Partial Ordering ; Dependency Graph ; Scheduling

My Lecture Notes – Fall '18 edited for Spring '20 Class

Partial Ordering is a Relation
Dependency Graph is a Directed Graph
Scheduling is a Table

partial ordering

Let's use β as a symbol for a **Relation**

Def. A Relation β is a **Partial Ordering Relation** iff the following 3 conditions hold:

(1) irreflexive (anti-reflexive) — our definition deviates from the standard definition of Partial Order Relation
a β̸ a where a is an object (integer, person, noun) — e.g. a is NOT before to a
e.g. if a,b,c are integers and **β is <** : 5 ≮ 5 ; is true ✔ (5 is NOT less than 5)

(2) antisymmetric
a β b ⟹ b β̸ a — e.g. if a is before b, then b can NOT be before a
5 < 10 then 10 ≮ 5 ; is true ✔

(3) transitive
a β b and b β c ⟹ a β c — e.g. if a is before b and b is before c, then a is before c
5 < 10 and 10 < 15 then 5 < 15 ✔

An example of a reflexive relation is the relation "is equal to" on the set of real numbers, since every real number is equal to itself.

mnemonic: 2 negatives 1 positive

example

- if β = 'in front of' , then β is a Partial Ordering Relation b/c all 3 hold e.g. a is NOT in front of a
- if β = 'behind' , then β is a Partial Ordering Relation b/c all 3 hold e.g. a is NOT behind a is true
- if β = 'parent' , then β is NOT a Partial Ordering Relation b/c in 3rd 'a' is NOT a parent of 'c' ('a' is 'c's grandparent)
- if β = '<' , then β is a Partial Ordering Relation b/c all 3 hold e.g. 5 not < 5 is true
- if β = '>' , then β is a Partial Ordering Relation b/c all 3 hold e.g. 5 not > 5 is true
- if β = '>=' , then β is NOT a Partial Ordering Relation because 5 not >= 5 it false
- if β = '=' , then β is NOT a Partial Ordering Relation b/c 5≠5 is false

on exam — entire lecture

dependency graph

Partial Ordering is designed by you. Partial Ordering is based on a lot of specific information e.g. start time, end time, amount of items (jobs), travel time/distance, etc… Imagine building a house: foundations before walls, walls before windows and doors, 1st floor before 2nd floor etc…

(1) x = 10
(2) y = 15
(3) z = 20
(4) w = x+x
(5) u = y*y
(6) v = x+y+z
(7) q = w*w   ← job id

given a list of tasks to be performed → make it into **Partial Ordering** →

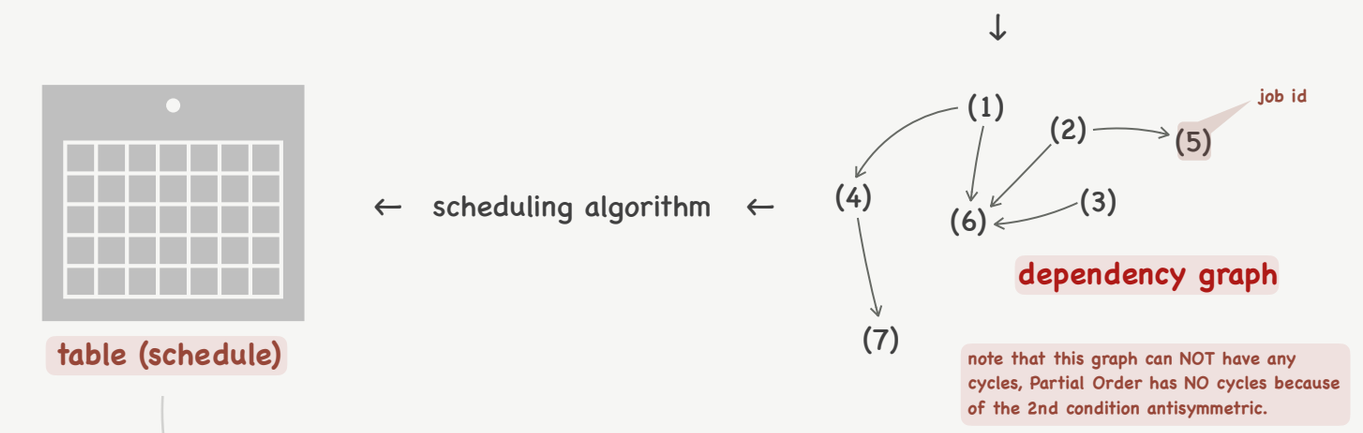
Here, Partial Ordering is based on the precedence of execution i.e. job (4) depends on job (1), hence, job (1) must be scheduled and completed before job (4) gets started.

