Relazione Progetto Intelligenza Artificiale

Francesco Stiro, 1000027019

1 Introduzione

Il Sudoku è un gioco di logica combinatorio con soddisfacimento di vincoli. Ne esistono diverse varianti, tuttavia in questa relazione si vuole illustrare una variante della famiglia di algoritmi BRKGA (Biased Random-Key Genetic Algorithms), in cui si cerca di risolvere automaticamente il sudoku classico. Nella fattispecie si parla di una griglia di $N^2 = (9x9)$ caselle, dove ciascuna di esse deve contenere esclusivamente numeri da 1 a 9, senza che però essi violino i vincoli. Essi sono i seguenti:

- **Restrizione sulle righe:** Tutti i numeri da 1 a 9 devono essere presenti in ogni riga della griglia, senza alcuna ripetizione.
- **Restrizione sulle colonne:** Tutti i numeri da 1 a 9 devono essere presenti in ogni colonna della griglia, senza alcuna ripetizione.
- **Restrizione sui sottoblocchi:** Tutti i numeri da 1 a 9 devono essere presenti in ogni sottoblocco 3x3 della griglia, senza alcuna ripetizione.
- **Unicità della soluzione:** La soluzione deve essere unica.

Il gioco comincia con la griglia parzialmente riempita. Il giocatore deve poi progressivamente riempirla fino al suo completamento, senza violare i vincoli sopracitati (Fig.1).

La difficoltà del sudoku varia sensibilmente in base al numero di indizi (caselle riempite) che si hanno a disposizione prima di iniziare a giocare. Chiameremo il numero di indizi presenti in una griglia all'inizio del gioco come givens. Il sudoku presentato in (Fig.1) ha givens = 30. Il numero di indizi tuttavia, non sono l'unico fattore a determinare la difficoltà di un sudoku: se i numeri dati sono distribuiti in modo tale da fornire poche indicazioni sulla soluzione, risolverlo presenta maggiori difficoltà; inoltre alcune configurazioni di numeri dati possono creare un numero consistente di vincoli rendendolo di conseguenza più complesso.

Sebbene in generale trovare la soluzione di un sudoku N x N sia un problema NP-Completo, poichè può essere ridotto, ad esempio, al problema della soddisfacibilità booleana (SAT), farlo su una griglia in input (9x9) richiede invece tempo O(1); tuttavia, essendo lo spazio delle soluzioni davvero molto

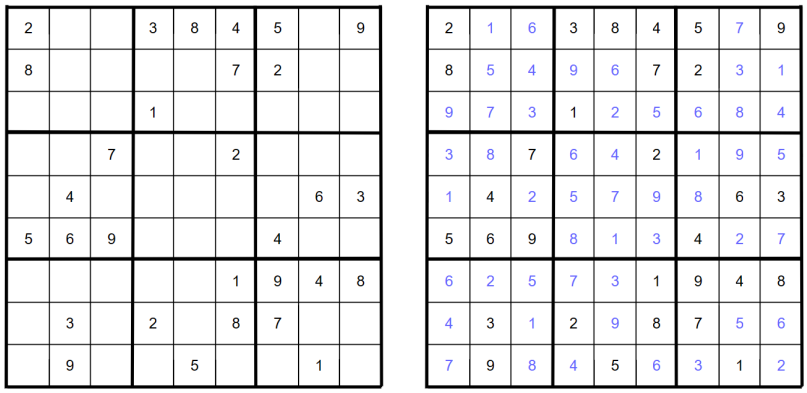

Figure 1: Esempio di sudoku di media difficoltà e relativa soluzione trovata dall'algoritmo BRKGA.

ampio, soprattutto per i sudoku più complicati, questa costante temporale può diventare un problema implementativo.

Infatti, se si volesse risolvere il sudoku tramite l'enumerazione totale di tutte le soluzioni, un ipotetico algoritmo (brute force) che prova tutte le possibili permutazioni degli elementi non fissi richiederebbe troppo tempo.

