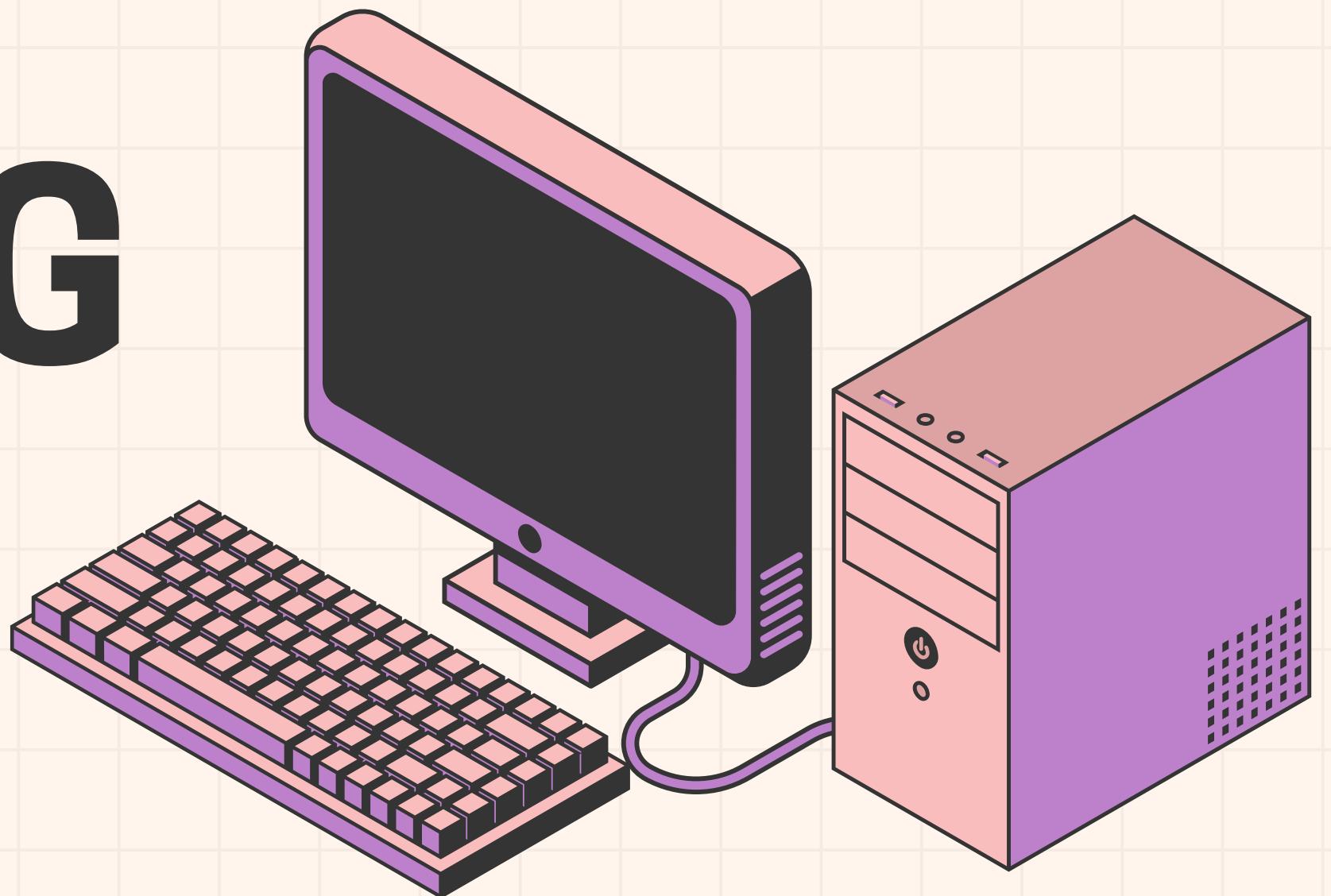


SCHEDULING MIN MIN

Groupe 1: Ali Ait Mahiddine, Ghiles
Kemiche, Noah Parisse, Julien Dumortier



Orchestration de Processus

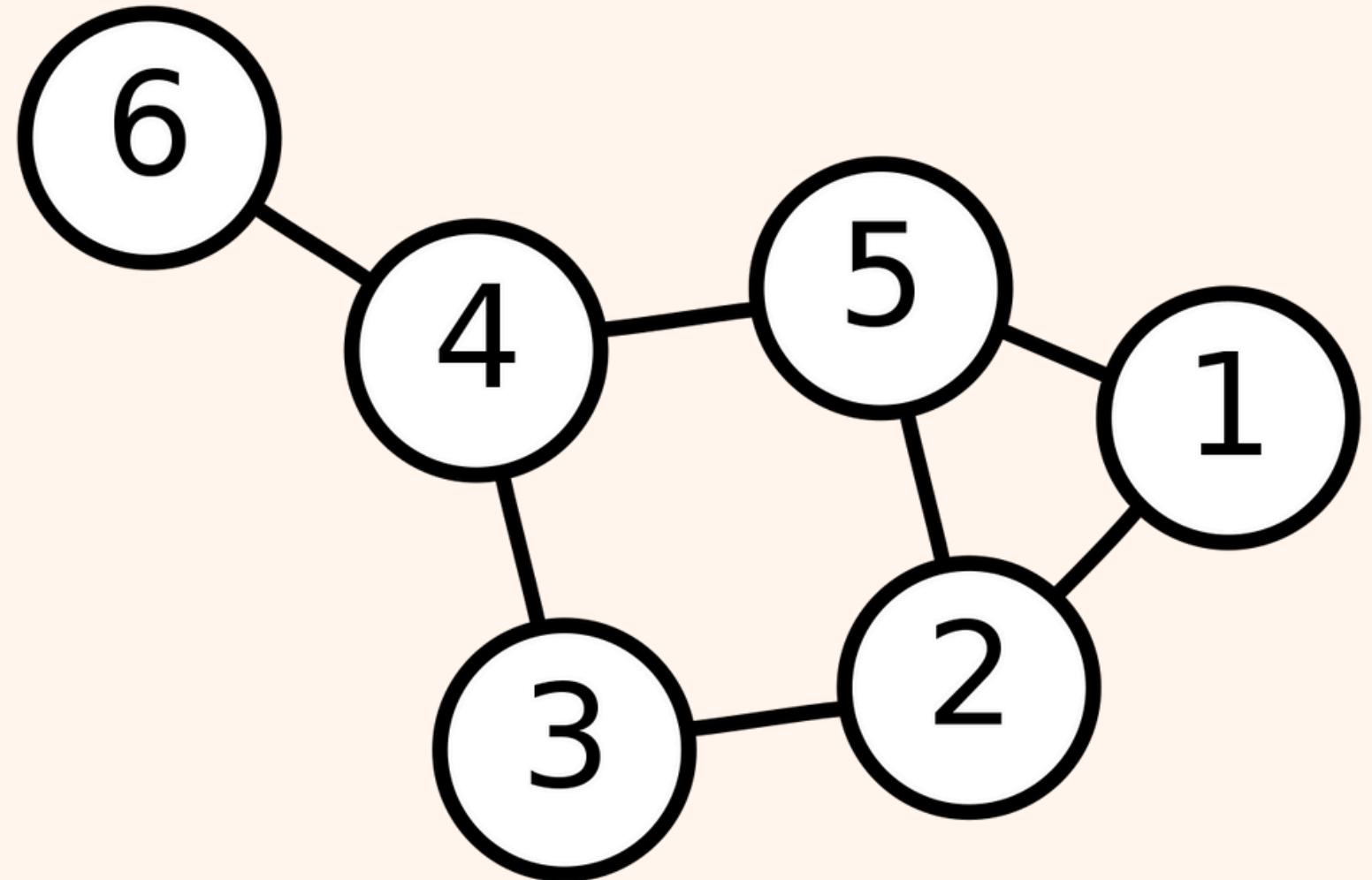
- **Coordination centralisée** : Un contrôleur gère l'exécution et l'enchaînement des tâches.
- **Automatisation des systèmes** : Permet de structurer et d'automatiser des processus complexes.
- **Gestion des erreurs** : Intègre des mécanismes de reprise et de compensation.
- **Applications** : Utilisée dans les microservices, les conteneurs (ex. Kubernetes) et les systèmes de workflow (ex. Airflow).

Hypothèses :

- Graphe des tâches disponible avec les dépendances entre processus
- Machines disponibles, avec le temps d'exécution de chaque tâche sur chaque machine

Objectif : Optimisation du temps d'exécution, aussi appelé makespan

Ce problème est lié à l'ordonnancement parallèle et à l'affectation de tâches en tenant compte des contraintes de dépendances et des capacités des machines.



Pour nous attaquer au problème, on a choisi une heuristique

Une heuristique ?

Une heuristique est une **méthode pragmatique** permettant de résoudre des problèmes ou prendre des décisions rapidement, souvent basée sur l'expérience ou l'intuition plutôt que sur une approche exhaustive.

Elle permet l'approximation de la solution optimale.
Nous avons choisi d'utiliser une heuristique en raison de la nature NP-Complet Du problème

MIN-MIN SCHEDULING ALGORITHM

PRINCIPE: PRIORISER LES TÂCHES LES PLUS COURTES EN PREMIER.

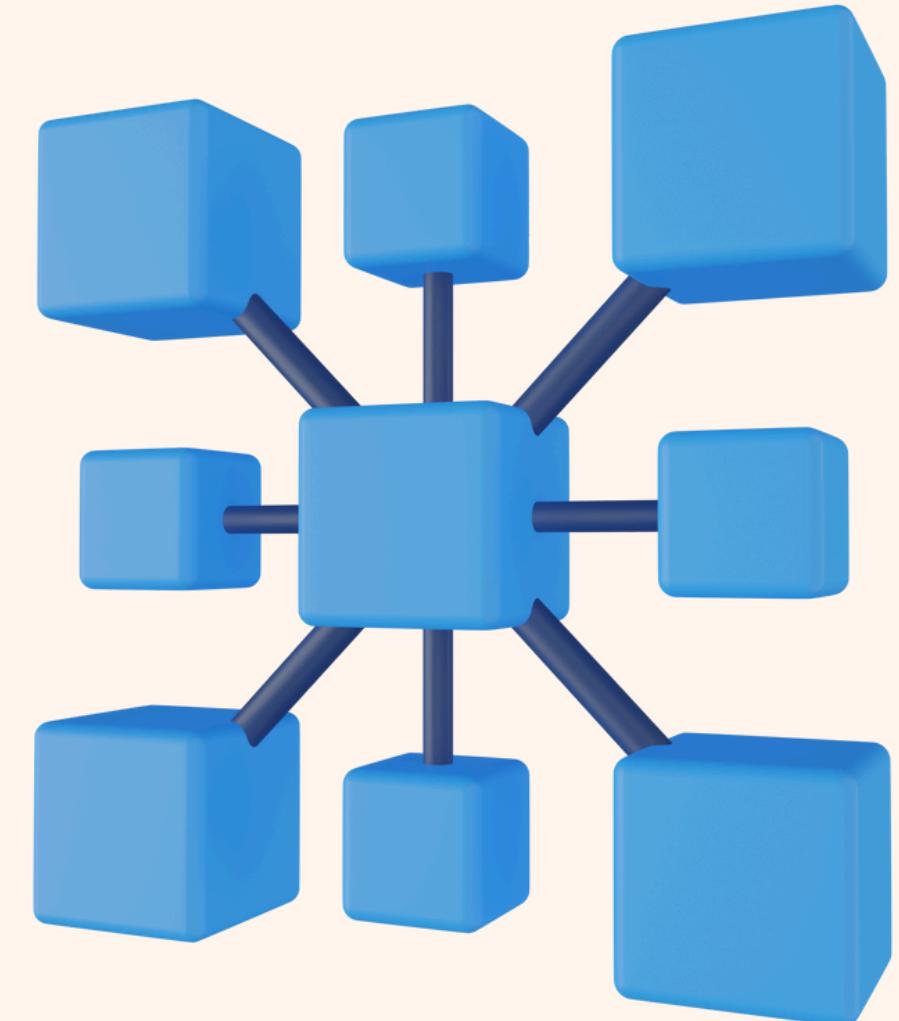
ÉTAPES:

- LISTER LES TÂCHES ET LEURS TEMPS D'EXÉCUTION SUR CHAQUE MACHINE.
- IDENTIFIER LA TÂCHE AVEC LE TEMPS D'ACHÈVEMENT LE PLUS PRÉCOCE (TEMPS ACTUEL + EXÉCUTION).
- ALLOUER CETTE TÂCHE À LA RESSOURCE CORRESPONDANTE.
- METTRE À JOUR LA DISPONIBILITÉ DE LA RESSOURCE.
- RÉPÉTER JUSQU'À ATTRIBUTION DE TOUTES LES TÂCHES.

