Implementazione metauristica "Random Iterated Greedy" per problemi di minimizzazione della tardiness totale su macchine parallele identiche e sequence-dependent setup-time jobs

Simone Mione - Matricola n. 142112

Algoritmi di Ottimizzazione - Scienze Informatice Unimore

Introduzione

Il problema in oggetto è un problema di scheduling di jobs su macchine parallele identiche con l'obiettivo di minimizzare la total tardiness. Viene proposta una dispatching rule e valutata comparandola con altri approcci esistenti, su uno stesso set di dati. Inoltre, è sviluppato un metodo metaeuristico greedy-based per migliorare la soluzione iniziale. I risultati mostrano che la metaeuristica proposta performa meglio degli approcci comparati in questo elaborato.

Info: Il seguente lavoro è un riassunto degli algoritmi presentati nel seguente articolo citato [[a]], un'implementazione di tali algoritmi, nonché test e risultati di essi. La parte di descrizione è ridotta all'essenziale per non risultare prolissi e ripetitivi su temi già affrontati nell'articolo. Per eventuali dettagli si rimanda ad esso, mentre la parte di implementazione non è presente nel citato articolo.

[a] Jae-Gon Kim, Seokwoo Song, BongJoo Jeong. (2020) Minimising total tardiness for the identical parallel machine scheduling problem with splitting jobs and sequence-dependent setup times. International Journal of Production Research 58:6, pages 1628-1643.

1 Descrizione del problema

Nomenclatura utilizzata in questo elaborato:

- m: numero di macchine
- M_k: macchina k,k=1,...,M
- n: numero di jobs
- J_i: job i, i=1,...,n
- p_i: processing time di J_i, i=1,...,n
- C_i:Completition time di J_i, i=1,...,n
- d_i: due date di J_i, i=1,...,n
- T_i: tardiness di J_i, i=1,...,n
- A_a: spec a, a=1,...,5 ($A_1 - A_5$ stanno rispettivamente per length, width, thickness, hardness e colour)
- $N_a(J_i)$: il numero di jobs con lo stesso valore della spec i di J_i, a=1,...,5

Specification	Job 1	2	3	4	5	6	7	8	9	10	Specification set-up time
Length (cm)	240	96	96	240	240	240	96	96	144	96	60
Width (in.)	12	36	24	36	36	12	36	48	36	24	15
Thickness (mm)	1.5	5.0	3.0	8.0	1.5	3.0	1.5	3.0	3.0	5.0	10
Hardness (No.)	7	8	9	8	7	9	7	9	9	7	20
Colour (No.)	1	1	2	2	2	1	9	1	1	2	15
p_i (min.)	444	189	474	313	644	253	578	645	459	361	in minutes
d_i (min.)	2315	2087	1614	2463	2037	2275	2142	1693	1754	1596	

Figure 1: Tabella dei dati utilizzati per i test

- S_ij: eventuale setup time che intercorre tra J_i e J_j se J_j è schedulato subito dopo J_i nella stessa macchina, *i*,*j*=1,...,*n*, *i*!=*j*
- S_a: il setup time della specifica *a*-esima

Il problema affrontato consiste in *n* jobs processati su *m* macchine parallele identiche. Ogni job J_i ha uno specifico processing time p_i e un due date d_i, e può essere processato arbitrariamente su qualunque macchina. Potrebbe intercorrere un setup time tra due job consecutivi sulla stessa macchina, se essi hanno valori diversi per qualche A_a. Tutte le macchine sono pronte al tempo 0, come anche i jobs. Non sono ammesse interruzioni e preemption, e non ci sono priorità. Una macchina può processare al massimo un job alla volta, ed il job può essere processato da una sola macchina. L'obiettivo è trovare uno schedule che minimizzi la tardiness totale. La tardiness è definita come T_i=max{$C_i - d_i$, 0}.

$$Minimize \sum_{i=1}^{n} T_i \tag{1}$$

È stato dimostrato che la minimizzazione della tardiness totale su una macchina singola è NP-Hard, quindi lo è anche in questo caso di studio in quanto l'ulteriore aggiunta del sequence-dependent setup-times tra i jobs complica il problema, come anche il fatto che abbiamo più macchine parallele.

