INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Méta-Heuristiques

Flexible Job Shop Scheduling Problem

Samy Barrech
Jonathan Poncy

Table des matières

1 INTRODUCTION

1.1 Présentation du projet

Dans ce projet, nous considérons un problème d'ordonnancement appelé "Flexible Job Shop". Nous disposons d'un ensemble de n travaux (jobs) devant être exécutés sur m machines. Chaque job se décompose en une liste d'activités qui doivent être réalisées dans l'ordre. Une activité est décrite par un ensemble d'opération de durée et de machine différente et il faut choisir l'opération qui minimise la durée totale que nécessite l'ensemble des jobs pour être terminé.

On supposera que les machines ne peuvent réaliser qu'une opération à la fois bien que la solution que nous proposons dans le cas d'une recherche à l'aide d'une heuristique, les machines peuvent supporter plusieurs opérations en simultané. Dans le cas de l'approche génétique, ce nombre est fixé à une seule opération en simultané. On suppose également qu'il y a également au moins deux jobs.

Une solution est admissible si elle respecte la contrainte d'ordre que l'on présentera plus tard dans ce rapport.

1.2 Structure des jeux de données

Voici un exemple de jeu de données :

```
3   3   1
3   2   2   4   1   3   1   2   2   1   3   2
3   1   1   2   1   3   1   1   2   4
2   1   2   4   1   3   3
```

La première ligne représente le nombre de jobs, le nombre de machines et enfin le nombre d'opérations que la machine peut réaliser en parallèle. Ensuite, on retrouve une ligne par job. Le premier nombre de la ligne représente le nombre d'activités pour ce job. Le second (que l'on va appeler $k \geq 1$) correspond à la liste des opérations que l'on peut choisir pour réaliser l'activité. On retrouve ensuite k paires correspondant au numéro de la machine et à la durée que prend l'opération, puis les données pour la seconde activité, etc.

1.3 Structure de notre projet

Le code des classes se retrouvent dans les fichiers *job.py*, *activity.py*, *operation.py* et *machine.py*. Nous ne rentrerons pas dans le détail de ces derniers, le code étant suffisament explicite.

Afin de pouvoir exécuter notre code, un certain nombre de librairies sont nécessaires. Il faudra installer *deap*, *colorama*, *termcolor*, *more-itertools*, *numpy* et *matplotlib*. Il est recommandé de les installer à l'aide du module *pip*.

1.3.1 Parsing des fichiers de données

```
def parse(path):
  with open(os.path.join(os.getcwd(), path), "r") as data:
    total_jobs, total_machines, max_operations = re.findall('\S+', data.readline
        ↪ ())
    number_total_jobs, number_total_machines, number_max_operations = int(
        ↪ total_jobs), int(total_machines), int(float(
      max_operations))
    jobs_list = []
    # Current job's id
    id_job = 1

    for key, line in enumerate(data):
      if key >= number_total_jobs:
        break
      # Split data with multiple spaces as separator
      parsed_line = re.findall('\S+', line)
      # Current job
      job = Job(id_job)
      # Current activity's id
      id_activity = 1
      # Current item of the parsed line
      i = 1

      while i < len(parsed_line):
        # Total number of operations for the activity
        number_operations = int(parsed_line[i])
        # Current activity
        activity = Activity(job, id_activity)
        for id_operation in range(1, number_operations + 1):
          activity.add_operation(Operation(id_operation, int(parsed_line[i + 2 *
              ↪  id_operation - 1]),
                         int(parsed_line[i + 2 * id_operation])))

        job.add_activity(activity)
        i += 1 + 2 * number_operations
        id_activity += 1

      jobs_list.append(job)
      id_job += 1

  # Machines
  machines_list = []
  for id_machine in range(1, number_total_machines + 1):
    machines_list.append(Machine(id_machine, number_max_operations))
```

1.3.2 Classe représentant les jobs

Un Job se décompose comme un identifiant, la liste des activités à réaliser et la liste des activités déjà réalisées.

1.3.3 Classe représentant les activités

Une activité est composée d'un pointeur vers le job auquel elle est rattachée, d'un identifiant d'activité, de la liste des opérations possibles et de l'opération réalisée.

1.3.4 Classe représentant les opérations

Une opération est décrite par un identifiant, une durée, l'identifiant de la machine qui doit procéder à l'opération, l'instant t où l'opération est commencée, un ordre d'arrivée dans la machine (utile pour dessiner le planning), et d'un booléen pour indiquer si l'opération est en cours de traitement.

1.3.5 Classe représentant les machines

Une machine a un identifiant, un booléen pour indiquer son état (travaille ou non), la liste des opérations réalisées par la machine, la liste des opérations en cours, un entier représentant le nombre maximal d'opérations en parallèle, un entier symbolisant l'échelle de temps, et une liste représentants les slots libres de la machine (utile pour dessiner le planning).

2 RÉSOLUTION À L'AIDE D'UNE HEURISTIQUE

2.1 Objectifs de la méthode

Cette méthode permet de trouver une borne supérieure de l'optimum très rapidement. Elle consiste à appliquer une heuristique au moment de choisir quelles opérations les machines vont traiter.

2.2 Algorithme du scheduler

2.2.1 Instanciation de la classe, paramètres de la méthode et initialisation

```
class Scheduler:
  def __init__(self, machines, max_operations, jobs):
    init()  # Init colorama for color display
    self.__original_stdout = sys.stdout
    self.__machines = machines
    self.__jobs_to_be_done = jobs
    self.__jobs_done = []
    self.__max_operations = max_operations
```

Le constructeur de la classe prend trois paramètres, la liste des machines, le nombre maximal d'opérations en parallèle et la liste des jobs construite avec le parseur. La variable *original_stdout* permet de supprimer la sortie texte et de la rétablir au besoin.

```
s = Scheduler(machines_list, number_max_operations, jobs_list)
```

Pour démarrer le scheduler, il suffit d'appeler la méthode *run* en passant en paramètre l'heuristique considérée et optionnelement mettre *verbose* à *True* ou *False* en fonction de si l'on veut un affichage ou non.

```
s.run(Heuristics.select_first_operation)
```

Lors de cet appel, la méthode *run* va commencer par supprimer la sortie standard si nécessaire et initialiser la variable *current_step* à 0. Cette variable représente l'instant t dans lequel le système se trouve.

```
  # Run the scheduler with an heuristic
  def run(self, heuristic, verbose=True):
    # Disable print if verbose is False
    if not verbose:
      sys.stdout = None

    current_step = 0
```

2.2.2 Déroulement de l'algorithme

```
while len(self.__jobs_to_be_done) > 0:
  current_step += 1

  best_candidates = heuristic(self.__jobs_to_be_done, self.__max_operations,
    ↪  current_step)
  for id_machine, candidates in best_candidates.items():
    machine = self.__machines[id_machine - 1]
    for activity, operation in candidates:
      if not (machine.is_working_at_max_capacity() or activity.is_pending):
        machine.add_operation(activity, operation)

  for machine in self.__machines:
    machine.work()

  for job in self.__jobs_to_be_done:
    if job.is_done:
      self.__jobs_to_be_done = list(
        filter(lambda element: element.id_job != job.id_job, self.
          ↪ __jobs_to_be_done))
      self.__jobs_done.append(job)
```

La variable *best_candidates* correspond au retour de l'heuristique considérée. Il s'agit d'un dictionnaire associant aux identifiants des machines la liste des opérations (ainsi que l'activité auxquelle elles sont rattachées) qu'elles devraient traiter à l'instant t.

La boucle *for* parcourant *best_candidates.items()* permet d'affecter l'opération sur la bonne machine.

La boucle *for* parcourant *self.__machines* permet de simuler le travail des machines pendant une unité de temps.

2.2.3 Fin de la méthode *run*

```
print(colored("[SCHEDULER]", "green"), "Done in " + str(current_step) + "
  ↪ units of time")

# Reenable stdout
if not verbose:
  sys.stdout = self.__original_stdout

return current_step
```

On affiche la durée totale que le planning prend et on réactive *std_out* si nécessaire.

2.3 Choisir l'opération la plus courte dans une activité comme heuristique

```
def select_first_operation(jobs_to_be_done, max_operations, _):
  best_candidates = {}
```

Lors de l'appel de cette heuristique, le paramètre de temps n'est pas utile, d'où la wild card utilisée et on commence par initialiser le dictionnaire *best_candidates*.

```
  for job in jobs_to_be_done:
    current_activity = job.current_activity
    best_operation = current_activity.shortest_operation

    if best_candidates.get(best_operation.id_machine) is None:
      best_candidates.update({best_operation.id_machine: [(current_activity,
         ↪ best_operation)]})
    elif len(best_candidates.get(best_operation.id_machine)) < max_operations:
      best_candidates.get(best_operation.id_machine).append((current_activity,
         ↪  best_operation))
    else:
      list_operations = best_candidates.get(best_operation.id_machine)

      for key, (_, operation) in enumerate(list_operations):
        if operation.duration < best_operation.duration:
          list_operations.pop(key)
          break

      if len(list_operations) < max_operations:
        list_operations.append((current_activity, best_operation))
```

Ensuite, pour chaque job qu'il reste à faire, on récupère l'activité en cours. Ici, on considère que la meilleure opération à choisir est celle qui a la durée la plus courte. On met à jour le dictionnaire en fonction de son état :

- Si la machine n'a aucune opération affectée, on ajoute la machine au dictionnaire avec l'activité et l'opération calculée précédemment.
- Si la machine a déjà des opérations affectées mais qu'elle ne travaille pas à capacité maximale, on ajoute l'activité et l'opération calculée précédemment à la liste
- Si la machine travaille à capacité maximale, on regarde s'il existe une opération de durée supérieure à celle calculée précédemment et si c'est le cas, on la remplace.

```
  return best_candidates
```

Enfin, on renvoie le dictionnaire calculé.

3 APPROCHE GÉNÉTIQUE

3.1 Que sont les algorithmes génétiques ?

Les algorithmes génétiques sont des méthodes évolutionnistes qui simulent une évolution naturelle, génération après génération, où les individus peuvent muter, se croiser ou se reproduire (cloner).

Ils permettent, lorsqu'il n'existe pas de méthode exacte (ou que la solution est inconnue), d'obtenir une solution approchée en un temps raisonnable.

Dans ce projet, nous nous basons sur une librairie Python qui s'appelle *Deap*.

