tags: `Intelligence artificielle`

Semestre automne 2021

TP IA: Labyrinthe

Author : Simon Meier
Date : 27.11.2021
Classe : INF 3 dlm-a

Table des matières :

1. Introduction

*Les algorithmes génétiques appartiennent à la famille des **algorithmes évolutionnistes**. Leur but est d'obtenir une solution approchée à un problème d'optimisation, lorsqu'il n'existe pas de méthode exacte (ou que la solution est inconnue) pour le résoudre en un temps raisonnable.* (source: Wikipédia - Algorithme génétique (https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique))

Durant le cours d'IA, un *travail pratique* (TP) à mener à bien consiste a proposer une résolution d'un labyrinthe de taille variable à l'aide d'un algorithme génétique.

Faisant partie de la catégorie des algorithmes d'optimisation, il est conçu pour trouver une potentielle solution et ensuite de l'optimiser, puis retourner peut-être une solution satisfaisante.

L'intérêt principal ici est d'apprendre le fonctionnement des algorithmes génétiques et leurs applications.

2. Encodage d'un chromosome

Un chromosome correspond en une succession de déplacement dans une des 4 directions possibles dans un labyrinthe; `up`, `down`, `left`, `right`.

Leurs longueur est défini par l'aire du labyrinthe, soit par la multiplication de la hauteur et largeur du labyrinthe: (`grid.shape[0] * grid.shape[1]`).

2.1 Codification des chromosomes

```
class Direction(Enum):
    """
    Possible directions in a grid.
    """
    UP = 0
    RIGHT = 1
    DOWN = 2
    LEFT = 3
```

2.2 Lecture

```
    def _decode(self, individual):
        """
        Transform the individual to a list(map()) of directions
        """
        return list(map(lambda val: Direction(val), individual)
```

3. Fonction de fitness

La fonction de `fitness` est la somme entre:

- La distance de manhattan entre la dernière coordonnée du chemin et la fin du labyrinthe.
- la longueur du chemin de l'individu comme malus s'il n'est pas arrivé à la fin.

Le score de `fitness` doit être le plus bas possible.

```
    def manhattan(self, depart, destination) -> int:
        """
        manhattan distance between two points
        """
        return abs(depart[0] - destination[0]) + abs(depart[1]

    def fitness(self, individual) -> tuple:
        """
        fitness function, will calculate a fitness with
        the manhattan distance of the point,
        the length of the path
        and a malus if it doesn't reach the end.
        """
        path = self.compute_chromosome(individual)
        return self.manhattan(self.end_cell, path[-1]) + len(in
```

`compute_coordinates()` :

Cette fonction permet le *"déplacement"* dans le labyrinthe en retournant une nouvelle coordonnée en fonction de la direction donnée par les individus.

```
def compute_coordinates(self, x, y, direction) -> tuple:
    """
    get the new x and y from a direction
    """
    if direction == Direction.UP:
        x -= 1
    elif direction == Direction.RIGHT:
        y += 1
    elif direction == Direction.DOWN:
        x += 1
    elif direction == Direction.LEFT:
        y -= 1
    return x, y
```

`compute_chromosome()` :

Premièrement, `directions` permet de récupérer les mouvements qu'un chromosome contient.
`x, y` est un tuple contenant les coordonnées de la fin du labyrinthe.
Ensuite, pour chaque composante de directions, une nouvelle coordonnée `(x, y)` est calculée avec la fonction `compute_coordinates()`.
Une comparaison est ensuite effectuée; `if 0 <= new_x < self.width , and 0 <= new_y < self.height , and self.grid[new_coordinates] == 0` , pour vérifier que ce "*déplacement*" ne fait pas sortir de la grille, ni ne passe un mur.
Elles sont ensuite attribuées aux coordonnées déclarées plus haut. Si le résultat correspond à la case de résolution du labyrinthe, alors elle `return result` .

```
def compute_chromosome(self, individual):
    """
    Transform an individual
    to a list of coordinates
    which will be the path.
    """
    directions = self._decode(individual)
    x, y = self.start_cell[0], self.start_cell[1]
    result = [(x, y)]

    for direction in directions:
        new_x, new_y = self.compute_coordinates(x, y, direc
        new_coordinates = (new_x, new_y)

        # If it's going out of the grid or against a wall,
        # make it not moving.
        if 0 <= new_x < self.width \
            and 0 <= new_y < self.height \
                and self.grid[new_coordinates] == 0:
            result.append(new_coordinates)
            x = new_x
            y = new_y

        # if at the end cut the chromosom
        if (x, y) == self.end_cell:
            return result

    return result
```

