ITC-2007: Curriculum-based Course Timetabling

Stefano Dottore

7 febbraio 2021

Sommario

Questa relazione descrive l'analisi e la progettazione di un algoritmo facente uso di diverse metaeuristiche per la risoluzione della Traccia 3 (*Curriculum Course Timetabling*) della competizione International Timetabling Competition 2007 [1].

1 Introduzione

Nelle sezioni che seguono sono descritte la traccia del problema, il modello formale dedotto da essa, un algoritmo per la costruzione di una soluzione iniziale ammissibile, la descrizione del neighbourhood adottato e l'implementazione di diverse metaeuristiche che ne facessero uso per fornire una soluzione di costo ridotto.

Infine, i risultati forniti dall'algoritmo sviluppato sono stati confrontati con i risultati ottenuti dai partecipanti dell'ITC2007.

2 Traccia

Il problema *Curriculum Course Timetabling* è riportato integralmente nei paragrafi che seguono e può essere così riassunto:

Dato un insieme di corsi C, per ogni corso $c \in C$ è definito un numero di lezioni l_c che devono essere assegnate fra le varie aule $r \in R$ nei diversi periodi della settimana, definiti dalla coppia ($d \in D$, $s \in S$). Una soluzione al problema consiste nel trovare un assegnamento (r, d, s) per ogni lezione di ogni corso in modo da non violare nessuno degli *Hard Constraints* imposti dalla traccia (H1, H2, H3, H4), minimizzando il costo introdotto dai *Soft Constraints* (S1, S2, S3, S4).

Si riporta di seguito la traccia completa:

The Curriculum-based timetabling problem consists of the weekly scheduling of the lecturesEntità for several university courses within a given number of rooms and time periods, where conflicts between courses are set according to the curricula published by the University and not on the basis of enrolment data.

2.1 Entities

Days, Timeslots, and Periods. We are given a number of teaching days in the week (typically 5 or 6). Each day is split in a fixed number of timeslots, which is equal for all days. A period is a pair composed of a day and a timeslot. The total number of scheduling periods is the product of the days times the day timeslots.

Courses and Teachers. Each course consists of a fixed number of lectures to be scheduled in distinct periods, it is attended by given number of students, and is taught by a teacher. For each course there is a minimum number of days that the lectures of the course should be spread in, moreover there are some periods in which the course cannot be scheduled.

Rooms. Each room has a capacity, expressed in terms of number of available seats. All rooms are equally suitable for all courses (if large enough).

Curricula. A curriculum is a group of courses such that any pair of courses in the group have students in common. Based on curricula, we have the conflicts between courses and other soft constraints.

The solution of the problem is an assignment of a period (day and timeslot) and a room to all lectures of each course.

2.2 Hard Constraints

H1: Lectures: All lectures of a course must be scheduled, and they must be assigned to distinct periods. A violation occurs if a lecture is not scheduled.

H2: RoomOccupancy: Two lectures cannot take place in the same room in the same period. Two lectures in the same room at the same period represent one violation . Any extra lecture in the same period and room counts as one more violation.

H3: Conflicts: Lectures of courses in the same curriculum or taught by the same teacher must be all scheduled in different periods. Two conflicting lectures in the same period represent one violation. Three conflicting lectures count as 3 violations: one for each pair.

H4: Availabilities: If the teacher of the course is not available to teach that course at a given period, then no lectures of the course can be scheduled at that period. Each lecture in a period unavailable for that course is one violation.

2.3 Soft Constraints

S1: RoomCapacity: For each lecture, the number of students that attend the course must be less or equal than the number of seats of all the rooms that host its lectures. Each student above the capacity counts as 1 point of penalty.

S2: MinimumWorkingDays: The lectures of each course must be spread into the given minimum number of days. Each day below the minimum counts as 5 points of penalty.

S3: CurriculumCompactness: Lectures belonging to a curriculum should be adjacent to each other (i.e., in consecutive periods). For a given curriculum we account for a violation every time there is one lecture not adjacent to any other lecture within the same day. Each isolated lecture in a curriculum counts as 2 points of penalty.

S4: RoomStability: All lectures of a course should be given in the same room. Each distinct room used for the lectures of a course, but the first, counts as 1 point of penalty.

3 Modello

Il primo passo compiuto in seguito all'analisi della traccia è stata la definizione formale del problema mediante modello matematico. Seppure non strettamente necessario allo sviluppo del risolutore facente uso di metaeuristiche, definire un modello formale ha consentito di acquisire una maggiore conoscenza del problema, permettendo di sviluppare l'algoritmo in modo più agevole.

A scopo didattico e per verificarne la correttezza, il modello è stato anche implementato mediante l'ottimizzatore *Gurobi*. Come prevedibile, data l'elevata complessità del problema, *Gurobi* è stato in grado di fornire la soluzione ottima in tempo ragionevole solo per le istanze più facili del problema (`toy` e `comp01`); questo giustifica lo sviluppo di un risolutore non esatto.

3.1 Simboli

I seguenti simboli rappresentano parametri di input del modello.

