Università degli Studi di Salerno

Hello Wordle

Progetto di Fondamenti di Intelligenza Artificiale

Autore: Christian Esposito
Matricola: 0512116472

Data: 7 gennaio 2025

Repository Github

Indice

1 Introduzione

Wordle è un videogioco browser gratuito sviluppato nel 2021 durante il lockdown da **Josh Wardle** e acquistato successivamente dal New York Times. Il gioco consiste nell'indovinare una parola di cinque lettere in lingua inglese impiegando non più di sei tentativi per riuscirci. Dopo ogni tentativo, il gioco fornisce un feedback visivo per ciascuna lettera della parola in modo da guidare il giocatore verso la soluzione. La versione ufficiale è disponibile facendo una veloce ricerca online e prevede una soluzione al giorno comune a tutti i giocatori, scelta casualmente dal database interno del gioco.

2 Definizione del problema

2.1 Obiettivo

L'obiettivo del progetto è lo sviluppo di due agenti intelligenti in grado di giocare e vincere a Wordle, minimizzando il numero di tentativi necessari, e comparare i loro risultati. Per ottenere i risultati previsti, sono state implementate ed analizzate due diverse soluzioni, la prima basata su un **algoritmo di ricerca best-first greedy**, la seconda utilizza un **algoritmo di ricerca minimax**.

2.2 Specifica PEAS

La descrizione PEAS dell'ambiente operativo dell'agente è la seguente:

- **Performance**: la misura di prestazione dell'agente è la capacità di indovinare la soluzione impiegando il minor numero di tentativi possibili;
- **Environment**: l'ambiente in cui opera l'agente è il gioco di Wordle, bisogna quindi indovinare una parola di cinque lettere in massimo sei tentativi. Le regole dell'ambiente prevedono l'invio di un feedback all'agente per ogni parola inserita assegnando un colore alle lettere che formano la parola:
  - Verde: la lettera è corretta e inserita nella giusta posizione;
  - Giallo: la lettera è presente nella parola ma in un'altra posizione;
  - Grigio: la lettera non è presente nella parola.
- **Actuators**: l'agente opera sull'ambiente inserendo parole scelte dal dizionario e aggiornando quest'ultimo tramite l'eliminazione di parole in base ai feedback ricevuti;
- **Sensors**: L'agente percepisce l'ambiente tramite i feedback delle parole e il dizionario.

2.2.1 Caratteristiche dell'ambiente

L'ambiente è:

- **Parzialmente osservabile**: l'agente può solo osservare il feedback sulle parole da lui proposte ma non la soluzione;
- **Deterministico**: lo stato dei feedback è determinato unicamente dalle scelte dell'agente;
- **Sequenziale**: la scelta di una parola dipende dal feedback ricevuto per la precedente;
- **Statico**: durante l'operazione dell'agente la parola da indovinare rimane la stessa e non viene influenzata in alcun modo;
- **Discreto**: Le azioni e le percezioni sono finite e limitate. Ci sono un numero limitato di lettere, parole, e tipi di feedback;
- **Singolo agente**: Solo un agente (il solver) interagisce con l'ambiente.

2.3 Analisi del problema

Come anticipato in precedenza, per risolvere il problema sono state impiegate due diverse strategie di ricerca: ricerca best-first greedy e ricerca minimax. Entrambe le implementazioni adottate di questi due algoritmi cercano di arrivare alla soluzione partendo da un dizionario che viene ridotto gradualmente in base ai feedback ricevuti. Il dizionario utilizzato per questo problema è ottenuto dall'unione dei due dizionari usati dalla versione ufficiale di Wordle: il primo contiene le parole che possono essere soluzioni, mentre il secondo, più ampio del primo, include le parole accettabili come tentativi ma che non verranno mai scelte come soluzione. Sono, in totale, 11271 le parole che formano il dizionario. Si denota, inoltre, che non è stata implementata una versione "pura" dell'algoritmo minimax ma una riadattata che verrà approfondita in seguito.