3.2 Instanciation de la classe et lancement de la méthode

```
s = GeneticScheduler(machines_list, jobs_list)
```

Le constructeur de la classe prend comme arguments la liste des machines et la liste des jobs que renvoie le parseur.

```
s.run_genetic(total_population=10, max_generation=100, verbose=True)
```

Pour lancer l'algorithme, la métode *run_genetic* prend comme paramètres le nombre total d'individus dans la population, la génération maximale et si l'utilisateur souhaite activer la sortie standard ou non. Tous ces paramètres sont facultatifs et valent respectivement 10, 100 et *True* par défaut.

3.3 Création d'un individu

```
def init_individual(self, ind_class, size):
    temp_jobs_list = copy.deepcopy(self.__jobs)
    temp_machines_list = copy.deepcopy(self.__machines)
```

Afin de trouver un individu initial, nous réalisons une copie profonde de la liste des jobs et des machines. En effet, ces dernières seront modifiées par le *scheduler* de la partie précédente ce qui empêcherait toute autre simulation par la suite.

```
    # Run the scheduler
    s = Scheduler(temp_machines_list, 1, temp_jobs_list)
    s.run(Heuristics.random_operation_choice, verbose=False)
```

Le *scheduler* est ensuite appelé sur les listes temporaires précédemment créer en utilisant une heuristique de choix aléatoire pour les opérations afin d'augmenter la diversité de la population et éviter de converger vers un optimum local.

```
    # Retriving all the activities and the operation done
    list_activities = []
    for temp_job in temp_jobs_list:
```

```
    for temp_activity in temp_job.activities_done:
      activity = self.__jobs[temp_activity.id_job - 1].get_activity(
          ↪ temp_activity.id_activity)
      operation = activity.get_operation(temp_activity.operation_done.
          ↪ id_operation)
      list_activities.append((temp_activity.operation_done.time, activity,
          ↪ operation))
```

Comme l'opération a été réalisée sur des listes temporaires, il est nécessaire de faire la correspondance des activités et des opérations sur les listes initiales, les objets étant différents. Les couples *activity* et *operation* sont ensuite stockés dans une liste avec le temps auquel l'opération commence afin de pouvoir trier cette liste pour respecter la contrainte d'ordre.

```
  # Ordering activities by time
  list_activities = sorted(list_activities, key=lambda x: x[0])
  individual = [(activity, operation) for (_, activity, operation) in
      ↪ list_activities]
  del temp_jobs_list, temp_machines_list
  return ind_class(individual)
```

La liste est donc triée par rapport au temps, puis on supprime cette composante de la liste pour créer notre individu. Les variables temporaires sont détruites et l'individu retourné.

3.4 Création de la population initiale

```
# Initialize a population
def init_population(self, total_population):
  return [self.__toolbox.individual() for _ in range(total_population)]
```

Notre population initiale correspond à $total\_population$ éléments. Chaque élément correspond à un individu retourné par la méthode *init_individual*.

3.5 Evaluation de la fonction objectif d'un individu

```
def evaluate_individual(self, individual):
  return self.compute_time(individual)[0],
```

L'évaluation d'un individu est très simple. Comme nous souhaitons minimiser le temps total que mettent les jobs, l'évaluation correspond logiquement au temps que met l'individu pour terminer. Cette évaluation fait donc appel à la méthode *compute_time*.

Cette évaluation repose sur une astuce. Comme les activités sont classées par ordre chronologique, il suffit de regarder à quel instant t_1 termine l'activité précédente du job et à quel instant t_2 termine la dernière activité sur la machine concernée par l'opération couramment considérée. Il suffit ensuite de prendre le max entre t_1 et t_2 pour avoir l'instant t auquel l'activité commence.

```
def compute_time(self, individual):
  # List matching the activities to the time it takes place
  list_time = []
  # Operation schedule on machines indexed by machines' id
  schedule = {}
  for machine in self.__machines:
    schedule.update({machine.id_machine: []})
  # Operation done indexed by job's id
  operations_done = {}
  for job in self.__jobs:
    operations_done.update({job.id_job: []})
```

On initialise une liste contenant pour chaque activité le temps auquel elles commencent (utile pour la simulation finale) et deux dictionnaires, un premier faisant la correspondance entre $machine\_id$ et liste des opérations réalisées par cette mmachine, le deuxième faisant la même chose mais par rapport aux identifiants des jobs.

```
  # For each item in individual, we compute the actual time at which the
    ↪ operation considered start
  for activity, operation in individual:
    # Get at which time the previous operation is done
    time_last_operation, last_operation_job = operations_done.get(activity.
       ↪ id_job)[-1] if len(
      operations_done.get(activity.id_job)) > 0 else (0, None)
    time_last_machine, last_operation_machine = schedule.get(operation.
       ↪ id_machine)[-1] if len(
      schedule.get(operation.id_machine)) > 0 else (0, None)

    if last_operation_machine is None and last_operation_job is None:
      time = 0
    elif last_operation_machine is None:
      time = time_last_operation + last_operation_job.duration
    elif last_operation_job is None:
      time = time_last_machine + last_operation_machine.duration
    else:
      time = max(time_last_machine + last_operation_machine.duration,
             time_last_operation + last_operation_job.duration)

    list_time.append(time)

    operations_done.update({activity.id_job: operations_done.get(activity.
       ↪ id_job) + [(time, operation)]})
    schedule.update({operation.id_machine: schedule.get(operation.id_machine)
       ↪ + [(time, operation)]})
```

On calcule ensuite les instants de temps comme indiqué précédemment.

```
    # We compute the total time we need to process all the jobs
    total_time = 0
    for machine in self.__machines:
      if len(schedule.get(machine.id_machine)) > 0:
        time, operation = schedule.get(machine.id_machine)[-1]
        if time + operation.duration > total_time:
          total_time = time + operation.duration

    return total_time, list_time
```

Pour calculer le temps total que prend le planning, on regarde pour chaque machine le moment où la dernière opération commence ainsi que sa durée et on prend la somme maximale. Enfin, on renvoie la durée totale et la liste des instants temporels.

3.6 Mutation d'un individu

```
  def mutate_individual(individual):
    # Select the possible candidates, meaning the activities with multiple
       ↪ choices for an operation
    candidates = list(filter(lambda element: len(element[0].next_operations) >
       ↪ 1, individual))
```

Pour réaliser la mutation d'un individu, on regarde les activités qui peuvent muter, autrement dit celles où plusieurs opérations sont possibles pour les réaliser.

```
    if len(candidates) > 0:
      mutant_activity, previous_operation = candidates[random.randint(0, len(
         ↪ candidates) - 1)]
      id_mutant_activity = [element[0] for element in individual].index(
         ↪ mutant_activity)
      mutant_operation = previous_operation
      while mutant_operation.id_operation == previous_operation.id_operation:
        mutant_operation = mutant_activity.next_operations[
          random.randint(0, len(mutant_activity.next_operations) - 1)]
      individual[id_mutant_activity] = (mutant_activity, mutant_operation)
```

S'il existe de telles activités, on en choisit une au hasard, et pour cette activité on choisit une opération différente.

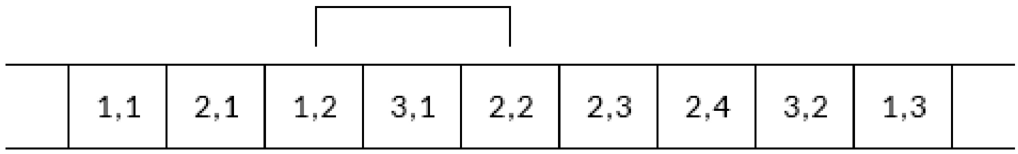
```
    # Remove the previous fitness value because it is deprecated
    del individual.fitness.values
    # Return the mutant
    return individual
```

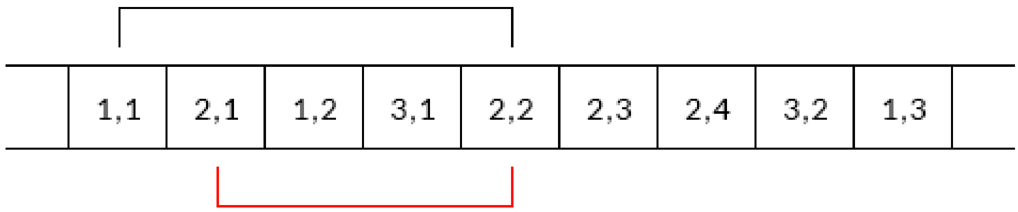
On supprime la composante *fitness* (qui correspond à la fonction objective) car sa valeur n'est plus à jour et on renvoie le mutant.

3.7 Permutation sur un individu

Une permutation sur un individu correspond à une permutation de deux couples activité/opération ne violant pas la contrainte d'ordre.

Exemple de planning en identifiant les activités par *(id_job, id_activity)* :





Par exemple, la première permutation est valide car elle respecte la contrainte d'ordre. Cependant, pour la deuxième, la contrainte d'ordre ne serait plus respectée car si j'échange le $(1,1)$ et le $(2,2)$, alors on aurait $(2,2)$ avant $(2,1)$ ce qui est impossible, de même avec $(1,1)$ et $(1,2)$.

```
def compute_bounds(permutation, considered_index):
  considered_activity, _ = permutation[considered_index]
  min_index = key = 0
  max_index = len(permutation) - 1
  while key < max_index:
    activity, _ = permutation[key]
    if activity.id_job == considered_activity.id_job:
      if min_index < key < considered_index:
        min_index = key
      if considered_index < key < max_index:
        max_index = key
    key += 1
  return min_index, max_index
```

Cette méthode permet de calculer les bornes min et max de permutation pour une activité donnée, autrement dit l'intervalle où l'on peut déplacer une activité sans que cela viole la contrainte d'ordre.

```
def permute_individual(self, individual):
  permutation_possible = False
  considered_index = considered_permutation_index = 0
  while not permutation_possible:
    considered_index = min_index = max_index = 0
    # Loop until we can make some moves, i.e. when max_index - min_index > 2
    while max_index - min_index <= 2:
      considered_index = random.randint(0, len(individual) - 1)
      min_index, max_index = self.compute_bounds(individual, considered_index)

    # Select a random activity within those bounds (excluded) to permute with
    considered_permutation_index = random.randint(min_index + 1, max_index -
       ↪ 1)
    min_index_permutation, max_index_permutation = self.compute_bounds(
       ↪ individual,
                                                   considered_permutation_index)
    if min_index_permutation < considered_index < max_index_permutation:
      permutation_possible = considered_index != considered_permutation_index

  # A possible permutation has been found
  individual[considered_index], individual[considered_permutation_index] =
     ↪ individual[
                                                     considered_permutation_index], \
                                                   individual[considered_index]
  return individual
```

Tant que l'on a pas trouvé une permutation possible, on sélectionne deux activités au hasard et on vérifie si elles sont permutables, c'est à dire si chacune appartient au domaine de permutation de l'autre. Une fois cette permutation trouvée, elle est effectuée et le nouvel individu est renvoyé.