2 Metodi di scheduling

In questa sezione verrano illustrati i metodi di scheduling implementati. Per gli esempi, sono stati usati i dati in figura 2.

2.1 Metodo di scheduling 0 - LPT + EDD

Il primo metodo di scheduling che vediamo è quello di utilizzare le regole LPT e EDD. La regola LPT (Longest Processing Time first) consiste nell'ordinare i jobs in ordine decrescente per il processing time ed assegnarli alle macchine via via che si liberano, in modo da migliorare il bilanciamento del carico. La regola EDD (Earliest Due Date first) consiste nell'ordinare i jobs in ordine crescente per il due date, in modo da evitare il più possibile che essi finiscano in ritardo. L'algoritmo qui utilizzato si divide in due fasi:

Step 1. Scegli J_i con il processing time più lungo ed assegnalo alla prima macchina disponibile. Ripeti fino a che tutti i jobs sono assegnati.

Step 2. Per ogni macchina, riordina i jobs in ordine crescente per la due date.

Questo è un algoritmo deterministico se non teniamo conto dell'aspetto multi-spec setup-times che caratterizza questo problema. In riferimento ai dati in tabella 2, dopo lo Step 1 lo schedule sarà:

- M1=(J8, J3, J9, J10, J6)
- M2=(J5, J7, J1, J4, J2)

e dopo lo Step 2:

- M1=(J10, J3, J8, J9, J6)
- M2=(J5, J2, J7, J1, J4)

Con una tardiness totale di 447 min.

2.2 Metodo di scheduling 1 - ATCS

Il secondo metodo implementato è stato preso in prestito dalla regola data da Lee e Pinedo. L' Apparent Tardiness Cost with Setups (ATCS) index è una regola molto usata nei problemi di scheduling per minimizzare la total tardiness. L'idea di base è quella di calcolare l'ATCS per ogni job non processato e pronto quando una macchina diventa libera. Successivamente, il job con l'ATCS più alto viene scelto ed assegnato sulla macchina libera. L'ATCS prende in considerazione i processing time, i minimum slack ed i setup-time in un singolo ranking. L'indice di J_i al tempo *t* è determinato da:

$$I_i(t,j) = \frac{w_i}{p_i} exp(-\frac{max(d_i - p_i - t, 0)}{K_1 \bar{p}}) exp(-\frac{S_{ij}}{K_2 \bar{s}}) \tag{2}$$

Dove:

- t: denota quando J_i completerebbe il suo processamento se eseguisse sulla macchina
- $\bar{s}$: è la media dei setup times
- $\bar{p}$: è la media dei processing times
- K_1 e K_2: sono due parametri, verranno mostrati di seguito

$$K_1 = 1.2ln(\frac{n}{m}) - R, \begin{cases} K_1 = J_1 - 0.5, & \text{se } \tau < 0.5 \\ K_1 = J_1 - 0.5 & \text{se } \eta < 0.5, \mu > 5 \end{cases} \tag{3}$$

Dove τ e *R* sono fattori associati al due date. τ è il due date tughtness e *R* è il due range factor. In riferimento ai dati in tabella 2, l'ATCS porta al seguente schedule:

- M1=(J10, J3, J9, J6, J1)
- M2=(J3, J2, J7, J5, J4)

Con una tardiness totale di 363 min.

E' ovvio quindi che l'ATCS è un miglior scheduler per questo problema rispetto a quello descritto nella sezione 1.1, questo perché prende in considerazione anche la caratteristica del multi-spec setup-time.