3 Soluzione del problema

Per permettere all'agente intelligente di risolvere la parola del giorno della versione ufficiale di Wordle si è ricorso all'uso di un'**API** esterna per poterla estrarre. La visualizzazione del lavoro dell'agente è stata invece realizzata tramite una demo webapp.

3.1 Tecnologie utilizzate

Il linguaggio di programmazione scelto per la soluzione del problema è **Python**. La webapp è stata realizzata tramite il framework open source **Flask**. Il dizionario è letto da un file di testo.

3.2 Funzione di feedback

La funzione per determinare il feedback di una parola è comune a entrambe le implementazioni. Questa inizializza dapprima il feedback per ogni lettera della variabile **guess** come *grey* e poi salva il numero di volte in cui ogni lettera della soluzione è presente al suo interno nella variabile **solution_char_counts**. Questo passaggio garantisce un feedback corretto anche per lettere presenti più volte. Viene poi effettuata la comparazione tra le lettere di **guess** e **solution** per assegnare correttamente il feedback *green* alle lettere inserite nella posizione corretta, aggiornando il conteggio dei caratteri. Viene infine effettuato un ulteriore controllo per assegnare correttamente i feedback *yellow*.

In una prima implementazione della funzione, il caso di lettere ripetute non era gestito, portando a casi in cui la lista delle parole candidate veniva restituita vuota. Questo non comprometteva però significativamente il tasso di successo dell'agente:

```
    def feedback_function(guess, solution):
    feedback = []
    for g, s in zip(guess, solution):
        if g == s:
            feedback.append("green")
        elif g in solution:
            feedback.append("yellow")
        else:
            feedback.append("grey")
    return feedback
```

Listing 1: Prima implementazione

La funzione definitiva utilizza la variabile *solution_char_counts* per contare il numero di occorrenze di ciascuna lettera presente nella soluzione. Se il numero di occorrenze di una lettera, salvato nella variabile è maggiore di zero, vengono assegnati i feedback *green* o *yellow* nei casi appropriati e il contatore associato a quella lettera viene decrementato di uno. Questo approccio consente di restituire feedback corretti nei casi in cui una stessa lettera appare più volte nella parola soluzione:

```
    def feedback_function(guess, solution):
    feedback = ["grey"] * len(guess)
    solution_char_counts = {s: solution.count(s) for s in solution}

    for i, (g, s) in enumerate(zip(guess, solution)):
        if g == s:
            feedback[i] = "green"
            solution_char_counts[g] -= 1

    for i, g in enumerate(guess):
        if feedback[i] == "grey" and g in solution_char_counts and
    solution_char_counts[g] > 0:
            feedback[i] = "yellow"
            solution_char_counts[g] -= 1

    return feedback
```

Listing 2: Versione definitiva

3.3 Soluzione con best-first Greedy

L'algoritmo best-first greedy è un tipo di algoritmo di ricerca best-first che utilizza un'euristica h(n) per guidare il processo decisionale. Questo valore rappresenta una stima del costo o della distanza verso l'obiettivo, e l'algoritmo seleziona sempre il nodo con il valore più promettente di h(n). L'approccio è implementato nella funzione *wordle_solver* utilizzando una *PriorityQueue*, dove le priorità sono assegnate esclusivamente in base al valore stimato di h(n).

3.3.1 Euristiche e priorità

L'euristica della soluzione è così definita:

- **h(n)**: è un punteggio assegnato a ogni parola, che indica la probabilità di essere la soluzione. Un punteggio più alto corrisponde a una maggiore probabilità.

L'euristica è definita nella funzione *heuristic* che conta la frequenza di tutte le lettere nelle parole che l'agente può usare per indovinare la soluzione, premiando quelle che presentano lettere più frequenti e diverse, mentre penalizza quelle che hanno lettere ripetute.

```
    def heuristic(word, candidates):
    letter_frequencies = {}
    for candidate in candidates:
        for letter in set(candidate):
            letter_frequencies[letter] = letter_frequencies.get(letter, 0) + 1

    unique_letters = len(set(word))
    score = sum(letter_frequencies.get(char, 0) for char in set(word))
    duplicate_penalty = len(word) - unique_letters

    return score + unique_letters - duplicate_penalty
```

Listing 3: Funzione heuristic

La differenza con un approccio greedy classico sta nel fatto che la soluzione punta a massimizzare, piuttosto che minimizzare, il valore dell'euristica.

