Projet complexité RO :
problème de synchronisation des données

Dhouha Meliane || 4DS8

Algorithme approché métaheuristique : Génétique
Contrainte : minimiser le temps de synchronisation d'une tâche

1. **Le pseudo-code de l'algorithme génétique (questions 1+2) :**
   1.1 fonction d'initialisation de la population :

```python
def initialiser_population(taille_pop, taille_chromosome):
    population = []
    for i in range(taille_pop):
        # Créer un chromosome aléatoire représentant un planning de synchronisation
        chromosome = generer_chromosome_aleatoire(taille_chromosome)
        population.append(chromosome)
    return population
```

Paramétres :

- `taille_pop` : Définit le nombre d'individus dans la population. Un nombre plus élevé augmente la diversité mais ralentit l'exécution.

- `taille_chromosome` : Représente la longueur de chaque solution (chromosome). Dans ce contexte, il correspond au nombre de périodes de synchronisation.

Fonction :

`initialiser_population()`

- **But** : Créer la population initiale de solutions aléatoires
- **Fonctionnement** : Génère des chromosomes aléatoires représentant différents plannings de synchronisation

1.2 fonction d'évaluation :

```python
def evaluer_fitness(chromosome):
    # Évaluer la qualité de la solution
    cout_total = 0
    # Calculer le coût de transfert des données
    cout_transfert = calculer_cout_transfert(chromosome)
    # Calculer le coût de stockage
    cout_stockage = calculer_cout_stockage(chromosome)
    # Calculer la pénalité pour les contraintes non respectées
    penalites = calculer_penalites(chromosome)

    cout_total = cout_transfert + cout_stockage + penalites
    return cout_total
```

Fonction : (en ajoutant : **temps_total = calculer_temps_total(chromosome)**)

`evaluer_fitness()`

- **But** : Évaluer la qualité de chaque solution
- **Fonctionnement** : Calcule trois composantes :
    - Coût de transfert des données
    - Coût de stockage
    - Pénalités pour les contraintes non respectées

1.3 fonction de sélection :

```python
def selection_parents(population, nb_parents):
    # Sélection par tournoi
    parents = []
    for _ in range(nb_parents):
        tournoi = random.sample(population, 3)  # Prendre 3 individus aléatoires
        meilleur = min(tournoi, key=lambda x: evaluer_fitness(x))
        parents.append(meilleur)
    return parents
```

Paramétres :

- `nb_parents` : Nombre de parents sélectionnés pour la reproduction.

Fonction :

`selection_parents()`

- **But** : Sélectionner les meilleurs individus pour la reproduction
- **Fonctionnement** :Utilise la méthode du tournoi : prend 3 individus aléatoires et sélectionne le meilleur

1.4 fonction de croisement :

```python
def croisement(parent1, parent2):
    # Croisement en un point
    point = random.randint(1, len(parent1)-1)
    enfant1 = parent1[:point] + parent2[point:]
    enfant2 = parent2[:point] + parent1[point:]
    return enfant1, enfant2
```

Fonction :

`selection_parents()`

- **But** : Combiner deux solutions parents pour créer deux nouvelles solutions
- **Fonctionnement** Utilise le croisement en un point : coupe les chromosomes parents en un point aléatoire et échange les parties

1.5 fonction de mutation :

```python
def mutation(chromosome, taux_mutation):
    # Mutation par inversion de bits
    for i in range(len(chromosome)):
        if random.random() < taux_mutation:
            chromosome[i] = 1 - chromosome[i]
    return chromosome
```

Paramétres :

- `nb_parents` : Nombre de parents sélectionnés pour la reproduction.