C	insieme dei corsi
R	insieme delle aule
D	insieme dei giorni
S	insieme degli slot per giorno
T	insieme degli insegnanti
Q	insieme dei curriculum
L	insieme delle lezioni di tutti i corsi

c_l	corso associato alla lezione l
t_c	insegnante del corso c
l_c	numero di lezioni del corso c
n_c^C	numero di studenti frequentanti il corso c
n_r^R	capienza dell'aula r
mwd_c	minimo numero di giorni su cui il corso c dovrebbe essere distribuito

$$b_{cq} = \begin{cases} 1 & \text{se il corso } c \text{ appartiene al curriculum } q \\ 0 & \text{altrimenti} \end{cases}$$

$$e_{ct} = \begin{cases} 1 & \text{se il corso } c \text{ è insegnato dall'insegnante } t \\ 0 & \text{altrimenti} \end{cases}$$

$$a_{cds} = \begin{cases} 1 & \text{se il corso } c \text{ può essere assegnato nel giorno } d, \text{ slot } s \\ 0 & \text{altrimenti} \end{cases}$$

α_1	penalità del *Soft Contraint* S1 = 1
α_2	penalità del *Soft Contraint* S2 = 5
α_3	penalità del *Soft Contraint* S3 = 2
α_4	penalità del *Soft Contraint* S4 = 1

3.2 Variabili

Per la definizione del modello sono state utilizzate le seguenti variabili.
(solo x_{crds} è effettivamente una variabile libera).

$$x_{crds} = \begin{cases} 1 & \text{se il corso } c \text{ è assegnato all'aula } r \text{ nel giorno } d, \text{ slot } s \\ 0 & \text{altrimenti} \end{cases}$$

$$y_{cd} = \begin{cases} 1 & \text{se } \sum_{r \in R} \sum_{s \in S} x_{crds} > 0 \\ 0 & \text{altrimenti} \end{cases}$$

(il corso c è assegnato almeno una volta il giorno d)

$$u_{qds} = \sum_{c \in C} \sum_{r \in R} x_{crds} \cdot b_{cq}$$

(numero di corsi appartenenti al curriculum q assegnati nel giorno d, slot s)

$$w_{qds} = \begin{cases} 1 & \text{se } u_{qds} > 0 \\ 0 & \text{altrimenti} \end{cases}$$

(il curriculum q ha almeno un corso assegnato nel giorno d, slot s)

$$z_{cr} = \begin{cases} 1 & \text{se } \sum_{d \in D} \sum_{s \in S} x_{crds} > 0 \\ 0 & \text{altrimenti} \end{cases}$$

(l'aula r è usata almeno una volta per il corso c)

3.3 Vincoli

I seguenti vincoli sono *Hard Constraints*, e non possono essere violati.

- **H1: Lectures (a)**
  Le lezioni assegnate per il corso c devono essere l_c

$$\sum_{r \in R} \sum_{d \in D} \sum_{s \in S} x_{crds} = l_c \qquad c \in C$$

- **H1: Lectures (b)**
  Al più un aula può essere usata per il corso c nel giorno d, slot s

$$\sum_{r \in R} x_{crds} \leq 1 \qquad c \in C,\ d \in D,\ s \in S$$

- **H2: RoomOccupancy**
  Al più un corso può essere assegnato all'aula r nel giorno d, slot s

$$\sum_{c \in C} x_{crds} \leq 1 \qquad r \in R,\ d \in D,\ s \in S$$

- **H3: Conflicts (a)**
  Al più un corso appartenente ad un curriculum q può essere assegnato nel giorno d, slot s

$$\sum_{c \in C} \sum_{r \in R} x_{crds} \cdot b_{cq} \leq 1 \qquad d \in D,\ s \in S,\ q \in Q$$

- **H3: Conflicts (b)**
  Al più un corso insegnato dall'insegnante t può essere assegnato nel giorno d, slot s

$$\sum_{c \in C} \sum_{r \in R} x_{crds} \cdot e_{ct} \leq 1 \qquad d \in D,\ s \in S,\ t \in T$$

- **H4: Availabilities**
  Il corso c può essere assegnato nel giorno d, slot s solo se è possibile farlo, ossia se non c'è un vincolo di non disponibilità per la terna (c, d, s).

$$\sum_{r \in R} x_{crds} \leq a_{cds} \qquad c \in C,\ d \in D,\ s \in S$$

3.4 Funzione obbiettivo

La funzione obbiettivo f da minimizzare rispetto ad una soluzione X è la somma pesata dei *Soft Constraints*.

$$\min f(X)$$

$$f(X) = \alpha_1 \sum_{c \in C} \sum_{r \in R} \sum_{d \in D} \sum_{s \in S} S1(c, r, d, s) + \alpha_2 \sum_{c \in C} S2(c) + \\ + \alpha_3 \sum_{q \in Q} \sum_{d \in D} \sum_{s \in S} S3(q, d, s) + \alpha_4 \sum_{c \in C} S4(c)$$

- **S1: RoomCapacity**

  Se un corso c assegnato ad un'aula r nel giorno d, slot s, ha un numero di studenti superiore ai posti dell'aula, è introdotta una penalità corrispondente al numero di studenti in eccesso.