(1) before (4) (i.e. job (1) before job (4))
(2) before (5)
(1) before (6)
(2) before (6)
(3) before (6)
(4) before (7)

→

(1) → (4)
(2) → (5)
(1) → (6)
(2) → (6)
(3) → (6)
(4) → (7)

construct a directed graph



note that this graph can NOT have any cycles, Partial Order has NO cycles because of the 2nd condition antisymmetric.
if (1) before (1) was allowed, it would form a cycle

job start time, job end time, job length (e.g. here jobLength = 1 unit of time), worker id, job id

time:	0	1	2	3	4
P_1	1	_	_	_	
P_2	2	_	_	_	
P_3	3	_	_	_	
P_4	5	_	_	_	
P_5	8	_	_	_	

scheduling (ALL)

Algorithm — Algorithm steps for Scheduling

- step0: initializations
  G ← graph (given)
  if (unlimited) proc ← ∅ // unlimited number of processors
  if (limited) proc ← ask how many // limited number of processors
  time ← 0 // we start at zero
- step1: node ← get unmarked orphan node from graph G // node = job
- step2: mark node // don't get the same node twice
- step3: insert node into Linked-List using Insertion Sort (ascending) — in our project → descending by #of dependents
- step4: repeat step 1 to 3 until no more unmarked orphan nodes in G
- step5: find an available processor // worker that performs job
  if(no processors available)
   if (unlimited) request a new processor // since unlimited
   if (limited) wait for next time slice // since limited
- step6: remove head from Linked-List & place it into available processor
  if (variable job time) fill the remaining time slices
- step7: repeat steps 5 to 6 until Linked-List is empty (or NO proc available)
- step8: time++
- step9: node ← find a finished job i.e. processor is idle
- step10: delete node & its outgoing edges from graph G
- step11: repeat step 9 to 10 until no more idle processors
- step12: repeat step 1 to 11 until G is empty

place nodes into LList (Graph → LList)
place nodes into Table (LList → Table)
delete finished nodes from G (Table → Graph) node = job

scheduling

There are **4 options** to do scheduling:
(1) Limited Processors , all jobs take same time (e.g. same number of clock cycles, minutes, days, weeks, …)
most realistic → (2) Limited Processors , jobs take variable time
(3) Unlimited Processors , all jobs take same time
(4) Unlimited Processors , jobs take variable time

Advantages of Scheduling: scheduling is done ahead of time

1. We know exactly how many processors (workers) we'll need in order to complete the entire project.
2. We know exactly how long it will take to complete the entire project.
3. We know exactly how many processors (workers) will be required for each **time slice**.

this 'time slice' e.g. 2nd week requires only a single processor (worker)

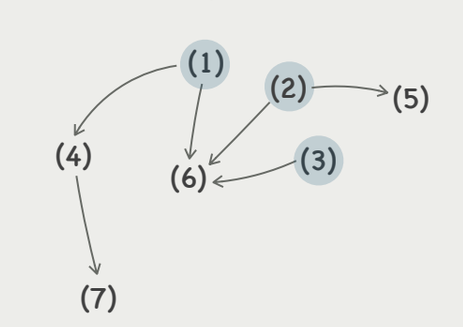
time:	0	1	2	3	4
P_1	1	4	6	9	
P_2	2	_	7	_	
P_3	3	_	_	_	
P_4	5	_	_	_	
P_5	8	_	_	_	

Scheduling is ideal when we have a set of tasks or jobs that are to be performed in a specific sequence. I.e., certain tasks cannot be performed before others are finished. Scheduling allows us to plan ahead and prepare a possible schedule according to which we know how long a particular project will take and how many workers it will require (for each time slice).

scheduling (1)

scheduling algorithm steps (informal)
(1) Limited Processors , all jobs take same time

Step1: identify the orphan nodes (nodes are jobs)



Step2: insert orphan nodes into Linked-List in ascending order
OPEN → 1 → 2 → 3 → null

Step3: find available time slot in table and insert Linked-List head into it

time:	0	1
P_1	1	
P_2	2	

limit of processors is 2 we get this in step 0
all jobs take the same amount of time, here both job 1 and job 2 take 1 unit of time
fixed time

Step3.1 repeat this until either no more slots or no more nodes

Step4: increment time : time++

Step5: delete node (job) 1 and 2 and their edges

Step6: repeat steps 1 to 5 until graph is empty

final schedule:

time:	0	1	2	3	4
P_1	1	3	5	7	
P_2	2	4	6	_	

Note that: if there is a mistake in the Partial Ordering i.e. there is a cycle, the graph G will never be empty. Consequently, also note that, if there are more nodes but none of them are orphans, the graph will never be empty.

