Improving filtering algorithms for the Disjunctive Constraint

Hamed Fahimi

OUTLINE

- SCHEDULING
- CONSTRAINT PROGRAMMING
- PRELIMINARIES
- PROPAGATION OF DISJUNCTIVE CONSTRAINT
- EXPERIMENTAL RESULTS
- CONCLUSION

OUTLINE

- SCHEDULING
- CONSTRAINT PROGRAMMING
- PRELIMINARIES
- PROPAGATION OF DISJUNCTIVE CONSTRAINT
- EXPERIMENTAL RESULTS
- CONCLUSION

What is Scheduling?

A hands-on application of scheduling!

- Where? In the wood product industry!

- The wood is wet at first and must be dried before being cut and used for construction.

- The **task** is to put the wood in a dryer and make sure it is solid and it won't deform. The **resource** is the dryer.

- There are so many loads to be put in the dryer. So, we have as many tasks as the number of loads.

A hands-on application of scheduling!

- The **earliest starting time** of a task is when the truck arrives with the wood.
- For each load, there is a **deadline** which is the time that the customer wants to have it ready.
- The **processing time** is the amount of time that the wood remains in the dryer to lose moisture and dry out.

Illustration of a task and its parameters

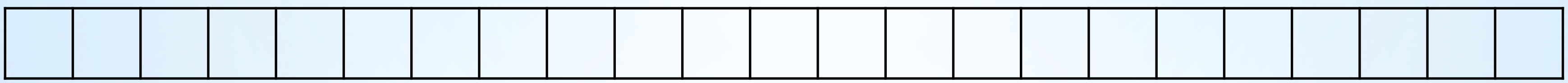


Illustration of a task and its parameters

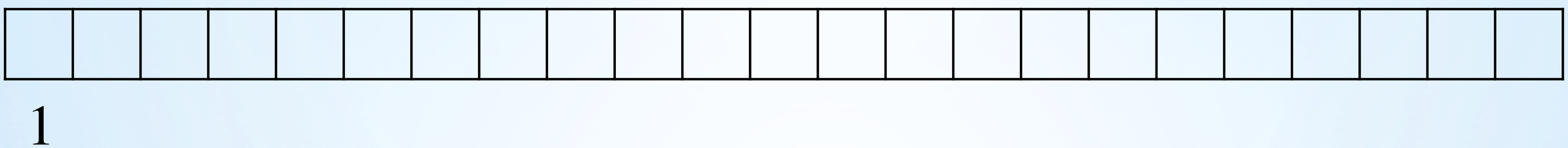


