Documentazione iTeam

Sviluppato da:
Gennaro Cecco
Luigi Giacchetti
Vincenzo Liguori
Emanuele Scarpa

« L'intelligenza è la capacità di evitare di fare il lavoro, pur riuscendo a farlo. »
- Linus Torvalds

Link Github:
https://github.com/GennaroCecco/iTeam

Indice

1 PREMESSA

iTeam si basa sul progetto di Ingegneria Del Software "Agency Formation". Il suo scopo, è quello di migliorare i processi di:

- Reclutamento Candidati
- Formazione Dipendenti
- Gestione Materiale di Formazione

Inoltre, permette la creazione di Team di dipendenti a cui è affidato un progetto. Questa funzionalità vieni migliorata attraverso l'intelligenza artificiale: *iTeam* .

2 BREVE PROPOSTA DI PROGETTO

Sviluppo di un agente intelligente che, sulla base dei requisiti di un progetto, riesca a formare un team adatto per quest'ultimo, analizzando le skills dei dipendenti e valutandone il loro grado di bontà.

3 SISTEMA

Definizioni, acronimi e abbreviazioni:

- TM: Team Manager.
- Tm: Team member.
- Soft-skill: Skill interpersonali.
- Geni: Dipendenti.
- Individuo: Team.
- avg_Skills: valore che indica la soluzione ottima, cioè 5.0.

3.1 Sistema Attuale

Al momento non è presente alcun sistema che gestisce la creazione automatica di un Team in base ai requisiti espressi dal TM. Quindi attualmente, per creare un team, bisogna controllare le singole skill di ogni dipendente e scegliere i dipendenti migliori.
Con questo tipo di approccio sorgono diversi svantaggi come:

- Nel caso si abbia una lunga lista di dipendenti, avremo lunghi periodi per l'esaminazione di ogni futuro Tm.
- Possibilità di non scegliere correttamente il dipendente più adatto a lavorare in quel team.

3.2 Sistema proposto

iTeam è un'IA che supporta l'attività di creazione di un Team, in modo tale da avere tempi molto più brevi per l'esaminazione del singolo dipendente. iTeam è capace di creare team di dipendenti in base ai requisiti del progetto da svolgere. I Tm verranno scelti in base alle skills richieste dal Progetto. Nel caso in cui venga richiesta una skill che un dipendente non possiede, ma avrà altre skills richieste, verrà comunque processato dall'IA. Il livello delle skill di un dipendente è segnato con un valore auto-valutativo compreso tra: 0 come il più basso e 5 come il più alto. Nel caso in cui il livello della skill sia 0, il dipendente non verrà ritenuto in possesso di quella competenza, in quanto ciò sta ad indicare che il dipendente non ha più dimestichezza con quella skill. Alla fine dell'esecuzione di iTeam avremo il team con il punteggio migliore per lavorare a quel progetto.

4 SPECIFICHE PEAS

L'ambiente viene descritto tramite la formulazione PEAS. Le specifiche PEAS sono utili per fornire una chiara definizione del problema e dell'ambiente in cui l'agente deve operare.
Le nostre specifiche sono:

4.1 Performance

Le performance dell'agente sono valutate attraverso il valore/punteggio del team restituito dall'IA. Ex: Prendiamo in considerazione due team, A con valutazione 4.0 e B con valutazione 2.5. iTeam sceglierà il team con valutazione più alta, in questo caso il team A.

4.2 Environment

4.2.1 Completamente osservabile:

I sensori dell'ambiente danno accesso allo stato completo dell'ambiente in ogni momento.

4.2.2 Agente singolo:

L'ambiente presenta un singolo agente.

4.2.3 Stocastico:

Lo stato successivo del team non è completamente determinato dallo stato corrente ma è condizionato da una componente randomica.

4.2.4 Episodico:

Le modifiche apportate all'ambiente sono indipendenti fra loro, quindi appartengono a due episodi atomici diversi.

4.2.5 Discreto:

Il numero di percezioni dell'agente è limitato in quanto ha un numero discreto di specifiche, azioni e percezioni possibili.

4.3 Actuators

Team suggerito sulle skill richieste e relativa pagina web.

4.4 Sensors

I sensori dell'agente consistono nei bottoni della pagina web.