3.1 Traitement du chromosome après son calcul

La fonction `upgrade_chromosome(self, individual)` permet de tamiser les composantes des individus.

- Pour chaque coordonnée reçue dans `directions`, si elle est possible, est ajoutée au chemin.
- Dans le cas contraire;
  - Utilisation d'un nombre aléatoire entre [0, 1] pour donner une direction aléatoire à l'individu.
  - Sinon l'individu prend la direction qui le fera aller le plus proche de la sortie du labyrinthe avec la fonction `try_better_path()`
  - Finalement, vérification que la nouvelle coordonnée soit valide.

```
def upgrade_chromosome(self, individual):
    """
    When a chromosome goes through a place it already went
    or went against a wall, this function will purify the
    chromosome from those impurities.
    """
    directions = self._decode(individual)
    x, y = self.start_cell[0], self.start_cell[1]
    path = [(x, y)]

    for i, direction in enumerate(directions):
        new_x, new_y = self.compute_coordinates(x, y, direction
        new_coordinates = (new_x, new_y)
        if 0 <= new_x < self.width \
            and 0 <= new_y < self.height \
                and self.grid[new_coordinates] == 0 \
                    and not new_coordinates in path:
            path.append(new_coordinates)
            x, y = new_x, new_y

        else:
            # Find a new random direction or the closest to the
            if random() < self.RPP:
                individual[i] = randint(0, len(Direction) - 1)
            else:
                individual[i] = self.try_better_path(path[-1],

            new_x, new_y = self.compute_coordinates(x, y, Direc
            new_coordinates = (new_x, new_y)

            # Verify if the new direction is possible
            if 0 <= new_x < self.width \
                and 0 <= new_y < self.height \
                    and self.grid[new_coordinates] == 0 \
                        and not new_coordinates in path:
                path.append(new_coordinates)
                x, y = new_x, new_y

        # If chromosome reaches the end of the labyrinth
        if (x, y) == self.end_cell:
            individual = individual[0:i]
            return
```

try_better_path()

Tentative de retour de la meilleure direction possible en fonction de la distance à l'arrivée.

```
def try_better_path(self, position, target) -> Direction:
    """
    Try to get the best direction for a point.
    """
    if abs(position[0] - target[0]) > abs(position[1] - target[
        if position[0] < target[0]:
            return Direction.DOWN
        else:
            return Direction.UP
    else:
        if position[1] < target[1]:
            return Direction.RIGHT
        else:
            return Direction.LEFT
```

4. Sélection

La diversité de la population permet de mesurer le “niveau de convergence”.

Si cette dernière est faible, la probabilité de converger vers un minima local sera plus grande, ce qui n’est pas voulu; un minima local induit que la population est prisonnière d’elle-même. Dans le cas présent, elle est difficile à éviter.

5. Crossover

Un cross-over sert principalement à induire de la diversité. Il paraît contre-productif d’utiliser ce genre de concept sur la problématique puisque le moindre changement dans un des gènes peut totalement détruire la qualité d’un chemin.

En revanche, comme nous sommes face à de nombreux *minima* locaux, il est déterminant de garder un *pool* de gène varié qui peut mener à la bonne solution.

