Smart Planner

Silvana Cafaro, Gennaro Foglia, Isabella Maria Sessa
Gruppo GennISilv

16 Febbraio 2024

Contents

1 Introduzione

Smart Planner è un progetto congiunto tra Fondamenti di Intelligenza Artificiale e Ingegneria del Software per la realizzazione di una agenda digitale. La componente "intelligente" è rappresentata da un tool che organizza automaticamente gli impegni dell'utente nel corso di una settimana. L'utente deve inoltre specificare una priorità per ogni impegno, per fornire all'algoritmo un'indicazione su quanto sia importante che quell'impegno venga inserito in fasce orarie considerate più produttive. Secondo un sondaggio effettuato da Slack su oltre 10.000 utenti, le ore meno produttive della giornata sarebbero quelle tra le 15:00 e le 18:00. Dallo stesso sondaggio è emerso anche che le fasce orarie considerate migliori sono quelle della mattina e della sera.
Abbiamo utilizzato queste informazioni come guida nell'ottimizzazione della nostra soluzione, come approfondiremo in seguito.

2 Obiettivi

L'obiettivo del progetto è riuscire a ottimizzare la distribuzione degli impegni nel corso di una settimana, in modo che gli impegni che l'utente considera importanti (alta priorità) siano il più possibile assegnati alle fasce orarie produttive piuttosto che a quelle sfavorevoli.

3 Definizione del problema

L'utente che interagisce con il tool inserisce una serie di impegni, specificando il nome e un valore di priorità da 1 a 5. Come accennato in precedenza abbiamo scelto di considerare le tre ore dalle 15 alle 18 come altamente sfavorevoli per un qualsiasi impegno, mentre le ore dalle 9 alle 11 e dalle 18 alle 20 come ottime scelte per schedularne uno. Ogni giorno è diviso in 12 fasce orarie di un'ora dalle 8 alle 20.

3.1 Specifica PEAS

3.1.1 P = performance

La prestazione dell'algoritmo è valutata come il punteggio della soluzione che restituisce. Nello specifico il punteggio di ogni soluzione è calcolato attraverso la funzione:

- (somma totale delle priorità degli impegni) - (somma totale delle priorità degli impegni schedulati nelle fasce 9-11 e 18-20) + (somma totale delle priorità degli impegni schedulati nella fasce 15-18).

Quindi un punteggio minore equivale a un risultato migliore.

3.1.2 E = ambiente

- **Agente singolo:** nell'ambiente esiste un solo agente con lo scopo di generare soluzioni migliori.
- **Completamente osservabile:** l'agente conosce la priorità e il nome di ogni impegno in qualsiasi momento.
- **Stocastico:** A causa di una componente casuale, lo stato successivo non è completamente dipeso dallo stato corrente e dalle scelte dell'agente.
- **Episodico:** L'esperienza dell'agente è divisa in "episodi" atomici, dove ciascun episodio consiste nell'eseguire una singola azione. La scelta dell'azione in ciascun episodio dipende dall'episodio stesso.
- **Statico:** Mentre l'agente delibera, la lista di impegni non può essere modificata.
- **Discreto:** L'ambiente fornisce un numero limitato di percezioni e azioni distinte, chiaramente definite

3.1.3 A = attuatori

Mostra a video uno scheduling possibile per gli impegni inseriti.

3.1.4 S = sensori

Input dall'utente: gli impegni, quindi nome e priorità

4 Soluzione del problema

4.1 Descrizione della soluzione

Dopo aver riflettuto sulla formulazione del problema, siamo giunti alla conclusione che un algoritmo di ottimizzazione fosse la scelta migliore.
Abbiamo deciso di non fermarci ad un singolo algoritmo, ma di esplorare tutte e tre le maggiori soluzioni viste a lezione: algoritmo Simulated Annealing, algoritmo Hill Climbing e Algoritmi Genetici. Di seguito riportiamo la modellazione e alcuni passi dell'implementazione di ognuno di essi. Infine delle considerazioni sulle loro performance e la scelta di quale di essi verrà effettivamente implementato nel progetto di Ingegneria del Software.