Fonction :

`mutation()`

- **But** : Introduire de petites modifications aléatoires
- **Fonctionnement** Inverse certains bits avec une probabilité définie par `taux_mutation`

1.6 algorithme génétique :

```python
def algorithme_genetique(params):
    population = initialiser_population(params.taille_pop, params.taille_chromosome)

    for generation in range(params.nb_generations):
        # Évaluation
        fitness_scores = [evaluer_fitness(chrom) for chrom in population]
```

```
    # Sélection
    parents = selection_parents(population, params.nb_parents)

    # Nouvelle population
    nouvelle_pop = []

    # Élitisme : garder les meilleurs
    elite = sorted(population, key=lambda x: evaluer_fitness(x))[:params.nb_elite]
    nouvelle_pop.extend(elite)
```

```
    # Croisement et mutation
    while len(nouvelle_pop) < params.taille_pop:
        parent1, parent2 = random.sample(parents, 2)
        enfant1, enfant2 = croisement(parent1, parent2)

        enfant1 = mutation(enfant1, params.taux_mutation)
        enfant2 = mutation(enfant2, params.taux_mutation)

        nouvelle_pop.extend([enfant1, enfant2])

    population = nouvelle_pop[:params.taille_pop]

# Retourner la meilleure solution
meilleure_solution = min(population, key=lambda x: evaluer_fitness(x))
return meilleure_solution
```

Fonction principale qui orchestre tout le processus

Paramétres :

- **`nb_generations`** : détermine le nombre d'itérations de l'algorithme. Plus il est élevé, plus l'algorithme a de chances de converger vers une bonne solution.
- **taux_mutation** : Contrôle la fréquence des mutations. Un taux trop élevé peut déstabiliser la convergence, trop faible peut mener à une convergence prématurée.
- **nb_parents** : Nombre de parents sélectionnés pour la reproduction. Influence la diversité génétique.
- **nb_elite :** Nombre de meilleurs individus conservés à chaque génération (élitisme).

Étapes :

1. Initialise la population
2. Pour chaque génération :
    - Évalue toutes les solutions
    - Sélectionne les parents
    - Conserve les meilleures solutions (élitisme)
    - Crée de nouvelles solutions par croisement et mutation
3. Retourne la meilleure solution finale

2. Adaptation de l'algorithme au problème de synchronisation des données (question 3):

2.1 Représentation:

- Chromosome = séquence de tâches de synchronisation
- Chaque gène représente une tâche avec son timing
- Format : [(tâche1, temps1), (tâche2, temps2), ...]

2.2 Initialisation:

- Générer des séquences aléatoires respectant les contraintes de précédence
- Assigner des temps de début valides pour chaque tâche

2.3 Sélection :

Les critères pour lesquelles on sélectionne les meilleurs parents par tournoi :

- Vérifier les conflits des ressources
- Calculer le temps total d'exécution
- Evaluer les temps d'inactivité

2.4 Opérateur génétique :

- Créer de nouvelles solutions par croisement et mutation

Croisement :

- Combiner deux solutions parents pour créer deux nouvelles solutions
- Utilise le croisement en un point : coupe les chromosomes parents en un point aléatoire et échange les parties

Mutation:

- Inverse certains bits (tâche , temps) avec une probabilité définie par `taux_mutation`

2.5 Condition d'arrêt :

- Atteindre la limite minimale de temps de synchronisation d'une tâche

3. Analyse de la complexité de l'algorithme (question 4) :

1. **Initialisation :**
- O(N × P) où N = nombre de tâches, P = taille de la population
2. **Évaluation d'une génération :**
- O(P × N²) car pour chaque individu (P), on doit vérifier les conflits entre tâches (N²)
3. **Sélection :**
- O(P × log P) avec une sélection par tournoi
4. **Croisement :**
- O(P × N) pour croiser les parents et vérifier la validité
5. **Mutation** :
- O(P × N) dans le pire cas

- **Complexité totale par génération** :
  $O(P \times N^2)$ dominé par l'évaluation des solutions

- **Complexité totale de l'algorithme :**
  $O(G \times P \times N^2)$ où G est le nombre de générations

Justification :

- La complexité est élevée mais acceptable car c'est un algorithme approché
- Le temps d'exécution peut être contrôlé par les paramètres **(P et G)**
- L'algorithme trouve généralement de bonnes solutions avant d'atteindre le nombre maximum de générations