5 DATASET

Il dataset utilizzato è stato creato artificialmente con il seguente script in Python:

```python
fake = Faker()

records = 1000000
print("Sto eseguendo %d records\n" % records)

fieldnames = ['id', 'name', 'surname', 'email', 'skill1', 'skill2', 'skill3', 'level1', 'level2', 'level3']
writer = csv.DictWriter(open("dataset.csv", "w"), fieldnames=fieldnames)

skill = ['Java', 'C', 'HTML', 'Python', 'React', 'Node', 'C++', 'javascript',
         'CSS', 'Ruby', 'C#', 'Android', 'SQL', 'PHP']

names = []
level = []
emails = []
for n in range(records):
    names.append(fake.name())
    emails.append(fake.email())
writer.writerow(dict(zip(fieldnames, fieldnames)))
for i in range(0, records):
    add1 = random.choice(skill)
    add2 = random.choice(skill)
    while add1 == add2:
        add2 = random.choice(skill)
    add3 = random.choice(skill)
    while add1 == add3 or add2 == add3:
        add3 = random.choice(skill)
    gener = random.choice(names).split()
    nome = gener[0]
    cognome = gener[1]
    email = nome[0].lower()+"." + cognome.lower()+str(i+6) + "@gmail.com"

    writer.writerow(dict([
        ('id', i+6),
        ('name', nome),
        ('surname', cognome),
        ('email', email),
        ('skill1', add1),
        ('skill2', add2),
        ('skill3', add3),
        ('level1', str(random.randint(0, 5))),
        ('level2', str(random.randint(0, 5))),
        ('level3', str(random.randint(0, 5)))]))

print("finito")
```

La scelta progettuale di generare il dataset artificialmente è dovuta alla necessità di rimanere fedeli alle precedenti specifiche del sistema su cui adopera iTeam, ovvero Agency Formation. Quest'ultimo prevedeva che ogni dipendente avesse a disposizione la possibilità di specificare (possedere) 3 skills contemporaneamente con un livello assegnatosi in maniera del tutto autonoma. Sono state generate anche informazioni ornamentali come: nome, cognome e email, in modo tale da avere un completo adattamento anche con il DB di Agency Formation. In questo modo abbiamo abilitato la possibilità di salvare i dipendenti del dataset nella base di dati.
Lo script genera in maniera randomica: nome, cognome, e-mail e tre skill con i rispettivi livelli.

- ID(integer): Identificatore del record;
- Nome(string): Nome della persona;
- Cognome(string): Cognome della persona;
- e-mail(string): Email della persona che viene generata sulla base del nome e del cognome e di un contatore;
- skill1/skill2/skill3(string): 3 Skills diverse che vengono scelte da una lista predefinita di linguaggi di programmazione;
- level1/level2/level3(integer): il valore numerico scelto casualmente da 0 a 5.

In totale vengono generati 1.000.000 dipendenti. La scelta di un dataset così ampio è basata sul fatto che iTeam eseguirà una "initialization" sulla popolazione, ovvero, andrà a scegliere quei dipendenti che avranno le skills richieste e quindi risulterà una popolazione ridotta. Inoltre si vuole avere la possibilità di testare il comportamento dell'IA e dei suoi vari operatori in situazioni diverse tra di loro.

Esempio: Dimensione dataset= 1.000.000 geni;
Skill richieste = (Java, HTML, CSS);
Dipendenti effettivi = 300.000* geni;
Popolazione = 75.000 individui (team).
*I valori sono unicamente indicativi.

6 SOLUZIONE

Data la composizione del problema – ovvero un problema di ottimizzazione – sono stati scelti gli algoritmi genetici perché utilizzano un approccio di esplorazione più ampia rispetto agli altri algoritmi di ricerca. Attraverso l'utilizzo di meccanismi come la selezione, il crossover e la mutazione, gli algoritmi genetici possono attraversare e valutare diverse regioni dello spazio delle soluzioni in parallelo. Gli algoritmi genetici, grazie alla loro capacità di esplorare diverse regioni dello spazio delle soluzioni, hanno una maggiore probabilità di superare i minimi locali e raggiungere soluzioni migliori. Sulla base di ciò abbiamo capito che ci servisse una strategia che ci permettesse di creare più team, valutarli, scegliere il migliore e nel caso modificarli durante l'esecuzione dell'algoritmo. Sono state scartate strategie che utilizzano:

- La teoria dei giochi – ricerca con avversari - non è applicabile in quanto abbiamo un singolo agente
- Non avendo rappresentato il nostro problema sottoforma di grafo o albero, sono state scartate strategie di ricerca come:
  - Algoritmi di Ricerca Informata: Considerano interi cammini dallo stato iniziale a quello obiettivo e non possono essere applicati in problemi con spazio degli stati grandi;
  - Algoritmi di Ricerca non Informata: Gli algoritmi di ricerca non informata possono essere influenzati dall'ordine in cui i nodi vengono esplorati. A seconda dell'ordine, potrebbe essere necessario esplorare molti nodi prima di trovare la soluzione desiderata.

