*, potrebbe scartare cammini che portano a soluzioni ottimali, rendendolo non ottimale e non completo.

5.1.4 Iterative Deepening A* (IDA*)

L'algoritmo Iterative Deepening A* (IDA*) è una variante dell'algoritmo A* che utilizza l'approfondimento iterativo. L'IDA* applica la stessa funzione di valutazione $f(n) = g(n) + h(n)$, ma esplora progressivamente lo spazio di ricerca con limiti crescenti di $f(n)$, similmente a come l'approfondimento iterativo espande progressivamente la profondità.

Pro:

- Ottimale: Trova la soluzione più economica se l'euristica $h(n)$ è ammissibile.
- Efficienza spaziale: Utilizza meno memoria rispetto ad A* poiché non mantiene tutti i nodi nella frontiera.

Contro:

- Complessità temporale: Può dover riesplorare più volte gli stessi nodi, soprattutto in spazi di ricerca con molti stati.

- Richiede una buona euristica: L'efficacia di IDA* dipende fortemente dalla qualità dell'euristica utilizzata.

Esempio di Utilizzo:

- **Utile:** In problemi di grande scala con limiti di memoria, come il gioco degli scacchi o il cubo di Rubik.
- **Non utile:** Quando il calcolo di una buona euristica è difficile o quando si dispone di memoria sufficiente per eseguire A*.

Pseudocodice:

```
function IDA*(problema)
limite <- f(problema.STATO-INIZIALE)
loop do
risultato, nuovo_limite <- DFS-LIMITATO(problema, limite)
if risultato != fallimento then return risultato
if nuovo_limite == infinito then return fallimento
limite <- nuovo_limite
```

Spiegazione:

- L'algoritmo inizia impostando un limite di $f(n)$ e utilizza una ricerca in profondità limitata basata su questo valore.
- Ogni volta che la ricerca fallisce, il limite di $f(n)$ viene incrementato per esplorare cammini più profondi o costosi.
- Questo processo continua finché non viene trovata una soluzione o finché non vengono esaurite le possibilità di esplorazione.

5.1.5 Ricerca Best-First Ricorsiva (RBFS)

La ricerca Best-First Ricorsiva (Recursive Best-First Search, RBFS) è un algoritmo che utilizza una strategia simile ad A*, ma impiega uno spazio di memoria lineare rispetto alla profondità della soluzione. RBFS espande il nodo migliore, ma invece di mantenere tutti i nodi nella memoria, gestisce un solo percorso alla volta, memorizzando informazioni sui nodi scartati per un eventuale ripristino.

Pro:

- Ottimale: Trova la soluzione ottimale se l'euristica $h(n)$ è ammissibile.
- Utilizza meno memoria rispetto ad A^* , con una complessità spaziale lineare.

Contro:

- Complessità temporale: Poiché può dover riesplorare più volte i nodi scartati, può risultare inefficiente in alcuni casi.
- Meno pratico rispetto ad A^* : La ricorsione può comportare una complessità temporale elevata in problemi molto profondi.

Esempio di Utilizzo:

- **Utile:** In problemi dove lo spazio di memoria è limitato e si desidera un algoritmo ottimale, come la risoluzione di puzzle complessi.
- **Non utile:** In problemi con molti cicli o dove la profondità della soluzione è molto alta.

Pseudocodice:

```
function RICERCA-BEST-FIRST-RICORSIVA(problema)
return RBFS(problema, CREA-NODO(problema.STATO-INIZIALE), infinite)
function RBFS(problema, nodo, f_limite)
if problema.TEST-OBIETTIVO(nodo.STATO) then return SOLUZIONE(nodo)
successori <- []
for each azione in problema.AZIONI(nodo.STATO) do
figlio <- NODO-FIGLIO(problema, nodo, azione)
successori <- AGGIUNGI(figlio, successori)
if successori è vuoto then return fallimento,
for each figlio in successori do
figlio.f <- max(figlio.g + figlio.h, nodo.f)
loop do
migliore <- NODO-MINIMO(successori)
if migliore.f > f_limite then return fallimento, migliore.f
alternativa <- SECONDO-NODO-MINIMO(successori)
risultato, migliore.f <- RBFS(problema, migliore, min(f_limite, alternativa.f))
if risultato fallimento then return risultato
```

Spiegazione:

- RBFS esplora il nodo con il valore $f(n)$ più basso, simile ad A^* , ma con un uso limitato della memoria.
- Se l'algoritmo incontra un nodo con un valore di $f(n)$ troppo alto, torna indietro e tenta di esplorare un cammino alternativo.
- L'algoritmo mantiene solo i nodi necessari in memoria, riducendo la complessità spaziale rispetto ad A^* , ma può dover esplorare più volte gli stessi cammini.

5.2 Conclusioni

Gli algoritmi di ricerca informata offrono strategie avanzate per esplorare lo spazio degli stati, utilizzando funzioni euristiche per guidare la ricerca verso la soluzione in modo più efficiente. A seconda del problema, è possibile scegliere l'algoritmo più adatto in base a diversi criteri come:

- **Completezza:** La capacità dell'algoritmo di trovare una soluzione se una soluzione esiste. Algoritmi come A^* e IDA^* garantiscono completezza, a condizione che l'euristica sia ben definita.
- **Ottimalità:** La capacità dell'algoritmo di trovare la soluzione più economica. A^* è ottimale quando l'euristica è ammissibile, mentre algoritmi come Greedy Best-First e Beam Search non garantiscono l'ottimalità.
- **Complessità temporale:** Il tempo necessario per trovare una soluzione. L'uso di buone euristiche può ridurre il tempo di ricerca, come accade per A^* e Greedy Best-First, ma algoritmi come IDA^* possono riesplorare nodi più volte, aumentando la complessità temporale.
- **Complessità spaziale:** La quantità di memoria utilizzata dall'algoritmo. Algoritmi come A^* possono richiedere molta memoria per memorizzare tutti i nodi esplorati, mentre IDA^* e RBFS riducono significativamente l'uso della memoria sacrificando in parte l'efficienza temporale.

Questa analisi aiuta a selezionare l'algoritmo di ricerca informata più appropriato in funzione del problema specifico, tenendo conto delle risorse computazionali disponibili e degli obiettivi di performance.

5.3 Quiz Time

Domanda 1: Quale algoritmo di ricerca informata utilizza la funzione di valutazione $f(n) = g(n) + h(n)$?

- A) Ricerca Greedy Best-First
- B) Algoritmo A^*
- C) Beam Search
- D) IDA^*

Risposta corretta: B) Algoritmo A*

Domanda 2: Qual è la differenza principale tra l'algoritmo A* e la ricerca Greedy Best-First?

- A) A* considera sia il costo del cammino accumulato sia la stima euristica, mentre Greedy Best-First usa solo la stima euristica.
- B) Greedy Best-First utilizza più memoria rispetto ad A*.
- C) A* è incompleto, mentre Greedy Best-First è completo.
- D) A* non utilizza una funzione euristica.

Risposta corretta: A) A* considera sia il costo del cammino accumulato sia la stima euristica, mentre Greedy Best-First usa solo la stima euristica.

Domanda 3: In quale algoritmo di ricerca si espandono solo i k nodi più promettenti a ogni livello?

- A) IDA*
- B) Beam Search
- C) A*
- D) RBFS

Risposta corretta: B) Beam Search

Domanda 4: Quale dei seguenti algoritmi utilizza un approccio ricorsivo per limitare l'uso della memoria, ma può riesplorare nodi già visitati?

- A) A*
- B) Beam Search
- C) RBFS
- D) IDA*

Risposta corretta: C) RBFS

Domanda 5: In quale contesto l'algoritmo IDA* è più vantaggioso rispetto all'A*?

- A) Quando lo spazio di ricerca è piccolo e la memoria non è un problema.
- B) Quando si ha una grande quantità di memoria disponibile.
- C) Quando lo spazio di ricerca è ampio e la memoria è limitata.

D) Quando si cerca una soluzione rapida senza preoccuparsi dell'ottimalità.

Risposta corretta: C) Quando lo spazio di ricerca è ampio e la memoria è limitata.

Domanda 6: Quale algoritmo di ricerca informata è garantito essere ottimale, a condizione che l'euristica sia ammissibile?

- A) A*
- B) Ricerca Greedy Best-First
- C) Beam Search
- D) RBFS

Risposta corretta: A) A*

Domanda 7: In quale algoritmo l'euristica $h(n)$ è utilizzata per determinare la selezione del nodo più promettente, senza considerare il costo accumulato $g(n)$?

- A) A*
- B) Ricerca Greedy Best-First
- C) IDA*
- D) Beam Search

Risposta corretta: B) Ricerca Greedy Best-First

Domanda 8: Quale delle seguenti caratteristiche descrive la ricerca Beam Search?

- A) Completo e ottimale
- B) Mantiene in memoria solo un numero limitato di nodi a ogni livello
- C) Richiede una grande quantità di memoria per mantenere tutti i nodi espansi
- D) Espande tutti i nodi di un livello prima di passare al successivo

Risposta corretta: B) Mantiene in memoria solo un numero limitato di nodi a ogni livello

Domanda 9: Quale algoritmo esplora progressivamente lo spazio di ricerca con limiti crescenti di $f(n)$?

- A) A*
- B) IDA*

- C) Beam Search
- D) RBFS

Risposta corretta: B) IDA*

Domanda 10: Quale algoritmo di ricerca informata è meno adatto per problemi con spazio di ricerca molto ampio e limiti di memoria?

- A) Ricerca Greedy Best-First
- B) A*
- C) Beam Search
- D) IDA*

Risposta corretta: B) A*

6 Algoritmi di Ricerca Locale

Gli algoritmi di ricerca locale sono utilizzati per risolvere problemi in cui il percorso verso la soluzione non è rilevante, ma ciò che conta è la configurazione finale. Questo approccio si differenzia dagli algoritmi di ricerca tradizionali, i quali tengono traccia dei cammini esplorati. Gli algoritmi di ricerca locale memorizzano solo lo stato corrente e tentano di migliorarlo iterativamente, rendendoli particolarmente utili per problemi di ottimizzazione. Data la loro natura, gli algoritmi di ricerca locale usando poca memoria (complessità spaziale spesso costante).

Gli algoritmi di ricerca locale si basano su una **funzione obiettivo** che misura quanto uno stato è vicino a una soluzione ottimale. Non esiste un vero e proprio “test obiettivo” ma un processo iterativo che cerca di migliorare lo stato corrente. Argomentiamo questo aspetto. Il test obiettivo è comune negli algoritmi di ricerca tradizionali, dove l’obiettivo è trovare uno stato che soddisfi esattamente un certo criterio (ad esempio, risolvere un puzzle o raggiungere un obiettivo specifico in un grafo). Negli algoritmi di ricerca locale, tuttavia, **ogni stato è già una soluzione in sé**. Il compito dell’algoritmo non è quindi trovare una soluzione qualsiasi, ma ottimizzare il risultato migliorando iterativamente lo stato corrente. Questa differenza implica che non si cerca di “raggiungere” uno stato obiettivo come negli algoritmi di ricerca tradizionali, ma di **migliorare progressivamente** una soluzione già esistente, guidati dalla funzione obiettivo. Inoltre, tutti gli stati considerati potrebbero essere soluzioni parzialmente valide, ma l’algoritmo cerca la soluzione “migliore” in termini di ottimizzazione, che spesso coincide con il **massimo globale** della funzione obiettivo. Per questa ragione, la funzione obiettivo gioca un ruolo cruciale nel guidare la ricerca verso soluzioni migliori.

Un esempio classico che illustra l’assenza di un test obiettivo negli algoritmi di ricerca locale è il problema delle N regine. In questo problema, lo stato iniziale può essere una configurazione completa di N regine su una scacchiera, ma in cui alcune regine si attaccano a vicenda. Ogni configurazione è già una soluzione “valida” in quanto contiene N regine piazzate, ma l’obiettivo è minimizzare o eliminare gli attacchi. In questo contesto, non esiste un test obiettivo standard, come verificare se si è raggiunto un determinato stato. Invece, si utilizza una funzione obiettivo che misura il numero di attacchi tra le regine. L’algoritmo di ricerca locale cerca quindi di ridurre questo numero iterativamente, migliorando la configurazione corrente. La ricerca non termina quando viene trovato uno stato specifico, ma quando si raggiunge una configurazione in cui il numero di attacchi non può più essere ridotto (ad esempio, quando il numero di attacchi è zero o non esiste una configurazione migliore).

Andando più nel dettaglio, un algoritmo di ricerca locale cercherà di ottimizzare la soluzione corrente muovendosi verso stati “vicini”, ovvero stati raggiungibili applicando una singola modifica alla soluzione corrente. Tali stati vicini formano la cosiddetta struttura dei vicini: per ciascuna soluzione s , viene definito un insieme di soluzioni vicine $N(s)$, che rappresenta i possibili stati raggiungibili con una singola mossa.

Per comprendere meglio il comportamento degli algoritmi di ricerca locale, possiamo immaginare lo **spazio degli stati** come un panorama, dove l’asse x rappresenta gli stati raggiungibili; e l’asse y rappresenta il valore della funzione obiettivo per ciascuno stato.

Alcune caratteristiche dello spazio degli stati:

- **Massimo Locale:** Un picco che rappresenta una soluzione sub-ottimale.
- **Piatto (plateau):** Una regione dello spazio con stati vicini che hanno tutti lo stesso valore.
- **Spalla:** Un plateau che presenta una leggera salita, offrendo la possibilità di miglioramento.
- **Massimo Globale:** La soluzione ottimale del problema.

6.1 Algoritmo Hill-Climbing

L'**Hill-Climbing** è uno degli algoritmi di ricerca locale più semplici. L'idea alla base è quella di scalare il "panorama" dello spazio degli stati fino a raggiungere un massimo o minimo globale.

Pseudocodice:

```
function HILL-CLIMBING(problema) returns uno stato massimo
  nodo_corrente <| CREA-NODO(problema.STATO-INIZIALE)
  loop do
    vicino <| il successore di nodo_corrente di valore più alto
    if vicino.VALORE <= nodo_corrente.VALORE then return nodo_corrente.STATO
  nodo_corrente <| vicino
```

Caratteristiche:

- **Pro:** È rapido, specialmente se la funzione obiettivo è ben definita.
- **Contro:** Può bloccarsi in un massimo locale o in un plateau.

Esempio: Problema delle 8 Regine Nella versione del problema delle 8 regine in cui ogni stato rappresenta una configurazione completa delle regine, l'Hill-Climbing può essere utilizzato per minimizzare il numero di attacchi tra le regine. Tuttavia, l'algoritmo potrebbe bloccarsi in una configurazione sub-ottimale con un numero minimo di attacchi, ma non zero.

6.2 Svantaggi dell'Hill-Climbing

L'algoritmo Hill-Climbing presenta alcuni problemi:

- **Massimi locali:** L'algoritmo potrebbe raggiungere un massimo locale e non essere in grado di proseguire oltre.
- **Plateau:** Una regione piatta può causare il blocco dell'algoritmo, che non riuscirà a migliorare la soluzione corrente.

- **Ridge (creste):** Porzioni dello spazio con una brusca variazione possono impedire all'algoritmo di trovare un percorso verso la soluzione ottimale.

6.3 Miglioramenti dell'Hill-Climbing

- **Hill-Climbing con mosse laterali:** L'algoritmo può fare mosse laterali su un plateau, ovvero accettare mosse che non migliorano immediatamente la funzione obiettivo, nella speranza di trovare una soluzione migliore in seguito.
- **Hill-Climbing stocastico:** Sceglie casualmente tra le mosse migliori, evitando di selezionare sempre la mossa più ovvia e aumentando così la diversificazione.
- **Riavvio casuale:** L'algoritmo può essere eseguito più volte con stati iniziali generati casualmente, aumentando le probabilità di trovare una soluzione globale.

6.4 Simulated Annealing

Il **Simulated Annealing** è una variante dell'Hill-Climbing che permette all'algoritmo di accettare soluzioni peggiori con una probabilità decrescente nel tempo, simulando un processo di raffreddamento graduale.