Illustration of a task and its parameters

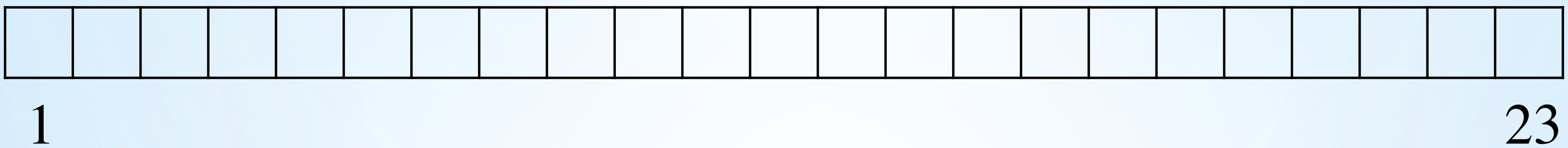


Illustration of a task and its parameters

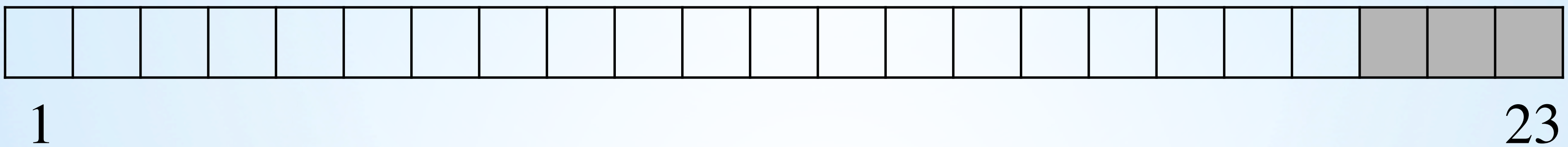


Illustration of a task and its parameters

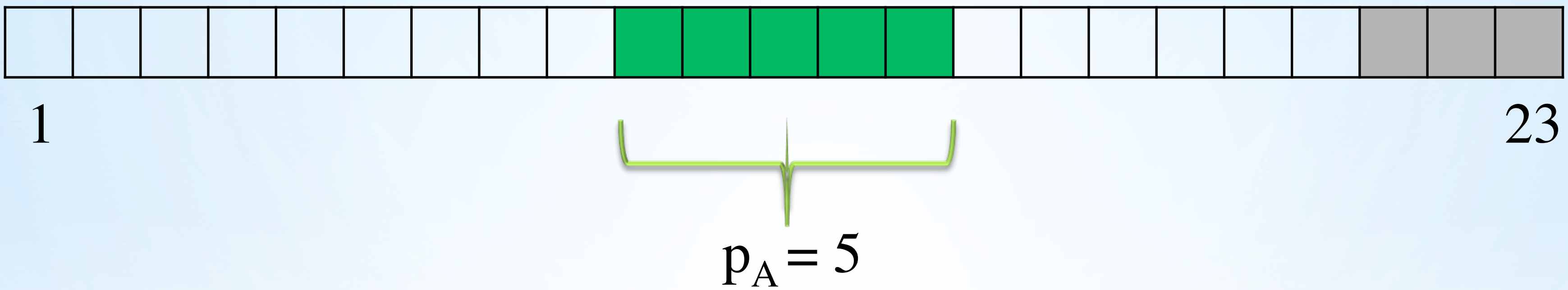


Illustration of a task and its parameters

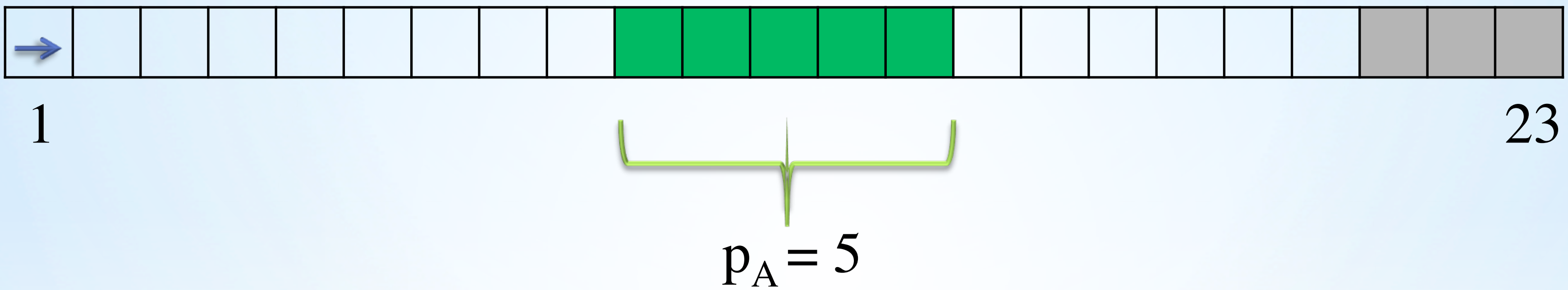


Illustration of a task and its parameters

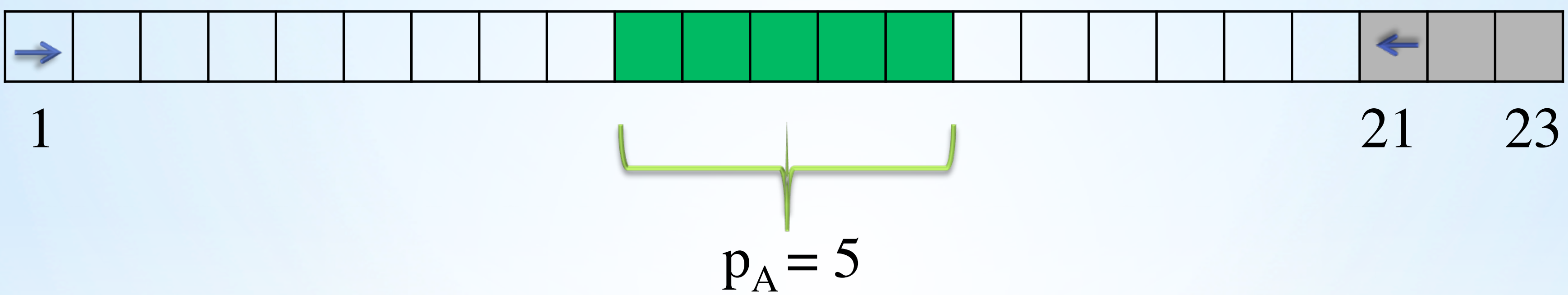


Illustration of a task and its parameters

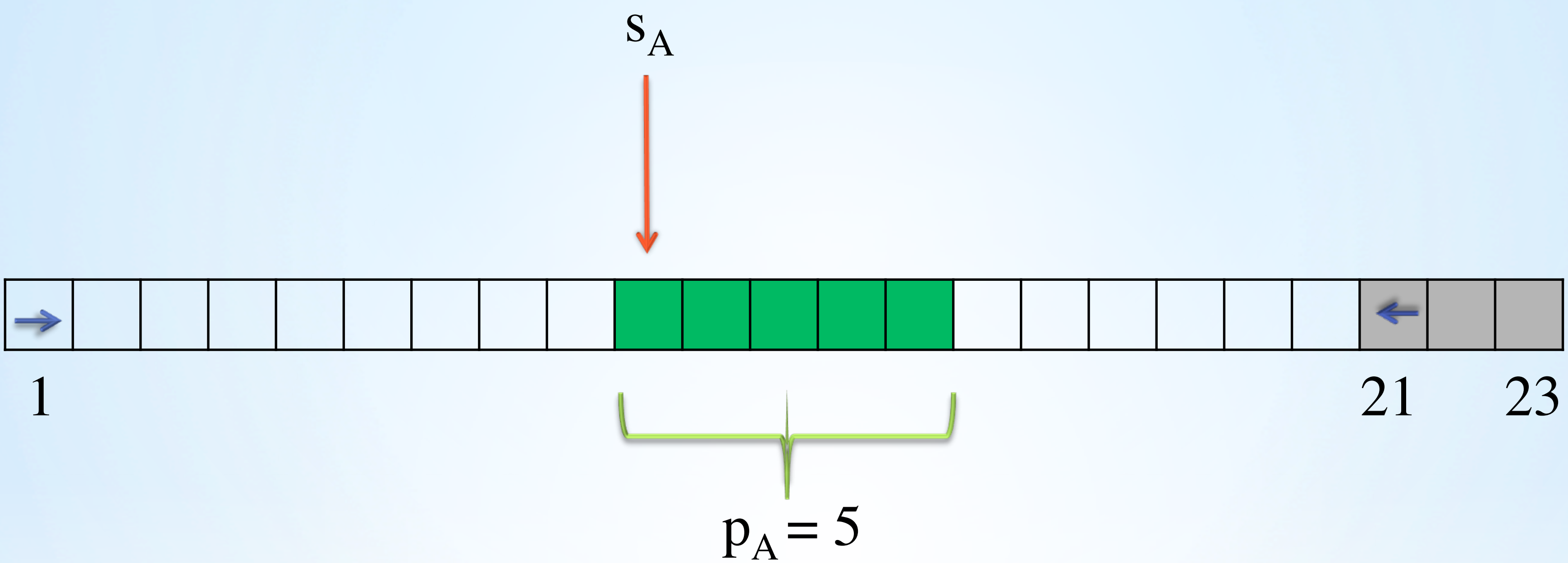


Illustration of a task and its parameters

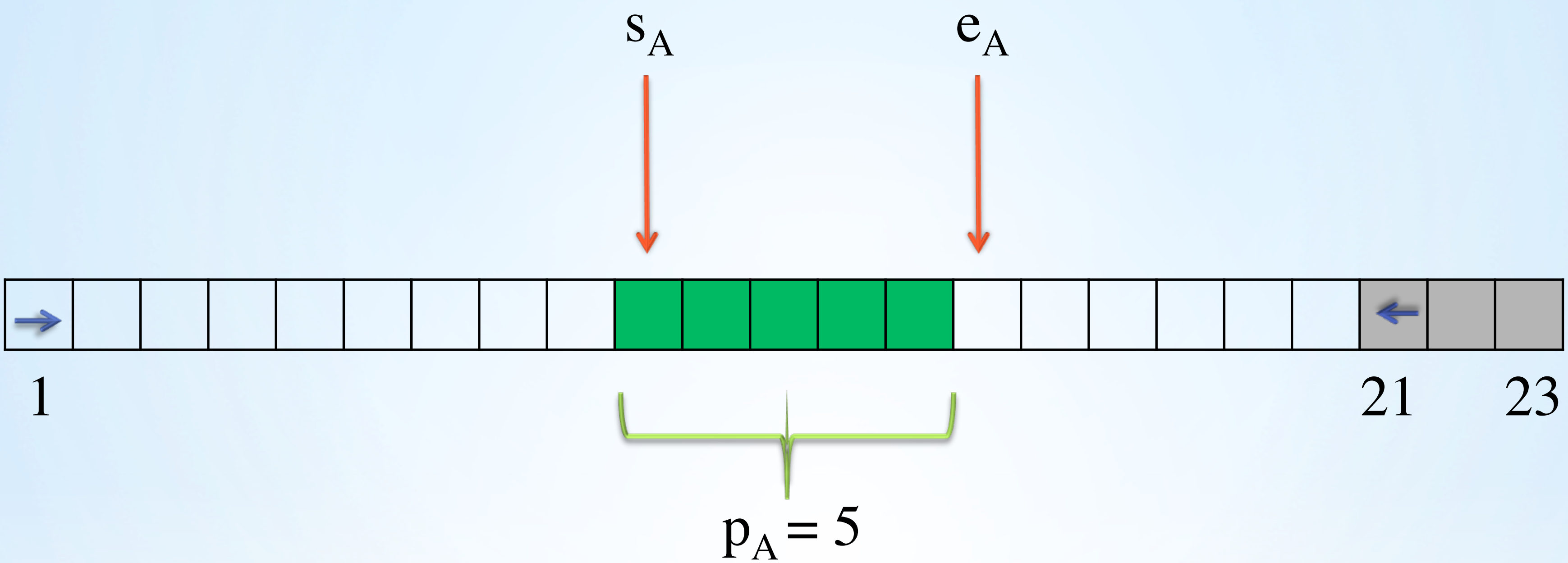


Illustration of a task and its parameters

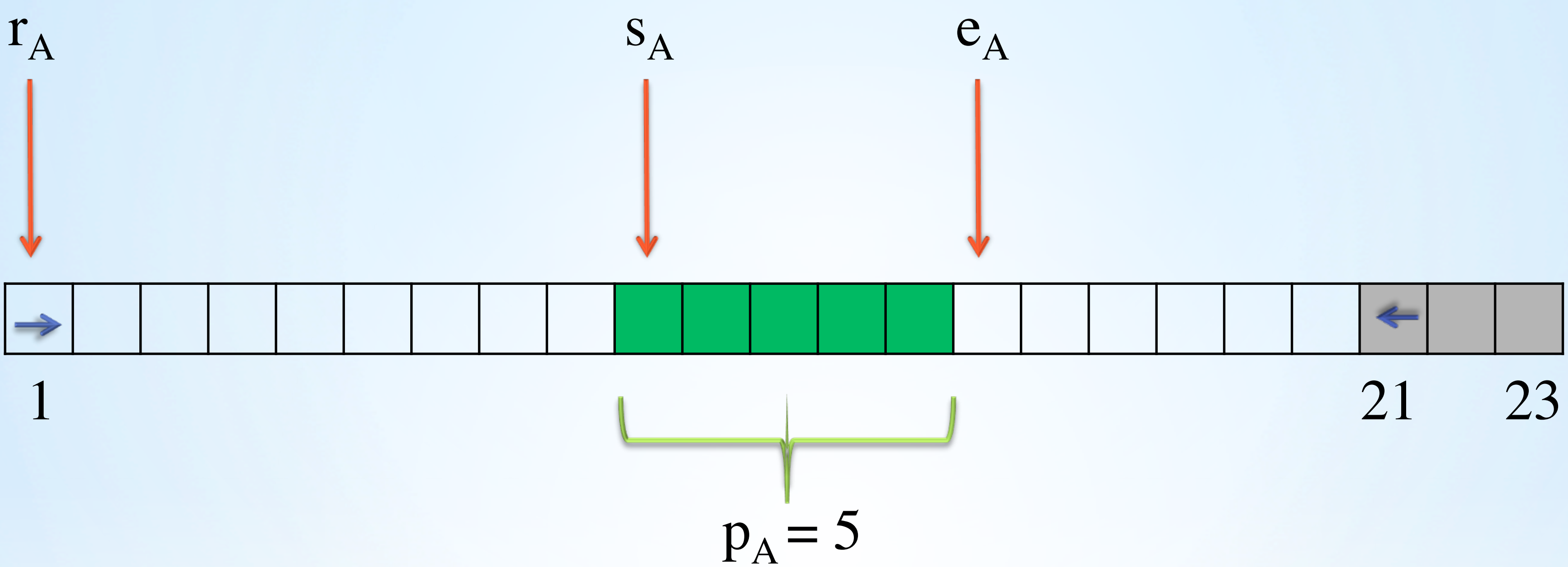


Illustration of a task and its parameters

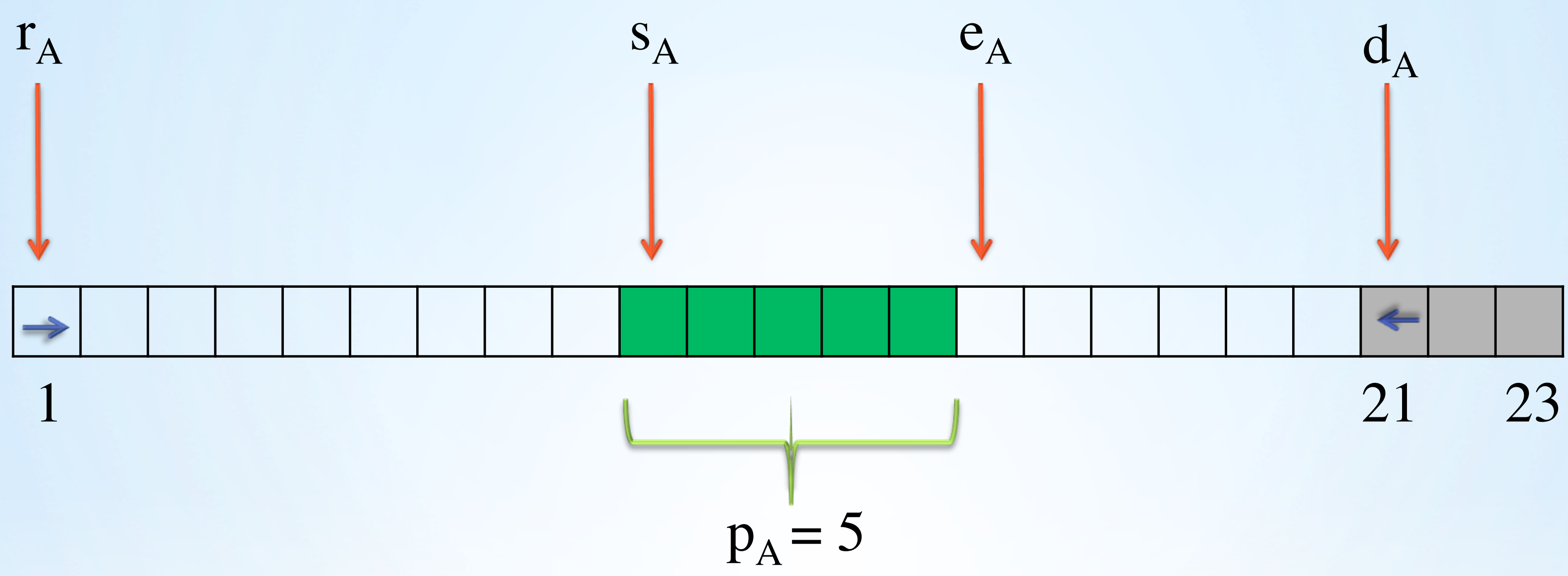


Illustration of a task and its parameters

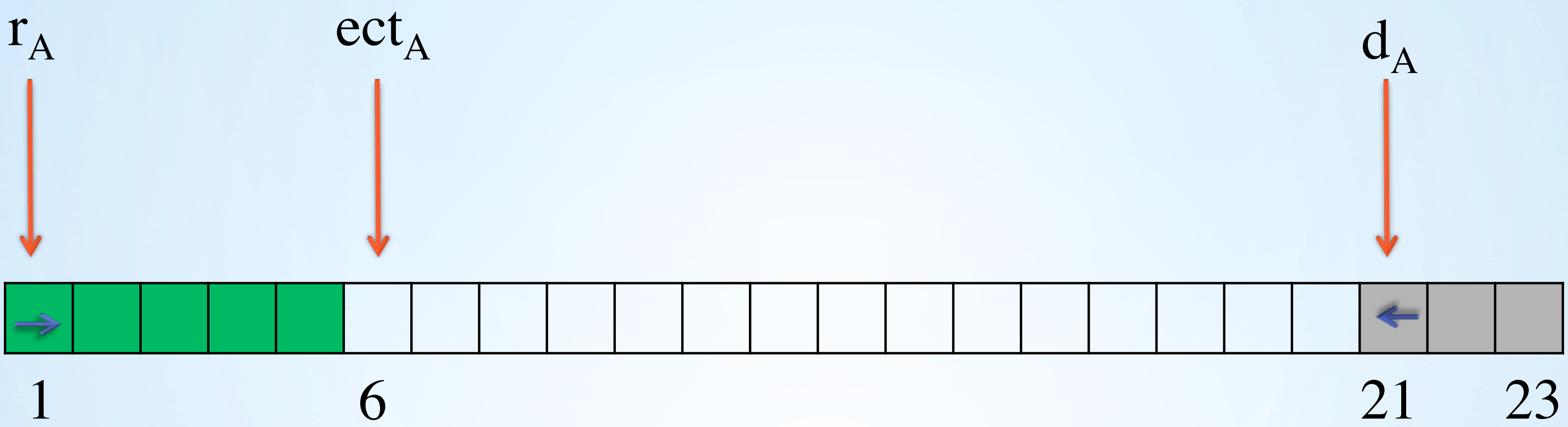


Illustration of a task and its parameters

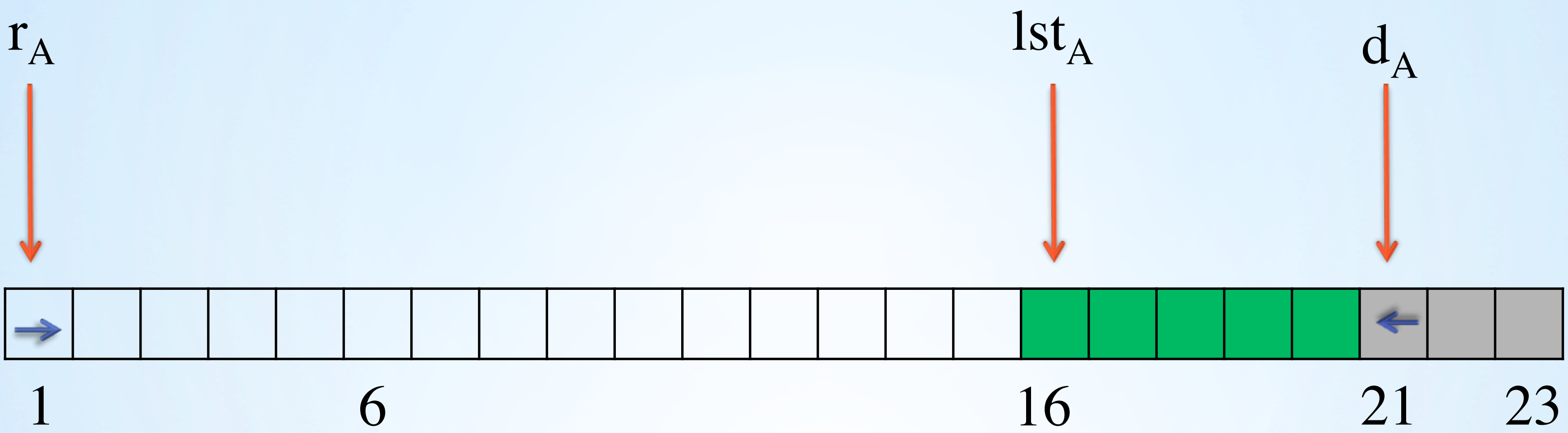


Illustration of a task and its parameters

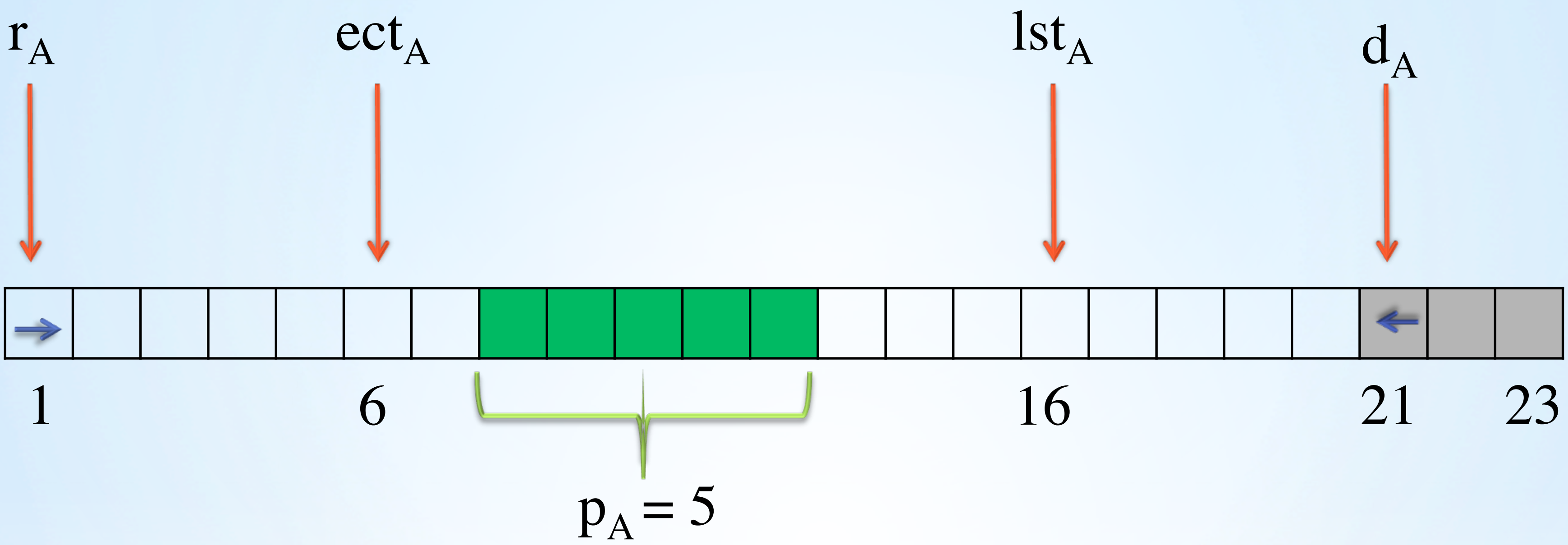
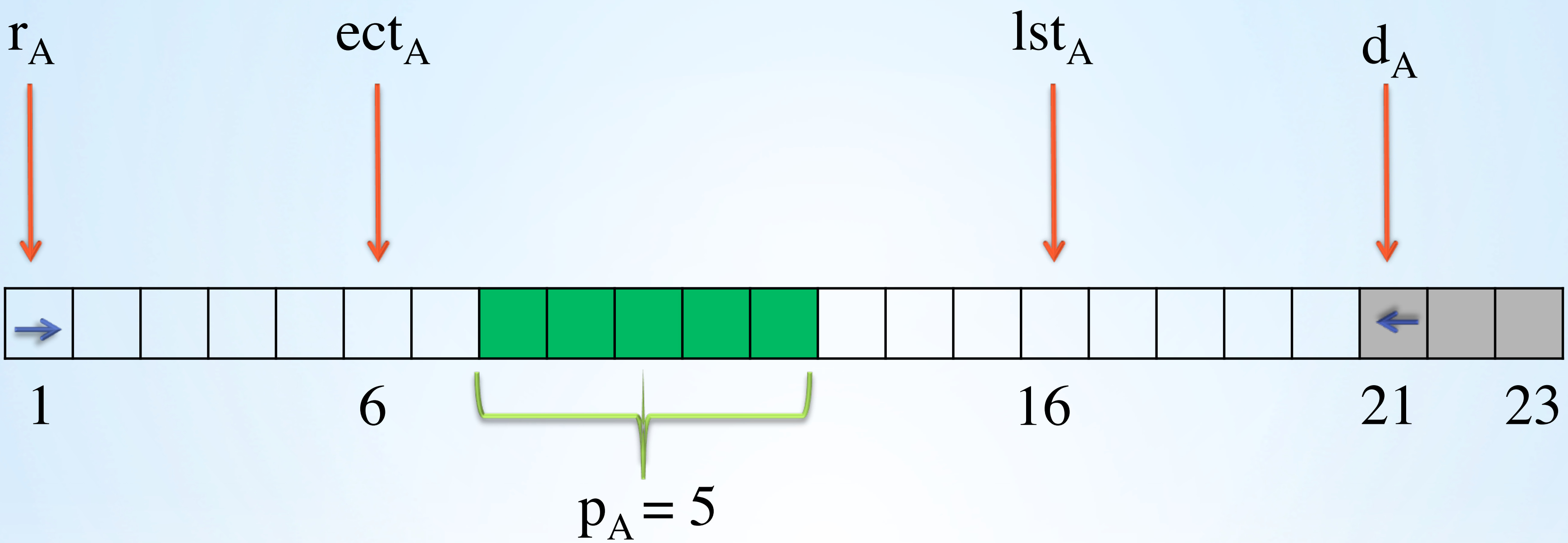
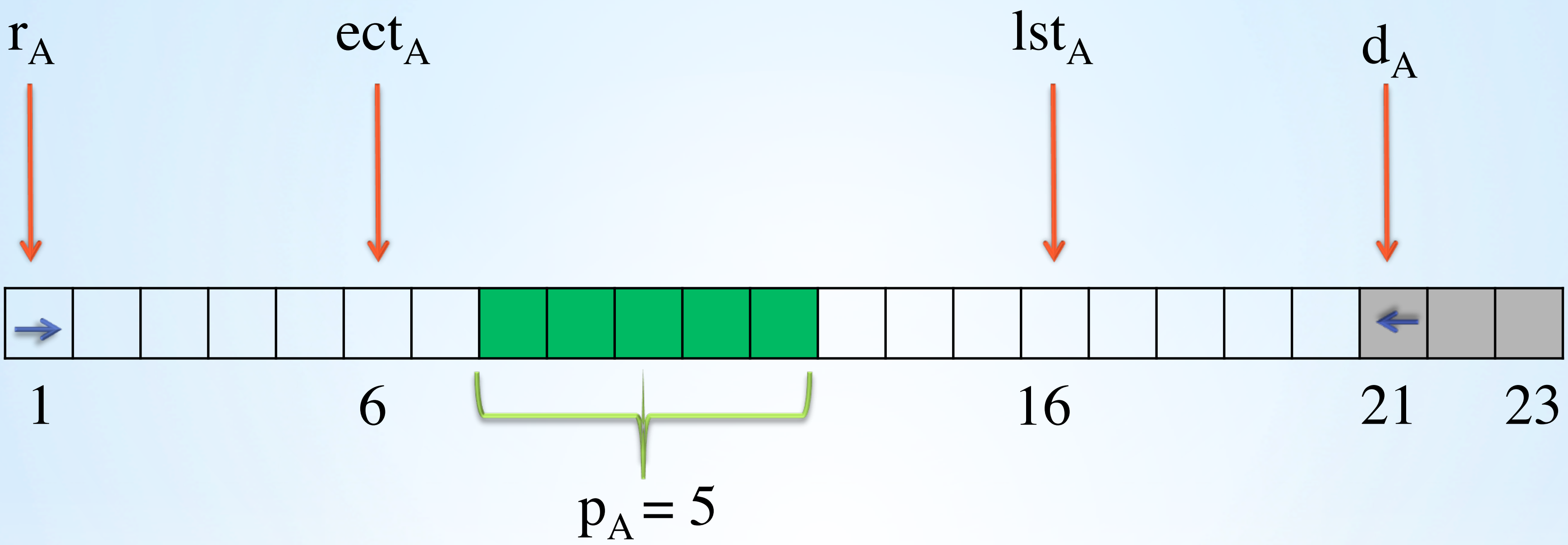


Illustration of a task and its parameters



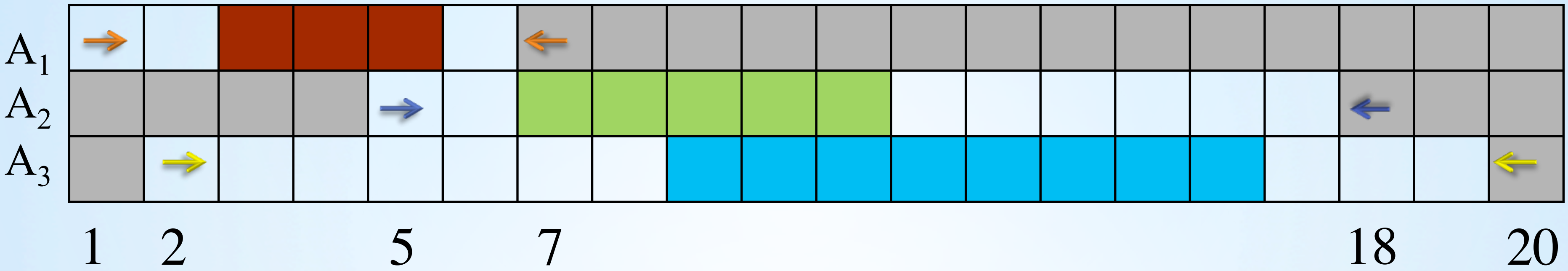
- We call the interval $[r_i, d_i)$ the **allowed execution interval** of task A_i.