Gli algoritmi genetici partono da un certo numero di possibili soluzioni, ognuna delle quali rappresenta un individuo, e provvede a farle evolvere nel corso dell'esecuzione: a ciascuna iterazione, esso opera una selezione di individui della popolazione corrente, impiegandoli per generare nuovi elementi della popolazione stessa, che andranno a sostituire un pari numero d'individui già presenti, e a costituire in tal modo una nuova popolazione per l'iterazione, o generazione, seguente. Tale successione di generazioni evolve verso una soluzione ottima del problema assegnato.

6.1 Soluzione ottima

La soluzione ottima è rappresentata dal team che ha come valutazione 5.0, ovvero il massimo valore assegnabile a un team.

$$fit(team) = avg(\sum_{d=1}^{4} dip_d)$$

Per ottenere la valutazione totale della soluzione candidata andremo a calcolare la media dei singoli dipendenti che formano il team, che si ottiene attraverso la media della somma delle singole skill del dipendente.

$$dip = avg(\sum_{s=1}^{3} skill_s)$$

Esempio:
OPT = 5.0
fit(Team1) = 3.70
fit(Team2) = 4.50.
Verrà scelto il team Team2 poiché **OPT − fit(Team2) < OPT − fit(Team1)**
Se il team scelto ha valutazione 5.0, significa che avremo la seguente situazione:

$$OPT - fit(team) = 0$$

$$5.0 - avg(\sum_{d=1}^{4} dip_d) = 0$$

Quindi il team preso in esame è **ottimo**.

7 OPERATORI ALGORITMO

Gli operatori utilizzati sono stati implementati da zero.

7.1 Selection

Sono stati implementati vari tipi di selezione:

7.1.1 Truncation

```java
public static ArrayList<TeamRefactor> truncationSelection(ArrayList<TeamRefactor> popolazione) {
    popolazione.sort(Comparator.comparing(TeamRefactor::getValoreTeam).reversed());
    ArrayList<TeamRefactor> selectedPop = new ArrayList<>();
    int portion = (int) (popolazione.size() * 0.5);
    for (int i = 0; i < portion; i++) {
        selectedPop.add(popolazione.get(i));
    }

    return selectedPop;
}
```

Il processo di selezione truncation consiste nel classificare gli individui in base alla loro fitness e poi selezionare un numero prefissato di migliori individui per la creazione della generazione successiva. Viene scelto un valore M uguale al 50% della popolazione, composta dai migliori individui (cioè con valore di fit più alto).

7.1.2 *Roulette-Wheel*

```java
public static ArrayList<TeamRefactor> rouletteWheel(ArrayList<TeamRefactor> popolazione,
                                                    ArrayList<String> skills) {
    HashMap<TeamRefactor, Double> population = new HashMap<>();
    double random = new Random().nextDouble() * totalSum;
    population = setProbabilityForRoulette(popolazione, skills);
    ArrayList<TeamRefactor> toReturn = new ArrayList<>();
    double sum = 0;
    for (int i = 0; i < popolazione.size(); i++) {
        sum += population.get(popolazione.get(i));
        if (random <= sum) {
            toReturn.add(popolazione.get(i));
        }
    }
    return toReturn;
}
```

Nell'operatore di Roulette-Wheel, ogni individuo viene associato a una fetta di una ruota immaginaria, il cui angolo di apertura è proporzionale alla sua fitness. Più alta è la fitness di un individuo, più grande sarà la sua fetta di ruota. La somma totale delle fitness di tutti gli individui nella popolazione rappresenta l'intera circonferenza della ruota. La probabilità che un individuo venga selezionato è data dalla seguente formula:

$$P(choice = i) = \frac{fitness(i)}{\sum_{j=1}^{n} fitness(j)}$$

Ogni individuo riceve una porzione di una ruota proporzionata al valore di valutazione relativo al resto delle valutazioni.
La selezione avviene attraverso un processo casuale. Viene generato un numero casuale tra 0 e la somma totale delle fitness. Quindi, la ruota viene fatta girare e l'individuo il cui intervallo corrisponde al numero generato viene selezionato. Poiché gli individui con fitness più alta avranno fette di ruota più ampie, hanno maggiori probabilità di essere selezionati rispetto agli individui con fitness più bassa, ma tutti gli individui hanno almeno una possibilità di essere scelti.

```java
public static HashMap<TeamRefactor, Double> setProbabilityForRoulette(ArrayList<TeamRefactor> popolazione,
                                                                       ArrayList<String> skills) {
    HashMap<TeamRefactor, Double> probability = new HashMap<>();
    double totalSum = 0.0;
    for (int i = 0; i < popolazione.size(); i++) {
        popolazione.get(i).calcolaFitness(skills);
        totalSum += popolazione.get(i).getValoreTeam();
    }
    for (int i = 0; i < popolazione.size(); i++) {
        popolazione.get(i).calcolaFitness(skills);
        probability.put(popolazione.get(i), popolazione.get(i).getValoreTeam() / totalSum);
    }
    return probability;
}
```