3.3.2 Filtraggio parole

La funzione *filter_candidates* assiste la funzione euristica andando a ridurre la lista. In base ai feedback ricevuti dall'agente, elimina le parole che non possono essere la soluzione. La lista aggiornata è quella che viene passata alla funzione *heuristic* all'interno di *wordle_solver*. Una precedente implementazione della funzione presentava una logica di filtraggio molto più accurata e rappresentativa delle regole di Wordle, rieseguendo, di fatto, i controlli della funzione di feedback:

```
    def filter_candidates(candidates, guess, feedback):
    filtered = []
    for word in candidates:
        valid = True
        word_counts = {char: word.count(char) for char in set(word)}
        used_counts = {}

        for i, (g, f) in enumerate(zip(guess, feedback)):
            if f == "green":
                if word[i] != g:
                    valid = False
                    break
            elif f == "yellow":
                if g == word[i] or word_counts.get(g, 0) <= used_counts.get(g, 0):
                    valid = False
                    break
            elif f == "grey":
                if g in word and word_counts[g] > 0:
                    valid = False
                    break

            if f in ["green", "yellow"]:
                used_counts[g] = used_counts.get(g, 0) + 1
```

```
24
25          if valid:
26              filtered.append(word)
27
28      return filtered
```

Listing 4: Prima implementazione

Con questa prima versione della funzione, nonostante il costo computazionale maggiore rispetto alla versione finale, l'agente riusciva comunque a mantenere un alto tasso di successo. Quest'implementazione è stata scartata poiché, in alcune combinazioni specifiche di soluzioni e scelte iniziali casuali dell'agente, si verificavano casi in cui la lista di parole filtrate ritornava vuota. Ciò induceva l'agente a dichiarare la sconfitta prima ancora di utilizzare tutti i sei tentativi a disposizione.

La funzione definitiva ha una strategia più semplice e immediata: le parole della lista che non presentano un feedback identico all'ultima parola proposta vengono scartate.

```
1       def filter_candidates(candidates, guess, feedback):
2           return [
3           word for word in candidates
4           if feedback_function(guess, word) == feedback
5           ]
```

Listing 5: Versione definitiva

3.4 Soluzione con Minimax

Gli algoritmi minimax sono utilizzati per prendere decisioni ottimali in giochi competitivi a somma zero tra due giocatori. Nonostante Wordle non sia né un gioco competitivo, né un gioco a somma zero, si è comunque scelto di adottare un approccio minimax in seguito ad alcune considerazioni.
Si è immaginato che il secondo giocatore, l'avversario da battere per far sì che l'agente riesca nel suo intento, non sia altro che il gioco di Wordle stesso che, idealmente, sceglie la parola più complicata, quella che possa dare i feedback più "vaghi" ossia quelli che escludono meno parole possibili dall'albero di ricerca. E con il gioco a ricoprire il ruolo del giocatore "Min", l'agente veste i panni del giocatore "Max" che vuole massimizzare i suoi risultati scegliendo le parole più utili per il filtraggio del dizionario.

3.4.1 Potatura euristica

In un contesto "puro", l'algoritmo minimax dovrebbe considerare tutti i possibili cammini di un albero di gioco. Tuttavia, in questo caso, ciò significherebbe analizzare circa $2,10 \times 10^{14}$ scenari (assumendo che il dizionario non venga mai ridotto), un numero non di poco conto. Si è quindi deciso di ricorrere alla funzione *heuristic*, definita per la soluzione con algoritmo best-first greedy, per calcolare i punteggi delle parole e permettere all'algoritmo minimax di effettuare la potatura dell'albero.
Una prima versione dell'euristica, implementata direttamente nella funzione *minimax_algorithm*, calcolava i punteggi basandosi esclusivamente sui feedback:

```
1       for word in candidates:
2           score = 0
3           for feedback in feedback_log:
4               simulated_feedback = feedback_function(word, feedback["guess"])
5               if simulated_feedback == feedback["feedback"]:
6                   score += 1
```

Listing 6: Prima implementazione dell'euristica

Questa implementazione era più veloce ma meno accurata della versione definitva, influendo leggermente sul tasso di successo dell'agente, che rimaneva comunque alto.