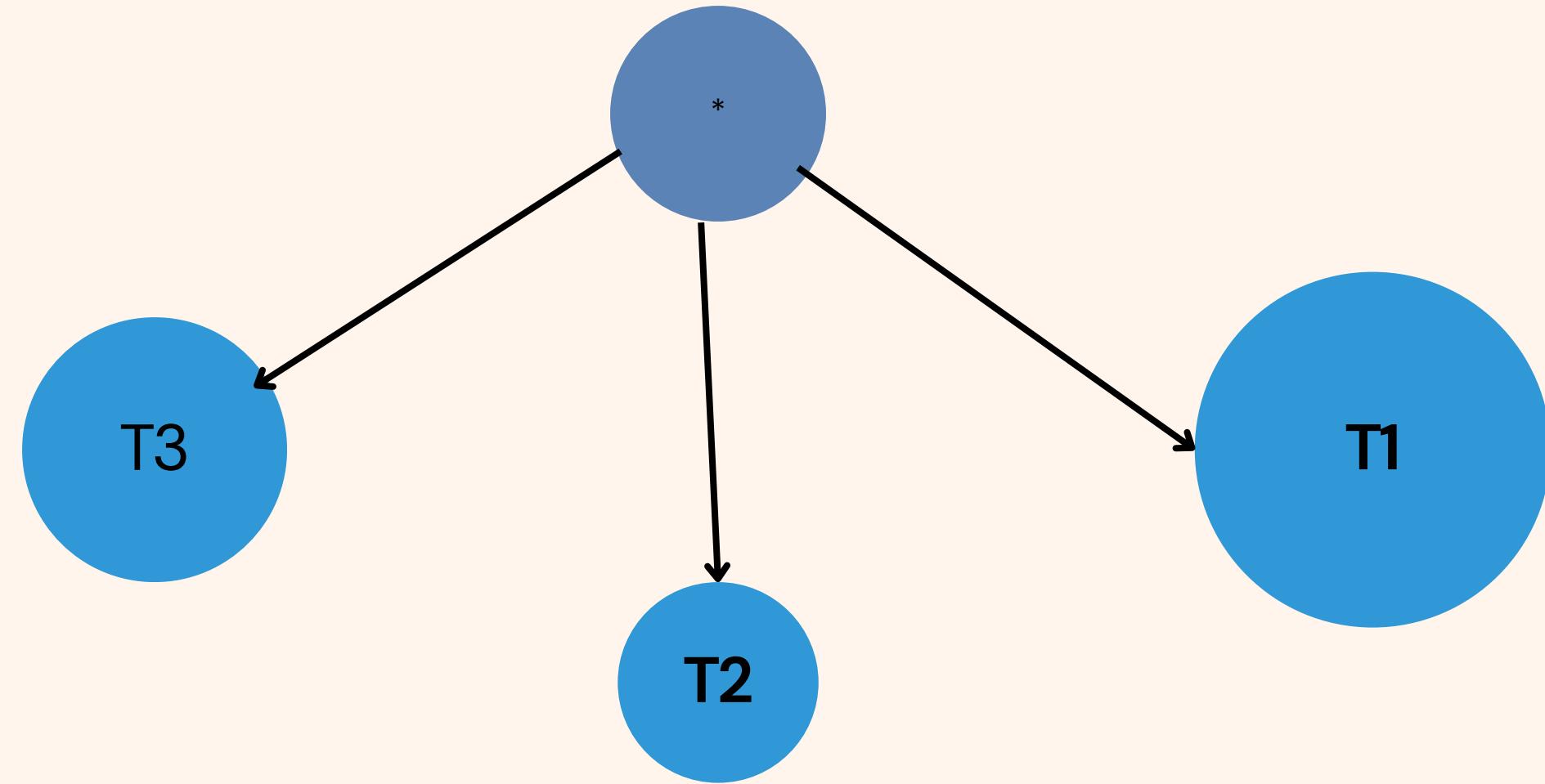
AVANTAGES: SIMPLE, EFFICACE POUR LES PETITES TÂCHES, RÉDUIT L'ATTENTE GLOBALE.

CAS D'USAGE: CALCUL DISTRIBUÉ, GRILLES DE CALCUL, CLOUDS HÉTÉROGÈNES.

À RETENIR: OPTIMAL POUR DES WORKLOADS DÉSÉQUILIBRÉS (MIX DE TÂCHES COURTES/LONGUES).

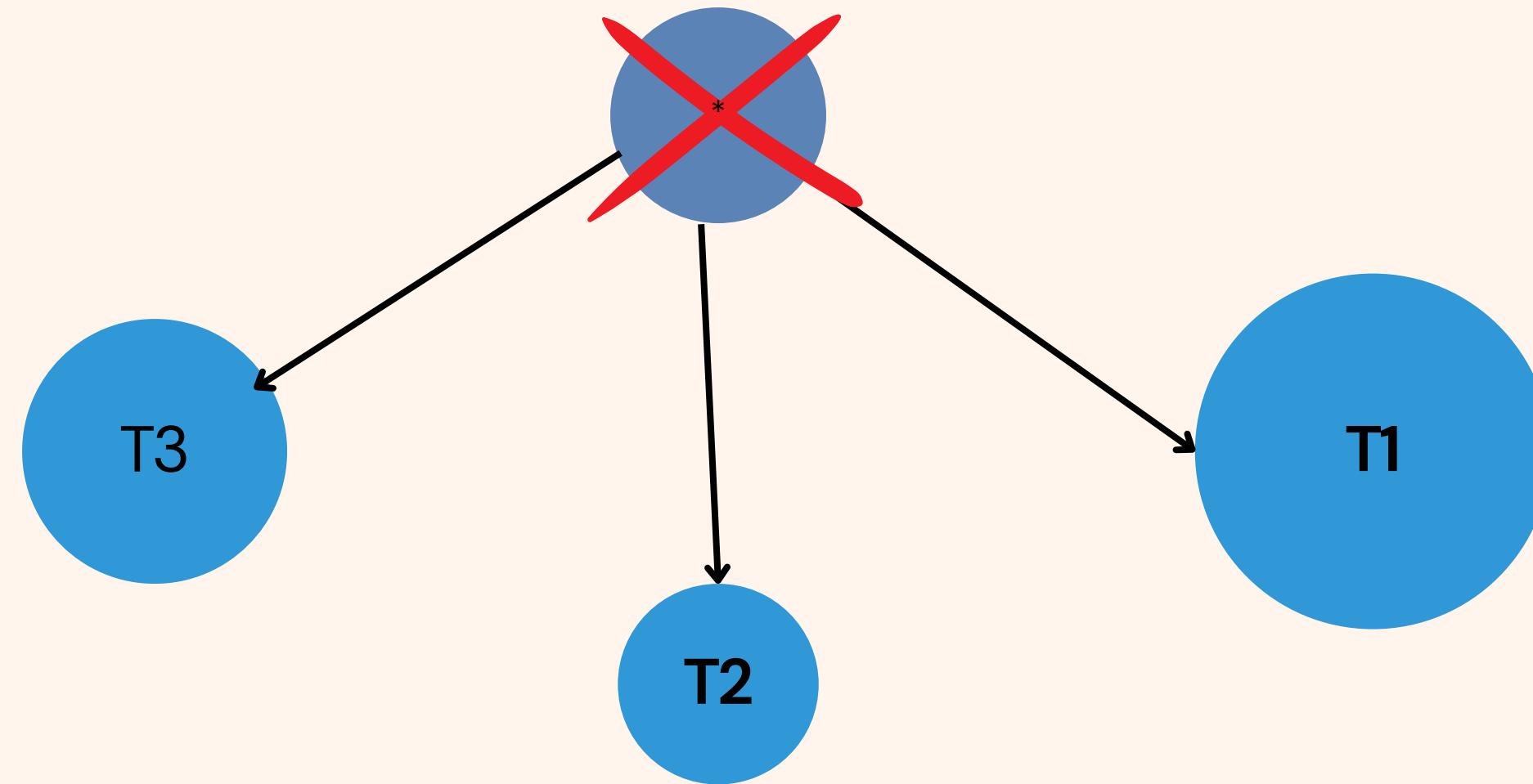


la tache *
prend 30
seconde être
exécuté



	T1	T2	T3	*
M1	40	20	60	30
M2	100	100	70	30

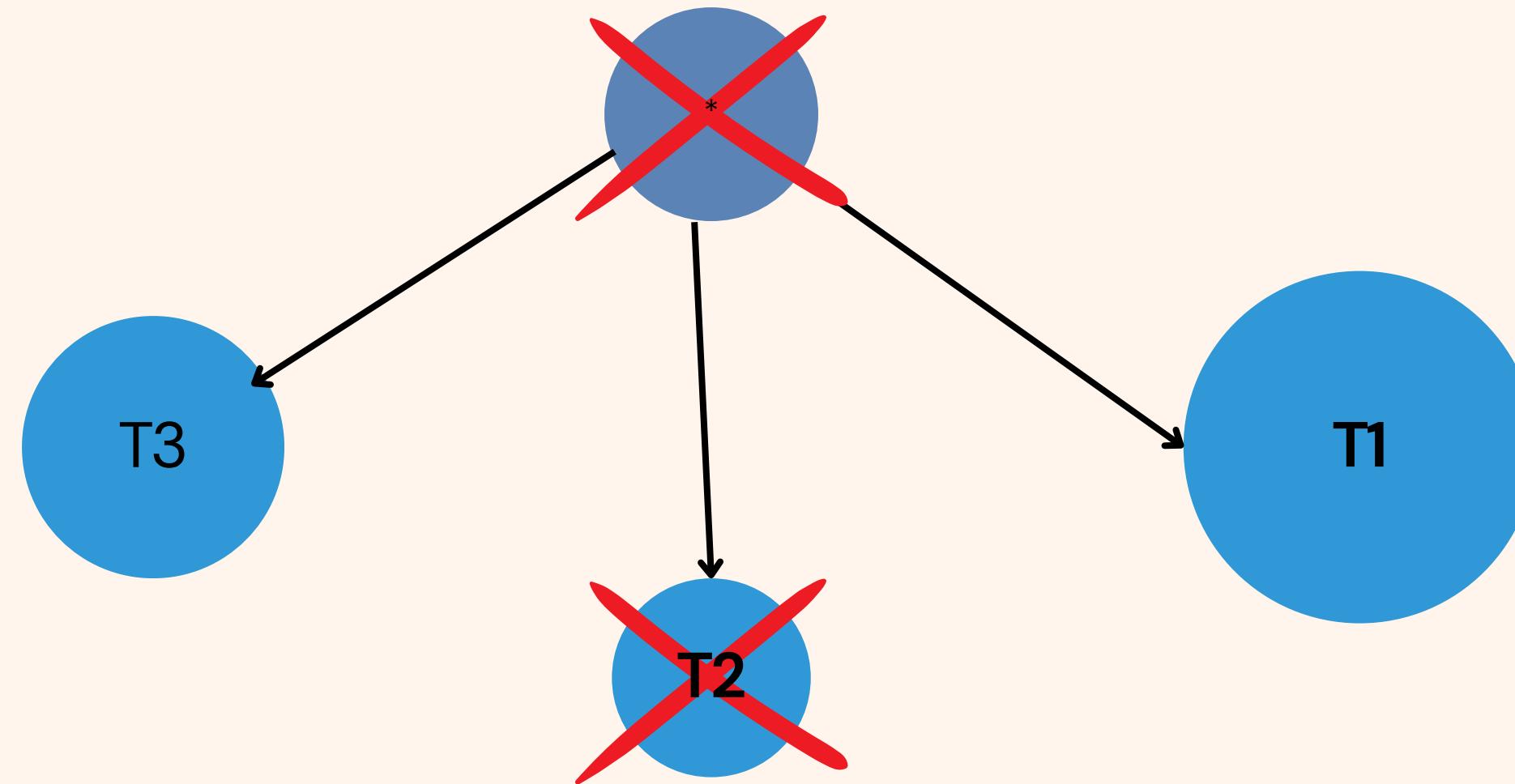
Tmp Total : 30s



	T1	T2	T3	*
M1	40	20	60	30
M2	100	100	70	30

M1 : *
M2 :