Il numero di permutazioni totali che è possibile fare in un sudoku è dato da:

$$P = \frac{(N^2 - givens)!}{\prod_{i=1}^{N}(N - x_i)!} \tag{1}$$

dove x_i sono il numero di ripetizioni nelle caselle fisse del numero i. Si può notare facilmente che al diminuire di givens, (i numeri fissi) P cresce molto velocemente, e di tutte le possibili permutazioni solo una è la soluzione. Gli algoritmi genetici, e in particolare l'algoritmo BRKGA, permettono di ottenere la soluzione ottima in un tempo ragionevole.

2 Algoritmo BRKGA

L'algoritmo BRKGA (Biased Random-key Genetic Algorithm) è una metaeuristica evolutiva che si basa su una rappresentazione dei cromosomi mediante una sequenza di valori generati casualmente (chiavi random) compresi tra 0 e 1. Ad ogni cromosoma, tramite una decodifica, viene associata una soluzione del problema di ottimizzazione che si deve risolvere e ne si calcola la qualità. Una popolazione di soluzioni viene quindi inizializzata e fatta evolvere: la elite population (composta dalle soluzioni migliori) viene mantenuta ad ogni generazione, mentre la restante parte della popolazione, (composta dalle soluzioni mutate e dalle soluzioni figlie generate da un operatore di crossover) viene scartata, dopo essere stata utilizzata per comporre la generazione successiva.

La popolazione continua ad evolversi fino al soddisfacimento di un criterio di arresto che potrebbe essere; un massimo numero di generazioni, l'aver raggiunto l'ottimo, oppure dopo un tempo di esecuzione fissato. L'algoritmo BRKGA si distingue dagli altri algoritmi genetici per l'uso di una popolazione parzialmente "biased" che consente di introdurre informazioni a priori o euristiche nel processo di evoluzione, migliorando così le prestazioni rispetto agli algoritmi genetici classici. Questo può essere utile quando si conoscono delle caratteristiche specifiche del problema che si sta risolvendo e si desidera guidare l'algoritmo verso soluzioni più promettenti o efficienti, facendone largamente uso per la risoluzione del problema del sudoku.

3 Implementazione dell'algoritmo BRKGA

3.1 Rappresentazione delle soluzioni e inizializzazione

Al fine di risolvere un dato problema, l'algoritmo BRKGA utilizza un array di chiavi randomiche per rappresentare le soluzioni del problema, tuttavia per semplicità, nella risoluzione del nostro problema si opera direttamente sulla matrice (9x9) la cui corrispondenza ad un array è immediata.

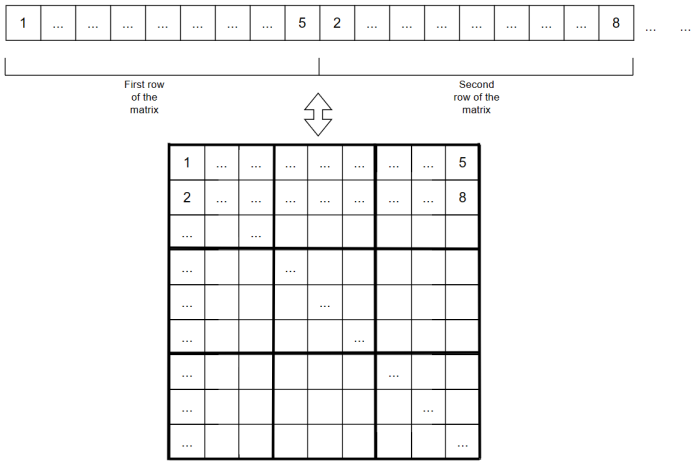

Figure 2: Corrispondenza Array - Matrice

La popolazione iniziale è quindi costituita da individui randomici. Ognuno degli individui della popolazione infatti contiene delle chiavi random da 1 a 9 nelle posizioni libere e con il vincolo che, durante la generazione dell'individuo, ogni sottoblocco 3x3 di esso debba contenere solo numeri distinti. In questa maniera si è certi che la popolazione iniziale non violi il vincolo sui sottoblocchi.

