PEDIBUS I

Un problema di Ricerca Operativa

A cura di Nicolò Salvi e Giacomo Manzoli

SOMMARIO

01

INTRODUZIONE

IL PEDIBUS

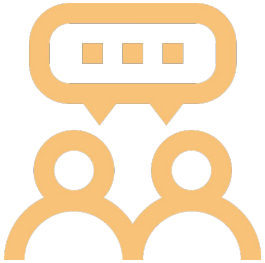
Pedibus (in inglese Walking bus) è il termine utilizzato per descrivere un sistema organizzato di trasporto scolastico a piedi per gli studenti.

L'idea di base è quella di creare un percorso sicuro e predefinito lungo il quale i bambini possono camminare a scuola insieme, solitamente sotto la supervisione di adulti, come genitori o volontari.

VANTAGGI

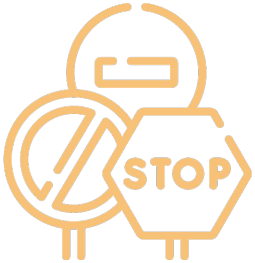
Contribuisce a ridurre il traffico intorno alle scuole

Promuove uno stile di vita attivo



Favorisce la socializzazione

Sensibilizza all'importanza della sicurezza stradale



02

IL PROBLEMA

INPUT

1. Indirizzi di *n* bambini (un bambino per indirizzo)
2. Indirizzo della scuola.

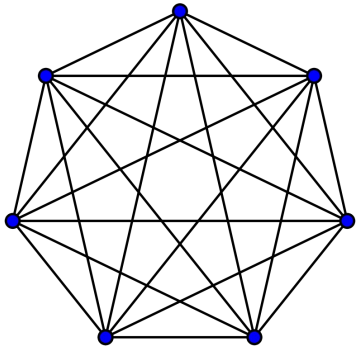
GOAL

- Determinare il numero **minimo** di percorsi che partono dalla casa di un bambino e terminano alla scuola passando da altre case, in modo tale che: ogni bambino *i* sia parte di esattamente (ex) 1 percorso e la durata del suo percorso tra la propria casa e la scuola sia non superiore a δ volte la sua distanza minima dalla scuola.

ASTRAZIONE

Consideriamo un grafo completo **G = (N, A)**

- I nodi **N = {0..n + 1}** rappresentano le fermate del pedibus, ovvero le case dei bambini e la scuola (nodo radice);
- Gli archi in **A = (i, j) : i, j ∈ N** rappresentano le connessioni tra i nodi;
- Il coefficiente $\mathbf{c_{i,j}}$ fornisce la lunghezza del cammino più breve dal nodo i al nodo j;
  - ➜ $c_{i,0}$ è la lunghezza del cammino più breve dal nodo i alla scuola
- Il parametro **δ > 1** rappresenta il rapporto massimo consentito tra la lunghezza del percorso valutato e la lunghezza del percorso più breve.

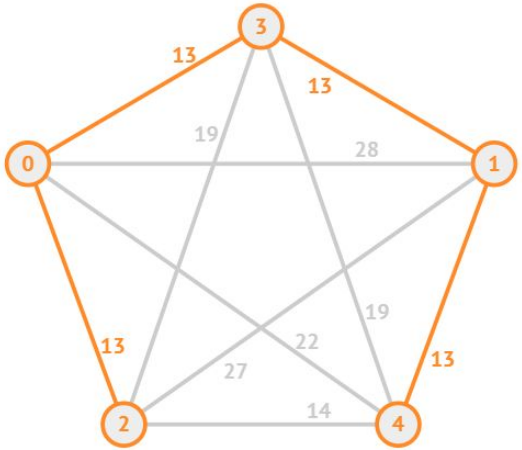


Un esempio di grafo completo

FORMALIZZAZIONE

Il problema può essere visto come un caso speciale di Spanning Tree, in cui l'obiettivo è trovare l'albero di copertura **realizzabile**, con radice in 0, avente il **minimo numero di foglie**.

Condizione:
Uno Spanning Tree è considerato **realizzabile** se nessun nodo è distante dalla radice più di δ volte il percorso più breve dal nodo stesso a 0 ($c_{i,0}$).



Esempio di **Spanning Tree** su un grafo K_5 con 2 foglie

03

MODELLO MATEMATICO

PARAMETRI

$N = \{0..n\}$	Insieme dei nodi: 0 indica la scuola, mentre gli altri nodi rappresentano le case dei bambini.
A	Insieme degli archi che rappresentano le connessioni (percorsi più brevi) tra le fermate della linea.
$c_{ij} > 0 \quad \forall (i, j) \in A$	Lunghezza del cammino più breve dal nodo i al nodo j.
$\delta > 1$	Rappresenta il rapporto massimo consentito tra la lunghezza del percorso valutato e la lunghezza del percorso più breve.

VARIABILI

Ogni bambino è rappresentato come un'unità di flusso uscente da un nodo (la sua casa).

$x_{ij} \in Z^+ \quad \forall (i, j) \in A$	Variabile intera che rappresenta il flusso che va dal nodo i al nodo j.
$y_{ij} \in \{0, 1\} \quad \forall (i, j) \in A$	Variabile binaria pari a 1 se viene utilizzato l'arco (i, j), 0 altrimenti.
$z_i \in \{0, 1\} \quad \forall i \in N$	Variabile binaria pari a 1 se il nodo i rappresenta una foglia, 0 altrimenti.
$d_i > 0 \quad \forall i \in N$	Variabile reale associata alla lunghezza dell'itinerario lungo la linea del pedibus da i a 0.

FUNZIONE OBIETTIVO

$$min \sum_{i=0}^{n} z_i$$

Funzione obiettivo: minimizzare il numero di foglie.

VINCOLO SULLA DISTANZA

(1) $d_i \leq \delta c_{i0} \quad \forall i \in N \setminus \{0\}$

La lunghezza dell'itinerario di ogni bambino non deve superare di δ volte il percorso più breve.

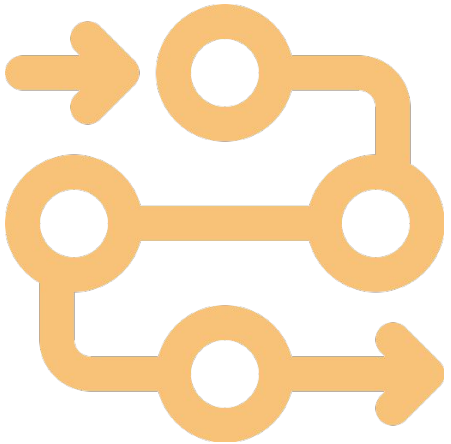
VINCOLI SUL FLUSSO

(2) $$\sum_{(i,j)\in A} x_{ij} = \sum_{(i,j)\in A} x_{ji} + 1 \quad \forall i \in N \setminus \{0\}$$

Per ogni nodo non radice, il flusso uscente è uguale al flusso entrante più 1, ovvero ad ogni nodo corrisponde un solo bambino.

(3) $$\sum_{(i,0)\in A} x_{io} = N - 1$$

I percorsi del pedibus devono servire tutti gli studenti, convergendo a scuola.



VINCOLI SU ARCHI E FLUSSI

(4) $\sum_{(i,j)\in A} y_{ij} = 1 \quad \forall i \in N \setminus \{0\}$

Da ogni nodo non radice deve esserci esattamente un arco che trasporta il flusso in uscita.

(5) $y_{ij} - x_{ij} \leq 0 \quad \forall (i,j) \in A$

Se l'arco (i, j) è attivo, il flusso deve essere positivo (≥ 1).

(6) $x_{ij} - N y_{ij} \leq 0 \quad \forall (i,j) \in A$

Per ogni arco (i, j) nella rete, il flusso attraverso di esso non può essere maggiore del numero di nodi nel grafo.