While inserting nodes (jobs) from the Graph into the LinkedList, (in our project) we performed insertion in descending order by the number of dependents. The second insertion option would be in ascending order by the job duration.

scheduling (1)

Algorithm — Algorithm steps for Scheduling (limited processors ; same job time)

- step0: initializations
  G ← graph (given)
  proc ← ask how many // since processor number is limited
  time ← 0 // we start at zero
- step1: node ← get an unmarked orphan node from graph G //node=job
- step2: mark node // b/c we don't want to get same node twice
- step3: insert node into Linked-List using Insertion Sort (ascending)
- step4: repeat step 1 to 3 until no more unmarked orphan nodes in G
- step5: find an available processor // worker that performs job
  if(no processors available)
   wait for next time slice // since limited
- step6: remove head from Linked-List & place it into available processor
- step7: repeat steps 5 to 6 until Linked-List is empty (or NO proc available)
- step8: time++
- step9: node ← find a finished job i.e. processor is idle
- step10: delete node & its outgoing edges from graph G
- step11: repeat step 9 to 10 until no more idle processors
- step12: repeat step 1 to 11 until G is empty

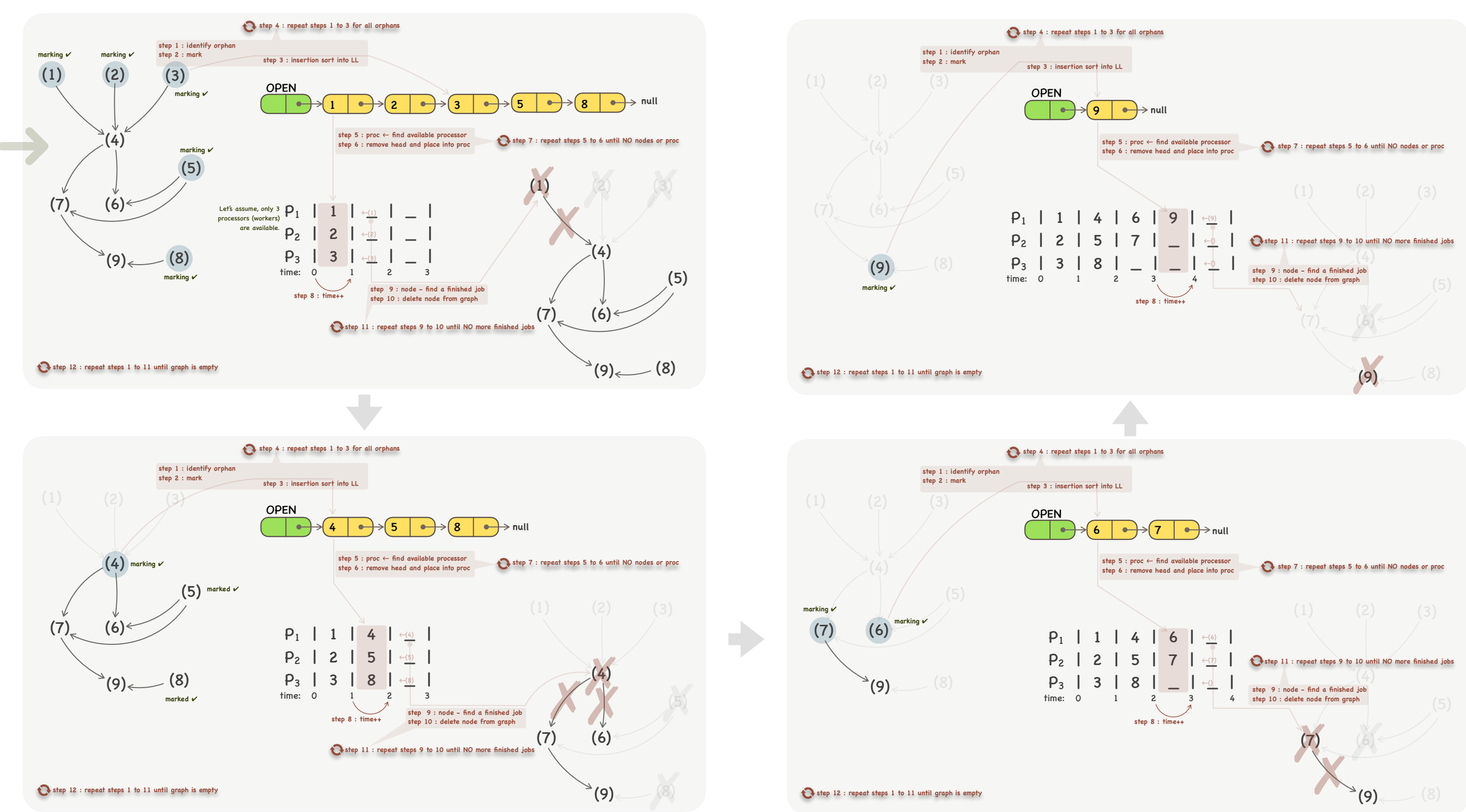
limited proc — same job time

schedule:

P_1	1	4	6	9	_
P_2	2	5	7	_	_
P_3	3	8	_	_	_
time:	0	1	2	3	4

Logistics
E.g. The entire project will take 4 weeks. In the 1st two weeks, we will need all 3 workers. In the 3rd week, we will only need 2 workers. In week 4 we will only need 1 worker.