Tmp Total : 30s

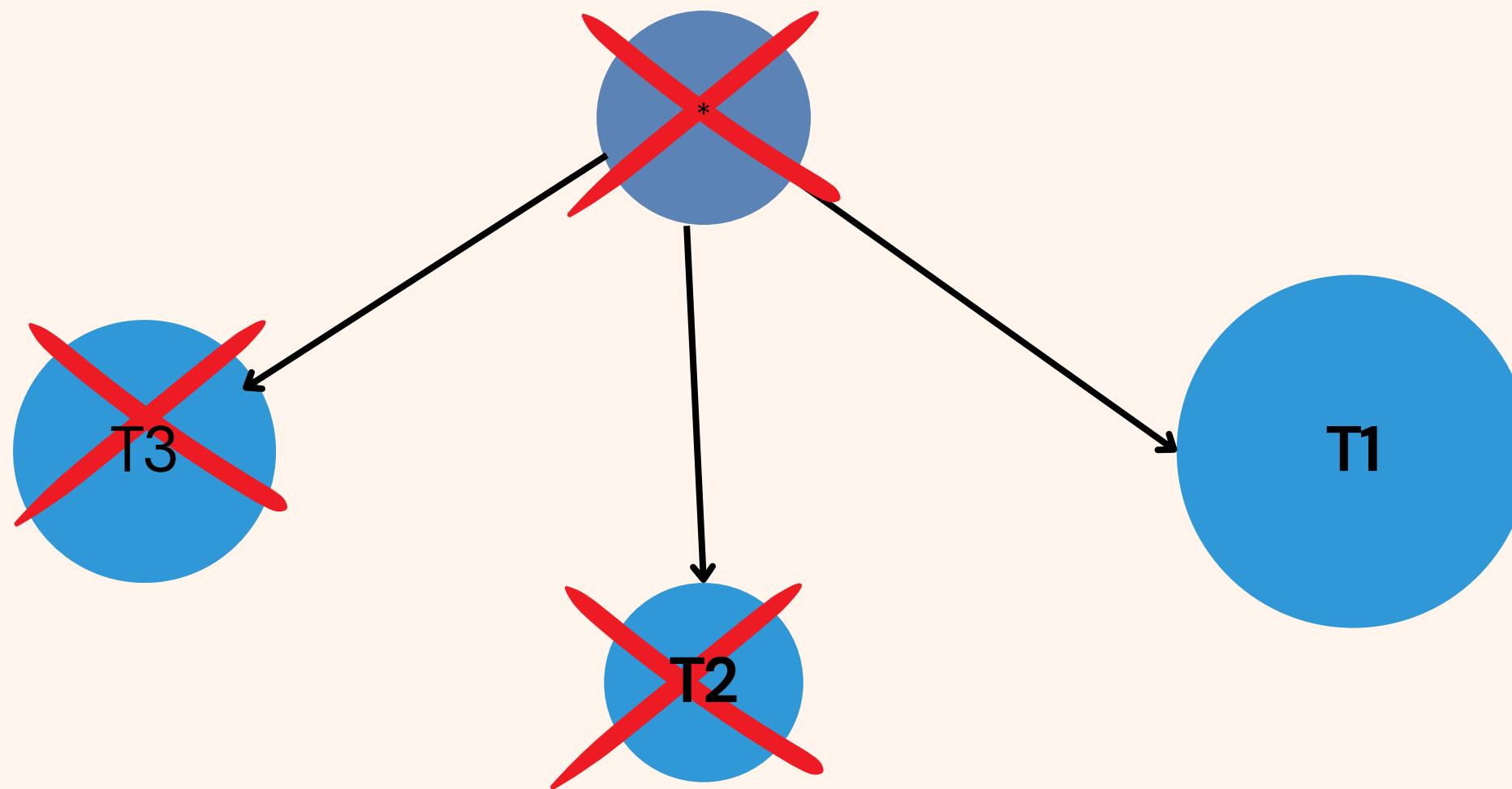


	T1	T2	T3	*
M1	40	20	60	30
M2	100	100	70	30

M1 : * → T2

M2 :