Pseudocodice:

```
function SIMULATED-ANNEALING(problema, velocità_raffreddamento)
  nodo_corrente <| CREA-NODO(problema.STATO-INIZIALE)
  for t <| 1 to
    T <| velocità_raffreddamento[t]
    if T = 0 then return nodo_corrente
    successivo <| un successore scelto a caso di nodo_corrente
    E <| successivo.VALORE - nodo_corrente.VALORE
    if E > 0 then nodo_corrente <| successivo
    else nodo_corrente <| successivo con probabilità  $e^{-E/T}$ 
```

Caratteristiche:

- **Pro:** Può evitare massimi locali grazie all'accettazione di soluzioni peggiori.
- **Contro:** L'efficacia dipende fortemente dalla funzione di raffreddamento e dalla temperatura iniziale.

6.5 Ricerca Local Beam

La **ricerca Local Beam** tiene traccia di k stati e ad ogni iterazione seleziona i successori migliori da un insieme di successori generati dai k stati. Questo approccio aumenta le probabilità di trovare una soluzione ottimale, evitando di concentrarsi su una singola regione dello spazio degli stati.

Vantaggi:

- Aumenta la diversità della ricerca.
- Consente di esplorare più regioni contemporaneamente.

6.6 Conclusioni

Gli algoritmi di ricerca locale sono strumenti potenti per risolvere problemi di ottimizzazione in cui la configurazione finale è ciò che conta. Sebbene presentino alcuni svantaggi, come il rischio di bloccarsi in massimi locali, varianti come il Simulated Annealing e la ricerca Local Beam possono aumentare le probabilità di trovare soluzioni ottimali.

6.7 Quiz Time

Domanda 1: Qual è la principale differenza tra gli algoritmi di ricerca locale e quelli tradizionali?

- A) Gli algoritmi di ricerca locale tengono traccia dell'intero percorso.
- B) Gli algoritmi di ricerca locale considerano solo lo stato corrente e cercano di migliorarlo iterativamente.
- C) Gli algoritmi tradizionali usano meno memoria degli algoritmi di ricerca locale.
- D) Gli algoritmi di ricerca locale memorizzano tutti i possibili stati esplorati.

Risposta corretta: B) Gli algoritmi di ricerca locale considerano solo lo stato corrente e cercano di migliorarlo iterativamente.

Domanda 2: Quale tra le seguenti è una caratteristica dello spazio degli stati negli algoritmi di ricerca locale?

- A) Tutti gli stati sono massimi globali.
- B) L'obiettivo è evitare i massimi locali.
- C) Gli stati non hanno valore obiettivo.

D) La funzione obiettivo non è rilevante.

Risposta corretta: B) L'obiettivo è evitare i massimi locali.

Domanda 3: Quale tra i seguenti è uno svantaggio dell'algoritmo Hill-Climbing?

- A) Può bloccarsi in massimi locali.
- B) Richiede molta memoria.
- C) Non è adatto ai problemi di ottimizzazione.
- D) È più lento di altri algoritmi.

Risposta corretta: A) Può bloccarsi in massimi locali.

Domanda 4: Quale tecnica permette all'algoritmo Hill-Climbing di superare un plateau?

- A) Riavvio casuale.
- B) Hill-Climbing con mosse laterali.
- C) Aumentare la dimensione del problema.
- D) Fermarsi e riprendere con un altro algoritmo.

Risposta corretta: B) Hill-Climbing con mosse laterali.

Domanda 5: Quale è l'obiettivo del Simulated Annealing?

- A) Evitare i massimi globali.
- B) Evitare massimi locali accettando soluzioni peggiori con una certa probabilità.
- C) Aumentare il tempo di esplorazione.
- D) Usare meno memoria rispetto a Hill-Climbing.

Risposta corretta: B) Evitare massimi locali accettando soluzioni peggiori con una certa probabilità.

Domanda 6: Qual è il principio alla base del Simulated Annealing?

- A) Scegliere casualmente il miglior successore.
- B) Raffreddamento graduale per ridurre la probabilità di accettare soluzioni peggiori nel tempo.
- C) Aumentare progressivamente la funzione obiettivo.

D) Esplorare più stati simultaneamente.

Risposta corretta: B) Raffreddamento graduale per ridurre la probabilità di accettare soluzioni peggiori nel tempo.

Domanda 7: Nella ricerca Local Beam, qual è il numero di stati tracciati durante la ricerca?

- A) Un solo stato alla volta.
- B) Un numero casuale di stati.
- C) Un numero predefinito di stati k .
- D) Tutti gli stati possibili.

Risposta corretta: C) Un numero predefinito di stati k .

Domanda 8: Perché non esiste un test obiettivo negli algoritmi di ricerca locale?

- A) Perché non è possibile definire un obiettivo finale.
- B) Perché tutti gli stati sono potenzialmente soluzioni valide e l'obiettivo è migliorare progressivamente lo stato corrente.
- C) Perché la ricerca locale non prevede soluzioni ottimali.
- D) Perché lo spazio degli stati è infinito.

Risposta corretta: B) Perché tutti gli stati sono potenzialmente soluzioni valide e l'obiettivo è migliorare progressivamente lo stato corrente.

Domanda 9: Qual è uno svantaggio del riavvio casuale negli algoritmi di ricerca locale?

- A) Richiede più tempo per convergere a una soluzione ottimale.
- B) Richiede molta memoria per memorizzare i riavvii.
- C) Non è efficace in spazi di ricerca piccoli.
- D) Non supera i massimi globali.

Risposta corretta: A) Richiede più tempo per convergere a una soluzione ottimale.

Domanda 10: Qual è lo scopo dell'Hill-Climbing stocastico?

- A) Scegliere sempre il miglior successore.
- B) Scegliere casualmente tra i successori migliori, introducendo diversificazione.

C) Evitare soluzioni peggiori.

D) Aumentare il numero di successori considerati.

Risposta corretta: B) Scegliere casualmente tra i successori migliori, introducendo diversificazione.

7 Algoritmi Genetici

Gli algoritmi genetici (GA) sono algoritmi di ricerca e ottimizzazione ispirati ai meccanismi evolutivi osservati in natura, come la selezione naturale, l'incrocio e la mutazione. Questi algoritmi sono stati sviluppati per risolvere problemi complessi dove i metodi tradizionali possono risultare inefficaci, specialmente in spazi di ricerca molto grandi o altamente non lineari.

L'idea alla base degli algoritmi genetici è di trattare ogni possibile soluzione a un problema come un individuo di una popolazione. A ogni iterazione (chiamata *generazione*), vengono selezionati gli individui migliori per la riproduzione, creando così una nuova generazione di soluzioni potenzialmente migliori. L'obiettivo è migliorare progressivamente la qualità delle soluzioni, fino a trovare la migliore possibile o una soluzione accettabile in un tempo ragionevole.

Gli algoritmi genetici sono ispirati alla teoria evolutiva di Darwin. In particolare:

1. Gli individui più **adatti** (quelli che si comportano meglio in un certo ambiente) hanno maggiori probabilità di sopravvivere e riprodursi.
2. Il processo di riproduzione mescola i **geni** dei genitori, creando una nuova generazione con variazioni genetiche che possono portare a individui ancora più adatti.
3. Le **mutazioni** occasionali, che introducono cambiamenti casuali nel patrimonio genetico, possono aiutare a esplorare nuove parti dello spazio di soluzioni, evitando di rimanere bloccati in soluzioni sub-ottimali.

Prima di proseguire, è importante rimarcare che i GA sono una **meta-euristica**, ossia un insieme di strategie generali per risolvere problemi di ricerca e ottimizzazione. Una meta-euristica è un approccio di alto livello per risolvere problemi di ottimizzazione complessi, che non richiede una conoscenza approfondita della struttura del problema specifico. Le meta-euristiche, a differenza delle tecniche euristiche tradizionali che sono solitamente progettate per problemi specifici, forniscono un quadro generale applicabile a una vasta gamma di problemi. Le meta-euristiche sono progettate per esplorare ampi spazi di ricerca che possono essere troppo vasti o complessi per metodi di ricerca esaustiva o deterministici. Introducono casualità nel processo di ricerca, il che aiuta a evitare il rischio di rimanere bloccati in soluzioni sub-ottimali o massimi locali. Inoltre, possono essere adattate a una vasta gamma di problemi di ottimizzazione, inclusi quelli per cui non è disponibile un modello matematico esatto. Gli algoritmi genetici si inseriscono perfettamente nel contesto delle meta-euristiche in quanto:

- **Esplorano lo spazio delle soluzioni** in modo ampio e non limitato da approcci deterministici, grazie alla combinazione di selezione, crossover e mutazione.
- **Gestiscono bene le non linearità**: Il processo di crossover e mutazione permette ai GA di affrontare problemi complessi e non lineari, esplorando diverse combinazioni di soluzioni.
- **Non richiedono conoscenza dettagliata del problema**: I GA operano in modo efficiente anche in situazioni dove la forma esatta della funzione obiettivo è sconosciuta, il che li rende una scelta ideale in contesti di ottimizzazione dove le informazioni disponibili sono incomplete o limitate.

Gli algoritmi genetici vengono utilizzati per risolvere problemi di ottimizzazione, nei quali l'obiettivo è trovare la miglior configurazione possibile tra tutte le soluzioni possibili. Questi algoritmi si basano su una **funzione obiettivo**, che assegna un punteggio (o “fitness”) a ciascuna soluzione in base a quanto è buona rispetto ai criteri definiti dal problema.

Uno dei principali vantaggi degli algoritmi genetici è la loro capacità di cercare soluzioni **globali**, evitando di rimanere bloccati in massimi locali, come può accadere con altri algoritmi di ricerca locale. L'introduzione di mutazioni casuali e la selezione basata sulla fitness aiutano a esplorare ampiamente lo spazio di soluzioni, migliorando la probabilità di trovare la soluzione ottimale globale.

Un algoritmo genetico standard può essere scomposto in diversi passaggi fondamentali:

1. Inizializzazione: La ricerca inizia con la creazione di una **popolazione iniziale** di individui, ciascuno dei quali rappresenta una soluzione potenziale al problema. Ogni individuo è generalmente rappresentato come una **stringa di bit** o un vettore di valori numerici (cromosomi), che codifica i parametri della soluzione. Questa popolazione iniziale può essere generata casualmente oppure basandosi su una stima preliminare delle soluzioni buone.

2. Funzione di fitness: La **funzione di fitness** valuta quanto un individuo (soluzione) è “adatto”. Più alta è la fitness, migliore è la soluzione. Questa funzione guida l'evoluzione della popolazione selezionando gli individui migliori per la riproduzione. La fitness può rappresentare il costo, la qualità, la performance o qualunque altro criterio legato al problema.

3. Selezione: Nella fase di selezione, gli individui con una fitness maggiore hanno più probabilità di essere scelti per la riproduzione. Metodi comuni per la selezione includono:

- **Roulette Wheel Selection:** Gli individui vengono scelti proporzionalmente alla loro fitness. Gli individui migliori hanno una maggiore probabilità di essere selezionati.
- **Torneo:** Un sottoinsieme casuale di individui compete in un “torneo”, e l'individuo con la fitness migliore viene selezionato.
- **Selezione Stocastica con Campionamento Universale:** Una variante della roulette wheel selection che riduce la varianza nei risultati della selezione. In questo metodo, un numero di frecce equidistanti viene lanciato sulla “ruota” della roulette, selezionando più individui contemporaneamente. Questo garantisce una selezione più uniforme e riduce la possibilità che individui con fitness molto alta o molto bassa siano selezionati in modo sproporzionato.
- **Selezione per Troncamento:** Viene scelto un sottoinsieme dei migliori individui della popolazione (es. i migliori 10%) e solo questi individui vengono utilizzati per la riproduzione. Questo metodo impone una forte pressione selettiva, poiché i migliori individui dominano completamente la generazione successiva.
- **Selezione Basata su Ranking:** Gli individui vengono ordinati in base alla loro fitness e le probabilità di selezione sono determinate in base al rango piuttosto che al valore assoluto della fitness. Questo metodo riduce l'effetto di differenze estreme nella fitness tra gli individui, limitando il dominio dei pochi individui migliori.

4. Crossover: Il **crossover** è un operatore genetico che combina il patrimonio genetico di due “genitori” per produrre “figli”. Esistono diverse strategie di crossover, tra cui:

- **Crossover a singolo punto:** Viene scelto un punto di crossover sulla stringa genetica dei genitori, e le parti successive al punto vengono scambiate tra i due genitori. Questo metodo è semplice da implementare e mantiene inalterata buona parte del materiale genetico di entrambi i genitori.
- **Crossover a due punti:** Due punti di crossover vengono scelti e le sezioni comprese tra i punti vengono scambiate. Questa tecnica fornisce una maggiore mescolanza dei geni rispetto al crossover a singolo punto e permette una più ampia esplorazione dello spazio delle soluzioni.
- **Crossover uniforme (Uniform Crossover):** Ogni gene del figlio viene scelto casualmente tra i geni corrispondenti dei due genitori, indipendentemente dalla loro posizione nella stringa genetica. Questo metodo garantisce una mescolanza maggiore di informazioni genetiche, ma può introdurre una forte perturbazione nelle strutture già ottimizzate dei genitori.
- **Crossover aritmetico:** Utilizzato principalmente per problemi con rappresentazioni numeriche, il crossover aritmetico combina i valori dei genitori attraverso operazioni matematiche. Ad esempio, dati due genitori con valori p_1 e p_2 , il figlio potrebbe avere un valore $c = \alpha p_1 + (1 - \alpha)p_2$, dove α è un fattore di mescolanza tra 0 e 1. Questo metodo è utile per mantenere una continuità nei parametri.
- **Crossover a k punti:** Estende l’idea del crossover a singolo e doppio punto a k punti. Si scelgono k punti sulla stringa genetica, e si alternano le sezioni di stringhe tra i genitori per formare il figlio. Maggiore è k , maggiore è la mescolanza tra i genitori, ma ciò può anche aumentare la casualità del processo.
- **Crossover ordinato (Order Crossover):** Utilizzato principalmente per problemi di ottimizzazione in cui l’ordine degli elementi è importante (ad esempio, il problema del commesso viaggiatore), il crossover ordinato preserva l’ordine relativo dei geni di un genitore e riempie i restanti geni con quelli dell’altro genitore, mantenendo il loro ordine.
- **Crossover di ciclo (Cycle Crossover):** Anch’esso utilizzato nei problemi in cui l’ordine degli elementi è importante, il crossover di ciclo costruisce il figlio identificando cicli tra i genitori. Il primo ciclo è preso da un genitore, e i cicli successivi sono presi dall’altro genitore, preservando così l’ordine.
- **Crossover a segmento casuale:** Viene scelto un segmento casuale della stringa genetica dei genitori e scambiato per produrre il figlio. Questa tecnica è utile per introdurre variazioni locali limitate, mantenendo comunque una parte consistente del patrimonio genetico originale.