VINCOLI SULLE FOGLIE

(7) $$z_i + \sum_{(j,i)\in A} y_{ji} \geq 1 \quad \forall i \in N$$

Se il nodo non è una foglia ($z_i = 0$), deve essere attraversato da almeno un arco entrante.

(8) $$\sum_{(j,i)\in A} y_{ji} + N z_i + \leq N \quad \forall i \in N$$

Se il nodo è una foglia ($z_i = 1$), esso non verrà attraversato da nessun arco in ingresso.

04

IMPLEMENTAZIONE

REQUISITI

L’implementazione degli approcci risolutivi è stata creata utilizzando **Python3.11**

Il solver utilizza i seguenti pacchetti aggiuntivi di Python:

- **Amply**: permette di caricare e manipolare dati in formato AMPL come strutture dati Python.
- **NumPy**: consente effettuare operazioni matematiche efficienti su array e matrici, rendendo così la manipolazione dei dati più agevole.
- **NetworkX** & **Matplotlib**: per la creazione e visualizzazione grafica dei risultati.

GENERATORE DI ISTANZE

generator.py

```
# Scelta dei Parametri
n
delta

# Generazione casuale delle coordinate
coordX = random(0,100)
coordY = random(0,100)

# Output su file con sintassi AMPL
with open(f'istanza_{n}.dat', 'w') as
f:
.
.
.
```

```
param n := 3
;

param delta := 1.5
;

param coordX :=
0   98
1   80
2   58
3    8
;

param coordY :=
0   60
1    6
2    3
3   87
;
```

MANIPOLAZIONE DEI DATI

parser.py

`input` ← istanza generata

`return` un **oggetto Amply** contenente i dati del file di input


```
'n' = {float}
'delta' = {float}
'coordX' = {ParamObject}
<ParamObject:{data}>
'coordY' = {ParamObject}
<ParamObject:{data}>
```

graph.py

`input` ← oggetto Amply

Fornisce metodi utili per la manipolazione dei dati, oltre a generare una matrice delle distanze euclidee tra i nodi:

N	0	1	2	3
0	0.0	56.9	69.6	94.0
1	56.9	0.0	22.0	108.4
2	69.6	22.0	0.0	97.8
3	94.0	108.4	97.8	0.0

05

APPROCCI RISOLUTIVI

05a.

GREEDY

GREEDY SEARCH

Gli algoritmi Greedy determinano una soluzione tramite decisioni parziali localmente ottime, senza mai modificare le decisioni prese.

Ecco alcune ragioni per cui lo abbiamo scelto:

- **Semplicità** di implementazione
- **Efficienza:** soluzioni accettabili in tempi rapidi
- **Benchmark** per successive implementazioni più complesse e time consuming

```
procedure Greedy_Search()
    Solution ← ∅;
    Evaluate the incremental costs of each element e ∈ E;
    while Solution is not a complete solution do
        Select the element s ∈ E with the smallest incremental
cost;
        Solution ← Solution ∪ {s};
        Update the incremental costs;
    end;
    return Solution;
end Greedy_Search.
```

IMPLEMENTAZIONE

```
def greedy_search():
  previous = ROOT
  leaves_counter = 1

  while not_visited:
    next_node ← find_next()

    if next_node is feasible:
     not_visited.remove(next_node)
     ...
     previous = next_node

    else:
     ...
     previous = ROOT
     leaves_counter += 1

  return leaves_counter
```

- Inizia a visitare il grafo dal nodo radice
- Individua il nodo migliore da aggiungere alla soluzione parziale tra quelli non ancora visitati
- Se viene individuato un nodo ammissibile, è aggiunto all'attuale percorso, poi rimosso dalla lista dei nodi non visitati
- Se nessun nodo è più ammissibile nel path attuale significa che il nodo precedente era una foglia
- Ripete il ciclo finchè ogni nodo non viene inserito in un percorso

FIND_NEXT

Questa funzione individua il prossimo nodo ottimale da aggiungere alla soluzione, evitando nodi già visitati e che non rispettano il vincolo δ .

```
def find_next(prev_node, not_visited, path_length):
  for node_x in not_visited:
    new_path_length = path_length + get_distance(prev_node, node_x)
    node_min_distance = root_distance(node_x)

      if new_path_length < node_min_distance * delta:
        if new_path_length < current_min_path_length:
          best_node = node_x
          current_min_path_length = new_path_length

  return best_node
```

05b.

GRASP

GRASP

La Greedy Randomized Adaptive Search Procedure è una euristica in cui ad ogni iterazione avvengono due fasi:

1. **Construction**: viene costruita una soluzione ammissibile;
2. **Local Search**: si costruisce un vicinato di soluzioni che viene poi analizzato, selezionando la migliore soluzione tra quelle vicine alla soluzione corrente.

```
procedure GRASP(Max Iterations,Seed)
    Read Input();
    for k = 1, . . . , Max Iterations do
        Solution ← Greedy_Randomized_Construction(Seed);
        Solution ← Local_Search(Solution);
        Update Solution(Solution,Best Solution);
    end;
    return Best Solution;
end GRASP.
```

MOTIVAZIONI

Greedy Search e **GRASP** presentano differenze significative:

- **Approccio risolutivo**: deterministico Vs probabilistico
- **Esplorazione delle soluzioni**: intensificazione Vs diversificazione
- **Complessità computazionale**: bassa Vs alta
- **Adattabilità**: mediocre Vs ottima
- **Qualità delle soluzioni**: stabile Vs variabile

Tuttavia, entrambe condividono l'aspetto Greedy, con le dovute differenze:

- **Riutilizzo del codice**

“ALMOST” GRASP

Durante la realizzazione della GRASP abbiamo riscontrato come già da una implementazione parziale, in cui viene omessa la fase di Local Search, si potessero ottenere risultati migliori rispetto all’euristica Greedy.

Abbiamo quindi deciso di rinominare tale strategia come “Almost” GRASP e di utilizzarla come ulteriore elemento di benchmark.

```
def almost_grasp(iterations, seed):

    for i in range(iterations):
        score ← greedy_randomized(seed)
        if score < best_score:
            best_score = score

    return best_score
```

GREEDY_RANDOMIZED

```
def greedy_randomized(seed):
  previous = ROOT
  leaves_counter = 1

  while not_visited:
    next_node ← rcl(seed)

    if next_node is feasible:
     not_visited.remove(next_node)
     ...
     previous = next_node

    else:
     ...
     previous = ROOT
     leaves_counter += 1

  return score
```

L'algoritmo è molto simile a quello che esegue una Greedy Search.

Le uniche differenze sono:

- **Parametro `seed`**
- **Funzione `rcl`**
- **Restituzione di uno `score`**

RCL

La funzione `rcl(seed)` popola una Restricted Candidate List (**RCL**) formata dagli elementi la cui aggiunta alla soluzione parziale corrente comporta i risultati migliori.

L'elemento da inserire nella soluzione parziale viene selezionato casualmente tra quelli presenti nella RCL (questo è l'aspetto probabilistico dell'euristica).

```
def rcl(seed, prev_node, not_visited, path_length):
  Candidate_list = []
    for node_x in not_visited:
      # Genero new_path_length e node_min_distance

      if new_path_length < node_min_distance * delta:
        Candidate_list.append(node_x)

        if new_path_length < current_min_path_length:
          curr_min_path_length = new_path_length
          best_distance_node = get_distance(prev_node, node_x)
```

```
    if candidate_list:
      restricted_candidate_list = []
      for node_x in candidate_list:
        new_node_distance = get_distance(previous_node, node_x)
        if new_node_distance <= best_distance_node * seed:
          restricted_candidate_list.append(node_x)

  return restricted_candidate_list[random]
```

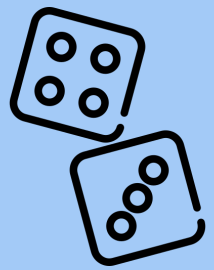
Per generare la `restricted_candidate_list` si selezionano solo i nodi aventi una distanza dal nodo precedente ≤ alla distanza tra il miglior nodo candidato (quello più vicino al nodo precedente tra quelli non ancora visitati) moltiplicata per un `seed`.