Tmp Total : 30s+20s

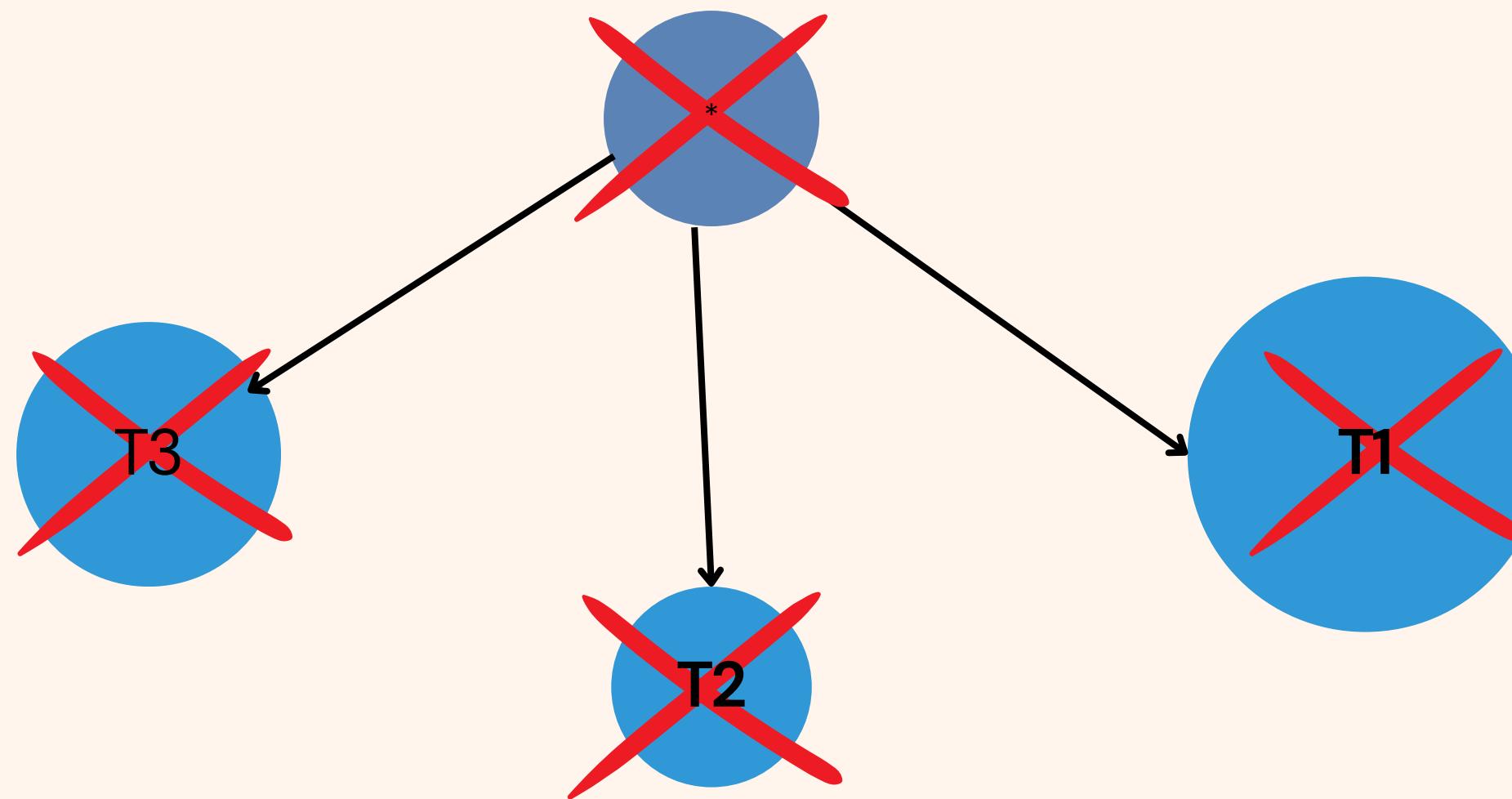


	T1	T2	T3	*
M1	40	20	60	30
M2	100	100	70	30

M1 : * → T2

M2 : T3

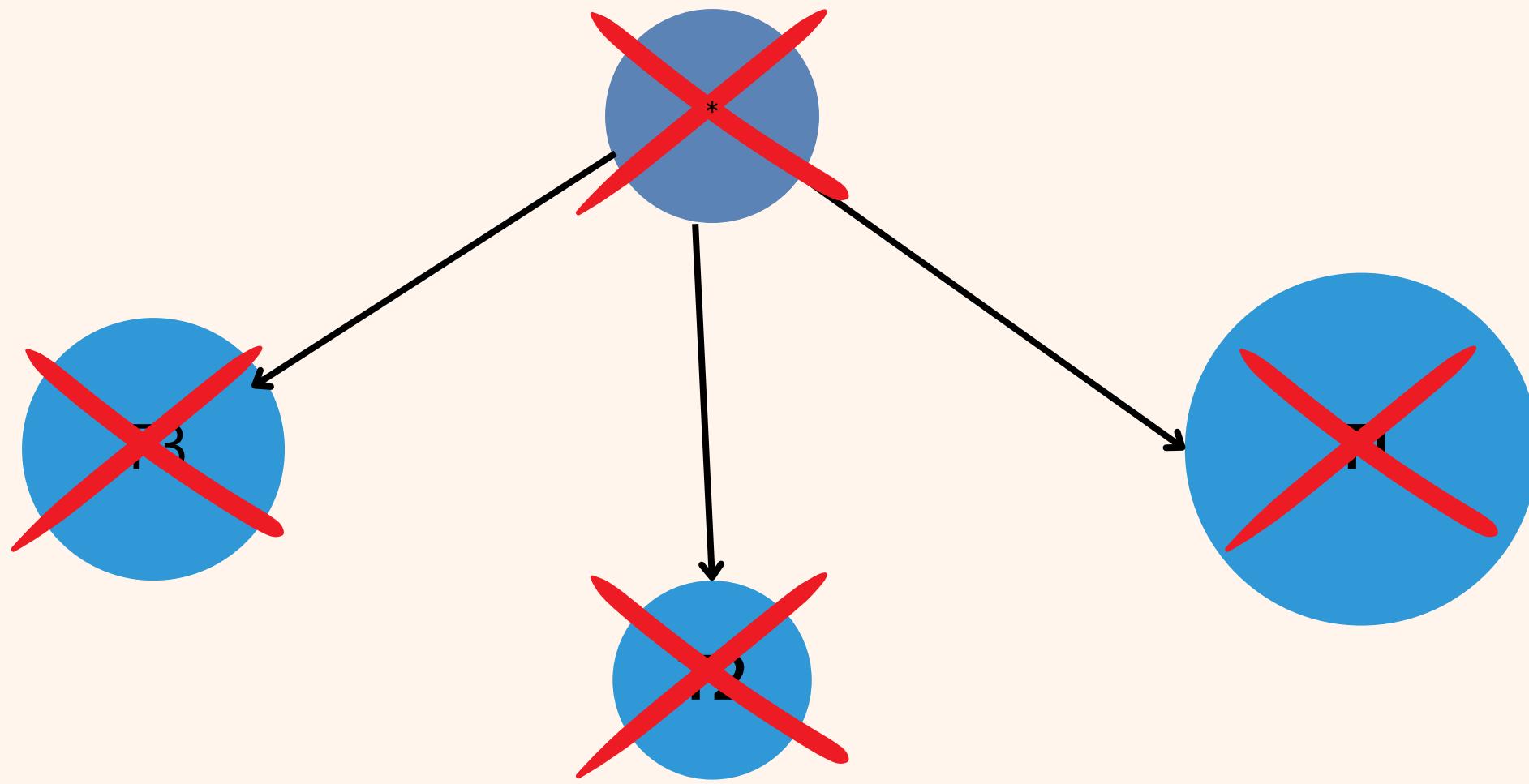
Tmp Total : 50s+50s



	T1	T2	T3	*
M1	40	20	60	30
M2	100	100	70	30

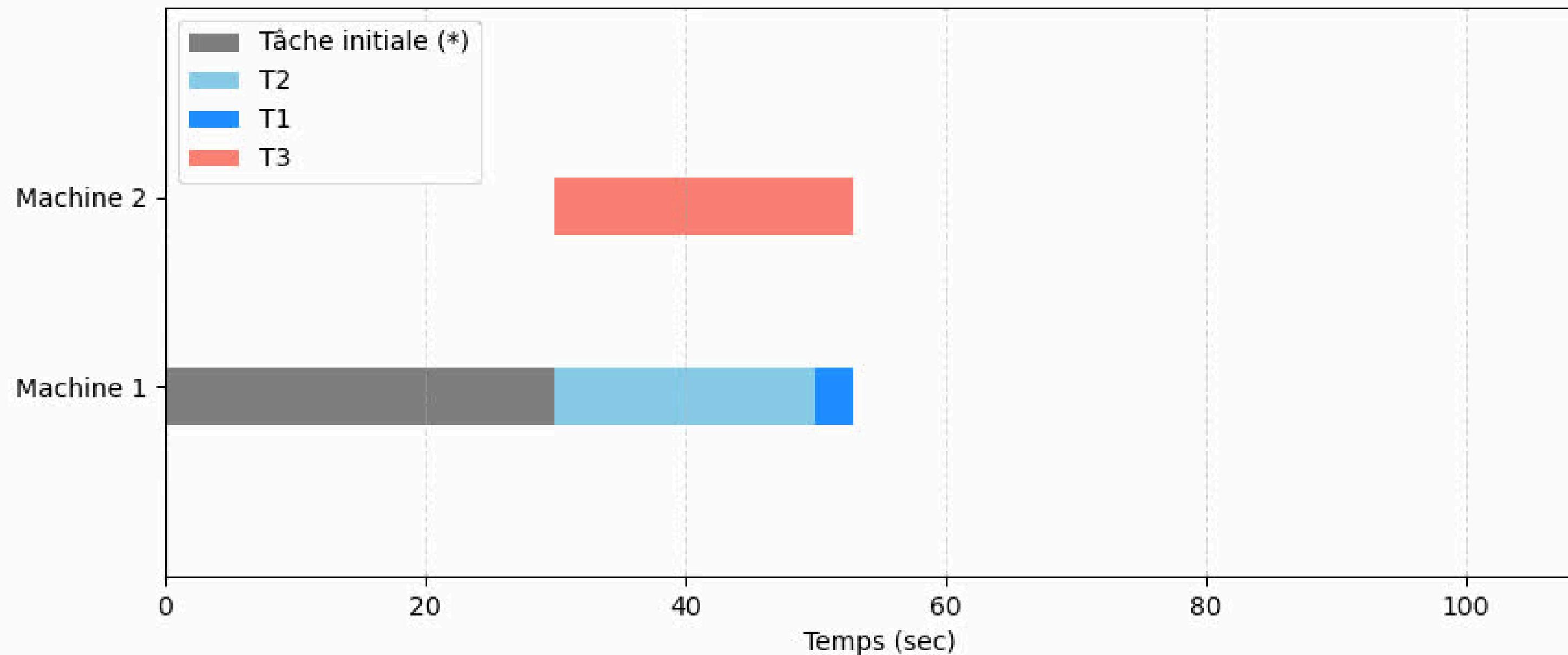
M1 : * → T2 → T1

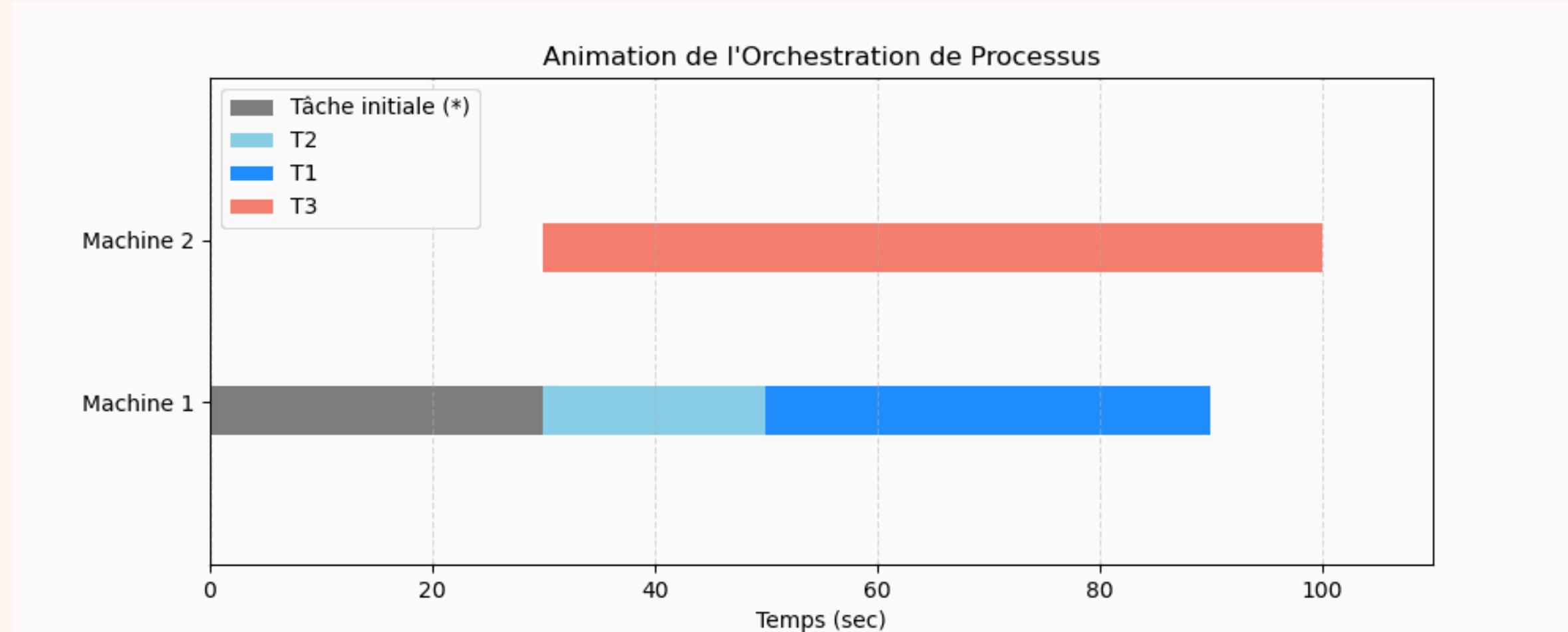
M2 : T3



Tmp Total : 100s

Animation de l'Orchestration de Processus





WOW!

Structures de données



Représentation avec Networkx

Module permettant la gestion efficace des DAGs.

Stockage des différents attributs des tâches sous la forme d'un tuple avec id de tâche et dictionnaire.

```
[('task1', {'time': 10, 'memory': 512})]
```

Un outil clé pour **modéliser les dépendances et optimiser l'ordonnancement**



Format d'entrée et sortie

Sous format demandé (Json adapté)

Fonctions élémentaires



read_graphe

Lit un graphe sous un format json et renvoie un graphe dans un format NetworkX.
Stocke les temps d'exécution et les besoins en mémoire dans la structure NetworkX



update_tasks

Prend en argument la liste des tâches qui viennent d'être exécutés et renvoie la liste des tâches qui peuvent être exécutés mise à jour



best_assignment_for_task

Calcule le temps d'achèvement possible sur chacune des machines, à l'aide d'un dictionnaire de planification.

Fonctions générales



min_min_schedule

Implémente l'algorithme Min-Min pour N machines.

L'algorithme sélectionne la tâche qui peut être terminée le plus tôt sur une machine donnée.
alternance entre update_tasks et best_assignment_for_task jusqu'à l'exécution de toutes les tâches



convert_schedule_to_json

Convertit le planning calculé au format JSON souhaité, afin de le déposer dans le dossier d'output.

Complexité temporelle en $O(t \log(t))$

Une complexité en t^2m

Avec :

- t : **nombre de tâches** du graphe
- e : **nombre d'arêtes** du graphe
- m : **nombre de machines** disponibles dans le problème

`min_min_schedule` : on boucle sur les tâches :

- appel à `update_tasks` : $O(t)$
- appel à `best_assignment_for_task` : $O(m)$
- sélectionne la meilleure tâche en $O(t)$

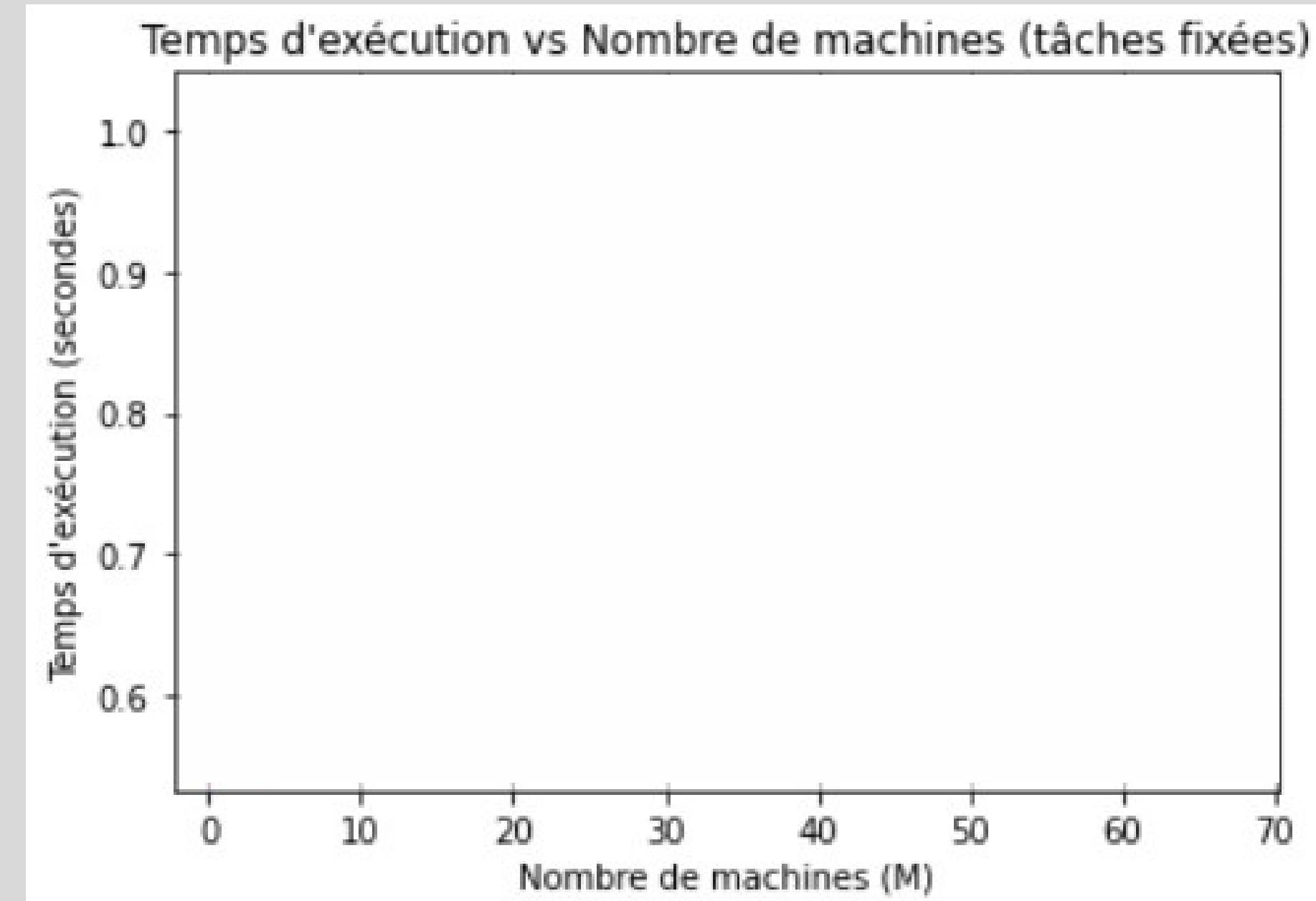
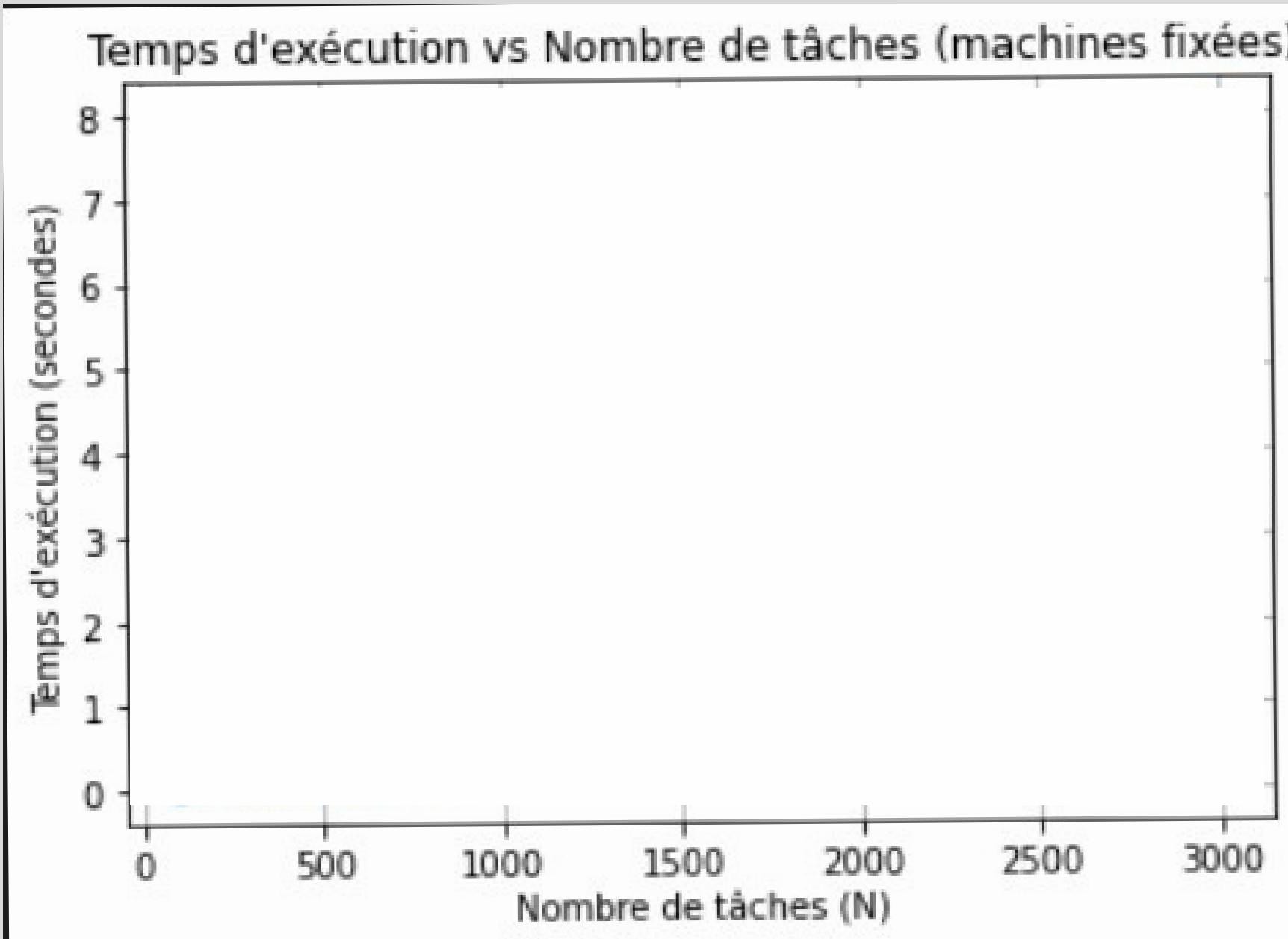
Le tout est donc de complexité totale : **$O(t^2m)$**

On regarde un peu en arrière :

