Relazione Snake IA

"La sfida principale della realizzazione di un'intelligenza artificiale sta nel modo di scrivere programmi che, nella massima misura possibile, producano comportamenti razionali con una piccola quantità di codice […]"

Lo scopo di questo progetto è stato quello di creare una IA capace di "giocare" al gioco Snake.
Il gioco consiste in un serpente capace di muoversi su una matrice quadrata, con lo scopo di evitare di uscire al di fuori della matrice, evitare di mangiare sé stesso e mangiare le mele che vengono posizionate in maniera casuale all'interno della matrice stessa. Quest'ultime cambiano posizione solo quando vengono mangiate e non compaiono nelle caselle della matrice occupate dal serpente.
Formalizzandolo il problema può essere così descritto:

- Stato iniziale

  Per poter modellare il problema si è utilizzato:

  - Due numeri interi *n* ed *m* necessari per rappresentare una matrice fittizia di dimensioni $n * m$ che modellano la tabella di gioco;
  - Una coda formata inizialmente da due punti adiacenti tali che $x_0 < n - 1$ e $y_0 < m - 1$ che modellano la posizione del serpente nella tabella di gioco;
  - Una direzione che permette di determinare sia la posizione del secondo punto del serpente sia la posizione in cui si posiziona la testa del serpente nella mossa successiva;
  - Un punto, che modella la mela, di coordinate casuali tali che $x \leq n$ ed $y \leq m$ e di coordinate diverse dalle coordinate presenti nella coda che rappresenta il serpente;
  - Lo stato del gioco che viene inizialmente impostato a "Running"

- Descrizione delle azioni possibili

  Il serpente ha la possibilità di effettuare tre possibili azioni: muoversi in Avanti, a Sinistra, a Destra, tutte le azioni sono relative alla direzione precedente e non assolute.

- Descrizioni di ciò che è compiuto da ogni azione (Modello di Transizione)

  Ogni azione elimina l'ultimo punto dalla coda e lo posiziona in testa, in caso di "muovi in avanti" non viene effettuata nessuna modifica alle coordinate del punto che rappresenta la testa del serpente altrimenti viene calcolata ed applicata la nuova posizione che provocherà lo spostamento del punto verso la direzione relativa scelta. Dopo lo spostamento vengono verificate le seguenti possibilità:

  - Se la testa ha coordinate $x > n$ oppure $y > m$ allora lo stato del gioco viene impostato a "Lose"

    - Se la testa ha le stesse coordinate di un punto presente all'interno della coda oltre sé stesso allora lo stato del gioco viene impostato a "Lose"
    - Se la testa ha le stesse coordinate del punto che modella la mela, viene duplicato l'ultimo punto della coda e viene posizionato in coda al serpente e viene generato, per sostituire il punto che modellava la mela, un nuovo punto con gli stessi vincoli inziali.
- Test obbiettivo

  L'obbiettivo del serpente è mangiare tutte le mele presenti sulla tabella di gioco. Il gioco termina e di conseguenza lo stato viene impostato a "Win", quando la dimensione del serpente è uguale a $n * m$.
- Funzione di Costo

  Ogni azione che esegue il serpente ha valore unitario.

Tale problema si è affrontato partendo da un ambiente realizzato con le seguenti caratteristiche:

- Ad agente singolo: sul tavolo da gioco esiste un solo agente il cui scopo è far aumentare la sua lunghezza e quindi il suo punteggio.
- Parzialmente osservabile: l'agente non conosce a priori la posizione della mela all'interno del tavolo di gioco.
- Deterministico: l'agente conosce lo stato successivo dell'ambiente perché è determinato dallo stato corrente e dall'azione eseguita.
- Statico: l'ambiente non muta mentre l'agente "pensa" la mossa o l'insieme di mosse da eseguire.
- Episodico: in ogni episodio, l'agente riceve una percezione e poi esegue una singola azione.