Illustration of a task and its parameters

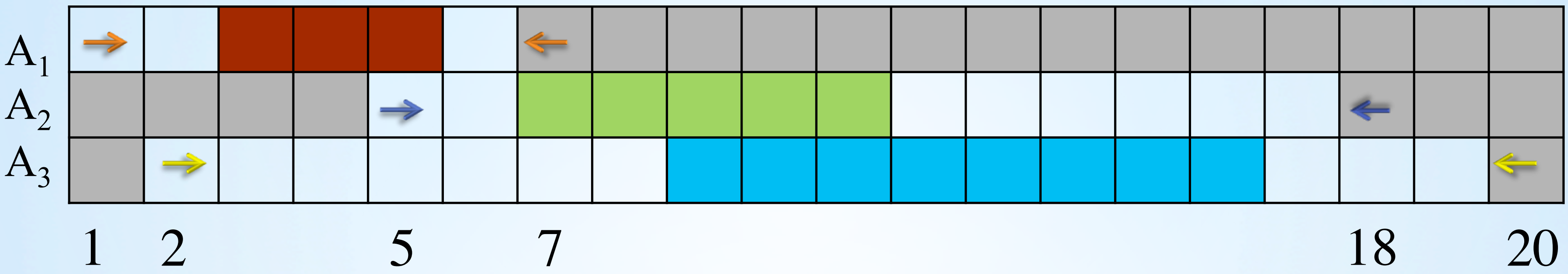


- We call the interval $[r_i, d_i)$ the **allowed execution interval** of task A_i.
- → The release time;
- ← The deadline;
- The number of colored cells = Processing time;
- Gray cells: Out of the allowed execution interval of the task.

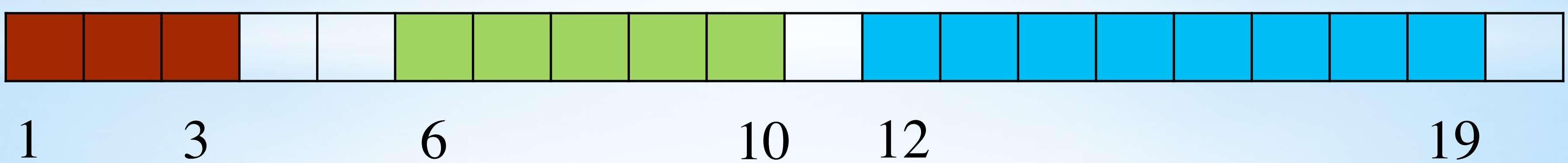
Disjunctive scheduling



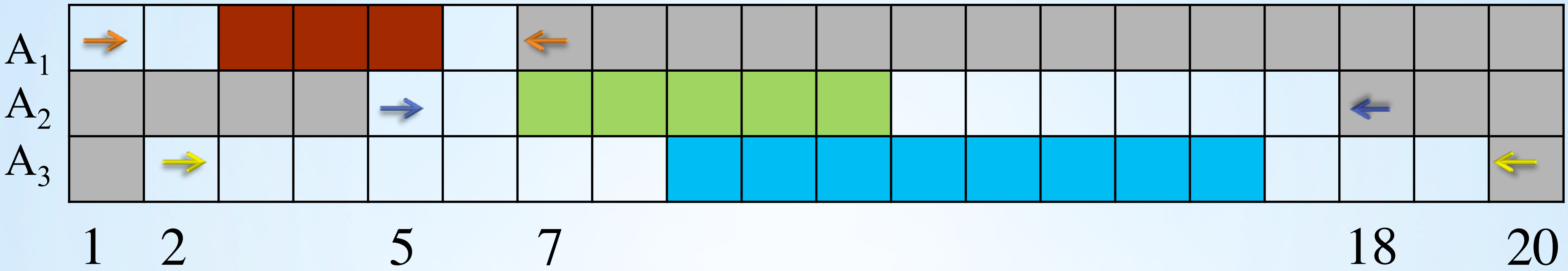
Disjunctive scheduling



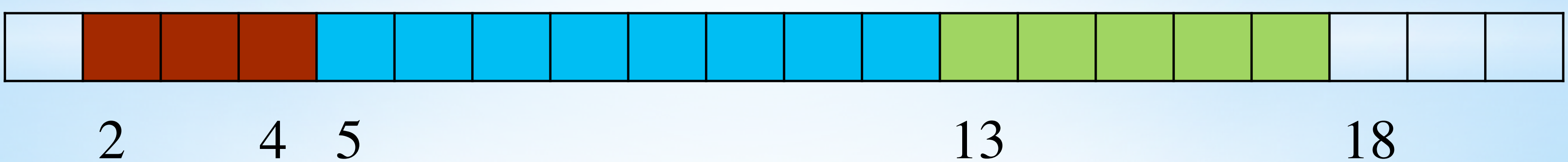
- A feasible schedule!



Disjunctive scheduling



- An alternative feasible schedule!

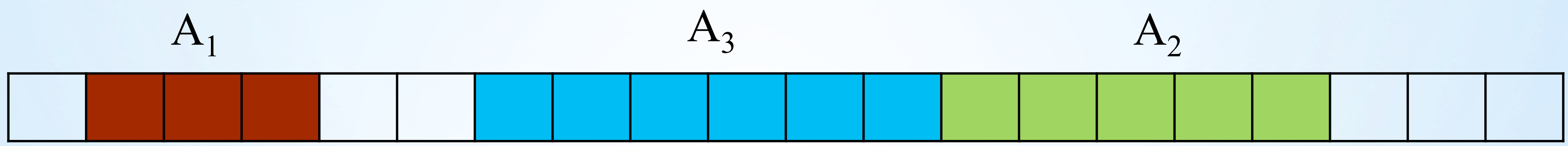


Scheduling classification with the tasks

- **Non-Preemptive Scheduling:**

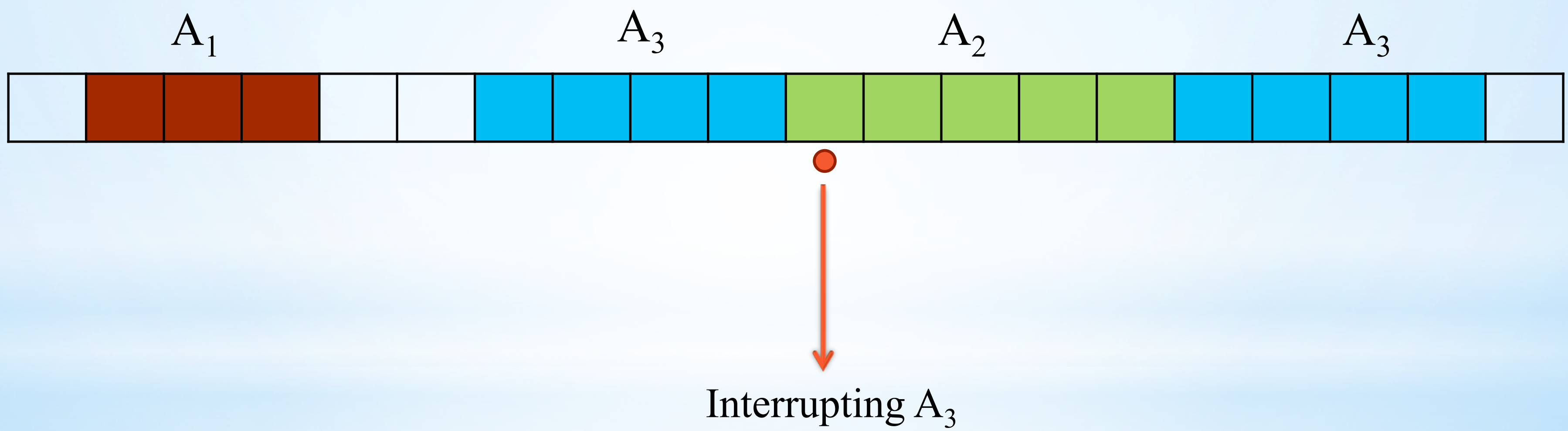
Scheduling classification with the tasks

- **Non-Preemptive Scheduling:**



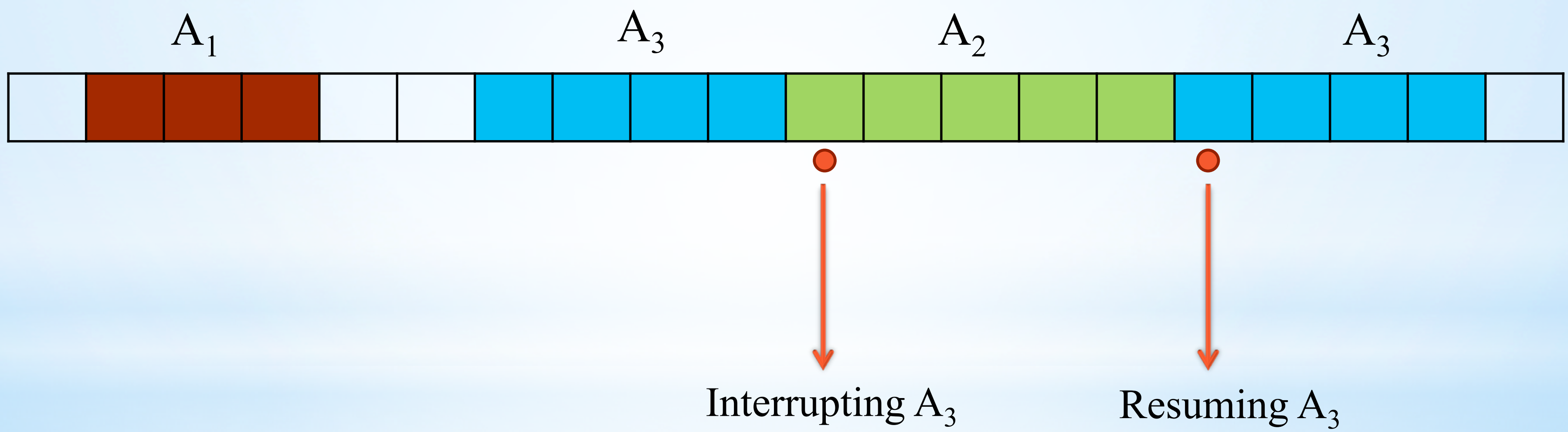
Scheduling classification with the tasks

- **Preemptive Scheduling:**



Scheduling classification with the tasks

- **Preemptive Scheduling:**



OUTLINE

- SCHEDULING
- CONSTRAINT PROGRAMMING
- PRELIMINARIES
- PROPAGATION OF DISJUNCTIVE CONSTRAINT
- EXPERIMENTAL RESULTS
- CONCLUSION

Definition of Constraint Programming

- Let $X = \{X_1,\ldots,X_n\}$ be a set of variables. A **constraint** C is a condition, imposed over a subset $X_C \subseteq X$, which describes a relation between the elements of X_C.

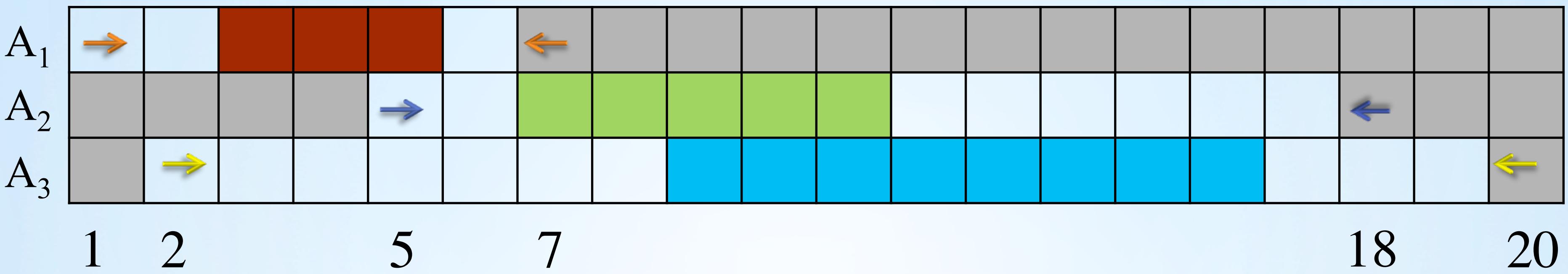
- An instance of a CSP is described by the sets

$X = \{X_1,\ldots,X_n\}$ $D = \{D(X_1) ,\ldots, D(X_n)\}$

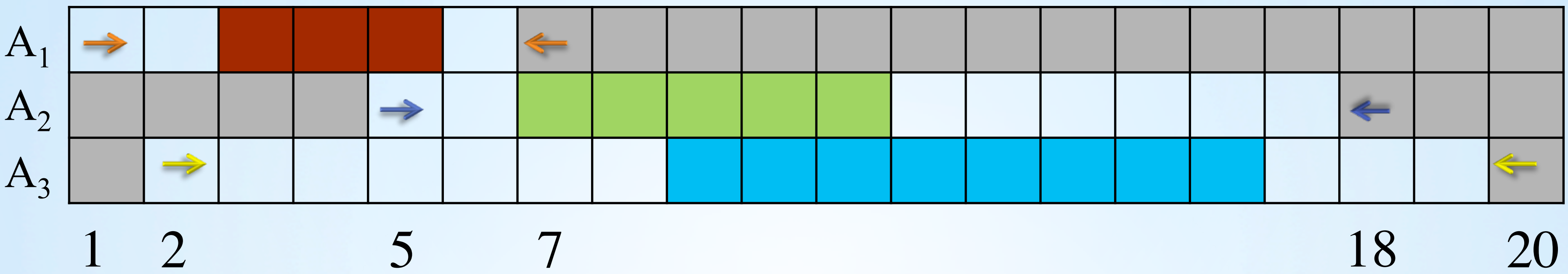
$C = \{C_1,\ldots,C_m\}$ $X' = \{X_{C1} ,\ldots,X_{Cm}\}$

- An assignment of values to the variables, which satisfies all of the constraints of a CSP, is called a **solution**. A solution for the constraint C is called a **support**.