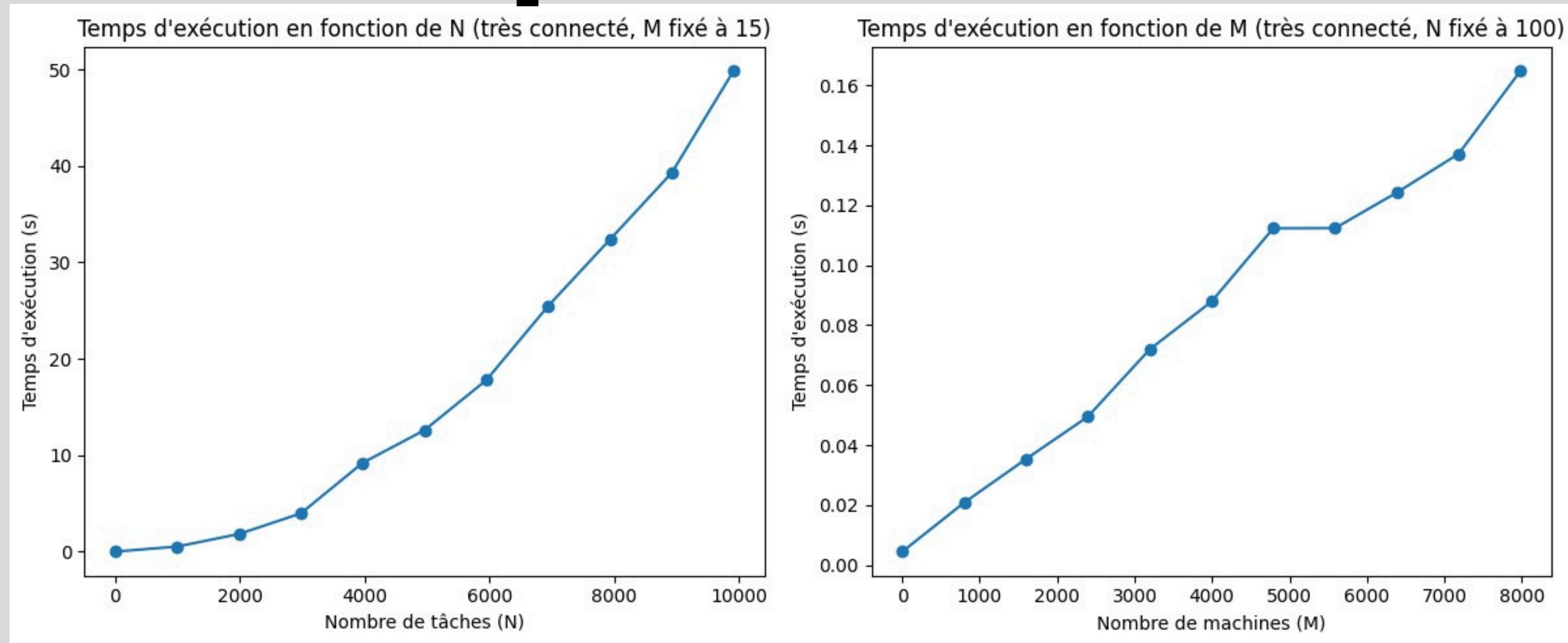
- Nous avons vu le fonctionnement de l'heuristique min-min
- Nous avons vu les éléments principaux de son implémentation

Passons à présent aux tests

Une complexité cohérente



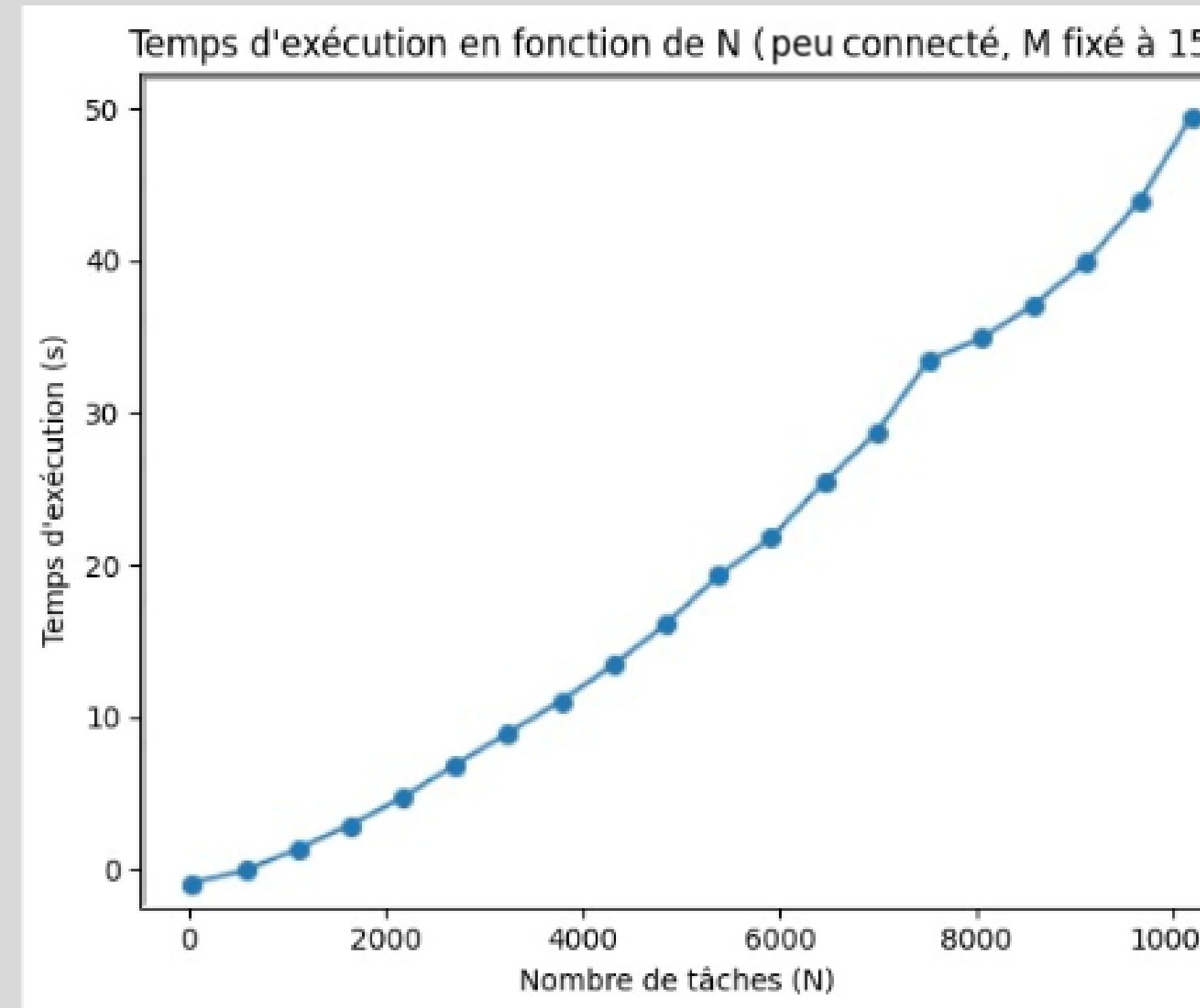
Une complexité cohérente



-> Profil **quadratique** en fonction de t

-> Profil **linéaire** en fonction de m 21

Une complexité cohérente



En moyenne dans des graphs peu connectés on se rapproche d'une courbe linéaire !

Explication des métriques

Charge Moyenne :

- **Temps total / Nombre de machines**
- **Exemple : $60 \text{ sec} / 3 \text{ machines} = 20 \text{ sec par machine.}$**

Chemin Critique :

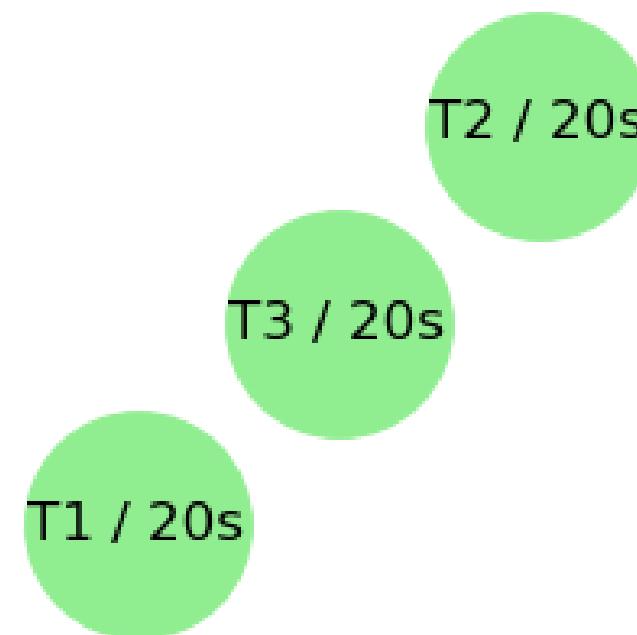
- **Enchaînement de tâches dépendantes**
- **Exemple : $T1 (20s) \rightarrow T2 (30s) \rightarrow T3 (50s) = 100 \text{ sec.}$**
-

LB (Lower Bound) :

- **Temps minimal théorique d'exécution = $\max(\text{charge moyenne, chemin critique})$**
- **Dans cet exemple, $LB = \max(20, 100) = 100 \text{ sec.}$**