Il est à noter qu'une permutation est forcément possible à partir du moment où le nombre de jobs est au moins égal à deux.

3.8 Déplacement d'une activité au sein d'un individu

```
def move_individual(self, individual):
  considered_index = min_index = max_index = 0
  # Loop until we can make some moves, i.e. when max_index - min_index > 2
  while max_index - min_index <= 2:
    considered_index = random.randint(0, len(individual) - 1)
    min_index, max_index = self.compute_bounds(individual, considered_index)
  # Loop until we find a different index to move to
  new_index = random.randint(min_index + 1, max_index - 1)
  while considered_index == new_index:
    new_index = random.randint(min_index + 1, max_index - 1)
  # Move the activity inside the scheduler
  individual.insert(new_index, individual.pop(considered_index))
  return individual
```

Plutôt que de permuter deux activités, il est également possible de choisir une activité, de calculer ses bornes de déplacements valides et de la déplacer dans cet intervalle. Cette opération diffère de la permutation dans le sens où l'ordre des autres éléments n'est pas modifié.

3.9 Évolution d'un individu

```
def evolve_individual(self, individual, mutation_probability,
    ↪ permutation_probability, move_probability):
  future_individual = copy.deepcopy(individual)
  if random.randint(0, 100) < mutation_probability:
    future_individual = self.mutate_individual(future_individual)
  if random.randint(0, 100) < permutation_probability:
    future_individual = self.permute_individual(future_individual)
  if random.randint(0, 100) < move_probability:
    future_individual = self.move_individual(future_individual)
  return future_individual
```

Un individu donné peut muter, subir une permutation, subir un déplacement ou une combinaison de ces opérations. On calcule pour cela trois probabilités, une par opération possible.

3.10 Réduction d'une population par tournoi

Lors du passage de la génération n à $n+1$, les individus résultant d'une évolution sont ajoutés à la liste contenant la population. On se retrouve dans une situation de surpeuplement car la taille de cette liste dépasse $total\_population$. Les individus vont donc participer à un tournoi jusqu'à retrouver une taille de population acceptable.

```
def run_tournament(population, total=10):
  # Because you can't have a bigger population as a result of the tournament,
      ↪ we assert that constraint
  assert total <= len(population)
  new_population = []
  while len(new_population) < total:
    first_individual = population[random.randint(0, len(population) - 1)]
    second_individual = population[random.randint(0, len(population) - 1)]
    if first_individual.fitness.values[0] < second_individual.fitness.values
        ↪ [0]:
      new_population.append(first_individual)
      population.remove(first_individual)
    else:
      new_population.append(second_individual)
      population.remove(second_individual)
  del population
  return new_population
```

Pour cela, on choisit aléatoirement deux individus de cette population. Celui dont la fonction objective est la plus faible remporte le tournoi et est ajouté à la liste des survivants (représentée par *new_population*). Cette opération va se répéter jusqu'à ce que la taille de la population de survivants soit égale à $total$.

3.11 Simulation sur les machines d'un individu

```
def run_simulation(self, individual):
  total_time, list_time = self.compute_time(individual)
  for key, (individual_activity, individual_operation) in enumerate(individual
      ↪ ):
    activity = self.__jobs[individual_activity.id_job - 1].get_activity(
        ↪ individual_activity.id_activity)
    operation = activity.get_operation(individual_operation.id_operation)
    operation.time = list_time[key]
    operation.place_of_arrival = 0
    activity.terminate_operation(operation)
  return total_time
```

Pour un individu donné, on simule son exécution afin de pouvoir le dessiner éventuellement.

3.12 Déroulement de l'algorithme

```
def run_genetic(self, total_population=10, max_generation=100, verbose=False):
  assert total_population > 0, max_generation > 0
  # Disable print if verbose is False
  if not verbose:
    sys.stdout = None

  creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
  creator.create("Individual", list, fitness=creator.FitnessMin)

  self.__toolbox.register("individual", self.init_individual, creator.
     ↪ Individual, size=1)
  self.__toolbox.register("mutate", self.mutate_individual)
  self.__toolbox.register("permute", self.permute_individual)
  self.__toolbox.register("evaluate", self.evaluate_individual)
```

Lors de l'appel de la méthode *run_genetic*, on commence par désactiver la sortie standard si demandé puis on initialise la libraire *Deap*. On indique que l'on cherche à minimiser la fonction objectif. On enregistre également auprès de *Deap* nos fonctions de création, de mutation, de permutation et d'évolution.

```
  print(colored("[GENETIC]", "cyan"), "Generating population")
  population = self.init_population(total_population)
```

On crée ensuite notre population initiale.

```
  best = population[0]
  best.fitness.values = self.evaluate_individual(best)
  print(colored("[GENETIC]", "cyan"), "Starting evolution for", max_generation
     ↪ , "generations")
  for current_generation in range(max_generation):
    # Generate mutation and permutation probabilities for the next generation
    mutation_probability = random.randint(0, 100)
    permutation_probability = random.randint(0, 100)
    move_probability = random.randint(0, 100)
    # Evolve the population
    print(colored("[GENETIC]", "cyan"), "Evolving to generation",
       ↪ current_generation + 1)
    mutants = list(set([random.randint(0, total_population - 1) for _ in
              range(random.randint(1, total_population))]))
    print(colored("[GENETIC]", "cyan"), "For this generation,", len(mutants),
       ↪ "individual(s) will mutate")
    for key in mutants:
      individual = population[key]
      population.append(
        self.evolve_individual(individual, mutation_probability,
           ↪ permutation_probability, move_probability))
```

```
    # Evaluate the entire population
    fitnesses = list(map(self.evaluate_individual, population))
    for ind, fit in zip(population, fitnesses):
      ind.fitness.values = fit
      if best.fitness.values[0] > ind.fitness.values[0]:
        print(colored("[GENETIC]", "cyan"), "A better individual has been
            ↪ found. New best time = ",
            ind.fitness.values[0])
        best = copy.deepcopy(ind)
    population = self.run_tournament(population, total=total_population)
```

On simule l'évolution pour $max\_generation$ générations. On définit les probabilités de mutation, de permutation et de déplacement puis on fait évoluer notre population et enfin on évalue la fonction objectif de chaque individu. Si un individu a une fonction objectif plus faible que *best*, alors cet individu devient le meilleur individu.

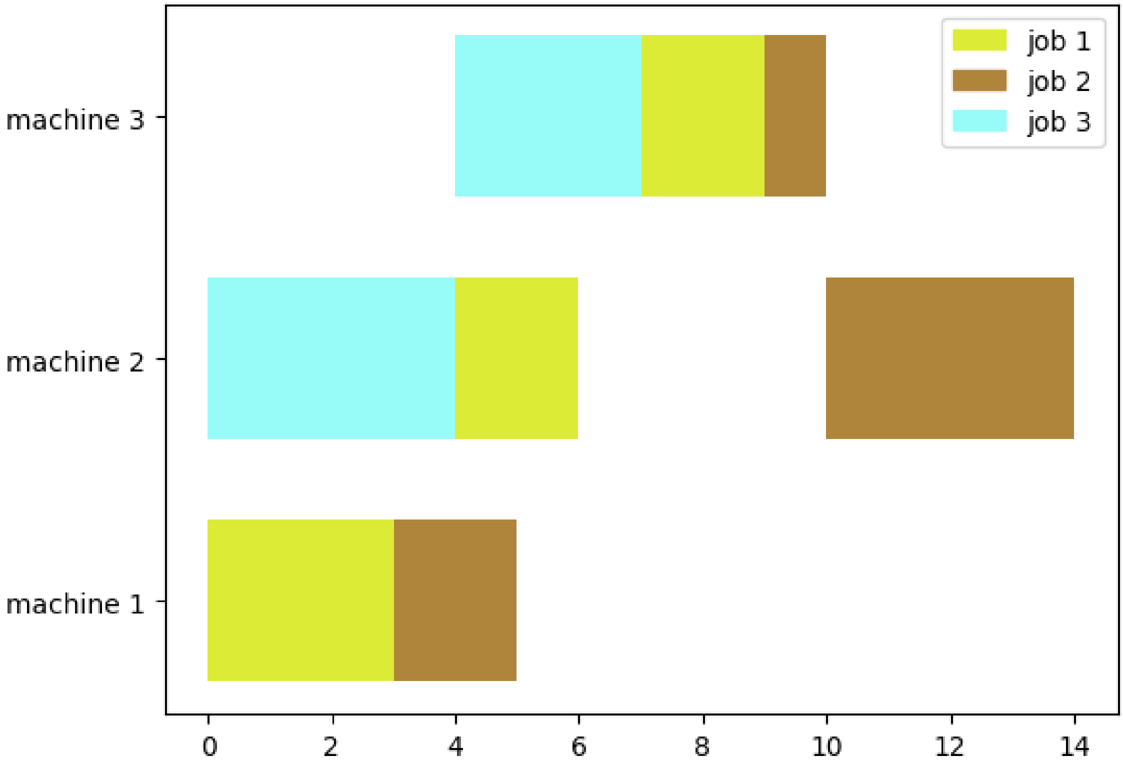
```
  print(colored("[GENETIC]", "cyan"), "Evolution finished")
  if self.constraint_order_respected(best):
    print(colored("[GENETIC]", "cyan"), "Best time found equals", best.fitness
        ↪ .values[0])
    print(colored("[GENETIC]", "cyan"), "Simulating work on machines")
    total_time = self.run_simulation(best)
    print(colored("[GENETIC]", "cyan"), "Simulation finished")
    print(colored("[GENETIC]", "cyan"), "Genetic scheduler finished")
  else:
    print(colored("[GENETIC]", "cyan"), "The individual doesn't match the
        ↪ constraint order")

  # Reenable stdout
  if not verbose:
    sys.stdout = self.__original_stdout

  return total_time
```

On affiche les résultats et on simule le meilleur planning trouvé. On réactive la sortie standard si besoin.