3.4.2 Funzione minimax

Visto quanto detto in precedenza, non esistono delle funzioni min e max vere e proprie, ma l'implementazione di *minimax_algorithm* segue una logica di massimizzazione come farebbe una funzione max. Va

a valutare per ogni parola il proprio punteggio, salvando la parola associata al punteggio più alto nella variabile *best_guess*. Il valore massimo raggiunto viene memorizzato in **best_score**. Successivamente la funzione *wordle_minimax* utilizza *best_guess* come tentativo e, in base al feedback, verifica se il gioco è terminato o se è necessario proseguire potando ulteriormente l'albero di ricerca.

4 Analisi dei Risultati

Le prestazioni di entrambi gli algoritmi sono state misurate eseguendo 1000 casi test, con le soluzioni scelte casualmente dal dizionario, applicando due strategie diverse:

- Partenza casuale: l'algoritmo inizia con una parola scelta casualmente dal dizionario;
- Partenza con "AROSE": l'algoritmo utilizza sempre la parola "AROSE" come primo tentativo.

La scelta di "AROSE" è stata determinata analizzando la distribuzione delle lettere nelle parole del dizionario e la sua efficienza nel ridurre rapidamente il numero di possibili soluzioni. Infine, le prestazioni sono state rappresentate tramite grafici realizzati utilizzando **Pyplot** dalla libreria **Mathplotlib**.

4.0.1 Algoritmo best-first greedy con parola iniziale casuale

Questo primo approccio si mostra già altamente promettente e segna l'inizio di un trend positivo per quanto riguarda i tassi di successo. Come mostrato in **Figura 1**, l'approccio ha mostrato un tasso di successo pari al **91,10%** a fronte di un tempo medio di soli **2,04 secondi**. La distribuzione dei tentativi si presenta molto bene, con più della metà dei test vinti in 4 tentativi o meno (precisamente il 51,4%). Questo trend dimostra come l'approccio greedy, anche con una parola iniziale casuale, riesca a bilanciare in modo eccellente accuratezza e velocità di calcolo, rendendolo una base solida per ulteriori ottimizzazioni.