Naturalmente bisognerà fare in modo che le operazioni di mutazione e crossover mantengano inalterata questa proprietà, altrimenti si rischia di generare nelle generazioni successive individui che non rispettano le regole del gioco. Per tenere traccia dei numeri nelle posizione fisse della griglia, viene fatto uso di una matrice associata. Essa semplifica di molto le operazioni sugli operatori di mutazione/ricerca locale.

3.2 Fitness

La fitness è la funzione che vogliamo minimizzare. Per come è costruita, questa funzione prende in input un vettore di N^2 interi, quindi $F : \mathbb{N}^{N^2} \to \mathbb{N}$, tuttavia alcuni elementi del vettore sono fissi, quindi la funzione:
$F(x_1, x_2, ..., x_k, y_1, y_2, ..., y_{N^2-k}) = F^*(x_1, x_2, ..., x_k)$ dove $x_1, x_2, ..., x_k$ sono gli elementi variabili e $y_1, y_2, ..., y_{N^2-k}$ sono gli elementi fissi. In questo modo possiamo abbassare la dimensionalità di F e scriverla in funzione degli elementi variabili. Da ora in avanti ci riferiamo a F^* come F. La F è così costruita:

$$F(x_1, x_2, ..., x_k) = \sum_{i=1}^{N} r_i + \sum_{i=1}^{N} c_i + \sum_{i=1}^{N} s_i \tag{2}$$

dove r_i, c_i, s_i corrispondono rispettivamente al numero di ripetizioni dei numeri in ogni riga, colonna e sottoblocco. Noi abbiamo generato individui che non presentano ripetizioni in ogni sottoblocco, di conseguenza la F può così essere semplificata:

$$F(x_1, x_2, ..., x_k) = \sum_{i=1}^{N} r_i + \sum_{i=1}^{N} c_i \tag{3}$$

Quello che vogliamo fare è minimizzare la F:

$$min(\sum_{i=1}^{N} r_i + \sum_{i=1}^{N} c_i) \tag{4}$$

Il codominio di F è $\mathbb{N}$ di conseguenza vogliamo trovare quella configurazione $(x_1^*, x_2^*, ..., x_k^*)$ tale che $F(x_1^*, x_2^*, ..., x_k^*) = 0$. Quella sarà la soluzione unica del sudoku.

3.3 Operatore di crossover

L'operatore di crossover consente di combinare le caratteristiche di due soluzioni genitoriali al fine di generare una (o più) nuove soluzioni figlie che mantengano alcune delle caratteristiche di entrambi i genitori. Questo ci consente di portare avanti nelle generazioni successive delle soluzioni promettenti che abbiano buone probabilità, tramite altre ricombinazioni e mutazioni, di generare l'ottimo cercato.

Nell'algoritmo BRKGA si è scelto in questo modo di implementare il crossover:

1. Si scelgono due genitori, il primo pescato sempre dalla popolazione di soluzioni "elite" e l'altro dalla popolazione "non-elite".

2. Si generano due figli: ciascuno di essi erediterà la maggior parte del materiale genetico dal padre di riferimento (genitore 1 o 2 rispettivamente). Il trasferimento delle informazioni avviene blocco per blocco, in modo da non violare il vincolo sui sottoblocchi. Con una certa probabilità P_c, tuttavia, avverrà un trasferimento di informazioni dall'altro genitore e non da quello di riferimento. Per chiarire questo punto, basti pensare al figlio 1 che erediterà in maniera standard informazioni sui blocchi del genitore 1, tuttavia con una certa probabilità il blocco verrà copiato dal genitore non di riferimento; lo stesso criterio vale per il figlio 2. Questo permette di mischiare le informazioni sui sottoblocchi dei genitori in modo tale da consentire una variabilità genetica (seppur mantenendo alcune caratteristiche promettenti), che possa portare a soluzioni interessanti.

3. I figli vengono aggiunti alla nuova generazione.

4. Si continua il procedimento fin quando non si riempie lo spazio riservato alle soluzioni figlie.

Gli individui della popolazione "elite" hanno la fitness più bassa di tutte le altre soluzioni, questo è il motivo per cui si sceglie di accoppiare questo tipo di soluzioni con tutte le altre.