Voir la section **[7] - Toolbox et `creator`** pour plus de détails sur les choix des fonctions de cross-over.

6. Mutation

Tout comme le cross-over, permet d'ajouter de la diversité aux individus.
La mutation permet à un gène, au sein d'un chromosome, d'être substitué à un autre de façon aléatoire.
Un taux de mutation est défini pour changer les population.
Il est nécessaire de choisir pour ce taux une valeur relativement faible, de manière à ne pas tomber dans une recherche aléatoire et à conserver le principe de *sélection* et d'*évolution*.
Elle sert à éviter une convergence prématurée de l'algorithme. Par exemple, lors d'une recherche d'extremum, la mutation sert à éviter la convergence vers un extremum local.
Voir la section **[7] - Toolbox et `creator`** pour plus de détails sur les choix des fonctions de mutations.

7. Toolbox et `creator`

Le *package* `DEAP` contient des outils pour travailler sur des population et les faire évoluer. Il faut initialiser un `creator` et une `toolbox`. La `toolbox` permet d'enregistrer des méthodes avec des paramètre par défaut sans changer la *signature de la méthode*, puis être rappelée plus tard, ce qui est très pratique.

`creator` :

- `FitnessMin` définit la manière dont le `fitness` des individus est traité. `weights` donne la priorité des éléments du tuple fitness associé aux individus. Dans ce cas `weights=(-1.0,)` signifie qu'il faut minimiser toutes les valeurs.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
```

- `Individual` définit le type de structure de données dont les individus sont issus. Il faut lui indiquer la manière dont le `fitness` est calculé. Dans le cas présent, le choix s'est porté sur une `list` et le `fitness` est calculé avec la méthode `FitnessMin` créée au préalable.

```
creator.create("Individual", list, fitness=creator.FitnessMin)
```

`toolbox` :

- `fitness` et `evaluate` correspondent aux méthodes `fitness` et `evaluate_population` codées. Elles sont ajoutées dans la `toolbox`.

```
self.toolbox.register("fitness", self.fitness)
self.toolbox.register("evaluate", self.evaluate_population)
```

- `init_gene` défini la manière dont est construit un gène, dans ce cas un nombre aléatoire entre 0 et 3 indiquant une direction.

```
self.toolbox.register("init_gene", randint, 0, 3)
```

- `mate` indique la manière dont les individus se 'reproduisent'. Parmi toutes les méthodes de cross-over, `cxOnePoint` et `cxTwoPoint` ont été testé. Il est difficile de dire l'impact exact de ces méthodes sur les solutions, la méthode `cxTwoPoint` a été choisie en fonction de la documentation: source (https://deap.readthedocs.io/en/master/api/tools.html#deap.tools.cxTwoPoint), car les deux individus gardent leur longueur originelle, ce qui n'est pas le cas avec `cxOnePoint`.

```
self.toolbox.register("mate", tools.cxTwoPoint)
```

- `mutate` indique la manière dont les individus mutent. Deux méthodes ont été testés:
  - `mutUniformInt` avec `low=0`, `up=3` et `indpb=MUTATION_PROBABILITY`, mute un individu en remplaçant ses attributs avec une valeur uniforme entre `low` et `up`.
  - `mutShuffleIndexes`, mélange les attributs d'un individu et retourne l'individu muté.
  - la méthode `mutShuffleIndexes` est celle retenue car les composantes des chemins des individus changent au fur et à mesure, ce ne sont pas seulement des directions, et donc il est moins intéressant de muter sur un interval. Le paramètre `indpb` correspond à la probabilité indépendante pour chaque instruction d'être changée de place.

```
self.toolbox.register("mutate", tools.mutShuffleIndexes, indpb=
```

- `select` défini la manière dont les individus sont sélectionnés. Dans le cas présent, la méthode `selTournament` est celle retenue puisqu'elle permet de minimiser la fitness en choisissant des individus aléatoirement et en confrontant leur fitness.