2.3 Metodo di scheduling 2 - ATCS + APD

Introduciamo prima l'index APD (Adjacent Processing Time) utilizzato per sviluppare questa euristica. L'indice APD prende in considerazione il processing time, il due date ed i setuptimes. In riferimento a questi ultimi, vengono considerati anche quanti valori di questi sono comuni tra i job (flessibilità). L'index è calcolato come segue:

$$APD_i = \frac{ln\{\sum_{a=1}^{5}[S_a * N_a(J_i) * d_i]\}}{p_i} \tag{4}$$

dove p_i è il processing time di J_i, S_a è il setup time della specifica *a*-esima, e $N_a(J_i)$ sta per il numero di jobs che hanno lo stesso valore della specifica *a* di J_i. Il job con APD_i minore andrà schedulato all'inizio. Definito l'APD index, si andrà ora ad integrare con l'ATCS formando l'ATCS_APD index, come segue:

$$I_i(t,j) = \frac{w_i}{p_i} exp(-\frac{max(d_i - p_i - t, 0)}{K_1 \bar{p}}) exp(-\frac{S_{ij}}{K_2 \bar{s}}) \mathbf{exp}(-\frac{\mathbf{1}}{\mathbf{APD_i \bar{s}}}) \tag{5}$$

dove $I_{ATCS\_APDi}(t,j)$ è l'indice del job *i* al tempo *t*, dato il job *j* che è l'ultimo job completato sulla macchina scelta, ovvero quella appena liberata. Il job con ATCS_APD maggiore è considerato quello con priorità maggiore. Di seguito vengono mostrati i passi per implementare il metodo ATCS_APD:

Step 1. Per ogni job *i* calcola l'APD_i e setta *t*=0.

Step 2. Scegli la macchina *k* che è disponibile al tempo *t* e calcola $I_{ATCS_APDi}(t, j)$ per ogni job *i* non schedulato.

Step 3. Il job *i* con $I_{ATCS_APDi}(t, j)$ maggiore è assegnato alla macchina *k* e viene settato il tempo *t* come il loading time della macchina *k*. Se ci sono ancora jobs non schedulati, allora torna a *Step 2.*, altrimenti fermati.

Sempre in riferimento ai dati in tabella 2, questo algoritmo porta al seguente schedule:

- M1=(J10, J2, J7, J5, J1)
- M2=(J3, J8, J9, J6, J4)

Con una tardiness totale di 170 min.

2.4 Metodo di scheduling 3 - ATCS_APD + RIG

Infine, in questa sezione viene mostrato il Random Iterated Greedy (RIG). Questa metaeuristica tenta di migliorare le soluzioni fornite da ATCS_APD, seguendo un approccio greedy-based.
L'approccio Iterated Greedy genera le soluzioni usando due fasi principali: distruzione e costruzione.
Nella fase di distruzione, alcuni jobs vengono rimossi dalla soluzione corrente. Nella fase di costruzione, vengono applicate delle regole per riottenere una soluzione completa.
Una volta ottenuta, viene usato un criterio per determinare se il nuovo scheduling viene accettato aggiornando il corrente. La procedura viene ripetuta fino a che non si raggiunge un criterio di stop. In questo caso, è impostato come numero massimo di iterazioni.
Questo studio propone un random iterated greedy per migliorare le soluzioni fornite da ATCS_APD.
RIG itera tra 3 diversi stages: distruzione, costruzione e movement. L'idea è di avere diversi tipi di distruzione e di costruzione tra cui scegliere casualmente ad ogni iterazione, cosi da poter avere più alte chances di trovare soluzioni migliori.

In particolare, si hanno 2 tipi di distruzione (DST_i e 3 tipi di costruzione (CST_j, definiti come:
Distruzione:

- Distruzione 1 (DST_1): Sceglie casualmente un job dalla macchina con la massima C_{max}
- Distruzione 2 (DST_2): Sceglie casualmente un job tra tutte le macchine

Costruzione:

- Costruzione 1 (CST_1): Inserisce il job scelto tra tutte le possibili posizioni della macchina dalla quale viene
- Costruzione 2 (CST_2): Inserisce il job scelto tra tutte le possibili posizioni della macchina dalla quale viene e della macchina col minor C_{max}
- Costruzione 3 (CST_3): Inserisce il job scelto tra tutte le possibili posizioni di tutte le macchine

I passi del RIG sono i seguenti: Ripetere i seguenti step fino a che non si raggiunge il numero massimo di iterazioni *l*=l_{max}:

Step 0. Inizializzazione: solo alla prima iterazione, prendere soluzione iniziale *X* la soluzione ottenuta da ATCS_APD. Per *l*>1, selezionare casualmente due jobs dalla soluzione e scambiarli in modo da trovare una nuova soluzione *X*.