4.2 Sviluppo

Tutte le soluzioni sono state implementate in linguaggio Java. Abbiamo preso questa scelta perché è il linguaggio con cui abbiamo avuto di più a che fare nella nostra carriera.
Per lo sviluppo degli algoritmi abbiamo lavorato con l'IDE Intellij di JetBrains, anche in questo caso per motivi di praticità e abitudine.
Non sono state usate librerie o particolari framework specifici, ma è stato fatto un ampio uso degli esempi forniti dal docente e dai tutor.

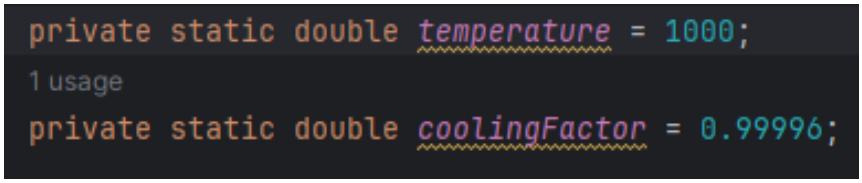
4.3 Classi comuni

Per tutti gli algoritmi ci siamo serviti di tre classi: Impegno, Giorno e Settimana. Esse modellano tre entità del nostro problema e forniscono i metodi necessari all'implementazione degli algoritmi. Un impegno è, come detto in precedenza, un'entità con un nome e una priorità. Un giorno è una collezione di impegni, che può restituire un impegno in una determinata fascia oraria e il proprio punteggio della funzione obiettivo. Una settimana è una collezione di 7 giorni che consente l'accesso a una determinata fascia oraria di un giorno e può restituire il punteggio totale della funzione obiettivo. I nostri algoritmi operano con gli oggetti di tipo Settimana, andando a modificare le posizioni degli impegni tra i giorni.
Infine tutti gli algoritmi sono stati testati con un set di 28 impegni, tale per cui l'ottimo globale è uguale a 0.

4.4 Simulated Annealing

L'obiettivo dell'algoritmo Simulated Annealing è navigare lo spazio degli stati il più possibile in una fase iniziale, riducendo sempre di più la zona di ricerca mano a mano che passa il tempo. Il concetto è lo stesso della tecnica annealing della metallurgia e, nel caso degli algoritmi di ricerca locale, punta ad risolvere il problema dei massimi locali senza sfociare nell'esplorazione totalmente casuale.

Il primo passo quindi è definire una "temperatura iniziale" e una "velocità di raffreddamento" che all'atto pratico influenzeranno il numero di iterazioni dell'algoritmo e la probabilità con cui soluzioni sfavorevoli verranno accettate, in virtù della necessità di "scendere a valle" per evitare plateau, spalle e massimi locali.
Il nostro valore iniziale per la temperatura era di 1000, mentre il fattore che moltiplicato per la temperatura da la velocità di raffreddamento era 0.995. Questa combinazione portava a una soluzione accettabile in meno di un decimo di secondo. Potendo permetterci di aspettare più tempo, abbiamo diminuito il fattore fino a raggiungere un compromesso tra il tempo di esecuzione e la bontà delle soluzioni prodotte. Tale compromesso è quello riportato in immagine, che mantiene l'esecuzione sotto i 3 secondi, come da requisito non funzionale del progetto di Ingegneria del Software, e produce soluzioni più che accettabili. Va detto che abbiamo provato anche combinazioni che concedessero all'algoritmo più tempo, ma nessuna comportava un serio aumento delle performance; quindi questa è risultata come la scelta migliore.