Viene restituito un nodo casuale tra quelli presenti nella RCL.

SEED

Rappresenta il fattore di randomizzazione dell'algoritmo. Può essere visto come un margine di confidenza per la generazione della RCL.

- **seed = 1** → Greedy Search
- **seed >> 1** → candidate_list == RCL



SCORE

Elemento utilizzato per valutare la bontà di una soluzione.

Esso è composto da due variabili:

- **Numero di percorsi**
- **Lunghezza totale dei percorsi in una soluzione**

Valutazione di una soluzione candidata:

```
if leaves <= best_leaves:
  if leaves < best_leaves  or
     length < best_length:
       best_leaves = leaves
       best_length = length
```

GRASP COMPLETA

Esegue un ciclo per **iterations** volte che parte da una soluzione generata da un algoritmo greedy e la migliora utilizzando una local search. Infine prende la soluzione migliore generata dopo tutti i cicli e la restituisce.

```
def grasp(iterations, seed):
  for i in range(iterations):
    solution, score ← greedy_randomized(seed)
    solution, score ← local_search(solution, score)

    if score < best_score:
      best_score = score

  return best_score
```

LOCAL SEARCH FIRST IMPROVEMENT

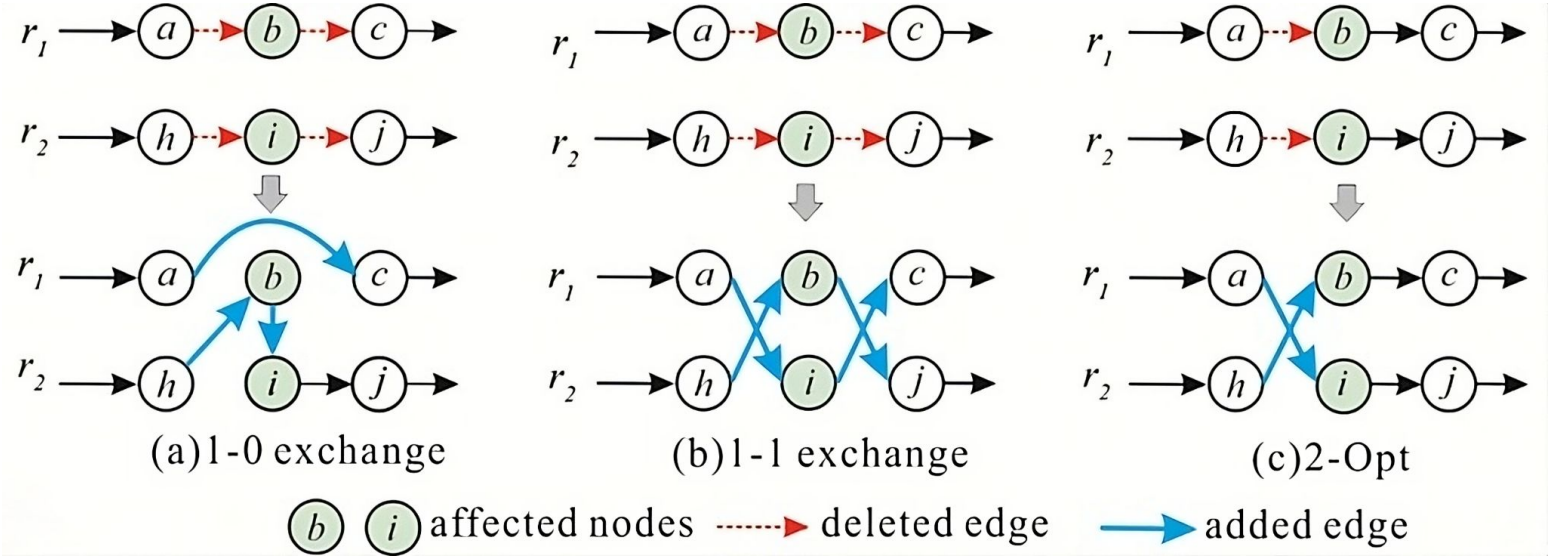
Parte da una soluzione iniziale, genera un intorno di soluzioni fino a che non ne trova una migliore.

```
procedure Local_Search(Solution)
    while better Solution is not found do
        Find s' ∈ N(Solution) with f(s') < f(Solution);
        Solution ← s;
    end;
    return Solution;
end Local_Search.
```

IMPLEMENTAZIONE

```
def local_search(initial_edges, initial_score, iterations):
    best_edges = initial_edges
    best_score = initial_score

    for i in range(iterations):
        new_paths, new_edges =  one_to_one_exchange(best_edges)
        new_score =  evaluate_solution(new_paths, new_edges)

        if new_score < best_score:
            best_score = new_score
            break

    return best_score
```

ONE_TO_ONE_EXCHANGE



ONE_TO_ONE_EXCHANGE

```
def one_to_one_exchange(current_edges):
    new_edges = current_edges
    new_paths = arch_to_path(new_edges)

    path1 = random.choice(new_paths)
    new_paths.remove(path1)
    path2 = random.choice(new_paths)
    new_paths.remove(path2)

    arc1 = random.choice(path1)
    arc2 = random.choice(path2)

    vertex1 = 0
    vertex2 = 0
    while vertex1 == 0:
        vertex1 = random.choice(arc1)
    while vertex2 == 0:
        vertex2 = random.choice(arc2)
```

Trasforma la lista di archi in una lista di percorsi ognuno con i suo archi

Scegli casualmente due percorsi diversi

Scegli casualmente due archi per ogni percorso

Scegli casualmente un nodo da ciascuna sottolista (evitando il nodo 0)

ONE_TO_ONE_EXCHANGE

```
...
    for arc in path1:
        if vertex1 in arc:
            arc[arc.index(vertex1)] =
vertex2
    for arc in path2:
        if vertex2 in arc:
            arc[arc.index(vertex2)] =
vertex1

    new_paths.append(path1)
    new_paths.append(path2)

    new_edges = path_to_arch(new_paths)

    return new_paths, new_edges
```

Sostituisci tutte le occorrenze di vertex1 in path1 con vertex2 e viceversa

Inserisce i due nuovi percorsi con nodo scambiato nella lista di percorsi

Converti una lista di percorsi in una lista di archi

ARCH_TO_PATH

```
def arch_to_path(edges):
    paths = []
    path = []

    for edge in edges:
        if edge[0] == 0:
            if path:
                paths.append(path)
            path = [edge]
        else:
            path.append(edge)

    if path:
        paths.append(path)

    return path
```

```
edges = [[0, 10], [10, 4],
[4, 2], [0, 8], [8, 9], [0,
3], [3, 5], [5, 7], [0, 1],
[1, 6]]

Produce:

path=[[[0, 10], [10, 4], [4,
2]], [[0, 8], [8, 9]], [[0,
3], [3, 5], [5, 7]], [[0,
1], [1, 6]]]
```

PATH_TO_ARCH

```
def path_to_arch(paths):
    edges = []

    for path in paths:
        edges.extend(path)

    return edges
```

path=[**[**[0, 10], [10, 4], [4, 2]**]**, **[**[0, 8], [8, 9]**]**, **[**[0, 3], [3, 5], [5, 7]**]**, **[**[0, 1], [1, 6]**]**]

Produce:

edges = [[0, 10], [10, 4], [4, 2], [0, 8], [8, 9], [0, 3], [3, 5], [5, 7], [0, 1], [1, 6]]