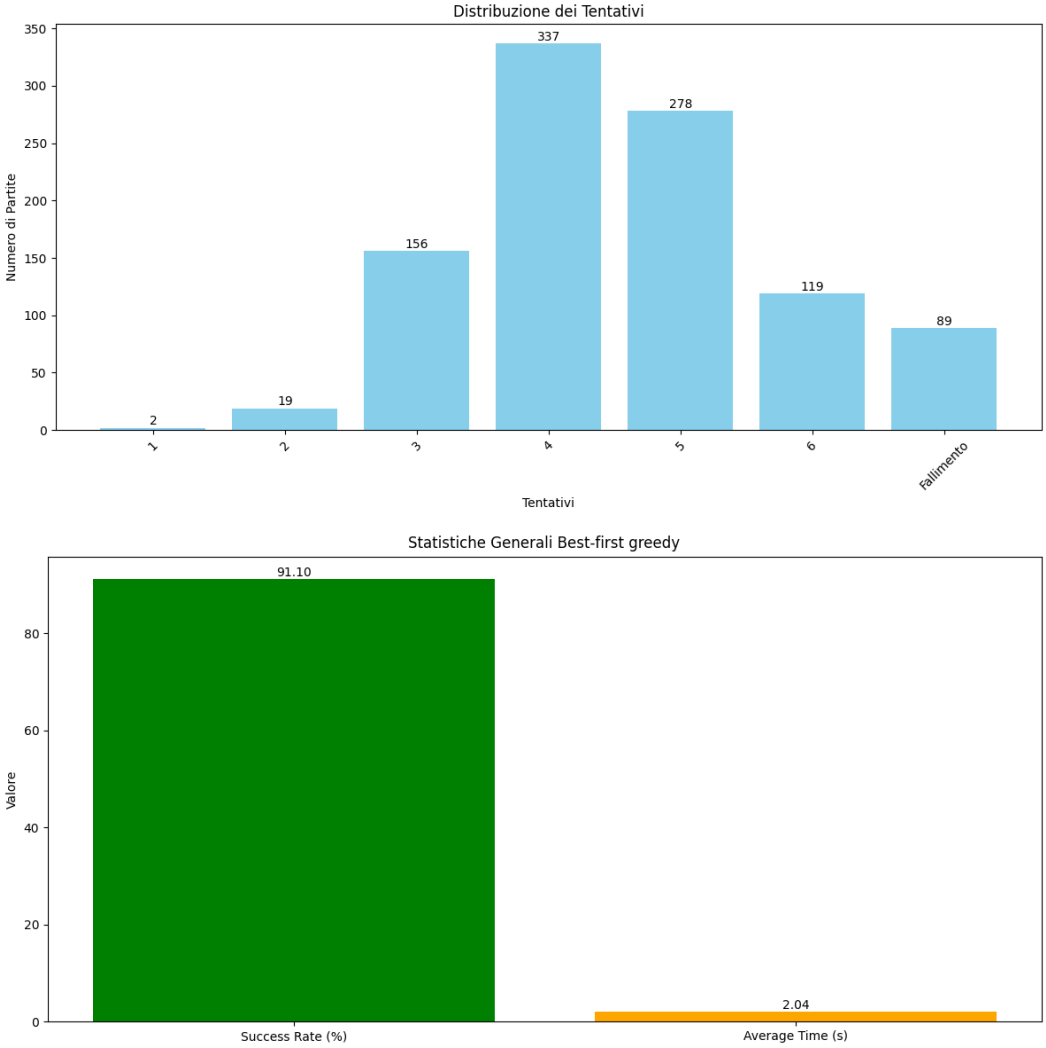

Figura 1: Statistiche algoritmo greedy con parola iniziale random

4.0.2 Algoritmo best-first greedy con parola iniziale "AROSE"

Partendo da "AROSE" come determinato in precedenza, l'algoritmo greedy continua a mostrarsi altrettanto efficiente (**Figura 2**), presentando un tasso di successo pari al **91%**. Inoltre, si è registrato un aumento percentuale dell'11,67% dei casi test vinti in 4 tentativi o meno. La differenza più significativa si ha però nel tempo medio di un test che in questo caso è pari a **0.12 secondi**, una riduzione percentuale del 94,12%. Questo dato dimostra che l'efficienza del filtraggio menzionata a paragrafo **3.5** è stata ben ottimizzata.

