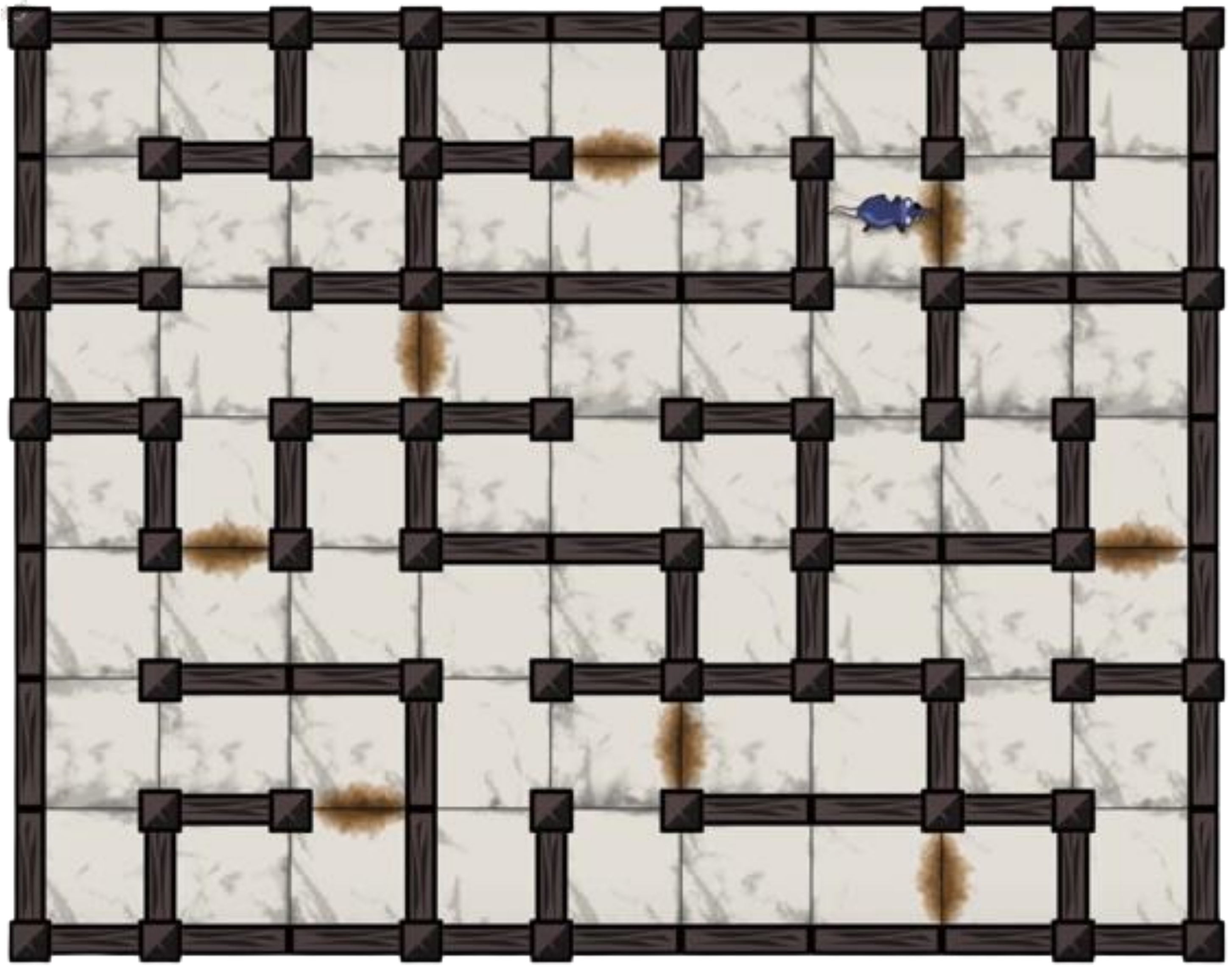
PyRat : almost a winning strategy

OUNZAR Aymane
RHARRASSI Mohamed
EL KARMI Sohaib

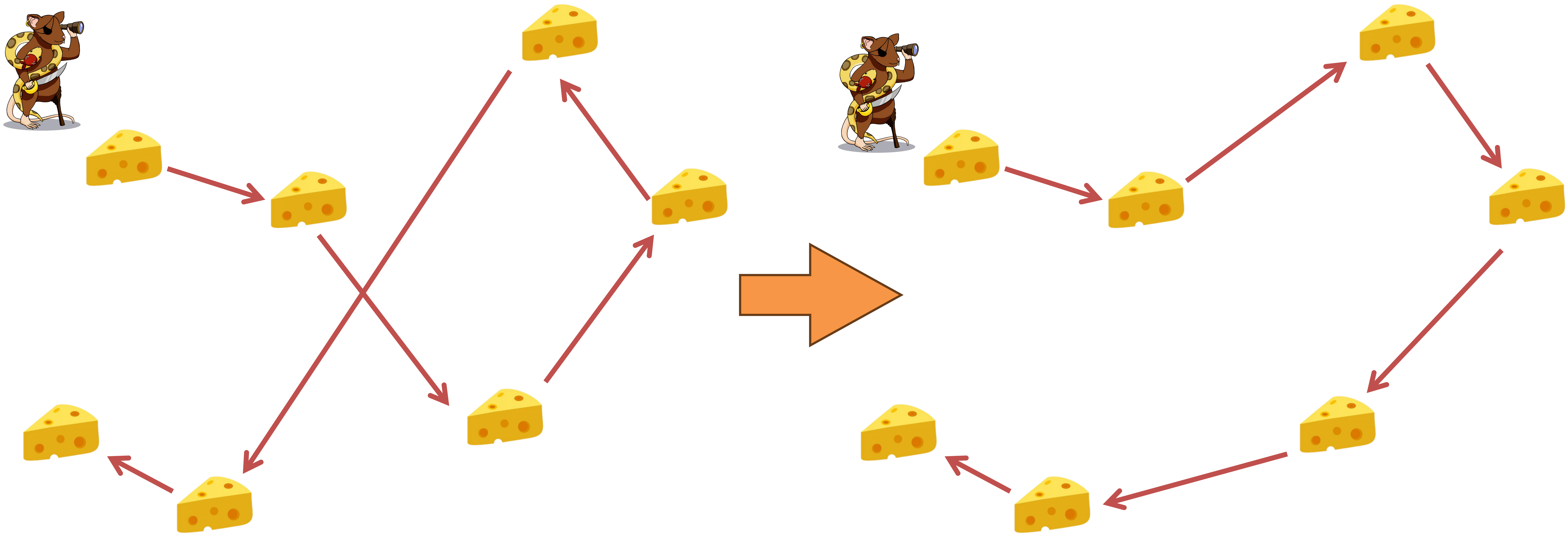


Plan

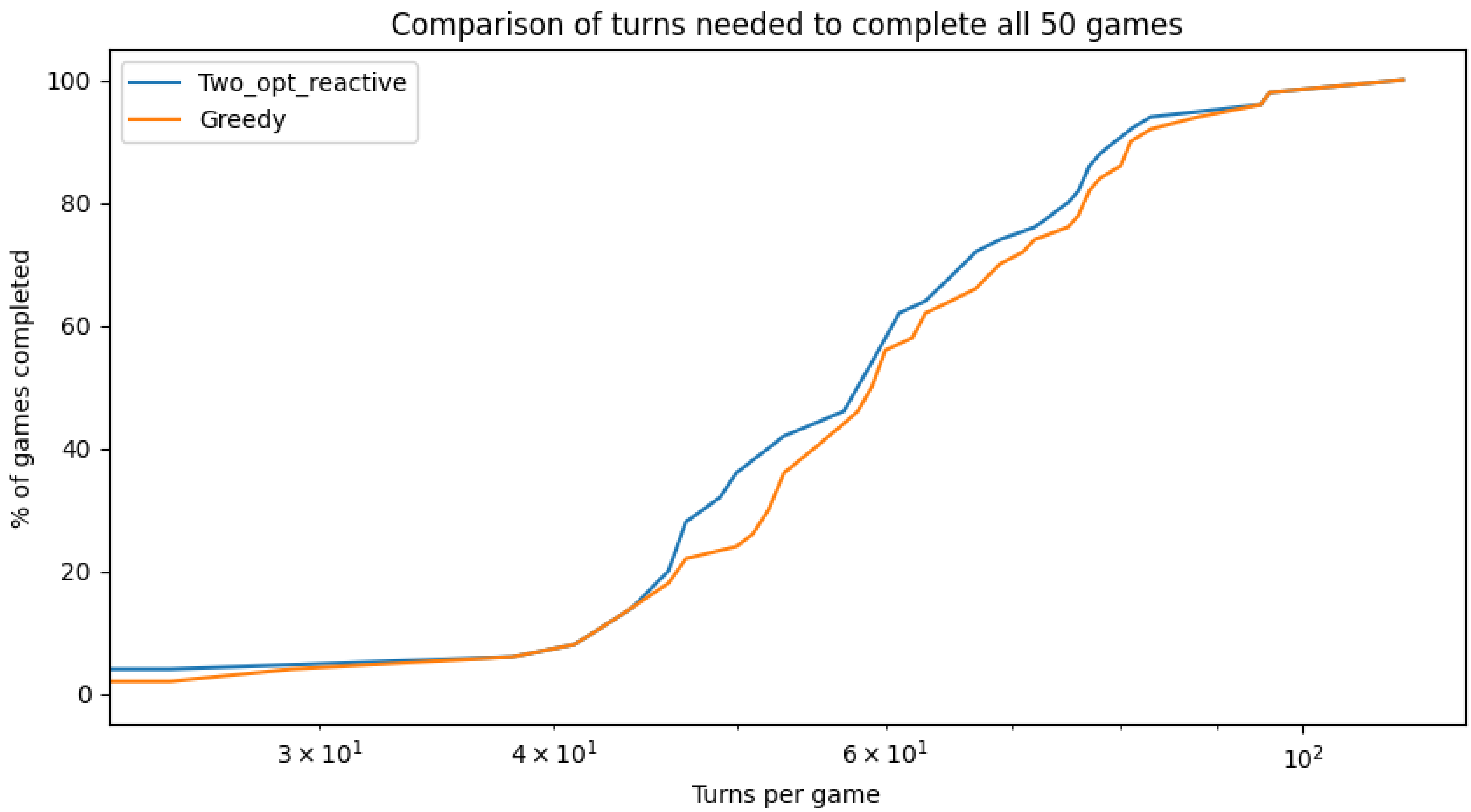
- Heuristic improvement of Greedy : 2-Opt .
- Greedy_density
- Conclusion

2-Opt

Principe



Comparaison :Tw-Opt and Greedy



Principe: Two_opt_reactive

- ***Preprocessing:***

1. Calcul du **meta_graph**

2.Calcul de **partial_path_0** donné par l'algorithme de **Greedy**

3. Calcul de **partial_path_1**, amélioration de **partial_path_0** par l'algorithme de **2-opt.**

4.   La variable **destination** prend la valeur **partial_path_1[0]**

5. La variable **route** prend la valeur **find_route(source,destination)**

- ***Turn:***

1. Mise à jour de la variable **pieces_of cheese.**

2. Mise à jour de l'attribut **destination (**ou pas) **,** prenant en compte si notre destination est (ou pas) dans la liste **pieces_of cheese.**

3. Si elle doit etre mise à jour , la destination est la première piece of cheese dans la liste **partial_path** et qui reste toujours dans la liste **pieces_of cheese.**

4. Détermination de **l'action suivante .**

Match: Two-Opt_reactive vs GreedyEachTurn

100 games

2_opt_reactive	GreedyEachTurn
41,5	58,5

Avantages/ Inconvénients

Avantages	***Inconvénients***
Nombre de turns réduit s'il jout seul	Il perd souvent (non adapté au Tournoi)
Tient compte des mouvements de l'adversaire	

Greedy_density

Principe: Density

Preprocessing:

1. Calcul du **meta_graph**

2. On définit la fonction **`surrounding_cells(self,cheese,maze)`** qui retourne une liste des 25 cellules qui entourent le fromage **`cheese.`**