(1) → (4)
(2) → (4)
(3) → (4)
(4) → (6)
(5) → (6)
(4) → (7)
(5) → (7)
(7) → (9)
(8) → (9)



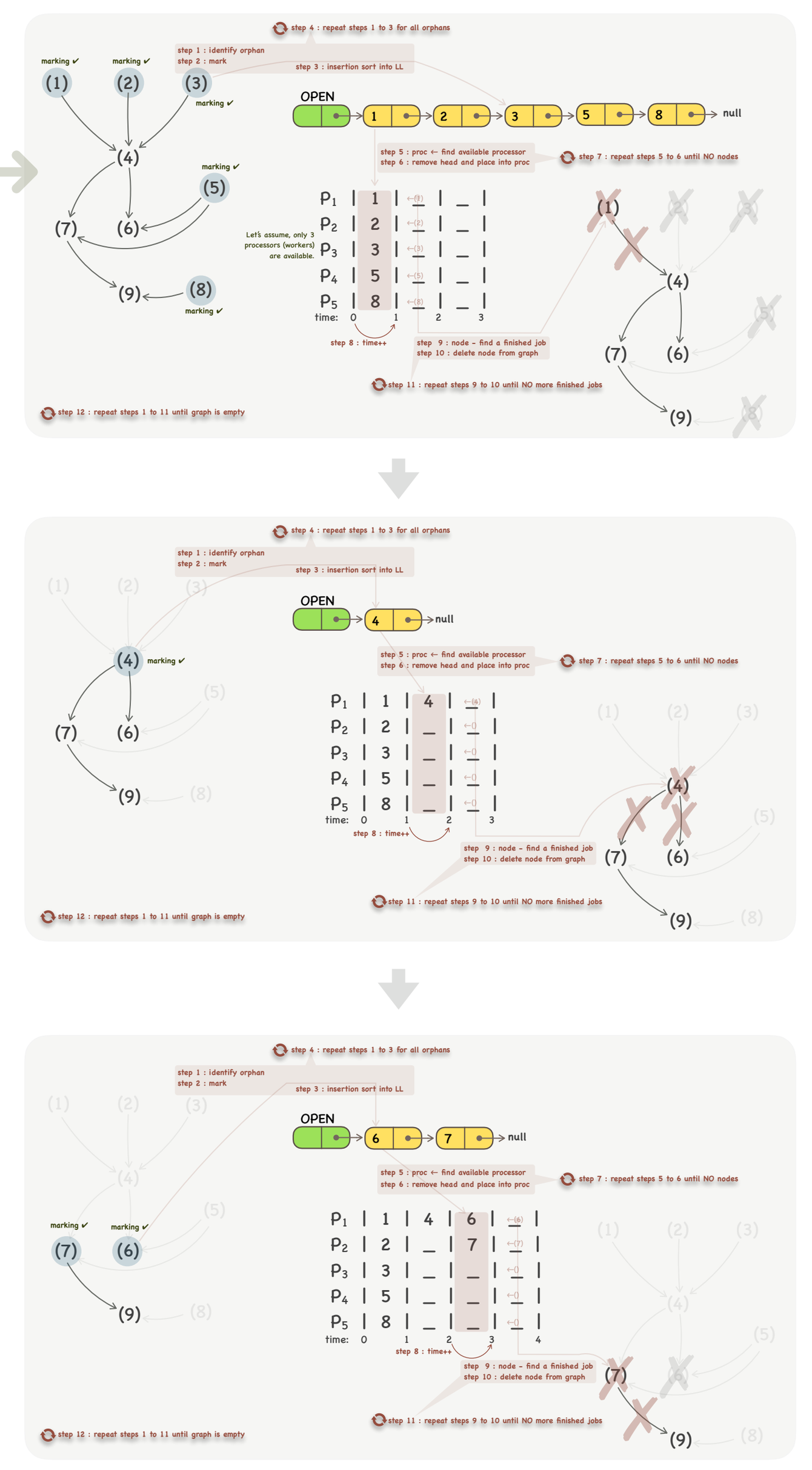
scheduling (3)