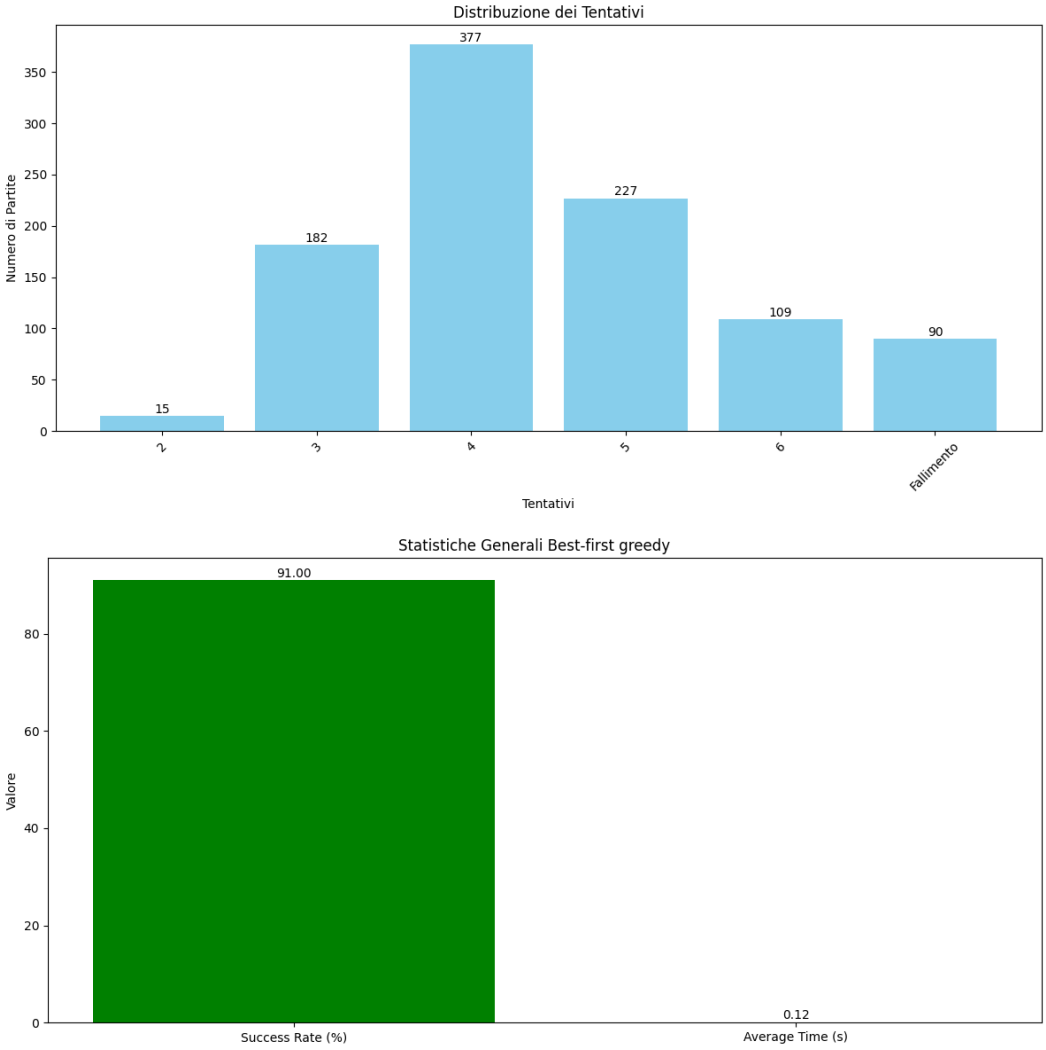

Figura 2: Statistiche algoritmo best-first greedy con parola iniziale "AROSE"

4.0.3 Algoritmo minimax con parola iniziale casuale

L'approccio con algoritmo minimax continua a mostrare un buon tasso di successo (**Figura 3**). Nonostante la presenza di 17 casi di fallimento in più rispetto all' approccio greedy, il tasso di successo è pari all'**89,40%** a fronte di **2,52 secondi** di tempo medio per un singolo testo. Il fatto che impieghi più tempo in media, è dovuto alla natura più esaustiva dell'approccio minimax. Ciò non è quindi uno svantaggio assoluto, ma una caratteristica relativa all'approccio. Un altro aspetto che emerge è che l'approccio minimax impiega più frequentemente 4 tentativi o più, segno che l'algoritmo greedy riesce a ottimizzare meglio la scelta delle parole e a convergere più rapidamente alla soluzione.