4 RÉSULTATS ET COMPARAISON DES MÉTHODES

4.1 Résultats avec un jeu de données simple

4.1.1 Recherche à l'aide d'une heuristique

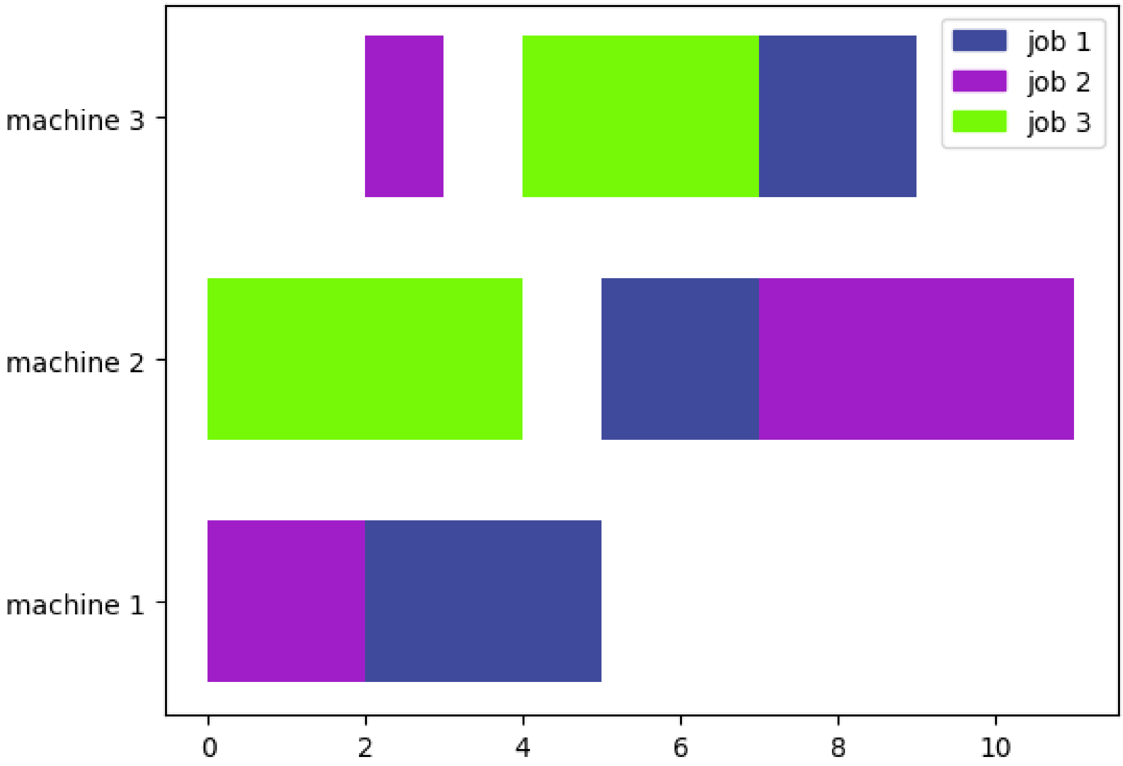


Résultats

Done in 14 units of time

Total time : 0.0033700000058161095 seconds

4.1.2 Recherche à l'aide de l'algorithme génétique



Résultats

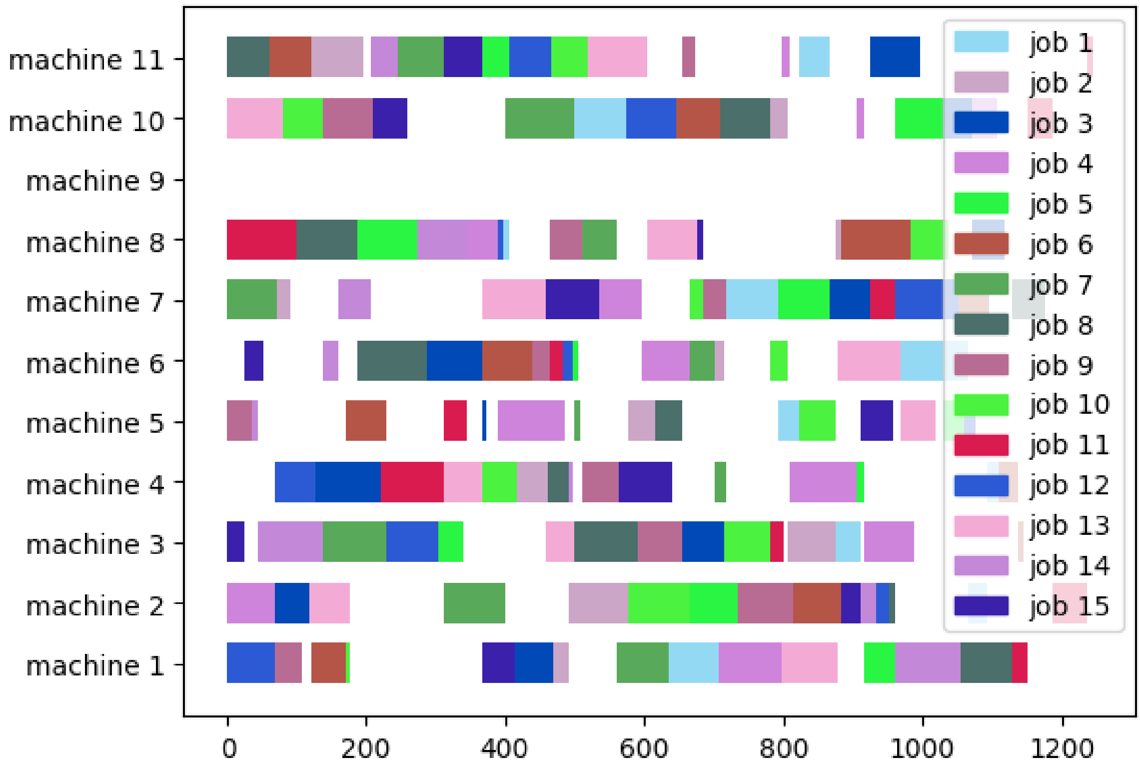
Population : 10, Max Generation : 100

Done in 11 units of time

Total time : 2.63803599998937 seconds

4.2 Barnes - setb4c9

4.2.1 Recherche à l'aide d'une heuristique

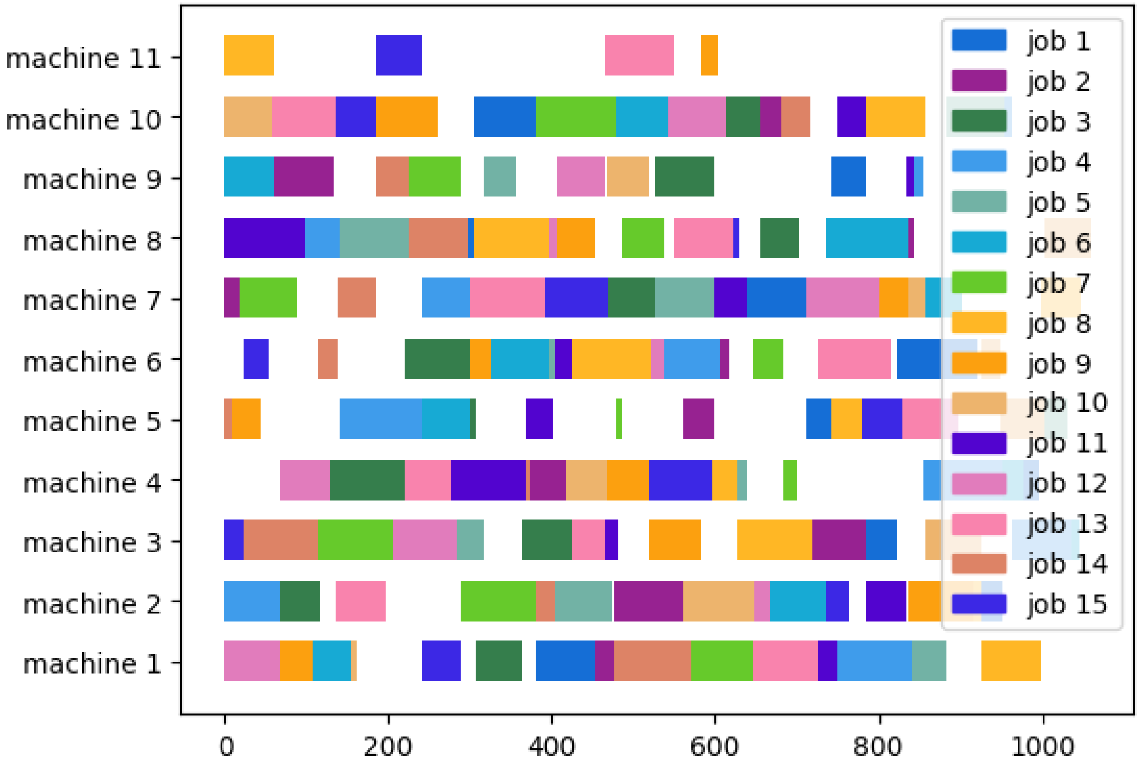


Résultats

Done in 1245 units of time

Total time : 0.04628799999773037 seconds

4.2.2 Recherche à l'aide de l'algorithme génétique

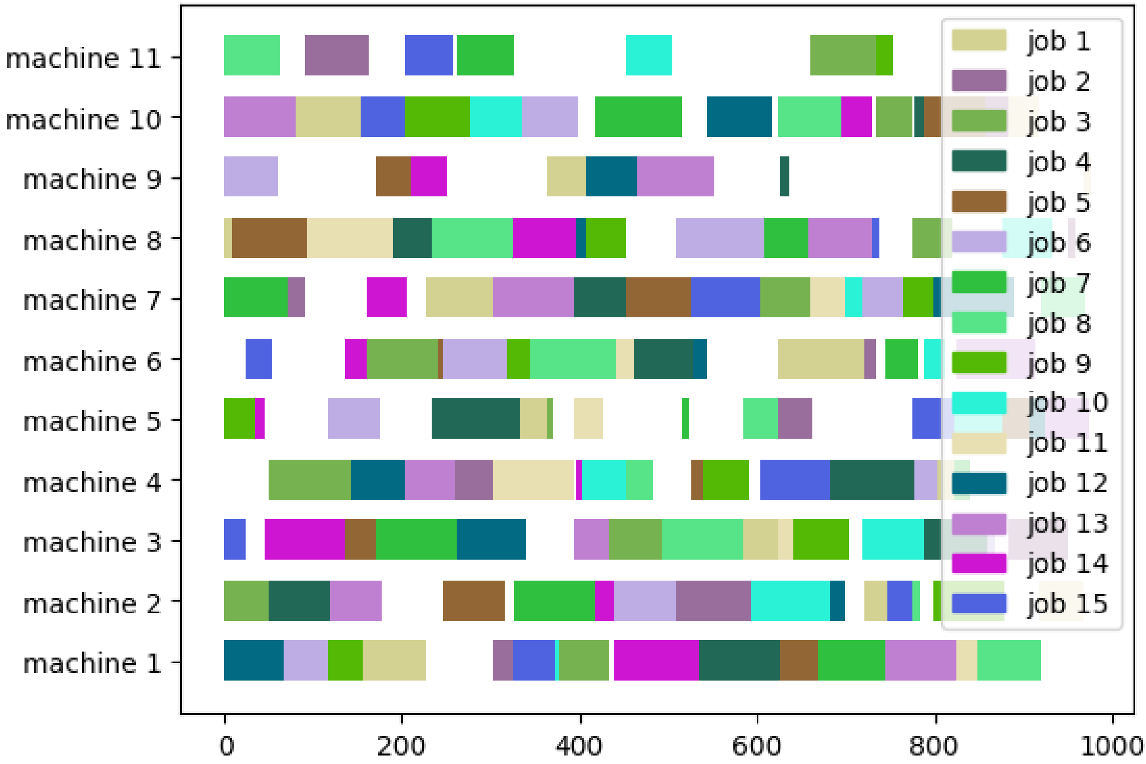


Résultats

Population : 10, Max Generation : 100

Done in 1059 units of time

Total time : 3.763153999999304 seconds



Résultats

Population : 200, Max Generation : 400

Done in 976 units of time

Total time : 198.2230869999994 seconds

5 BENCHMARKS DE L'APPROCHE GÉNÉTIQUE

Il s'agira dans cette partie d'évaluer le temps que met notre approche génétique en fonction des paramètres d'entrée. Il peut en effet être intéressant de savoir si notre algorithme évolue de façon linéaire ou exponentielle par rapport à ces derniers.

Le fichier de données considéré correspond au *Barnes - setb4c9*. En effet, il comporte 15 jobs et 11 machines ce qui le rend intéressant de par la taille du problème.

Le code servant à réaliser les benchmarks est disponible dans le fichier *benchmarks.py*.

5.1 Principe de benchmarking

5.1.1 Instanciation de la classe

```
class Benchmarks:
  def __init__(self, path, start=0, stop=4, samples=20):
    init()  # Init colorama for color display
    self.__size = list(set(np.logspace(start, stop, num=samples, dtype=np.int)))
    self.__name = path.split('/')[-1].split('.')[0]
    self.__jobs_list, self.__machines_list, self.__number_max_operations = parse
        ↪ (path)
```

On crée une liste de $samples$ valeurs entre $start$ et $stop$ espacées de façon logarithmique. Ceci permet d'otenir beaucoup de valuers faibles et moins de valeurs importantes ce qui est intéressant pour comparer la fonction objective pour différents couples taille de population et génération maximale par la suite.

5.1.2 Benchmarks en fonction de la taille de la population

```
  # Benchmarks the genetic scheduler when we increase total population
  def population(self, max_generation=100):
    benchmarks_population = []

    print(colored("[BENCHMARKS]", "yellow"), "Gathering computation time for
        ↪ different population sizes")
    for size in self.__size:
      print(colored("[BENCHMARKS]", "yellow"), "Current population size =", size
          ↪ )
      start = timeit.default_timer()
      temp_machines_list, temp_jobs_list = copy.deepcopy(self.__machines_list),
          ↪ copy.deepcopy(self.__jobs_list)
      s = GeneticScheduler(temp_machines_list, temp_jobs_list)
      total_time = s.run_genetic(total_population=size, max_generation=
          ↪ max_generation, verbose=False)
      stop = timeit.default_timer()
```

```
        print(colored("[BENCHMARKS]", "yellow"), "Done in", stop - start, "seconds
            ↪ ")
        benchmarks_population.append((size, max_generation, stop - start,
            ↪ total_time))
        del s, temp_machines_list, temp_jobs_list
    print(colored("[BENCHMARKS]", "yellow"), "Gathering for different population
        ↪  sizes completed")

    Drawer.plot2d(self.__name + "_benchmarks_population", [element[0] for
        ↪ element in benchmarks_population],
            [element[2] for element in benchmarks_population],
            "Time as a function of population size for " + self.__name + " (" +
                ↪ str(
              max_generation) + " generations)", "Population size", "Time (in
                  ↪ seconds)", approximate=True)

    return benchmarks_population
```

Cette fonction permet d'étudier la complexité temporelle en fonction de la taille des populations à génération maximale fixée (ici 100 par défaut).

On notera la présence du paramètre *approximate* lors de l'appel à *Drawer.plot2d* qui permettra d'approximer la courbe obtenue. Les raisons seront données dans la suite de ce rapport.

