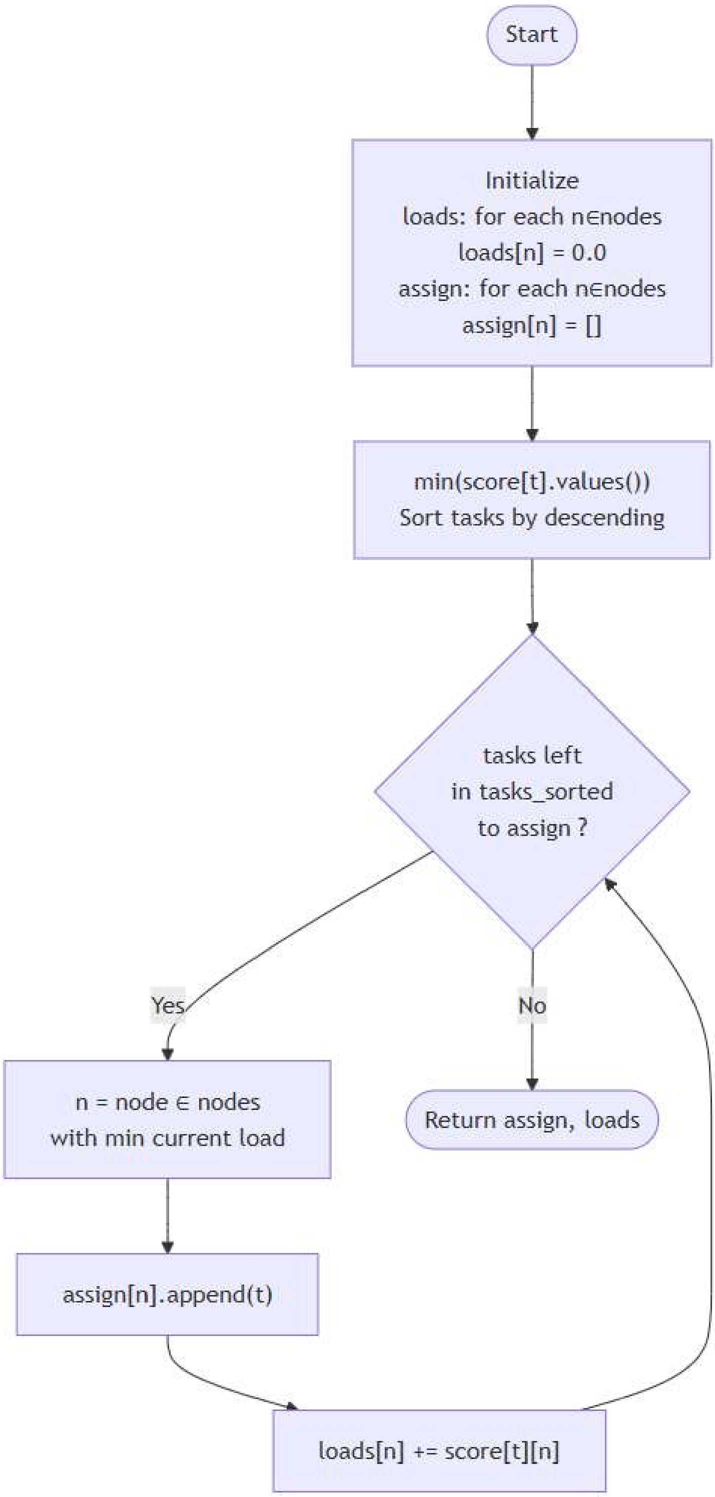
Algorithm1: LPT-Seed Greedy Scheduler

```
Input:
  T = {$t_1$, $t_2$, …, $t_m$}      // set of m tasks
  N = {$n_1$, $n_2$, …, $n_k$}      // set of k nodes
  S[t][n]                  // execution time (or cost) of task t on node n

Output:
  A[n]                     // for each node n, the ordered list of tasks assigned to it
  L[n]                     // for each node n, the total load after assignment

Procedure LPT_Seed(T, N, S):

  // 1) Initialize loads and assignment lists
  for each node n in N do
      L[n] ← 0.0
      A[n] ← []       // empty list
  end for

  // 2) Precompute each task's "best possible" (minimum) time
  Define bestTime(t) = min { S[t][n] : n ∈ N }

  // 3) Sort tasks by descending bestTime → largest first
  T_sorted ← sort T by key bestTime(t), in descending order

  // 4) Greedily assign each task to the least-loaded node
  for each task t in T_sorted do
      // find the node with current smallest load
      n* ← arg min { L[n] : n ∈ N }

      // assign task t to node n*
      append t to A[n*]

      // update that node's load by the actual execution time on n*
      L[n*] ← L[n*] + S[t][n*]
  end for

  // 5) Return the final assignment and loads
  return A, L

End Procedure
```