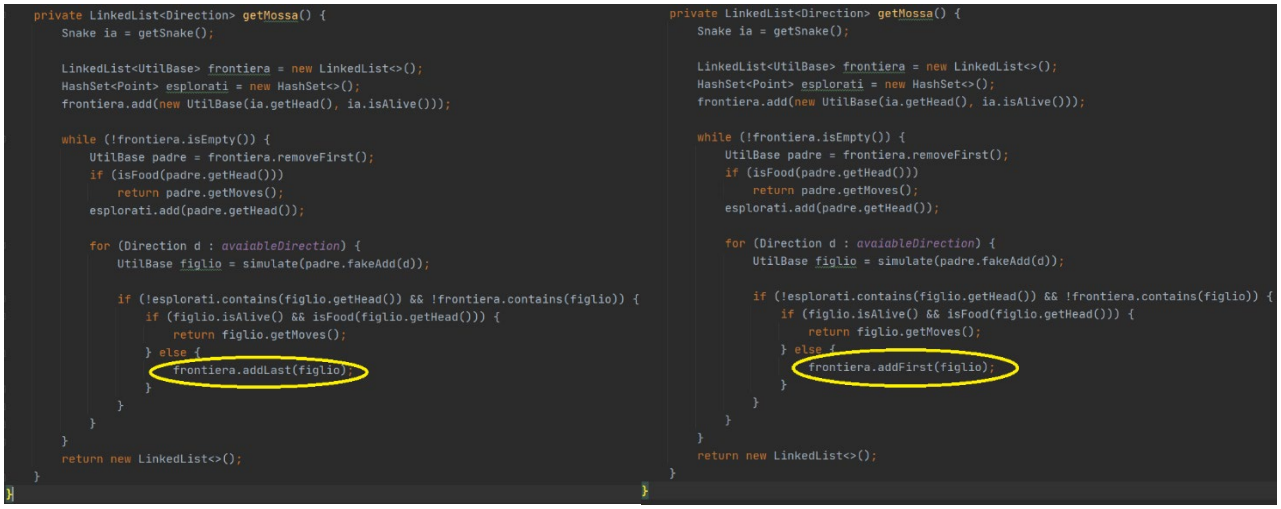
Per la realizzazione del programma agente è stato deciso di affrontare questo problema implementando non un'unica soluzione, ma diverse soluzioni utilizzando gli algoritmi di ricerca studiati durante il corso di "Fondamenti di Intelligenza Artificiale A.A. 2020/21", in modo da valutarne i diversi punti di forza e debolezza.

Ricerca non informata

I primi due algoritmi che sono stati analizzati e implementati sono la Ricerca in Ampiezza e la Ricerca in Profondità, entrambi fanno parte degli algoritmi di ricerca non informata, con questo termine ci si riferisce al fatto che le strategie non dispongono di informazioni aggiuntive sugli stati, oltre a quelle fornite nella definizione del problema. La ricerca in ampiezza è una semplice strategia di ricerca nella quale si espande prima il nodo radice, quindi i suoi successori, poi i loro successori e così via, mentre, nella ricerca in profondità si espande sempre per primo il nodo più profondo nella frontiera fino a raggiungere le foglie dell'albero di ricerca ed in maniera ricorsiva si torna indietro.

Per poter implementare entrambi gli algoritmi sono stati utilizzati:

- Una LinkedList di oggetti UtilBase, per realizzare la frontiera (l'insieme dei nodi da esplorare), che in Java può essere utilizzata come coda sia di tipo FIFO (First In First Out) che di tipo LIFO (Last In First Out);
- La classe UtilBase che offre i metodi necessari a gestire l'insieme di direzioni che verrà restituito nel caso in cui l'insieme di mosse porta al raggiungimento dell'obbiettivo;
- Un HashSet di Point, per evitare i cammini ridondanti, la cui implementazione Java permette di effettuare a tempo costante le operazioni di aggiunta, rimozioni e ricerca;



```java
private LinkedList<Direction> getMossa() {
    Snake ia = getSnake();

    LinkedList<UtilBase> frontiera = new LinkedList<>();
    HashSet<Point> esplorati = new HashSet<>();
    frontiera.add(new UtilBase(ia.getHead(), ia.isAlive()));

    while (!frontiera.isEmpty()) {
        UtilBase padre = frontiera.removeFirst();
        if (isFood(padre.getHead()))
            return padre.getMoves();
        esplorati.add(padre.getHead());

        for (Direction d : avaiableDirection) {
            UtilBase figlio = simulate(padre.fakeAdd(d));

            if (!esplorati.contains(figlio.getHead()) && !frontiera.contains(figlio)) {
                if (figlio.isAlive() && isFood(figlio.getHead())) {
                    return figlio.getMoves();
                } else {
                    frontiera.addLast(figlio);
                }
            }
        }
    }
    return new LinkedList<>();
}
```

```java
private LinkedList<Direction> getMossa() {
    Snake ia = getSnake();

    LinkedList<UtilBase> frontiera = new LinkedList<>();
    HashSet<Point> esplorati = new HashSet<>();
    frontiera.add(new UtilBase(ia.getHead(), ia.isAlive()));

    while (!frontiera.isEmpty()) {
        UtilBase padre = frontiera.removeFirst();
        if (isFood(padre.getHead()))
            return padre.getMoves();
        esplorati.add(padre.getHead());

        for (Direction d : avaiableDirection) {
            UtilBase figlio = simulate(padre.fakeAdd(d));

            if (!esplorati.contains(figlio.getHead()) && !frontiera.contains(figlio)) {
                if (figlio.isAlive() && isFood(figlio.getHead())) {
                    return figlio.getMoves();
                } else {
                    frontiera.addFirst(figlio);
                }
            }
        }
    }
    return new LinkedList<>();
}
```

Si può notare dagli estratti di codice che l'implementazione di entrambi gli algoritmi è identica a meno della posizione con cui i figli generati vengono aggiunti alla frontiera – testo evidenziato.
Nel caso della ricerca in ampiezza il figlio viene posto in coda alla frontiera, mentre per la ricerca in profondità il figlio viene posto in cima alla frontiera.

Dal punto di vista prestazionale conosciamo che:

- La ricerca in ampiezza è completa, se la soluzione esiste, perché espanderà tutti i nodi che precedono il nodo obbiettivo fino a raggiungerlo ed è ottima perché il costo di cammino è una funzione monotona non decrescente della profondità del nodo. Come conseguenza del fatto che espande tutti i nodi precedenti al nodo obbiettivo conosciamo che la sua complessità temporale è $O(b^d)$ che nel nostro problema si traduce con un fattore di ramificazione b di 3 (le possibili azioni che il serpente può effettuare) e d, che indica la profondità del nodo obbiettivo più vicino allo stato iniziale, variabile in quanto non è noto a priori la disposizione della mela. Mentre dal punto di vista della memoria occupata, dato che ogni nodo generato rimane in memoria, ci saranno $O(b^{d-1})$ nodi nell'insieme esplorato e $O(b^d)$ nodi nella frontiera, perciò la complessità complessiva è $O(b^d)$.

- La ricerca in profondità, essendo stata realizzata su un grafo finito, che evita stati ripetuti e cammini ridondanti, è completa, se la soluzione esiste, perché alla fine espanderà tutti i nodi, ma non sarà mai ottima perché sul suo cammino può trovare un cammino sub-ottimo. Per quanto riguarda la complessità temporale è limitata dalla dimensione dello spazio degli stati e generà tutti gli $O(b^m)$ nodi, dove m è la profondità massima di un nodo, bisogna precisare che m può essere molto più grande di d. Mentre dal punto di vista della memoria occupata, essendo l'implementazione uguale a quella della ricerca in ampiezza sarà $O(b^m)$, anche se si potrebbe migliorare fino ad ottenere O(bm) oppure, se si utilizza il backtraking, O(m).

Per testare nella pratica i limiti di questi due algoritmi sono stati eseguiti diversi test per entrambi gli algoritmi facendo variare la grandezza del tavolo da gioco.
I risultati che sono visibili nella tabella sottostante si compongono dei seguenti dati:

- La media dei nodi esplorati
- La media dei nodi necessari per raggiungere una soluzione
- La media del tempo necessario per raggiungere una soluzione
- Il punteggio massimo, tradotto in lunghezza massima che il serpente riesce a raggiungere in una partita.