7.1.3 *K-way Tournament*

```
public static ArrayList<TeamRefactor> tournamentSelection(ArrayList<TeamRefactor> popolazione,
                                                          ArrayList<String> skills, int tournamentSize,
                                                          int NumberOfIndividualForTournament) {
    ArrayList<TeamRefactor> toReturn = new ArrayList<>();
    for (int j = 0; j < tournamentSize; j++) {
        TeamRefactor best = null;
        ArrayList<TeamRefactor> pop = new ArrayList<>();
        for (int i = 0; i < NumberOfIndividualForTournament; i++) {
            TeamRefactor ind = popolazione.get(new Random().nextInt(popolazione.size()));
            while (pop.contains(ind)) {
                ind = popolazione.get(new Random().nextInt(popolazione.size()));
            }
            pop.add(ind);
        }
        //start tournament
        for (int i = 0; i < pop.size(); i++) {
            TeamRefactor ind = pop.get(i);
            ind.calcolaFitness(skills);
            if (best == null || ind.getValoreTeam() > best.getValoreTeam()) {
                best = ind;
            }
        }
        toReturn.add(best);
    }
    return toReturn;
}
```

Questo tipo di selezione prevede un numero "n" di tornei dove vengono messi a confronto "k" team scelti in maniera random dalla popolazione. La dimensione della popolazione che viene passata a questo operatore è scelta casualmente. Il tipo di torneo mette a confronto 3 team, e tra questi viene scelto il migliore. Tutto questo viene eseguito ciclicamente per tutta la dimensione della popolazione che inizialmente è passata all'operatore e alla fine avremo una nuova popolazione. Il Kway Tournament consente di promuovere la diversità nella selezione degli individui migliori, poiché più partecipanti vengono confrontati tra loro. Ciò aumenta le probabilità di trovare soluzioni migliori e favorisce una maggiore esplorazione dello spazio delle soluzioni.

7.1.4 *Rank Selection*

```java
public static ArrayList<TeamRefactor> rankSelection(ArrayList<TeamRefactor> popolazione, ArrayList<String> skills) {
    double sum = 0.0;
    HashMap<TeamRefactor, Double> probabilites = new HashMap<>();
    ArrayList<TeamRefactor> selectedPop = new ArrayList<>();
    probabilites = getProbabilityForRank(popolazione);
    for (int i = 0; i < popolazione.size(); i++) {
        popolazione.get(i).calcolaFitness(skills);
        sum += popolazione.get(i).getValoreTeam();
    }
    for (int i = 0; i < popolazione.size(); i++) {
        Random r = new Random();
        double randomValue = 0.0 + (sum - 0.0) * r.nextDouble();
        for (int j = 0; j < popolazione.size(); j++) {
            if (probabilites.get(popolazione.get(j)) < randomValue) {
                selectedPop.add(popolazione.get(j));
                break;
            }
        }
    }
    return selectedPop;
}
```

Si basa sulla classificazione degli individui, piuttosto che sul loro valore di fit. Il grado n è associato al migliore individuo, mentre il peggiore individuo viene associato al grado 1. Quindi, in base al loro grado, ogni individuo i ha la probabilità di essere selezionato dall'espressione:

$$P(i) = \frac{rank(i)}{n * (n-1)}$$

```java
public static HashMap<TeamRefactor, Double> setProbabilityForRoulette(ArrayList<TeamRefactor> popolazione,
                                                                       ArrayList<String> skills) {
    HashMap<TeamRefactor, Double> probability = new HashMap<>();
    double totalSum = 0.0;
    for (int i = 0; i < popolazione.size(); i++) {
        popolazione.get(i).calcolaFitness(skills);
        totalSum += popolazione.get(i).getValoreTeam();
    }
    for (int i = 0; i < popolazione.size(); i++) {
        popolazione.get(i).calcolaFitness(skills);
        probability.put(popolazione.get(i), popolazione.get(i).getValoreTeam() / totalSum);
    }
    return probability;
}
```