5.1.3 Benchmarks en fonction de la génération maximale

```
# Benchmarks the genetic scheduler when we increase max generation
def generation(self, population_size=100):
    benchmarks_generation = []

    print(colored("[BENCHMARKS]", "yellow"), "Gathering computation time for
        ↪ different generation numbers")
    for size in self.__size:
        print(colored("[BENCHMARKS]", "yellow"), "Current max generation =", size)
        start = timeit.default_timer()
        temp_machines_list, temp_jobs_list = copy.deepcopy(self.__machines_list),
            ↪ copy.deepcopy(
          self.__jobs_list)
        s = GeneticScheduler(temp_machines_list, temp_jobs_list)
        total_time = s.run_genetic(total_population=population_size,
            ↪ max_generation=size, verbose=False)
        stop = timeit.default_timer()
        print(colored("[BENCHMARKS]", "yellow"), "Done in", stop - start, "seconds
            ↪ ")
        benchmarks_generation.append((population_size, size, stop - start,
            ↪ total_time))
        del s, temp_machines_list, temp_jobs_list
    print(colored("[BENCHMARKS]", "yellow"), "Gathering for different population
```

```
    ↪  sizes completed")

  Drawer.plot2d(self.__name + "_benchmarks_generation", [element[1] for
    ↪ element in benchmarks_generation],
        [element[2] for element in benchmarks_generation],
        "Time as a function of max generation for " + self.__name + " (" +
            ↪ str(
          population_size) + " individuals)", "Max generation", "Time (in
              ↪ seconds)")

  return benchmarks_generation
```

Cette fonction permet d'étudier la complexité temporelle en fonction de la génération maximale à taille de population fixée (ici 100 par défaut).

5.1.4 Benchmarks en fonction de la taille de la population et de la génération maximale

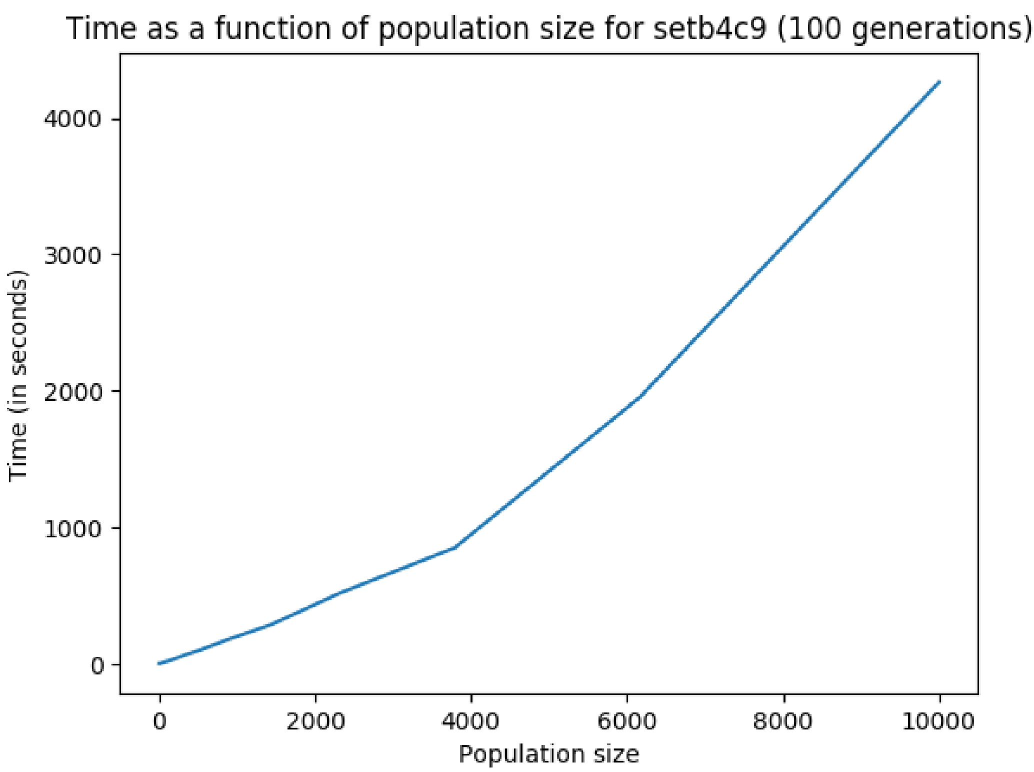
```
# Benchmarks the genetic scheduler for different couples of population size
  ↪ and max generation
def population_and_generation(self):
  import itertools
  benchmarks_population_and_generation = []
  params = itertools.product(self.__size, repeat=2)
  print(colored("[BENCHMARKS]", "yellow"),
      "Gathering times for different couples of population size and max
        ↪ generation")
  for population, generation in params:
    print(colored("[BENCHMARKS]", "yellow"), "Current population size =",
      ↪ population, ", max generation =",
        generation)
    start = timeit.default_timer()
    temp_machines_list, temp_jobs_list = copy.deepcopy(self.__machines_list),
      ↪ copy.deepcopy(
      self.__jobs_list)
    s = GeneticScheduler(temp_machines_list, temp_jobs_list)
    total_time = s.run_genetic(total_population=population, max_generation=
      ↪ generation, verbose=False)
    stop = timeit.default_timer()
    print(colored("[BENCHMARKS]", "yellow"), "Done in", stop - start, "seconds
      ↪ ")
    benchmarks_population_and_generation.append((population, generation, stop
      ↪ - start, total_time))
    del s, temp_machines_list, temp_jobs_list
  print(colored("[BENCHMARKS]", "yellow"), "Gathering for different couples
    ↪ completed")

  # Plot graph with solution time as Z axis
  Drawer.plot3d(self.__name + "_benchmarks_generation_with_solution_time",
```

```
            [element[0] for element in benchmarks_population_and_generation],
            [element[1] for element in benchmarks_population_and_generation],
            [element[3] for element in benchmarks_population_and_generation],
            "Best time found as a function of population size and max generation
               ↪ ", "Population size",
            "Max generation", "Total time")

    # Plot graph with computation time as Z axis
    Drawer.plot3d(self.__name + "_benchmarks_generation_with_computation_time",
            [element[0] for element in benchmarks_population_and_generation],
            [element[1] for element in benchmarks_population_and_generation],
            [element[2] for element in benchmarks_population_and_generation],
            "Computation time as a function of population size and max
               ↪ generation", "Population size",
            "Max generation", "Computation time")

    return benchmarks_population_and_generation
```