* Per una questione di tempo, soltanto per la realizzazione della prima tabella si è fatto concludere l'algoritmo fino alla morte del serpente negli altri casi è stata fatta una media su dieci ricerche.

Ricerca in Ampiezza

Test 1: Tavolo da gioco 50*50

Tentativo	Media Nodi Esplorati	Media Nodi per raggiungere una soluzione	Media tempo necessario per trovare una soluzione (millisecondi)	Punteggio (Lunghezza Finale)
1	1298,0344	35,95074	26,014778	203
2	1590,1919	31,787878	13,606061	99
3	1345,4833	37,402683	21,771812	149
4	1526,5625	36,38889	22,04861	144
5	1152,2858	31,756302	15,184874	119
Media:	1382,51158	34,6572986	19,725227	142,8

Test 2: Tavolo da gioco 500*500

Tentativo	Nodi Esplorati	Nodi per raggiungere una soluzione	Tempo necessario per trovare una soluzione
1	90443	419	6447
2	79630	393	5276
3	210449	710	20496
4	65022	355	3941
5	6525	427	6525
6	93525	427	6770
7	5232	97	232
8	143041	531	12449
9	26663	225	1167
10	128850	502	10709
Media:	84938	408,6	7401,2

Test 3: Tavolo da gioco 1000*1000

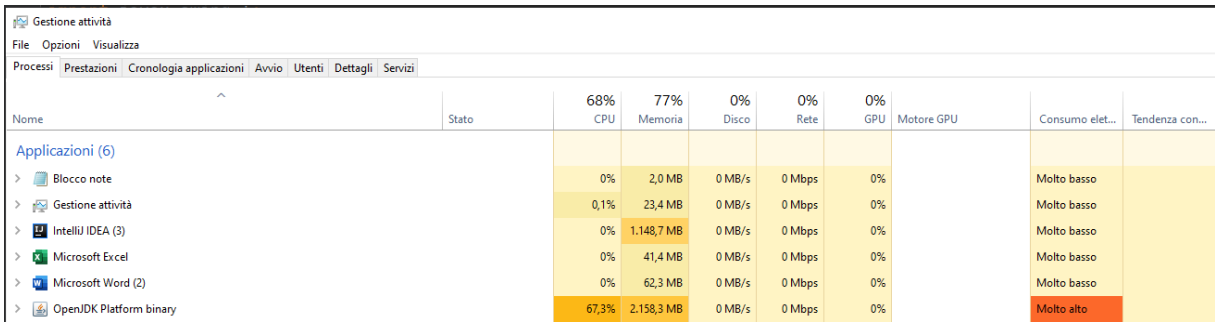
Tentativo	Nodi Esplorati	Nodi per raggiungere una soluzione	Tempo necessario per trovare una soluzione
1	598453	1097	106348
2	558701	1054	104285
3	201786	629	19956
4	101504	445	7682
5	805467	1368	164152
Media:	453182,2	918,6	80484,6

Ricerca in Profondità

Test 1: Tavolo da gioco 50*50

Tentativo	Media Nodi Esplorati	Media Nodi per raggiungere una soluzione	Media tempo necessario per trovare una soluzione (millisecondi)	Punteggio (Lunghezza Finale)
1	1028,8918	255,7027	85,5946	37
2	1208,74	294,12	111,62	50
3	967,44617	264,44617	96,323074	65
4	934,2879	262,78787	90,166664	66
5	965,8409	331,75	81,068184	44
Media:	1021,041354	281,761348	92,9545044	52,4

Test 2: Tavolo da gioco 500*500



La Java Virtual Machine può utilizzare al massimo ¼ della memoria disponibile sulla macchina quindi in un algoritmo come quello della ricerca in profondità che non trova un percorso ottimo e in cui i nodi da salvare in memoria sono b^m (per il tipo di implementazione scritta) la memoria viene saturata molto più velocemente di quanto una soluzione viene trovata. Si è deciso pertanto di evitare di portare risultati falsati dall'overhead prodotto dai content switch della memoria.