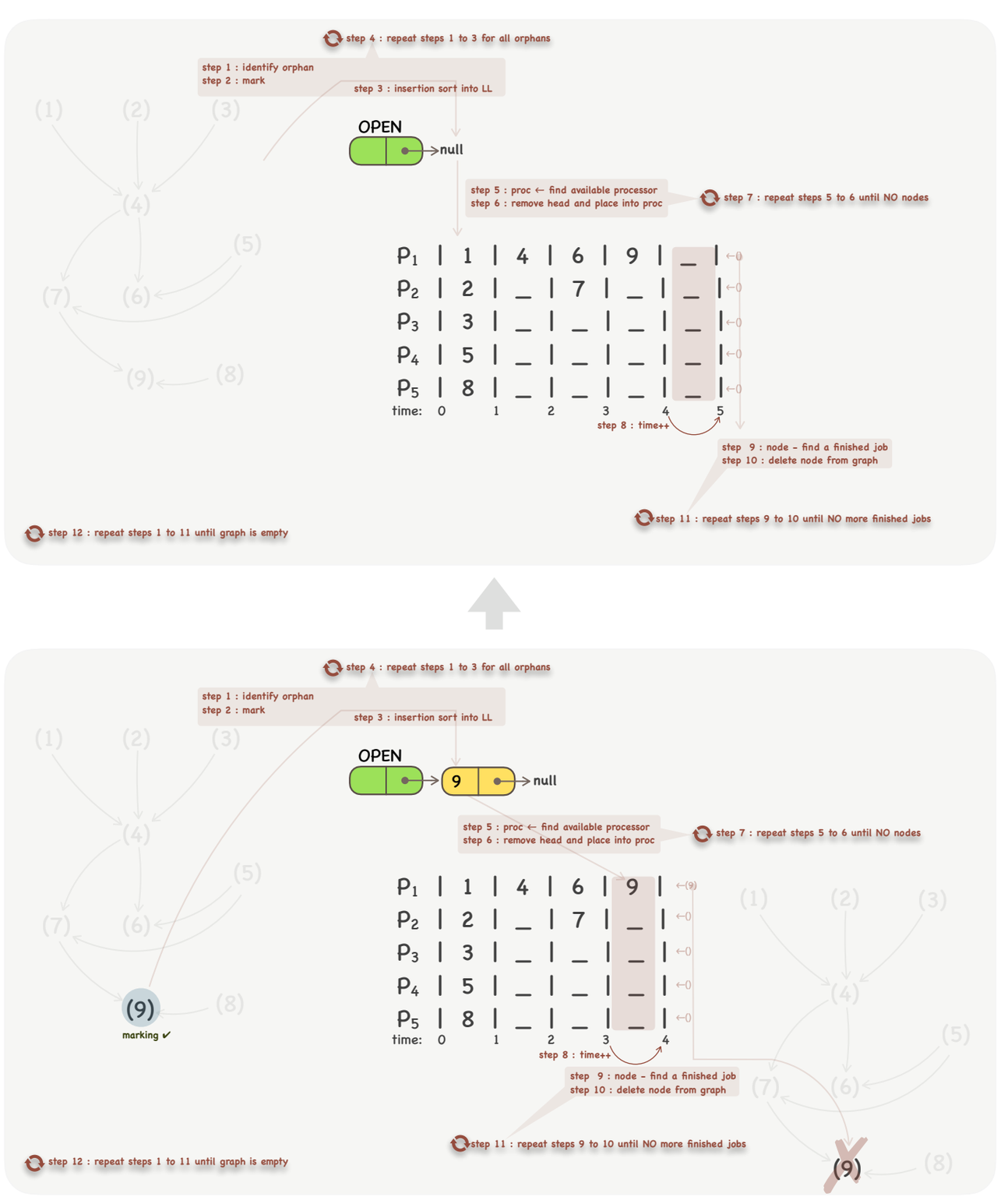
Algorithm — Algorithm steps for Scheduling (unlimited processors ; same job time)

- step0: initializations
  G ← graph (given)
  proc ← ∅ // since processor number is unlimited
  time ← 0 // we start at zero
- step1: node ← get an unmarked orphan node from graph G // node=job
- step2: mark node // b/c we don't want to get same node twice
- step3: insert node into Linked-List using Insertion Sort (ascending)
- step4: repeat step 1 to 3 until no more unmarked orphan nodes in G
- step5: find an available processor // worker that performs job
  if(no processors available)
   request a new processor // since unlimited
- step6: remove head from Linked-List place it into available processor
- step7: repeat steps 5 to 6 until Linked-List is empty
- step8: time++
- step9: node ← find a finished job i.e. processor is idle
- step10: delete node & its outgoing edges from graph G
- step11: repeat step 9 to 10 until no more idle processors
- step12: repeat step 1 to 11 until G is empty

unlimited proc — same job time

(1) → (4)
(2) → (4)
(3) → (4)
(4) → (6)
(5) → (6)
(4) → (7)
(5) → (7)
(7) → (9)
(8) → (9)





s/t: such that
b/c: because
btw: by the way
iff: if and only if
vs : versus
wrt : with respect to

scheduling (4)