Example (Disjunctive problem)



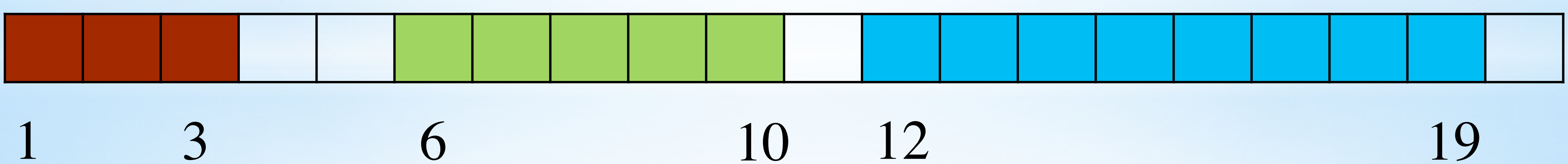
Example (Disjunctive problem)



- $S=\{S_1, S_2, S_3\}$
- $S_1 \in [1, 4], S_2 \in [5,13], S_3 \in [2,12]$
- $(S_i + p_i \leq S_j) \lor (S_j + p_j \leq S_i)$ (for i,j=1,2,3 & i≠j)

Example (Disjunctive problem)

- $S=\{S_1, S_2, S_3\}$
- $S_1 \in [1, 4], S_2 \in [5,13], S_3 \in [2,12]$
- $(S_i + p_i \leq S_j) \vee (S_j + p_j \leq S_i)$ (for $i,j=1,2,3$ & $i\neq j$)



- (1, 6, 12) is a support.

Disjunctive Constraint

- Let $I = \{A_1,\ldots,A_n\}$ be a set of tasks with unknown starting times S_i, and known processing time p_i ($1 \leq i \leq n$).

- **Variables:** $X = \{S_1,\ldots,S_n\}$;
- **Domains:** $D(S_i)=[r_i , lst_i]$;
- **Constraint:** No more than one task executes at each time t.

- The constraint DISJUNCTIVE($[S_1,\ldots,S_n]$) is satisfied, if for all pairs of tasks ($i \neq j$)

$$S_i + p_i \leq S_j \text{ or } S_j + p_j \leq S_i$$

Constraint filtering

- Initially, the domains of a CSP may include values which are not consistent with some constraints of the problem.

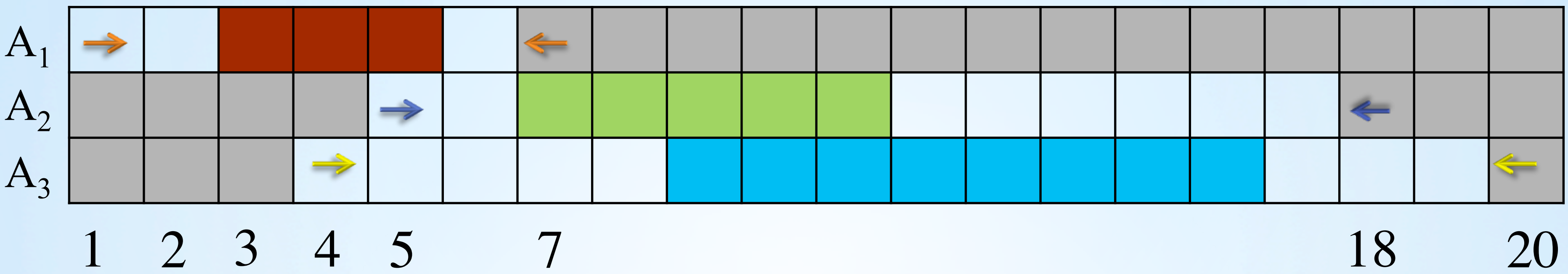
- To reduce the search space, solvers use *filtering algorithms* associated to each constraint.

- Filtering algorithms keep on excluding values of the domains that do not lead to a feasible solution, until it is not possible to prune the domains of variables further.

Example (Disjunctive constraint)

- There is no chance to start task A_3 at its release time, as A_1 would not execute. Thus, the values {2, 3} should be filtered from the domain of A_3 .

Example (Disjunctive constraint)



- There is no chance to start task A_3 at its release time, as A_1 would not execute. Thus, the values {2, 3} should be filtered from the domain of A_3 .

- The values {2, 3} are out of the allowed execution interval of A_3.

Disjunctive Constraint

- It is NP-Complete to determine whether there exists a solution to Disjunctive constraint.
- It is NP-Hard to filter out all values that do not lead to a solution.
- Nonetheless, there exist rules that detect in polynomial time some filtering of the domains of the tasks.
- Our goal is to improve some existing filtering algorithms for the Disjunctive constraint.

OUTLINE

- SCHEDULING
- CONSTRAINT PROGRAMMING
- PRELIMINARIES
- PROPAGATION OF DISJUNCTIVE CONSTRAINT
- EXPERIMENTAL RESULTS
- CONCLUSION

Preliminaries

- We aim to design filtering algorithms, which are faster than the previously known algorithms.

- To achieve this goal, there are two major operations, to take advantage of:

- Sorting in linear time;

- Union-Find data structure.

- Since all the time points can be encoded with fewer than 32 bits, *radix sort* sorts them in linear time.

Union-Find Data structure

Function (Gabow & Tarjan, 1983)	Operation	Complexity
Union-Find(n)	Initializes n disjoint sets $\{0\}, \{1\},\ldots, \{n - 1\}$	$O(n)$

Union-Find Data structure

Function (Gabow & Tarjan, 1983)	Operation	Complexity
Union-Find(n)	Initializes n disjoint sets $\{0\}, \{1\},\ldots, \{n - 1\}$	$O(n)$
Union(a, $a+1$)	Merges the set that contains the element a with the set that contains the element $a+1$	$O(1)$

Union-Find Data structure

Function (Gabow & Tarjan, 1983)	Operation	Complexity
Union-Find(n)	Initializes n disjoint sets $\{0\}, \{1\}, \ldots, \{n - 1\}$	$O(n)$
Union(a, a+1)	Merges the set that contains the element a with the set that contains the element a+1	$O(1)$
FindSmallest(a)	Returns the smallest element of the set that contains a	$O(1)$

Union-Find Data structure

Function (Gabow & Tarjan, 1983)	Operation	Complexity
Union-Find(n)	Initializes n disjoint sets $\{0\}, \{1\}, \ldots, \{n - 1\}$	$O(n)$
Union(a, $a+1$)	Merges the set that contains the element a with the set that contains the element $a+1$	$O(1)$
FindSmallest(a)	Returns the smallest element of the set that contains a	$O(1)$
FindGreatest(a)	Returns the greatest element of the set that contains a	$O(1)$

OUTLINE

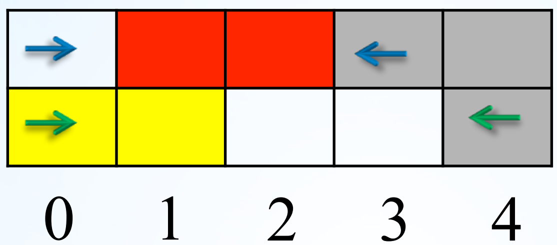
- SCHEDULING
- CONSTRAINT PROGRAMMING
- PRELIMINARIES
- PROPAGATION OF DISJUNCTIVE CONSTRAINT
- EXPERIMENTAL RESULTS
- CONCLUSION

Time-Tabling

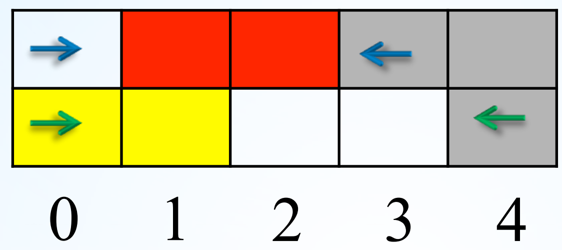
- A technique to filter the Disjunctive constraint.

- It consists of finding the necessary usage of the resource over a time interval.

Time-Tabling

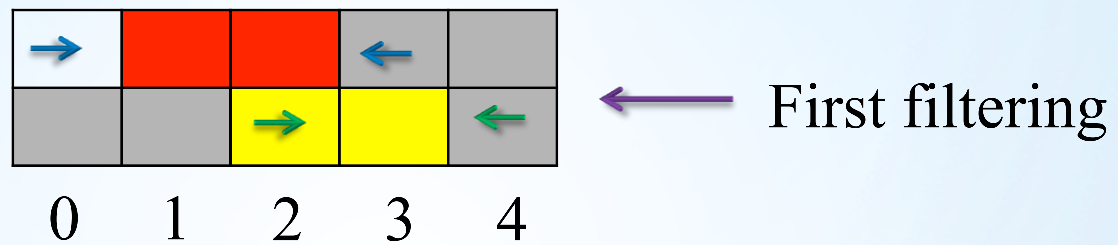


Time-Tabling

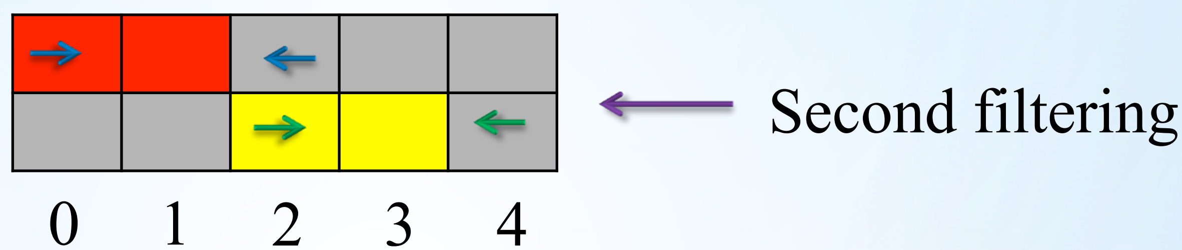


- If $lst_i < ect_i$ for a task i, then the interval $[lst_i, ect_i)$ is called the *fixed part* of i.

Time-Tabling



Time-Tabling

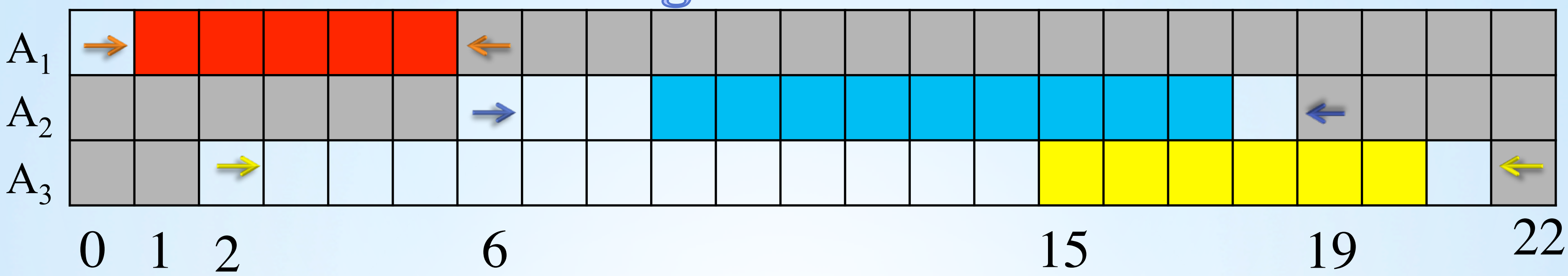


Time-Tabling

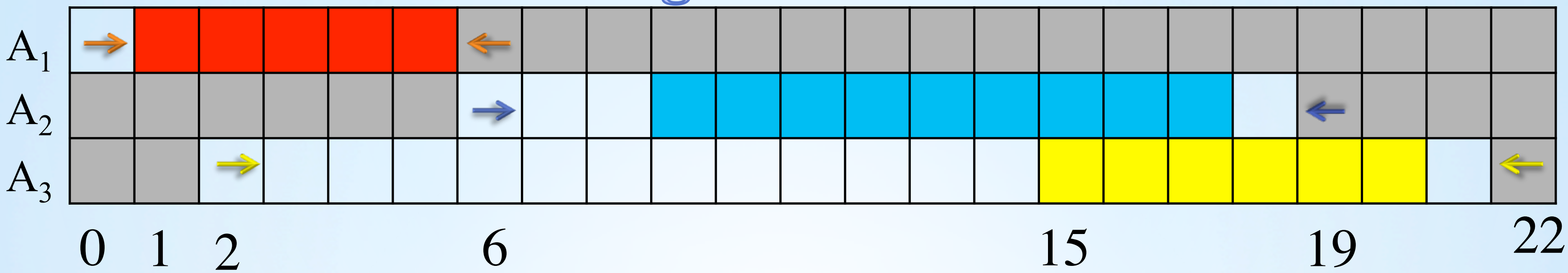
- Ouellet & Quimper presented an algorithm for Time-Tabling on a more general case in $O(n\log(n))$.

- We took advantage of Union-Find to achieve a linear time algorithm for Time-Tabling in the Disjunctive case.

The strategy of our Time-Tabling algorithm

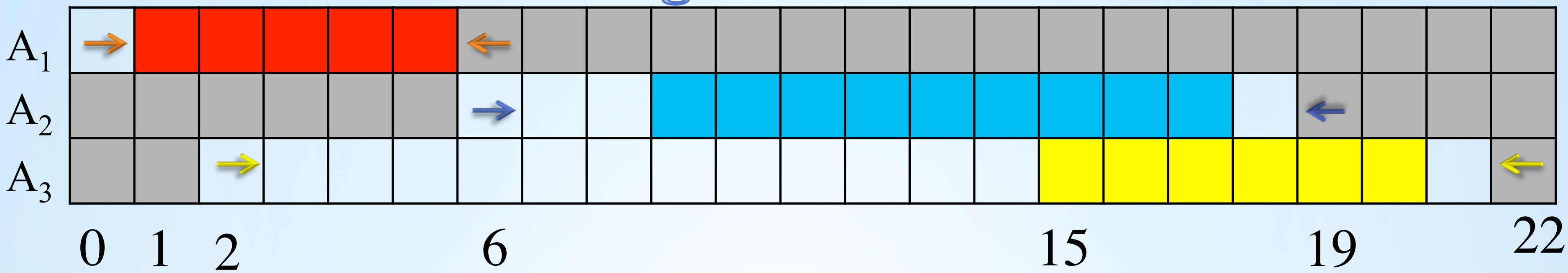


The strategy of our Time-Tabling algorithm



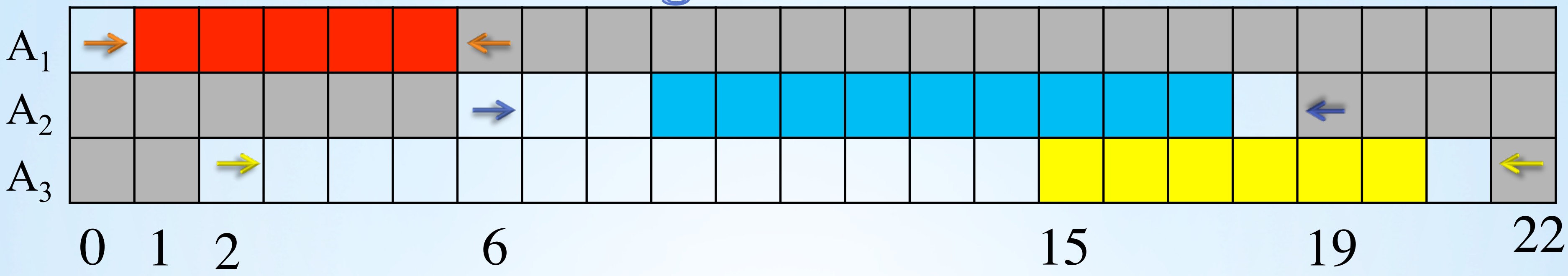
- First, we list the fixed parts of the tasks which have fixed part.

The strategy of our Time-Tabling algorithm

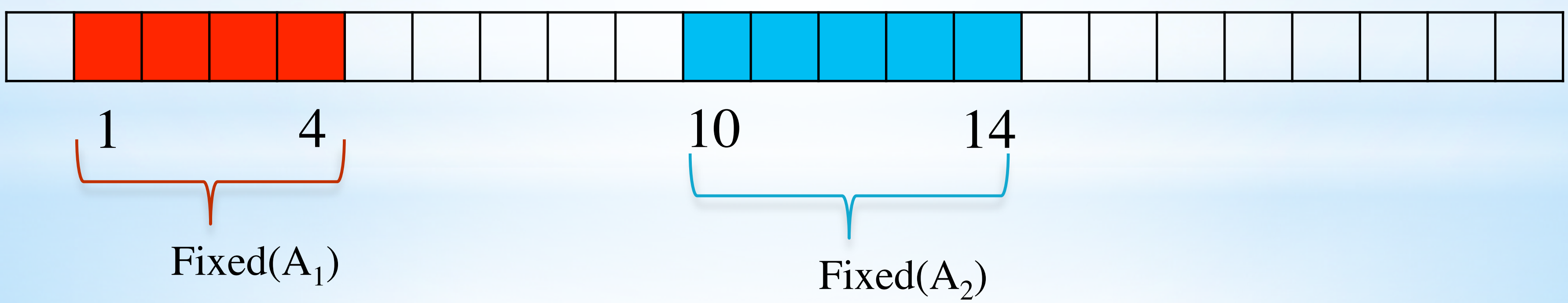


- First, we list the fixed parts of the tasks which have fixed part.
- A_1 and A_2 have fixed parts.