Exemples

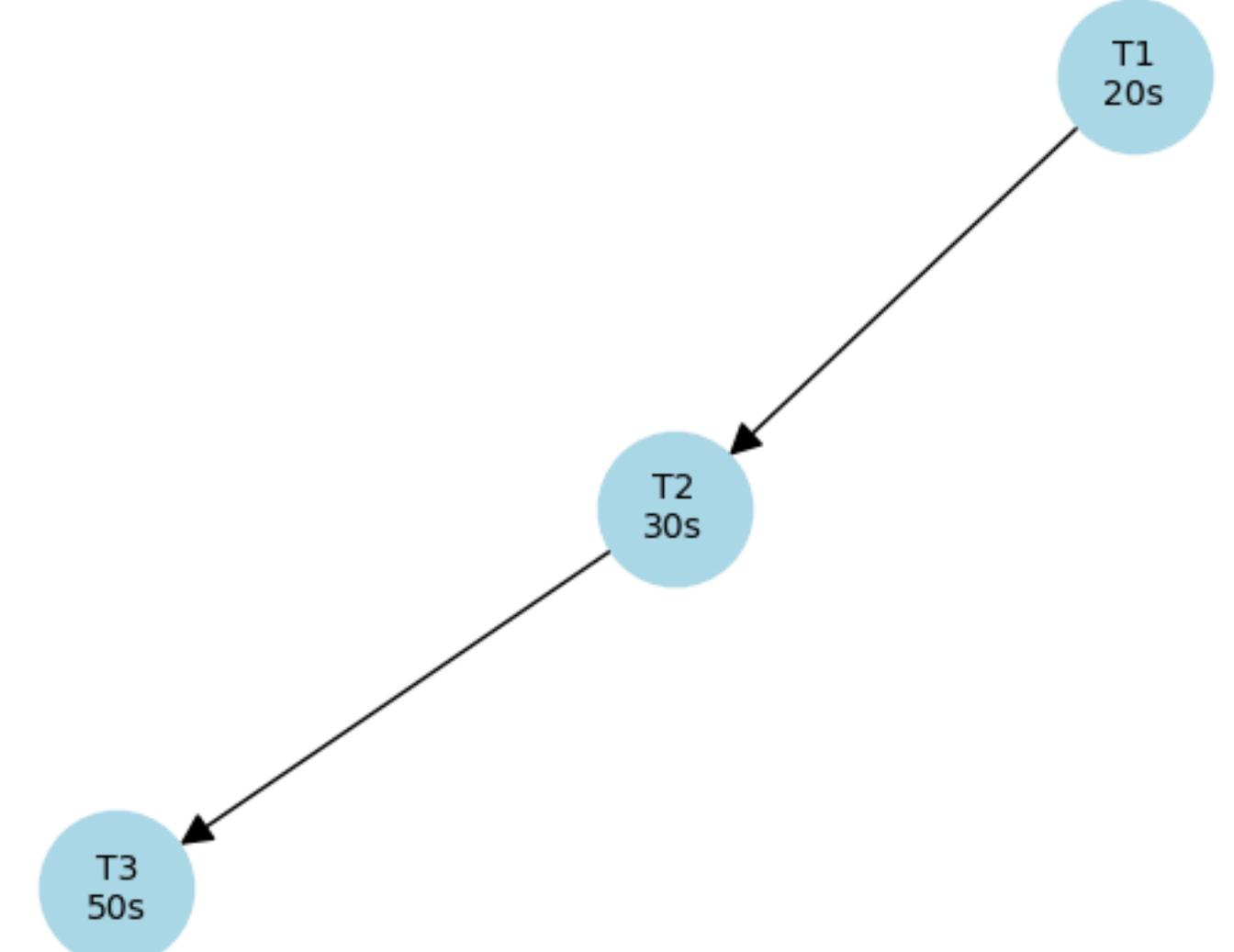
Répartition de la Charge Moyenne



temps d'exécution 20 secondes

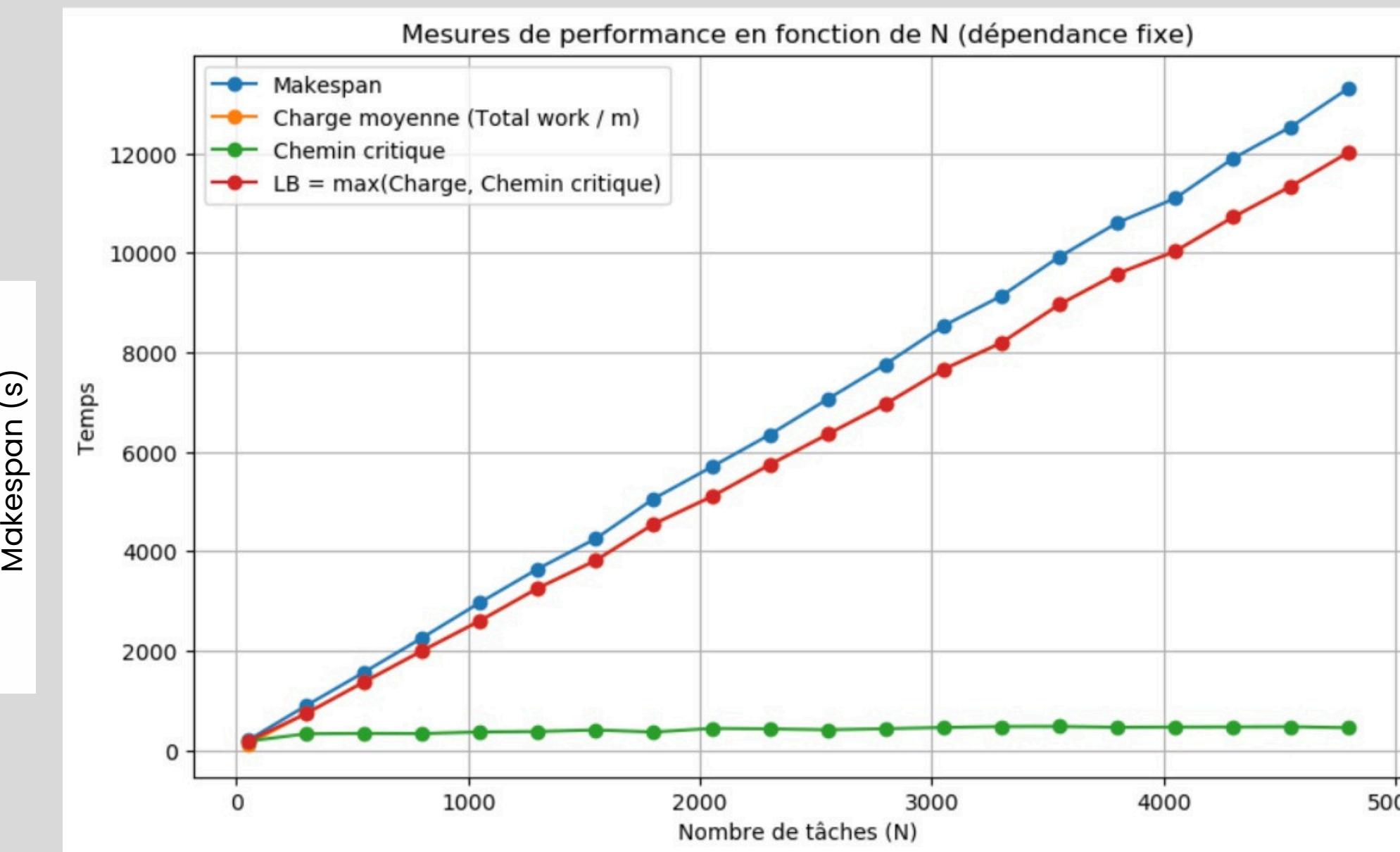
Avec nbr de machine ≥ 3

Chemin Critique



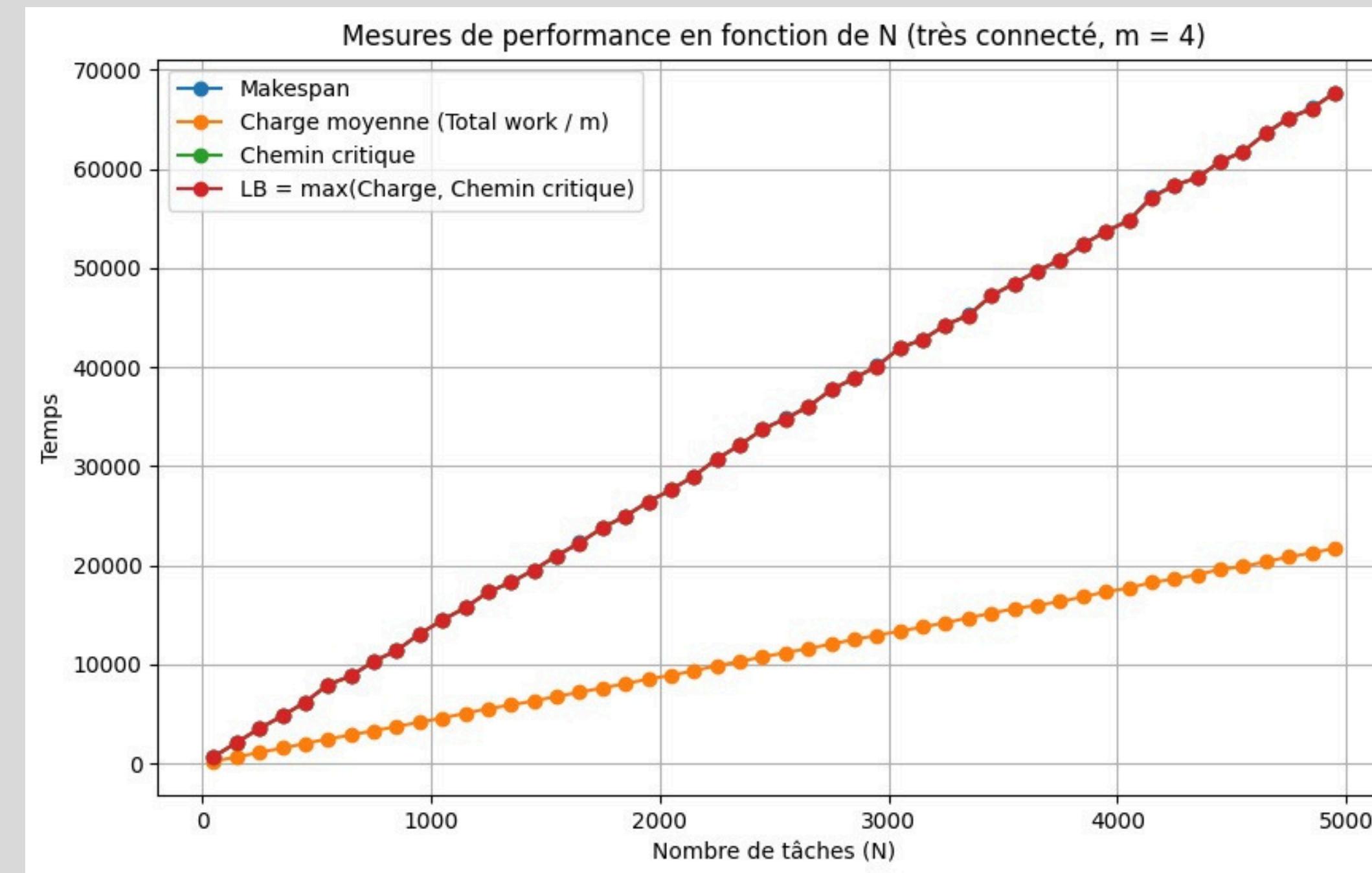
**temps d'exécution
100 secondes minimum
(exécution séquentielle)**

Un makespan optimisé



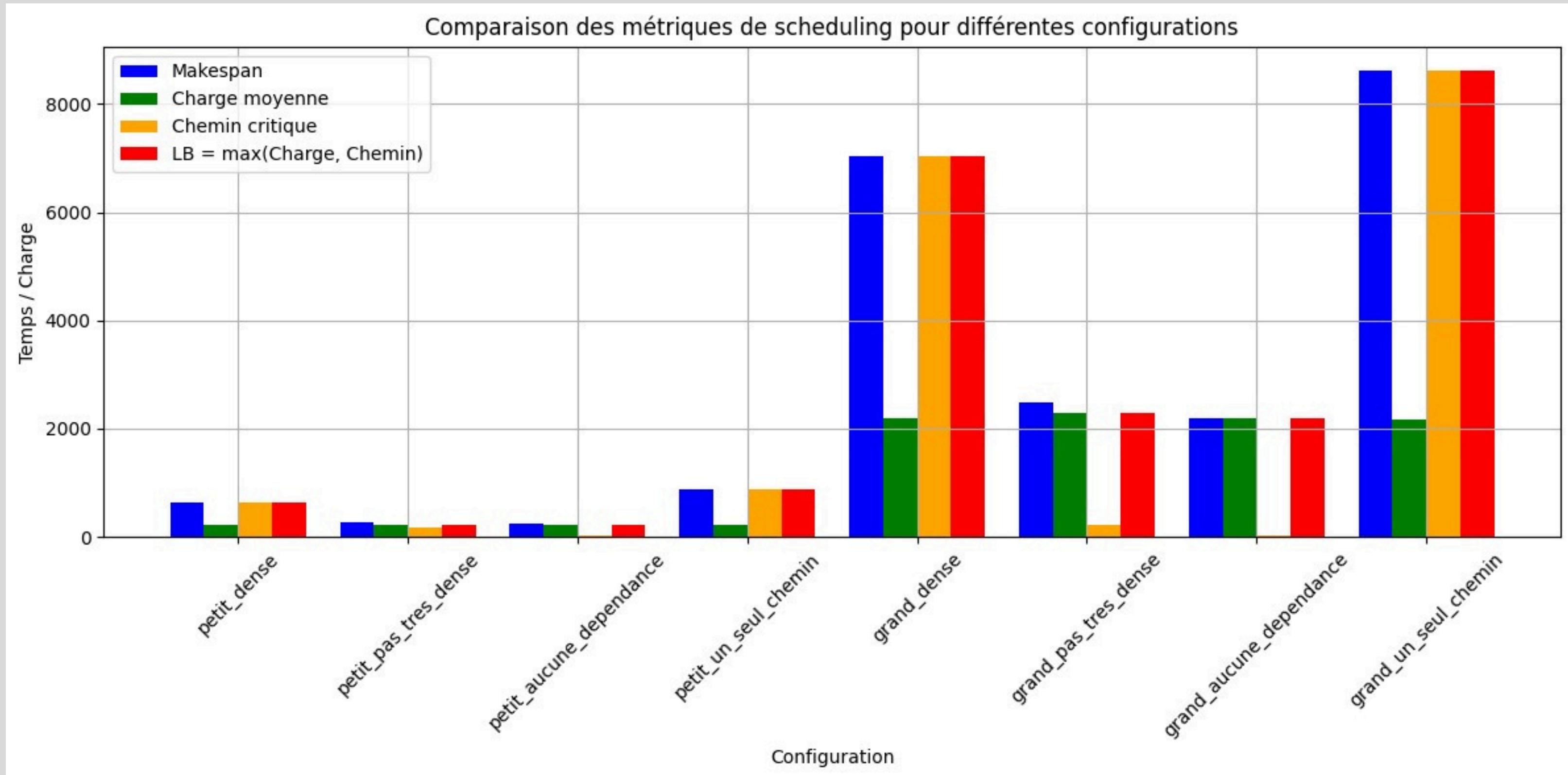
Solutions très proche du lower bound théorique ici
dans le cas où la charge moyenne est la lower
bound
-> résultats satisfaisants

Un makespan optimisé



Encore une fois des résultats proche du lower bound théorique
ici dans le cas d'un long chemin critique

Tests sur différents types de graphs



Le transfert sur le Cloud

Gestion des dépendances

NetworkX a pu être importé pour la manipulation de graphes.

Matplotlib n'était pas disponible dans l'environnement.

Solution : génération d'un JSON contenant les coordonnées (nœuds, arêtes, points de courbes) afin de pouvoir tracer les graphes localement ou dans un environnement compatible.



Le transfert sur le Cloud

Gestion des ressources

Etant donné le graph des tâches et dépendances nous avons construit un script bash qui adapte la puissance de notre unique lambda fonction (via l'appel CLI) à la taille du graph pour n'utiliser que des ressources nécessaires au calcul de l'ordonnancement.