5. Mutazione: La **mutazione** è un operatore che introduce variazioni casuali negli individui. Questo operatore è essenziale per mantenere la diversità genetica all’interno della popolazione e per evitare che l’algoritmo si blocchi in massimi locali, esplorando così nuove aree dello spazio di ricerca che il crossover da solo non coprirebbe. A seconda della rappresentazione degli individui (binaria, numerica, o ordinata), esistono diverse tecniche di mutazione che possono essere applicate. Di seguito sono elencate le più comuni:

- **Mutazione a bit singolo (Flip Bit Mutation):** In una rappresentazione binaria, un bit viene selezionato casualmente e il suo valore viene invertito (da 0 a 1 o da 1 a 0). Questa è la forma più semplice di mutazione ed è ampiamente utilizzata in problemi con codifica binaria.
- **Mutazione a più bit (Multi-bit Mutation):** Simile alla mutazione a bit singolo, ma invece di invertire un solo bit, vengono invertiti k bit casuali nella stringa genetica. Questo introduce variazioni più ampie nel cromosoma rispetto alla mutazione a singolo bit.
- **Mutazione gaussiana (Gaussian Mutation):** Utilizzata per rappresentazioni numeriche, questa tecnica altera i valori dei geni aggiungendo un valore casuale derivato da una distribuzione gaussiana con media zero. Il risultato è una variazione continua e “leggera” dei parametri, ideale per problemi di ottimizzazione in cui la precisione dei valori numerici è cruciale.
- **Mutazione di scambio (Swap Mutation):** In una rappresentazione permutativa (come nel problema del commesso viaggiatore), due geni vengono selezionati casualmente e scambiati di posizione. Questo tipo di mutazione è utile per problemi in cui l’ordine degli elementi è significativo, poiché altera la sequenza degli elementi senza cambiare la loro presenza.
- **Mutazione per inversione (Inversion Mutation):** In una stringa genetica, viene selezionato un segmento casuale e l’ordine dei geni in quel segmento viene invertito. Questo metodo è particolarmente utile per problemi di ottimizzazione dell’ordine, poiché altera localmente la disposizione degli elementi.
- **Mutazione casuale completa (Random Resetting):** In una rappresentazione numerica, un gene selezionato casualmente viene sostituito da un nuovo valore casuale all’interno del dominio permesso per quel gene. Questa tecnica può introdurre cambiamenti drastici in una soluzione, rendendola utile in situazioni in cui è necessario esplorare rapidamente una parte diversa dello spazio di ricerca.
- **Mutazione per permutazione (Permutation Mutation):** Utilizzata in rappresentazioni a permutazione (ad esempio, nel problema del commesso viaggiatore), un sottoinsieme di geni viene selezionato casualmente e il loro ordine viene permutato casualmente. Questo tipo di mutazione introduce nuove combinazioni di elementi senza alterare il contenuto della soluzione.
- **Mutazione uniformemente distribuita (Uniform Mutation):** In una rappresentazione numerica, un gene viene alterato di una quantità casuale derivata da una distribuzione uniforme, piuttosto che da una distribuzione gaussiana. Questo metodo può introdurre variazioni più ampie rispetto alla mutazione gaussiana, esplorando porzioni più distanti dello spazio di ricerca.
- **Mutazione per inserimento (Insertion Mutation):** Un gene viene selezionato casualmente, rimosso dalla sua posizione originale e reinserito in una posizione differente all’interno della stessa stringa. Questo tipo di mutazione è utile nei problemi di ottimizzazione in cui l’ordine degli elementi è importante.
- **Mutazione a scala (Non-Uniform Mutation):** Utilizzata per la rappresentazione numerica, la mutazione a scala introduce variazioni più piccole man mano che l’algoritmo avanza verso le generazioni finali. Questo riduce la variabilità verso la fine della ricerca, permettendo una maggiore precisione quando si è vicini alla soluzione ottimale.
- **Mutazione adattativa (Adaptive Mutation):** In questo metodo, la probabilità di mutazione varia dinamicamente durante l’evoluzione in base alla diversità della popolazione o al progresso

dell'algoritmo. Se la popolazione diventa troppo omogenea, la probabilità di mutazione aumenta per introdurre nuova variabilità, mentre se la popolazione è già altamente diversificata, la probabilità di mutazione viene ridotta.

6. Terminazione: L'algoritmo genetico può essere terminato quando viene soddisfatta una condizione di arresto, come:

- Un numero fisso di generazioni.
- Una soglia di fitness raggiunta.
- Nessun miglioramento significativo dopo un certo numero di generazioni.

7.1 Progettazione di algoritmi genetici: qualche osservazione

Progettare un algoritmo genetico efficace richiede una comprensione approfondita delle caratteristiche del problema e un'attenta configurazione dei vari componenti dell'algoritmo. Di seguito vengono fornite alcune linee guida pratiche per aiutare gli studenti a scegliere le migliori strategie da adottare durante la progettazione di un GA.

7.1.1 Comprendere il problema e lo spazio delle soluzioni

Il primo passo per configurare correttamente un GA è comprendere a fondo il problema da risolvere:

- **Tipo di problema:** Identificare se il problema è continuo, discreto, combinatorio o un mix di questi. Ad esempio, problemi di ottimizzazione continua richiederanno una rappresentazione numerica, mentre problemi combinatori, come il commesso viaggiatore, potrebbero richiedere una rappresentazione basata su permutazioni.
- **Funzione obiettivo:** Valutare se la funzione obiettivo è semplice da calcolare o computazionalmente costosa. Se la valutazione della fitness è molto costosa, potrebbe essere utile ridurre la dimensione della popolazione o utilizzare metodi come la *elitism* per preservare i migliori individui e ridurre il numero di valutazioni necessarie.

7.1.2 Dimensione della popolazione

La dimensione della popolazione è un parametro critico che influisce sull'efficienza del GA:

- **Popolazioni piccole:** Consentono un'evoluzione più rapida, ma possono portare a una perdita di diversità genetica e a una convergenza prematura verso soluzioni sub-ottimali.
- **Popolazioni grandi:** Offrono una maggiore esplorazione dello spazio di ricerca e riducono il rischio di convergenza prematura, ma richiedono più tempo e risorse computazionali.

- **Regola pratica:** Se non hai una conoscenza specifica del problema, inizia con una popolazione moderata (es. 50-100 individui) e regola il parametro in base ai risultati iniziali. Un buon approccio è quello di eseguire test con diverse dimensioni di popolazione e confrontare le performance.

7.1.3 Scelta della strategia di selezione

La scelta della strategia di selezione può influenzare significativamente la velocità di convergenza e la qualità delle soluzioni finali:

- **Selezione a Roulette:** Utilizza questa tecnica se vuoi che gli individui con fitness minore abbiano comunque una probabilità di essere selezionati. È utile per mantenere la diversità genetica nelle prime fasi dell'evoluzione.
- **Selezione a Torneo:** Se vuoi aumentare la pressione selettiva (ossia, accelerare la selezione dei migliori individui), usa la selezione a torneo. Se la tua popolazione sta convergendo troppo rapidamente, puoi ridurre la dimensione del torneo per mantenere una maggiore variabilità.
- **Selezione Elitaria:** Integra una piccola percentuale di selezione elitaria (ad esempio, il 5%) per assicurarti che i migliori individui non vengano persi nel processo di crossover e mutazione.

7.1.4 Frequenza di crossover e mutazione

La scelta delle probabilità di **crossover** e **mutazione** determina l'equilibrio tra esplorazione e sfruttamento dello spazio delle soluzioni:

- **Crossover:** Generalmente, una probabilità di crossover elevata (70-90%) è consigliata per favorire la combinazione delle caratteristiche migliori degli individui. In problemi dove la struttura della soluzione è particolarmente importante (es. il problema del commesso viaggiatore), potrebbe essere necessario sperimentare con tecniche di crossover specifiche come il *crossover ordinato*.
- **Mutazione:** La mutazione ha un ruolo cruciale nell'introdurre nuova diversità genetica. Inizia con una bassa probabilità di mutazione (0.1-1%) e aumenta gradualmente se osservi che la popolazione sta convergendo troppo rapidamente su una soluzione sub-ottimale. La *mutazione gaussiana* è utile per problemi continui, mentre la *mutazione a bit singolo* è efficace per rappresentazioni binarie.

7.1.5 Scelta delle tecniche di crossover e mutazione

La scelta delle tecniche di crossover e mutazione dipende dalla rappresentazione del problema:

- **Crossover:** Sperimenta con *crossover a singolo punto* e *crossover a due punti* per rappresentazioni binarie, mentre per rappresentazioni numeriche, il *crossover aritmetico* può aiutare a mantenere la continuità tra le soluzioni.

- **Mutazione:** Per problemi combinatori, la *mutazione di scambio* o la *mutazione per inversione* possono rivelarsi efficaci nel mantenere soluzioni valide. In rappresentazioni numeriche, la *mutazione gaussiana* è spesso una scelta ragionevole, mentre la *mutazione uniformemente distribuita* può essere preferibile se vuoi esplorare nuove aree del problema.

7.1.6 Parametri di terminazione

Stabilire i criteri di terminazione è fondamentale per evitare che l'algoritmo continui a evolversi inutilmente:

- **Numero di generazioni:** Se non conosci il comportamento del problema, un buon punto di partenza è fissare un numero di generazioni massimo (ad esempio, 100-500), valutando poi i risultati.
- **Convergenza:** Un altro criterio comune è arrestare l'algoritmo se non si osservano miglioramenti significativi della fitness per un numero di generazioni consecutivo (es. 20 generazioni).
- **Fitness desiderata:** Se la funzione obiettivo ha un valore noto ottimale, puoi arrestare il GA una volta che la soluzione raggiunge un fitness vicino o uguale a tale valore.

7.1.7 Bilanciare esplorazione ed esploitazione

Il bilanciamento tra esplorazione di nuove soluzioni e sfruttamento delle soluzioni migliori è essenziale:

- **Esplorazione:** Se l'algoritmo sembra convergere troppo rapidamente verso una soluzione sub-ottimale, aumentare la probabilità di mutazione o ridurre la pressione selettiva può aiutare a esplorare nuove aree dello spazio di ricerca.
- **Exploitation:** Se l'algoritmo non sembra convergere e continua a esplorare senza miglioramenti significativi, potrebbe essere utile aumentare la pressione selettiva (ad esempio, con un torneo più grande) o ridurre la probabilità di mutazione per focalizzarsi maggiormente sulle soluzioni più promettenti.

7.1.8 Testare e affinare

La configurazione di un GA non è fissa e può richiedere vari tentativi per trovare il giusto equilibrio tra i diversi parametri. Alcune linee guida pratiche per ottimizzare le prestazioni del GA includono:

- **Sperimentazione iterativa:** Inizia con parametri standard e poi effettua piccoli aggiustamenti. Monitorare il comportamento del GA in termini di tempo di esecuzione e miglioramento della fitness è essenziale per identificare le configurazioni ottimali.

- **Analisi dei risultati:** Valuta la diversità della popolazione e i progressi delle soluzioni per capire se l'algoritmo è troppo orientato allo sfruttamento (e quindi rischia di perdere la diversità genetica) o se sta esplorando eccessivamente senza convergere.
- **Misurazione della convergenza:** Usa grafici per monitorare la fitness media e massima nel tempo, per identificare eventuali stagnazioni o fenomeni di convergenza prematura.

7.2 Analisi degli Algoritmi Genetici

Vantaggi:

- **Flessibilità:** Gli algoritmi genetici possono essere applicati a una vasta gamma di problemi, inclusi quelli che non possono essere descritti da modelli matematici chiari.
- **Robustezza:** Sono in grado di gestire problemi complessi con molti parametri, fornendo soluzioni accettabili anche quando i metodi esatti falliscono.
- **Esplorazione globale:** La combinazione di selezione, crossover e mutazione permette agli algoritmi genetici di esplorare ampiamente lo spazio delle soluzioni, evitando di rimanere bloccati in soluzioni sub-ottimali.

Svantaggi:

- **Convergenza lenta:** A causa della loro natura probabilistica, gli algoritmi genetici possono richiedere molte iterazioni per convergere verso una soluzione ottimale.
- **Richiedono molte valutazioni della fitness:** Ogni nuova generazione richiede una valutazione della fitness per tutti gli individui, il che può essere computazionalmente costoso, specialmente in problemi complessi.
- **Convergenza prematura:** A volte l'algoritmo può convergere troppo presto verso una soluzione sub-ottimale, specialmente se la popolazione perde diversità genetica.

7.3 Algoritmi Genetici: Miglioramenti

Gli algoritmi genetici possono essere ulteriormente potenziati per affrontare problemi complessi o per migliorare la loro efficacia in scenari specifici. Di seguito sono presentati alcuni miglioramenti comunemente adottati che possono essere applicati per ottimizzare la performance dei GA in contesti particolari, come i problemi multi-obiettivo, l'elitarismo e altre tecniche avanzate.

7.3.1 Elitarismo

L'**elitarismo** è una strategia che prevede di conservare un certo numero di individui con la fitness più elevata da una generazione alla successiva. Questo assicura che le migliori soluzioni non

vadano perse durante il crossover o la mutazione. L'elitarismo può migliorare significativamente la convergenza del GA, mantenendo sempre le migliori soluzioni trovate finora.

Vantaggi:

- Garantisce che il GA non perda le soluzioni migliori già trovate.
- Aumenta la probabilità di convergenza verso una soluzione ottimale o vicina all'ottimale.

Considerazioni:

- Un numero troppo elevato di individui elitari può ridurre la diversità genetica, aumentando il rischio di convergenza prematura.
- Una buona pratica è utilizzare una piccola percentuale della popolazione (ad esempio, il 1-5%) come individui elitari.

7.3.2 Algoritmi Genetici Multi-Obiettivo

Molti problemi reali richiedono l'ottimizzazione di più obiettivi contemporaneamente, che spesso possono essere in conflitto tra loro (ad esempio, massimizzare la qualità riducendo al minimo i costi). Gli **algoritmi genetici multi-obiettivo** (*Multi-Objective Genetic Algorithms*, MOGA) sono progettati per trovare un insieme di soluzioni ottimali che rappresentino compromessi tra questi obiettivi, noti come soluzioni di **Pareto ottimali**. In un contesto multi-obiettivo, una soluzione è detta **Pareto ottimale** se non esiste un'altra soluzione che migliora uno degli obiettivi senza peggiorarne almeno un altro. Questo concetto porta a un insieme di soluzioni, noto come **fronte di Pareto**, dove ciascuna soluzione rappresenta un compromesso ottimale tra gli obiettivi. Il ruolo degli algoritmi genetici multi-obiettivo è trovare e mantenere un insieme di soluzioni distribuite lungo questo fronte, permettendo al decisore di scegliere una soluzione in base alle preferenze tra i vari obiettivi.

Una soluzione A è detta **dominata** da un'altra soluzione B se B è migliore di A per almeno un obiettivo e non è peggiore per nessuno degli altri obiettivi. Un algoritmo genetico multi-obiettivo cerca di eliminare le soluzioni dominate, concentrandosi su quelle che non sono dominate da nessun'altra, ovvero le soluzioni Pareto ottimali.

Supponiamo di progettare un'auto e di voler ottimizzare sia il consumo di carburante sia le prestazioni. Aumentare le prestazioni dell'auto (ad esempio, la velocità massima) potrebbe comportare un aumento del consumo di carburante, mentre ridurre il consumo potrebbe penalizzare le prestazioni. Un algoritmo genetico multi-obiettivo, applicato a questo problema, cercherebbe di trovare il fronte di Pareto che contiene tutte le configurazioni ottimali di prestazioni e consumo, lasciando al progettista la scelta del miglior compromesso.

Strategie principali:

- **NSGA-II (Non-dominated Sorting Genetic Algorithm II):** È uno degli algoritmi genetici multi-obiettivo più noti. Classifica le soluzioni basandosi sulla dominanza di Pareto, mantenendo un equilibrio tra esplorazione dello spazio di ricerca e diversità delle soluzioni.
- **SPEA2 (Strength Pareto Evolutionary Algorithm 2):** Un altro metodo popolare che tiene conto della dominanza di Pareto e introduce meccanismi per mantenere la diversità nella popolazione durante la ricerca delle soluzioni ottimali.

Vantaggi:

- Permettono di ottenere un insieme di soluzioni, ognuna delle quali rappresenta un buon compromesso tra gli obiettivi.
- Mantengono una diversità elevata nelle soluzioni, utile per problemi complessi e con molteplici criteri di valutazione.

Considerazioni:

- Implementare algoritmi genetici multi-obiettivo richiede una maggiore complessità computazionale rispetto ai GA standard, a causa della necessità di valutare e mantenere un set di soluzioni non dominate.
- È importante mantenere un buon bilanciamento tra esplorazione e sfruttamento delle soluzioni, per garantire una copertura omogenea del fronte di Pareto.