Cette fonction permet d'étudier la complexité temporelle en fonction de la taille des populations et de la génération maximale mais aussi l'évolution de la fonction objective du meilleur individu en fonction de ces mêmes paramètres.

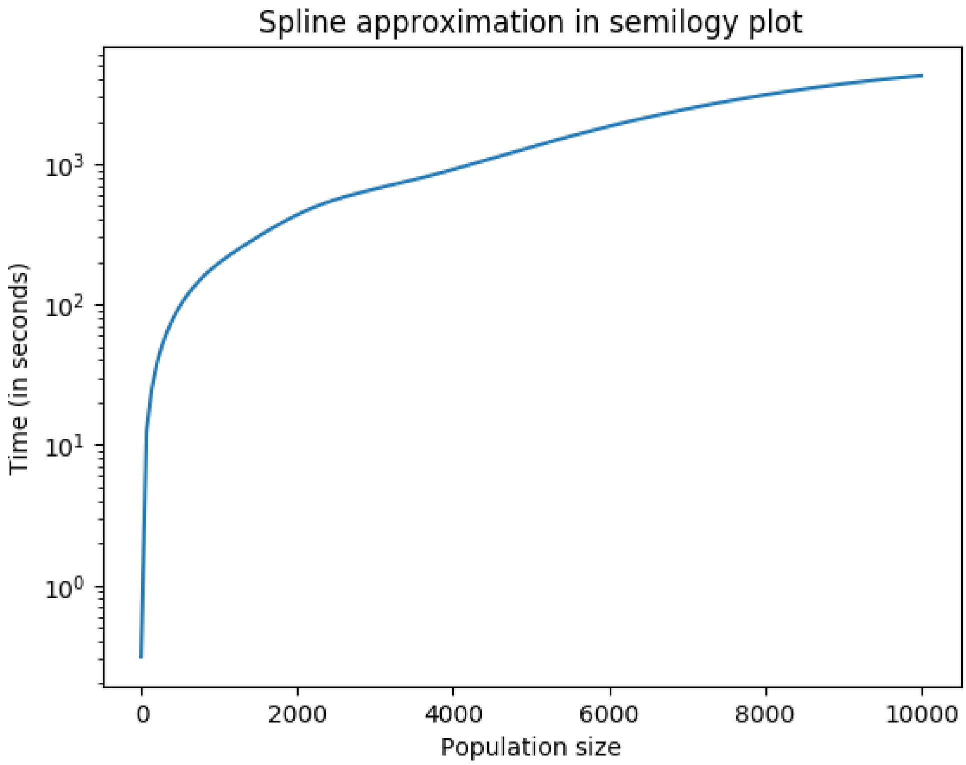
5.2 Comparaison du temps d'exécution par rapport à la taille de la population

La génération maximale est ici fixée à 100. Nous faisons évoluer la taille de la population de façon logarithmique entre 1 et 10000.



A $max\_generation$ fixé, le temps de calcul n'est pas linéaire en fonction du nombre d'individus dans la population. Il est intéressant de savoir s'il est polynomial ou exponentiel.

Afin d'obtenir de meilleurs résultats, on commence par faire une interpolation par spline de notre ensemble de points. Une spline est une fonction polynomiale par morceaux et ce type d'interpolation locale est plus efficace qu'une interpolation globale. Nous ne rentrerons pas plus dans les détails de ce qu'est une spline, la littérature sur internet étant très complète à ce sujet. Cette spline permettra de densifier notre ensemble de points de façon artificielle et sera utile pour calculer des résidus.



En utilisant une échelle logarithmique pour les ordonnées, on peut déjà éliminer l'hypothèse d'une complexité temporelle exponentielle. En effet, dans un tel repère, les représentations graphiques des fonctions exponentielles sont des droites.

Maintenant que ce point est réglé, évaluons différentes interpolations polynomiales.

```
[BENCHMARKS] Gathering for different population sizes completed
[DRAWER] Polynomial approximation of degree 2 -> Residual =  1085.752897233403
[DRAWER] Polynomial approximation of degree 3 -> Residual =  1194.3432612101085
[DRAWER] Polynomial approximation of degree 4 -> Residual =  307.8303484796994
[DRAWER] Polynomial approximation of degree 5 -> Residual =  2185.608299190138
[DRAWER] Polynomial approximation of degree 6 -> Residual =  10461.196494441645
[DRAWER] Polynomial approximation of degree 7 -> Residual =  67400.98242831718
[DRAWER] Polynomial approximation of degree 8 -> Residual =  396669.4412198472
[DRAWER] Best approximation found is a polynomial of degree 4
         Coefficients: [-7.42871044e-13  1.37393298e-08 -4.51060640e-05  2.46547342e-01
 -4.06355322e+00]
         Residual: 307.8303484796994
```

Soit *xdata* la liste des tailles de population évaluées par l'algorithme de benchmarks. On commence par construire une liste de 150 points compris entre $xdata[0]$ et $xdata[-1]$, autrement dit la première et la dernière valeur de la liste *xdata*.

```
x = np.linspace(xdata[0], xdata[-1], 150)
```

On évalue ainsi le résidu pour des polynômes de degrés différents, ici entre 2 et 8. Le résidu correspond à la somme des norme 2 de la différence entre notre approximation polynomiale et les valeurs de la spline :

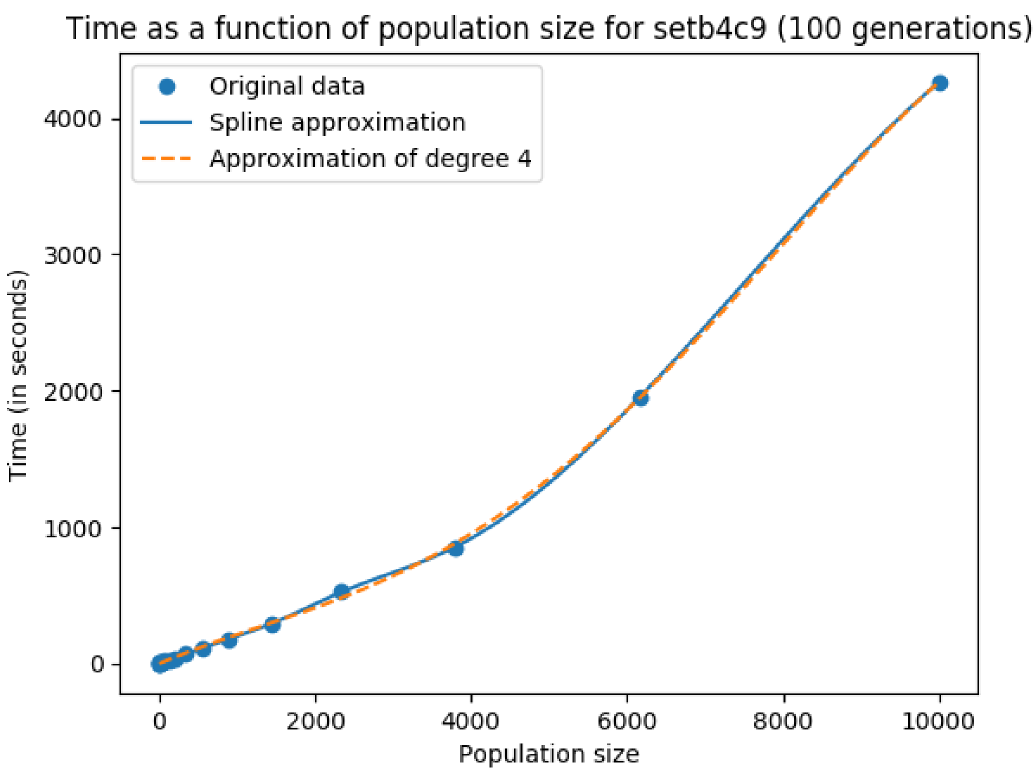
$$r_i = \sum_{j=1}^{150} \left\| s(x_j) - p_i(x_j) \right\|$$

Ici, r_i correspond au résidu du polynôme de degré i, s correspond à notre spline et p_i à la fonction polynomiale de degré i obtenue de la façon suivante :

```
# Find a polynomial to fit the spline
coefficients = np.polyfit(xdata, ydata, i)
p_i = np.poly1d(coefficients)
```