EVALUATE_SOLUTION

```
def evaluate_solution(paths, edges):
  total_length = 0
  leaves_counter = 0
  for path in paths:
    if is_feasible(path):
      continue
    else:
      return float("inf")

  for edge in edges:
    total_length +=
    get_edge_distance(edge)
      for node in edge:
        if node == ROOT:
          leaves_counter += 1

  return score
```

Richiama la funzione is_feasible che verifica che il vincolo sulla distanza sia rispettato

Nel caso in cui sia rispettato il vincolo valuta il numero di foglie e la lunghezza

EVALUATE_FEASIBILITY

```
def is_feasible(path):
  curr_path_length = 0

  for edge in path:
    curr_path_length +=
    get_edge_distance(edge)

    if curr_path_length >
       get_root_distance(edge[1])*delta
:
       return False

  return True
```

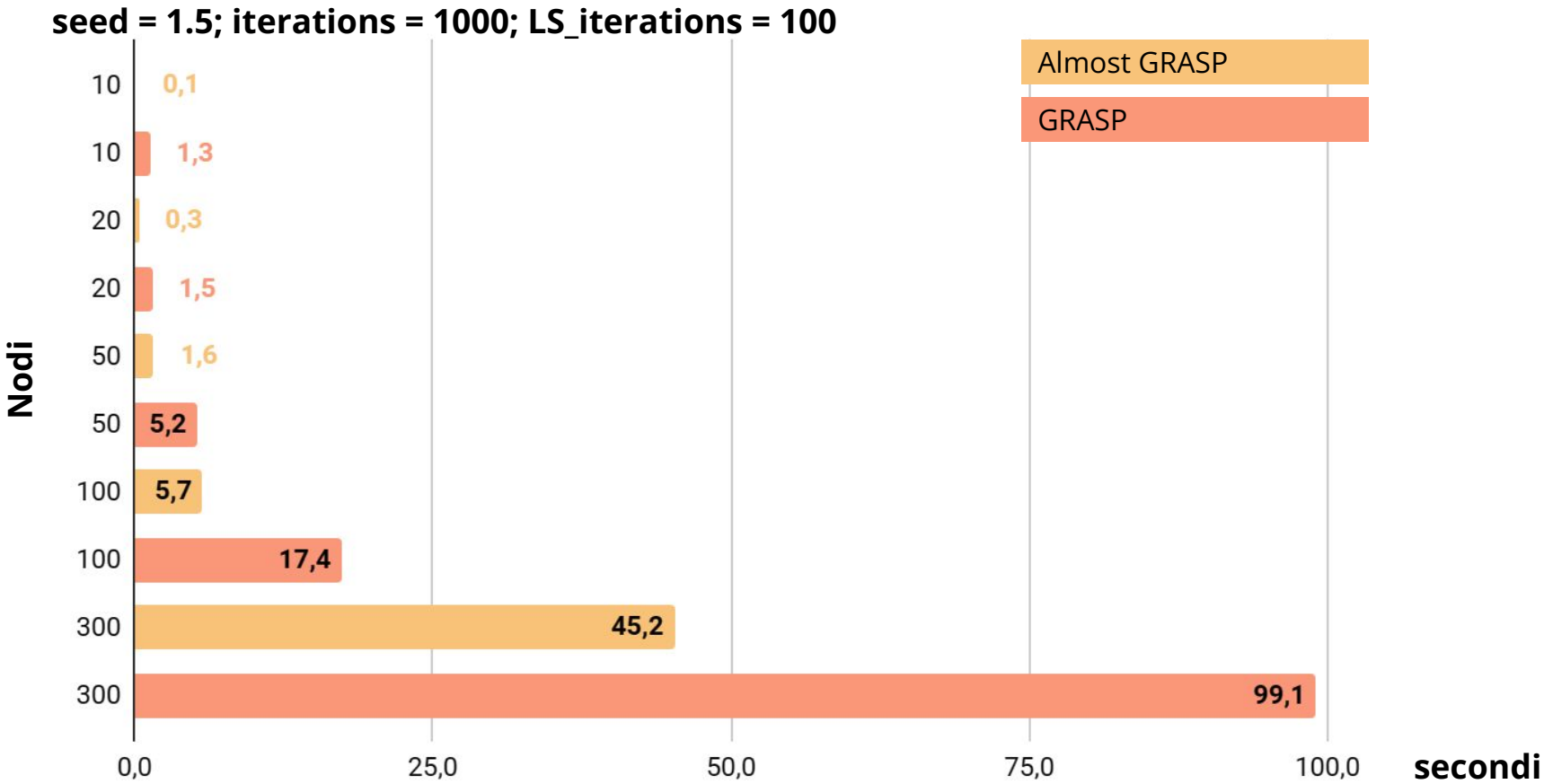
Verifica che il vincolo sulla distanza sia rispettato