Algorithm 1: Crossover

```
   Input: Parent1, Parent2
   Output: Offspring1, Offspring2
 1 foreach sub-block do
 2 |   if random < P_c then
 3 |   |   Copy the block from parent 2 to offspring1;
 4 |   |   Copy the block from parent 1 to offspring2;
 5 |   end
 6 |   else
 7 |   |   Copy the block from parent 1 to offspring1;
 8 |   |   Copy the block from parent 2 to offspring2;
 9 |   end
10 end
11 return offspring1,offspring2
```

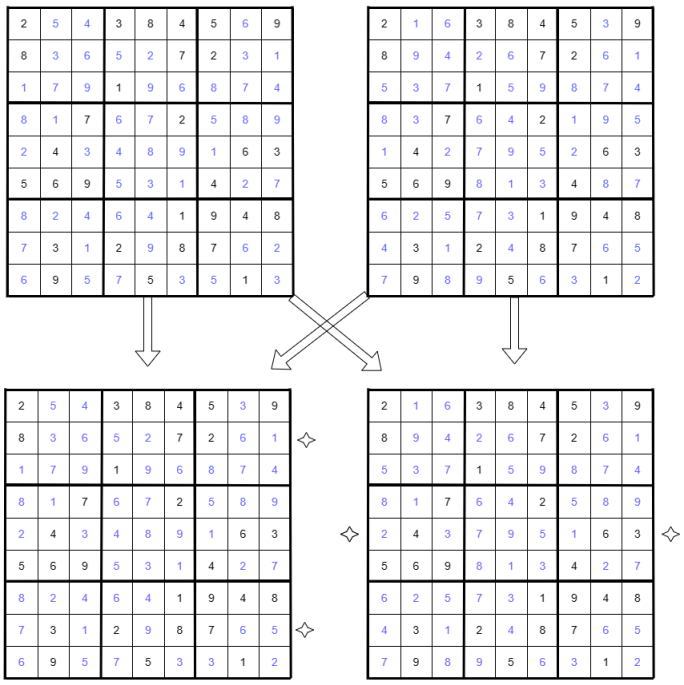

Figure 3: Esempio di crossover. Le stelle accanto i blocchi stanno ad indicare che il blocco è stato copiato dal genitore non di riferimento.

3.4 Operatore di mutazione

L'operatore di mutazione è necessario per garantire quella variabilità genetica alla popolazione in modo tale che possa essere in grado di evadere dagli ottimi locali. Per mantenere inalterati gli individui della popolazione rispetto al vincolo sui sottoblocchi, l'operatore di mutazione è implementato scambiando randomicamente due numeri all'interno dello stesso sottoblocco qualora lo scambio non coinvolga nessun numero che sta nelle posizione fisse della griglia. La mutazione di ogni blocco avviene con probabilità P_m, inoltre con probabilità $P_r << P_m$ il blocco viene invece reinizializzato randomicamente.

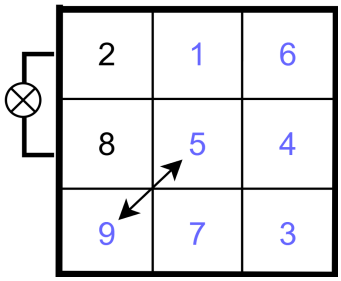

Figure 4: Blocco prima della mutazione

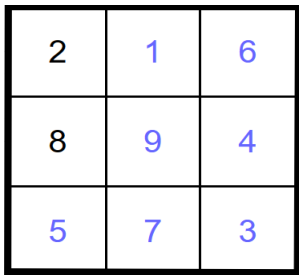

Figure 5: Blocco dopo la mutazione

Questo esempio illustra l'operatore di mutazione applicato ad un sottoblocco. Qualunque scambio che coinvolga i numeri nelle posizione fisse, ovvero 2 e 8 (in neretto) non potrà essere effettuato.