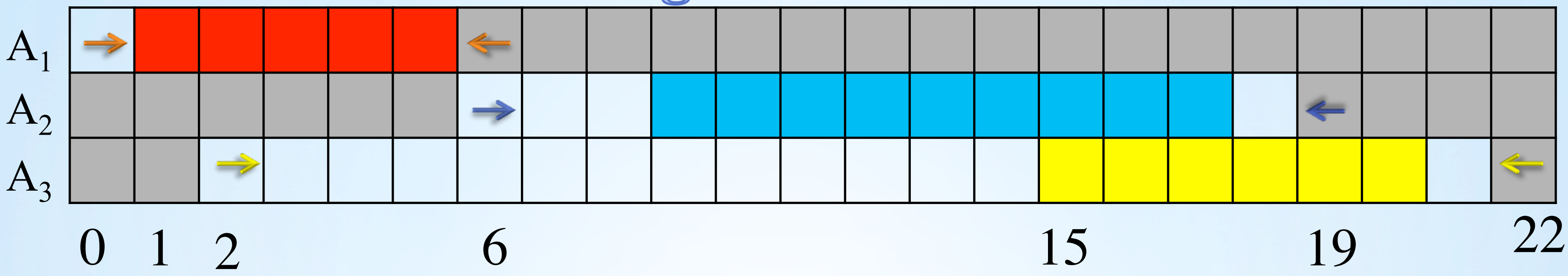
The strategy of our Time-Tabling algorithm



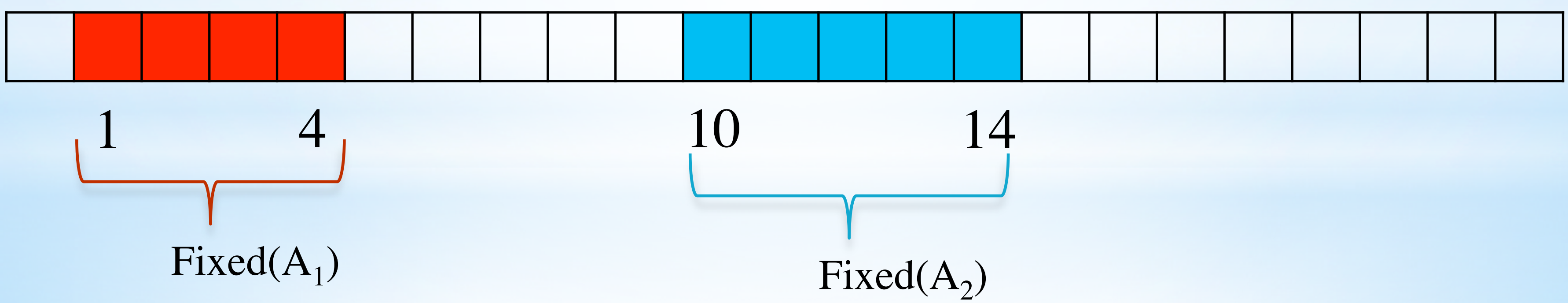
- First, we list the fixed parts of the tasks which have fixed part.
- A_1 and A_2 have fixed parts.



The strategy of our Time-Tabling algorithm

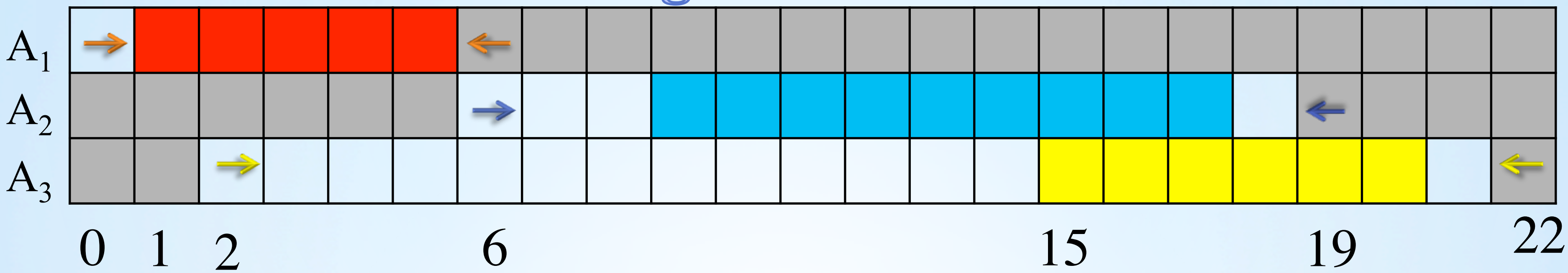


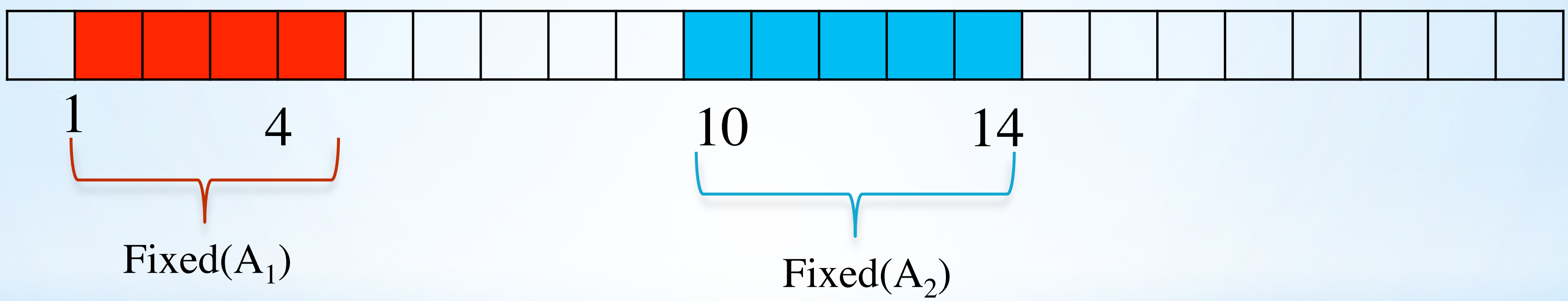
- First, we list the fixed parts of the tasks which have fixed part.
- A_1 and A_2 have fixed parts.

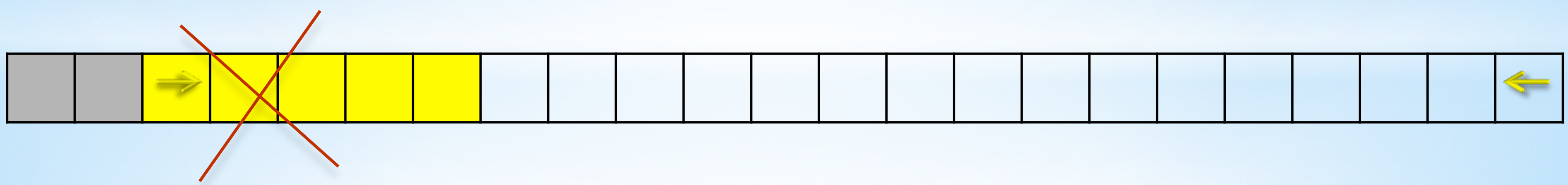


- We process the tasks in increasing order of processing times.

The strategy of our Time-Tabling algorithm

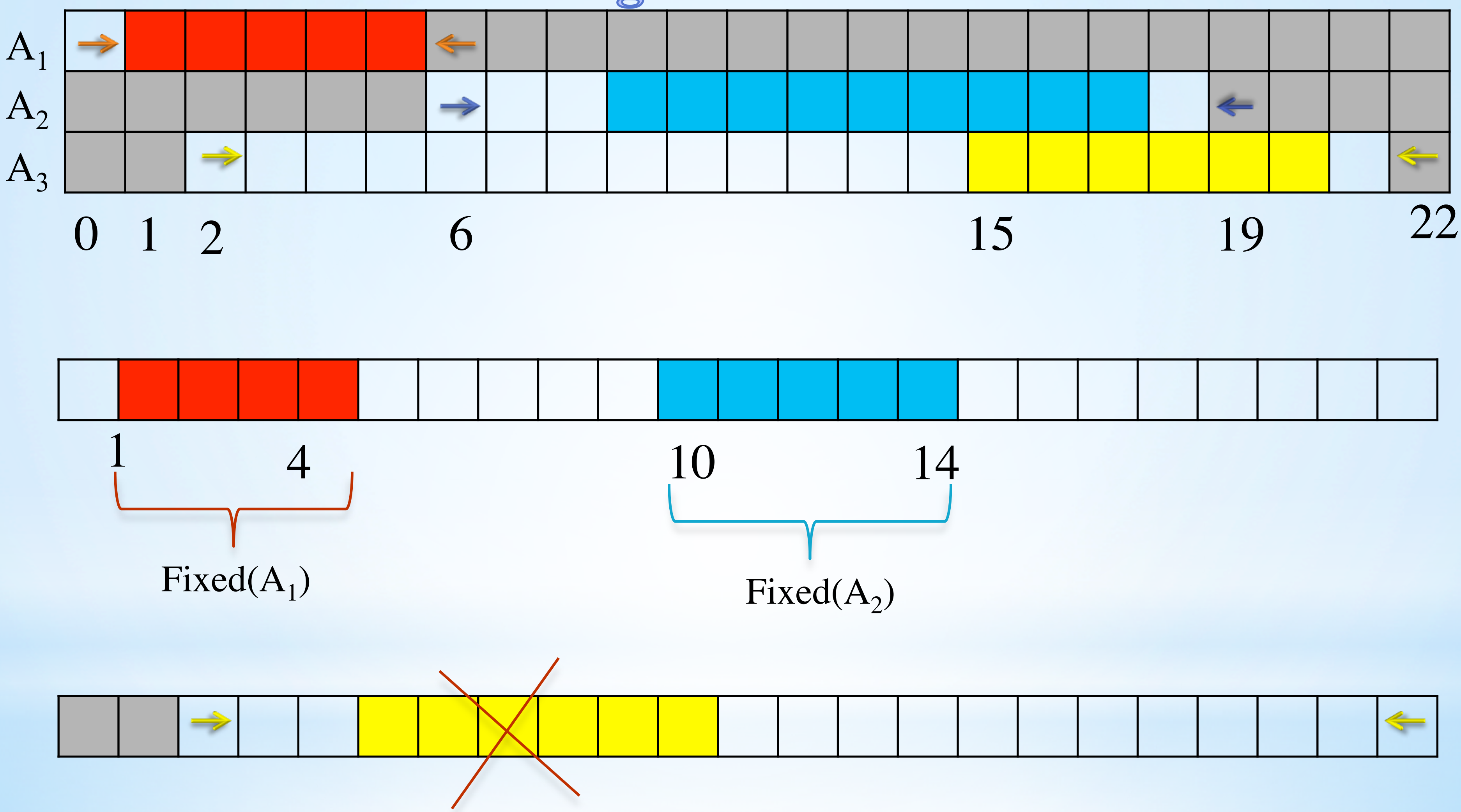






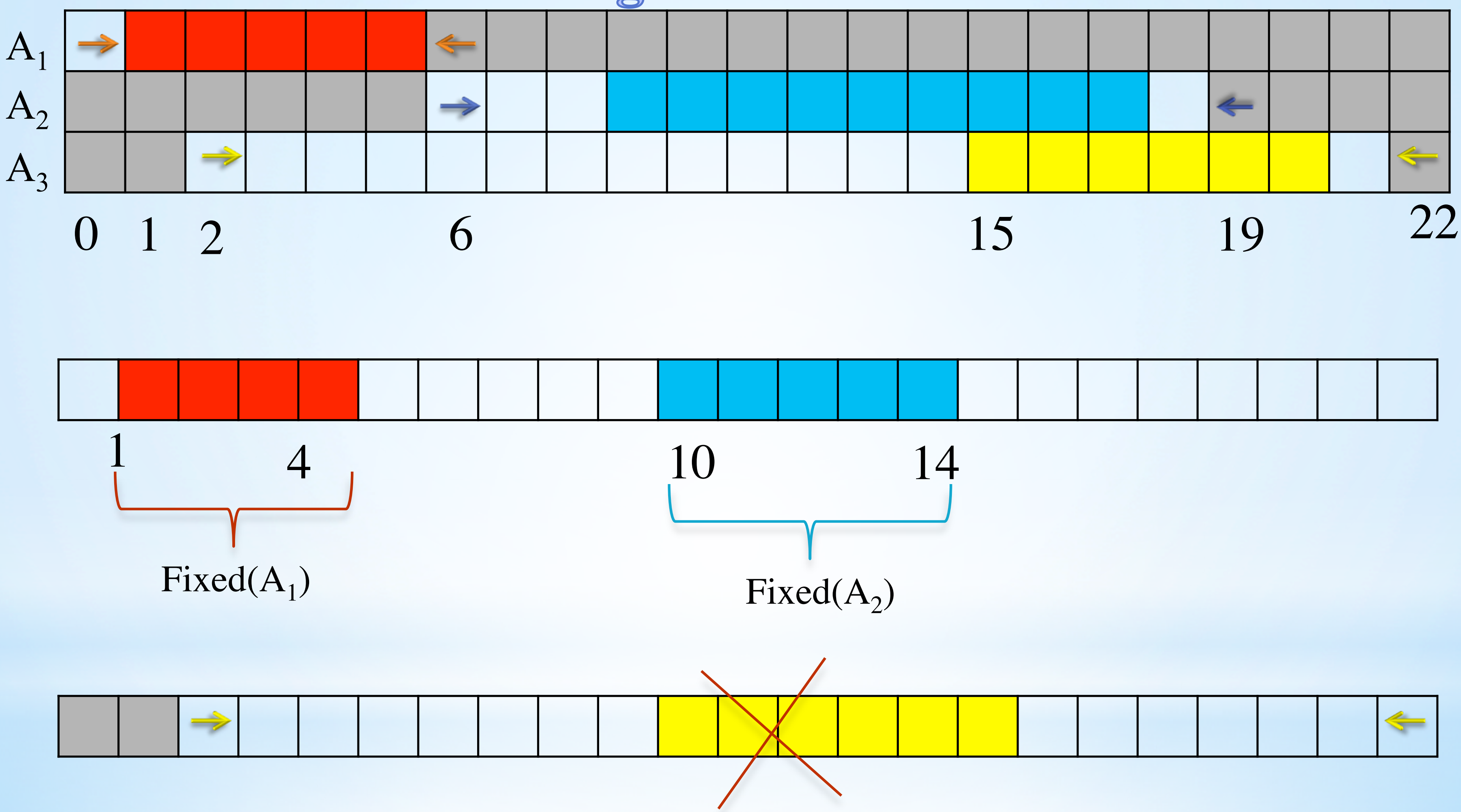
- A_3 cannot be scheduled at 2.

The strategy of our Time-Tabling algorithm



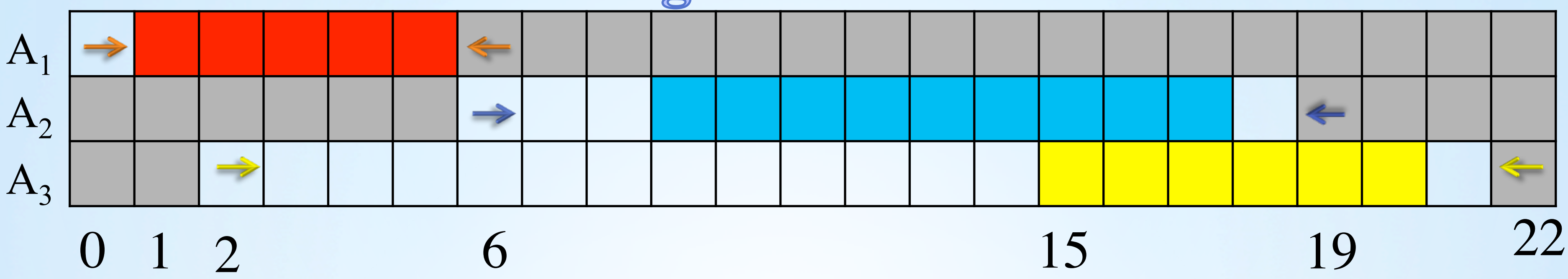
- A_3 does not fit in [5,9].