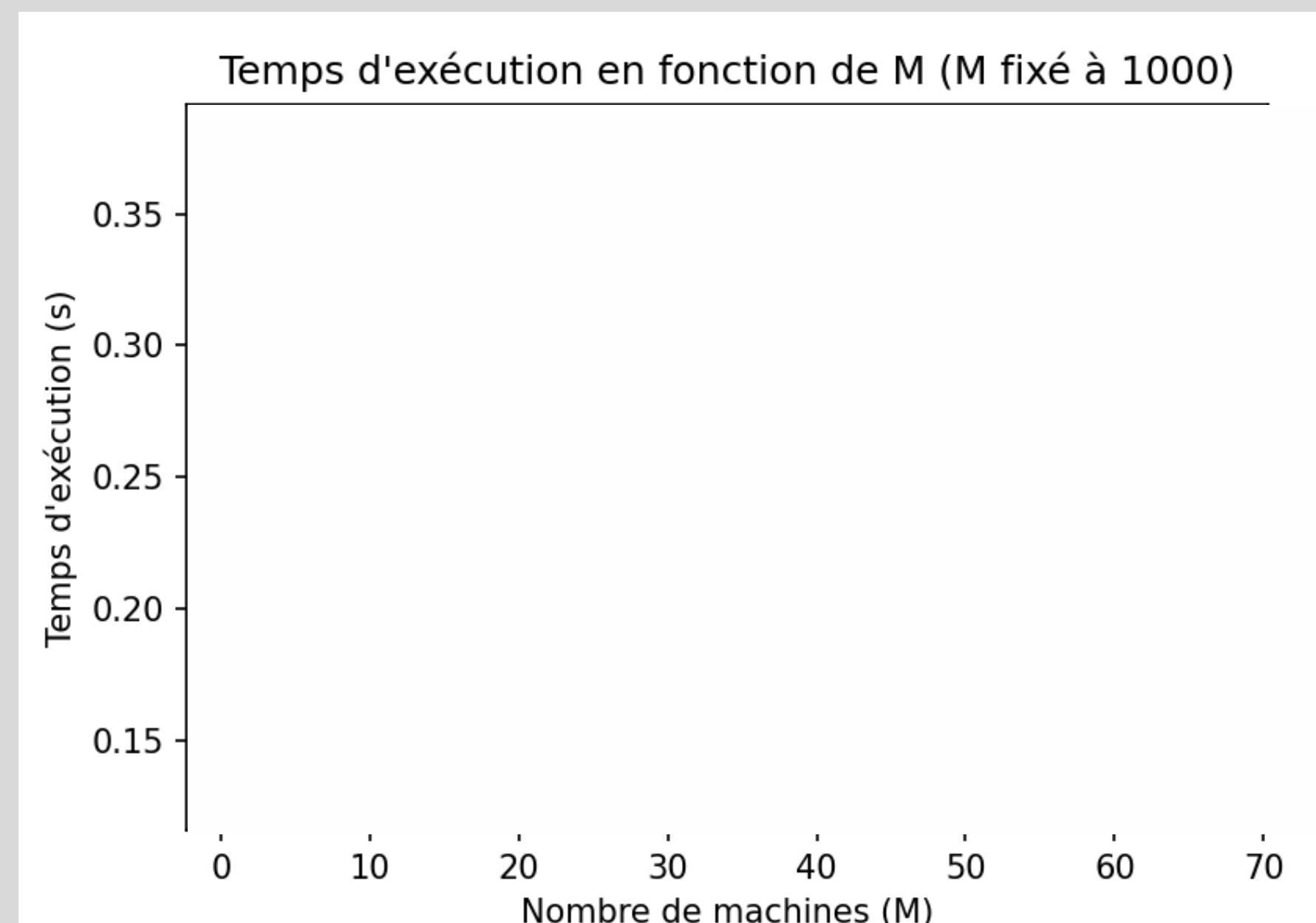
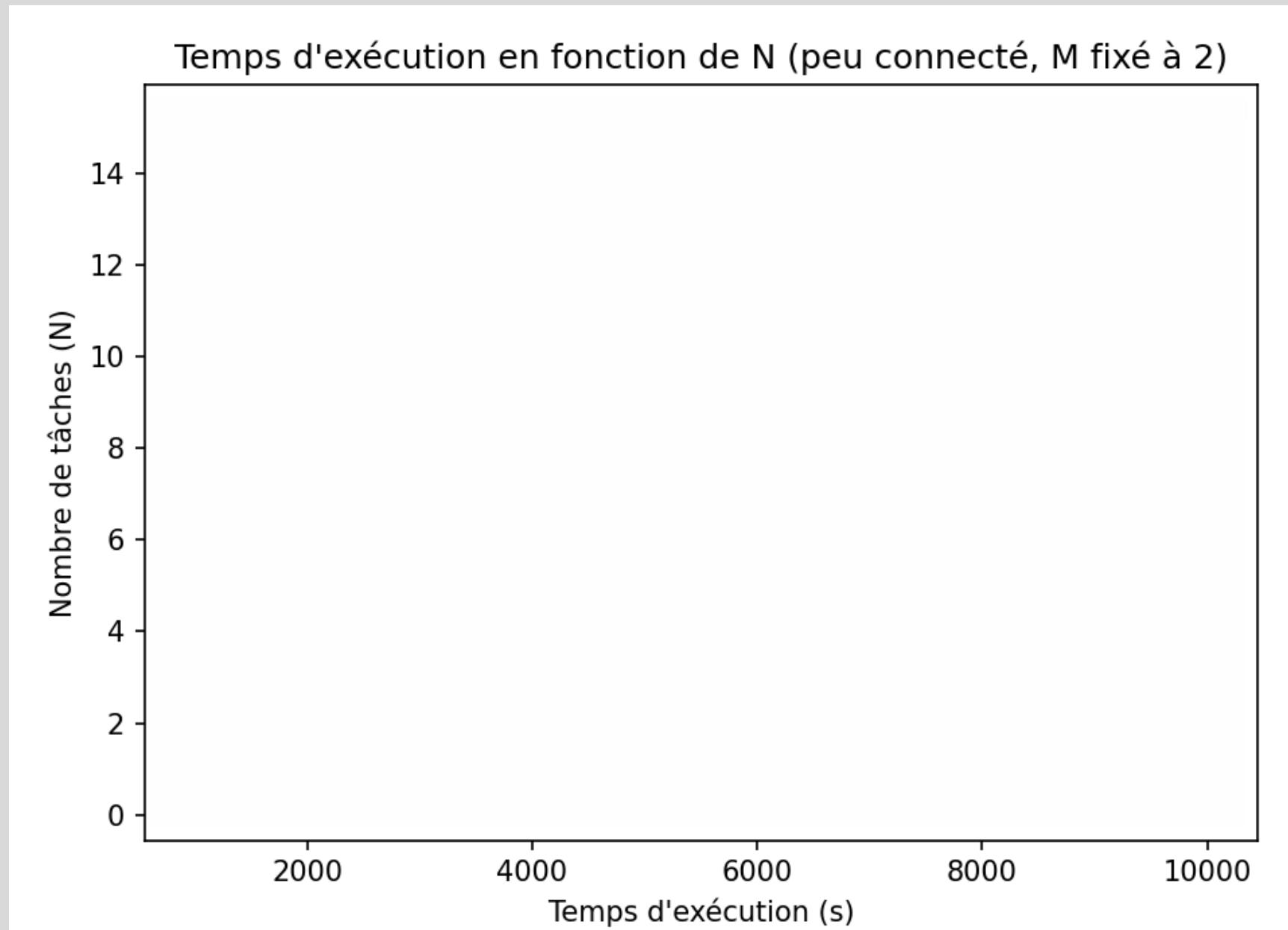
Le transfert sur le Cloud

Etat final :

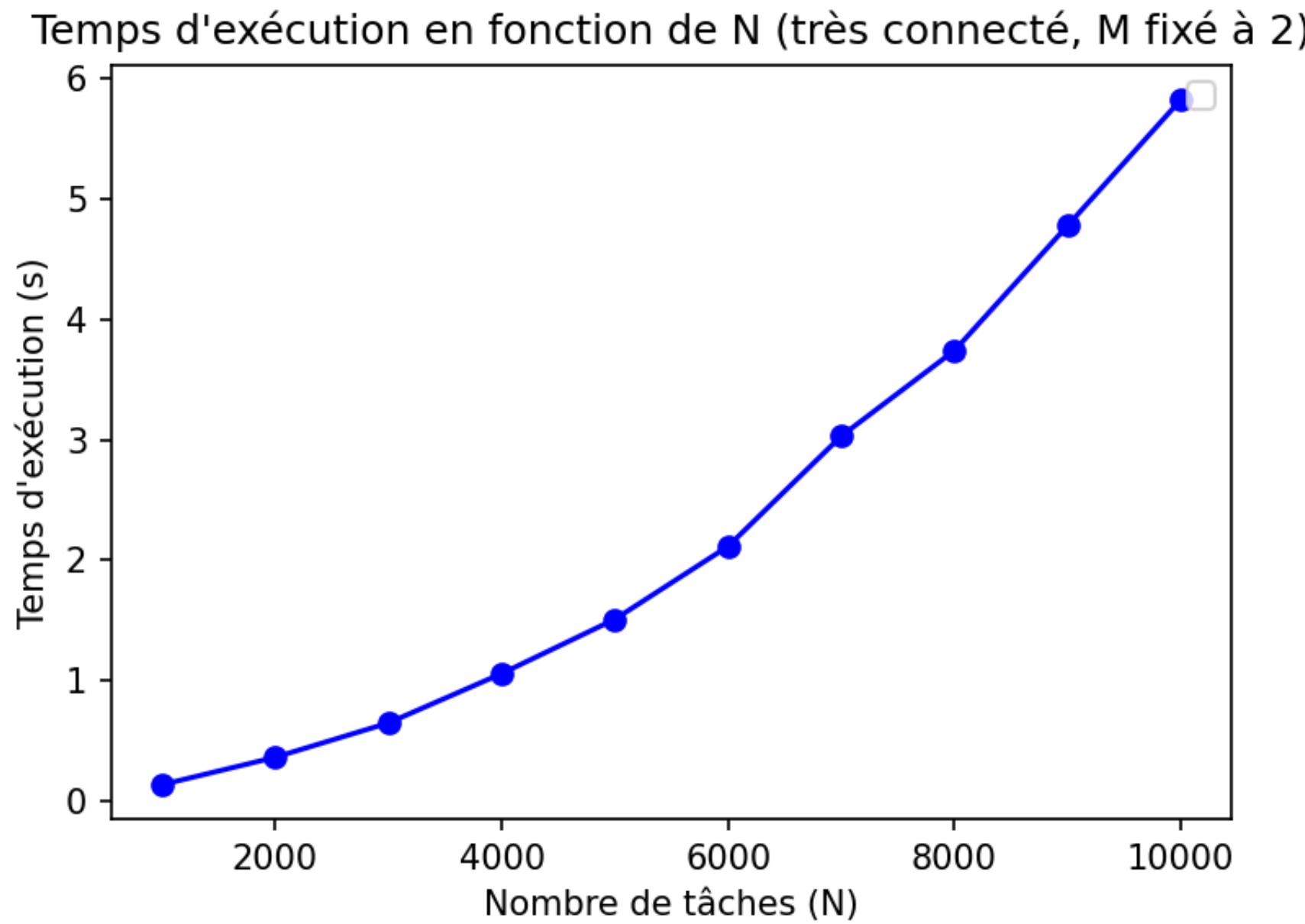
- L'implémentation est finalisé et fonctionnelle sur tout types de graphs.
- nombre de machines adaptable
- Puissance de la lambda fonction adaptable au graph grâce au script bash
- On a pu faire tourner notre fonction sur des graphes de 100k