Algorithm 2: Mutation

Input: Individual
Output: Mutated Individual

```
foreach sub-block do
    if number of non-given numbers inside that sub-block >= 2 then
        if random < P_m then
            Swap two random non-given numbers on that sub-block
        end
        else if random < P_r then
            Reinitialize that sub-block
        end
    end
end
```

3.5 Euristica per una convergenza più rapida

Sebbene gli operatori illustrati in precedenza siano sufficienti per risolvere il problema, per le configurazioni più difficili il numero di generazioni necessarie per trovare un ottimo, cresce di molto. Per questo motivo è stata implementata una strategia di ricerca locale che permette di migliorare considerevolmente le prestazioni. Per capire perchè essa si renda necessaria, basta prendere in considerazione l'esempio in figura:

Supponiamo che solo queste due righe violino i vincoli del gioco del sudoku e tutti gli altri vengano rispettati, altresì supponiamo che il numero 9 e il numero 2 appartengano entrambi allo stesso sottoblocco. Per ottenere la soluzione basterebbe scambiare il numero 9 con il numero 2. Per ottenerla tuttavia dovremmo aspettare che l'operatore di mutazione scambi solo le due caselle

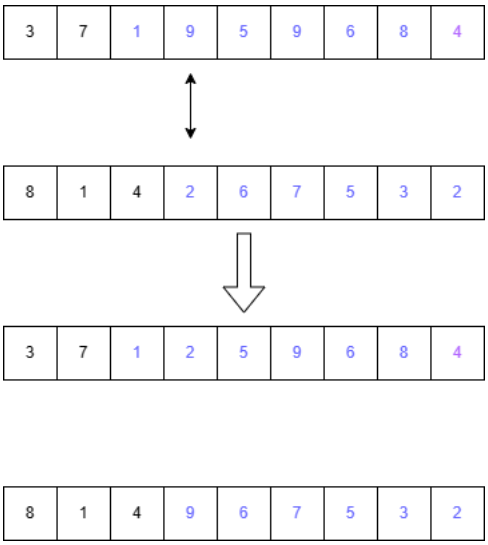

Figure 6: Miglioramento della soluzione tramite l'euristica.

all'interno dello stesso sottoblocco e non alteri gli altri, oppure che l'operatore di crossover mantenga le informazioni relative a tutti gli altri sottoblocchi e copi esclusivamente il blocco mutato ad un eventuale figlio. Tutto ciò ovviamente richiede un percorso più lungo per arrivare alla soluzione, quando in realtà ai nostri occhi è immediata. Per questo motivo è stata implementata una strategia di ricerca locale il cui funzionamento è il seguente:

1. Si cercano tutte le righe che contengono almeno un elemento ripetuto e si aggiungono ad un insieme R.
2. Si comincia estraendo le righe da R una alla volta, sia la riga estratta r.
3. Viene creato un insieme $R^{'} = R \setminus r$.
4. Una alla volta, vengono estratte le righe $r^{'}$ da $R^{'}$ e confrontate con r. Ogni riga r verrà confrontata con tutte le righe $r^{'} \in R^{'}$.
5. Siano $x_1, x_2, ..., x_n$ gli elementi duplicati di r e siano $y_1, y_2, ..., y_{n'}$ gli elementi duplicati di $r^{'}$. Se $x_i \notin r^{'}$ per qualche $i \in 1, ..., n$ e contemporaneamente $y_j \notin r$ per qualche $j \in 1, ..., n^{'}$ allora scambio x_i in r con y_j in $r^{'}$.
6. Ripetere (4) fino a che $R^{'} = \emptyset$
7. Ripetere (2) fino a che $R = \emptyset$

Lo stesso esatto procedimento viene applicato anche per le colonne.