Step 1. Distruzione: Selezionare a caso un DST_i e usarlo su *X*

Step 2. Costruzione: Selezionare a caso un CST_j per costruire una nuova soluzione $\bar{X}$

Step 3. Movement: Se $\bar{X}$ è migliore di X, allora $X \leftarrow \bar{X}$ e torna a *Step 1.*. Altrimenti, setta $l = l + 1$ e torna a *Step 0.*.

Quindi, ricapitolando, si settano le condizioni iniziali nello step 0. Dopodiché, viene scelta casualmente una combinazione di DST_i e CST_j per generare una nuova soluzione. All'ultimo step, viene definita la condizione di fine.

3 Implementazione

Per implementare il codice è stato utilizzato il linguaggio Python. Sono state implementate 3 classi, esse sono implementate nel loro omonimo file sorgente .py:

- Job: la classe job descrive un job. Esso è definito da 4 campi:
  1. id_number: un numero identificativo, id_number=1,...,n
  2. specs: la lista delle specs che descrivono il job reale. Nell'esempio, length, width, color, ... Quindi, i vari A_a
  3. processing_time: il tempo di processamento del job
  4. due_date: il due date del job

  Inoltre, contiene diversi metodi utili, come ad es. get_min_due_date che prende in ingresso una lista di jobs e restituisce il job col due date più vicino; utilizzata per implementare EDD.

- Machine: la classe machine descrive una macchina. Essa è definita da 2 campi:
  1. id_number: un numero identificativo, id_number=1,...,m
  2. jobs: la lista dei jobs, in ordine, schedulati su questa macchina

  Inoltre, contiene diversi metodi utili, come ad es. exchange_jobs che prende in ingresso una lista di macchine e scambia due jobs a caso; utilizzata per implementare la prima fase del RIG.

- Spec: la classe spec descrive una specifica. Essa è definita da 3 campi:
  1. name: il nome della specifica (es. length)
  2. value: il valore che quella specifica ha per il job
  3. setup_time: il valore dell'eventuale setup time

Il file utilities contiene delle funzioni utili per debug e stampe. Il file datas contiene i dati hardcodati utilizzati come esempio. Infine, il file main, contiene l'implementazione dei vari scheduling descritti da questo studio. Nella prossima sezione vedremo un po' più nel dettaglio tale codice.

3.1 Codifica

Di seguito è inserito il codice per lo scheduler 0. Il primo step consiste nell'assegnare alle macchine i job in ordine dal più lungo (in termini di processing time) al più corto, quindi con la regola LPT. Il secondo step, prende lo schedule creato dallo step 1 e li riordina secondo la EDD, quindi dal job con la due date più vicina.

main.py

```
# ------------ Scheduler 0 ----------------
def step_1_sched_corrente(jobs, machines):
  # ordino la lista dei jobs in ordine decrescente
  # per processing time
  lpt = LPT(jobs)
  # assegno i primi job delle macchine
  for k in machines:
    k._jobs.append(lpt[0])
    lpt.remove(lpt[0])

  # assegno ogni job ancora non schedulato alla
  # prima macchina libera
  for job in lpt:
    k = Machine.first_machine_avaible(machines)
    k._jobs.append(job)

  return machines

def step_2_sched_corrente(machines: list):
  # per ogni macchina, assegno alla macchina la
  # sua lista dei jobs ordinata con la regola EDD
  for k in machines:
    mins_dd = EDD(k._jobs)
    k._jobs = mins_dd
```

Per lo scheduluer 1 viene implementato il codice per calcolare l'ATCS index. Calcola quindi l'ATCS per ogni job e assegna alla prima macchina libera il job con l'indice maggiore.