7.2 Elitism

L'operatore di elitismo permette il salvataggio di una piccola parte della popolazione chiamata Elite. In questa Elite, ci sono gli individui migliori per quella generazione. Se l'Elite partecipa al crossover, e alla mutazione, l'algoritmo restituisce in poche generazioni una soluzione molto vicina all'ottimo, se non proprio una soluzione ottima. Questo operatore viene utilizzato solo per eseguire dei test.

```
public static ArrayList<TeamRefactor> elitism(ArrayList<TeamRefactor> popolazione, ArrayList<TeamRefactor> off,
                                               ArrayList<String> skills) {
    ArrayList<TeamRefactor> population = ordina(popolazione,skills);
    ArrayList<TeamRefactor> offSpring = ordina(off,skills);
    int compElite = (int) (population.size() * elitism_size);
    ArrayList<TeamRefactor> newPop = new ArrayList<>();
    int secondIndex = population.size() - compElite;
    for (int i = 0; i < compElite; i++) {
        newPop.add(population.get(i));
    }
    for (int i = 0; i < secondIndex; i++) {
        newPop.add(offSpring.get(i));
    }
    return newPop;
}
```

7.3 Crossover

La probabilità di crossover è del 80%.
Sono stati implementati 3 tipi di crossover:

7.3.1 *One Point*

```java
public static ArrayList<TeamRefactor> onePointCrossover(TeamRefactor team1, TeamRefactor team2) {
    TeamRefactor crossedTeam1 = new TeamRefactor();
    TeamRefactor crossedTeam2 = new TeamRefactor();
    ArrayList<TeamRefactor> toReturn = new ArrayList<>();
    ArrayList<DipendenteRefactor> son1 = new ArrayList<~>();
    ArrayList<DipendenteRefactor> son2 = new ArrayList<~>();
    ArrayList<DipendenteRefactor> dips1 = new ArrayList<~>();
    ArrayList<DipendenteRefactor> dips2 = new ArrayList<~>();
    dips1.addAll(team1.getDipendenti());
    dips2.addAll(team2.getDipendenti());
    if (new Random().nextDouble() < prob_crossover) {
        int randomOnePointCrossover = new Random().nextInt(dips1.size());
        for (int i = 0; i < randomOnePointCrossover; i++) {
            son1.add(dips1.get(i));
            son2.add(dips2.get(i));
        }
        for (int i = randomOnePointCrossover; i < dips2.size(); i++) {
            son1.add(dips2.get(i));
            son2.add(dips1.get(i));
        }
        crossedTeam1.setDipendenti(son1);
        crossedTeam2.setDipendenti(son2);
        toReturn.add(crossedTeam1);
        toReturn.add(crossedTeam2);
    } else {
        toReturn.add(team1);
        toReturn.add(team2);
    }
    return toReturn;
}
```

Il one point crossover sceglie casualmente un punto dell'individuo da cui effettuare lo scambio dei geni.

7.3.2 *Two Point*

```java
public static ArrayList<TeamRefactor> twoPointCrossover(TeamRefactor team1, TeamRefactor team2) {
    TeamRefactor crossedTeam1 = new TeamRefactor();
    TeamRefactor crossedTeam2 = new TeamRefactor();
    ArrayList<TeamRefactor> toReturn = new ArrayList<>();
    ArrayList<DipendenteRefactor> son1 = new ArrayList<>();
    ArrayList<DipendenteRefactor> son2 = new ArrayList<>();
    ArrayList<DipendenteRefactor> dips1 = new ArrayList<>();
    ArrayList<DipendenteRefactor> dips2 = new ArrayList<>();
    dips1.addAll(team1.getDipendenti());
    dips2.addAll(team2.getDipendenti());
    if (new Random().nextDouble() < prob_crossover) {
        int randomFirstPoint = new Random().nextInt(dips1.size());
        int randomSecondPoint = new Random().nextInt(dips1.size());
        if (randomFirstPoint == team1.getDipendenti().size() - 1) {
            randomSecondPoint = team1.getDipendenti().size();
        }
        while (randomFirstPoint >= randomSecondPoint) {
            randomSecondPoint = new Random().nextInt(dips1.size());
        }
        for (int i = 0; i < randomFirstPoint; i++) {
            son1.add(dips1.get(i));
            son2.add(dips2.get(i));
        }
        for (int j = randomFirstPoint; j < randomSecondPoint; j++) {
            son1.add(dips2.get(j));
            son2.add(dips1.get(j));
        }
        for (int k = randomSecondPoint; k < team1.getDipendenti().size(); k++) {
            son1.add(dips1.get(k));
            son2.add(dips2.get(k));
        }
        crossedTeam1.setDipendenti(son1);
        crossedTeam2.setDipendenti(son2);
        toReturn.add(crossedTeam1);
        toReturn.add(crossedTeam2);
    } else {
        toReturn.add(team1);
        toReturn.add(team2);
    }
    return toReturn;
}
```

Il two point crossover sceglie casualmente due punti dell'individuo e scambia i geni compresi.