$$S1(c, r, d, s) = (x_{crds} \cdot (n_c^C - n_r^R)) \cdot ind_1(c, r)$$

$$ind_1(c, r) = \begin{cases} 1 & \text{se } (n_c^C - n_r^R) > 0 \\ 0 & \text{altrimenti} \end{cases}$$

- **S2: MinimumWorkingDays**

  Se le lezioni del corso c non sono distribuite in almeno mwd_c giorni, è introdotta una penalità corrispondente alla differenza fra il numero di giorni richiesto e il numero di giorni su cui le lezioni sono effettivamente distribuite.

$$S2(c) = (mwd_c - \sum_{d \in D} y_{cd}) \cdot ind_2(c)$$

$$ind_2(c) = \begin{cases} 1 & \text{se } (mwd_c - \sum_{d \in D} y_{cd}) > 0 \\ 0 & \text{altrimenti} \end{cases}$$

- **S3: CurriculumCompactness**

  Se le lezioni dei corsi appartenenti al curriculum q assegnate nel giorno d, slot s, non hanno lezioni dello stesso curriculum q negli slot adiacenti, è introdotta una penalità per ognuna di esse.

$$S3(q, d, s) = u_{qds} \cdot ind_3(q, d, s)$$

$$ind_3(q, d, s) = \begin{cases} 1 & \text{se } w_{qds} \wedge \overline{w_{qds-1}} \wedge \overline{w_{qds+1}} \\ 0 & \text{altrimenti} \end{cases}$$

- **S4: RoomStability**

  Se il corso c ha lezioni assegnate in più di un'aula, una penalità è introdotta per ogni aula oltre la prima.

$$S4(c) = ((\sum_{r \in R} z_{cr}) - 1) \cdot ind_4(c)$$

$$ind_4(c) = \begin{cases} 1 & \text{se } ((\sum_{r \in R} z_{cr}) - 1) > 0 \\ 0 & \text{altrimenti} \end{cases}$$

4 Algoritmo

L'algoritmo sviluppato prevede innanzitutto la costruzione di una soluzione ammissibile (per la quale sono tenuti in considerazione solamente gli *Hard Constraints*) ed in seguito l'uso di una o più metaeuristiche che eseguendo mosse del *neighbourhood* consentono di minimizzare il costo $f(X)$ della soluzione.

4.1 Costruzione di una soluzione ammissibile

La prima fase dell'algoritmo consiste nella costruzione di una iniziale soluzione iniziale ammissibile (che rispetti gli *Hard Constraints* H1, H2, H3, H4).

La prima idea formulata per la costruzione di una soluzione ammissibile è un algoritmo *greedy* che consiste nell'assegnare, per ognuna delle l_c lezioni di ogni corso c, la prima terna (aula, giorno, slot) disponibile che non violi nessuno degli *Hard Constraints*, rimuovendo poi tale terna da quelle ancora assegnabili. Se una lezione non può essere assegnata a nessuna delle terne (aula, giorno, slot) rimanenti senza violare un *Hard Constraint*, il processo di costruzione fallisce.

Nella sua forma descritta, il processo di costruzione si è dimostrato fallire frequentemente per istanze medio/difficili. È stata quindi introdotta una modifica che ha consentito di fornire una soluzione ammissibile con alta probabilità: assegnare le lezioni l non in ordine prestabilito (e.g., nella prima idea erano assegnate secondo l'ordine con cui i corsi apparivano nell'istanza di input), ma secondo una *difficoltà di assegnamento* stimata con la seguente formula:

$$rank(l) = difficulty(c_l) = \#H3a(c_l) + \#H3b(c_l) + \#H4(c_l)$$

Dove:

$$\#H3a(c) = \sum_{q \in Q} b_{cq}$$

(numero di curriculum a cui il corso c appartiene)

$$\#H3b(c) = \sum_{c' \in C} e_{c' t_c}$$

(numero di corsi insegnati dall'insegnante t_c del corso c)

$$\#H4(c) = \sum_{d \in D} \sum_{s \in S} (1 - a_{cds})$$

(numero di non disponibilità del corso c)

Ordinando le lezioni dei corsi in modo decrescente rispetto al *rank* descritto, è stato ottenuto un costruttore di soluzione ammissibili con buone probabilità di successo (a seconda della difficoltà intrinseca dell'istanza). È stato tuttavia necessario apporta un'ultima modifica all'algoritmo di costruzione, principalmente per due ragioni:

- Nel caso in cui l'ordine di l'assegnamento di lezioni ritenuto più facile dovesse fallire, non ci sarebbe altro modo di costruire una soluzione ammissibile.
- Avendo implementato un meccanismo di *multistart* all'interno del risolutore, è sorta l'esigenza di produrre più soluzioni iniziali ammissibili diverse l'una dall'altra, in modo da coprire quanto più possibile lo spazio delle soluzioni.