main.py

```
# ------------ Scheduler 1 - ATCS --------------------
def ATCS(jobs, machines, setup_times, tao=0.9, R=0.2):

  # calcolo s come media dei setup times
  # calcolo p come media dei processing times
  s = Spec.get_setup_time_avg(setup_times)
  p = Job.get_processing_time_avg(jobs)

  # calcolo parametri di ATCS
  K1 = 1.2*np.log(len(jobs)/len(machines))-R
  #if tao<0.5 or (nano<0.5 and micro>0.5):
  if tao<0.5:
    K1 -= 0.5
  A2 = 1.8 if tao<0.8 else 2.0
  K2 = tao/(A2*math.sqrt(s/p))
  wi = 1

  U = deepcopy(jobs)

  # Step 1 - Prendo la macchina libera al tempo t,
  # poi per ogni job non schedulato:
  #     a. Calcolo l'ATCS index
  #     b. Assegno alla macchina selezionata il job
  # con ATCS_APD index maggiore
  print_d("Index␣ATCS:", 2)
  while U:
    I_i = dict()
    # prendo la macchina libera al tempo t
    k = Machine.first_machine_avaible(machines)
      # job_j    l'ultimo job processato
      # dalla macchina selezionata.
      # Serve per il setup time
    job_j = k._jobs[-1] if k._jobs else None
    # a. calcolo l'ATCS index
    for job_u in U:
      tmp_machine = deepcopy(k)
      tmp_machine._jobs.append(job_u)
      t = tmp_machine.get_free_time()
      I_i[job_u] = ((wi/job_u._processing_time)
        *math.exp(-max(job_u._due_date
        -job_u._processing_time-t, 0)
        /(K1*p))
        *math.exp(- Job.get_setup_time(job_j,job_u)
        /(K2*s)))

    I_i = dict(sorted(I_i.items(),
    key=lambda item: item[1], reverse=True))

    # b. assegno alla macchina il job con index
    # maggiore alla macchina
    for j in I_i:
      print_d(j, 2)
      k._jobs.append(j)
      U.remove(j)
      break
```

Per lo scheduluer 2 viene implementato il codice per calcolare, oltre l'ATCS index, anche l'APD index, grazie al quale è possibile calcolare l'ATCS_APD index. Calcola quindi l'ATCS_APD per ogni job e assegna alla prima macchina libera il job con l'indice minore.

main.py

```
# ------------ Scheduler 2 - ATCS_APD ----------------
def APD(jobs, setup_times):
  apd = dict()
  for job in jobs:
    tmp = 0
    for setup_name, setup_time in setup_times.items():
      tmp += setup_time
        * Job.get_Na(job,jobs,setup_name)
        * job._due_date
    apd[job] = np.log(tmp)/job._processing_time

  return apd

def ATCS_APD(jobs, machines, setup_times, tao=0.5, R=0.8):

  # calcolo parametri di ATCS (vedere ATCS)
  # s,p,K1,K2,A2,wi

  U = deepcopy(jobs)

  # step 0 -  Calcolo l'APD per ogni job e setto t=0
  apds = APD(U, setup_times)
  t = 0
  # Step 1 - Prendo la macchina libera al tempo t, poi
  # Per ogni job non schedulato:
  #     a. Calcolo l'ATCS_APD index
  #     b. Assegno alla macchina selezionata il job
  # con ATCS_APD index maggiore
  #     c. Imposto t come il loading machine
  while U:
    I_i = dict()
    # prendo la macchina libera al tempo t
    k = Machine.free_machine_at_t(machines, t)
    # job_j e' l'ultimo job processato dalla macchina selezionata.
    # Serve per il setup time
    job_j = k._jobs[-1] if k._jobs else None
    # a. calcolo l'ATCS_APD index
    for job_u in U:
      apd_i = apds[job_u]
      I_i[job_u] = ((wi/job_u._processing_time)
        *math.exp(-max(job_u._due_date
        -job_u._processing_time-t, 0)
        /(K1*p))
        *math.exp(-Job.get_setup_time(job_j,job_u)
        /(K2*s))
        *math.exp(-1/(apd_i*s)))

    I_i = dict(sorted(I_i.items(),
    key=lambda item: item[1], reverse=True))

        # b. assegno alla macchina il job con index
        # maggiore alla macchina
        for j in I_i:
      # c. imposto il tempo t al loading time
      t = k.get_free_time() + Job.get_setup_time(job_j, j)
      k._jobs.append(j)
      U.remove(j)
      break
```