7.3.3 Strategia dell'archivio

La **strategia dell'archivio** è un miglioramento spesso usato negli algoritmi genetici multi-obiettivo. In questa strategia, un archivio di soluzioni viene mantenuto durante l'intero processo evolutivo per salvare le soluzioni non dominate (soluzioni di Pareto ottimali) che vengono trovate. L'archivio è generalmente limitato in dimensione, e soluzioni nuove entrano nell'archivio solo se non sono dominate dalle soluzioni esistenti o se sostituiscono soluzioni dominate già presenti nell'archivio.

Vantaggi:

- Permette di preservare un set di soluzioni non dominate, garantendo una diversità di soluzioni Pareto ottimali.
- Aiuta a conservare le migliori soluzioni trovate durante l'evoluzione, evitando che vengano perse nel processo di mutazione o crossover.

Considerazioni:

- L'archivio deve essere gestito in modo efficiente per evitare che diventi troppo grande o che vengano mantenute soluzioni ridondanti.
- Implementare una strategia di aggiornamento dell'archivio richiede un attento bilanciamento tra la conservazione delle soluzioni di qualità e l'inserimento di nuove soluzioni promettenti.

7.3.4 Algoritmi Genetici Adattivi

Negli **algoritmi genetici adattivi**, i parametri chiave come le probabilità di mutazione e crossover non rimangono fissi, ma vengono modificati dinamicamente durante l'evoluzione, in base all'andamento dell'algoritmo. Questo tipo di approccio aiuta a bilanciare meglio esplorazione ed esploitazione durante le varie fasi dell'evoluzione.

Vantaggi:

- Permettono di migliorare l'esplorazione nelle prime fasi del GA e lo sfruttamento nelle fasi finali.
- Possono ridurre il rischio di convergenza prematura e migliorare la qualità delle soluzioni finali.

Considerazioni:

- La progettazione di una strategia di adattamento efficace richiede una buona comprensione del comportamento del GA e del problema in questione.
- Parametri dinamici possono aggiungere complessità all'implementazione e richiedere maggiore tempo di sperimentazione.

7.3.5 Island Model (Modello ad isole)

Il **modello ad isole** divide la popolazione in sotto-popolazioni (isole), ciascuna delle quali evolve in modo indipendente per un certo numero di generazioni. Periodicamente, gli individui migliori migrano da un'isola all'altra, scambiando informazioni genetiche. Questo approccio aumenta la diversità genetica e migliora l'esplorazione.

Vantaggi:

- Migliora la diversità genetica mantenendo più popolazioni separate.
- Riduce il rischio di convergenza prematura in una singola popolazione.

Considerazioni:

- Il tasso e la frequenza della migrazione devono essere regolati con attenzione per bilanciare il mantenimento della diversità con il miglioramento della fitness.
- Implementare un modello ad isole richiede una gestione più complessa delle sotto-popolazioni.

7.3.6 Crossover e Mutazione Non-Uniformi

Il **crossover non-uniforme** e la **mutazione non-uniforme** variano l'intensità con cui questi operatori influenzano gli individui man mano che l'algoritmo procede. Nelle prime generazioni, si favorisce una maggiore esplorazione (con mutazioni e crossover più ampi), mentre nelle fasi avanzate l'intensità diminuisce per concentrarsi su un'esplorazione più locale e su piccole modifiche per migliorare soluzioni già promettenti.

Vantaggi:

- Migliora l'equilibrio tra esplorazione iniziale e sfruttamento finale.
- Permette di adattarsi meglio alle diverse fasi dell'evoluzione, riducendo il rischio di blocchi su massimi locali.

Considerazioni:

- Richiede un controllo accurato dei parametri che regolano la riduzione dell'intensità di mutazione e crossover.
- Troppa riduzione può portare a una mancanza di esplorazione nelle ultime fasi, mentre troppa esplorazione può rallentare la convergenza.

7.3.7 Ibridi con altre tecniche di ottimizzazione

Un approccio comune è combinare i GA con altre tecniche di ottimizzazione, creando algoritmi **ibridi**, anche detti **algoritmi memetici**. Ad esempio, una volta che il GA ha trovato una soluzione promettente, può essere usato un metodo di ottimizzazione locale (come l'*hill climbing*) per affinare ulteriormente quella soluzione.

Vantaggi:

- Permette di ottenere i benefici di più metodi: esplorazione globale dai GA e ottimizzazione locale da tecniche come la discesa del gradiente o l'*hill climbing*.
- Migliora la qualità delle soluzioni finali e accelera la convergenza.

Considerazioni:

- Richiede una buona integrazione tra i diversi metodi per evitare conflitti o sovrapposizioni.
- La scelta di quando e come applicare l'ottimizzazione locale è fondamentale per l'efficacia dell'algoritmo ibrido.

8 Conclusioni

Gli algoritmi genetici rappresentano una potente metodologia di ottimizzazione, particolarmente utile quando lo spazio di ricerca è vasto e complesso. Sebbene abbiano alcuni svantaggi, la loro flessibilità consente di indirizzare molte limitazioni, fornendo uno strumento adattabile in varie circostanze.

9 Quiz time

Domanda 1: Qual è il ruolo principale della **funzione di fitness** in un algoritmo genetico?

- A) Misurare quanto una soluzione è vicina all'ottimo desiderato.
- B) Generare nuove soluzioni.
- C) Aumentare la diversità genetica.
- D) Selezionare i geni per la mutazione.

Risposta corretta: A) Misurare quanto una soluzione è vicina all'ottimo desiderato.

Domanda 2: Quale dei seguenti operatori è responsabile del **mescolamento** dei geni tra due genitori per creare nuovi figli?

- A) Mutazione.
- B) Selezione.
- C) Crossover.
- D) Fitness.

Risposta corretta: C) Crossover.

Domanda 3: Nella **selezione a torneo**, cosa determina la scelta degli individui?

- A) La dimensione del cromosoma.

- B) La fitness degli individui in un gruppo selezionato casualmente.
- C) La probabilità di mutazione.
- D) Il numero di generazioni.

Risposta corretta: B) La fitness degli individui in un gruppo selezionato casualmente.

Domanda 4: Quale è l'obiettivo dell'**operatore di mutazione** in un algoritmo genetico?

- A) Mescolare il materiale genetico tra genitori.
- B) Aumentare la diversità genetica della popolazione.
- C) Eliminare le soluzioni sub-ottimali.
- D) Stabilire la fitness iniziale degli individui.

Risposta corretta: B) Aumentare la diversità genetica della popolazione.

Domanda 5: Quale strategia di crossover prevede lo scambio di segmenti di cromosomi a più di due punti di taglio?

- A) Crossover a singolo punto.
- B) Crossover a due punti.
- C) Crossover uniforme.
- D) Crossover a k punti.

Risposta corretta: D) Crossover a k punti.

Domanda 6: In quale situazione un **GA** potrebbe convergere verso una soluzione sub-ottimale?

- A) Quando la funzione di fitness è molto semplice.
- B) Quando la diversità genetica diminuisce troppo rapidamente.
- C) Quando la dimensione della popolazione è troppo grande.
- D) Quando viene usata una selezione a torneo troppo piccola.

Risposta corretta: B) Quando la diversità genetica diminuisce troppo rapidamente.

Domanda 7: Quale dei seguenti è un vantaggio dell'uso di **elitarismo** in un GA?

- A) Mantiene sempre le migliori soluzioni nelle generazioni successive.

- B) Aumenta la velocità di mutazione.
- C) Aumenta il numero di individui per generazione.
- D) Garantisce una maggiore diversità genetica.

Risposta corretta: A) Mantiene sempre le migliori soluzioni nelle generazioni successive.

Domanda 8: Nella **mutazione gaussiana**, quale delle seguenti affermazioni è corretta?

- A) I valori dei geni vengono invertiti.
- B) I valori dei geni vengono modificati di una quantità casuale derivata da una distribuzione gaussiana.
- C) I valori dei geni vengono scambiati tra gli individui.
- D) I valori dei geni vengono eliminati.

Risposta corretta: B) I valori dei geni vengono modificati di una quantità casuale derivata da una distribuzione gaussiana.

Domanda 9: Quando viene utilizzato il **modello ad isole** negli algoritmi genetici, cosa succede durante la **migrazione**?

- A) Gli individui migliori di una popolazione vengono scambiati tra le isole.
- B) Gli individui peggiori di una popolazione vengono eliminati.
- C) Viene aumentata la probabilità di mutazione.
- D) Viene ridotta la dimensione della popolazione.

Risposta corretta: A) Gli individui migliori di una popolazione vengono scambiati tra le isole.

Domanda 10: Cosa succede durante il processo di **crossover uniforme**?

- A) I cromosomi vengono scambiati a un punto fisso.
- B) Ogni gene del figlio viene selezionato casualmente da uno dei genitori.
- C) I migliori individui vengono scelti in base alla fitness.
- D) I geni vengono scambiati solo in uno specifico segmento.

Risposta corretta: B) Ogni gene del figlio viene selezionato casualmente da uno dei genitori.

10 Esercizi sulla Progettazione di un Algoritmo Genetico

Di seguito vengono proposti diversi esercizi che, a titolo di esempio, mostrano come progettare un algoritmo genetico per la risoluzione di problemi reali.

Scenario: Ottimizzazione della Pianificazione dei Turni Lavorativi Supponiamo di essere incaricati di progettare un algoritmo genetico per risolvere il problema della pianificazione dei turni di lavoro in un'azienda. L'azienda ha bisogno di ottimizzare l'assegnazione dei turni di lavoro per 50 dipendenti, assicurando che:

- I dipendenti rispettino i vincoli legali sulle ore di lavoro massime settimanali.
- Ogni turno sia coperto dal numero richiesto di dipendenti, garantendo competenze specifiche (es. almeno un tecnico, un supervisore, ecc.).
- I dipendenti siano distribuiti in modo uniforme tra i turni per evitare sovraccarico di lavoro.
- Le preferenze dei dipendenti (giorni liberi e disponibilità) siano rispettate il più possibile.

Il nostro obiettivo è quindi massimizzare la soddisfazione dei dipendenti e garantire che i turni siano bilanciati e conformi ai vincoli legali.

Domanda:

Progetta un algoritmo genetico per risolvere questo problema. Si discuta delle seguenti scelte in termini di progettazione di un GA:

- **1. Rappresentazione della popolazione:** Come rappresenteresti i cromosomi per questo problema? Quale forma di codifica sceglieresti?
- **2. Funzione di fitness:** Come potresti definire una funzione di fitness che bilanci i diversi obiettivi (rispetto dei vincoli, soddisfazione dei dipendenti, ecc.)?
- **3. Operatori di crossover:** Quale tecnica di crossover utilizzeresti per combinare i cromosomi? Perché?
- **4. Operatori di mutazione:** Come definiresti un operatore di mutazione efficace per esplorare nuove soluzioni senza rompere i vincoli?
- **5. Parametri dell'algoritmo:** Che dimensione di popolazione, probabilità di crossover e probabilità di mutazione sceglieresti? Come bilanceresti esplorazione ed esploitazione?
- **6. Miglioramenti avanzati:** Considereresti di applicare elitismo, un modello ad isole o un'archiviazione delle soluzioni? Perché?

Soluzione suggerita

1. Rappresentazione della popolazione: Ogni cromosoma può essere rappresentato come una matrice di dimensioni 50×7 , dove 50 è il numero di dipendenti e 7 rappresenta i giorni della settimana. Ogni elemento della matrice contiene un valore binario o intero che indica se un dipendente è assegnato a un turno specifico in un determinato giorno e quale ruolo ricopre. Ad esempio, un "1" potrebbe indicare che il dipendente è assegnato al turno, e un "0" che non lo è.

2. Funzione di fitness: La funzione di fitness deve considerare i seguenti fattori:

- Penalizzare assegnazioni che violano i vincoli legali (es. superamento delle ore settimanali).
- Premiare soluzioni che rispettano le preferenze dei dipendenti per giorni liberi o turni preferiti.
- Penalizzare squilibri nella distribuzione dei turni tra i dipendenti.
- Premiare soluzioni che rispettano i requisiti di competenze per ciascun turno.

La fitness potrebbe essere una somma pesata di queste componenti, con penalità assegnate a violazioni dei vincoli.

3. Operatori di crossover: Un **crossover a due punti** potrebbe essere una buona scelta, poiché permette di scambiare blocchi di turni tra i due genitori, mantenendo la struttura di base del cromosoma (ossia la matrice dei turni). Alternativamente, un **crossover a segmento casuale** potrebbe funzionare bene se i segmenti rappresentano blocchi di giorni consecutivi.

4. Operatori di mutazione: La mutazione potrebbe consistere nel **cambiare l'assegnazione di un dipendente** in uno specifico turno, modificando il suo stato da "libero" a "occupato" o viceversa. Un'altra opzione è la **mutazione di scambio**, che potrebbe scambiare i turni tra due dipendenti in modo casuale. Questo permetterebbe di esplorare nuove soluzioni senza rompere i vincoli troppo severamente.

5. Parametri dell'algoritmo:

- **Dimensione della popolazione:** Una popolazione di 100-200 individui potrebbe essere un buon punto di partenza, offrendo un equilibrio tra esplorazione e capacità computazionale.
- **Probabilità di crossover:** Una probabilità elevata (80-90%) è raccomandata per assicurare una combinazione efficace di soluzioni genitoriali.
- **Probabilità di mutazione:** Una probabilità di mutazione moderata (1-5%) può garantire che la popolazione mantenga la diversità senza distruggere soluzioni promettenti.

Bilanciare esplorazione ed esploitazione è cruciale. Se si osserva una convergenza prematura verso soluzioni sub-ottimali, aumentare leggermente la probabilità di mutazione potrebbe aiutare a mantenere la diversità.

6. Miglioramenti avanzati:

- **Elitarismo:** Può essere utile applicare una forma di elitarismo (ad esempio, conservare il 5% delle migliori soluzioni) per garantire che le soluzioni migliori non vengano perse.
- **Modello ad isole:** Dividere la popolazione in sottogruppi che evolvono indipendentemente potrebbe migliorare la diversità genetica e ridurre il rischio di convergenza prematura. La migrazione periodica tra isole può essere implementata per scambiare soluzioni promettenti.
- **Archiviazione delle soluzioni:** Se il problema prevede l'ottimizzazione di più obiettivi (ad esempio, bilanciare tra soddisfazione dei dipendenti e rispetto dei vincoli), mantenere un archivio di soluzioni Pareto ottimali potrebbe essere utile.

Scenario: Ottimizzazione del Layout di una Fabbrica Immagina di dover progettare un algoritmo genetico per ottimizzare il layout di una fabbrica. L'obiettivo è minimizzare la distanza totale percorsa dai materiali tra le varie stazioni di lavoro, mantenendo i macchinari disposti in modo da rispettare vincoli di spazio e sicurezza. La fabbrica ha 10 macchine che devono essere posizionate su un'area di lavoro rettangolare con spazi fissi predefiniti.

Gli obiettivi principali del problema sono:

- Ridurre al minimo la distanza totale che i materiali devono percorrere tra le macchine.
- Massimizzare l'efficienza del layout mantenendo i macchinari adiacenti alle stazioni complementari.
- Rispettare i vincoli di spazio tra le macchine e garantire che vengano rispettate le norme di sicurezza.

Domanda:

Progetta un algoritmo genetico per risolvere questo problema. Descrivi le scelte ragionevoli per la progettazione di un GA, rispondendo alle seguenti domande:

- **1. Rappresentazione della popolazione:** Quale tipo di rappresentazione (codifica) useresti per modellare il layout delle macchine all'interno della fabbrica?
- **2. Funzione di fitness:** Come definirai una funzione di fitness che prenda in considerazione sia la minimizzazione delle distanze che il rispetto dei vincoli di sicurezza?
- **3. Operatori di crossover:** Quale tipo di crossover useresti per combinare i layout genitoriali? Perché questa scelta?

- **4. Operatori di mutazione:** Come applicheresti un operatore di mutazione efficace per introdurre variazioni nei layout senza rompere i vincoli?
- **5. Parametri dell'algoritmo:** Come decideresti la dimensione della popolazione, la probabilità di crossover e la probabilità di mutazione in questo scenario?
- **6. Miglioramenti avanzati:** Considereresti l'uso di elitarismo, modello ad isole o archiviazione delle soluzioni in questo contesto? Spiega il perché.