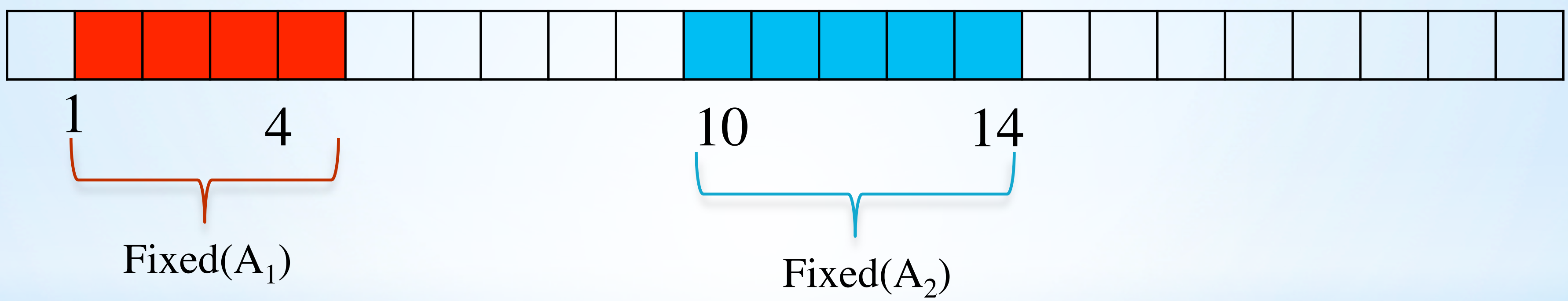
The strategy of our Time-Tabling algorithm

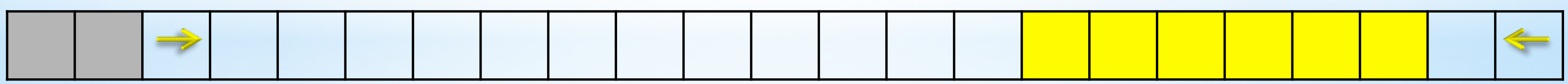


- A_3 cannot be scheduled at 10.

The strategy of our Time-Tabling algorithm

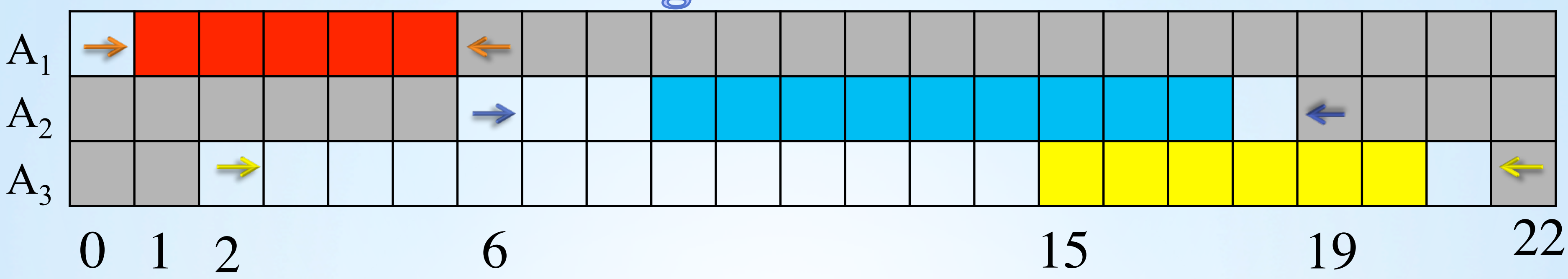


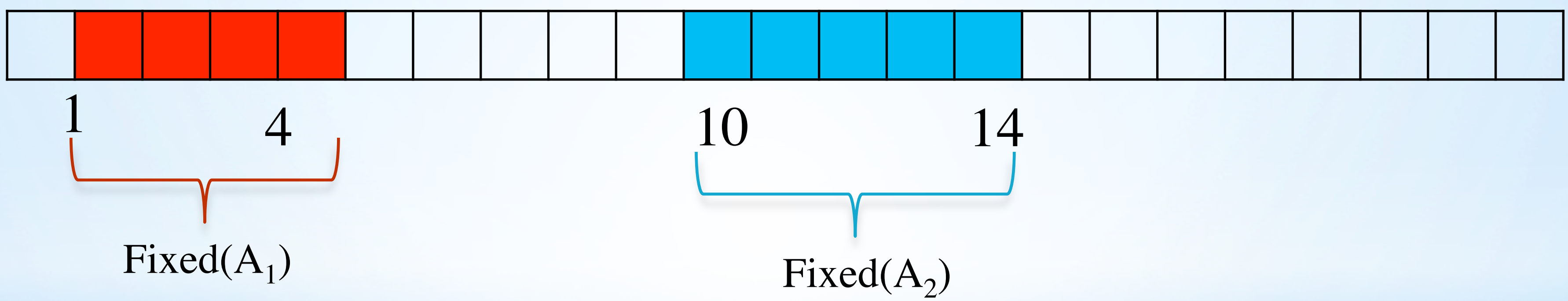


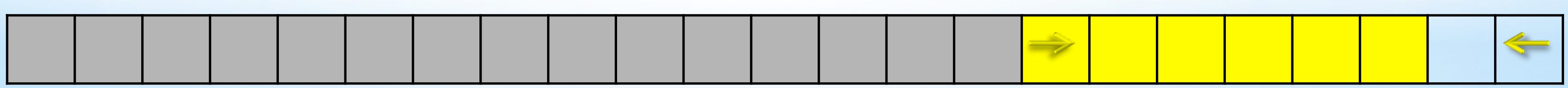


- Hence, A_3 jumps over two fixed parts.

The strategy of our Time-Tabling algorithm

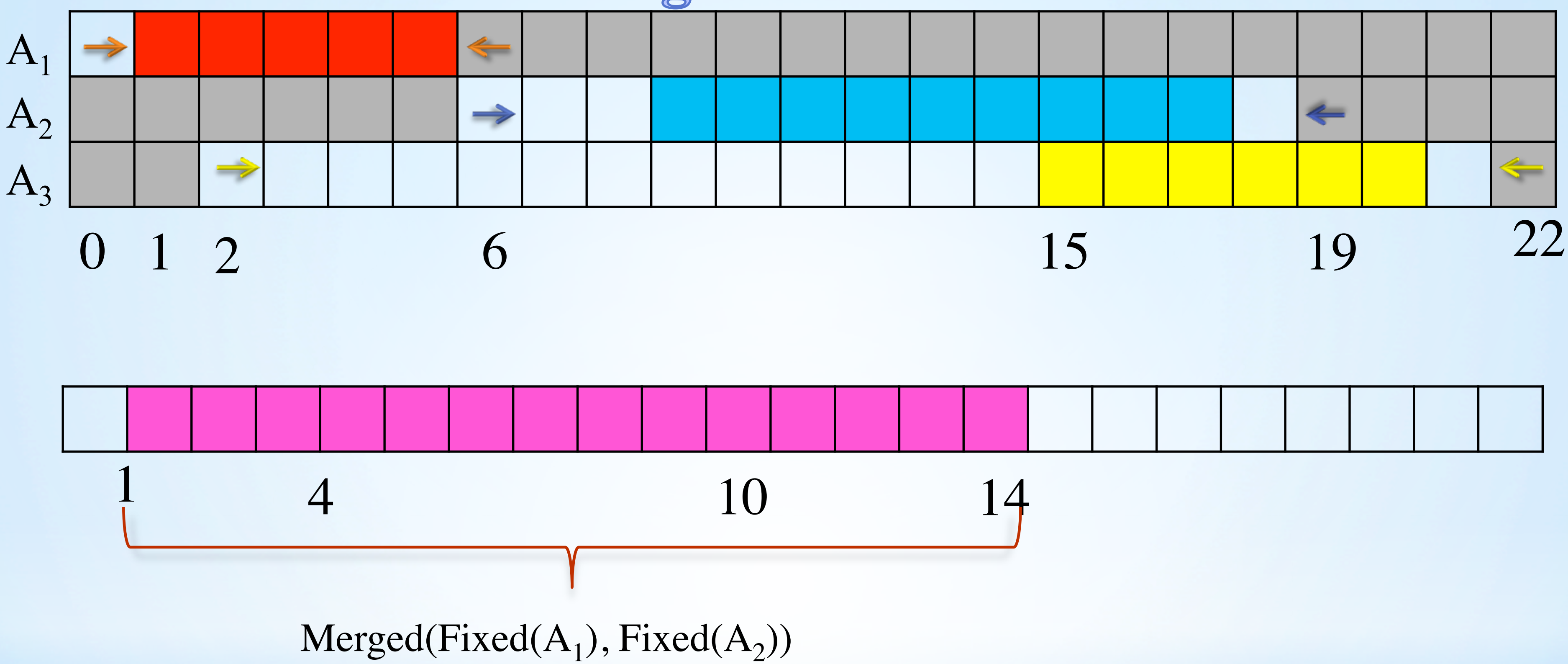






- The domain of A_3 after filtering.

The strategy of our Time-Tabling algorithm



- Since the tasks are being processed in increasing order of processing times, the next tasks will not fit in [0,14], neither. At this point, Union-Find merges the fixed parts of A_1 and A_2 to one set in constant time!

The strategy of our Time-Tabling algorithm

- Jumping over a fixed part takes constant time.
- Merging the fixed parts reduces the number of jumps.
- That is how we achieve a linear time algorithm!

Θ-Tree

- Given a set of tasks, if we schedule them at their earliest starting time, with preemption, what will the completion time of the last task be?

Θ-Tree

- Given a set of tasks, if we schedule them at their earliest starting time, with preemption, what will the completion time of the last task be?

- This value is called the "*Earliest Completion Time*" of a set of tasks.

Θ-Tree

- Given a set of tasks, if we schedule them at their earliest starting time, with preemption, what will the completion time of the last task be?

- This value is called the "*Earliest Completion Time*" of a set of tasks.

- Vilím introduced a data structure called Θ-Tree that computes the earliest completion time of a set of task Θ.

Θ-Tree

- Given a set of tasks, if we schedule them at their earliest starting time, with preemption, what will the completion time of the last task be?

- This value is called the "*Earliest Completion Time*" of a set of tasks.

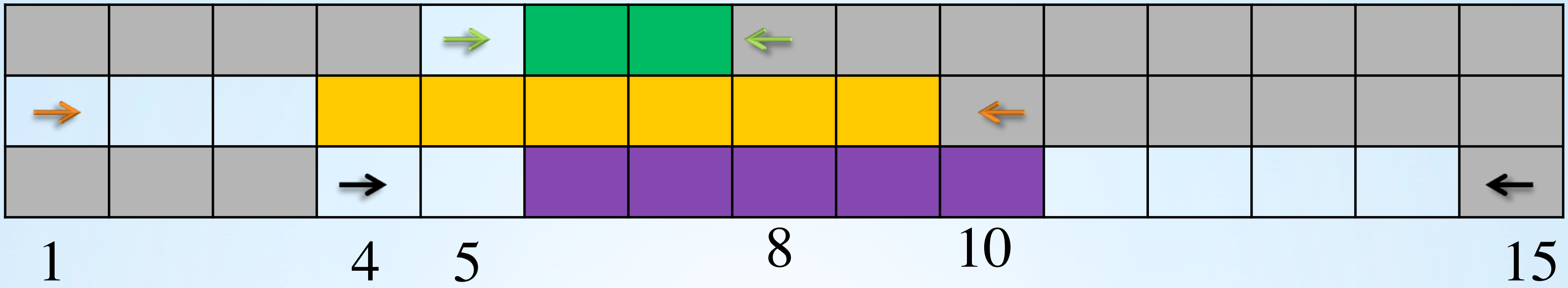
- Vilím introduced a data structure called Θ-Tree that computes the earliest completion time of a set of task Θ.

- One can insert a task into Θ or remove a task from Θ and update the computation in O(log(n)) time.

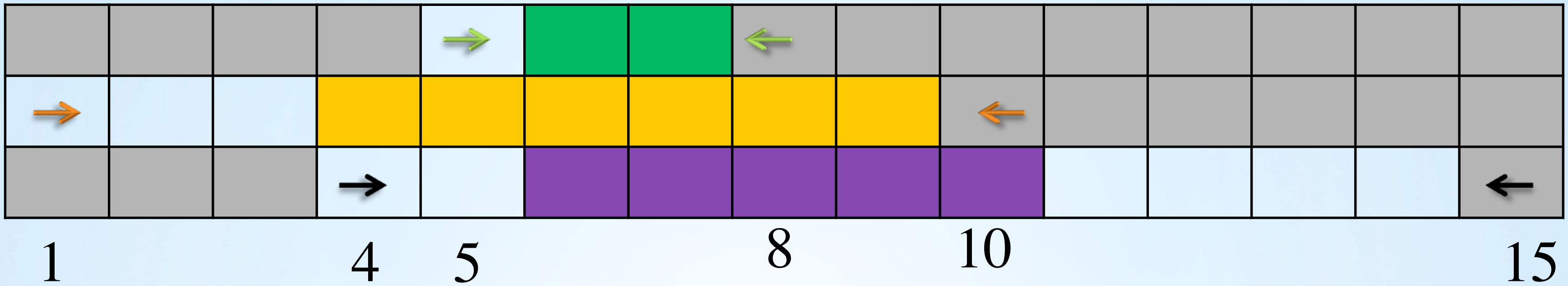
Time line

- We introduced this idea to improve upon the Θ-tree.
- What does it do?
- This data structure is initialized with an empty set of tasks $\Theta = \varnothing$.
- It is possible to add, in constant time, a task to Θ. The task will be scheduled at the earliest time as possible with preemption.
- It is possible to compute the earliest completion time of Θ in constant time, at any time.