Di seguito riportiamo l'implementazione dell'algoritmo.

```
public static void main(String[] args){
    Settimana attuale = createSettimana();
    Settimana best = attuale.clona();
    int i = 0;

    long inizio = System.currentTimeMillis();

    for(double t = temperature; t > 1; t *= coolingFactor, i++){
        //creo un successore andando a scambiare di posizione in maniera casuale due impegni
        Settimana vicino = attuale.clona();

        String index1 = getRandomIndex();
        String index2 = getRandomIndex();
        vicino.swap(index1,index2);

        //ottengo i due valori di score
        int score1 = attuale.getTotalScore();
        int score2 = vicino.getTotalScore();

        //calcolo la probabilità di accettare la nuova soluzione e in caso la accetto
        if(Math.random() < probability(score1, score2, t))
            attuale = vicino.clona();

        if(attuale.getTotalScore() < best.getTotalScore())
            best = attuale.clona();
    }

    long fine = System.currentTimeMillis();
```

Possiamo escludere le variabili *i*, *inizio* e *fine* che tengono traccia del numero di iterazioni del ciclo e del tempo di esecuzione, ma che non sono necessarie per il funzionamento della soluzione.
Come si può vedere, il primo passo è generare in maniera casuale uno stato iniziale. Una volta istanziato anche lo stato migliore, che all'inizio coincide con quello iniziale, si può avviare il ciclo. Nel ciclo viene generato un successore dello stato corrente andando a scambiare di posizione casualmente due impegni. Si calcolano per entrambi gli stati i valori della funzione obiettivo e il metodo *probability* restituisce la probabilità di accettazione del successore. Se il valore del successore è minore di quello dello stato attuale allora la probabilità di accettazione è 1, altrimenti è $e^{(\frac{-\Delta E}{t})}$ con ΔE che rappresenta la differenza tra il valore dello stato attuale e quello del successore. La t infine rappresenta la temperatura attuale. L'ultimo step del ciclo confronta il valore dello stato attuale con quello dello stato migliore per valutare se si è trovato lo stato con il miglior valore fino a quel momento.

Per quanto riguarda le performance dell'algoritmo, su 10 esecuzioni abbiamo avuto i seguenti risultati.

N° esecuzione	Valore migliore
1	26
2	30
3	26
4	26
5	29
6	26
7	26
8	26
9	25
10	30
Media	27

4.5 Hill Climbing

Per quanto riguarda Hill Climbing, è un algoritmo che punta a restituire molto velocemente una soluzione "scalando" lo spazio degli stati. All'atto pratico prevede di generare vicini dello stato attuale e spostarsi in quello con il valore migliore; se non ne esistono allora si ferma e restituisce lo stato corrente. Abbiamo testato questa implementazione, ma i risultati erano troppo poco convincenti, quindi abbiamo deciso di apportare delle modifiche.

```
public static void main(String[] args){
    Settimana attuale = createSettimana();
    int i = 0;
    boolean continua = true;

    long inizio = System.currentTimeMillis();
    while(continua){
        i++;

        //creo un vicino andando a scambiare di posizione in maniera casuale due impegni
        Settimana vicino = attuale.clona();
        String index1 = getRandomIndex();
        String index2 = getRandomIndex();
        vicino.swap(index1,index2);

        if(attuale.getTotalScore() >= vicino.getTotalScore()) {
            attuale = vicino.clona();
        }
        //per altre 1000 volte cerca di trovare un vicino migliore, altrimenti ferma la ricerca
        else{
            int j;
            for(j = 0; j < 1000; j++){
                Settimana vicino2 = attuale.clona();
                index1 = getRandomIndex();
                index2 = getRandomIndex();
                vicino2.swap(index1,index2);
                if(attuale.getTotalScore() >= vicino.getTotalScore()) {
                    attuale = vicino.clona();
                    break;
                }
            }
            if(j == 1000)
                continua = false;
        }
    }
    long fine = System.currentTimeMillis();
```

Per le variabili *i*, *inizio* e *fine* valgono le stesse considerazioni dell'algoritmo Simulated Annealing.
Anche in questo caso un vicino viene generato andando a scambiare casualmente due impegni dello stato attuale, ma l'algoritmo non si ferma nel momento in cui genera un vicino con un valore della funzione obiettivo peggiore: continua per almeno altre mille volte la procedura di generazione di un vicino, nella speranza di spostarsi dalla situazione di stallo. Se nemmeno questa ricerca riesce, allora lo stato attuale è la soluzione, altrimenti la ricerca continua dal nuovo stato attuale migliore del precedente. Questa soluzione ha apportato dei benefici, purtroppo non significativi quanto sperato.
Di seguito riportiamo le performance dell'algoritmo su 10 esecuzioni.