Soluzione suggerita

1. Rappresentazione della popolazione: Per rappresentare il layout della fabbrica, un cromosoma può essere una permutazione delle 10 macchine disposte in una griglia di dimensioni predefinite. Ogni gene nel cromosoma rappresenta una macchina e la sua posizione nella griglia. Ad esempio, una sequenza potrebbe indicare che la macchina 1 si trova nella posizione (1,1), la macchina 2 nella posizione (1,2), e così via.

2. Funzione di fitness: La funzione di fitness deve valutare due aspetti chiave:

- **Minimizzazione delle distanze:** Sommare la distanza totale percorsa dai materiali tra le macchine. Questo potrebbe essere calcolato usando la distanza euclidea o manhattan tra le macchine, a seconda delle regole di movimento stabilite.
- **Rispetto dei vincoli di sicurezza:** Penalizzare layout che non rispettano i vincoli di spazio e sicurezza. Ad esempio, se due macchine sono posizionate troppo vicine tra loro o bloccano una via d'uscita di emergenza, viene applicata una penalità nella funzione di fitness.

La fitness finale potrebbe essere una combinazione pesata tra minimizzazione delle distanze e penalità per violazioni dei vincoli.

3. Operatori di crossover: Un **crossover per permutazioni**, come il *crossover ordinato*, sarebbe appropriato in questo caso, poiché ogni layout rappresenta una permutazione delle macchine. Questo garantisce che i figli generati mantengano un layout valido, in cui ogni macchina è posizionata una sola volta e non vengono introdotti duplicati.

4. Operatori di mutazione: La **mutazione di scambio** sarebbe efficace in questo contesto. Questa operazione seleziona due macchine casualmente e scambia le loro posizioni. In questo modo, si esplorano nuove configurazioni di layout senza rompere i vincoli di base (ogni macchina rimane posizionata una sola volta). Alternativamente, una **mutazione per inserimento** potrebbe spostare una macchina da una posizione a un'altra all'interno della griglia, spostando tutte le altre di conseguenza.

5. Parametri dell'algoritmo:

- **Dimensione della popolazione:** Una popolazione di 50-100 individui potrebbe essere sufficiente, considerando che il problema non è di dimensioni eccessivamente grandi, ma comunque complesso per via dei vincoli spaziali e delle possibili permutazioni.
- **Probabilità di crossover:** Un crossover ad alta probabilità (80-90%) è raccomandato per generare nuove soluzioni da combinazioni di layout parentali.
- **Probabilità di mutazione:** Una bassa probabilità di mutazione (1-5%) è appropriata per evitare che i layout migliori vengano alterati troppo spesso, ma è comunque necessaria per esplorare configurazioni alternative.

6. Miglioramenti avanzati:

- **Elitarismo:** Includere un 5% di elitarismo può assicurare che i layout migliori non vengano persi durante il crossover e la mutazione.
- **Modello ad isole:** Se il problema sembra convergere troppo rapidamente su soluzioni sub-ottimali, un modello ad isole potrebbe essere utile. Ogni isola potrebbe esplorare configurazioni diverse di layout e occasionalmente scambiare individui migliori tra le isole.
- **Archiviazione delle soluzioni:** Se esistono più obiettivi contrastanti (ad esempio, minimizzazione della distanza e ottimizzazione dello spazio), un archivio di soluzioni Pareto ottimali potrebbe essere utile per mantenere una varietà di soluzioni ben bilanciate tra i vari obiettivi.

Scenario: Ottimizzazione di una Rete di Trasporti Immagina di essere incaricato di ottimizzare la rete di trasporti di una città. L'obiettivo è ridurre al minimo il tempo medio di percorrenza tra diverse stazioni e nodi della rete, considerando vincoli come:

- La capacità massima delle strade e dei mezzi di trasporto.
- I costi di costruzione o ristrutturazione delle infrastrutture.
- La necessità di servire tutte le aree della città in modo efficiente.
- Il rispetto dei limiti di budget.

L'obiettivo dell'algoritmo genetico sarà quello di trovare una configurazione ottimale della rete, che migliori l'efficienza complessiva dei trasporti, riducendo i tempi di percorrenza e i costi, mantenendo al contempo la rete operativa all'interno dei vincoli di budget e capacità.

Domanda:

Progetta un algoritmo genetico per risolvere questo problema di ottimizzazione della rete di trasporti. Descrivi le scelte progettuali per ciascuno dei seguenti aspetti:

- **1. Rappresentazione della popolazione:** Come rappresentaresti una rete di trasporti all'interno di un cromosoma? Quale codifica useresti per modellare le connessioni tra le stazioni e le capacità delle strade?
- **2. Funzione di fitness:** Come definiresti una funzione di fitness che tenga conto sia della minimizzazione del tempo di percorrenza sia dei vincoli di budget e capacità?
- **3. Operatori di crossover:** Quale tecnica di crossover useresti per combinare le reti genitoriali? Perché questa scelta è appropriata?
- **4. Operatori di mutazione:** Quale tipo di mutazione applicheresti per esplorare nuove soluzioni di rete? Come eviteresti di rompere i vincoli di capacità o budget?
- **5. Parametri dell'algoritmo:** Quali scelte faresti per la dimensione della popolazione, la probabilità di crossover e la probabilità di mutazione? Qual è il giusto equilibrio tra esplorazione e sfruttamento in questo contesto?
- **6. Miglioramenti avanzati:** Utilizzeresti tecniche come elitarismo, modello ad isole o strategie di archiviazione per migliorare la performance dell'algoritmo? Spiega il perché.

Soluzione suggerita

1. Rappresentazione della popolazione: Ogni cromosoma potrebbe rappresentare la rete di trasporti come una **matrice di adiacenza**, dove ciascun nodo della rete rappresenta una stazione e ciascun arco rappresenta una strada o una connessione di trasporto. I valori nei nodi potrebbero rappresentare le capacità delle strade o il costo di costruzione o manutenzione. In alternativa, la rete può essere rappresentata come un grafo, dove ogni gene del cromosoma descrive una connessione tra due stazioni, insieme ai parametri della capacità e del costo.

2. Funzione di fitness: La funzione di fitness dovrebbe essere basata su due criteri principali:

- **Tempo di percorrenza:** La fitness dovrebbe premiare le reti che minimizzano il tempo medio di percorrenza tra le varie stazioni, calcolato usando le distanze e la velocità di percorrenza lungo ciascuna connessione.
- **Vincoli di capacità e budget:** La funzione dovrebbe penalizzare reti che superano i limiti di capacità (ad esempio, strade o mezzi di trasporto sovraccarichi) o che eccedono il budget totale disponibile per la costruzione o la manutenzione della rete.

Il valore finale della fitness potrebbe essere una combinazione pesata del tempo medio di percorrenza e delle penalità per il superamento dei vincoli di capacità e budget.

3. Operatori di crossover: Un **crossover basato su grafi**, come il *crossover per grafi minimi*, potrebbe essere utilizzato per combinare due reti genitoriali, assicurandosi che la rete risultante rimanga connessa e valida. In alternativa, un **crossover a segmenti** può essere usato per combinare blocchi di collegamenti tra i nodi, garantendo che le soluzioni figlie ereditino buone configurazioni di connessioni dai genitori.

4. Operatori di mutazione: La **mutazione per aggiunta o rimozione di collegamenti** sarebbe utile in questo contesto. Potresti aggiungere un nuovo collegamento tra due stazioni, migliorando potenzialmente la connettività della rete, o rimuovere una connessione per ridurre i costi. In alternativa, una **mutazione per modifica della capacità** potrebbe cambiare la capacità di una strada esistente, aumentando o diminuendo il numero di mezzi che può gestire, senza rompere i vincoli di capacità o budget.

5. Parametri dell'algoritmo:

- **Dimensione della popolazione:** Iniziare con una popolazione di 100-200 individui permetterebbe di esplorare una varietà di configurazioni di rete. Dato che il problema riguarda la gestione di infrastrutture complesse, una dimensione della popolazione più ampia favorisce una migliore esplorazione.
- **Probabilità di crossover:** Una probabilità di crossover alta (80-90%) potrebbe essere utile per generare nuove configurazioni di rete efficaci combinando layout promettenti dai genitori.
- **Probabilità di mutazione:** Una bassa probabilità di mutazione (1-5%) può essere sufficiente per introdurre variazioni senza alterare troppo le configurazioni valide di rete.

6. Miglioramenti avanzati:

- **Elitarismo:** Mantenere il 5% delle migliori reti tra le generazioni può garantire che le soluzioni migliori non vengano perse, specialmente se le configurazioni trovate ottimizzano bene il tempo di percorrenza e rispettano i vincoli.
- **Modello ad isole:** Il modello ad isole può essere utile se si osserva che l'algoritmo sta convergendo troppo rapidamente su soluzioni sub-ottimali. Dividere la popolazione in sotto-popolazioni che evolvono indipendentemente e che scambiano individui periodicamente può aumentare la diversità genetica.
- **Archiviazione delle soluzioni:** Se ci sono più obiettivi, come minimizzare i tempi di percorrenza e mantenere i costi sotto controllo, l'uso di un archivio di soluzioni Pareto ottimali può aiutare a conservare configurazioni di rete ben bilanciate tra questi obiettivi.

Scenario: Correzione Automatica di Bug nel Codice Sorgente Immagina di dover utilizzare un algoritmo genetico (GA) per risolvere automaticamente un difetto (bug) nel codice sorgente di un programma. Il programma in questione è un semplice algoritmo di ordinamento, ma in alcune circostanze specifiche, produce un output errato a causa di un bug logico. L'obiettivo è trovare una correzione al codice che elimini il bug e renda l'algoritmo di ordinamento corretto in tutti i casi di test forniti.

Il codice sorgente è rappresentato come una sequenza di istruzioni, e il GA dovrà esplorare varianti di questo codice per trovare una versione corretta. Le modifiche al codice possono includere la rimozione di linee, la sostituzione di operatori o variabili, o l'inserimento di nuove istruzioni.

Domanda:

Progetta un algoritmo genetico per risolvere questo problema. Discuta le scelte progettuali nei seguenti ambiti:

- **1. Rappresentazione della popolazione:** Come rappresenteresti il codice sorgente del programma all'interno di un cromosoma? Come modelleresti le possibili modifiche (inserzioni, cancellazioni, modifiche) al codice?
- **2. Funzione di fitness:** Come definiresti una funzione di fitness che valuta l'accuratezza di una variante di codice? Come integreresti l'esecuzione dei test di validità del programma?
- **3. Operatori di crossover:** Quale tecnica di crossover utilizzeresti per combinare due varianti di codice sorgente? Come garantiresti che il codice risultante rimanga sintatticamente valido?
- **4. Operatori di mutazione:** Come implementeresti un operatore di mutazione che modifica il codice sorgente? Quali tipi di modifiche sarebbero applicabili (es. cambio di variabili, modifiche di operatori, aggiunta o rimozione di linee)?
- **5. Parametri dell'algoritmo:** Che dimensione di popolazione, probabilità di crossover e probabilità di mutazione sceglieresti? Come bilanceresti l'esplorazione dello spazio delle soluzioni e la ricerca di una correzione rapida?
- **6. Miglioramenti avanzati:** Considereresti di applicare tecniche avanzate come elitismo o un modello ad isole in questo scenario? Se sì, come e perché?

Soluzione suggerita

1. Rappresentazione della popolazione: Ogni cromosoma può rappresentare una variante del codice sorgente come una sequenza di istruzioni del programma. Ad esempio, ogni gene nel cromosoma potrebbe rappresentare una singola istruzione del codice (come un'assegnazione, un ciclo o una condizione). Le possibili modifiche al codice possono essere modellate attraverso l'inserimento, la rimozione o la sostituzione di singole istruzioni o blocchi di codice.

2. Funzione di fitness: La funzione di fitness deve valutare l'accuratezza di ciascuna variante del codice. Questo può essere fatto eseguendo la versione modificata del codice su una suite di casi di test predefiniti. La fitness potrebbe essere calcolata in base a:

- **Numero di test passati:** Ogni variante di codice viene valutata in base a quanti test riesce a superare con successo.
- **Correttezza del risultato:** Se il programma produce output parzialmente corretto, la fitness può essere proporzionale alla distanza tra l'output prodotto e quello atteso (ad esempio, usando la differenza di stringhe o di valori numerici).
- **Efficienza:** Penalità possono essere applicate se la variante di codice introduce inefficienze (ad esempio, cicli inutili o complessità computazionale elevata).

3. Operatori di crossover: Un **crossover a singolo punto** o **crossover a blocchi** potrebbe essere appropriato per combinare varianti di codice sorgente. L'idea è scambiare blocchi di codice tra i due genitori, mantenendo la sintassi valida. Prima di applicare il crossover, il codice può essere segmentato in blocchi logici (ad esempio, funzioni o cicli) per assicurarsi che le parti scambiate tra i genitori risultino sensate.

4. Operatori di mutazione: L'operatore di mutazione potrebbe applicare le seguenti modifiche al codice sorgente:

- **Sostituzione di variabili o operatori:** Cambiare variabili usate nel codice o sostituire operatori logici e aritmetici.
- **Inserimento di nuove istruzioni:** Aggiungere nuove istruzioni in punti casuali del codice, come una nuova assegnazione o una condizione.
- **Rimozione di linee:** Eliminare una o più linee di codice che potrebbero essere responsabili del bug.

Le mutazioni devono essere progettate per mantenere la validità sintattica del codice, ad esempio verificando che i blocchi aperti (cicli, condizioni) vengano correttamente chiusi.

5. Parametri dell'algoritmo:

- **Dimensione della popolazione:** Una popolazione di 50-100 individui può essere adeguata, considerando la necessità di esplorare varianti di codice molto diverse, ma anche di mantenere la gestione computazionale sostenibile.
- **Probabilità di crossover:** Un crossover frequente (70-80%) è utile per esplorare combinazioni di soluzioni promettenti.
- **Probabilità di mutazione:** Una mutazione moderata (5-10%) può garantire che il codice sorgente esplori nuovi percorsi senza alterare eccessivamente le soluzioni corrette. Se l'algoritmo convergesse troppo rapidamente, un aumento della mutazione potrebbe mantenere la diversità genetica.

6. Miglioramenti avanzati:

- **Elitarismo:** Mantenere una piccola percentuale (5-10%) di soluzioni con fitness elevata tra le generazioni può garantire che le correzioni migliori non vengano perse.
- **Modello ad isole:** Se il problema è molto complesso, potrebbe essere utile suddividere la popolazione in sotto-popolazioni che evolvono indipendentemente e che scambiano soluzioni periodicamente, per evitare di rimanere bloccati in soluzioni locali.

11 Algoritmi di Ricerca con Avversari

Gli algoritmi di ricerca con avversari sono una categoria di algoritmi utilizzati per risolvere problemi in cui due o più agenti interagiscono tra loro, come avviene, ad esempio, nei giochi strategici. A differenza dei problemi di ricerca tradizionali, in cui un agente cerca di raggiungere un obiettivo senza opposizione, in questo contesto ogni agente deve prendere decisioni ottimali considerando le possibili mosse degli avversari.

Un esempio classico è il gioco degli scacchi, in cui ogni mossa non solo migliora la posizione di un giocatore, ma deve anche rispondere alle azioni dell'avversario. Per questo motivo, la pianificazione delle mosse richiede una strategia che massimizzi i guadagni in uno scenario collaborativo o competitivo.

Il metodo più noto per risolvere questi problemi è l'algoritmo **minimax**, che mira a minimizzare la massima perdita possibile. Insieme alla potatura **alfa-beta**, che riduce il numero di mosse da considerare senza compromettere la correttezza della soluzione, questi algoritmi sono alla base dell'intelligenza artificiale nei giochi e in altri ambienti competitivi.