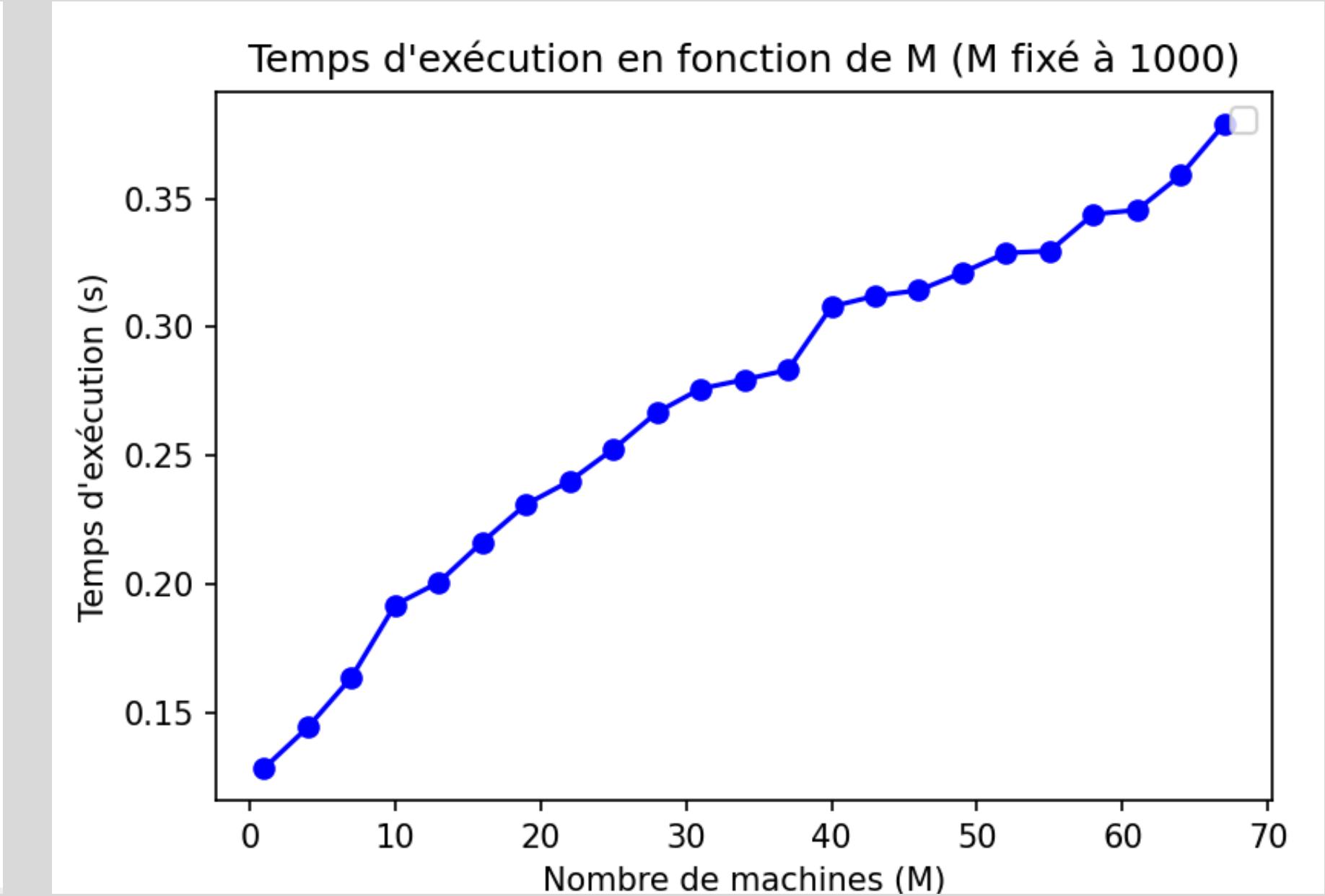
Une complexité cohérente



Une complexité cohérente

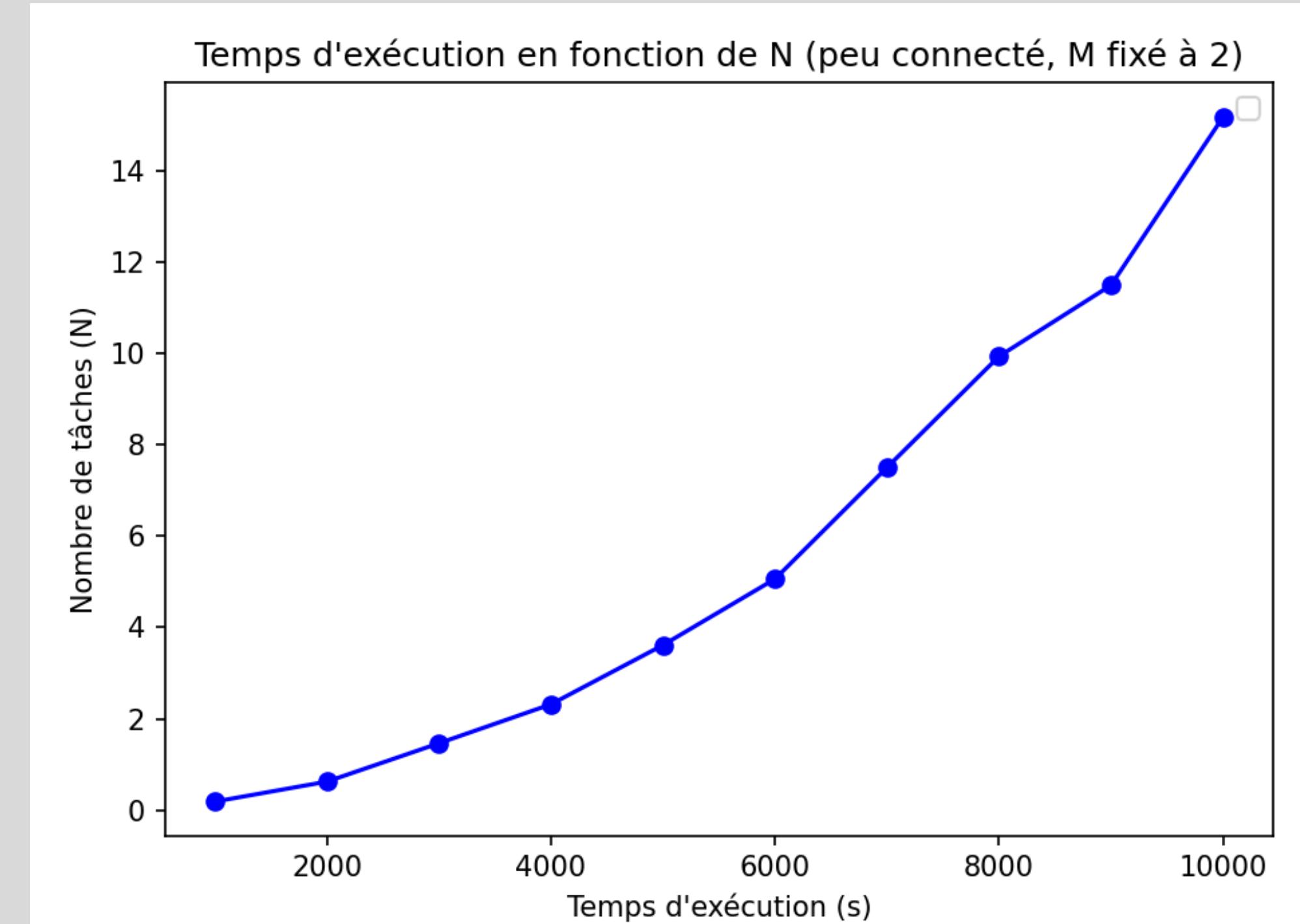


-> Profil **quadratique** en fonction de t



-> Profil **linéaire** en fonction de m 32

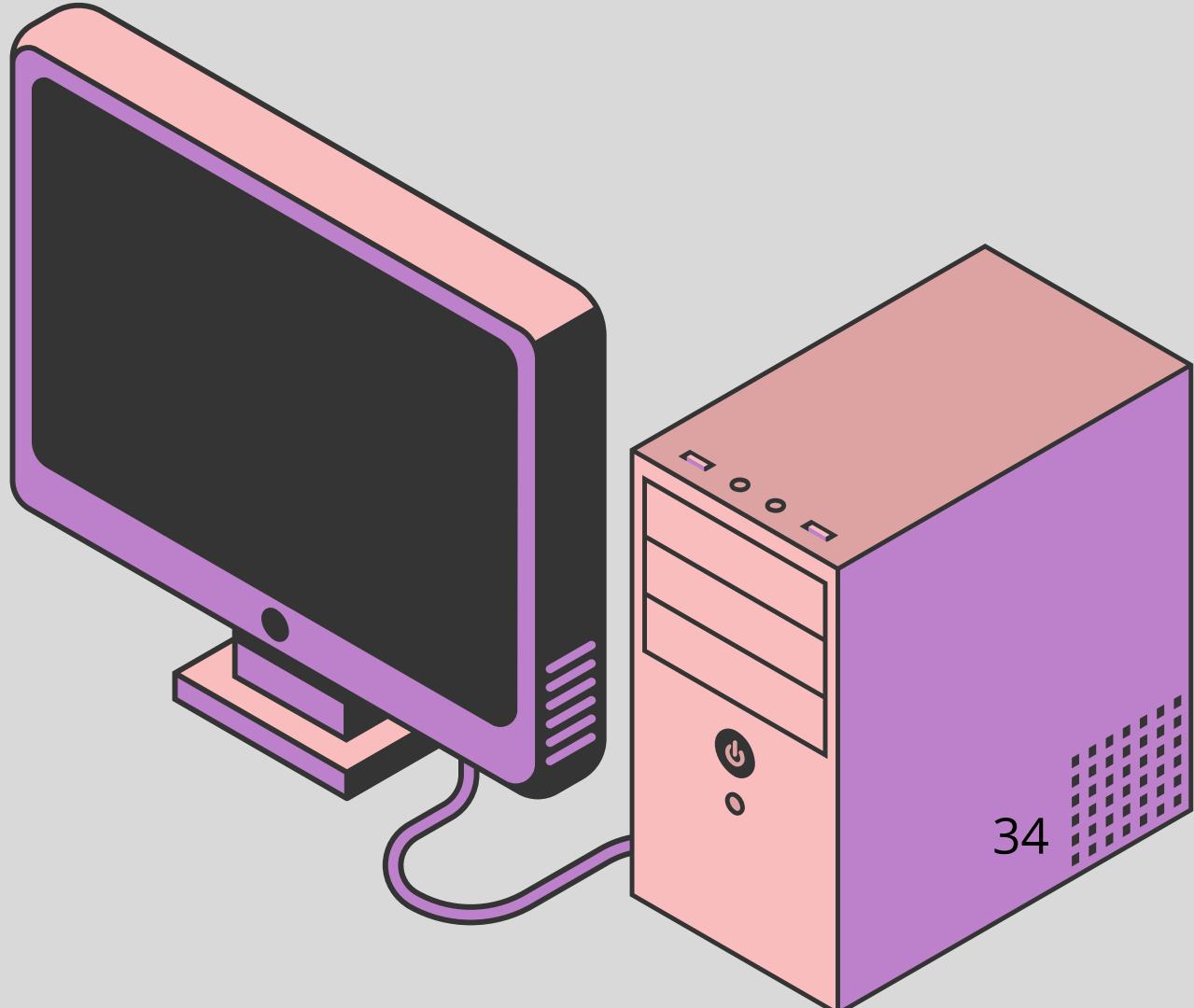
Une complexité cohérente



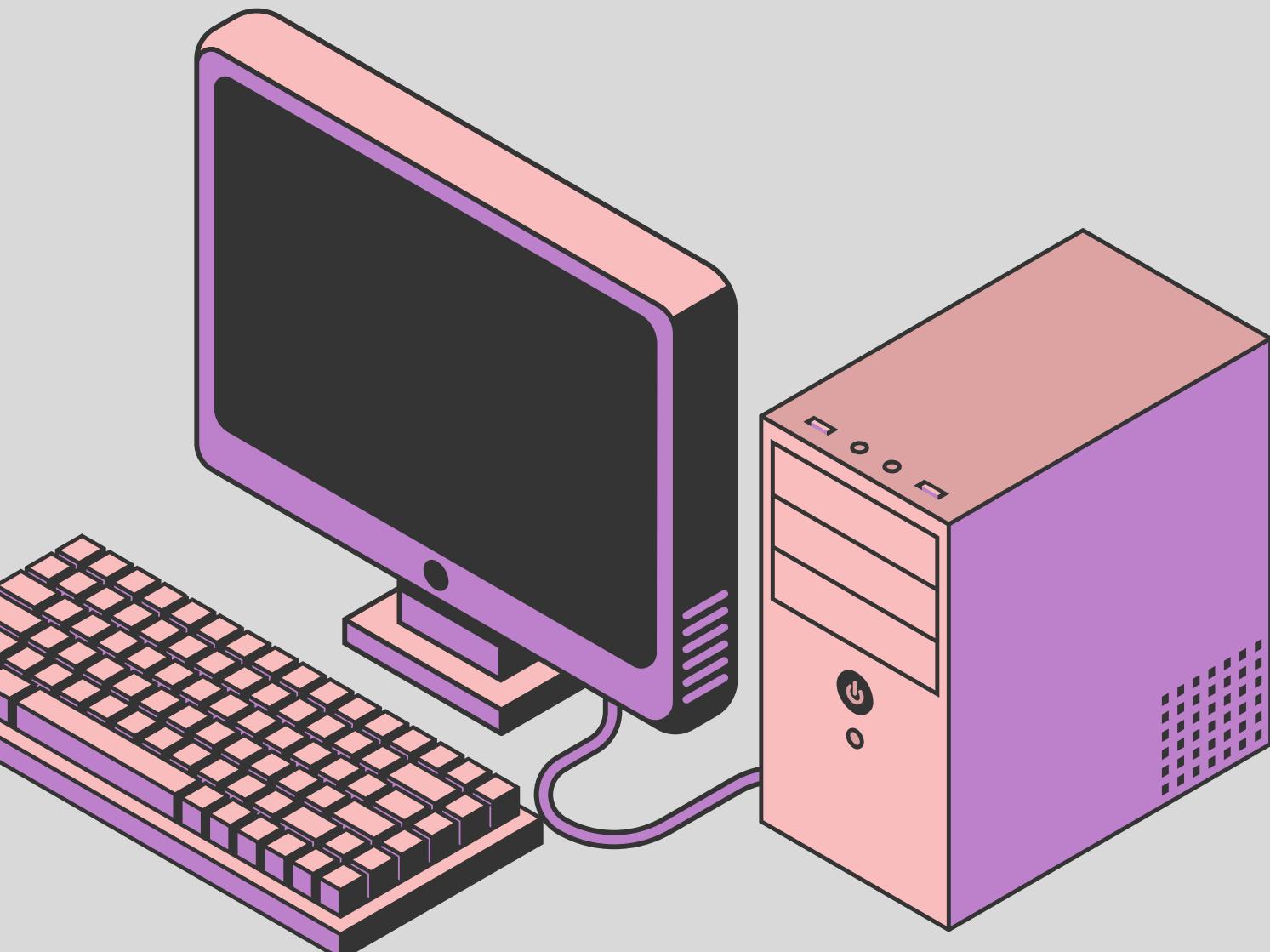
Dynamique linéaire un peu moins observée sur le Cloud,
dépend peut être des graphes générés

Ce qu'il faut retenir !

- Nous avons **réussi l'implémentation de l'heuristique Min Min et l'avons optimisé**
- Nous avons fait des **tests sur l'efficacité de notre algorithme sur tout types de graphs que ce soit en local ou sur le cloud**
- L'algorithme est totalement opérationnel et l'utilisation de ressource est adapté à la à la taille du graph



Si nous avions eu accès à plusieurs lambda fonctions :



- Nous aurions géré le traitement des graphs par des lambda fonctions différentes selon leurs taille.
- Tout cela aurait été géré par une lambda fonction mère qui se serait **chargé de rediriger le graph vers la lambda fonction adapté à sa taille**

Merci pour votre attention !