06

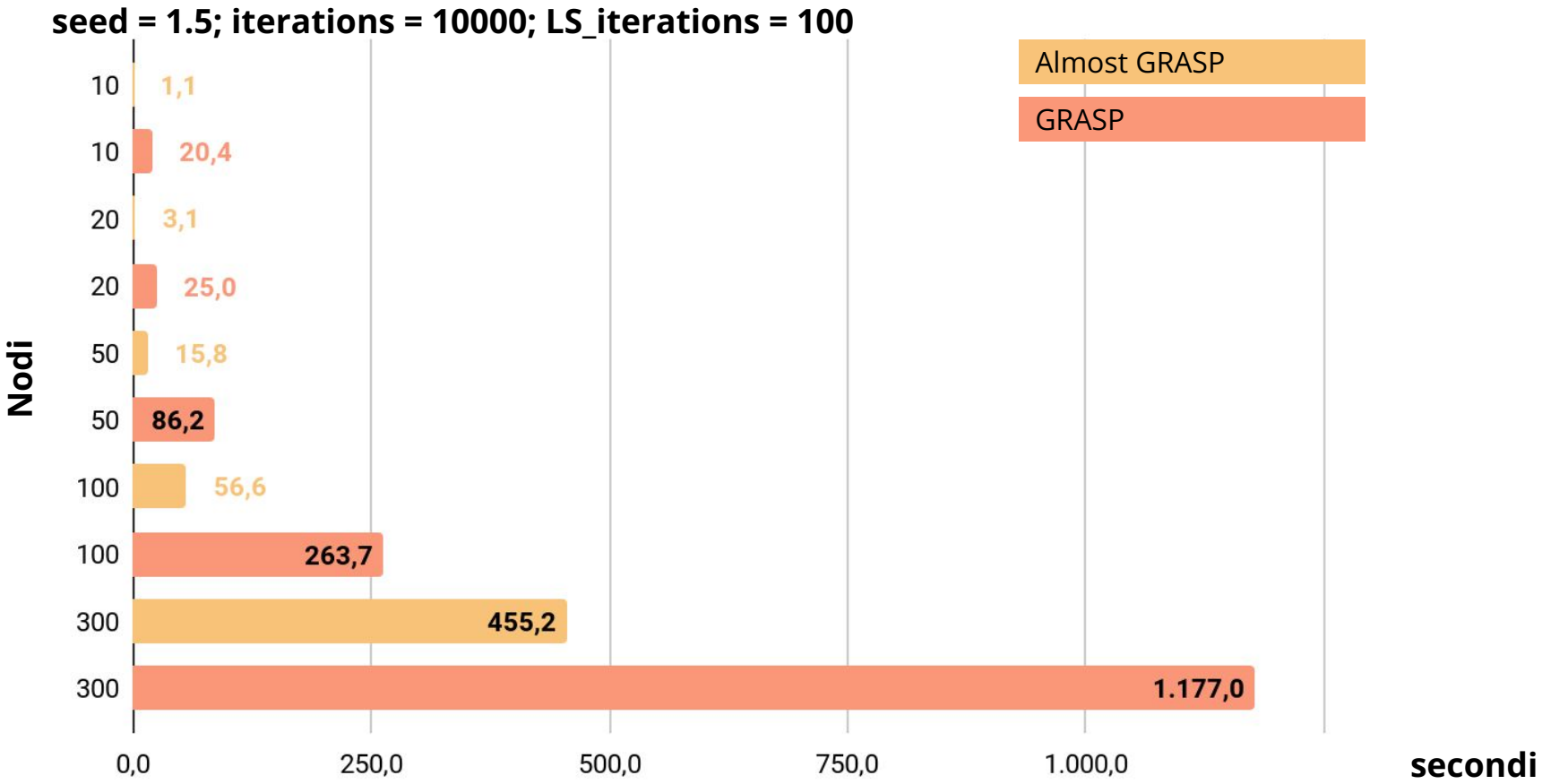
TEST E RISULTATI

SCELTA DEL NUMERO DI ITERAZIONI

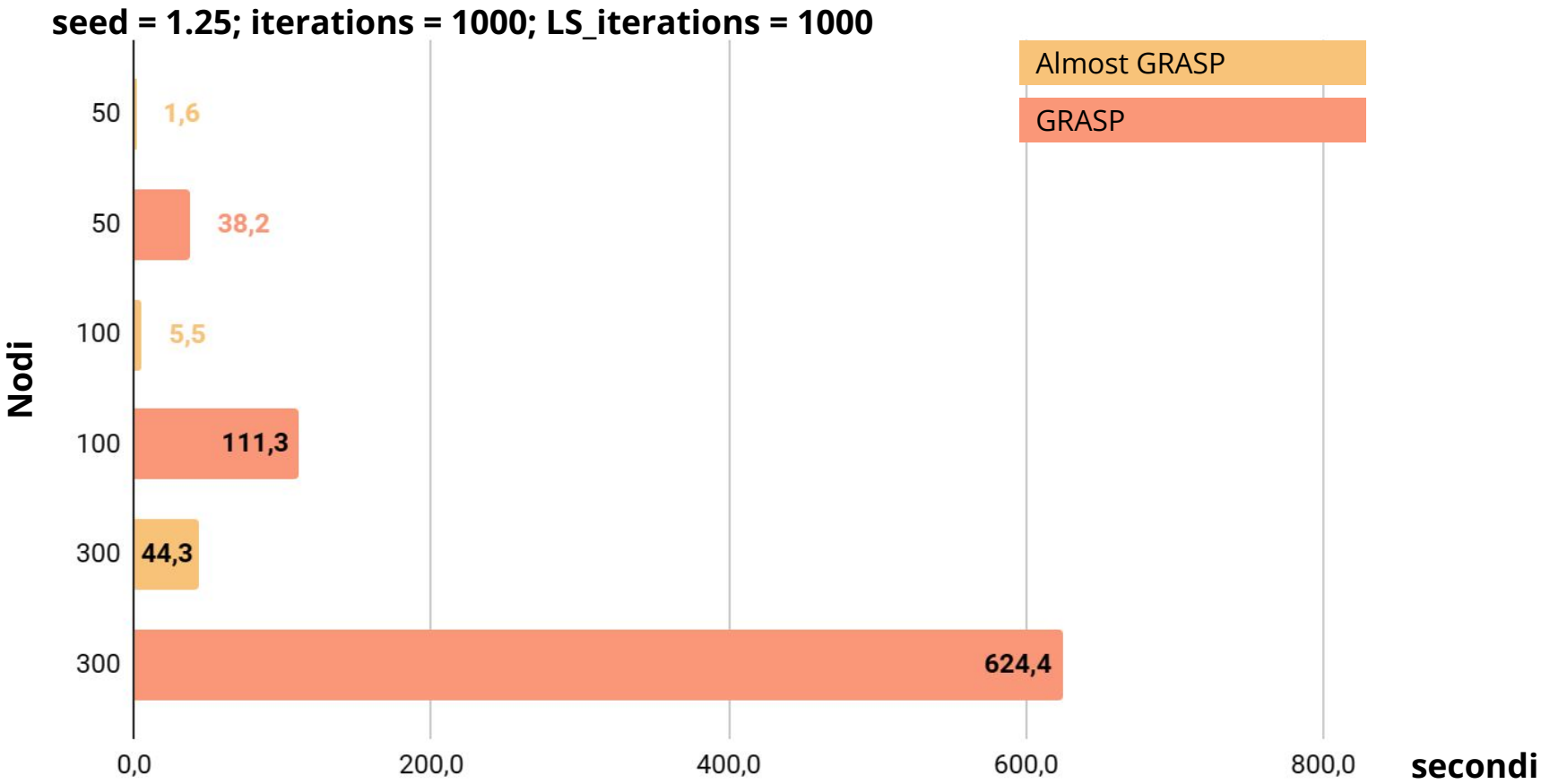
TEMPI DI ESECUZIONE



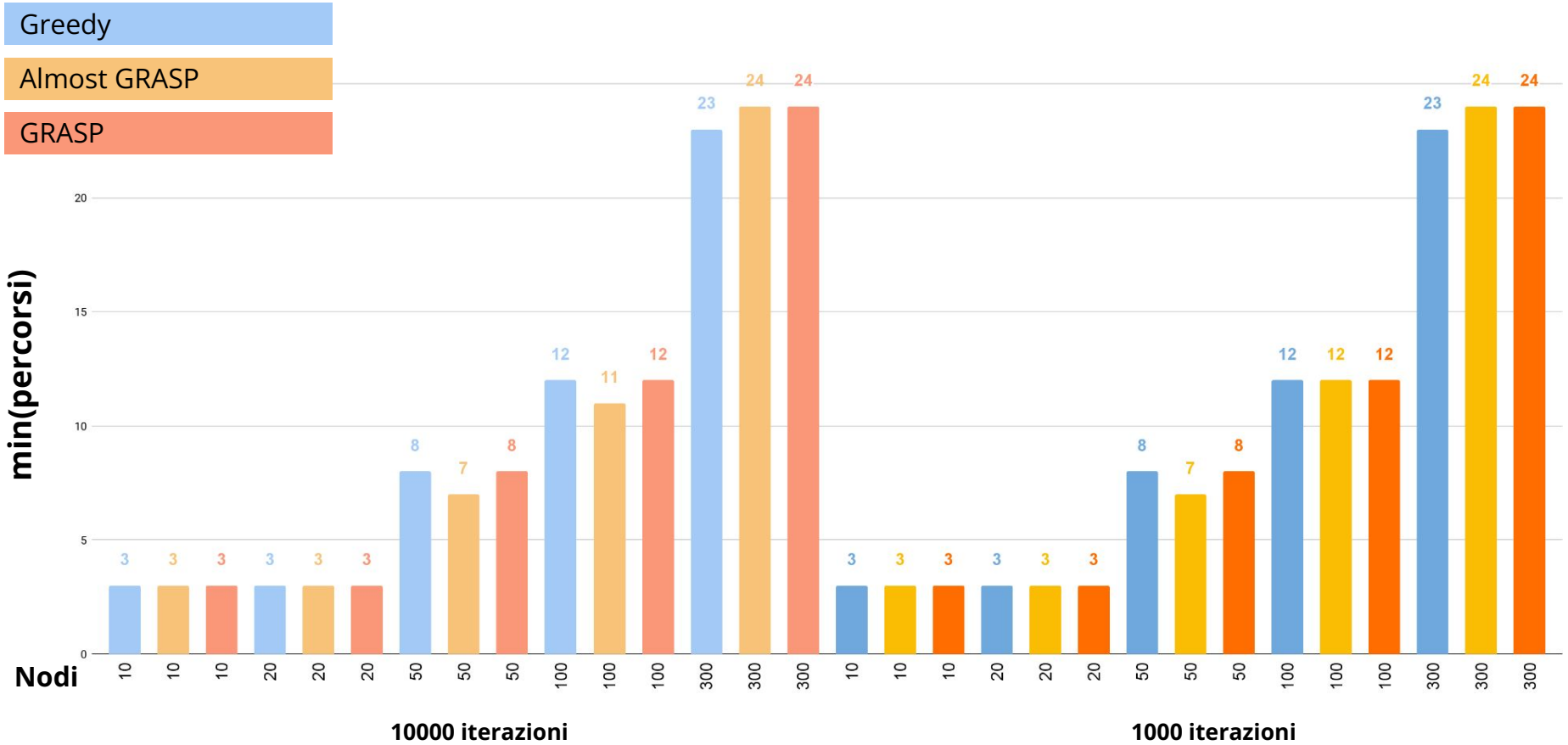
TEMPI DI ESECUZIONE



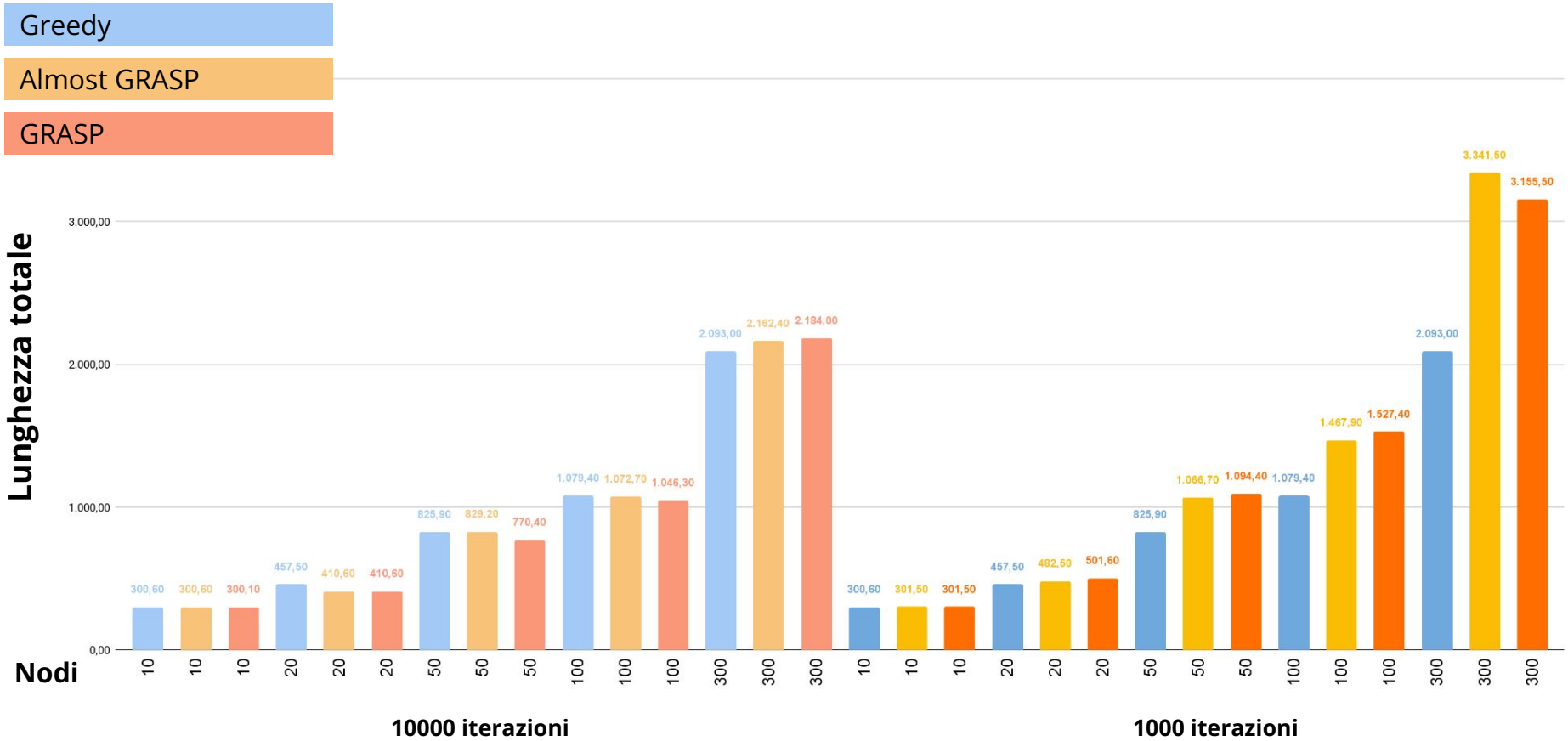
TEMPI DI ESECUZIONE



PERCORSI AL VARIARE DELLE ITERAZIONI



LUNGHEZZE AL VARIARE DELLE ITERAZIONI



SCELTA DEL SEED

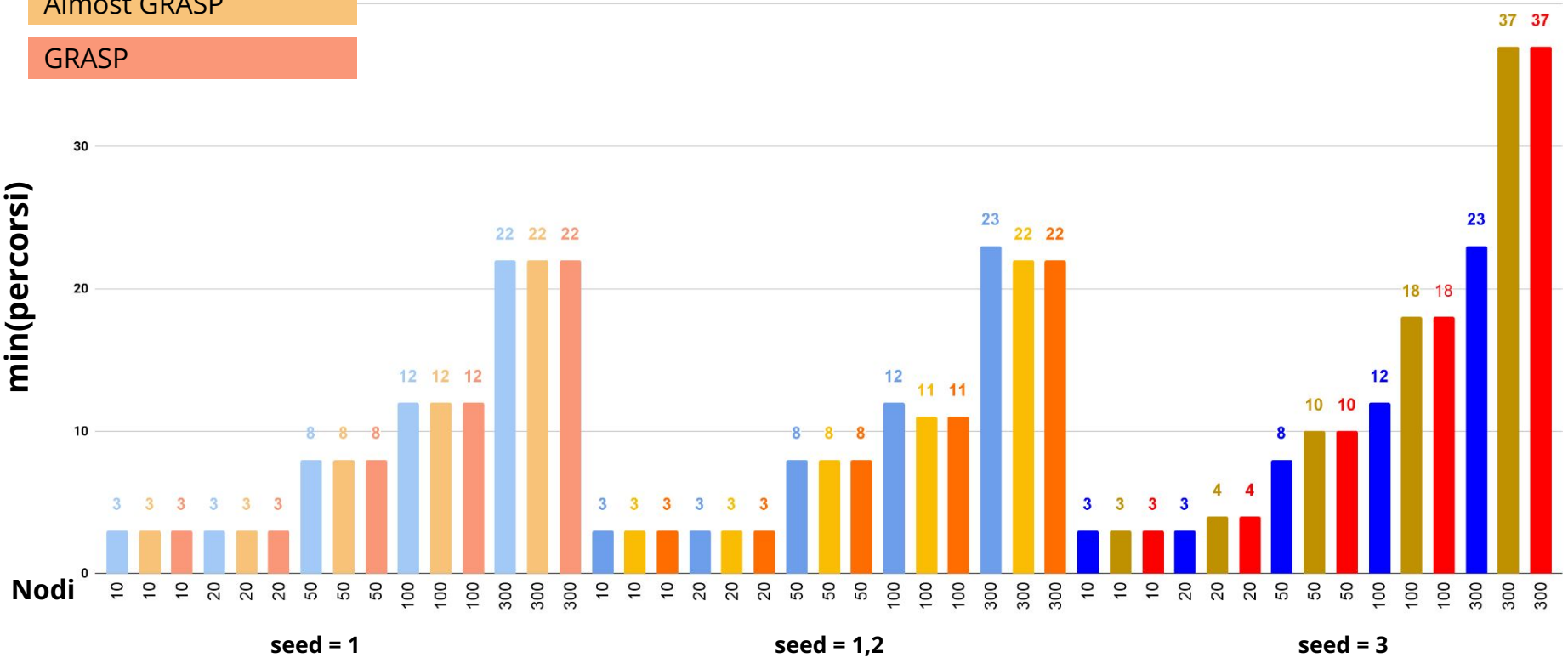
NUMERO DI PERCORSI AL VARIARE DEL SEED

Greedy