11.1 Ambienti Multi-Agente

Gli ambienti multi-agente rappresentano un contesto in cui più agenti agiscono all'interno dello stesso ambiente e devono considerare le azioni degli altri agenti e gli effetti che queste producono. In particolare, discutiamo gli **ambienti competitivi**, in cui gli obiettivi degli agenti sono in conflitto. Questi problemi sono noti come **problemi di ricerca con avversari** o, semplicemente, **giochi**. La teoria dei giochi è una branca dell'economia che considera ogni ambiente multi-agente come un gioco, sia che l'interazione sia cooperativa o competitiva. I giochi possono essere classificati in base a due dimensioni principali:

- **Condizioni di Scelta:**

- Giochi con informazione perfetta: gli stati del gioco sono completamente noti agli agenti.
- Giochi con informazione imperfetta: gli stati del gioco non sono completamente noti.

- **Effetti della Scelta:**

- Giochi deterministici: gli stati sono determinati solo dalle azioni degli agenti.
- Giochi stocastici: gli stati sono influenzati anche da fattori esterni.

11.2 Definizione di un Gioco

Formalmente, un gioco può essere definito come un problema di ricerca con i seguenti componenti:

- s_0 : stato iniziale.

- $GIOCATORE(s)$: definisce il giocatore a cui tocca fare una mossa nello stato s .
- $AZIONI(s)$: restituisce l'insieme delle mosse lecite nello stato s .
- $RISULTATO(s, a)$: il modello di transizione che definisce il risultato della mossa a .
- $TEST_TERMINAZIONE(s)$: restituisce *vero* se la partita è finita, *falso* altrimenti.
- $UTILITÀ(s, p)$: funzione di utilità (o funzione di payoff) che assegna un valore numerico finale per il giocatore p nello stato terminale s .

Come esempio, consideriamo il gioco del tris. Qui i due giocatori MAX e MIN si alternano piazzando rispettivamente una X o una O sulla griglia vuota. Il gioco termina quando uno dei giocatori ottiene tre simboli consecutivi o quando la griglia è piena (in caso di pareggio). In primis, l'albero di gioco rappresenta tutti i possibili stati raggiungibili durante la partita.

Lo stato iniziale s_0 sarà composto da una griglia vuota. Le azioni $AZIONI(s)$ corrispondono a piazzare una X o una O in una casella vuota. Il $RISULTATO(s, a)$ consiste nel piazzare una X o una O sulla griglia, il che porta alla modifica della configurazione della griglia. Il test di terminazione $TEST_TERMINAZIONE(s)$ darà esito affermativo quando uno dei due giocatori ottiene tre simboli consecutivi o quando tutte le caselle sono occupate (in caso di pareggio). Infine, la funzione di utilità $UTILITÀ(s, p)$ sarà assegnata in base al risultato, ovvero:

- +1: Vittoria di MAX.
- 0: Pareggio.
- -1: Vittoria di MIN.

11.3 Strategia Ottima e Minimax

In un problema di ricerca con avversari, l'obiettivo di un giocatore è trovare una strategia ottima, che **massimizzi il risultato nel caso peggiore**. Questo perché viene imposta l'assunzione che l'avversario giochi in modo ottimale, ovvero si giochi contro un giocatore infallibile, il quale cercherà sempre di minimizzare il risultato del primo giocatore. Pertanto, il giocatore deve pianificare le sue mosse in modo da garantire il miglior risultato possibile, anche nel caso in cui l'avversario scelga sempre la strategia più sfavorevole. Questo approccio, noto come **minimax**, analizza l'intero albero di gioco, esplorando tutte le mosse possibili e le contromosse dell'avversario, fino a trovare la sequenza di azioni che massimizza il punteggio nel caso peggiore.

A proposito di strategie ottime, vale la pena approfondire questo discorso nel contesto del celeberrimo **dilemma del prigioniero**, un classico esempio della teoria dei giochi. In questo scenario, due prigionieri sono accusati di un crimine e posti in celle separate, senza possibilità di comunicare tra loro. Ogni prigioniero può scegliere se **collaborare** con l'altro rimanendo in silenzio, oppure **tradire** confessando il crimine e accusando l'altro. Le possibili pene dipendono dalle scelte di entrambi:

- Se entrambi i prigionieri restano in silenzio (**collaborano**), riceveranno una pena ridotta (ad esempio, 1 anno di prigione ciascuno).
- Se uno tradisce e l'altro collabora, il prigioniero che ha tradito verrà rilasciato (0 anni), mentre l'altro riceverà la pena massima (5 anni).
- Se entrambi tradiscono, riceveranno una pena moderata (ad esempio, 3 anni ciascuno).

Il **dilemma** sta nel fatto che, per ciascun prigioniero, tradire sembra essere la strategia ottima a breve termine, poiché offre la possibilità di evitare completamente la prigione, indipendentemente dalla scelta dell'altro. Tuttavia, se entrambi i prigionieri tradiscono, il risultato complessivo è peggiore rispetto a quello che si otterrebbe con la collaborazione reciproca.

Questo esempio illustra chiaramente come una strategia ottima in un contesto competitivo non sempre conduca al miglior risultato complessivo. Nel dilemma del prigioniero, la strategia di equilibrio di Nash prevede che entrambi i prigionieri tradiscano, poiché ciascuno cerca di minimizzare la propria perdita nel caso peggiore. Tuttavia, la **cooperazione** sarebbe la strategia che massimizza il risultato complessivo, portando a un'esplorazione più approfondita delle strategie ottime in giochi non cooperativi.

Per tornare alla risoluzione di un problema di ricerca con avversari, introduciamo adesso gli elementi alla base dell'algoritmo minimax. Il **valore minimax** di un nodo rappresenta l'utilità di quel nodo, assumendo che entrambi i giocatori giochino in modo ottimale. Può essere formalizzato come:

$$\begin{aligned}
 MINIMAX(n) &= UTILITÀ(s) \quad \text{se lo stato è terminale;} \\
 MINIMAX(n) &= \max_{a \in AZIONI(s)} MINIMAX(RISULTATO(s, a)) \quad \text{se è il turno di MAX;} \\
 MINIMAX(n) &= \min_{a \in AZIONI(s)} MINIMAX(RISULTATO(s, a)) \quad \text{se è il turno di MIN.}
 \end{aligned}$$

Fatte queste premesse, lo pseudocodice dell'algoritmo minimax è il seguente:

```
function DECISIONE-MINIMAX(s)
  return argmax_aAZIONI(s) VALORE-MIN(RISULTATO(s, a))
```

```
function VALORE-MAX(s)
  if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
  v <- -infinite
  for each a in AZIONI(s) do
    v <- MAX(v, VALORE-MIN(RISULTATO(s, a)))
  return v
```

```
function VALORE-MIN(s)
  if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
  v <- +infinite
  for each a in AZIONI(s) do
    v <- MIN(v, VALORE-MAX(RISULTATO(s, a)))
  return v
```

Spieghiamo più nel dettaglio il funzionamento dell'algoritmo:

- **Funzione DECISIONE-MINIMAX(s):** Questa funzione viene chiamata dallo stato iniziale s del gioco. Il compito di questa funzione è determinare quale sia la migliore azione che MAX dovrebbe scegliere. Viene chiamata una funzione di supporto, **VALORE-MIN**, per valutare il valore delle possibili mosse di MAX. L'azione scelta è quella che massimizza il valore di MIN-IMAX, cioè la mossa che garantisce a MAX il miglior risultato possibile, assumendo che MIN giochi in modo ottimale.
- *argmax* restituisce l'azione che massimizza il valore ritornato da **VALORE-MIN**, cioè la migliore mossa per MAX.
- **Funzione VALORE-MAX(s):** Questa funzione viene chiamata quando è il turno di MAX. Il suo compito è esplorare tutte le mosse possibili che MAX può fare dallo stato attuale s , e per ciascuna mossa, chiamare **VALORE-MIN** per vedere come MIN risponderà. La funzione ricorsiva continua a esplorare l'albero di gioco fino a raggiungere uno stato terminale, cioè uno stato in cui il gioco è terminato. Se il gioco è terminato (determinato dal test di terminazione), la funzione ritorna il valore dell'utilità di quello stato, calcolata attraverso la funzione di utilità (ad esempio, +1 se MAX ha vinto, -1 se MIN ha vinto, o 0 in caso di pareggio). Se il gioco non è terminato, la funzione itera attraverso tutte le azioni possibili per MAX e chiama **VALORE-MIN** su ogni stato risultante dalla mossa di MAX. Il valore massimo (da qui il nome "VALORE-MAX") tra tutte le mosse possibili viene scelto come il valore di ritorno.
- **Funzione VALORE-MIN(s):** Questa funzione viene chiamata quando è il turno di MIN. Il suo funzionamento è simile a quello di **VALORE-MAX**, ma con la differenza che MIN cerca di minimizzare il valore ottenuto da MAX. Quindi, anziché restituire il massimo, questa funzione restituisce il minimo dei valori che si possono ottenere dopo ogni mossa di MIN. Se lo stato s è terminale, viene restituito il valore dell'utilità dello stato. Altrimenti, si itera su tutte le mosse possibili per MIN, chiamando **VALORE-MAX** per ogni stato risultante dalle mosse di MIN, e si restituisce il valore minimo tra quelli calcolati, poiché MIN cerca di minimizzare il risultato di MAX.

Consideriamo un esempio di tris in cui è il turno di MAX. La funzione **DECISIONE-MINIMAX** chiamerà **VALORE-MAX**, che inizierà a esplorare tutte le possibili mosse di MAX. Per ogni mossa, **VALORE-MIN** sarà chiamata per calcolare la risposta ottimale di MIN. Questo processo continua ricorsivamente fino a che non si raggiunge uno stato terminale (una vittoria, una sconfitta o un pareggio). A questo punto, le funzioni risalgono l'albero di gioco, restituendo i valori minimax fino alla radice, dove viene identificata la mossa ottima per MAX.

Da un punto di vista computazionale, l'algoritmo minimax ha le seguenti proprietà:

- **Completezza:** L'algoritmo è completo se l'albero di gioco è finito.
- **Ottimalità:** L'algoritmo è ottimo se l'avversario gioca in modo ottimo.
- **Complessità temporale:** La complessità temporale è $O(b^m)$, dove b è il fattore di ramificazione e m è la profondità massima dell'albero.
- **Complessità spaziale:** Richiede $O(bm)$ di memoria.

11.4 Potatura Alfa-Beta

Le prestazioni computazionali dell'algoritmo minimax non consentono la risoluzione di problemi reali - questo perché, molto spesso, l'albero di gioco è molto complesso (vedi esempio del tris), il che porta l'algoritmo ad essere impraticabile. La **potatura alfa-beta** è una tecnica che permette di ottimizzare l'algoritmo minimax riducendo il numero di nodi da esplorare. Consente di eliminare porzioni dell'albero che non influenzeranno la decisione finale, mantenendo comunque lo stesso risultato del minimax standard. La potatura viene implementata aggiornando due parametri, **alfa** e **beta**, che rappresentano i migliori valori che i giocatori MAX e MIN possono garantire.

Lo pseudocodice della potatura alfa-beta è il seguente:

```
function RICERCA-ALFA-BETA(s)
  v <- VALORE-MAX(s, -, +)
  return l'azione in AZIONI(s) con valore v

function VALORE-MAX(s, alfa, beta)
  if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
  v <- -infinite
  for each a in AZIONI(s) do
    v <- MAX(v, VALORE-MIN(RISULTATO(s, a), alfa, beta))
    if v >= beta then return v
    alfa <- MAX(alfa, v)
  return v

function VALORE-MIN(s, alfa, beta)
  if TEST-TERMINAZIONE(s) then return UTILITÀ(s)
  v <- +infinite
  for each a in AZIONI(s) do
    v <- MIN(v, VALORE-MAX(RISULTATO(s, a), alfa, beta))
    if v <= alfa then return v
    beta <- MIN(beta, v)
  return v
```

La potatura alfa-beta consente di esplorare solo $O(b^{m/2})$ nodi, dimezzando la profondità esplorata rispetto all'algoritmo minimax standard. Analizziamo più nel dettaglio il suo funzionamento:

- **Funzione RICERCA-ALFA-BETA(s):** Questa funzione inizia l'esplorazione dell'albero di gioco dalla radice s , utilizzando la funzione **VALORE-MAX** per iniziare il turno di MAX. I parametri iniziali di α e β sono impostati rispettivamente a $-\infty$ e $+\infty$, poiché all'inizio non c'è ancora nessuna informazione sui migliori valori possibili per MAX o MIN.
- **Funzione VALORE-MAX(s, α , β):** Questa funzione viene chiamata quando è il turno di MAX. L'obiettivo di MAX è massimizzare il valore, quindi esplora tutte le azioni possibili e chiama **VALORE-MIN** per vedere come risponderà MIN. Ogni volta che viene trovata una mossa con un valore superiore al valore massimo attuale, **α** viene aggiornato con questo nuovo valore:

- Se si scopre che il valore corrente di MAX è maggiore o uguale a **beta** (il miglior valore garantito a MIN), non ha senso esplorare ulteriori mosse per MAX, poiché MIN non permetterebbe mai che MAX ottenga un valore maggiore di beta. Questo processo è noto come **potatura**, e l'esplorazione del resto delle azioni viene interrotta.
- Alla fine, viene restituito il valore massimo trovato tra tutte le mosse, oppure il valore che ha causato la potatura.
- **Funzione VALORE-MIN(s, alfa, beta):** Questa funzione viene chiamata quando è il turno di MIN. L'obiettivo di MIN è minimizzare il valore, quindi esplora tutte le azioni possibili e chiama **VALORE-MAX** per vedere come risponderà MAX. Ogni volta che viene trovata una mossa con un valore inferiore al valore minimo attuale, **beta** viene aggiornato con questo nuovo valore.
 - Se il valore corrente di MIN è minore o uguale a **alfa** (il miglior valore garantito a MAX), non ha senso esplorare ulteriori mosse per MIN, poiché MAX non permetterebbe mai che MIN ottenga un valore inferiore a alfa. Anche in questo caso, si effettua una potatura.
 - Alla fine, viene restituito il valore minimo trovato tra tutte le mosse, oppure il valore che ha causato la potatura.

La potatura funziona grazie al confronto continuo tra i valori di **alfa** e **beta**, aggiornati durante l'esplorazione dei rami dell'albero di gioco:

- **Alfa** viene aggiornato nel turno di MAX e rappresenta il miglior valore che MAX può garantire.
- **Beta** viene aggiornato nel turno di MIN e rappresenta il miglior valore che MIN può garantire.

Ogni volta che si scopre che il valore di un ramo non migliorerà oltre i limiti di alfa o beta, quel ramo può essere eliminato, poiché non influenzerà il risultato finale. Questo consente di ridurre significativamente il numero di nodi da esplorare.

Consideriamo un esempio di tris con potatura alfa-beta. Se MIN sa che una mossa garantisce un valore massimo che MAX non accetterà mai (ad esempio, MIN ha già trovato una mossa che può limitare MAX a un certo valore), non ha senso esplorare ulteriori mosse. Allo stesso modo, se MAX sa che una mossa garantisce un valore che MIN non potrà mai migliorare, potrà smettere di considerare quel ramo dell'albero.

Ordinamento Dinamico delle Mosse L'ordinamento dinamico delle mosse è una tecnica che migliora ulteriormente l'efficienza della **potatura alfa-beta**. L'idea di base è ordinare le mosse più promettenti all'inizio del processo di esplorazione. Se le migliori mosse vengono esaminate per prime, si possono effettuare più potature precoci, riducendo così il numero di nodi da esplorare.

L'ordinamento dinamico delle mosse è efficace perché:

- Aumenta la probabilità di aggiornare i valori di **alfa** e **beta** rapidamente, consentendo potature precoci nei rami meno promettenti.

- Migliora la velocità complessiva dell'algoritmo, soprattutto in giochi complessi come gli scacchi, dove il numero di possibili mosse è elevato.
- Riduce il tempo di ricerca necessario per identificare la mossa ottima, specialmente nei livelli superiori dell'albero di gioco, dove le decisioni hanno maggiore impatto sul risultato finale.