Tali esigenze si traducono nell'introduzione di un semplice meccanismo di *randomizzazione* dell'ordine di assegnamento delle lezioni; questo non può tuttavia essere completamente aleatorio perchè porterebbe ad avere un costruttore di soluzioni totalmente casuale che si è rivelato spesso non essere in grado fornire una soluzione iniziale ammissibile.

La *difficoltà di assegnamento* di una lezione l è stata invece solo leggermente modificata, per un valore casuale, rispetto al valore di *difficulty*(c_l). In definitiva, l'ordine di assegnamento delle lezioni è stato decretato ordinando in modo decrescente le lezioni secondo la seguente formula:

$$rank(l, rr) = difficulty(c_l) \cdot norm(\mu = 1, \sigma = rr)$$

Dove $norm$ è un generatore casuale che segue una distribuzione normale di media $\mu = 1$ e deviazione standard $\sigma = rr$ ($r$anking $r$andomness), di default impostato a 0.33. Si osserva che impostando il parametro rr a 0 verrà generata sempre la stessa soluzione deterministica (quella ritenuta più facile, in cui l'ordine di assegnamento delle lezioni rispetta la *difficulty*), mentre aumentando il valore di rr si otterranno ordini di assegnamento che daranno luogo a soluzioni sempre più "casuali", ma che al contempo avranno una bassa probabilità di essere ammissibili. È ora possibile definire l'algoritmo che genera una soluzione iniziale ammissibile.

Algorithm 1: Generatore di una soluzione iniziale ammissibile

```
Input: m, rr
Output: a feasible solution for m

// m: model
// rr: ranking randomness (default = 0.33)

begin
    do
        sol ← try_generate_feasible_solution(m, rr);
    while sol is NULL;
    return sol;
end

function try_generate_feasible_solution(m, rr):
    sol ← solution();
    L ← lectures of m sorted by descending rank(l, rr);
    foreach l ∈ L do
        foreach (r, d, s) ∈ (R, D, S) do
            if l can be assigned to (r, d, s) without violations then
                sol[l] ← (r, d, s);
                (R, D, S) ← (R, D, S) \ {(r, d, s)};
                break;
            end
        end
        if sol[l] is not assigned then
            return NULL
        end
    end
    return sol;
end
```

4.2 Neighbourhood

Elemento essenziale su cui si basano le metaeuristice sviluppate è la definizione di un *neighbourhood*. L'algoritmo sviluppato fa uso di un unico e semplice *neighbourhood* N definito come segue.

Data una soluzione s, il *neighbourhood* $N(s)$ consiste nell'assegnare ad una lezione l precedentemente assegnata ad una terna (aula, giorno, slot) $= (r, d, s)$, una nuova terna (r', d', s'). Se una lezione l' era già assegnata alla terna (r', d', s'), questa viene assegnata alla terna (r, d, s) a cui era assegnata l.

In sostanza, il *neighbourhood* implementato può essere interpretato come *swap* di due lezioni (con l'estensione che viene contemplato anche lo scambio fra una lezione ed un assegnamento (aula, giorno, slot) in cui non è presente alcuna lezione).

La dimensione del *neighbourhood* completo è dunque:

$$|N(s)| = L \cdot R \cdot D \cdot S$$

4.3 Metaeuristiche

Nei prossimi paragrafi saranno descritte le metaeuristiche implementate all'interno del risolutore; alcune di queste sono utilizzate come strategie di risoluzione nell'algoritmo proposto, altre non si sono rivelate all'altezza ma sono comunque state lasciate all'interno del progetto.

Si fornisce, per comodità, la seguente definizione di sottoinsieme del *neighbourhood* N composto solo dalle mosse che possono essere effettuate senza violare gli *Hard Constraitns*:

$$N^*(s) : \{mv \in N(s) \mid move\_is\_feasible(mv, s)\}$$

4.3.1 Local Search

La prima strategia sviluppata con cui è stato possibile confrontare le strategie sviluppate in seguito per giudicarne la bontà è stata la *Local Search* (impropriamente inserita all'interno della categoria delle metaeuristiche).

Data una soluzione iniziale s, il processo di *Local Search* consiste nell'ispezionare $N(s)$ accettando mosse di *swap* solo se (sono ammissibili e) diminuiscono il costo della soluzione; iterando il procedimento fino a che non è più presente alcuna mossa in $N(s)$ che migliori il costo della soluzione attuale.

Algorithm 2: Local Search

```
Input: s
Output: s'

// s: initial feasible solution

begin
    do
        improved ← false;
        foreach move ∈ N*(s) do
            if move_cost(move, s) < 0 then
                s ← apply_move(s, move);
                improved ← true;
                break
            end
        end
    while improved is true;
    return s;
end
```

4.3.2 Hill Climbing

La seconda strategia implementata risulta simile alla *Local Search*, ma ha permesso, attuando la semplice modifica di accettare mosse di $N(s)$ anche se mantengono invariato il costo $f(s)$ della soluzione attuale, di ottenere risultati decisamente migliori (si veda 5).

Le mosse del *neighbourhood* non sono ispezionate in sequenza (perchè si avrebbe una buona probabilità di creare un ciclo di mosse, che richiederebbe una strategia simile alla *Tabu Search* per essere evitato), ma sono scelte casualmente; questo introduce la necessità di un parametro max_idle che definisca dopo quanti tentativi per la quale il costo della soluzione non diminuisce interrompere il processo.