```
self.toolbox.register("select", tools.selTournament)
```

- `init_individual` indique la manière de créer un individu et la longueur de chaque chromosome, dans ce cas la méthode `init_gene` est utilisé un nombre de fois égale à la longueur d'un chromosome, soit ici l'aire du labyrinthe.

```
self.toolbox.register("init_individual", tools.initRepeat, crea
```

- `init_population` défini la manière dont la population est crée, dans ce cas une `list` est remplie avec des éléments défini par `init_individual` .

```
self.toolbox.register("init_population", tools.initRepeat, list
```

8 Main Loop

Dans la `main loop`, exceptés les initialisations des différents outils, les opérations principales sont les suivantes:

- 1. Select:
  - Séléctionner les meilleurs individus de la population et les mettre dans `offsprings`.
- 2. Populate:
  - Initialiser et rajouter aux *offsprings* des nouveaux individus.
- 3. Appliquer les mutations
  - Pour chaque individu, application d'un cross-over et d'une mutation avec une certaine probabilité.
- 4. Evaluer:
  - Evaluation de la population.
- 5. Sortir la meilleur solution:
  - attribution à `best` de la meilleure solution actuelle, puis comparaison de la solution avant et de la nouvelle `best` pour savoir laquelle est la meilleure.
  - si la solution est différente de la précédente, alors mis à jours de la solution la plus ancienne, modification de la constante de mutation et incrémentation de `steps` ; tentatvie de sortir des minimums locaux.

```
def run(self):
    """
    Genetic algorithm to find the solution.
    """
    start_time = inter_time = time.time()

    self.toolbox_init()

    steps = 0

    population = self.toolbox.init_population(n=POPULATION_SIZE

    if self.width >= 30:
        self.CXP = 0.04
        self.MUP = 0.40
        self.RPP = 0.5

    print(f"CXP : {self.CXP}, MUP: {self.MUP}, RPP: {self.RPP}"

    self.evaluate_population(population)

    solution = None

    # ------------------------------------------------------------
    # Main Loop
    # ------------------------------------------------------------

    while inter_time - start_time < self.time:

        old_solution = solution

        # ------------------------------------------------------
        # 1. Selection
        # ------------------------------------------------------
        offsprings = self.toolbox.select(population, len(popula
        offsprings = list(map(self.toolbox.clone, offsprings))

        # ------------------------------------------------------
        # 2. Populate
        # ------------------------------------------------------
        for i in range(0, len(population) - len(offsprings)):
            offsprings.append(self.toolbox.init_individual())

        # Apply crossover and mutation on the offspring
        for offspring1, offspring2 in zip(offsprings[::2], offs
            if random() < self.CXP:
                self.toolbox.mate(offspring1, offspring2)
```

```
                self.toolbox.mate(offspring1, offspring2)

        # ----------------------------------------------------
        # 3. Applying mutations and upgrade on children
        # ----------------------------------------------------
        for offspring in offsprings:
            # add a new component
            if random() < self.MUP:
                self.toolbox.mutate(offspring)
            self.upgrade_chromosome(offspring)

        # ----------------------------------------------------
        # 4. Evaluate population
        # ----------------------------------------------------

        self.evaluate_population(offsprings)

        population = offsprings

        # ----------------------------------------------------
        # 5. Search for the best solution in all the offsprings
        # ----------------------------------------------------

        best = self.find_best(offsprings)

        if not solution or best.fitness.values[0] < solution.fi
            solution = best

        if old_solution != solution:
            old_solution = solution
            self.MUP = 0.02
            # print(f"The best solution was find after {time.ti
        else:
            steps += 1

        # Add a gene to all individual and increase the probabi
        # if there is 5 or more genertation without new best so
        if steps >= 3 :
            if self.MUP < 0.2:
                self.MUP *= 1.5
            [ind.insert(randint(0, len(ind)-1), randint(0,3)) f

        inter_time = time.time()

    return self.compute_chromosome(solution)
```