N° esecuzione	Valore migliore
1	79
2	74
3	64
4	45
5	72
6	84
7	67
8	77
9	73
10	63
Media	69,8

4.6 Genetic Algorithm

Gli algoritmi genetici (GA) sono una meta-euristica che consente di definire algoritmi di ricerca. Essendo ispirati alla genetica, come suggerisce il nome, si propongono di evolvere una popolazione di individui producendo di volta in volta soluzioni sempre migliori rispetto alla funzione obiettivo.
Il normale ciclo di ricerca di un GA prevede

1 l'inizializzazione della popolazione;

2 la valutazione di ogni individuo secondo la funzione obiettivo, che d'ora in poi chiameremo funzione di fitness;

3 la selezione di alcuni individui della popolazione, detti mating pool, mediante un operatore di selezione;

4 l'accoppiamento di questi individui al fine di generarne di nuovi, chiamati offsprings, che abbiano i geni di quelli di partenza mischiati mediante un operatore di crossover;

5 la mutazione casuale dei nuovi individui al fine di aumentare la porzione di spazio degli stati esplorata dall'algoritmo, mediante un operatore di mutazione;

6 ricominciare dal punto 2 fino al raggiungimento di una stopping condition, o search budget.

Oltre alla scelta degli operatori e alla definizione della funzione di fitness e del search budget, esistono altri parametri in grado di alterare il comportamento di un GA, quali: la dimensione della popolazione iniziale, la dimensione del mating pool, la probabilità di mutazione, la probabilità di crossover, il numero di individui che partecipano al crossover. Inoltre la stessa codifica degli individui è un aspetto da non sottovalutare.
Consapevoli di questa varietà e motivati come detto a tentare soluzioni diverse, abbiamo progettato il nostro GA di conseguenza. Di seguito approfondiremo ogni aspetto della progettazione, senza entrare troppo nell'ambito dell'implementazione per questioni di sinteticità e leggibilità del documento. Ad ogni modo in fondo allo stesso è presente il link alla repository GitHub che contiene tutti gli algoritmi che stiamo trattando.

4.6.1 Funzione di fitness

Come funzione di fitness abbiamo scelto di considerare la stessa funzione obiettivo utilizzata per i precedenti algoritmi.

4.6.2 Codifica degli individui

Per questioni di semplicità abbiamo utilizzato le stesse classi utilizzate dai precedenti algoritmi: Impegno, Giorno e Settimana. Abbiamo apportato delle leggere modifiche alla classe Settimana per poterla adattare meglio all'implementazione in un GA.

4.6.3 Inizializzazione degli individui

La popolazione iniziale, di dimensione che può essere variata agendo su una specifica variabile del codice, viene inizializzata creando degli oggetti di tipo Settimana in cui gli impegni da schedulare sono posizionati in modo casuale. Un individuo è ammissibile solo se contiene tutti gli impegni, ciascuno ripetuto una sola volta.

4.6.4 Algoritmo di selezione

Abbiamo implementato due algoritmi di selezione: Roulette Wheel e Truncation. Nel primo gli individui ricevono una probabilità di selezione pari al valore della loro fitness relativa all'intera popolazione; nel secondo si compie un ordinamento degli individui in base al valore di fitness e si selezionano gli M individui migliori. L'algoritmo Roulette Wheel accetta che un individuo sia selezionato più volte, Truncation invece no. Il parametro M può essere modificato agendo sul valore di una costante nel codice.