Start
Initialize
loads: for each n∈nodes
loads[n] = 0.0
assign: for each n∈nodes
assign[n] = []
min(score[t].values())
Sort tasks by descending
tasks left
in tasks_sorted
to assign ?
Yes
No
n = node ∈ nodes
with min current load
Return assign, loads
assign[n].append(t)
loads[n] += score[t][n]

Algorithm2: Local Search Load Balancer

Input:

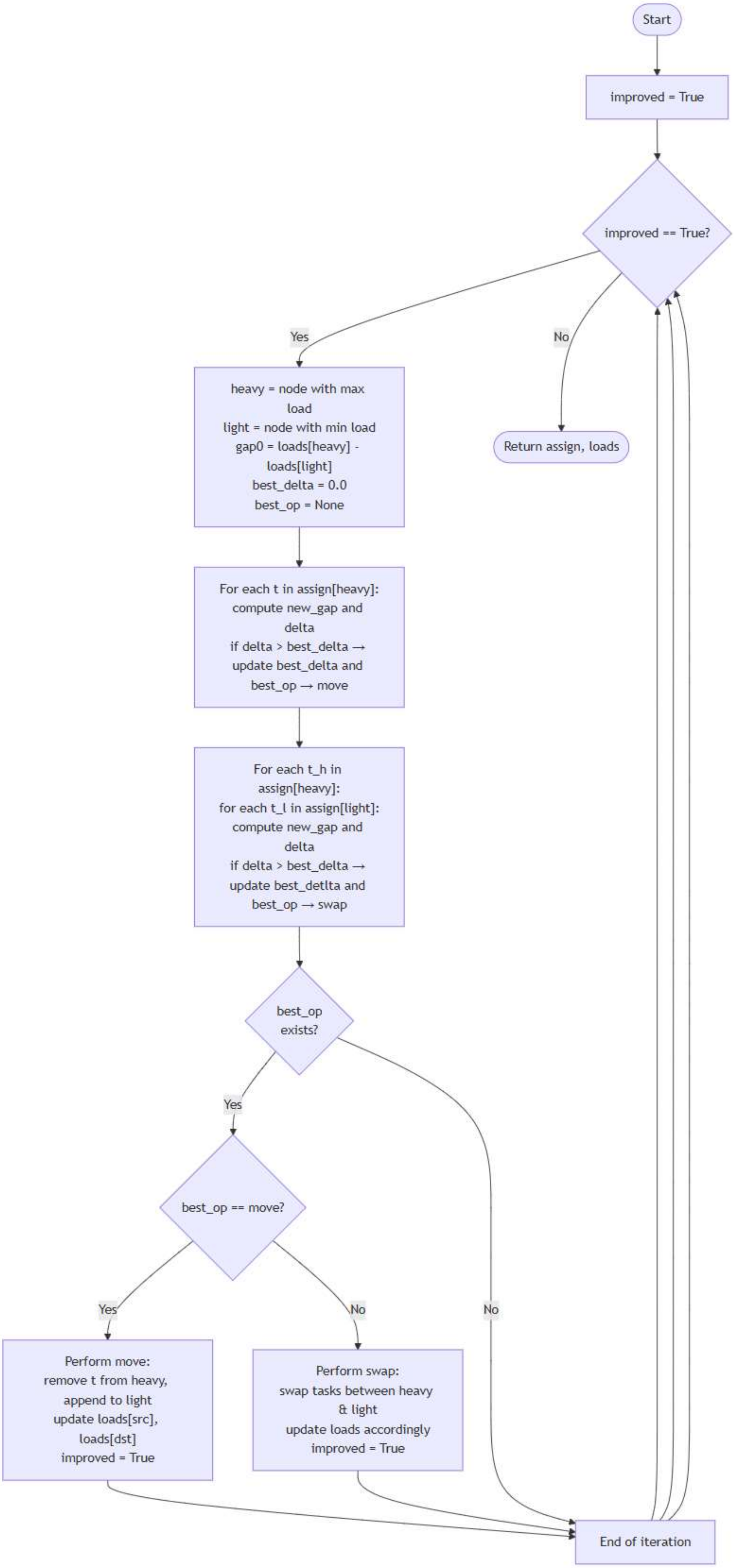
- A ⟵ initial assignment, a map from each node n ∈ N to a list of tasks
- L ⟵ initial loads, a map from each node n ∈ N to its total load
- S ⟵ score matrix, where S[t][n] is the execution time of task t on node n
- N ⟵ set of all nodes