7.3.3 *Uniform*

```java
public static ArrayList<TeamRefactor> uniformCrossover(TeamRefactor team1, TeamRefactor team2) {
    TeamRefactor crossedTeam1 = new TeamRefactor();
    TeamRefactor crossedTeam2 = new TeamRefactor();
    ArrayList<TeamRefactor> toReturn = new ArrayList<>();
    ArrayList<DipendenteRefactor> son1 = new ArrayList<>();
    ArrayList<DipendenteRefactor> son2 = new ArrayList<>();

    ArrayList<DipendenteRefactor> dips1 = new ArrayList<>();
    ArrayList<DipendenteRefactor> dips2 = new ArrayList<>();
    dips1.addAll(team1.getDipendenti());
    dips2.addAll(team2.getDipendenti());

    if (new Random().nextDouble() < prob_crossover) {
        for (int i = 0; i < dips1.size(); i++) {
            double prob = new Random().nextDouble();
            if (prob < prob_uniform) {
                son1.add(dips2.get(i));
                son2.add(dips1.get(i));
            } else {
                son1.add(dips1.get(i));
                son2.add(dips2.get(i));
            }
        }
        crossedTeam1.setDipendenti(son1);
        crossedTeam2.setDipendenti(son2);
        toReturn.add(crossedTeam1);
        toReturn.add(crossedTeam2);
    } else {
        toReturn.add(team1);
        toReturn.add(team2);
    }
    return toReturn;
}
```

L'uniform crossover scambia i geni calcolandoli con la *prob_uniform* del 50%. In questo modo non abbiamo una perdita di informazione genetica nel caso fosse troppo alta, e allo stesso tempo non abbiamo una mancanza di diversità genetica nella popolazione.

7.4 Mutation

```java
public static TeamRefactor mutation(TeamRefactor team, ArrayList<TeamRefactor> popolazione) {
    TeamRefactor newTeam = team;
    double prob = new Random().nextDouble();
    TeamRefactor tmp = new TeamRefactor();
    int pos = new Random().nextInt(popolazione.size());
    tmp = popolazione.get(pos);
    if (prob < prob_mutation) {
        int posDip = new Random().nextInt(tmp.getDipendenti().size());
        if (!newTeam.getDipendenti().contains(tmp.getDipendenti().get(posDip))) {
            newTeam.getDipendenti().remove(posDip);
            newTeam.getDipendenti().add(posDip, tmp.getDipendenti().get(posDip));
        }
    }
    return newTeam;
}
```

L'operatore di mutazione ci permette di modificare un gene dell'individuo corrente con un gene di un individuo scelto casualmente dalla popolazione. Questo tipo di mutazione prende il nome di Random Resetting. È stato scelto questo tipo di mutation in quanto le altre non avrebbero dato risultati significativi nell'evoluzione della nostra generazione. Gli altri tipi di mutazione che sono stati analizzati sono:

- Bitflip
- Swap
- Scrumble
- Inversion

La probabilità di mutazione è del 60%.

7.5 Evaluate

```java
public static ArrayList<TeamRefactor> evaluate
        (ArrayList<TeamRefactor> population, ArrayList<String> skills) {
    ArrayList<TeamRefactor> toReturn = new ArrayList<>();
    double best = 5.0;
    for (TeamRefactor team : population) {
        team.calcolaFitness(skills);
        if ((avg_Skills - team.getValoreTeam()) < best) {
            toReturn.add(team);
            best = team.getValoreTeam();
        }
    }
    return toReturn;
}
```

L'obiettivo di questo operatore è quello di scegliere gli individui migliori per la generazione corrente e ritorna un insieme di individui il cui valore, sottratto a quello di *avg_Skills*, si avvicina di più allo zero. In questo modo garantiamo di non prendere individui peggiori di quello corrente.

8 OPERATORI SCELTI

Selection: K-Way Tournament.
Crossover: One Point Crossover con una probabilità dell'80%.
Mutation: La probabilità di mutation è del 60%.
Evaluate: L'evaluate ritorna l'insieme di invidui la cui differenza con *avg_Skills* sia più vicina a 0.
Si è dovuto trovare un compromesso sugli operatori di Selection e Crossover. La scelta è ricaduta però sulle implementazioni sopra citate, in quanto hanno dato i risultati migliori.