Algorithm 3: Local search

```
   Input: Population
   Output: Improved Population
 1 foreach Individual in Population do
 2 |   Add all the illegal rows (columns) to the set R;
 3 |   while R ≠ ∅ do
 4 |   |   Extract a row (column) from R;
 5 |   |   Construct the set R';
 6 |   |   while R' ≠ ∅ do
 7 |   |   |   Extract a row (column) from R' randomly;
 8 |   |   |   if repeated numbers are in the same sub-block then
 9 |   |   |   |   if repeated numbers do not exist in both rows (columns)
   |   |   |   |    then
10 |   |   |   |   |   Swap these numbers;
11 |   |   |   |   end
12 |   |   |   end
13 |   |   end
14 |   end
15 end
```

3.6 Panoramica generale dell'algoritmo

Il funzionamento è il seguente: si inizializza una nuova popolazione di individui randomici (contenenti chiavi random da 1 a 9 nelle caselle libere), in modo tale che il vincolo su ogni sottoblocco venga rispettato. Dopodichè, fino a che un criterio di riavvio non venga soddisfatto (nel nostro caso un massimo numero di generazioni), si ordina la popolazione sulla base della fitness, si aggiorna la migliore soluzione trovata e si inizializza la nuova generazione con gli elementi migliori della precedente. Successivamente verranno aggiunti degli individui randomici che costituiscono i "mutants" all'interno della nuova popolazione e i restanti individui verranno generati tramite l'operatore di crossover spiegato precedentemente: a questi ultimi si applicherà l'operatore di mutazione di cui si è discusso. Infine a tutti gli individui della popolazione verrà applicato il criterio di ricerca locale per cercare di diminuire la loro fitness. L'algoritmo terminerà quando verrà trovata la soluzione ottima. Nel caso in cui la soluzione ottima non sia stata trovata entro un certo numero di generazioni, l'algoritmo verrà riavviato.

Il riavvio si rende necessario in caso di spazi di ricerca particolarmente complessi e/o con molti ottimi locali. Il riavvio permetterà infatti di evitare il blocco in ottimi locali: se l'algoritmo genetico converge prematuramente a una soluzione che potrebbe non essere ottimale, il riavvio consentirà di esplorare ulteriormente lo spazio di ricerca e cercherà soluzioni migliori. Inoltre aiuterà a mantenere la diversità genetica o cercherà di migliorare in generale le prestazioni, ovvero permetterà di trovare una soluzione in tempi più brevi.

Algorithm 4: BRKGA Revised

Input: $|P|, |P_e|, |P_m|$, MNG (Maximum Number of Generations)
Output: ξ^* (best solution)

$count = 0$;
Initialize value of the best solution found: $F^* \leftarrow \infty$
while *True* **do**
    Initialize population;
    while $F^* \neq 0 \wedge count < MNG$ **do**
        Find best solution ξ^+ in $P : \xi^+ \leftarrow \text{argmin}\{F(\xi)|\xi \in P\}$
        if $F(\xi^+) < F^*$ **then**
            $\xi^* \leftarrow \xi^+$
            $F^* \leftarrow F(\xi^+)$
            $count \leftarrow 0$
        end
        Partition P into two sets P_e and $P_{\bar{e}}$
        Initialize population of next generation $P^+ \leftarrow P_e$;
        Generate set P_m of mutants;
        Add P_m to population of next generation: $P^+ \leftarrow P^+ \cup P_m$;
        for $i \leftarrow 1$ *to* $0.5(|P| - |P_e| - P_m)$ **do**
            Take randomly an Individual from P_e and another one from $P_{\bar{e}}$ (I_1, I_2)
            offspring1, offspring2 = Crossover(I_1, I_2)
            $P^+ \leftarrow P^+ \cup$ {offspring1,offspring2}
        end
        Update population: $P \leftarrow P^+$
        foreach *Individual in* P **do**
            Localsearch(Individual)
        end
        foreach *Individual in* $P - P_e - P_m$ **do**
            Individual = Mutation(Individual)
            Replace the old Individual with the new one;
        end
        $count \leftarrow count + 1$
    end
    if $F^* == 0$ **then**
        return ξ^*
    end
end