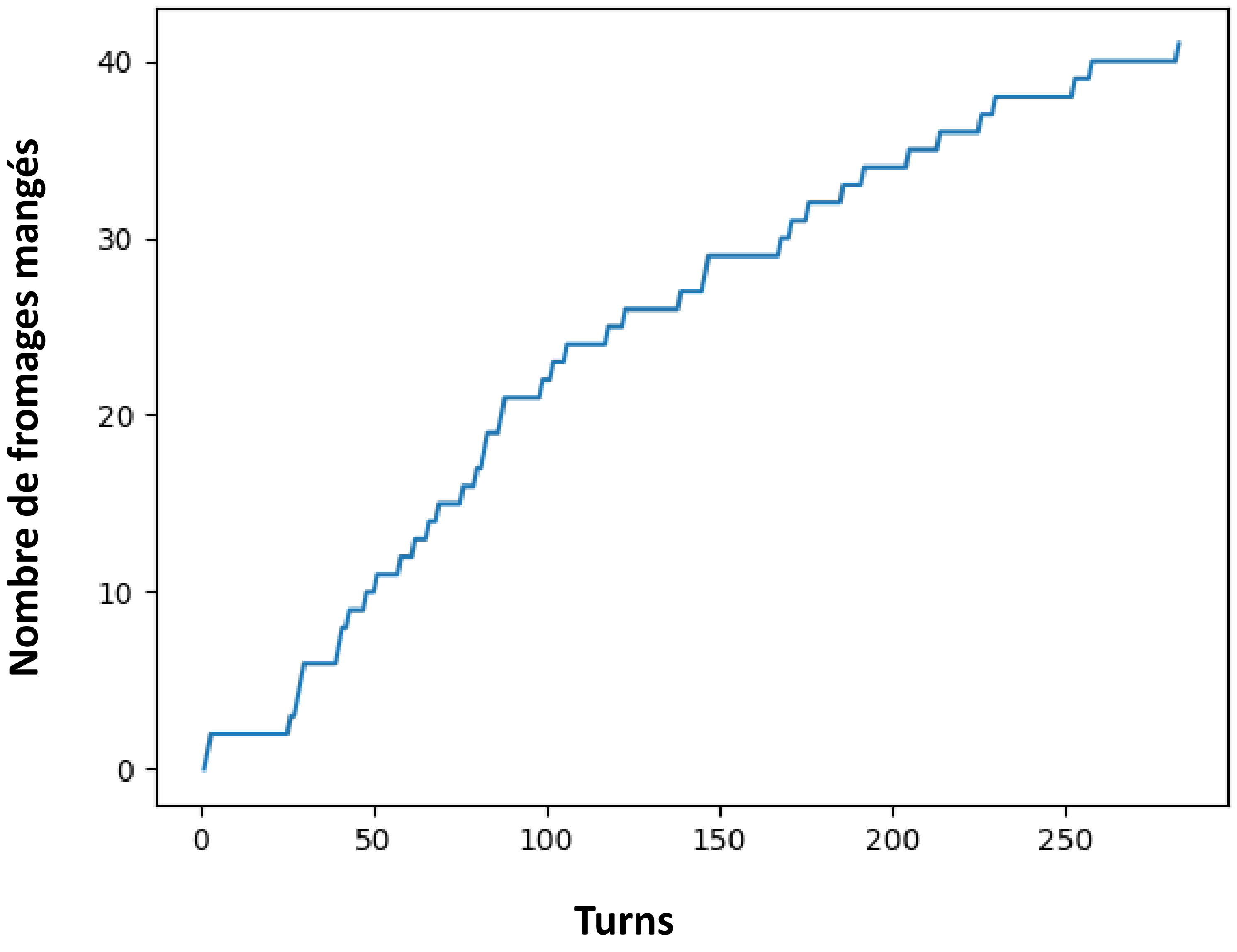
3. On définit la fonction **`density(self,cheese,pieces_of_cheese,maze)`** qui nous donne la densité d'un fromage.

4. Lorsqu'on perform l'algorithme greedy pour trouver le fromage suivant , au lieu d'utiliser la distance comme critère, on utilise $\frac{distance}{densité}$.