Algorithm 3: Hill Climbing

Input: s, max_idle
Output: s'

// s: initial feasible solution
// max_idle: maximum number of non-improving iterations

begin
  $idle \leftarrow 0$;
  $current\_cost \leftarrow solution\_cost(s)$;
  do
    $move \leftarrow$ random move from $N^*(s)$;
    if $move\_cost(move, s) \leq 0$ **then**
      $s \leftarrow apply\_move(s, move)$;
    end
    if $solution\_cost(s) < current\_cost$ **then**
      $current\_cost \leftarrow solution\_cost(s)$;
      $idle \leftarrow 0$;
    else
      $idle \leftarrow idle + 1$;
    end
  while $idle < max\_idle$;
  return s;
end

4.4 Tabu Search

La prima"vera metaeuristica" che è stata implementata è la Tabu Search; anche se, come verrà discusso nella sezione 5, i risultati ottenuti non sono stati all'altezza delle aspettative.

Il processo di *Tabu Search* effettua la miglior mossa possibile del *neighbourhood* ma, a differenza di *Local Search* e *Hill Climbing*, può accettare anche mosse che peggiorano il costo della soluzione con il fine di sfuggire ad un mi-

nimo locale; a supporto di ciò, per evitare che una volta usciti da un minimo locale si ritorni nello stesso, è mantenuta una *Tabu List* che memorizza un certo numero di mosse recentemente eseguite, impedendo che le mosse inverse ad esse siano eseguite.

Nell'implementazione proposta per la *Tabu Search*, la *Tabu List* è stata in realtà attuata mediante una matrice di dimensione $|N(s)| = L \cdot R \cdot D \cdot S$ e non mediante una lista/vettore; questo è stato possibile perchè la dimensione del *neighbourhood* ne consente la memorizzazione ed ha il beneficio di consentire un accesso $o(1)$ alla *Tabu List*.

Il parametro principale della *Tabu Search* è la *tabu tenure* (tt) che indica il numero di iterazioni per la quale una mossa è considerata *tabu-active*; è stato inoltre aggiunto un parametro opzionale ρ che determina un coefficiente di penalità in relazione al numero di volte che una mossa è diventata *tabu*.

La formula per calcolare il numero di iterazioni per la quale una mossa è *tabu-active* è la seguente:

$$tabu\_length(m) = tt + \rho \cdot freq(m)$$

Infine, è stato introdotto anche l'*Aspiration Criteria* che consente di accettare una mossa *tabu-active* nel caso in cui questa migliori il costo della miglior soluzione trovata fino a quel momento.

Algorithm 4: Tabu Search

Input: s, tt, ρ, max_idle
Output: s'

// s: initial feasible solution
// tt: tabu tenure (default = 120)
// ρ: frequency penalty (default = 0)
// max_idle: maximum number of non-improving iterations

```
begin
    idle ← 0;
    iter ← 0;
    tabu ← tabu_list();
    current_cost ← solution_cost(s);
    best_cost ← solution_cost(s);
    do
        best_move_cost = ∞ ;
        best_moves = [];
        // Find the subset of best non tabu-active moves
        foreach move ∈ N*(s) do
            if move_cost(move, s) ≤ best_move_cost AND
                  (tabu.is_allowed(move, iter) OR
                   current_cost + move_cost(move, s) < best_cost) then
                if move_cost(move) < best_move_cost then
                    best_move_cost ← move_cost(move, s);
                    best_moves = [];
                end
                best_moves.append(move);
            end
        end
        // Apply a move and make it tabu-active
        best_move ← pick at random from best_moves;
        s ← apply_move(s, best_move);
        tabu.ban(best_move, iter);
        if solution_cost(s) < current_cost then
            current_cost ← solution_cost(s);
            idle ← 0;
        else
            idle ← idle + 1;
        end
        if solution_cost(s) < best_cost then
            best_cost ← solution_cost(s);
        end
        iter ← iter + 1;
    while idle < max_idle;
    return s;
end
```

4.5 Simulated Annealing

Ultima ad essere implementata, *Simulated Annealing* è stata la metaeuristica che fra quelle proposte ha fornito i risultati migliori.

L'idea di *Simulated Annealing* è di generare mosse casualmente ed accettare, con una certa probabilità, anche mosse che peggiorano il costo della soluzione. La probabilità p con cui una mossa è accettata dipende sia dalla temperatura T attuale del sistema, la quale viene decrementata secondo un determinato *Simulated Annealing Schedule*, che dal costo Δ della mossa:

$$p(m) = e^{-\Delta/T}$$

Il valore di T viene decrementato ad ogni ciclo k per il *cooling rate* cr:

$$T_{k+1} = T_k \cdot cr$$

Ogni ciclo k (per la quale T è mantenuta costante) dura $\tau \cdot |N(s)|$ iterazioni, dove τ di default è $\frac{1}{8}$.