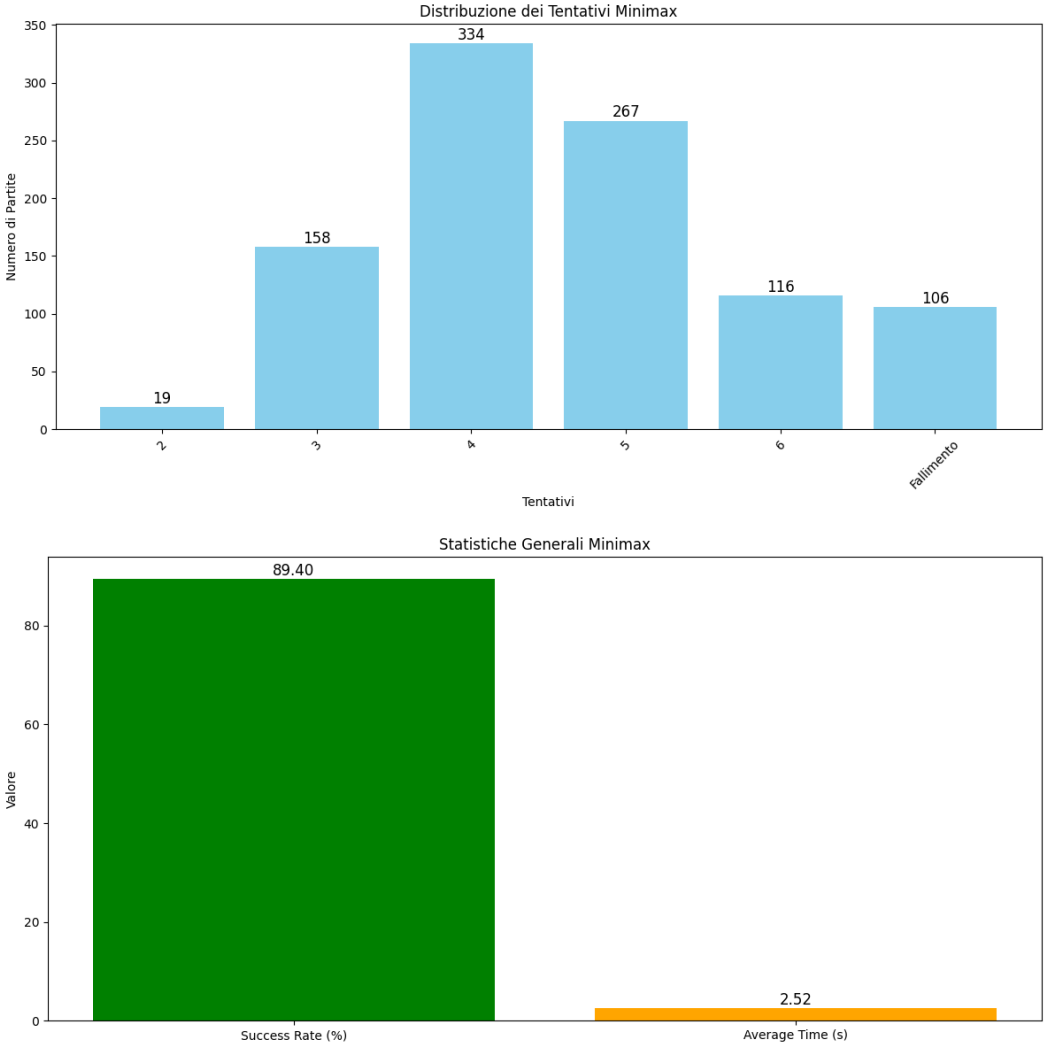

Figura 3: Statistiche algoritmo minimax con parola iniziale random

4.0.4 Algoritmo minimax con parola iniziale "AROSE"

Anche l'algoritmo minimax mostra miglioramenti quando comincia la partita con "AROSE" (**Figura 4**). Il tasso di successo, pari al **90,50%**, si avvicina di più alle strategie con greedy mantenendo un'alta competitività con quegli approcci, e il tempo medio di un test scende del 94,44% al valore di **0,14 secondi**. Migliora anche l'efficacia dell'algoritmo che vede un aumento del 3,5% dei casi di test vincenti in 4 tentativi o meno rispetto alla partenza con parola casuale. Questi risultati confermano ulteriormente come la scelta di una parola iniziale ben ponderata possa influenzare profondamente le prestazioni complessive dell'algoritmo.