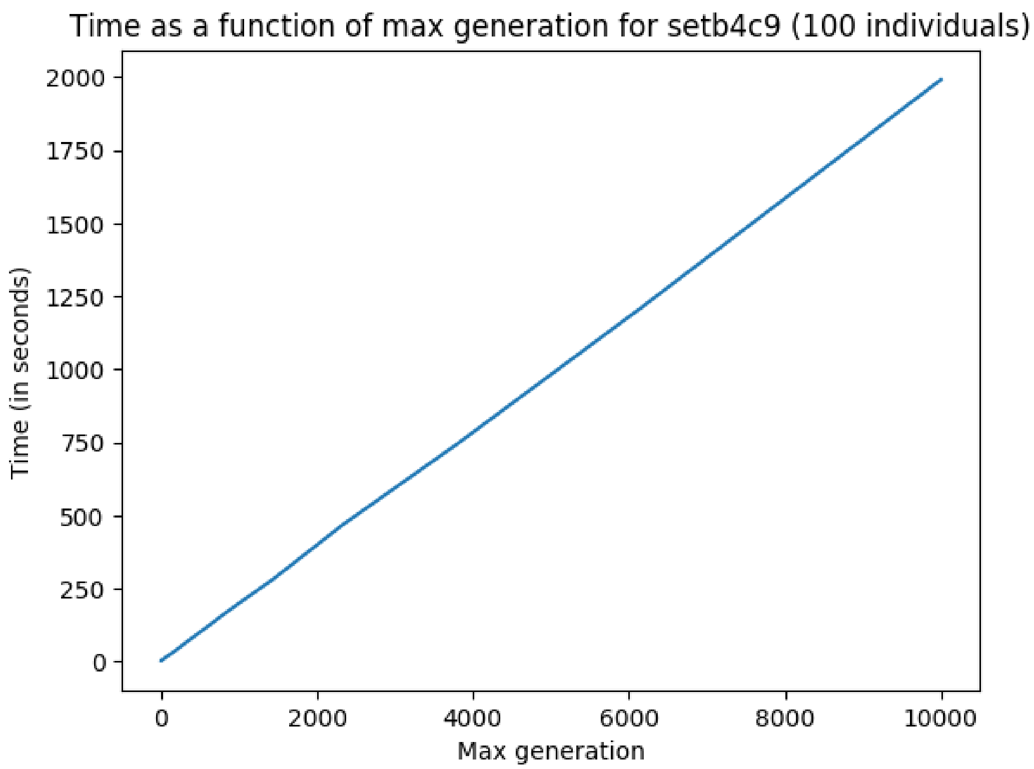
Nous avons fait le choix de construire l'interpolation polynomiale en se basant uniquement sur les points calculés par l'algorithme de benchmarks et non sur la spline. Les calculs sont plus rapides car il y a moins de points et la perte de précision dans le calcul des coefficients est négligeable, ces derniers nous intéressant peu.

Le résidu le plus faible est obtenu pour un polynôme de degré 4, notre complexité temporelle en fonction de la taille de la population est donc en $\mathcal{O}(n^4)$.



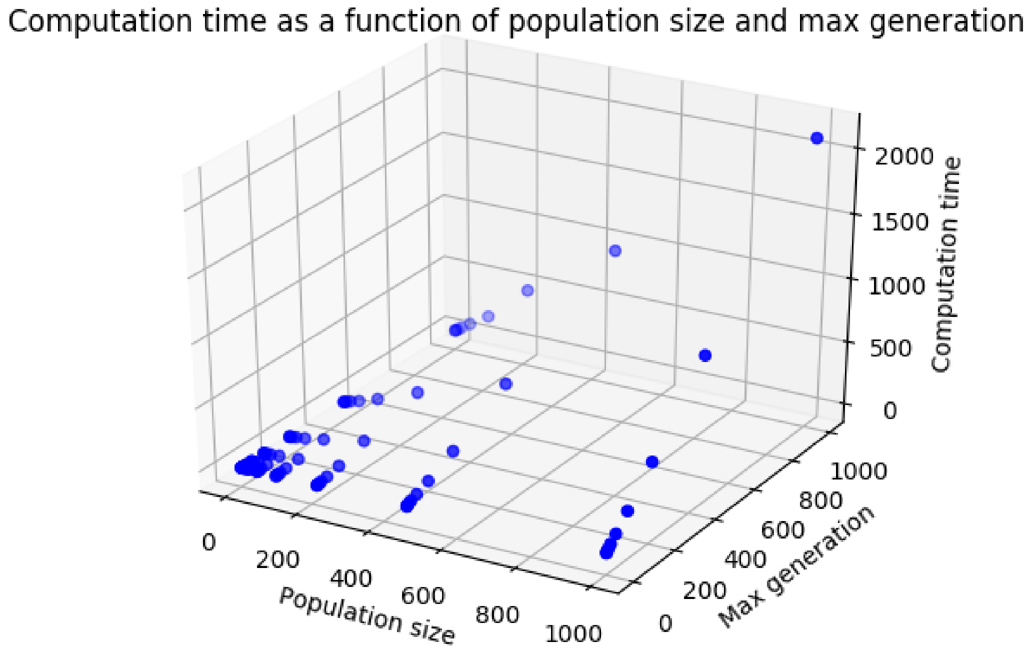
5.3 Comparaison du temps d'exécution par rapport à la génération maximale

La taille de la population est ici fixée à 100. Nous faisons évoluer la génération maximale de façon logarithmique entre 1 et 10000.



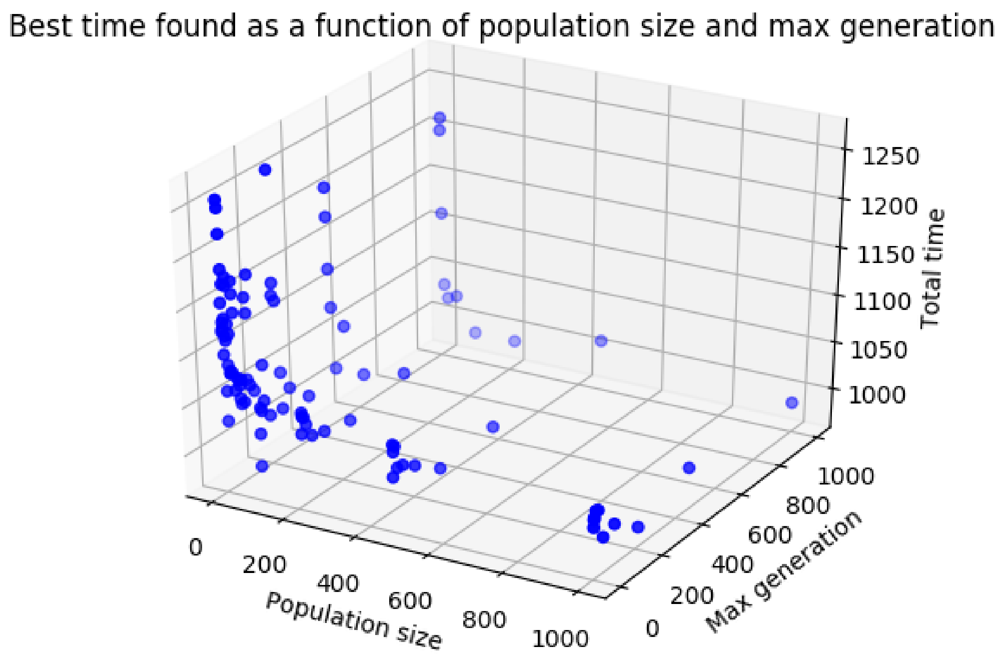
A $population\_size$ fixé, le temps de calcul est linéaire en fonction de la génération maximale. Ce résultat n'est pas surprenant, mais il nous semblait important de le vérifier.

5.4 Comparaison du temps d'exécution par rapport à la taille de la population et la génération maximale



On retrouve bien le côté linéaire par rapport à la génération maximale et polynomial par rapport à la taille de la population.

5.5 Comparaison de la fonction objective par rapport à la taille de la population et la génération maximale



Nous avons également décidé de regarder le temps de la solution renvoyé par l'algorithme génétique en fonction de la taille de la population et de la génération maximale.

Sur des tailles de populations petites, la solution renvoyée s'améliore très rapidement et fortement. Par contre, ces évolutions sont moins distinguables pour des tailles de populations importantes.

6 ÉVALUATION DE LA QUALITÉ DES SOLUTIONS RENVOYÉES PAR L'APPROCHE GÉNÉTIQUE

6.1 Méthode d'évaluation de la qualité

Cette algorithme prend en entrée le chemin d'accès vers un répertoire contenant des fichiers *.fjs*. Pour chacun des fichiers présents dans ce répertoire, l'algorithme génétique sera exécuté cinq fois avec les mêmes paramètres. Il s'agira ensuite de calculer le temps moyen des solutions (leur fonction objective), mais aussi le temps d'exécution moyen pour chaque fichier.

Le code correspondant est disponible dans le fichier *evaluatesolutions.py*.

6.2 Résultats obtenus

Nous avons décidé de confronter notre algorithme aux résultats connus pour le *Brandimarte_Data*. Dans les tableaux suivants, LB correspond aux bornes inférieures connues et UB aux bornes supérieures connues de l'optimum. n correspond au nombre de job et m au nombre de machines.

Les temps moyens d'exécution sont donnés en secondes.

6.2.1 Résultats de notre algorithme pour une population de 50 individus et une génération maximale de 150

Instance	$n \times m$	LB	UB	Temps moyen	Temps d'exécution moyen
Mk01	10×6	**40**	**40**	44.4	6.91
Mk02	10×6	**26**	**26**	40.8	11.1
Mk03	15×8	**204**	**204**	235.6	22.6
Mk04	15×8	**60**	**60**	72.8	10.71
Mk05	15×4	**172**	**172**	189	12.02
Mk06	10×15	**57**	**57**	114.2	26.73
Mk07	20×5	**139**	**139**	198	15.45
Mk08	20×10	**523**	**523**	523	22.13
Mk09	20×10	**307**	**307**	391.4	34.11
Mk10	20×15	189	193	317	39.95

```
[EVALUATION] Resulting average time for files in data/Brandimarte_Data/Text
        Mk01.fjs - Average time = 44.4 for an average computation time of 6.90957259999999 seconds
        Mk02.fjs - Average time = 40.8 for an average computation time of 11.096101399999998 seconds
        Mk03.fjs - Average time = 235.6 for an average computation time of 22.599957599999982 seconds
        Mk04.fjs - Average time = 72.8 for an average computation time of 10.71180260000001 seconds
        Mk05.fjs - Average time = 189.0 for an average computation time of 12.024026000000003 seconds
        Mk06.fjs - Average time = 114.2 for an average computation time of 26.731190800000014 seconds
        Mk07.fjs - Average time = 198.0 for an average computation time of 15.45096100000003 seconds
        Mk08.fjs - Average time = 523.0 for an average computation time of 22.125283000000035 seconds
        Mk09.fjs - Average time = 391.4 for an average computation time of 34.114897 seconds
        Mk10.fjs - Average time = 317.0 for an average computation time of 39.95423800000003 seconds
```