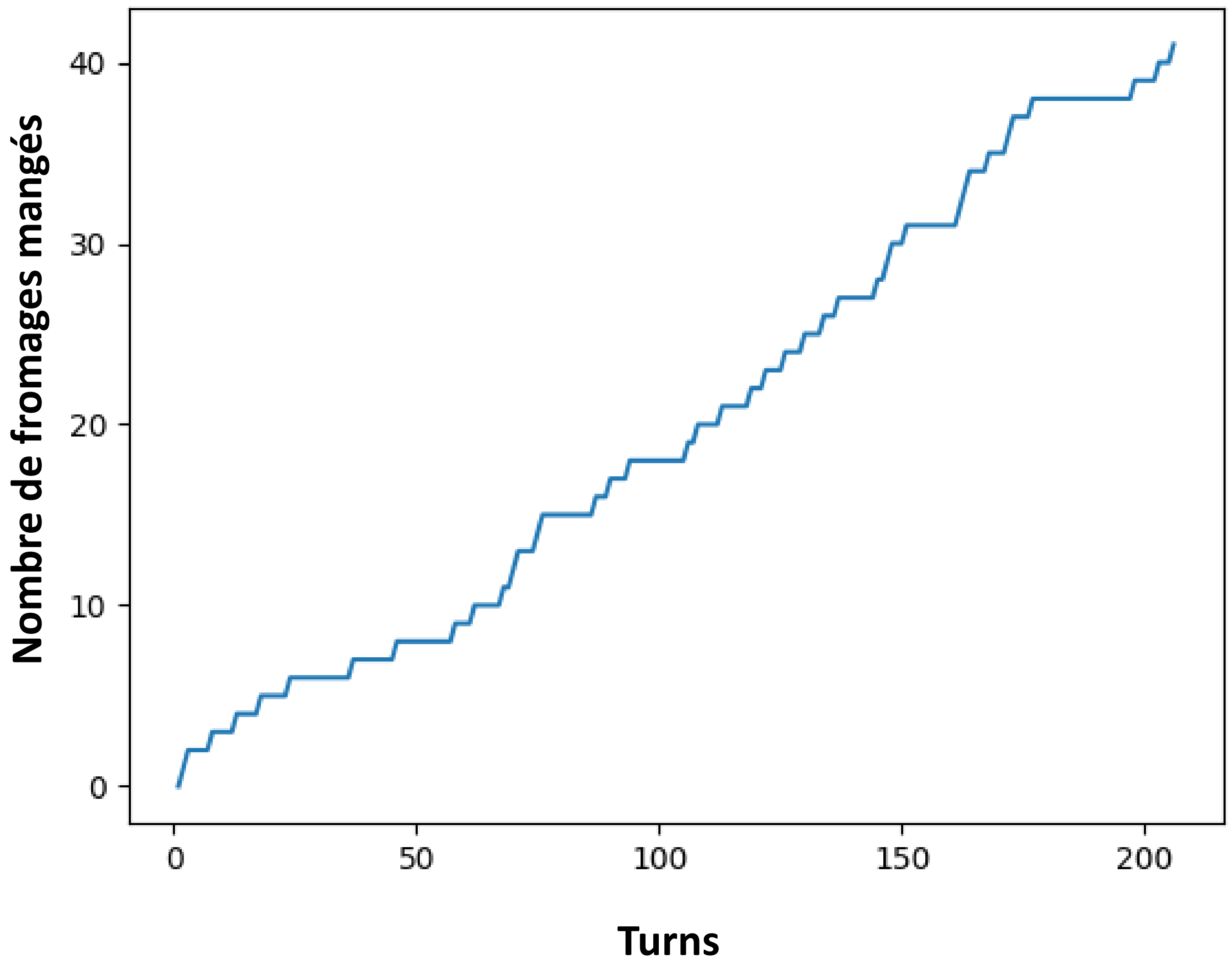
Turn

A chaque tour on recalcule la density .