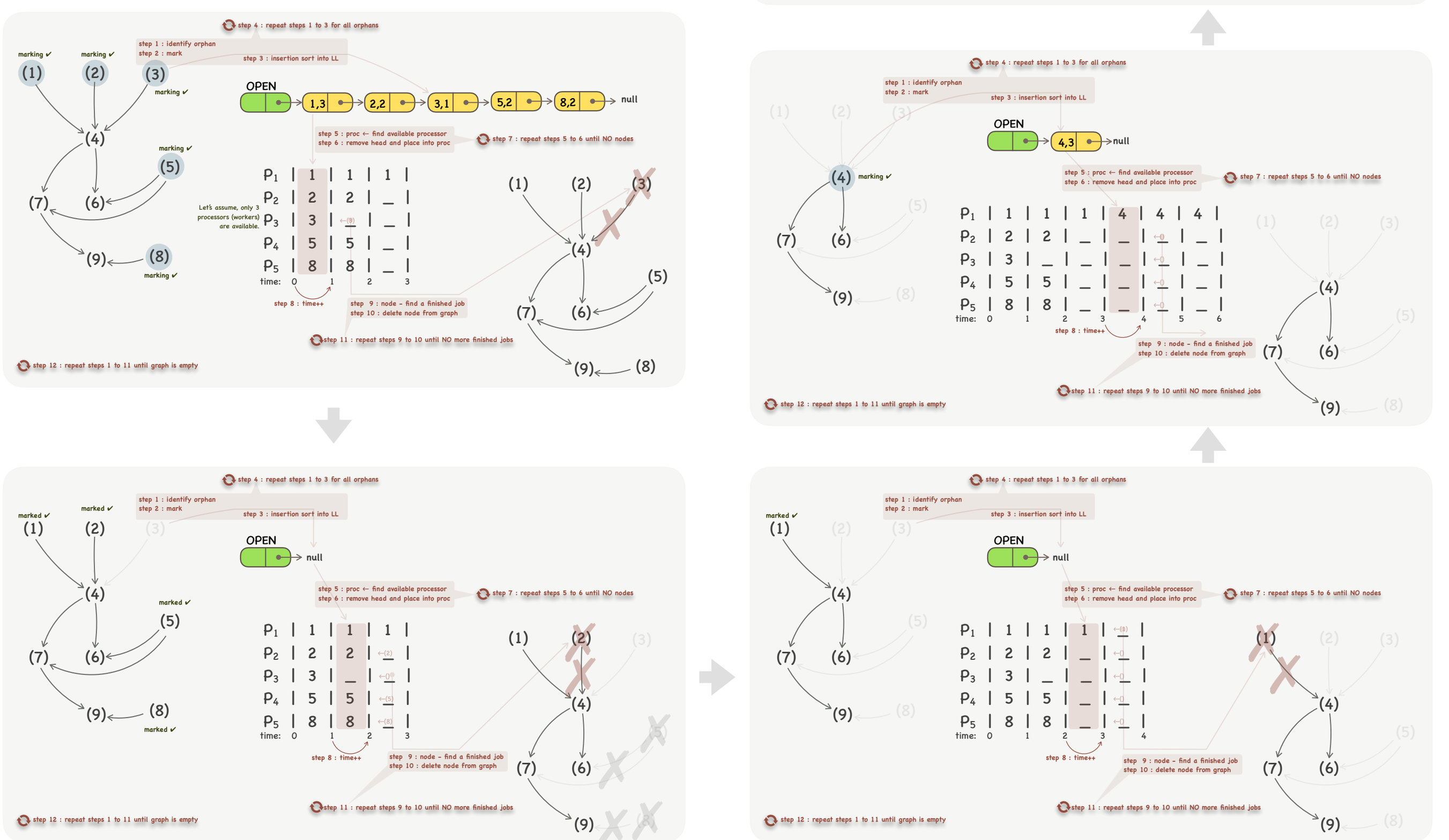
Algorithm — Algorithm steps for Scheduling (unlimited processors ; variable job time)

- step0: initializations
  G ← graph (given)
  proc ← ∅ // since processor number is unlimited
  time ← 0 // we start at zero
- step1: node ← get an unmarked orphan node from graph G // node=job
- step2: mark node // b/c we don't want to get same node twice
- step3: insert node into Linked-List using Insertion Sort (ascending)
- step4: repeat step 1 to 3 until no more unmarked orphan nodes in G
- step5: find an available processor // worker that performs job
  if(no processors available)
   request a new processor // since unlimited
- step6: remove head from Linked-List & place it into available processor
  populating the required time slices
- step7: repeat steps 5 to 6 until Linked-List is empty
- step8: time++
- step9: node ← find a finished job i.e. processor is idle
- step10: delete node & its outgoing edges from graph G
- step11: repeat step 9 to 10 until no more idle processors
- step12: repeat step 1 to 11 until G is empty

unlimited proc — variable job time

job id, job time length

(1.3) → (4.3)
(2.2) → (4.3)
(3.1) → (4.3)
(4.3) → (6.2)
(5.2) → (6.2)
(4.3) → (7.1)
(5.2) → (7.1)
(7.1) → (9.1)
(8.2) → (9.1)



scheduling (2)