Time line

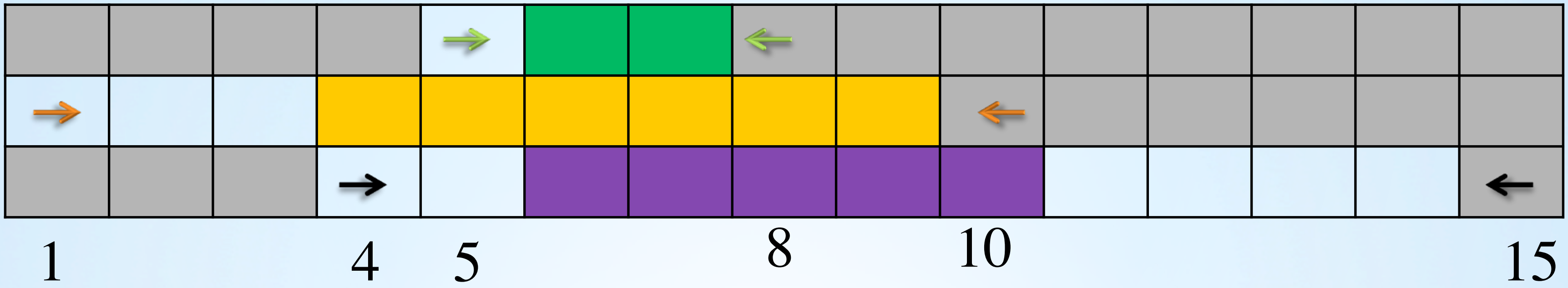


Time line



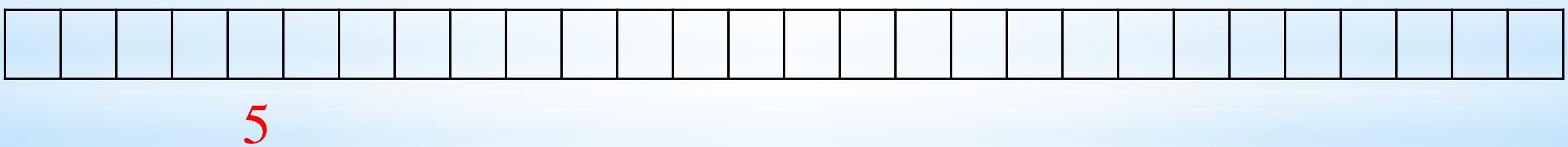
est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5

Time line



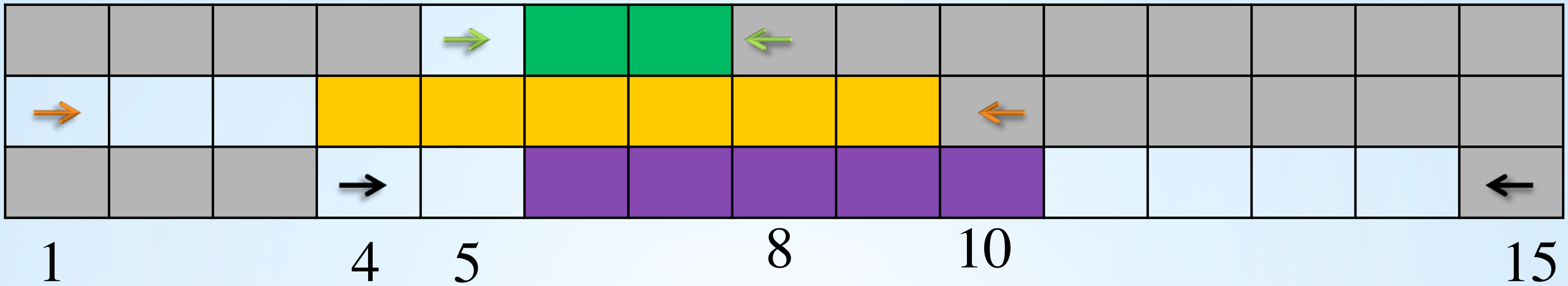
- The time line is a line with markers for important dates. The important dates are the release times of the tasks and one time point that is late enough.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



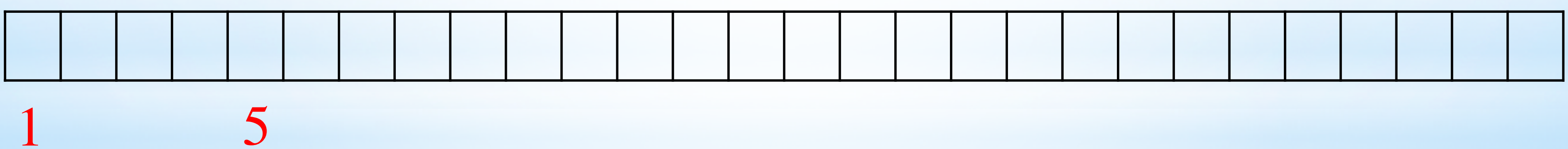
$\{\ \} \rightarrow \{\ \} \rightarrow \{5\} \rightarrow \{\ \}$

Time line



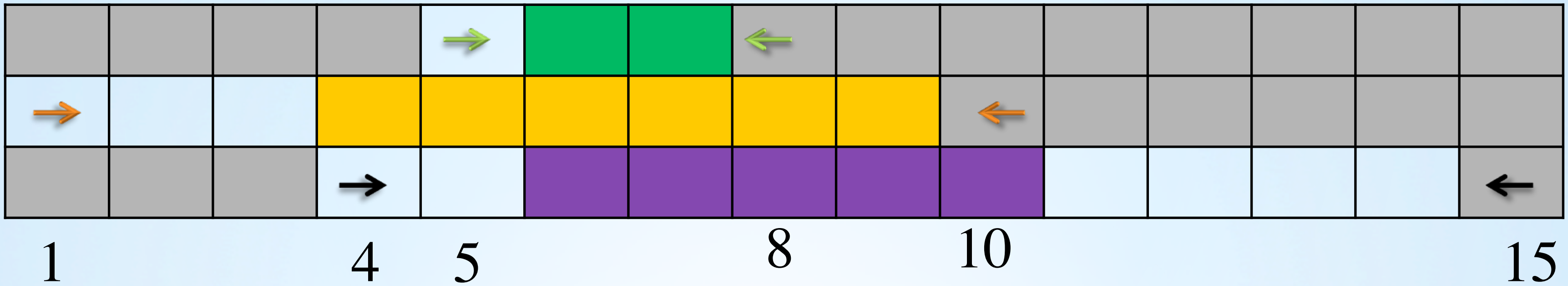
- The time line is a line with markers for important dates. The important dates are the release times of the tasks and one time point that is late enough.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



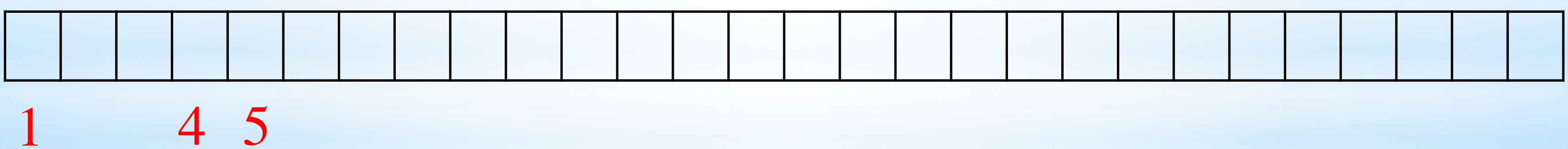
$\{1\} \rightarrow \{\ \} \rightarrow \{5\} \rightarrow \{\ \}$

Time line



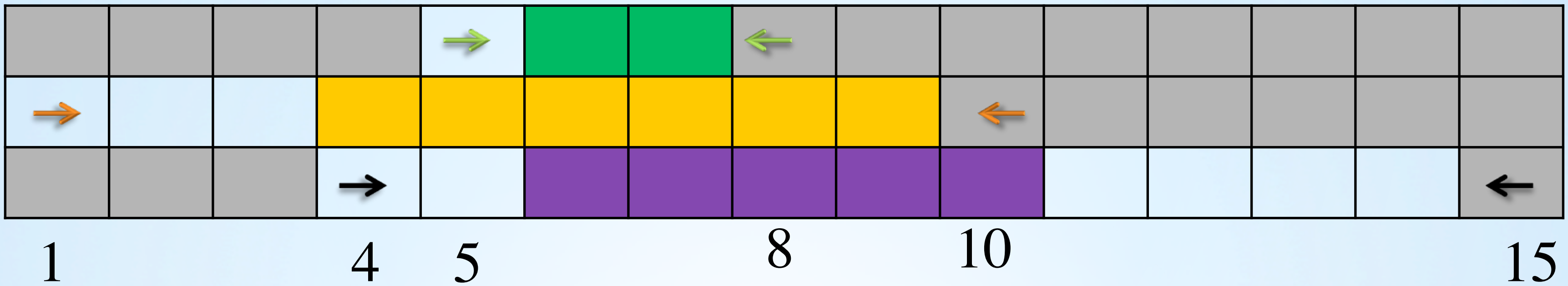
- The time line is a line with markers for important dates. The important dates are the release times of the tasks and one time point that is late enough.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



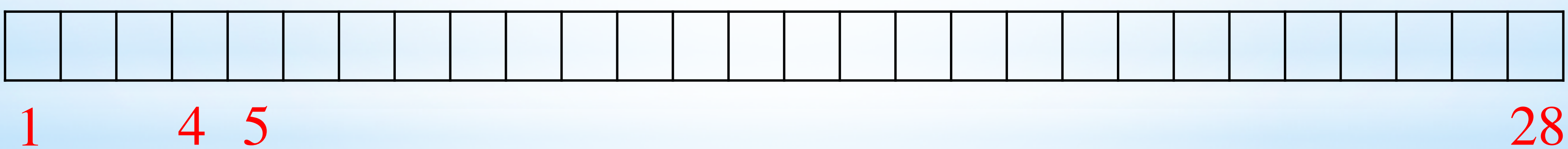
$\{1\} \rightarrow \{4\} \rightarrow \{5\} \rightarrow \{\ \}$

Time line



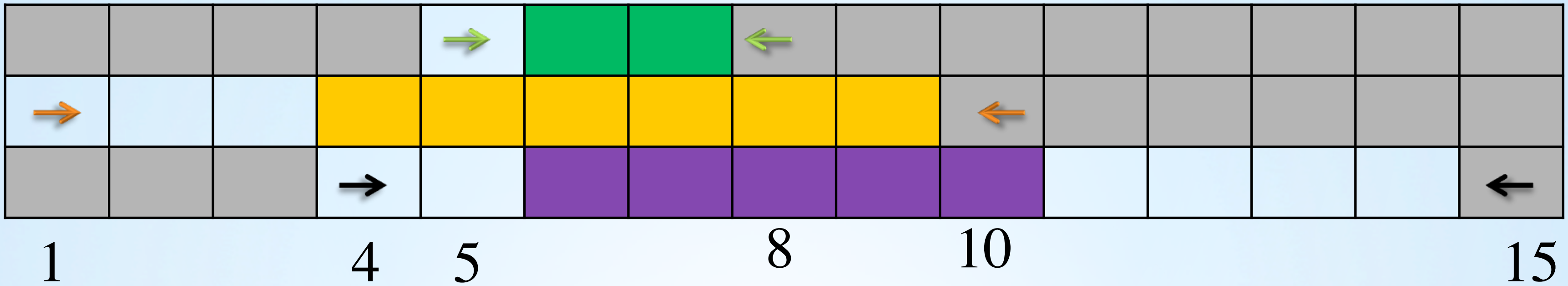
- The time line is a line with markers for important dates. The important dates are the release times of the tasks and one time point that is late enough.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



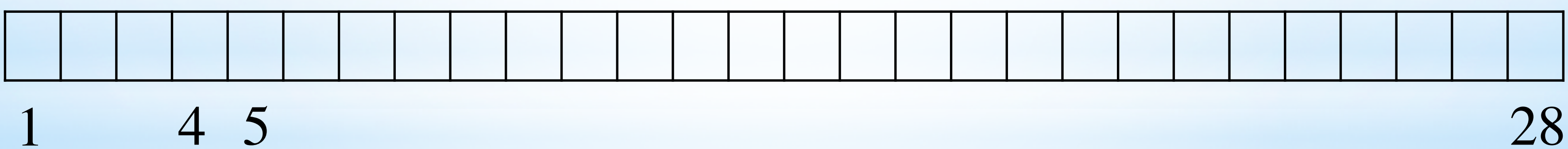
$\{1\} \rightarrow \{4\} \rightarrow \{5\} \rightarrow \{28\}$

Time line



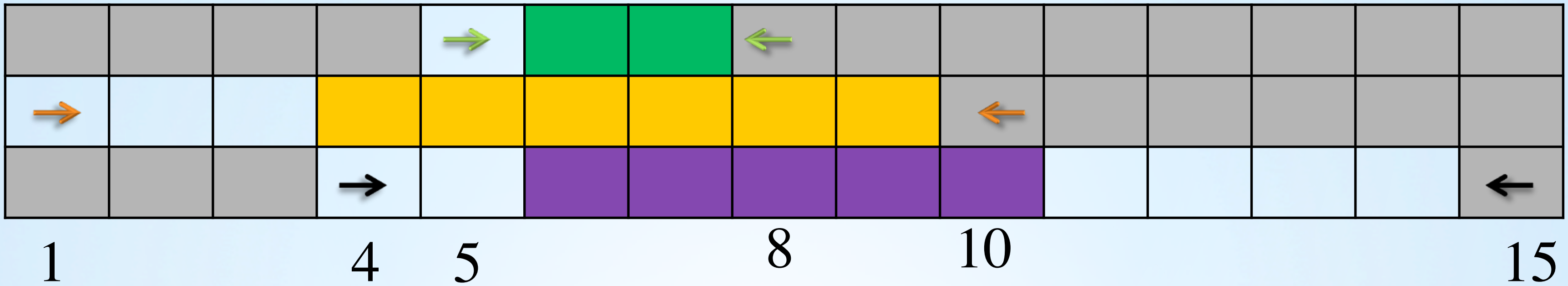
- Between each two consecutive time points, there is a capacity that denotes the amount of time that the resource is available through.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



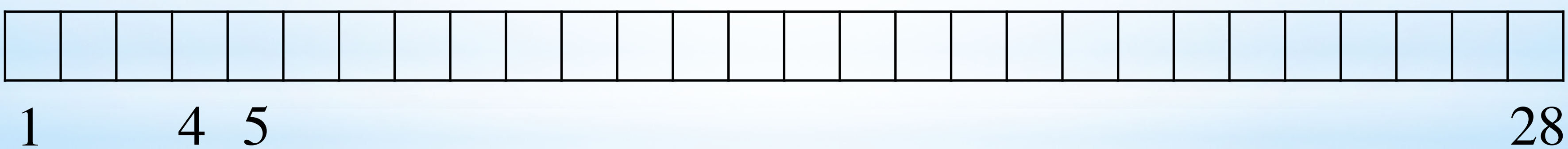
$$\{1\} \xrightarrow{3} \{4\} \xrightarrow{1} \{5\} \xrightarrow{23} \{28\}$$

Time line



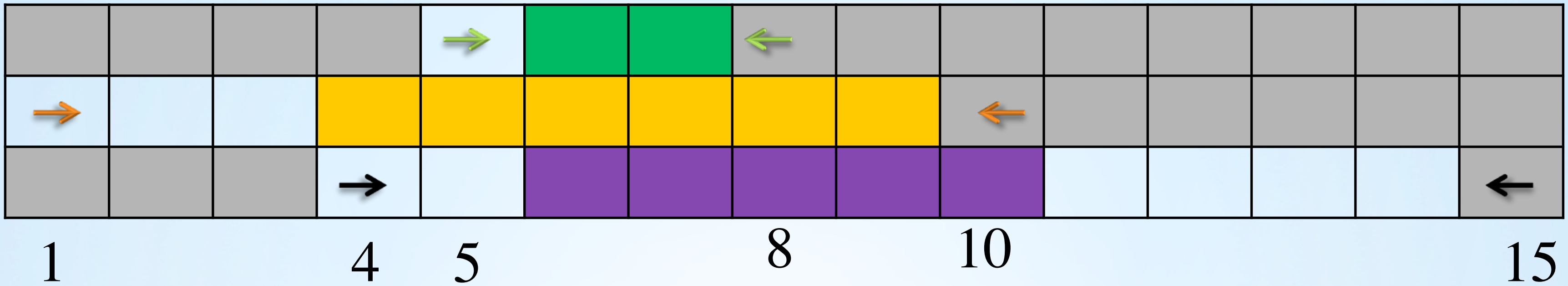
- Initially, the capacities are equal to the difference between the consecutive time points.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



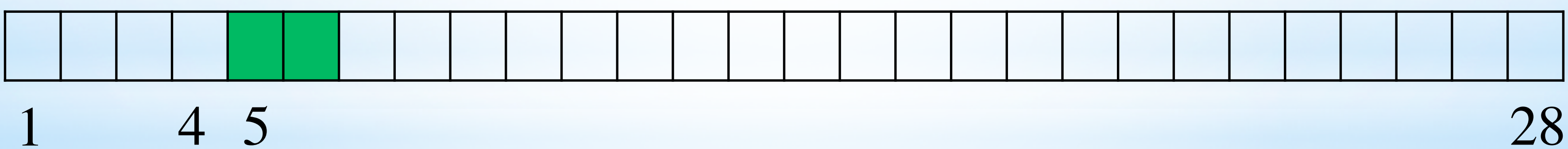
$$\{1\} \xrightarrow{3} \{4\} \xrightarrow{1} \{5\} \xrightarrow{23} \{28\}$$