Algorithm 5: Simulated Annealing

Input: s, T_{init}, T_{min}, cr, τ
Output: s'

// s: initial feasible solution
// T_{init}: initial temperature (default = 1.4)
// T_{min}: minimum temperature to reach (default $\in [0.08, 0.12]$)
// cr: cooling rate (default = 0.965)
// τ: temperature length factor (default = 0.125)

```
begin
    T ← T_init;
    T_len ← τ · L · R · D · S;
    while T > T_min do
        for it < T_len do
            move ← random move from N*(s);
            Δ ← move_cost(move, s);
            if rand(0, 1) < e^{−Δ/T} then
                s ← apply_move(s, move);
            end
        end
        T ← T · cr;
    end
    return s;
end
```

4.6 Risolutore

Il progetto sviluppato non è vincolato ad alcuna metaeuristica specifica, al contrario, fa uso di un "risolutore" generico che può essere configurato per utilizzare una qualsiasi metaeuristica o sequenza di metaeuristiche.

Nel caso in cui il risolutore sia configurato per utilizzare più di una metaeuristica, l'esecuzione di esse avverrà secondo una politica *Round Robin* fino a che una condizione di arresto non si verifica (limite di tempo o limite di cicli).

Questo ha consentito di creare un algoritmo flessibile, in cui possono essere cambiate sia le metaeuristiche utilizzate che i rispettivi parametri.

5 Risultati

In questa sezione saranno confrontati i risultati delle metaeuristiche sviluppate, sia fra di esse (tabella 2), che rispetto ai risultati noti per le istanze provate (tabella 3).

5.1 Confronto fra le metaeuristiche

Per confrontare i risultati delle diverse strategie implementate sono state effettuate, per ogni metaeuristica, 10 esecuzioni di tutti i dataset compresi fra `comp01` e `comp07`. La durata di ogni esecuzione rispetta il tempo limite definito dalla traccia dell'ITC2007 e, nel caso della macchina su cui è stato sviluppato l'algoritmo (Arch Linux, Kernel 5.9.11, Intel i7 4790K @ 4.4GHz, 16GB RAM), è stato calcolato essere 168 secondi (tempo determinato mediante il programma `benchmark_my_linux_machine` fornito dagli organizzatori della competizione).

Di seguito sono riportati i parametri utilizzati per ciascuno dei *benchmark* ed i risultati ottenuti.

Tabella 1: Parametri utilizzati per il *benchmark* delle metaeuristiche

LS_{multi}	HC_{multi}

`solver.methods=ls`
`solver.multistart=true`
	`solver.methods=hc`
	`solver.multistart=true`
	`hc.max_idle=120000`
	`hc.max_idle_near_best_coeff=3`
	`hc.near_best_ratio=1.02`

TS	SA $(+LS)$

`solver.methods=ts`
`solver.multistart=false`
`ts.max_idle=-1`
`ts.tabu_tenure=120`
`ts.frequency_penalty_coeff=0`
	`solver.methods=sa,ls`
	`solver.multistart=false`
	`solver.restore_best_after_cycles=50`
	`sa.initial_temperature=1.4`
	`sa.cooling_rate=0.965`
	`sa.min_temperature=0.12`
	`sa.temperature_length_coeff=0.125`
	`sa.near_best_ratio=1.05`
	`sa.reheat_coeff=1.015`
	`ls.max_distance_from_best_ratio=1.02`

Tabella 2: Confronto dei risultati delle metaeuristiche ($t = 168$)

Dataset	LS_{multi}		HC_{multi}		TS		SA (+LS)			
	avg	σ	avg	σ	avg	σ	avg	σ	$best$	$best_{t=\infty}$
`comp01`	20.4	1.4	5	0	5.4	0.5	5	0	5	5
`comp02`	204.1	9.8	119.0	5.2	135.7	21.0	48.0	4.3	42	36
`comp03`	182.1	5.8	117.4	5.8	156.8	11.5	74.2	3.9	69	66
`comp04`	111.4	5.4	53.7	2.9	60.1	8.0	40.9	1.3	39	37
`comp05`	538.4	29.8	515.4	15.3	722.2	103.3	392.9	40.4	341	305
`comp06`	168.5	5.5	87.4	5.9	101.2	11.5	51.9	3.3	45	40
`comp07`	159.7	7.2	56.1	4	55.3	3.4	25.5	2.5	22	14

Local Search: le prime prove effettuate con LS diedero dei risultati non molto soddisfacenti in quanto l'algoritmo si interrompeva al primo minimo locale trovato. L'impiego del *multistart* ha logicamente consentito di migliorare i risultati ottenuti, ma nonostante questo i risultati forniti sono stati i peggiori fra le metaeuristiche testate. *Local Search* si è rivelata comunque utile ai fini del progetto, sia come termine di paragone per le metaeuristiche testate successivamente, sia perchè, seppur a margine, si affianca a *Simulated Annealing* per formare la sequenza di metaeuristiche utilizzate di default dal risolutore.