Commento dei risultati

Se si analizzano le prestazioni degli algoritmi in condizioni di normalità (tavolo da gioco 50*50), si può affermare, che l'algoritmo di ricerca con approfondimento in ampiezza riesce a ottenere un punteggio maggiore rispetto alla sua controparte, la ricerca in profondità, con prestazioni temporali leggermente superiori. Inoltre, si è compreso il significato di ottimalità di una soluzione e di come questo può incidere sulle prestazioni, sia temporali che spaziali.

Ricerca informata

Come terzo algoritmo è stato analizzato la ricerca A* che appartiene agli algoritmi di ricerca informata. Una strategia di ricerca informata può trovare soluzioni in modo più efficiente di una strategia non informata perché oltre a conoscere gli stati del problema conosce informazioni aggiuntive sul problema dette funzioni euristiche.

La funzione euristica più diffusa, utilizzata anche in questo progetto, è h(n) ovvero il costo stimato del cammino più conveniente dallo stato del nodo n a uno stato obbiettivo, che nel nostro problema si traduce nella distanza tra la posizione della testa del serpente e la posizione della mela. Avendo modellato il problema su una griglia è stato deciso di utilizzare come metodo per calcolare la distanza tra due punti la geometria del taxi o distanza Manhattan che calcola la distanza tra due punti come la somma del valore assoluto delle differenze delle loro coordinate.

L'implementazione della ricerca A* si basa sull'algoritmo di ricerca a costo uniforme a cui però viene modificato il test di valutazione dei nodi che combina g(n), il costo per raggiungere il nodo, e h(n), il costo per andare da li all'obbiettivo, formando f(n), ovvero, il costo stimato della soluzione più conveniente che passa per n.

```
private LinkedList<Direction> getMossa() {
    Snake ia = getSnake();

    PriorityQueue<UtilEuristic> frontiera = new PriorityQueue<>();
    HashSet<Point> esplorati = new HashSet<>();
    frontiera.add(new UtilEuristic(ia.getHead(), ia.isAlive(), f: 0));

    while (!frontiera.isEmpty()) {
        UtilEuristic padre = frontiera.poll();
        if (padre.isAlive() && isFood(padre.getHead())) {
            return padre.getMoves();
        }
        esplorati.add(padre.getHead());

        for (Direction d : avaiableDirection) {
            UtilEuristic figlio = simulate(padre.fakeAdd(d));

            if (!esplorati.contains(figlio.getHead()) && !frontiera.contains(figlio))
                frontiera.add(figlio);
            else
                for (UtilEuristic u : frontiera)
                    if (u.equals(figlio) && u.compareTo(figlio) > 0) {
                        frontiera.remove(u);
                        frontiera.add(figlio);
                        break;
                    }
        }
    }
    return new LinkedList<>();
}
```

Come possiamo notare l'implementazione è molto simile alla ricerca in ampiezza e alla ricerca in profondità, infatti, le modifiche effettuate rispetto agli algoritmi precedenti sono:

- Il test di valutazione non viene applicato quando il nodo viene generato ma quando il nodo viene estratto dalla frontiera

- La frontiera non è rappresentata da una coda ma un heap che viene ordinato in base alla funzione euristica gestita dalla classe UtilEuristic.
- Se il nodo è già presente nella frontiera lo si aggiorna se la funzione euristica è migliore (nel caso in cui arriviamo nello stesso nodo ma da un cammino differente e migliore).

L'algoritmo A* per sua natura è completo, ottimo e ottimamente efficiente. Completo perché esiste un numero finito di nodi di costo minore o uguale a C* (il costo di cammino della soluzione ottima). Ottimo perché h(n) è sia ammissibile che consistente, l'ammissibilità deriva dal fatto che l'euristica non sbaglia mai per eccesso la stima del costo per arrivare all'obbiettivo, mentre la consistenza esiste perché per ogni nodo n e ogni successore n', il costo stimato per raggiungere l'obbiettivo partendo da n non è superiore al costo di passo per arrivare a n' sommato al costo stimato per andare da lì all'obbiettivo. Lo svantaggio di questo algoritmo sta nel fatto che il numero di nodi all'interno dello spazio di ricerca del confine dell'obbiettivo cresce esponenzialmente con la lunghezza della soluzione. Mentre in problemi con costi di passo costanti, come il problema che stiamo analizzando, il tempo di esecuzione è analizzata in termini di errore relativo dell'euristica, infatti, risulta $O(b^{\varepsilon d})$ dove $\varepsilon = \frac{h^* - h}{h^*}$ ed è l'errore relativo.