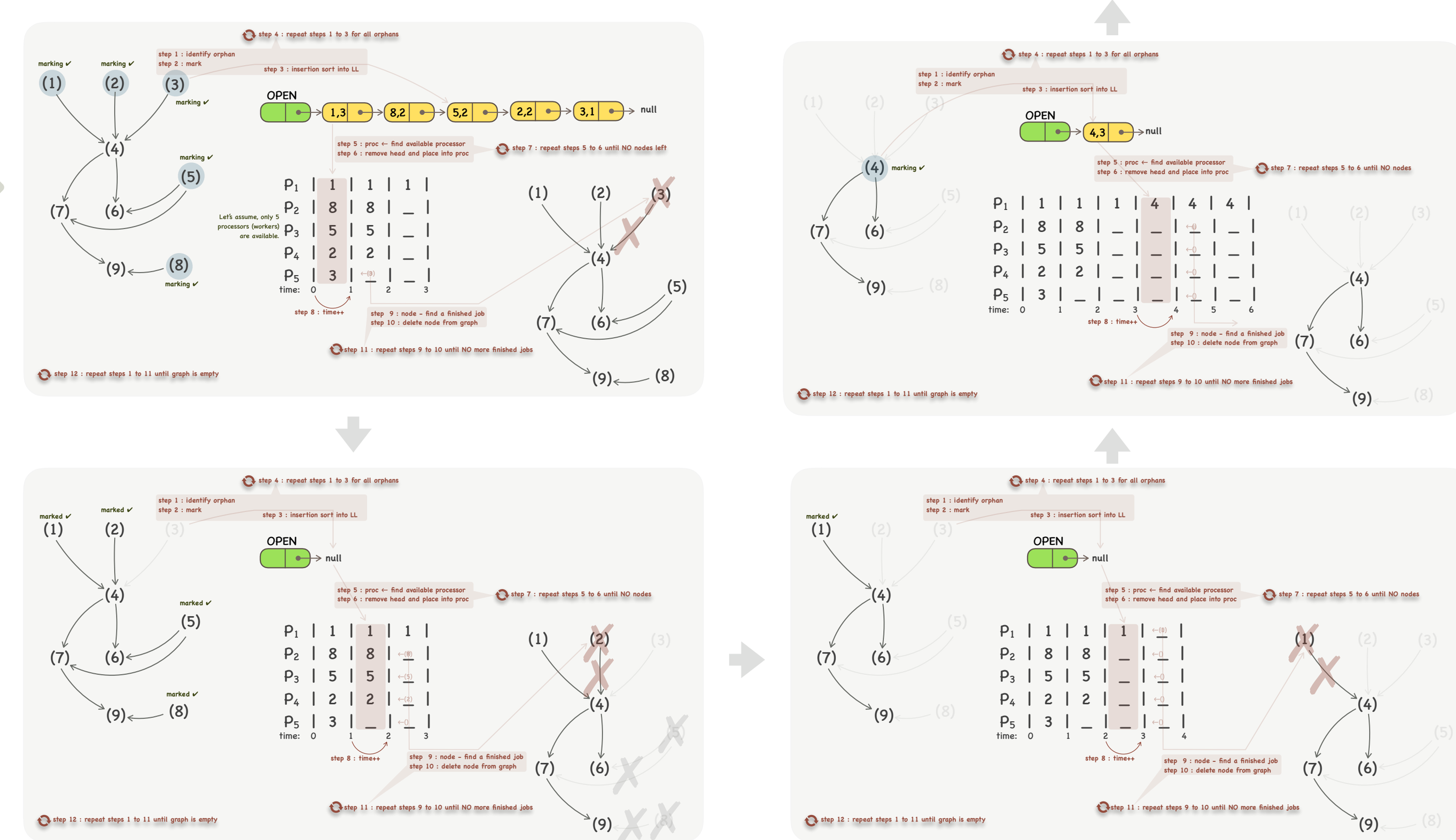
Algorithm — Algorithm steps for Scheduling (limited processors ; variable job time)