4.6.5 Algoritmo di crossover

In questo caso abbiamo immaginato di vedere gli individui come matrici di 12 righe (le fasce orarie) e 7 colonne (i giorni). Il crossover tra due individui avviene dividendoli in due orizzontalmente a una riga scelta in modo casuale e mischiando le 4 parti ottenute con la formula: superiore del primo con inferiore del secondo e viceversa. Vengono così generati due individui successori dei precedenti. L'algoritmo evita la duplicazione di impegni e reinserisce in posizioni casuali quelli che dovessero essere persi nell'operazione di divisione.
Va detto che questa soluzione aggiunge una componente di casualità dipendente dalla configurazione dei genitori. Infine il crossover avviene strettamente tra due individui accoppiati in maniera casuale; in caso di mating pool di dimensione dispari viene tagliato l'ultimo elemento.

4.6.6 Algoritmo di mutazione

I due algoritmi di mutazione implementati sono Swap e Scramble. Swap scambia di posizione due eventi scelti casualmente, Scramble ne seleziona un certo numero e li permuta casualmente, per poi reinserirli nelle posizioni lasciate vuote. Il numero di impegni scelti da Scramble è modificabile nel codice agendo sul valore di una costante.

4.6.7 Search budget

Il nostro search budget è rappresentato da due variabili che indicano il numero massimo di generazioni da generare e il numero di generazioni da generare senza che ci siano miglioramenti, prima di fermarsi. Entrambe sono modificabili facilmente.

Adesso riportiamo le performance dell'algoritmo in base ad alcune configurazioni di operatori e parametri.
Glossario:

- RW = Roulette Wheel;
- TR = Truncation;
- SW = Swap;
- SC = Screamble;
- size = numero di individui selezionati nel caso di TR, numero di impegni permutati nel caso di SC.

Size popolazione	Probabilità mutazione		N° esecuzione	Valore migliore
1	1		1	40
			2	42
Selection	Mutation		3	46
RW	SW		4	41
			5	40
			6	42
			7	46
			8	33
			9	50
			10	38
			Media	41,8

1

Size popolazione	Probabilità mutazione		N° esecuzione	Valore migliore
5	1		**1**	47
			2	47
Selection	**Mutation**		**3**	38
RW	SW		**4**	40
			5	48
			6	37
			7	38
			8	46
			9	38
			10	37
			Media	41,6

2

Size popolazione	Probabilità mutazione		N° esecuzione	Valore migliore
1	1		**1**	40
			2	35
Selection	**Mutation**		**3**	37
RW	SC size 8		**4**	39
			5	39
			6	42
			7	27
			8	36
			9	42
			10	44
			Media	38,1

3

Size popolazione	Probabilità mutazione		N° esecuzione	Valore migliore
5	1		**1**	38
			2	34
Selection	**Mutation**		**3**	41
RW	SC size 8		**4**	42
			5	41
			6	38
			7	45
			8	33
			9	42
			10	44
			Media	39,8

4

Size popolazione	Probabilità mutazione		N° esecuzione	Valore migliore
1	1		**1**	31
			2	29
Selection	**Mutation**		**3**	34
TR size 2	SW		**4**	32
			5	24
			6	28
			7	34
			8	32
			9	33
			10	28
			Media	30,5

Size popolazione	Probabilità mutazione		N° esecuzione	Valore migliore
3	1		**1**	32
			2	31
Selection	**Mutation**		**3**	33
TR size 3	SW		**4**	30
			5	37
			6	24
			7	35
			8	31
			9	27
			10	29
			Media	30,9

Size popolazione	Probabilità mutazione		N° esecuzione	Valore migliore
1	1		**1**	31
			2	27
Selection	**Mutation**		**3**	31
TR size 2	SC size 3		**4**	27
			5	31
			6	37
			7	29
			8	27
			9	29
			10	35
			Media	30,4

Size popolazione	Probabilità mutazione		N° esecuzione	Valore migliore
5	1		1	30
			2	30
Selection	Mutation		3	28
TR size 3	SC size 3		4	32
			5	29
			6	33
			7	27
			8	33
			9	32
			10	25
			Media	29,9