8.1 Test eseguiti a supporto della tesi

Sono stati eseguiti 10 test su ogni combinazione degli operatori:

8.1.1 K-Way Tournament - One Point

Punteggio medio: 4,93
Tempo medio: 12 secondi
Iterazioni:

(1) Valutazione: 5 in 11,499 secondi
(2) Valutazione: 4,67 in 20,711 secondi
(3) Valutazione: 5 in 10,814 secondi
(4) Valutazione: 5 in 11,108 secondi
(5) Valutazione: 4,67 in 21,500 secondi
(6) Valutazione: 5 in 10,854 secondi
(7) Valutazione: 5 in 9,900 secondi
(8) Valutazione: 5 in 10,197 secondi
(9) Valutazione: 5 in 11,24 secondi
(10) Valutazione: 5 in 10,379 secondi

8.1.2 K-Way Tournament - Two Point

Punteggio medio: 4,89
Tempo medio: 16 secondi
Iterazioni:

(1) Valutazione: 5 in 12,682 secondi
(2) Valutazione: 4,92 in 23,425 secondi
(3) Valutazione: 4,92 in 23,632 secondi
(4) Valutazione: 5 in 10,855 secondi
(5) Valutazione: 4,75 in 21,309 secondi
(6) Valutazione: 5 in 11,567 secondi
(7) Valutazione: 5 in 12,976 secondi
(8) Valutazione: 4,58 in 19,354 secondi
(9) Valutazione: 4,75 in 21,929 secondi
(10) Valutazione: 5 in 10,966 secondi

8.1.3 K-Way Tournament - Uniform

Punteggio medio: 4,85
Tempo medio: 14 secondi
Iterazioni:

(1) Valutazione: 5 in 11,486 secondi
(2) Valutazione: 4,92 in 23,799 secondi
(3) Valutazione: 5 in 11,136 secondi
(4) Valutazione: 4,75 in 20,842 secondi
(5) Valutazione: 5 in 10,828 secondi
(6) Valutazione: 5 in 9,452 secondi
(7) Valutazione: 4,33 in 21,742 secondi
(8) Valutazione: 4,75 in 24,837 secondi
(9) Valutazione: 5 in 10,125 secondi
(10) Valutazione: 4,75 in 21,211 secondi

8.1.4 Truncation - Uniform

Punteggio medio: 4,16
Tempo medio: 3,47 secondi
Iterazioni:

(1) Valutazione: 4,67 in 3,244 secondi
(2) Valutazione: 4,33 in 3,299 secondi
(3) Valutazione: 4,33 in 3,821 secondi
(4) Valutazione: 4,08 in 3,517 secondi
(5) Valutazione: 4,25 in 3,366 secondi
(6) Valutazione: 4,08 in 3,517 secondi
(7) Valutazione: 3,92 in 3,441 secondi
(8) Valutazione: 3,83 in 3,592 secondi
(9) Valutazione: 4,25 in 3,477 secondi
(10) Valutazione: 3,92 in 3,444 secondi

8.1.5 Truncation - One Point

Punteggio medio: 4,07
Tempo medio: 3,38 secondi
Iterazioni:

(1) Valutazione: 4,17 in 3,602 secondi
(2) Valutazione: 4,25 in 3,695 secondi
(3) Valutazione: 4,08 in 3,775 secondi
(4) Valutazione: 4,08 in 3,142 secondi
(5) Valutazione: 4,08 in 3,000 secondi
(6) Valutazione: 3,92 in 2,898 secondi
(7) Valutazione: 3,92 in 3,800 secondi
(8) Valutazione: 4,17 in 3,000 secondi
(9) Valutazione: 3,92 in 3,800 secondi
(10) Valutazione: 4,08 in 3,142 secondi

8.1.6 Truncation - Two Point

Punteggio medio: 4,13
Tempo medio: 3,66 secondi
La truncation, nonostante il tempo medio di esecuzione decisamente breve, non ha mai raggiunto l'ottimo.

8.1.7 Rank Selection - One Point

Punteggio medio: 3,63
Tempo medio: 16 secondi
Iterazioni:

(1) Valutazione: 4 in 16,315 secondi
(2) Valutazione: 4,08 in 16,182 secondi
(3) Valutazione: 3,58 in 16,383 secondi
(4) Valutazione: 3,42 in 15,310 secondi
(5) Valutazione: 3,75 in 16,799 secondi
(6) Valutazione: 3,5 in 16,323 secondi
(7) Valutazione: 3,75 in 15,434 secondi
(8) Valutazione: 2,92 in 14,639 secondi
(9) Valutazione: 3,58 in 15,624 secondi
(10) Valutazione: 3,75 in 15,955 secondi

8.1.8 Rank Selection - Two Point e Uniform Crossover

Dato che il tempo medio è simile a quello della K-Way Tournament Two Point ma riporta punteggi medi drasticamente minori, è stato considerato non necessario riportare i risultati dei test.