Una strategia comune per implementare l'ordinamento dinamico delle mosse consiste nell'utilizzare una funzione di valutazione euristica che approssima la qualità di una mossa prima che l'intero albero venga esplorato. Le mosse che risultano migliori in base all'euristica vengono esplorate per prime. Un esempio di questa tecnica è l'utilizzo dell'**euristica del valore del materiale** negli scacchi, che ordina le mosse in base alla capacità di catturare pezzi dell'avversario o di avanzare in una posizione vantaggiosa.

11.5 Decisioni Perfette in Tempo Reale

Il concetto di **decisioni perfette in tempo reale** si riferisce alla capacità di un algoritmo di prendere la miglior decisione possibile entro un limite di tempo prestabilito. In giochi complessi, è spesso impraticabile esplorare l'intero albero di gioco a causa dei limiti di tempo imposti, come accade negli scacchi competitivi o nei videogiochi in tempo reale.

Esistono diverse tecniche per garantire che un algoritmo di ricerca con avversari possa prendere decisioni in tempo reale:

- **Approfondimento Iterativo:** L'algoritmo esplora l'albero di gioco a profondità crescente, fornendo una mossa valida anche se non è stato completato l'intero processo di ricerca. Se viene raggiunto il limite di tempo, l'algoritmo restituisce la mossa migliore trovata fino a quel momento.
- **Euristiche Veloci:** Le funzioni di valutazione euristica vengono utilizzate per stimare il valore di stati non terminali in modo rapido, evitando l'esplorazione completa dei rami dell'albero.
- **Timeout:** Quando il tempo disponibile scade, l'algoritmo ritorna la mossa che ha esplorato più a fondo fino a quel momento, anche se non ha raggiunto la profondità massima.

11.6 Ulteriori Miglioramenti della Potatura Alfa-Beta

Oltre all'ordinamento dinamico delle mosse, esistono ulteriori miglioramenti alla potatura alfa-beta che permettono di ottimizzare ulteriormente l'esplorazione degli alberi di gioco:

- L'approfondimento iterativo combina il concetto di ricerca in profondità limitata con la potatura alfa-beta. In questo approccio, l'albero di gioco viene esplorato fino a una certa profondità, e poi questa profondità viene incrementata in successive iterazioni. Questo garantisce che, anche se l'algoritmo viene interrotto prima della fine, ci sarà comunque una mossa valida da eseguire, basata sull'esplorazione più profonda raggiunta fino a quel punto.

- Una tecnica chiamata **finestra aspettativa** può essere applicata per migliorare ulteriormente la potatura alfa-beta. Invece di esplorare l'intero intervallo di valori di alfa e beta, l'algoritmo parte con una finestra ristretta intorno al valore atteso della mossa migliore. Se il valore della mossa rientra nella finestra, non è necessario esplorare ulteriori rami. Se invece il valore esce fuori dalla finestra, la finestra viene estesa e l'esplorazione continua.
- Le **tabelle di trasposizione** sono strutture dati che memorizzano i risultati di stati già esplorati in precedenza. Questo è utile nei giochi dove uno stesso stato può essere raggiunto attraverso diverse sequenze di mosse. Utilizzando una tabella di trasposizione, l'algoritmo evita di esplorare nuovamente stati già visitati, migliorando l'efficienza complessiva.
- La **potatura scout** o **negascout** è una variante della potatura alfa-beta che cerca di ridurre ulteriormente il numero di nodi esplorati attraverso una tecnica più aggressiva. Negascout esplora prima un ramo del sottoproblema con una finestra ristretta, e solo se trova una mossa migliore, esplora gli altri rami. Questo riduce la quantità di rami da esplorare rispetto alla potatura alfa-beta standard.
- La **potatura multi-cut** è una tecnica che cerca di interrompere l'esplorazione di sottorami all'interno di un livello di profondità se si scopre che più di un certo numero di rami stanno portando a valori non promettenti. Questa tecnica consente di eliminare grandi sezioni dell'albero che non influenzeranno il risultato finale.

11.7 Conclusioni

La ricerca con avversari rappresenta un campo cruciale dell'intelligenza artificiale, in particolare per la risoluzione di problemi competitivi in cui gli agenti devono prendere decisioni strategiche tenendo conto delle azioni degli avversari. Questo ambito trova applicazione in numerosi settori, come i giochi a turni (ad esempio, scacchi e Go), la pianificazione militare, i sistemi di negoziazione automatica, e la sicurezza informatica. Algoritmi come il **minimax** e la **potatura alfa-beta** consentono di affrontare questi problemi in modo efficiente, migliorando le capacità decisionali anche in situazioni complesse e dinamiche. I continui miglioramenti, come l'ordinamento dinamico delle mosse, l'approfondimento iterativo e l'uso di tecniche di potatura avanzate, garantiscono che le strategie ottimali siano raggiunte con un minor costo computazionale, rendendo questi algoritmi fondamentali per lo sviluppo di intelligenze artificiali avanzate in ambienti multi-agente.

11.8 Quiz Time

Domanda 1: Qual è l'obiettivo principale dell'algoritmo minimax in un problema di ricerca con avversari?

- A) Massimizzare il punteggio medio del giocatore MAX.
- B) Massimizzare il punteggio di MAX assumendo che MIN giochi in modo ottimale.
- C) Minimizzare il numero di mosse di MIN.
- D) Assicurare che MAX vinca sempre.

Risposta corretta: B) Massimizzare il punteggio di MAX assumendo che MIN giochi in modo ottimale.

Domanda 2: In che modo la potatura alfa-beta migliora l'efficienza dell'algoritmo minimax?

- A) Calcolando il valore massimo e minimo simultaneamente.
- B) Esplorando solo metà dell'albero di gioco.
- C) Evitando di esplorare rami dell'albero che non influenzeranno la decisione finale.
- D) Utilizzando una funzione di valutazione euristica.

Risposta corretta: C) Evitando di esplorare rami dell'albero che non influenzeranno la decisione finale.

Domanda 3: Qual è il vantaggio principale dell'ordinamento dinamico delle mosse?

- A) Migliora la qualità delle mosse scelte.
- B) Riduce il numero di mosse disponibili per ciascun giocatore.
- C) Aumenta la probabilità di potature precoci, riducendo il numero di nodi da esplorare.
- D) Permette di esplorare l'intero albero di gioco.

Risposta corretta: C) Aumenta la probabilità di potature precoci, riducendo il numero di nodi da esplorare.

Domanda 4: Nel contesto della ricerca con avversari, cosa rappresenta il valore **alfa** nella potatura alfa-beta?

- A) Il miglior valore che MIN può garantire finora.
- B) Il peggior valore che MIN può ottenere.
- C) Il miglior valore che MAX può garantire finora.
- D) La differenza tra i valori di MAX e MIN.

Risposta corretta: C) Il miglior valore che MAX può garantire finora.

Domanda 5: Qual è il principale obiettivo della funzione di utilità in un gioco competitivo?

- A) Determinare il numero di mosse rimanenti nel gioco.
- B) Assegnare un valore numerico a uno stato terminale del gioco per determinare chi ha vinto.
- C) Ridurre il tempo di esplorazione dell'albero di gioco.

D) Ordinare le mosse in modo dinamico.

Risposta corretta: B) Assegnare un valore numerico a uno stato terminale del gioco per determinare chi ha vinto.

Domanda 6: In che tipo di giochi è comunemente applicato l'algoritmo minimax?

- A) Giochi di cooperazione.
- B) Giochi di strategia con avversari, come scacchi e tris.
- C) Giochi di fortuna con lanci di dadi.
- D) Giochi stocastici.

Risposta corretta: B) Giochi di strategia con avversari, come scacchi e tris.

Domanda 7: Che cos'è una **decisione perfetta in tempo reale**?

- A) La scelta della migliore mossa possibile entro un tempo limitato.
- B) La capacità di battere sempre l'avversario.
- C) La decisione di esplorare solo i rami principali dell'albero.
- D) La capacità di prevedere ogni mossa dell'avversario.

Risposta corretta: A) La scelta della migliore mossa possibile entro un tempo limitato.

Domanda 8: Quando viene applicata la **potatura alfa** nell'algoritmo alfa-beta?

- A) Quando il valore minimo trovato per MIN è inferiore a beta.
- B) Quando il valore di MAX supera o uguaglia beta.
- C) Quando il valore di MIN supera alfa.
- D) Quando il valore di MAX è minore di alfa.

Risposta corretta: B) Quando il valore di MAX supera o uguaglia beta.

Domanda 9: Qual è l'obiettivo dell'**approfondimento iterativo** in un contesto di decisioni in tempo reale?

- A) Ridurre il numero di mosse esplorate.
- B) Garantire che la mossa migliore sia trovata entro un limite di tempo predefinito.

- C) Esplorare solo una parte limitata dell'albero di gioco.
- D) Migliorare l'efficienza delle mosse di MIN.

Risposta corretta: B) Garantire che la mossa migliore sia trovata entro un limite di tempo predefinito.

Domanda 10: Quale delle seguenti tecniche memorizza stati precedentemente esplorati per evitare di analizzarli nuovamente?

- A) Finestra aspettativa.
- B) Tabella di trasposizione.
- C) Ordinamento dinamico delle mosse.
- D) Potatura multi-cut.

Risposta corretta: B) Tabella di trasposizione.

11.9 Esercizi sulla Progettazione di un Algoritmo di Ricerca con Avversari

Di seguito vengono proposti diversi esercizi che, a titolo di esempio, mostrano come progettare un algoritmo di ricerca con avversari per la risoluzione di problemi reali.

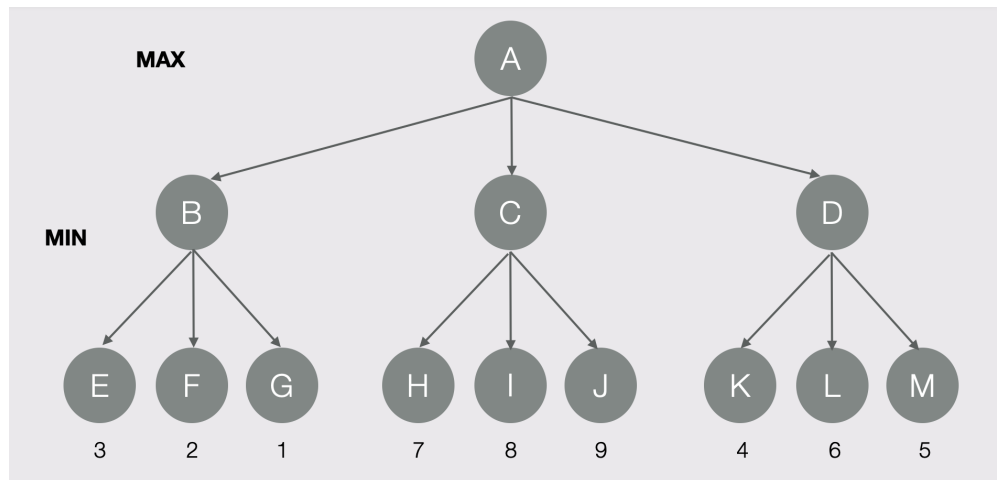


Figure 1: Esercizio 1: Albero di gioco.

Domanda:

Considerando l'albero di gioco rappresentato in Figura 1:

- Calcolare il valore di gioco minimax dei nodi A, B, C e D usando l'algoritmo minimax standard.
- Quale mossa verrà selezionata dal giocatore MAX usando minimax?
- Elencare i nodi (stati terminali o nodi interni) che l'algoritmo alfa-beta elimina.

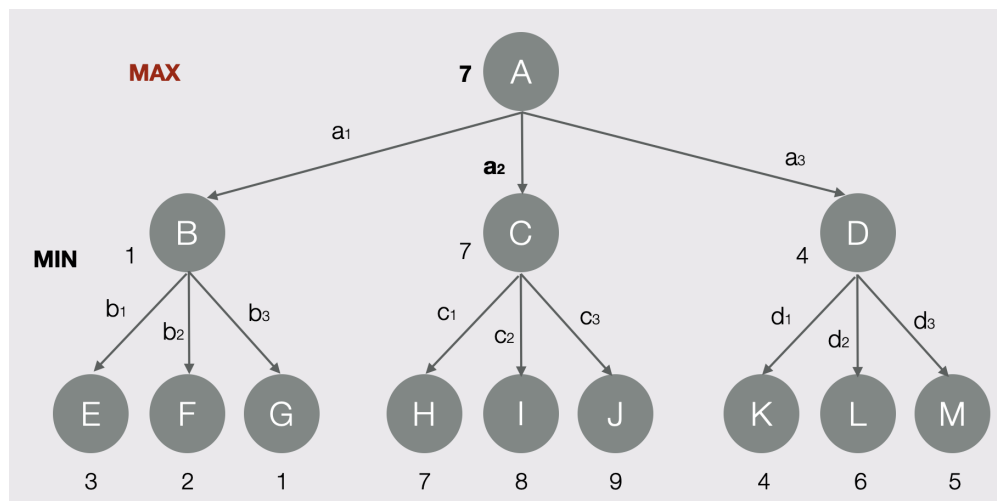


Figure 2: Esercizio 1: Soluzione.

Soluzione suggerita

Figura 2 presenta la soluzione al primo punto dell'esercizio. Ricordiamo che MIN selezionerà sempre il nodo con utilità più bassa. Nelle varie iterazioni dell'algoritmo, quindi, i nodi B, C e D, che saranno di competenza di MIN, avranno un valore minimax rappresentato dal minimo valore minimax degli stati terminali raggiungibili da essi. Per quanto riguarda MAX, questo invece sceglierà il nodo con il valore di utilità maggiore. Pertanto, il valore minimax del nodo A sarà pari a 7, ovvero il massimo valore minimax degli stati raggiungibili da esso.

In risposta al secondo punto dell'esercizio, possiamo quindi determinare che MAX selezionerà la mossa a_2 , che lo porterà nello stato C.

La potatura alfa-beta, in questo caso, escluderebbe i nodi L ed M. La ragione è la seguente: MIN esplorerà man mano i nodi per assegnare i valori minimax. Fino all'assegnamento del valore minimax per il nodo D, l'esplorazione non può terminare. Invece, raggiunto D, l'utilità del primo nodo esplorato (K) è 4. MIN non sceglierà mai un valore più alto di una scelta che ha già a disposizione - ossia il nodo C. I nodi eliminati saranno perciò L ed M. Lo sviluppo completo dell'esercizio è riportato nelle slide aggiuntive, nella cartella relativa al materiale di tutorato, della piattaforma e-learning.

Scenario: Sistema di Negoziazione Automatica. Immagina di progettare un sistema di negoziazione automatica per un'azienda che compra materie prime (Napoli) da un fornitore (Fornitore). La negoziazione riguarda il prezzo per un determinato quantitativo di prodotto. Napoli vuole ottenere il prezzo più basso possibile, mentre il Fornitore cerca di massimizzare il prezzo di

vendita. Entrambi possono fare offerte e contro-offerte, cercando di raggiungere un compromesso accettabile entro un certo numero di mosse (turni).

Domanda: Progetta un algoritmo di negoziazione automatica per Napoli basato su minimax con potatura alfa-beta, che permetta di trovare la migliore strategia negoziale. L'obiettivo di Napoli è minimizzare il costo, mentre il Fornitore cerca di massimizzare il profitto. Considera che il numero massimo di turni per raggiungere un accordo è 5, e che ciascun giocatore conosce solo le proprie preferenze (cioè il prezzo ideale per ogni parte non è noto all'avversario).

Soluzione Suggesta:

- Modella il problema come un albero di decisioni, in cui ciascun nodo rappresenta uno stato della negoziazione. Ogni azione corrisponde a un'offerta o contro-offerta di uno dei due partecipanti.
- Definisci la funzione **UTILITÀ(s)**: per Napoli, l'utilità è la differenza tra il prezzo ideale e l'offerta attuale (più bassa è, meglio è). Per il Fornitore, l'utilità è la differenza tra l'offerta attuale e il prezzo massimo che vorrebbe ottenere (più alta è, meglio è).
- Implementa la funzione **VALORE-MAX** per simulare il turno del Fornitore, che cerca di massimizzare il proprio profitto. Similmente, implementa la funzione **VALORE-MIN** per Napoli, che cerca di minimizzare il costo.
- Applica la potatura **alfa-beta** per evitare di esplorare combinazioni di offerte non promettenti e migliorare l'efficienza della negoziazione.
- Considera una soglia per la negoziazione: se nessun accordo viene raggiunto entro 5 turni, entrambe le parti devono abbandonare il negoziato, con un payoff negativo per entrambe.