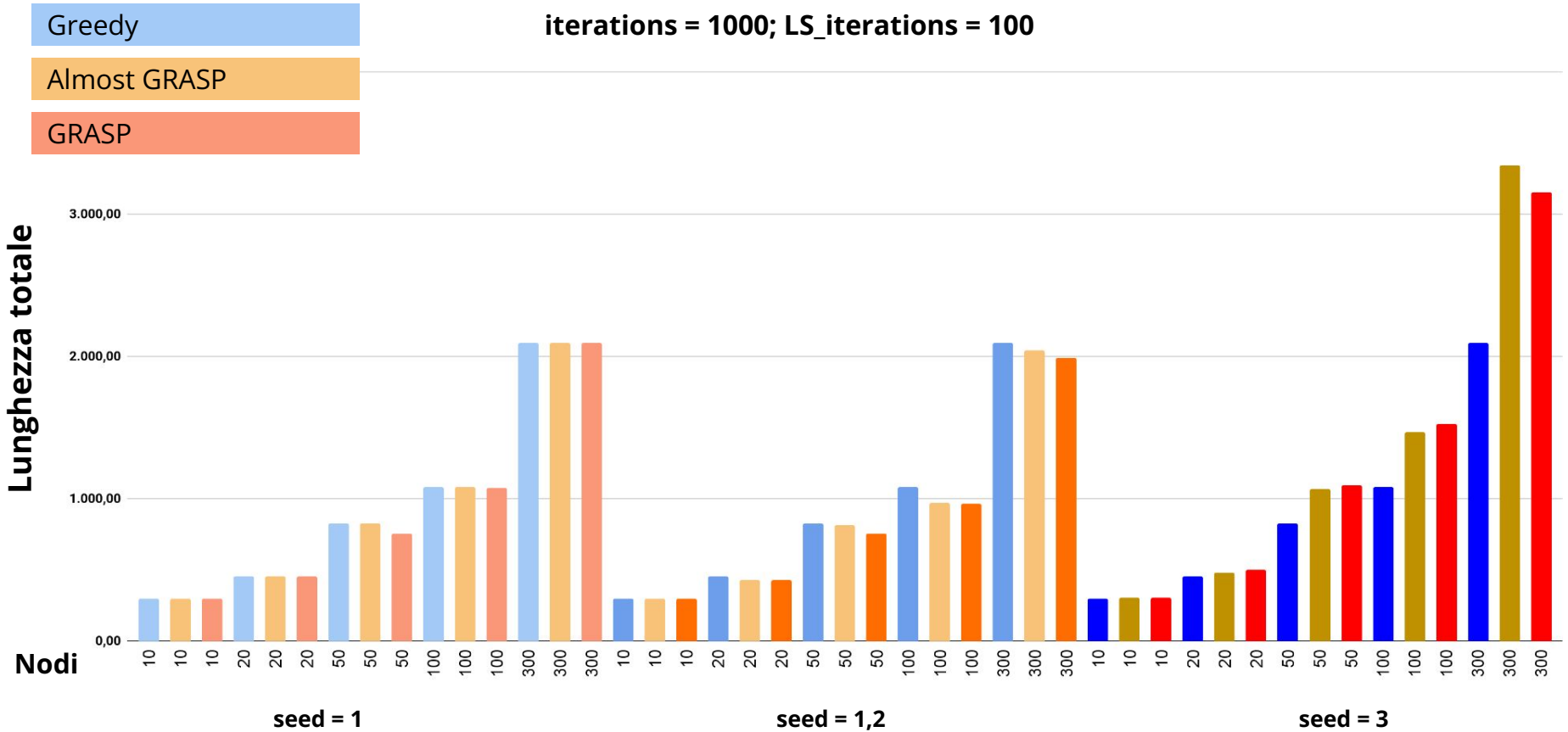
Almost GRASP

GRASP

iterations = 1000; LS_iterations = 100



LUNGHEZZA DELLA SOLUZIONE AL VARIARE DEI SEED



SCELTA FINALE DEI PARAMETRI

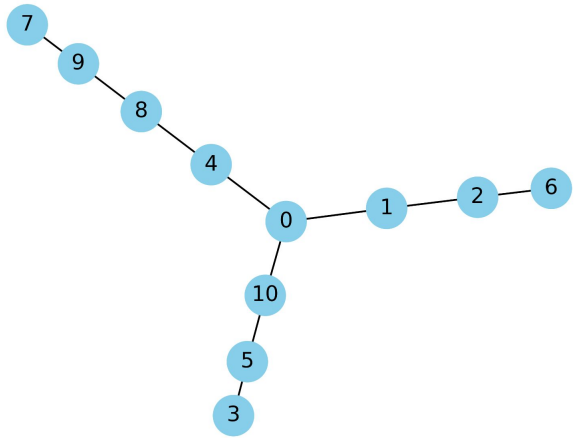
```
if  nodi < 30:
    seed = 2
    iterations = 1000
else:
    seed = 1.25
    iterations = 1000

local_search_iterations = 1000
```

RISULTATI

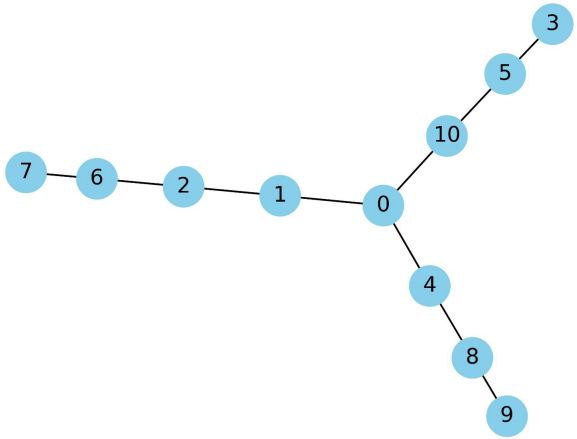
Nodi	Algoritmo	Numero minimo percorsi	Lunghezza tot. percorsi	T. esecuzione (secondi)
10	Greedy Search	**3**	300,60	-
	Almost GRASP	**3**	300,10	0,1
	GRASP	**3**	300,10	10,8
20	Greedy Search	**3**	457,50	-
	Almost GRASP	**3**	454,50	0,3
	GRASP	**3**	433,20	3,9
50	Greedy Search	**8**	825,90	-
	Almost GRASP	**7**	817,80	1,6
	GRASP	**7**	800,10	38,2
100	Greedy Search	**12**	1.079,40	-
	Almost GRASP	**11**	964,00	5,5
	GRASP	**11**	958,40	111,3
300	Greedy Search	**23**	2.093,00	-
	Almost GRASP	**22**	2.070,50	44,3
	GRASP	**22**	1.992,40	624,4