4 Risultati sperimentali

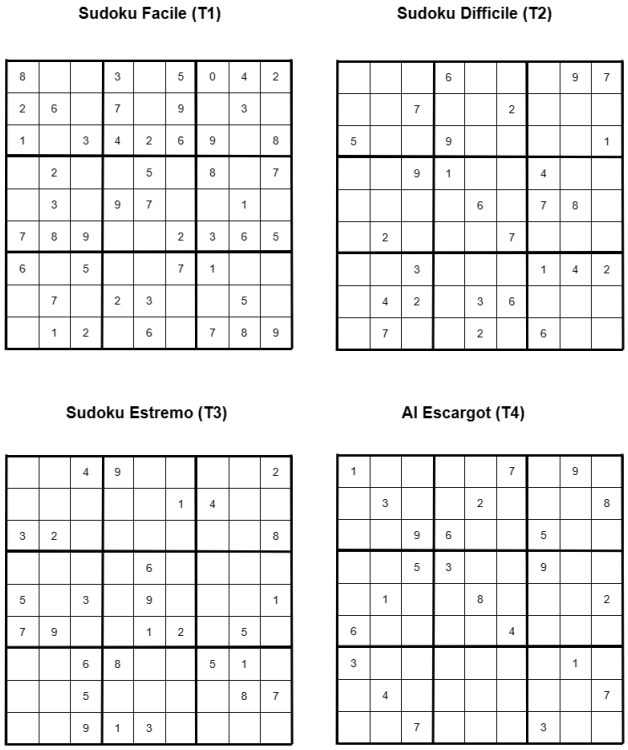

Figure 7: Sudoku di difficoltà crescente su cui sono stati effettuati i test.

I sudoku in figura sono stati utilizzati per la valutazione delle prestazioni dell'algoritmo. Oltre ai test case forniti, si vuole mettere alla prova l'algoritmo anche con il sudoku (T4) noto per essere uno dei sudoku più difficili da risolvere data la sua struttura piena di vincoli. Di seguito la tabella relativa ai risultati ottenuti:

Puzzle	Success	Avg Generations	Avg Restart	Avg Exec Time	Population
T1	20	2.4	0	0.46s	150
T2	20	10.8	0	21.63s	1500
T3	20	9.7	0	17.69s	1500
T4	20	22.11	3.2	349.8s	3000

Table 1: Risultati sperimentali dell'algoritmo.

L'algoritmo è stato eseguito su ogni istanza di input 20 volte, e i dati relativi al numero di generazioni medie e il numero di riavvii medi sono indicati in tabella. I test sono stati effettuati da una macchina che monta un Intel Core i5

9600k (3.70 Ghz),una RAM DDR4 (16 GBytes) ed eseguito su sistema operativo Windows 11. Come si può notare dai dati sopraelencati, l'unica istanza in cui l'algoritmo richiede di essere riavviato è (T4) a causa della complessità del problema. Per tutte le altre istanze, invece, riesce a trovare la soluzione ottima abbastanza velocemente seppur si faccia uso di una popolazione abbastanza numerosa. Di seguito tutti i dati relativi ai parametri utilizzati durante i test:

Parametro	Valore
Population (P)	150/1500/3000
Elite Population size (EP)	$0.25 \cdot P$
Mutants Population Size (MP)	$0.05 \cdot P$
Offsprings Population Size	$P - EP - MP$
P_c	0.1
P_m	0.3
P_r	0.05

Table 2: Parametri dell'algoritmo.

Questi valori sono stati selezionati sperimentalmente, testando varie combinazioni di essi, e cercando di trovare un equilibrio tra l'esplorazione dello spazio di ricerca e il numero di generazioni richieste per trovare una soluzione.

5 Conclusioni

In questo progetto si è fatto uso della metaeuristica BRKGA al fine di risolvere un problema complesso. L'algoritmo progettato si è dimostrato capace di risolvere anche istanze di problemi molto difficili e con uno spazio delle soluzioni molto ampio. Possibili tentativi per un miglioramento delle prestazioni potrebbero consistere nel tentare un approccio con più di una popolazione, oppure di integrare delle strategie di esclusione di alcune configurazioni in modo da ridurre lo spazio di ricerca.