8.1.9 Roulette Wheel

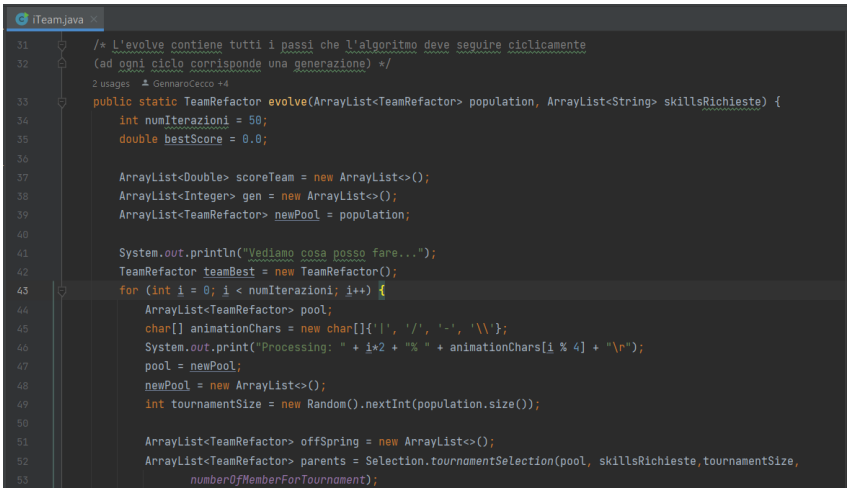
I test non sono stati riportati in quanto il punteggio e il tempo medio sono stati i peggiori fra tutti gli operatori, anche su un range di popolazione molto basso.

9 STOPPING CONDITION

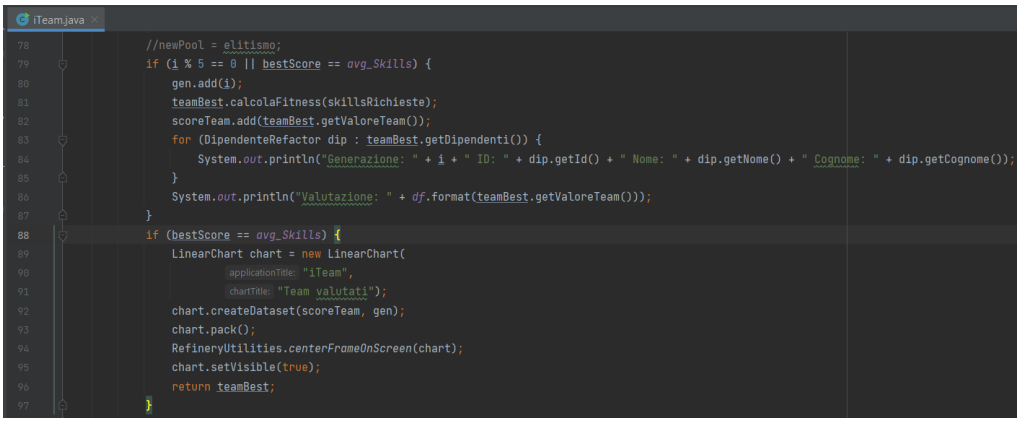
Le condizioni di arresto di iTeam sono due:

- Il numero totale di iterazioni: 50
- Il raggiungimento di una soluzione ottima

Vengono eseguite 50 iterazioni in quanto con un numero maggiore le soluzioni offerte dall'algoritmo non cambiano e quindi, utilizzando un numero maggiore di iterazioni, si ha soltanto un aumento del tempo di esecuzione senza miglioramenti nelle soluzioni proposte.



Per quanto riguarda la seconda stopping condition, abbiamo deciso di interrompere la ricerca all'algoritmo una volta trovata la soluzione ottima, in quanto l'unico fattore che potesse cambiare erano i dipendenti che avrebbero potuto far parte del team.



Il tempo di esecuzione per 50 iterazioni è di circa 15 secondi sull'intero dataset (se la dimensione del dataset diminuisce, anche il tempo di esecuzione lo farà). È stato scelto come trade-off quello della qualità del valore di ritorno dell'algoritmo, a discapito del vincolo descritto nel design goal, "tempo di risposta", di Agency Formation, perchè abbiamo assunto che il tempo di risposta del nostro algoritmo non sia tanto elevato per il compito che svolge, in quanto in assenza di esso i tempi per la creazione dei team sarebbero nettamente maggiori.