8

Come visibile abbiamo deciso di riportare prove in cui a variare sono unicamente la size della popolazione, la combinazione di operatore di selezione e di mutazione e dei loro parametri. La probabilità di mutazione, il numero di iterazioni massime e il numero di iterazioni senza miglioramenti massime sono rimasti invariati rispettivamente con valori 1, 1000 e 0. Il motivo di queste scelte è che nel corso della sperimentazione ci siamo resi conto che variare quei parametri aveva pochi effetti migliorativi e al contrario spesso portava a un peggioramento delle performance, quindi abbiamo stabilito dei punti fermi e ricominciato a valutare l'algoritmo da lì.
Inoltre anche aumentare a due cifre la size della popolazione non ha mai portato benefici, quindi i test riportati sono limitati a size unitarie e molto piccole.
La combinazione migliore è risultata essere quella che sfrutta Truncation a 3 individui, Scramble di 3 impegni e ha una size della popolazione di 5, riportata nella figura 8. Nonostante ciò i risultati delle figure 5, 6 e 7 sono anch'essi altrettanto ottimi, con medie che differiscono di pochi decimali da quella della figura 8.
Sono invece molto più alte le medie dei test in figura 1, 2, 3 e 4, che hanno in comune l'utilizzo dell'operatore di selezione Roulette Wheel. Possiamo quindi concludere che il suo utilizzo deteriora le performance dell'algoritmo.

4.7 Confronto

Iniziamo il confronto riportando i tre valori medi dei test effettuati dagli algoritmi:

- **Simulated Annealing** = 27
- **Hill Climbing** = 68,8
- **Genetic Algorithm** = 29,9

Quello che emerge è un netto sconfitto, con gli altri due molto vicini tra di loro. Partendo dallo sconfitto possiamo dire che ci aspettavamo i risultati peggiori, ma il valore così diverso dalla media degli altri due ci fa pensare di poter ancora migliorare qualcosa sul piano della strategia di ricerca. Del resto Hill Climbing resta di gran lunga l'algoritmo più rapido a giungere a una soluzione, quindi in un caso in cui il tempo fosse un fattore importante allora varrebbe la pena continuare a cercare di perfezionarlo e trovare un compromesso migliore tra tempi di esecuzione e bontà della soluzione.
Simulated Annealing e il GA sono molto vicini tra di loro in termini di punteggio, ma hanno due sostanziali differenze. La prima è nei tempi di esecuzione: il miglior compromesso che abbiamo trovato per l'algoritmo SA rispetta perfettamente il nostro requisito di restituire una risposta in meno di 3 secondi, ma l'algoritmo genetico porta a una media di pochissimo superiore in un decimo del tempo.
La seconda differenza è che c'è ancora una lunga lista di operatori e configurazioni per il GA che non abbiamo provato e con una più attenta selezione degli operatori e dei parametri, frutto di uno studio più approfondito e di ulteriori test, probabilmente le sue prestazioni potrebbero migliorare e non di poco; di contro il nostro algoritmo SA non ha migliorato le sue performance nemmeno variando il tempo di ricerca con quell'obiettivo e non può contare sulla lunga lista di combinazioni che rendono gli algoritmi genetici così flessibili.
La nostra conclusione è che l'algoritmo migliore per il nostro progetto sia Simulated Annealing, che produce i migliori risultati in assoluto rispettando a pieno il requisito del tempo di risposta. Ciononostante in condizioni diverse la scelta migliore potrebbe essere continuare a sviluppare l'algoritmo genetico, a scapito anche dell'algoritmo Hill Climbing, che risulta conveniente solo con forti restrizioni sui tempi di esecuzione.
In questa repository GitHub è stato caricato il progetto che contiene tutto il codice scritto da noi per implementare le soluzioni, oltre alle istruzioni per riprodurre i nostri test.

4.8 Conclusione

Produrre questo progetto è stata un'esperienza stimolante e appagante. Lo studio degli algoritmi di ricerca, che hanno interessato il gruppo GennISilv fin dalla prima lezione dedicata, ci ha aiutato nella realizzazione dell'applicazione Smart Planner nel modo migliore secondo noi, e ci ha permesso di applicare le nozioni apprese dalla teoria in un modo molto divertente.