6.2.2 Résultats de notre algorithme pour une population de 200 individus et une génération maximale de 500

Instance	$n \times m$	LB	UB	Temps moyen	Temps d'exécution moyen
Mk01	10×6	**40**	**40**	42.4	108.91
Mk02	10×6	**26**	**26**	34.4	202.02
Mk03	15×8	**204**	**204**	207.4	359.07
Mk04	15×8	**60**	**60**	69	166.65
Mk05	15×4	**172**	**172**	179.4	176.70
Mk06	10×15	**57**	**57**	96.2	439.49
Mk07	20×5	**139**	**139**	169.2	225.43
Mk08	20×10	**523**	**523**	523	353.90
Mk09	20×10	**307**	**307**	351.4	528.68
Mk10	20×15	189	193	282.8	568.81

```
[EVALUATION] Resulting average time for files in data/Brandimarte_Data/Text
        Mk01.fjs  - Average time = 42.4  for an average computation time of 108.91428319999977 seconds
        Mk02.fjs  - Average time = 34.4  for an average computation time of 202.0261765999996 seconds
        Mk03.fjs  - Average time = 207.4  for an average computation time of 359.0703437999997 seconds
        Mk04.fjs  - Average time = 69.0  for an average computation time of 166.648080200002 seconds
        Mk05.fjs  - Average time = 179.4  for an average computation time of 176.69973620000238 seconds
        Mk06.fjs  - Average time = 96.2  for an average computation time of 439.48598379999896 seconds
        Mk07.fjs  - Average time = 169.2  for an average computation time of 225.42877779998836 seconds
        Mk08.fjs  - Average time = 523.0  for an average computation time of 353.8992253999997 seconds
        Mk09.fjs  - Average time = 351.4  for an average computation time of 528.6716818000016 seconds
        Mk10.fjs  - Average time = 282.8  for an average computation time of 568.8121499999979 seconds
```

6.2.3 Résultats de notre algorithme pour une population de 500 individus et une génération maximale de 1000

Instance	$n \times m$	LB	UB	Temps moyen	Temps d'exécution moyen
Mk01	10×6	**40**	**40**	41.6	521.59
Mk02	10×6	**26**	**26**	31.2	928.05
Mk03	15×8	**204**	**204**	204	1880.24
Mk04	15×8	**60**	**60**	67	879.17
Mk05	15×4	**172**	**172**	178.6	959.05
Mk06	10×15	**57**	**57**	82.2	1845.56
Mk07	20×5	**139**	**139**	155.2	1181.74
Mk08	20×10	**523**	**523**	523	1978.39
Mk09	20×10	**307**	**307**	339	2680.10
Mk10	20×15	189	193	263.6	3228.08

```
[EVALUATION] Resulting average time for files in data/Brandimarte_Data/Text
        Mk01.fjs  - Average time = 41.6  for an average computation time of 521.592349999999 seconds
        Mk02.fjs  - Average time = 31.2  for an average computation time of 928.0481557999985 seconds
        Mk03.fjs  - Average time = 204.0  for an average computation time of 1880.2244417999987 seconds
        Mk04.fjs  - Average time = 67.0  for an average computation time of 879.1675653999991 seconds
        Mk05.fjs  - Average time = 178.6  for an average computation time of 959.0589938000048 seconds
        Mk06.fjs  - Average time = 82.2  for an average computation time of 1845.5627617999999 seconds
        Mk07.fjs  - Average time = 155.2  for an average computation time of 1181.7380269999994 seconds
        Mk08.fjs  - Average time = 523.0  for an average computation time of 1978.3938149999972 seconds
        Mk09.fjs  - Average time = 339.0  for an average computation time of 2680.1026947999985 seconds
        Mk10.fjs  - Average time = 263.6  for an average computation time of 3228.079152800006 seconds
```

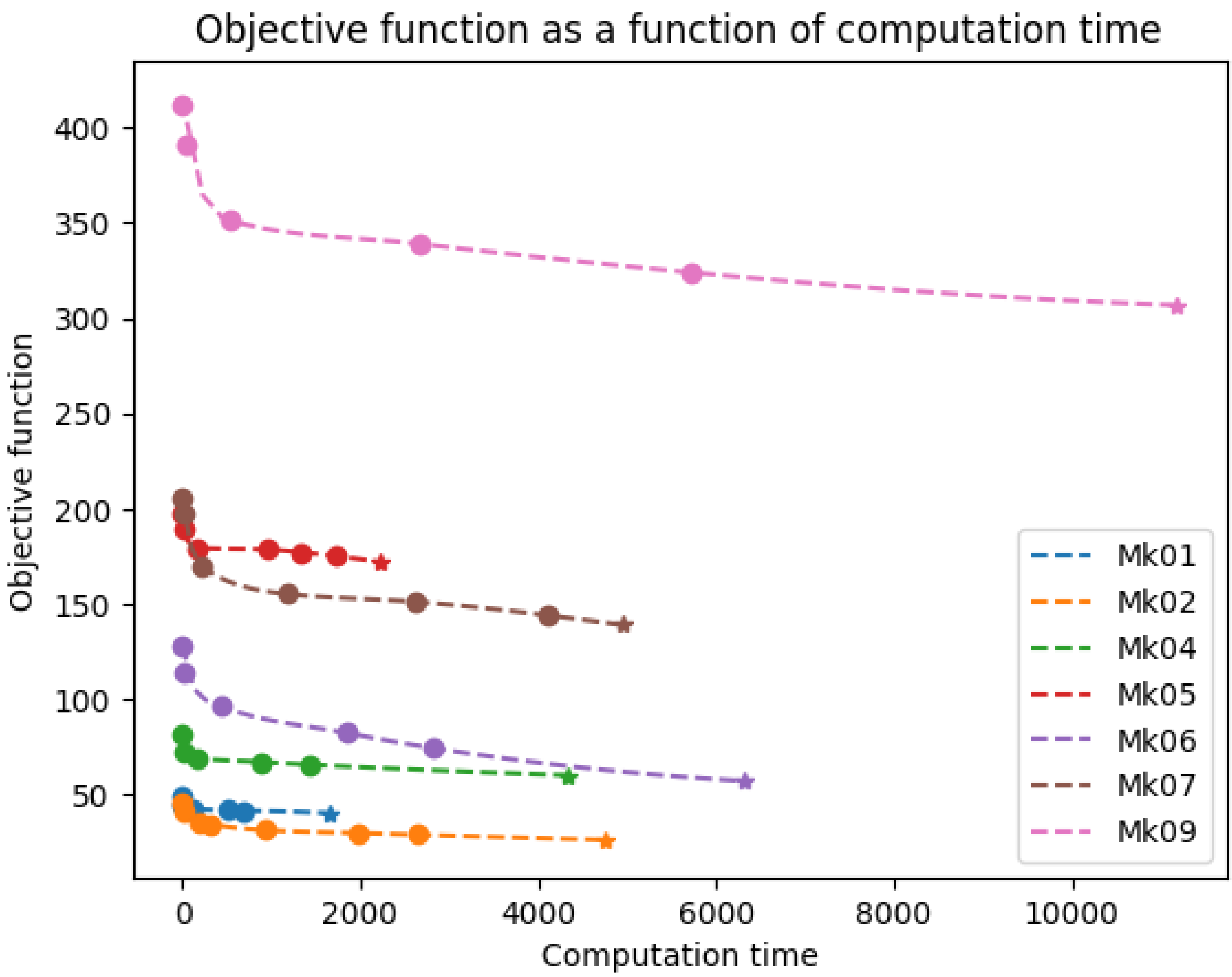
6.2.4 Interpolation des résultats et temps de calcul nécessaire pour arriver à l'optimum

Supposons que la meilleure solution trouvée soit fonction du temps de calcul. En effet, plus l'algorithme génétique tourne longtemps, plus il a de chance de converger vers l'optimum.

Nous ne nous intéresserons pas aux fichiers *Mk03* et *Mk08* car les solutions optimales ont été trouvées précédemment, ni à *Mk10* car l'optimum n'est pas connu.

```
Mk01 - Guessed computation time needed to get to optimum: 1664.926399781044 seconds
Mk02 - Guessed computation time needed to get to optimum: 4751.882261275811 seconds
Mk04 - Guessed computation time needed to get to optimum: 4332.952164446403 seconds
Mk05 - Guessed computation time needed to get to optimum: 2215.6149772116373 seconds
Mk06 - Guessed computation time needed to get to optimum: 6316.134541537323 seconds
Mk07 - Guessed computation time needed to get to optimum: 4939.822785061955 seconds
Mk09 - Guessed computation time needed to get to optimum: 11165.522414700212 seconds
```

En relançant un certain nombre d'instance de l'algorithme génétique, nous sommes parvenu à interpoler les fonctions objectives en fonction du temps de calcul pour chaque fichier à l'aide d'un algorithme appelé *Piecewise Cubic Hermite Interpolating Polynomial*. Nous ne rentrerons pas dans les détails techniques de cet algorithme. A partir de ces interpolations, nous cherchons pour quel temps de calcul nous obtenons la fonction objective optimale.



Les ronds correspondent aux points résultant des évaluations de l'algorithme génétique, les étoiles aux optima connus et les courbes en pointillé correspondent aux splines résultant des interpolations.

Ces résultats sont cependant à prendre avec des pincettes, absolument rien ne les garantie principalement du fait du peu de nombres de points que l'on a calculé (il nous a fallu plusieurs jours pour calculer cet ensemble de points étant donné que l'on a repris le principe d'évaluation des solutions décrit précédemment, la contrainte temporelle nous a donc limité sur la densité de cet ensemble).

7 CONCLUSION

Le résultat final répond aux objectifs que nous nous étions fixés au début du projet. En effet, notre modélisation est exploitable et répond aux attentes du sujet.

Nous finalisons ce projet avec le sentiment d'avoir enrichi nos connaissances en programmation de par l'acquisition de nouvelles notions ainsi que par l'application de points vus en cours. D'autre part il nous a fallu apprendre à s'organiser sur le long terme et à diviser les tâches tout en prenant compte du niveau de chacun. Ainsi, nous pensons que ce travail collectif fût aussi enrichissant pour nous quant à l'organisation d'un projet de groupe.