Anche in questo caso si è proceduto andando ad effettuare diversi test, di seguito riportati, andando a variare le dimensioni del tavolo da gioco.

*Ricerca A**

Test 1: Tavolo da gioco 50*50

Tentativo	Media Nodi Esplorati	Media Nodi per raggiungere una soluzione	Media tempo necessario per trovare una soluzione (millisecondi)	Punteggio (Lunghezza Finale)
1	81,07692	33,895103	2,034965	143
2	86,57143	35,652172	1,3540373	161
3	139,04477	37,77612	3,0895522	201
4	169,37816	38,634453	4,785714	238
5	114,71352	37,18919	1,7621622	185
Media:	118,15696	36,6294076	2,60528614	185,6

Test 2: Tavolo da gioco 500*500

Tentativo	Nodi Esplorati	Nodi per raggiungere una soluzione	Tempo necessario per trovare una soluzione
1	404	403	99
2	479	478	119
3	438	437	135
4	686	685	235
5	746	745	279
6	964	963	333
7	758	757	204
8	462	461	164
9	370	369	130
10	178	177	63
Media:	548,5	547,5	176,1

Test 3: Tavolo da gioco 5000*5000

Tentativo	Nodi Esplorati	Nodi per raggiungere una soluzione	Tempo necessario per trovare una soluzione
1	6668	6667	9536
2	8986	8985	19555
3	9144	9143	19290
4	6061	6060	7761
5	7187	7186	10336
Media:	7609,2	7608,2	13295,6

Commento dei risultati

Analizzando i dati ottenuti possiamo osservare come l'algoritmo A* riuscendo ad esplorare solo i nodi strettamente necessari, raggiunge l'obbiettivo molto più velocemente dei precedenti algoritmi riuscendo a funzionare anche su tavoli da gioco molto più grandi (5000*5000) con tempi di ricerca ridotti al minimo, tutto grazie alla sua caratteristica di essere completo, ottimo e ottimamente efficiente.

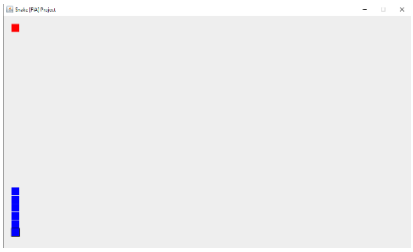
Ricerca locale

Si è deciso di implementare due algoritmi di ricerca locale: l'Hill Climbing e il Simulated Annealing sopravvalutando, però, la tipologia di problemi che questo tipo di ricerca può risolvere. Gli algoritmi visti fin qui sono progettati per esplorare sistematicamente gli spazi di ricerca. Questa sistematicità la si ottiene mantenendo in memoria uno o più cammini e registrando quali alternative sono esplorate in ogni punto del cammino. Per molti problemi il camino verso l'obbiettivo è irrilevante, ciò che conta è la configurazione finale. Gli algoritmi di ricerca locale operano su un singolo nodo corrente invece che su cammini multipli e in generale si spostano solo nei nodi immediatamente adiacenti.
L'algoritmo di ricerca Hill Climbing si compone di un semplice ciclo che si muove continuamente verso l'alto in quello che viene definito il panorama degli spazi, e termina quando raggiunge un picco che non ha vicini di valore più alto. L'Hill Climbing procede molto rapidamente verso la soluzione, perché di solito è abbastanza facile migliorare uno stato sfavorevole. Sfortunatamente spesso rimane bloccato per le seguenti ragioni:

- Massimi Locali: un massimo locale è un picco più alto degli stati vicini, ma inferiore al massimo globale.
- Creste: le creste danno origine a una serie di massimi locali
- Plateau: un'area piatta del panorama dello spazio degli stati.

```
private Direction getMossa(){
    Snake ia = getSnake();

    UtilEuristic corrente = new UtilEuristic(ia.getHead(), ia.isAlive(), getDistanceFromApple(ia));
    for(Direction d : avaiableDirection){
        UtilEuristic vicino = (UtilEuristic) simulateOneDirectoin(d);
        if(vicino.isAlive() && vicino.compareTo(corrente) < 0)
            corrente = vicino;
    }
    return corrente.getMoves().getFirst();
}
}
```



Questo tipo di implementazione sul problema dello Snake risente dei massimi locali, per poter risolvere questo problema, non potendo utilizzare il riavvio casuale, si è pensato di muovere in maniera casuale il serpente se una volta uscito dal ciclo lo stato corrente non ha trovato una mossa. Di seguito il codice modificato.

```
    private Direction getMossa(){
        Snake ia = getSnake();

        UtilEuristic corrente = new UtilEuristic(ia.getHead(), ia.isAlive(), getDistanceFromApple(ia));
        for(Direction d : avaiableDirection){
            UtilEuristic vicino = simulateOneDirectoin(d);
            if(vicino.isAlive() && vicino.compareTo(corrente) < 0)
                corrente = vicino;

        }
        if (corrente.getMoves().size() == 0){
            int avaiableDirectionSize = avaiableDirection.length;
            Random r = new Random();
            corrente.getMoves().add(avaiableDirection[r.nextInt(avaiableDirectionSize)]);
        }
        return corrente.getMoves().getFirst();
    }
}
```

L'algoritmo Simulated Annealing si compone di un ciclo all'interno del quale viene decisa una mossa casuale tra quelle disponibili che viene sempre accettata se migliora la situazione corrente, altrimenti viene accettata con probabilità p. La probabilità decresce esponenzialmente con la "cattiva qualità" della mossa e con la "temperatura" T che scende costantemente.

```
private LinkedList<Direction> getMossa() {
    ready = false;
    Snake ia = getSnake();

    Random r = new Random();
    int avaiableDirectionSize = avaiableDirection.length;
    UtilBase current = new UtilBase(ia.getHead(), ia.isAlive());
    for (double t = temperature; t > 1; t *= coolingFactor) {
        UtilBase next = simulate(current.fakeAdd(avaiableDirection[r.nextInt(avaiableDirectionSize)]));
        if (next.isAlive())
            if (Math.random() <= Utilities.probability(getDistanceFromApple(next.getHead()), getDistanceFromApple(current.getHead()), t))
                current = new UtilBase(next);
    }
    return current.getMoves();
}
```

Come si può vedere dagli estratti di codice entrambi gli algoritmi semplificano notevolmente il codice rispetto gli algoritmi trattati in precedenza e portano con sé due vantaggi importanti:

- Usano una quantità di memoria costante.
- Possono spesso trovare soluzioni ragionevoli in spazi degli stati grandi o infiniti in cui gli algoritmi sistematici non sono applicabili.

Per completezza, si riportano di seguito i test effettuati con entrambi gli algoritmi, stavolta solo su un tipo di tavolo da gioco.

Hill Climbing

Test 1: Tavolo da gioco 50*50

Tentativo	Punteggio (Lunghezza Finale)
1	63
2	43
3	54
4	34
5	52
Media:	49,2

Simulated Annealing

Test 1: Tavolo da gioco 50*50

Tentativo	Punteggio (Lunghezza Finale)
1	5
2	7
3	4
4	6
5	10
Media:	6,4

Conclusioni

Durante questo progetto ho avuto la possibilità di analizzare diversi algoritmi per poter meglio comprendere "l'arte" della ricerca. Attraverso un percorso di sperimentazione ho implementato diversi algoritmi di ricerca potendo analizzare quelle che sono le caratteristiche principali, prima teoricamente e poi mettendole in pratica, tra cui le prestazioni temporali, spaziali, di completezza e ottimalità.
Concludo potendo affermare che l'algoritmo migliore che ho implementato è stato l'algoritmo A* il cui risultato è offerto dalle sue ottime caratteristiche.