Infine vediamo lo scheduler RIG. Per esso si sono implementate le diverse funzioni per le fasi di distruzione, costruzione e movement. La prima, è definita da 2 diversi tipi di distruzione; la seconda, è definita da 3 diversi tipi di costruzione; ed infine il movement che controlla se sono state trovare nuove soluzioni ammissibili. Quindi, la funzione RIG, itera su queste tre funzioni allo scopo di trovare soluzioni migliori rispetto a quella di partenza fino al raggiungimento del numero di iterazioni massime.

main.py

```
# ------------ Scheduler 3 - ATCS_APD + Greedy -------
def RIG(jobs, machines, setup_times, l_max=100000):

  EDD_m(jobs, machines)
  ATCS_APD(jobs, machines, setup_times)

  tardiness_iniziale =
    max(Machine.get_total_tardiness_per_machine(machines))
  switch = False
  l = 0
  best_sol = [tardiness_iniziale, machines]
  while l < l_max:
    # scambio a caso due jobs
    if switch:
      tardiness_now =
        max(Machine.get_total_tardiness_per_machine(machines))
      if tardiness_now < best_sol[0]:
        best_sol[0] = tardiness_now
        best_sol[1] = deepcopy(machines)

        if not switch:
                Machine.exchange_jobs(machines)
                l += 1

    # scelgo a caso i tipi di distr e costr da usare
        desc_type = randint(1,2)
        cons_type = randint(1,3)
        job = descruction(desc_type, machines)
        new_xes = construction(cons_type, job, machines)
        switch, machines = movement(machines, new_xes)

  return best_sol
```

main.py

```
def construction(cons_type, job: Job, machines):
  new_xes = []
  if cons_type == 1:
    # inserisci il job in qualunque posizione
    # della macchina dalla quale viene
    for k in machines:
      if k.is_job_of_this_machine(job):
        for pos in range(len(k._jobs)):
          all_m =
          [machine for machine in machines if machine != k]
          tmp_cpy = deepcopy(k)
          tmp_cpy._jobs.remove(job)
          tmp_cpy._jobs.insert(pos, job)
          all_m.append(tmp_cpy)
          new_xes.append(all_m)

        return new_xes

  elif cons_type == 2:
    # inserisci il job in tutte le posizione
    # della macchina dalla quale viene
    # e dalla macchina col minor Cmax
    k_min_c = Machine.get_machine_with_min_c(machines)
    for k in machines:
      if k.is_job_of_this_machine(job):
        for pos in range(len(k._jobs)):
          all_m =
            [machine for machine in machines if machine != k]
          tmp_cpy = deepcopy(k)
          tmp_cpy._jobs.remove(job)
          tmp_cpy._jobs.insert(pos, job)
          all_m.append(tmp_cpy)
          new_xes.append(all_m)
          if k != k_min_c:
            for pos in range(len(k_min_c._jobs)):
              all_m =
                [machine for machine in machines if machine != k_min_c]
              tmp_cpy = deepcopy(k_min_c)
              if job in tmp_cpy._jobs:
                tmp_cpy._jobs.remove(job)
                tmp_cpy._jobs.insert(pos, job)
                all_m.append(tmp_cpy)
                new_xes.append(all_m)
    return new_xes

  elif cons_type == 3:
    # inserisci il job in tutte le posizioni
    # di tutte le macchine
    for k in machines:
      for pos in range(len(k._jobs)):
        all_m =
          [machine for machine in machines if machine != k]
        tmp_cpy = deepcopy(k)
        if job in tmp_cpy._jobs:
          tmp_cpy._jobs.remove(job)
          tmp_cpy._jobs.insert(pos, job)
          all_m.append(tmp_cpy)
          new_xes.append(all_m)
    return new_xes
```

main.py

```
def descruction(desc_type, machines):

  if desc_type == 1:
    # seleziono a random un job dalla macchina con max Cmax
    max_c_machine = Machine.get_machine_with_max_c(machines)
    return max_c_machine.get_random_job()

  elif desc_type == 2:
    # seleziono a random un job da tutte le macchine
    return Machine.get_random_job_among_all(machines)

def movement(old_x, new_xes):
  old_tardiness = []
  for k in old_x:
    old_tardiness.append(k.get_total_tardiness())

  best_tard = max(old_tardiness)
  best_sched = old_x
  for new_x in new_xes:
    new_x_tardiness = []
    for k in new_x:
      new_x_tardiness.append(k.get_total_tardiness())
    new_x_tardiness = max(new_x_tardiness)
    if new_x_tardiness < best_tard:
      best_tard = new_x_tardiness
      best_sched = new_x

  changed = True if best_tard != max(old_tardiness) else False

  return changed, best_sched
```