Hill Climbing: nonostante la semplicità della strategia, che si discosta da *Local Search* solo per l'accettare anche mosse che mantengo invariato il costo della soluzione, i risultati ottenuti da HC_{multi} sono stati tutto sommato soddisfacenti, posizionandosi 2^a fra le metaeuristiche testate. Dalle prove effettuate è risultato evidente che l'accettare *side-moves* generate casualmente consenta all'algoritmo di fuggire a minimi locali poco profondi, garantendo una ricerca in profondità migliore di *Local Search*.

Tabu Search: contrariamente alle aspettative, *Tabu Search* non ha fornito risultati particolarmente soddisfacenti, risultando in un caso anche peggiore di LS_{multi}. Sono state effettuate diverse analisi per comprendere la ragione di tale fenomeno; l'implementazione non dovrebbe essere il problema, considerato che la *Tabu List* è stata anche implementata mediante matrice ad accesso $o(1)$; la scelta dei parametri sicuramente potrebbe avere spazio di miglioramento, ma nonostante le prove effettuate con *tabu tenure* di diversi ordini di grandezza differenti, i risultati non sono mai migliorati significativamente. Un *profiling* attento di come il tempo di esecuzione fosse speso ha dato luce a quello che sembrerebbe essere il problema: la dimensione del *neighbourhood*. Questo non è un problema per *Hill Climbing* e *Simulated Annealing* in quanto le mosse sono generate ed effettuate casualmente, mentre *Tabu Search* ha la necessità di ispezionare tutto il *neighbourhood* per poter scegliere la mossa non *tabu-active* migliore possibile; questo porta *Tabu Search* ad effettuare una quantità di mosse per unità di tempo centinaia se non migliaia di volte minore rispetto alle

altre metaeuristiche citate. Come si evince da [2], questo è un fenomeno noto; facendo ispezionare all'algoritmo tutto il *neighbourhood* si ottengono soluzioni generalmente buone ma in tempi proibitivi per istanze medio/grandi. Come descritto da *Fred Glover* e *Eric Taillard*, esistono diversi modi per risolvere questa problematica: dal conservare una lista di possibili mosse candidate *elite* da ispezionare in modo esclusivo o prima del resto del *neighbourhood* nell'iterazione successiva, all'usare strutture di memoria a medio/lungo termine per individuare le caratteristiche delle mosse/traiettorie che hanno portato a miglioramenti più efficaci e circoscrivere la ricerca ad un sottoinsieme del *neighbourhood*. Dato il limitato tempo a disposizione e la complessità di tali idee, non sono state apportate modifiche alla *Tabu Search* originariamente sviluppata.

Simulated Annealing: è la metaeuristica che ha fornito i migliori risultati, nonchè la strategia scelta di default per il risolutore. In un primo momento *Simulated Annealing* sembrava non essere in grado di fornire risultati soddisfacenti all'interno degli stretti tempi consentiti; la difficoltà di integrare questa metaeuristica è infatti celata non tanto nell'implementazione, al quanto semplice, quanto nel *tuning* dei parametri della stessa. Dopo diversi tentativi per cercare il giusto connubio fra intensificazione e diversificazione, quindi fra temperatura iniziale, temperatura minima, cooling rate e lunghezza della temperatura, sono stati individuati dei parametri che risultano essere adeguati per le istanze testate (rispetto a tempi di computazione previsti).

Infine, si evidenziano diverse accortezze che hanno consentito di migliorare i risultati rispetto a SA nella sua forma base. Innanzitutto, il risolutore è stato implementato per consentire non uno ma più cicli di SA, ognuno dei quali riparte dalla soluzione prodotta dal ciclo precedente con una temperatura iniziale reimpostata a T_{init}; questo approccio ha consentito di mantenere una temperatura iniziale abbastanza bassa ed un cooling rate non troppo aggressivo, al fine di non "perdere" troppo tempo ad ispezionare soluzioni con costi eccessivi ed effettuare una buona ispezione al termine del ciclo SA delle soluzioni candidate ad essere nuove migliori. È stato poi introdotto un coefficiente di *reheat*: se al termine di un ciclo di SA il minimo globale non è stato migliorato, la temperatura iniziale del ciclo successivo sarà incrementata a:

$$T_{init,i+1} = T_{init,i} \cdot SA_{reheat}$$

Questo conferisce all'algoritmo una strategia di diversificazione dinamica, consentendogli di cercare più lontano dall'attuale soluzione se questa non è migliorata per un lungo periodo. Ad ogni modo, se dopo un determinato numero di iterazioni (di default 50) il costo della miglior soluzione non riesce ad essere migliorato, la soluzione migliore è imposta come soluzione corrente e la temperatura è reimpostata a T_{init}

Altra accortezza introdotta: anche la temperatura minima da raggiungere è variabile; di default è mantenuta ad un valore abbastanza alto di 0.12, ma è decrementata a circa 0.08 se il costo della soluzione corrente è abbastanza vicino ($< 5\%$) rispetto al costo della miglior soluzione trovata fino a quel momento;

in questo modo è possibile evitare computazione nel caso in cui sarebbe improbabile scendere sotto ad un minimo globale, iniziando al più presto il ciclo SA successivo.