Scenario: Sicurezza Informatica. Una compagnia deve proteggere la sua rete informatica da un attacco hacker. L'azienda ha a disposizione risorse limitate (ad esempio, firewall, monitoraggio del traffico, e crittografia dei dati) per difendersi, mentre l'hacker cerca di sfruttare vulnerabilità nella rete. L'hacker può provare a violare la rete attraverso diverse tecniche (ad esempio, phishing, attacco DDoS, o exploit di software non aggiornato), e la compagnia deve decidere quali difese implementare in ogni turno. Ogni azione ha un costo in termini di risorse.

Domanda: Progetta un algoritmo che permetta all'azienda di pianificare le sue difese utilizzando la ricerca con avversari. L'azienda (MAX) deve massimizzare la protezione della rete, mentre l'hacker (MIN) deve massimizzare i danni. Considera che le risorse per la difesa sono limitate e che ogni tecnica di difesa può coprire una certa percentuale della rete.

Soluzione Suggesta:

- Definisci ogni stato come una configurazione delle difese attualmente in uso. Ogni azione rappresenta una decisione su come allocare le risorse disponibili per implementare una difesa (ad esempio, aggiungere un firewall o aggiornare un software vulnerabile).
- L'obiettivo del **VALORE-MAX** è massimizzare la sicurezza complessiva della rete, misurata come una percentuale della rete protetta. L'obiettivo del **VALORE-MIN** è minimizzare la percentuale di rete protetta (cioè massimizzare i danni).

- La funzione di utilità **UTILITÀ(s)** per MAX potrebbe essere la percentuale della rete protetta meno il costo delle risorse impiegate, mentre per MIN potrebbe essere la percentuale della rete non protetta moltiplicata per il valore potenziale del danno.
- Applica l'algoritmo di potatura **alfa-beta** per ottimizzare la ricerca delle strategie difensive, eliminando rami dell'albero che non migliorano significativamente la protezione della rete.
- Considera che ogni attacco scoperto riduce le risorse di MAX per i turni successivi, aggiungendo un elemento di pianificazione strategica.

Scenario: Pianificazione della Produzione e Competizione. Due aziende, Alfa e Beta, competono nella produzione e vendita di un prodotto tecnologico. Alfa può scegliere quanto investire in R&D per migliorare il proprio prodotto, e Beta può scegliere quante risorse allocare per ridurre i costi di produzione. Ogni azienda vuole massimizzare il proprio profitto: Alfa massimizza il valore del proprio prodotto tramite l'innovazione, mentre Beta cerca di ottenere un vantaggio riducendo i costi. Le decisioni prese in ciascuna fase influiscono sul mercato e, di conseguenza, sui profitti di entrambe le aziende.

Domanda: Progetta un algoritmo di ricerca con avversari che aiuti l'azienda Alfa a pianificare il proprio investimento in R&D, sapendo che Beta cercherà di massimizzare i propri profitti riducendo i costi. Le due aziende si influenzano a vicenda, quindi ogni decisione di Alfa impatta i profitti di Beta e viceversa.

Soluzione Suggesta:

- Modella il problema come un albero di decisioni, dove ogni stato rappresenta una configurazione delle risorse allocate da Alfa per l'innovazione e da Beta per ridurre i costi.
- Definisci una funzione di utilità **UTILITÀ(s)** per Alfa basata sull'incremento di valore del prodotto e sui costi sostenuti per l'R&D, e una per Beta che rifletta il risparmio sui costi di produzione e l'impatto sulla qualità del prodotto.
- Implementa le funzioni **VALORE-MAX** per Alfa e **VALORE-MIN** per Beta, simulando la competizione tra le due aziende. Alfa cerca di massimizzare il valore del proprio prodotto, mentre Beta minimizza i costi di produzione.
- Utilizza la potatura **alfa-beta** per eliminare rami che non portano a vantaggi significativi per Alfa o Beta, accelerando la ricerca della migliore strategia competitiva.

Scenario: Difesa Militare. In un contesto militare, una nazione deve pianificare la difesa delle proprie postazioni strategiche. Il nemico può attaccare diverse postazioni con mezzi differenti (artiglieria, attacchi aerei, etc.). La nazione ha a disposizione risorse limitate (es. difese aeree, unità di terra, ecc.) che devono essere allocate in modo ottimale per difendere le postazioni più vulnerabili.

Domanda: Progetta un algoritmo di ricerca con avversari che aiuti la nazione a distribuire le proprie risorse di difesa per minimizzare i danni degli attacchi nemici. Considera che il nemico

cercherà di massimizzare i danni, e che entrambi i giocatori conoscono le potenziali vulnerabilità delle postazioni.

Soluzione Suggestita:

- Modella il problema come un albero di decisioni, in cui ciascun stato rappresenta una configurazione delle difese assegnate alle postazioni. Ogni azione del nemico è un tentativo di attacco a una specifica postazione.
- Definisci una funzione di utilità **UTILITÀ(s)**: per la nazione, l'utilità è la somma delle postazioni difese con successo meno il costo delle risorse allocate. Per il nemico, l'utilità è la somma dei danni inflitti.
- Implementa le funzioni **VALORE-MAX** per la nazione, che cerca di minimizzare i danni, e **VALORE-MIN** per il nemico, che cerca di massimizzarli.
- Applica la potatura **alfa-beta** per ottimizzare la distribuzione delle risorse di difesa, esplorando solo le configurazioni più promettenti.
- Considera anche una variabile di tempo: se il nemico attacca troppo rapidamente, le difese potrebbero essere meno efficaci, aggiungendo una componente dinamica al problema.

Scenario: Pianificazione di Risorse Energetiche tra Nazioni. Consideriamo tre nazioni (A, B e C) che competono per ottenere accesso a risorse energetiche limitate (ad esempio, petrolio, gas naturale, energia rinnovabile). Ogni nazione ha obiettivi diversi: massimizzare la propria produzione energetica, minimizzare i costi di acquisizione delle risorse, e mantenere la stabilità interna (evitando crisi energetiche). Tuttavia, ogni nazione può interferire con le strategie delle altre, limitando il loro accesso alle risorse e imponendo barriere economiche o politiche.

Ogni nazione deve decidere, in ciascun turno, come allocare le proprie risorse diplomatiche ed economiche per ottenere una quota delle risorse disponibili, mentre cerca di evitare che le altre nazioni ottengano vantaggi eccessivi. Il gioco termina dopo un numero predefinito di turni o quando le risorse sono esaurite.

Domanda: Progetta un algoritmo di ricerca multi-player basato su minimax che permetta a ciascuna nazione di pianificare la propria strategia. L'obiettivo è trovare una strategia ottimale per la nazione A, sapendo che B e C cercheranno di massimizzare i loro risultati. Ogni nazione ha risorse limitate per negoziare e competere.

Soluzione Suggestita:

- Modella ogni stato del gioco come una configurazione delle risorse energetiche disponibili e della percentuale già acquisita da ciascuna nazione.
- Le azioni possibili includono negoziazioni, barriere economiche e l'acquisto di quote di risorse.
- Definisci la funzione **UTILITÀ(s)** per ciascuna nazione:

- Per la nazione A, l'utilità potrebbe essere la quantità di risorse energetiche acquisite meno il costo diplomatico e politico delle azioni intraprese.
- Per le nazioni B e C, l'utilità è simile, ma ciascuna nazione cercherà anche di minimizzare il vantaggio ottenuto dalle altre nazioni.
- Implementa il minimax multi-player per simulare le mosse di ciascuna nazione e ottimizzare la strategia per la nazione A. Ogni funzione di valore (VALORE-A, VALORE-B, VALORE-C) dovrà tenere conto delle strategie degli altri giocatori.
- Usa un approccio con potatura **alfa-beta multi-player** per ridurre l'esplorazione dei rami dell'albero di gioco non promettenti, considerando che ciascuna nazione agisce razionalmente per massimizzare i propri guadagni.

Scenario: Allocazione di Risorse in una Startup con Investitori. In una startup innovativa, tre investitori principali (Investitore A, Investitore B e Investitore C) stanno competendo per ottenere il controllo di una parte significativa della proprietà della società. Ogni investitore può scegliere quanto investire in ogni turno, ma ci sono risorse limitate disponibili per finanziare la crescita della startup, e la proprietà totale non può superare il 100

Ogni investitore ha un obiettivo diverso: A vuole massimizzare il controllo della società per ottenere influenza sulle decisioni, B cerca di ottenere il massimo ritorno sull'investimento, e C vuole mantenere un equilibrio tra il rischio e la proprietà, evitando di sovra-investire. Tuttavia, ogni azione di un investitore influenza il comportamento degli altri, e la startup può decidere di accettare solo un certo livello di investimento per non perdere il controllo.

Domanda: Progetta un algoritmo di minimax multi-player che permetta agli investitori di pianificare le loro mosse per massimizzare i loro obiettivi, sapendo che le azioni degli altri investitori potrebbero compromettere i loro guadagni futuri. Ciascun investitore deve bilanciare il rischio di sovra-investire e ottenere una quota troppo bassa di proprietà.

Soluzione Suggerita:

- Modella lo stato del gioco come una configurazione delle quote di proprietà della startup già acquisite e del capitale rimanente per ciascun investitore.
- Le azioni possibili includono l'investimento diretto, la negoziazione di condizioni particolari o il ritiro temporaneo dal mercato per valutare le azioni degli altri investitori.
- Definisci una funzione di utilità **UTILITÀ(s)** per ciascun investitore:
 - L'utilità per A è la percentuale di controllo ottenuto nella startup meno il capitale investito.
 - L'utilità per B è il ritorno sull'investimento previsto, basato sul valore futuro della startup.
 - L'utilità per C è una combinazione di rischio ridotto e proprietà moderata, che evita eccessivi investimenti.

- Implementa l'algoritmo di minimax multi-player per calcolare le migliori strategie di investimento, tenendo conto delle mosse degli altri due investitori. Le funzioni **VALORE-A**, **VALORE-B** e **VALORE-C** devono essere implementate per simulare le decisioni di ciascun investitore, massimizzando i propri obiettivi.
- Applica la potatura **alfa-beta multi-player** per ridurre il numero di scenari da esplorare, eliminando quelli che non portano a miglioramenti significativi per gli investitori.
- Considera anche la possibilità di stabilire coalizioni temporanee tra gli investitori per ottenere un vantaggio sugli altri, creando così una dinamica più complessa tra gli agenti.

Scenario: Scelta del Prezzo tra Due Competitor. Due aziende (Azienda A e Azienda B) stanno lanciando un nuovo prodotto sul mercato e devono decidere quale prezzo impostare. Entrambe le aziende vogliono massimizzare i loro profitti, ma sanno che il prezzo influenzerà anche la quota di mercato che ciascuna otterrà. Ciascuna azienda ha due strategie:

- Prezzo Alto (H)
- Prezzo Basso (L)

I profitti delle aziende dipendono dalle scelte di prezzo di entrambe:

- Se entrambe scelgono un **Prezzo Alto (H, H)**, ottengono 10 unità di profitto ciascuna.
- Se entrambe scelgono un **Prezzo Basso (L, L)**, ottengono 5 unità di profitto ciascuna.
- Se una azienda sceglie **Prezzo Alto (H)** e l'altra **Prezzo Basso (L)**, quella che sceglie il Prezzo Basso ottiene 15 unità di profitto, mentre quella che sceglie il Prezzo Alto ottiene 2 unità.

Domanda: Calcola l'Equilibrio di Nash e individua gli Ottimi Paretiani per questo problema.

Soluzione Suggesta: Forma tabellare del gioco:

Azienda A / Azienda B	H	L
H	(10, 10)	(2, 15)
L	(15, 2)	(5, 5)

- **Equilibrio di Nash:** Il punto **(L, L)** è l'Equilibrio di Nash. Se una delle due aziende sceglie di ridurre il prezzo (L), l'altra non ha un incentivo a passare al Prezzo Alto (H), poiché i suoi profitti sarebbero minori.
- **Ottimi Paretiani:** L'esito **(H, H)** è un Ottimo Paretiano, poiché entrambe le aziende ottengono un profitto maggiore rispetto a qualsiasi altro risultato (10 unità ciascuna), ma non è un Equilibrio di Nash, poiché una delle due aziende ha un incentivo a deviare e ridurre il prezzo per ottenere un profitto maggiore.

Scenario: Scelta del Turno di Lavoro tra Due Imprese. Due imprese, Impresa A e Impresa B, competono per offrire i loro servizi in una determinata area urbana. Entrambe devono decidere in quale turno offrire i loro servizi: al **mattino (M)** o al **pomeriggio (P)**. Entrambe le imprese vogliono massimizzare i loro ricavi, ma sanno che lavorare nello stesso turno riduce i guadagni a causa della concorrenza diretta.

I ricavi dipendono dalle scelte di turno delle imprese:

- Se entrambe scelgono di lavorare in turni diversi (M, P) o (P, M), ciascuna ottiene 12 unità di ricavo.
- Se entrambe lavorano nello stesso turno (M, M) o (P, P), ciascuna ottiene 8 unità di ricavo.

Domanda: Trova l'Equilibrio di Nash e gli Ottimi Paretiani per questo problema.

Soluzione Suggerita: Forma tabellare del gioco:

Impresa A / Impresa B	M	P
M	(8, 8)	(12, 12)
P	(12, 12)	(8, 8)

- **Equilibrio di Nash:** Entrambi i punti (M, P) e (P, M) sono Equilibri di Nash, poiché se una delle imprese decide di cambiare turno, i loro ricavi diminuiranno.
- **Ottimi Paretiani:** Gli esiti (M, P) e (P, M) sono anche Ottimi Paretiani, poiché entrambi massimizzano i ricavi per entrambe le imprese.

Scenario: Scelta di Pubblicità tra Due Ristoranti. Due ristoranti, Ristorante A e Ristorante B, competono per attirare clienti in una piccola città. Ciascun ristorante può decidere se investire in **pubblicità (P)** o non fare pubblicità (N). Fare pubblicità ha un costo, ma può aumentare la clientela. Se solo uno dei due ristoranti fa pubblicità, ottiene molti più clienti, mentre l'altro ristorante ne perde.

I guadagni dei ristoranti dipendono dalle loro decisioni:

- Se entrambi fanno **pubblicità (P, P)**, ciascuno ottiene 8 unità di guadagno, poiché condividono il mercato ma devono sostenere i costi della pubblicità.
- Se uno fa **pubblicità (P)** e l'altro no (N), quello che fa pubblicità ottiene 12 unità di guadagno, mentre l'altro ne ottiene 4.
- Se nessuno fa **pubblicità (N, N)**, ciascun ristorante ottiene 6 unità di guadagno, risparmiando i costi pubblicitari ma senza un vantaggio competitivo.

Domanda: Trova l'Equilibrio di Nash e gli Ottimi Paretiani per questo problema.

Soluzione Suggestita: Forma tabellare del gioco:

Ristorante A / Ristorante B	P	N
P	(8, 8)	(12, 4)
N	(4, 12)	(6, 6)

- **Equilibrio di Nash:** Il punto **(P, P)** è l'Equilibrio di Nash. Se un ristorante sceglie di non fare pubblicità (N), mentre l'altro fa pubblicità (P), subirà una perdita di clienti significativa.
- **Ottimi Paretiani:** L'esito **(N, N)** è un Ottimo Paretiano, poiché entrambi i ristoranti ottengono un guadagno di 6 unità senza dover sostenere i costi della pubblicità. Tuttavia, non è un Equilibrio di Nash, poiché se uno dei due ristoranti decide di fare pubblicità, può migliorare il proprio guadagno.