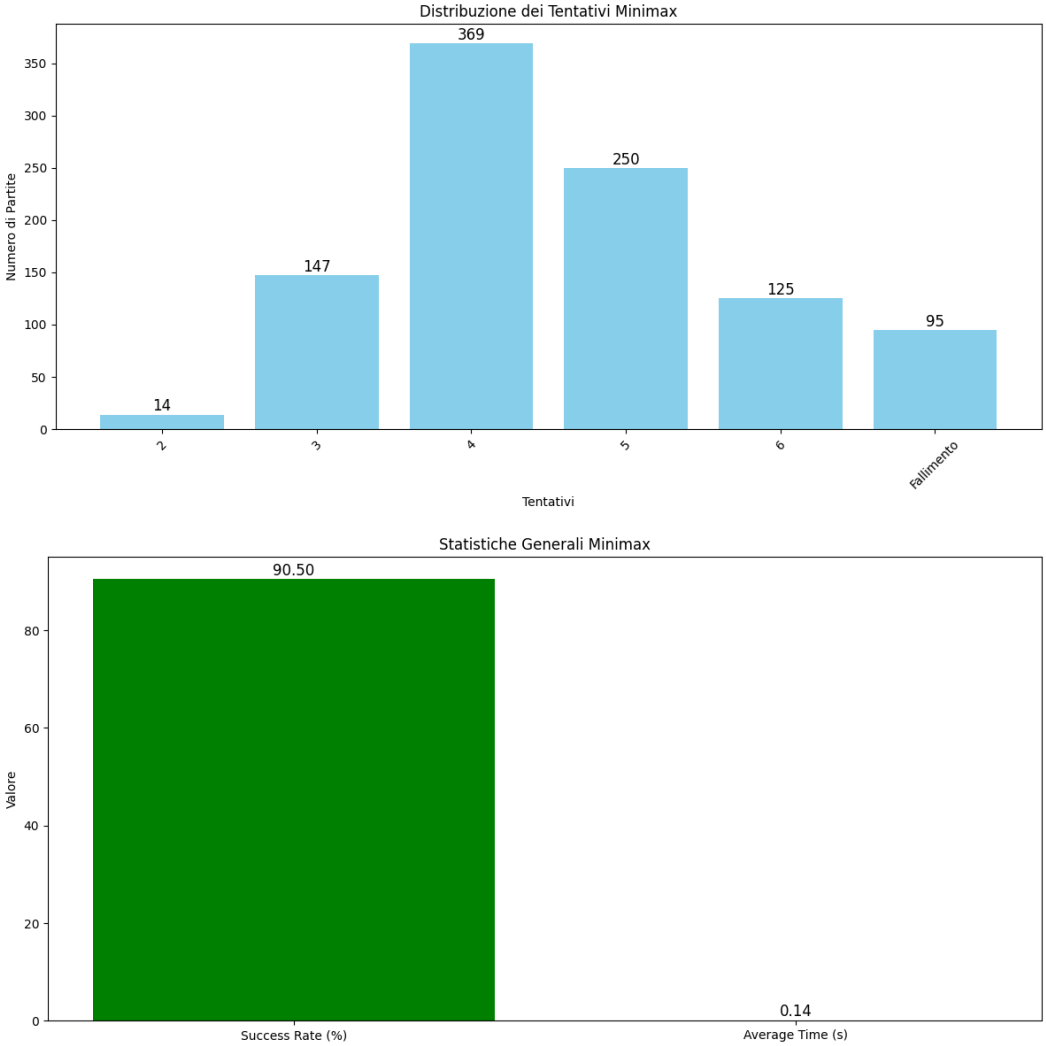

Figura 4: Statistiche algoritmo minimax con parola iniziale "AROSE"

5 Sviluppi futuri

Ci sono diverse aree di miglioramento e ottimizzazione che si possono esplorare. In primis, sembra che la maggior parte dei casi di fallimento si verifichi quando l'agente è a una lettera dal trovare la soluzione, ma sono presenti molte parole candidate come soluzione. Vi è quindi il rischio che l'agente esaurisca prima i tentativi che le parole candidate. Una possibile soluzione per prevenire questa situazione consiste nel modificare la strategia di risoluzione, facendo in modo che l'agente inserisca parole che contengano le lettere che differenziano i candidati in esame, ma che non siano mai state rilevate nei feedback precedenti. In questo modo, si riduce il numero di parole residue e di conseguenza le probabilità di individuare la soluzione nei tentativi successivi crescono. Altre idee di sviluppo includono l'integrazione di tecniche di machine learning al fine di creare un modello predittivo che apprenda dai dati dei tentativi precedenti, migliorando la capacità dell'agente di selezionare la parola migliore in base a pattern ricorrenti.