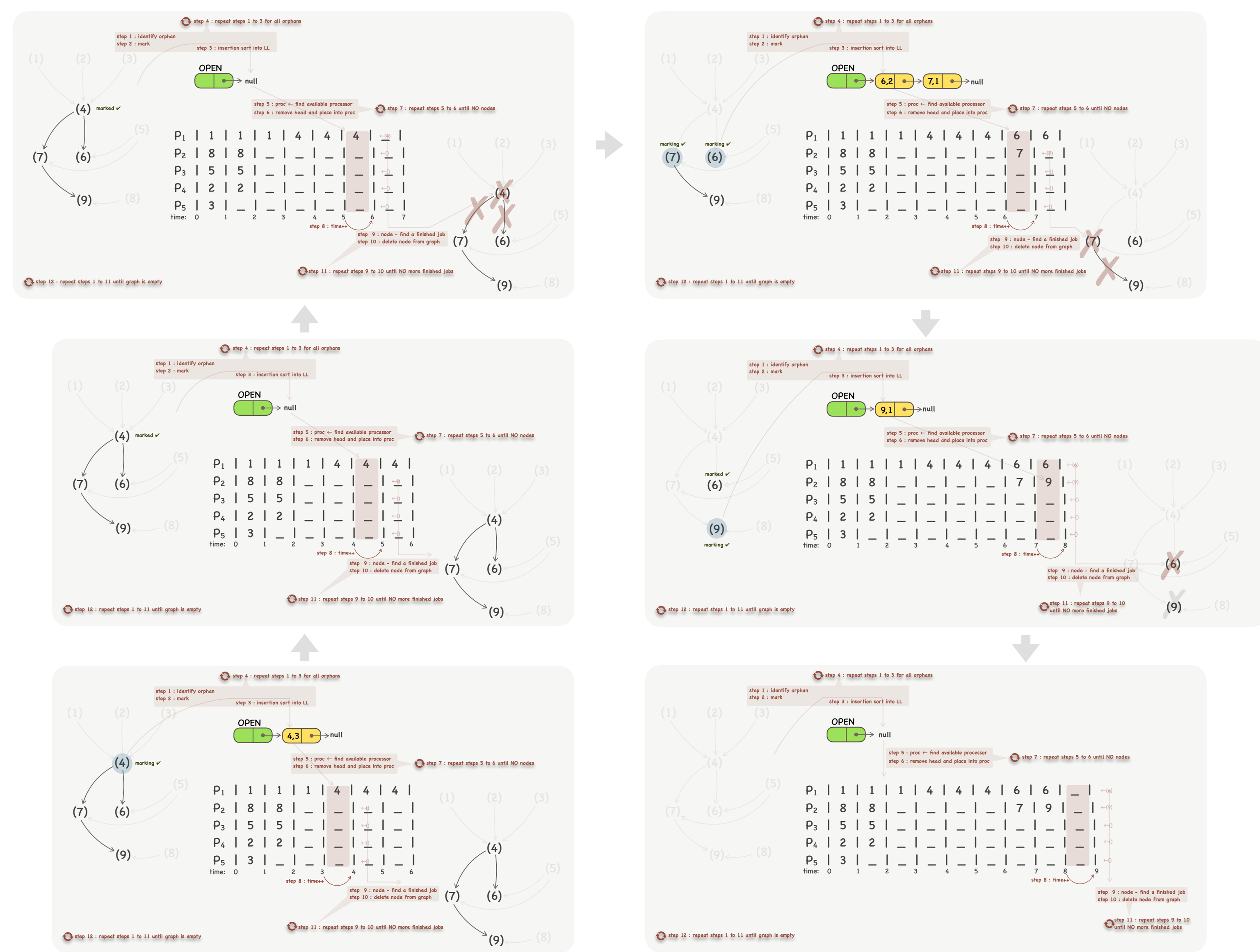
- step0: initializations
  G ← graph (given)
  proc ← ask how many // since processor number is limited
  time ← 0 // we start at zero
- step1: node ← get an unmarked orphan node from graph G //node=job
- step2: mark node // b/c we don't want to get same node twice
- step3: insert node into Linked-List using Insertion Sort (ascending) **descending by JobTime** (note: ascending by JobTime is more efficient)
- step4: repeat step 1 to 3 until no more unmarked orphan nodes in G
- step5: find an available processor // worker that performs job
  if(no processors available)
   wait for next time slice // since limited
- step6: remove head from Linked-List & place it into available processor
  populating the required time slices
- step7: repeat steps 5 to 6 until Linked-List is empty or no more processors are available
- step8: time++
- step9: node ← find a finished job i.e. processor is idle
- step10: delete node & its outgoing edges from graph G
- step11: repeat step 9 to 10 until no more idle processors
- step12: repeat step 1 to 11 until G is empty

limited proc — variable job time

job id, job time length

(1.3) → (4.3)
(2.2) → (4.3)
(3.1) → (4.3)
(4.3) → (6.2)
(5.2) → (6.2)
(4.3) → (7.1)
(5.2) → (7.1)
(7.1) → (9.1)
(8.2) → (9.1)





Notes are based on the lectures given by Dr. Tsaiyun Phillips in Fall 2018

Did you find a spelling error, grammar mistake, or any other discrepancy? Send an email to mdz.zone@iCloud.com