Ed infine, si osserva che la strategia utilizzata di default dal risolutore non è solo SA ma è $SA + LS$: una esecuzione di *Local Search* sarà eseguita nel caso in cui il costo della soluzione ottenuta al termine di una ciclo SA sia abbastanza vicino al costo della miglior soluzione trovata fino a quel momento ($< 2\%$). Si è osservato che questo ha un impatto praticamente nullo sul tempo di computazione "rubato" ad SA, ma dà una forte garanzia sul fatto che il valore ottenuto da SA sia effettivamente un minimo locale; nella maggior parte dei casi questo è vero ed il ruolo di LS diventa irrilevante, ma in alcuni casi, essendo SA un processo aleatorio, questo non si verifica ed LS è in grado di decrementare il costo della soluzione (generalmente di qualche unità), ottenendo una nuova miglior soluzione.

5.2 Confronto con risultati noti

Per stabilire la bontà dell'algoritmo sviluppato, i risultati ottenuti sono stati confrontati con quelli noti pubblicamente. In particolare, si propongo due confronti:

I Confronto fra il miglior risultato di 10 esecuzioni (con tempo massimo di 168 secondi) per le istanze `comp01` $\div$ `comp07` rispetto ai risultati ottenuti dai partecipanti all'ITC2007 [3].

II Confronto fra il miglior risultato ottenuto (senza limite di tempo) ed il miglior risultato ottenuto da T. Müller (vincitore della Track *Course Curriculum Timetabling* dell'ITC2007) [4].

Da entrambi i confronti si evince che l'algoritmo sviluppato compete con l'algoritmo del vincitore della competizione. Per il confronto I, in 4 casi su 7 è stato fornito un risultato migliore o uguale degli altri partecipanti. Per il confronto II, in 5 casi su 7 è stato fornito un risultato migliore (o uguale) a quello ottenuto dal vincitore della competizione.

Tabella 3: Confronto I

Dataset	SA ($+LS$)	T. Muller	Lu Hao	Atzuna et al	Geiger	Clark et al
`comp01`	**5**	**5**	**5**	**5**	**5**	10
`comp02`	**42**	51	55	50	111	111
`comp03`	**69**	84	71	82	128	119
`comp04`	39	37	43	**35**	72	72
`comp05`	341	**330**	309	312	410	426
`comp06`	**45**	48	53	69	100	130
`comp07`	22	**20**	28	42	57	110

Tabella 4: Confronto II

Dataset	SA $(+LS)$ *best*	T. Muller [4] *best*
`comp01`	**5**	**5**
`comp02`	**36**	43
`comp03`	**66**	72
`comp04`	37	**35**
`comp05`	305	**298**
`comp06`	**40**	41
`comp07`	**14**	**14**

6 Dettagli implementativi

L'algoritmo è stato sviluppato in C e non sarà commentato nella sua interezza in questa relazione; ad ogni modo se ne sottolineano gli aspetti chiave.

La struttura dati utilizzata per memorizzare una soluzione è stata di fondamentale importanza; inizialmente è stato sufficiente utilizzare una matrice `timetable` di dimensione $C \cdot R \cdot D \cdot S$ in cui ogni elemento rispecchiasse esattamente il significato della variabile x_{crds} del modello formale (il corso c utilizza l'aula r nel giorno d, slot s). Usare solo questa struttura di memoria, seppur consenta ogni sorta di computazione necessaria, è risultato ben presto poco agevole.

Uno dei punti più critici (i.e. eseguito per la maggior parte del tempo) delle metaeuristiche utilizzate dal risolutore è infatti prevedere se una mossa darà luogo ad una soluzione ammissibile e quale sarà il costo che introdurrà se venisse effettuata; una strategia potrebbe essere applicare la mossa alla soluzione e calcolarne il costo da zero: questo risulta computazionalmente ingestibile, nonchè superfluo, dato che la maggior parte della soluzione rimane invariata in seguito ad una singola mossa. È stato quindi necessario prevedere il costo della soluzione in base alle singole lezioni coinvolte dalla mossa e per farlo in modo efficiente è risultato conveniente mantenere all'interno della soluzione strutture dati di supporto che memorizzino dati pre-computati, ricavabili dalla matrice `timetable` (questo naturalmente introduce l'esigenza di mantenere consistenti tali strutture rispetto alla matrice principale).

In seguito all'implementazione di un buon meccanismo di previsione, integrare le metaeuristiche è risultato tutto sommato agevole.

Riferimenti bibliografici

[1] *Internation Timetabling Competition 2007*. URL: `http://www.cs.qub.ac.uk/itc2007/`.

[2] Fred Glover e Eric Taillard. "A Users Guide to Tabu Search". In: *Ann. Oper. Res.* 41 (mar. 1993), pp. 1–28. DOI: `10.1007/BF02078647`.

[3] *Internation Timetabling Competition 2007 - The Finalists.* URL: http://www.cs.qub.ac.uk/itc2007/winner/finalorder.htm.

[4] Tomáš Müller. "ITC2007 solver description: A hybrid approach". In: *Annals of Operations Research* 172 (gen. 2008), pp. 429–446. DOI: 10.1007/s10479-009-0644-y.