Output:

- (A, L) ⟵ an improved assignment and corresponding loads

```
Procedure LOCAL_SEARCH(A, L, S, N)
improved ← true
while improved do
  improved ← false

  ⟶ Identify the two extreme nodes
  heavy  ← $\arg\max_{n \in N}$ L[n]      ▻ node with largest load
  light  ← $\arg\min_{n \in N}$ L[n]      ▻ node with smallest load
  $gap_0$  ← L[heavy] − L[light]

  best_gain ← 0.0
  best_op   ← None

  ⟶ Phase 1: single-task moves from heavy → light
  for each task t in A[heavy] do
    new_load_heavy ← L[heavy] − S[t][heavy]
    new_load_light ← L[light] + S[t][light]
    other_loads    ← { L[n] : n ∈ N \ {heavy, light} }
    gap_new        ← max(new_load_heavy, new_load_light, max(other_loads))
                    − min(new_load_heavy, new_load_light, min(other_loads))
    gain           ← $gap_0$ − gap_new
    if gain > best_gain then
      best_gain ← gain
      best_op   ← (“move”, t)
    end if
  end for

  ⟶ Phase 2: two-task swaps between heavy and light
  for each t_h in A[heavy] do
    for each t_l in A[light] do
      new_load_heavy ← L[heavy] − S[t_h][heavy] + S[t_l][heavy]
      new_load_light ← L[light] − S[t_l][light] + S[t_h][light]
      gap_new        ← max(new_load_heavy, new_load_light, max(other_loads))
                      − min(new_load_heavy, new_load_light, min(other_loads))
      gain           ← $gap_0$ − gap_new
      if gain > best_gain then
```

```
        best_gain ← gain
        best_op   ← (“swap”, t_h, t_l)
      end if
    end for
  end for

  ⟶ Apply the best local improvement, if any
  if best_op ≠ None then
    improved ← true
    if best_op[0] = “move” then
      t ← best_op[1]
      remove t from A[heavy]
      append t to A[light]
      L[heavy] ← L[heavy] − S[t][heavy]
      L[light] ← L[light] + S[t][light]
    else  ▻ swap
      (t_h, t_l) ← (best_op[1], best_op[2])
      replace t_h with t_l in A[heavy]
      replace t_l with t_h in A[light]
      L[heavy] ← L[heavy] − S[t_h][heavy] + S[t_l][heavy]
      L[light] ← L[light] − S[t_l][light] + S[t_h][light]
    end if
  end if

end while

return (A, L)
End Procedure
```


Start
improved = True
improved == True?
Yes
No
heavy = node with max load
light = node with min load
gap0 = loads[heavy] - loads[light]
best_delta = 0.0
best_op = None
Return assign, loads
For each t in assign[heavy]:
compute new_gap and delta
if delta > best_delta → update best_delta and best_op → move
For each t_h in assign[heavy]:
for each t_l in assign[light]:
compute new_gap and delta
if delta > best_delta → update best_detlta and best_op → swap
best_op exists?
Yes
best_op == move?
Yes
No
No
Perform move:
remove t from heavy, append to light
update loads[src], loads[dst]
improved = True
Perform swap:
swap tasks between heavy & light
update loads accordingly
improved = True
End of iteration

Algorithm 3: Multi-Start Load Balancer

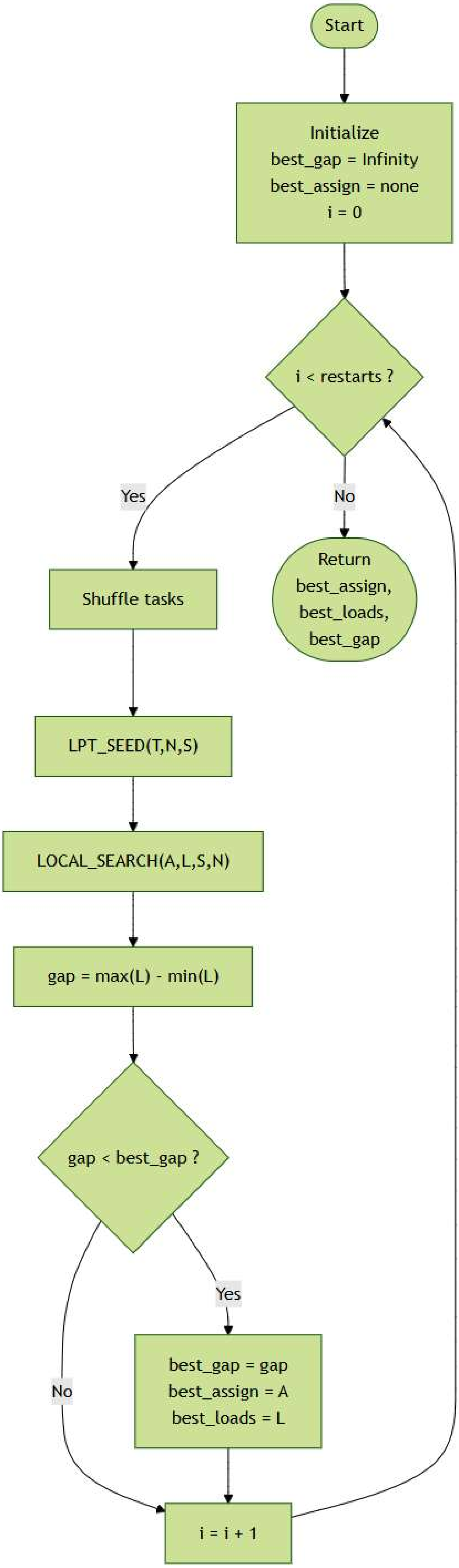
Input:

- T ← list of tasks
- N ← set of nodes
- S ← score matrix S[t][n] = cost of running task t on node n
- restarts ← number of random restarts (default 20)
- seed ← RNG seed for reproducibility (default 42)

Output:

- A_best ← assignment map node → list of tasks with smallest gap
- L_best ← corresponding node loads
- gap_best ← load imbalance gap = max(L_best) − min(L_best)

```
Procedure BALANCE(T, N, S, restarts, seed)
Initialize RNG with seed
best_gap  ← +∞
A_best    ← None
L_best    ← None

for i from 1 to restarts do
  shuffle(T)                          ▻ randomize task order
  (A, L) ← LPT_SEED(T, N, S)          ▻ initial greedy assignment
  (A, L) ← LOCAL_SEARCH(A, L, S, N)   ▻ refine via local moves/swaps
  gap   ← $\max\{ L[n] : n \in N \}$
          $- \min\{ L[n] : n \in N \}$    ▻ compute current imbalance
  if gap < best_gap then
    best_gap  ← gap
    A_best    ← copy of A
    L_best    ← copy of L
  end if
end for

return (A_best, L_best, best_gap)
End Procedure
```


Start
Initialize
best_gap = Infinity
best_assign = none
i = 0
i < restarts ?
Yes
No
Shuffle tasks
Return
best_assign,
best_loads,
best_gap
LPT_SEED(T,N,S)
LOCAL_SEARCH(A,L,S,N)
gap = max(L) - min(L)
gap < best_gap ?
Yes
No
best_gap = gap
best_assign = A
best_loads = L
i = i + 1

All in one

```
Procedure BALANCE(T, N, S, restarts, seed):
  Initialize random(seed)
  best_gap ← +∞
  best_A   ← None
  best_L   ← None

  for i = 1 to restarts do
    shuffle(T)

    // Stage 1: LPT Seed
    (A, L) ← LPT_SEED(T, N, S)

    // Stage 2: Local Search
    (A, L) ← LOCAL_SEARCH(A, L, S, N)

    // Evaluate
    gap ← max { L[n] } – min { L[n] }
    if gap < best_gap then
      best_gap ← gap
      best_A   ← deepcopy(A)
      best_L   ← deepcopy(L)
    end if
  end for

  return (best_A, best_L, best_gap)
End Procedure
```

```
Procedure LPT_SEED(T, N, S):
  for each n in N:
    load[n] ← 0
    assign[n] ← empty list
  end for

  // sort tasks by descending "best-case" time
  T_sorted ← sort T by key t ↦ min { S[t][n] for n in N }, descending

  for each t in T_sorted do
    n* ← arg min { load[n] for n in N }
    append t to assign[n*]
    load[n*] ← load[n*] + S[t][n*]
  end for
  return (assign, load)
End Procedure



Procedure LOCAL_SEARCH(A, L, S, N):
  repeat
    improved ← false

    heavy ← arg max { L[n] }
    light ← arg min { L[n] }
    gap0  ← L[heavy] – L[light]
    best_gain ← 0
    best_op   ← None

    // Try moving any t from heavy → light
    for t in A[heavy]:
      compute gain if moved
      record if gain > best_gain
    end for

    // Try swapping any t_h↔t_l between heavy/light
    for t_h in A[heavy]:
      for t_l in A[light]:
        compute gain if swapped
        record if gain > best_gain
      end for
    end for

    if best_op exists:
      apply best_op to (A, L)
      improved ← true
    end if
  until not improved

  return (A, L)
End Procedure
```