4 Risultati e conclusioni

Eseguendo il codice, vediamo il seguente output:

Command Line

```
$ python3 main.py

        ------ Scheduler 0 - Corrente ----------
        Risultato per scheduling corrente
        Machine #1: (10, 3, 8, 9, 6, )
        Machine #2: (5, 2, 7, 1, 4, )
        Tardiness totale:  447

        ------ Scheduler 1 - ATCS ----------
        Risultato per scheduling con ATCS
        Machine #1: (10, 8, 2, 9, 6, )
        Machine #2: (3, 7, 5, 4, 1, )
        Tardiness totale:  363

        ------ Scheduler 2 - APD_ATCS ----------
        Risultato per scheduling con ATCS_APD
        Machine #1: (10, 2, 7, 5, 4, )
        Machine #2: (3, 8, 9, 6, 1, )
        Tardiness totale:  170

        ------ Scheduler 3 - APD_ATCS + GREEDY ----------
        Tardinss iniziale:  170
        Risultato per scheduling con ATCS_APD + GREEDY
        Machine #1: (10, 3, 8, 6, 1, )
        Machine #2: (9, 5, 7, 2, 4, )
        Tardiness totale:  38
```

Vediamo quindi come si ha sempre un maggior miglioramento nei risultati. Lo scheduler 0 è quello che performa peggio, questo perché non prende in considerazione il setup time, che è una caratteristica fondamentale per la realtà del problema. Lo scheduler 1 migliora un po' le performance, prendendo in questo caso in considerazione anche i setup time. Lo scheduler 2 migliora di parecchio le performance, poiché oltre a prendere in considerazione l'ATCS (e quindi i setup time), prende in considerazione anche più dettagli dei dati di riferimento, ovvero il numero di job che condividono gli stessi valori delle specs. Lo scheduler 3, l'oggetto di questo lavoro, riesce a migliorare lo scheduler 2 utilizzando un greedy-based approach, iterando su 3 fasi (distruzione, costruzione e movement) cosi da cercare nuove soluzioni migliori.

Nella figura 4 vediamo un confronto tra le soluzioni trovate dai vari metodi di scheduling.

Nella figura 4 vediamo un confronto, per lo scheduler RIG, per vari livelli di iterazioni_max. Vediamo sia la soluzione trovata che il tempo di completamento.

Notice: I tempi sono stati presi su una macchina con SO Ubuntu LTS 20.04 a 64bit, 12x Intel core i7-9750H @ 2.60GHz, 16GB di RAM