GreedyEachTurn_density



GreedyEachTurn



Match: GreedyEachTurn_density vs GreedyEachTurn

100 games

GreedyEachTurn_density

62

GreedyEachTurn

38

Match: GreedyEachTurn_density vs Two_opt_reactive

100 games

GreedyEachTurn_density

64,5

2_opt_reactive

35,5

Conclusion

Notre approche est bien une version un peu plus poussée des algorithmes “greedy”, mais il peut être modifié en tenant compte ses inconvénients pour être plus performant.

Merci de votre attention ! 🧀

Annexe

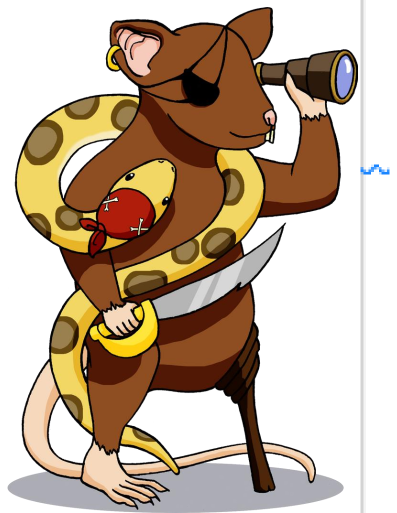
2_opt

```python
def two_opt(self:Self , partial_path:list,meta_graph:dict,initial_time:float,time_limit:float):
    """
    Applies the 2-opt heuristic to optimize the given partial path.

    In:
        * partial_path (list): A list of vertices representing the ordered sequence of nodes to visit.
        * meta_graph (dict): A meta-graph where each vertex is connected to other vertices with distanc
            - The meta-graph's edges should include the shortest route between connected nodes.

    Out:
        * list: A list of vertices representing the optimized path after applying the 2-opt heuristic.
    """
    time_1=time.time()
    time_2=time.time()
    delta_time=time_2-time_1+initial_time
    def path_length(path):
        return sum([meta_graph[path[i]][path[i+1]][0] for i in range(len(path)-1)])
    # Initialize the best path as the given partial path.
    best_path = partial_path.copy()
    best_length=path_length(best_path)
    # Initialize the flag to indicate whether the path has been improved.
    improved = True

    # Iterate until no further improvements can be made.
    while improved and delta_time<time_limit:
        # Set the flag to False to check for improvements in this iteration.
        improved = False
```



2_opt

```
            # Iterate over all pairs of edges in the path.
            for i in range(1, len(partial_path) - 1):
                for j in range(i + 1, len(partial_path)-1):
                    new_path=best_path.copy()
                    new_path[i] ,new_path[j]= new_path[j],new_path[i]
                    new_path_length=path_length(new_path)
                    if new_path_length<best_length:
                        # Reverse the segment between i and j in the path.
                        best_path[i] ,best_path[j]= best_path[j],best_path[i]
                        best_length=new_path_length
                        # Update the flag to indicate an improvement.
                        improved = True
            time_2=time.time()
            delta_time=time_2-time_1+initial_time
        return best_path
```

Density

```python
def surrounding_cells(self,cheese,maze):
    rows=maze.height
    cols=maze.width
    # Convert cell index to 2D coordinates
    row = cheese // cols
    col = cheese % cols

    # Determine the bounds of the 5x5 area
    start_row = max(0, row - 2)
    end_row = min(rows - 1, row + 2)
    start_col = max(0, col - 2)
    end_col = min(cols - 1, col + 2)

    # Collect surrounding cell indices
    result = []
    for r in range(start_row, end_row + 1):
        for c in range(start_col, end_col + 1):
            result.append(r * cols + c)

    return result
```

Density

```
def density(self,cheese,pieces_of_cheese,maze):
    area=self.surrounding_cells(cheese,maze)
    density=0
    for cell in area:
        if cell in pieces_of_cheese:
            density+=1
    return density/25
```

Density

```python
def nearest(self,position,pieces_of_cheese,maze):
    min_weight = float('inf')
    destination = position
    for target in pieces_of_cheese:
        weight=self.graph[position][target][0]/self.density(target,pieces_of_cheese,maze)
        # Update the nearest piece of cheese if the current one is closer.
        if weight < min_weight:
            min_weight = weight
            destination = target
    return destination
```