RISULTATI CON 10 NODI

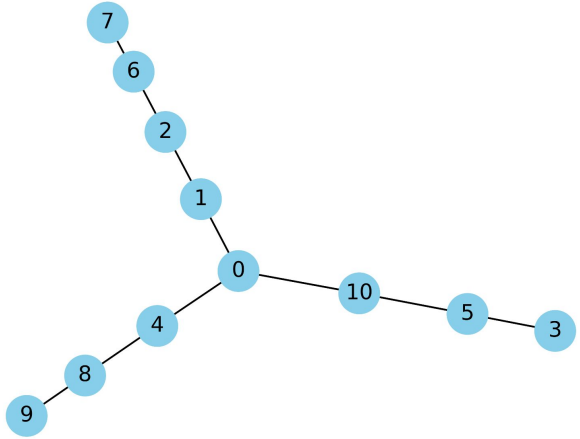
Greedy Search
delta = 1.5



Almost GRASP
delta = 1.5
Seed = 2

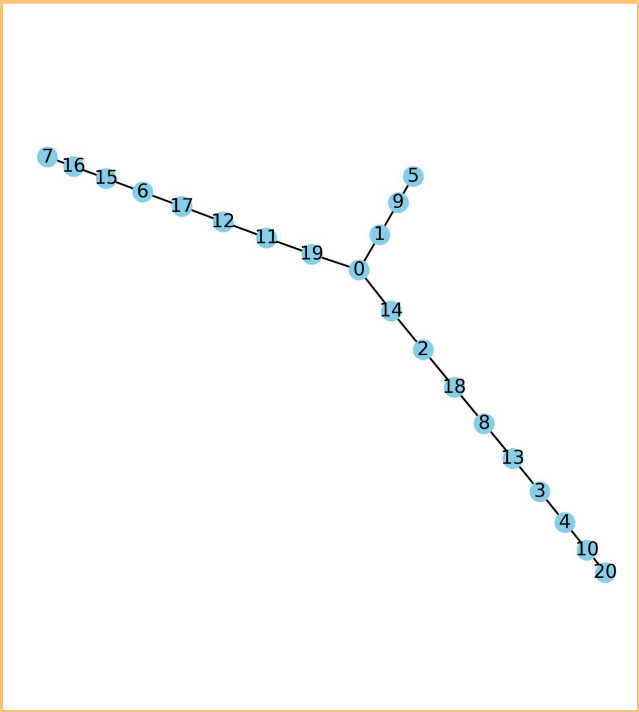


GRASP
delta = 1.5
Seed = 2

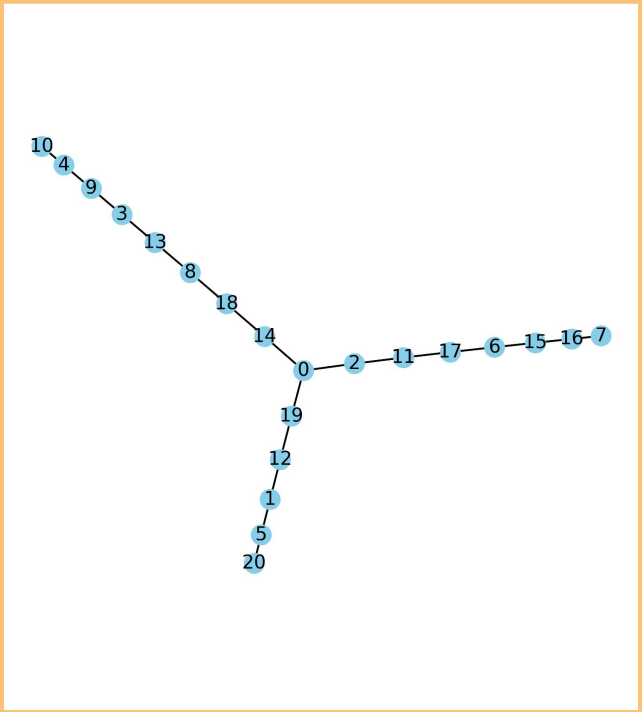


RISULTATI CON 20 NODI

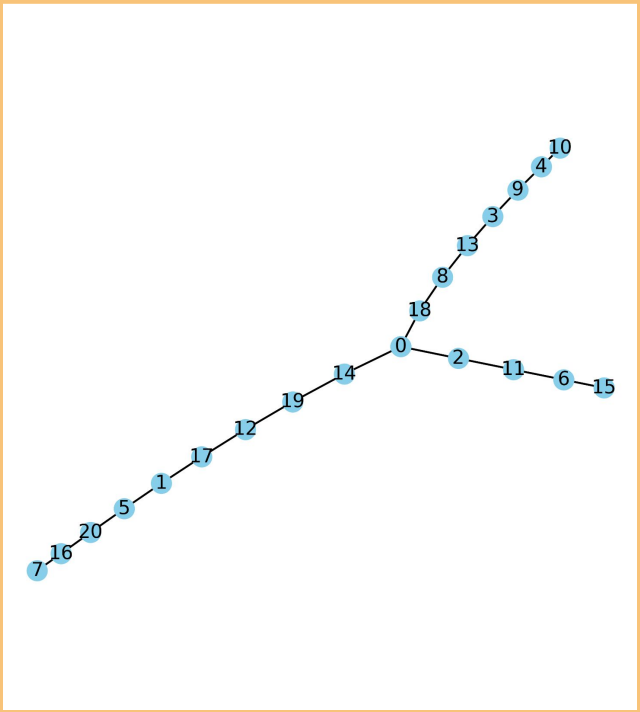
Greedy Search
delta = 1.5



Almost GRASP
delta = 1.5
Seed = 2

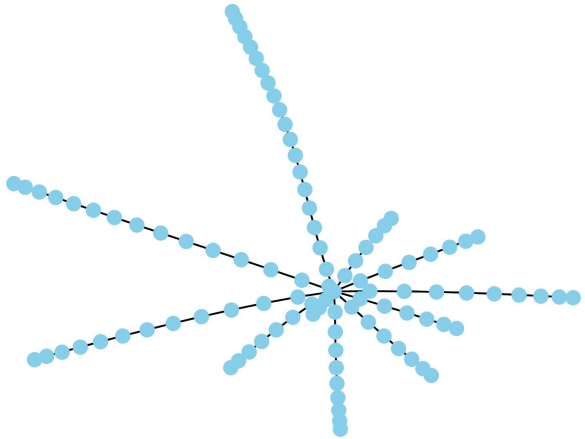


GRASP
delta = 1.5
Seed = 2

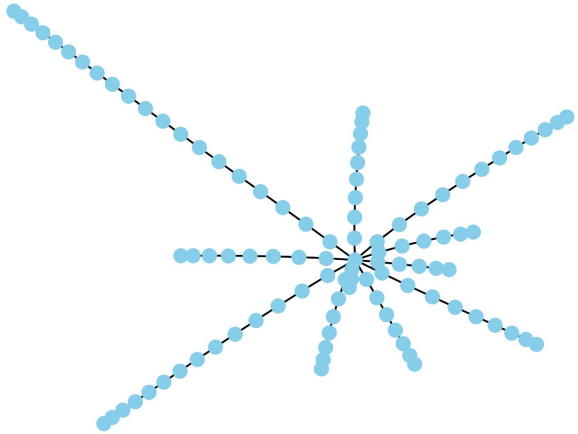


RISULTATI CON 100 NODI

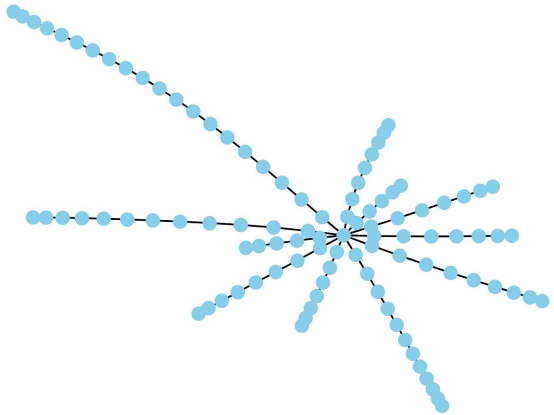
Greedy Search
delta = 1.5



Almost GRASP
delta = 1.5
Seed = 1.25



GRASP
delta = 1.5
Seed = 1.25



Fonts & colors used

This presentation has been made using the following fonts:

Oswald
(https://fonts.google.com/specimen/Oswald)

Open Sans
(https://fonts.google.com/specimen/Open+Sans)

#a3caf7 #eb573d #5fa4ab #f8c379 #fa9778 #051934