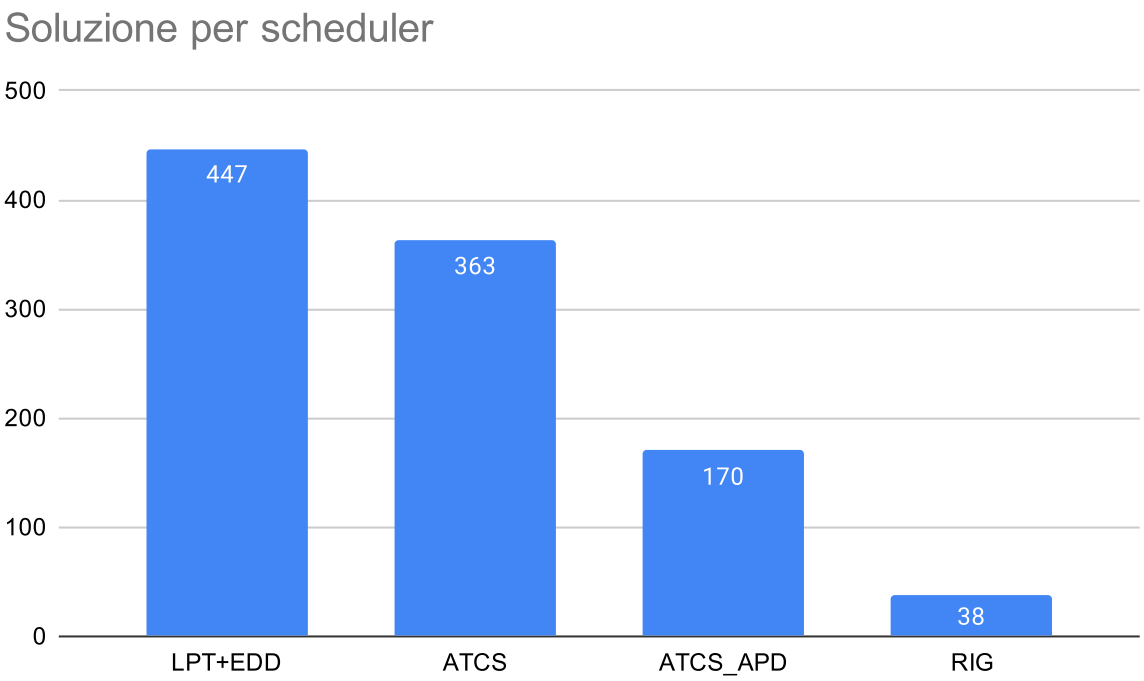

Figure 2: Soluzione calcolata per ogni scheduler

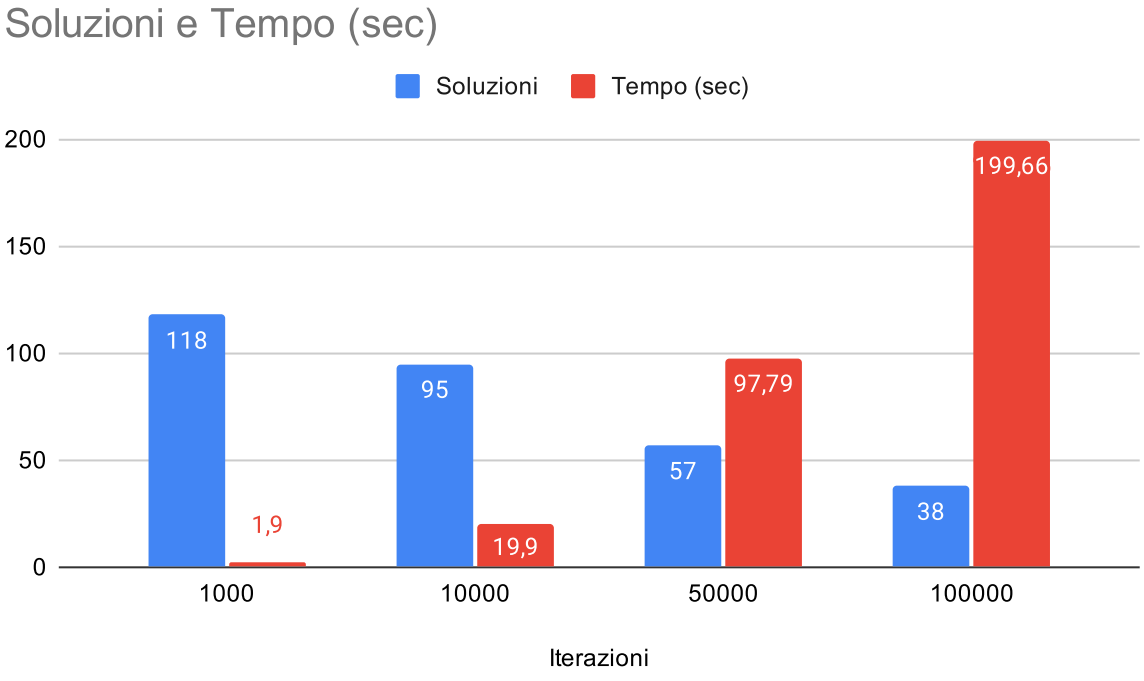

Figure 3: Soluzione e tempo di processamento di RIG per diversi valori di iterazioni_max