Time line



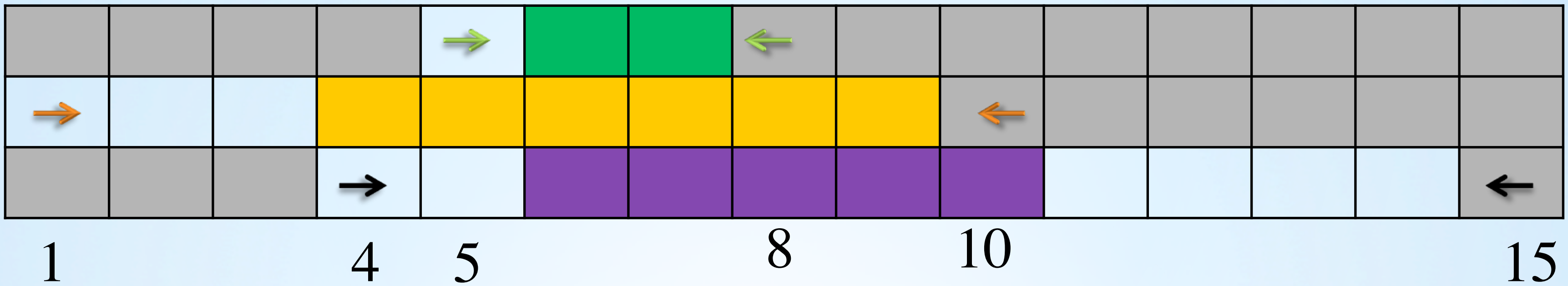
- We schedule the tasks, one by one. After scheduling, the free times will reduce.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



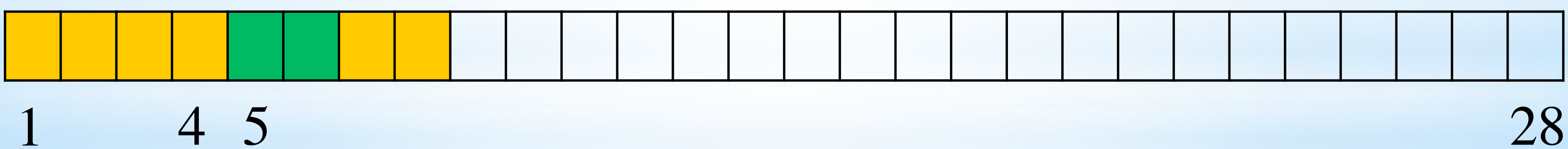
$$\{1\} \xrightarrow{3} \{4\} \xrightarrow{1} \{5\} \xrightarrow{21} \{28\}$$

Time line



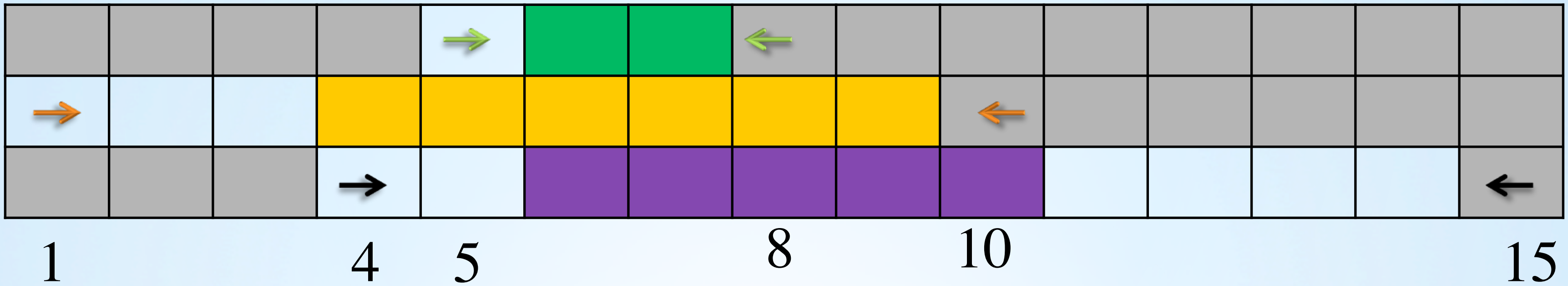
- We schedule the tasks, one by one. After scheduling, the free times will reduce.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



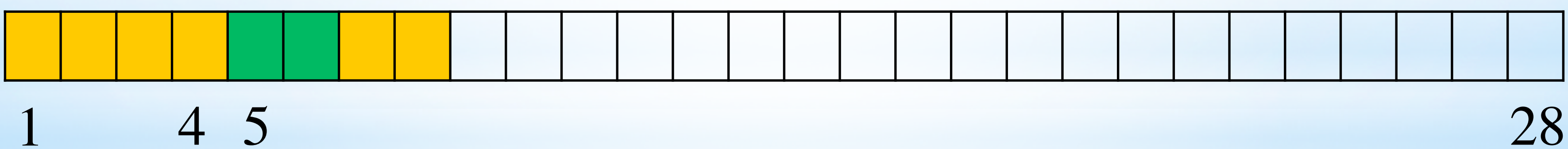
$$\{1\} \xrightarrow{0} \{4\} \xrightarrow{0} \{5\} \xrightarrow{19} \{28\}$$

Time line



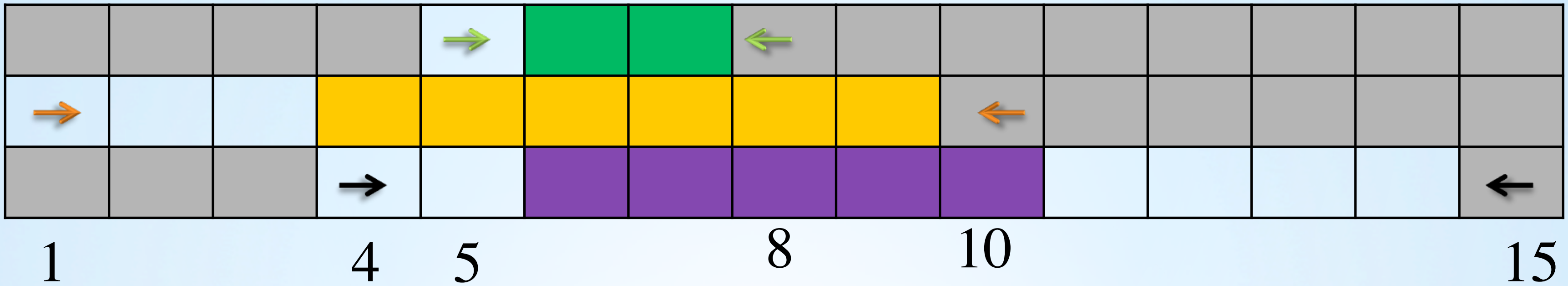
- Once a capacity equals null, the corresponding time points will be merged by Union-Find.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



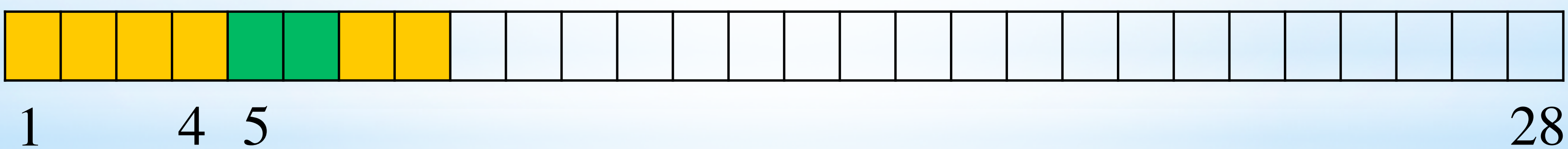
$$\{1\} \xrightarrow{0} \{4\} \xrightarrow{0} \{5\} \xrightarrow{19} \{28\}$$

Time line



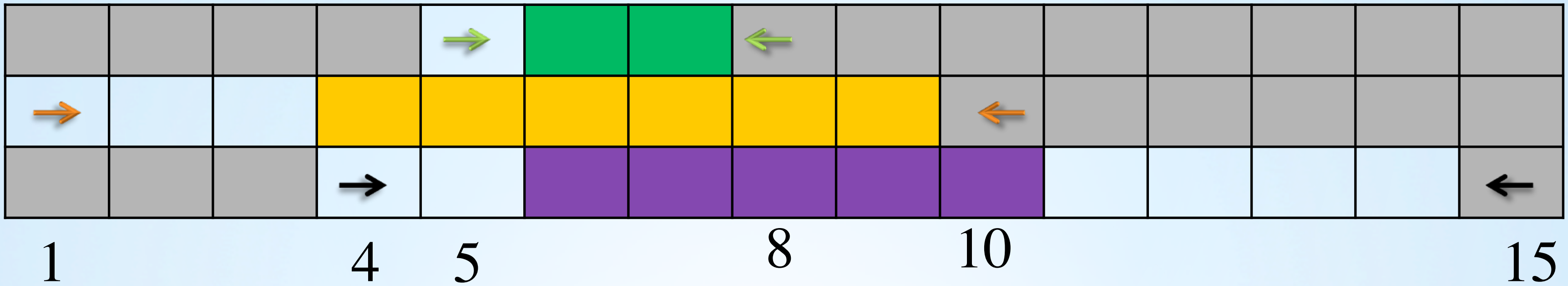
- Once a capacity equals null, the corresponding time points will be merged by Union-Find.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



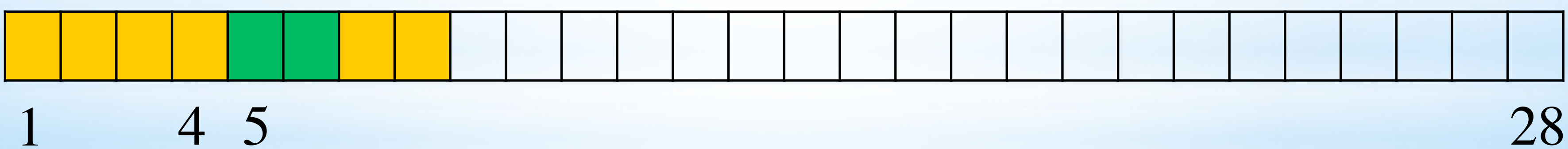
$$\{1,4,5\} \xrightarrow{19} \{28\}$$

Time line



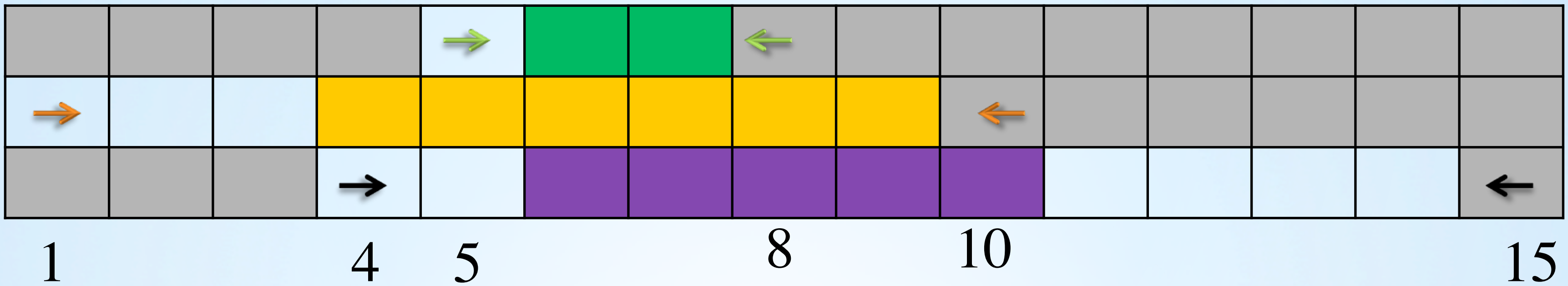
- That allows to run a linear search over the time line for periods that have free time. This search will jump over the occupied regions in constant time.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



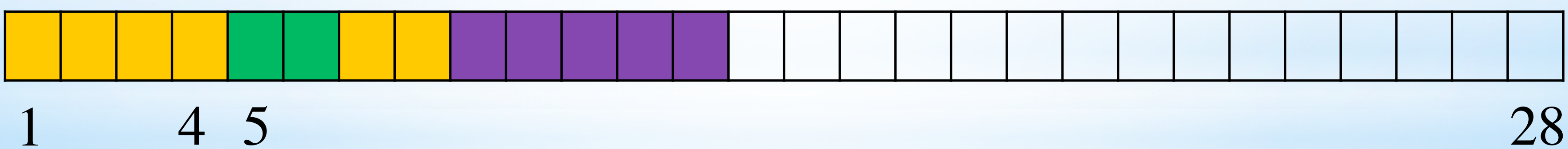
$$\{1,4,5\} \xrightarrow{19} \{28\}$$

Time line



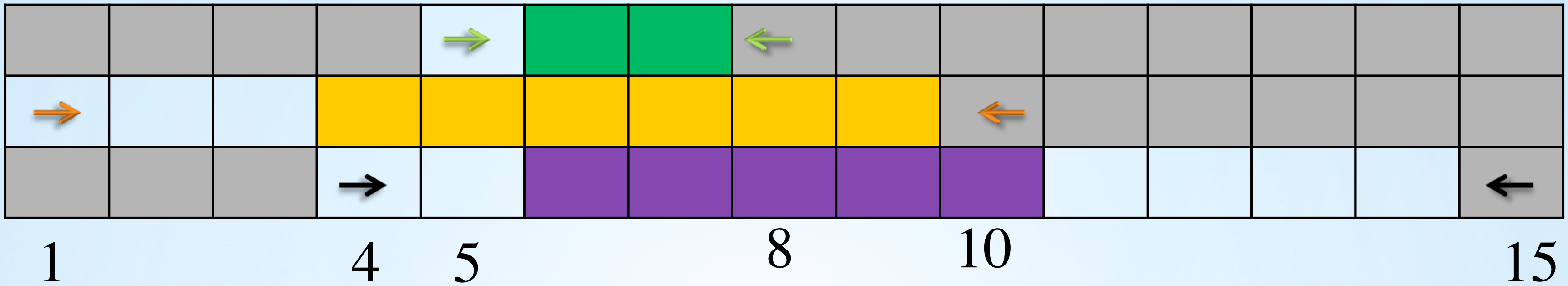
- That allows to run a linear search over the time line for periods that have free time. This search will jump over the occupied regions in constant time.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



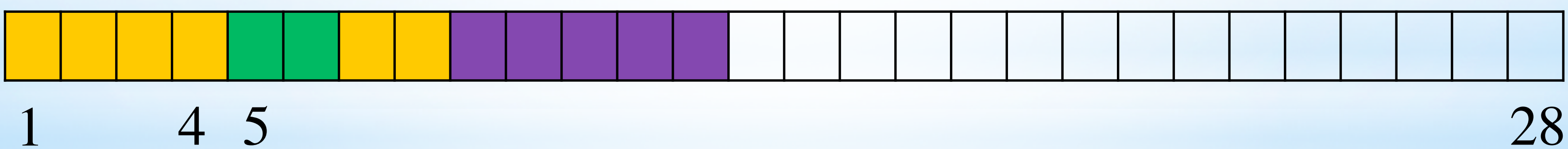
$$\{1,4,5\} \xrightarrow{14} \{28\}$$

Time line



- That allows to run a linear search over the time line for periods that have free time. This search will jump over the occupied regions in constant time.

est_i	lct_i,	p_i
5	8	2
1	10	6
4	15	5



$$\{1,4,5\} \xrightarrow{14} \{28\}$$

- The earliest completion time will be computed in constant time by 28-14 = 14!

Θ-Tree and TimeLine comparison

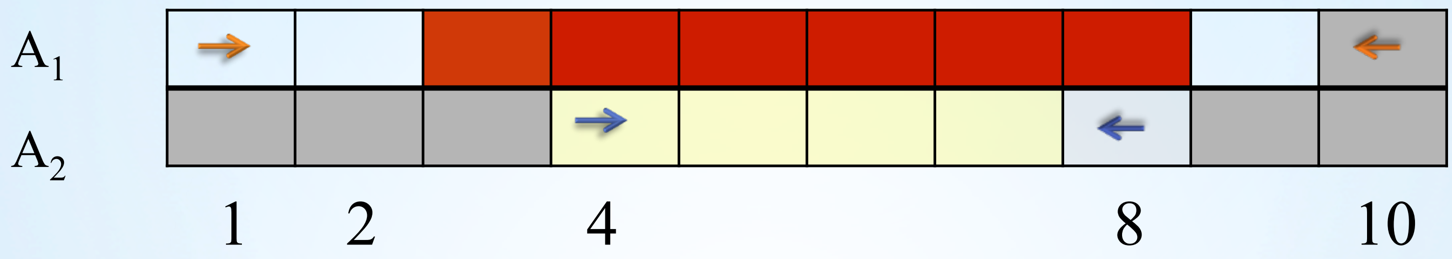
Operation	Θ-Tree	Time line
Adding a task to the schedule	O(log(n))	O(1)
Computing the earliest completion time	O(1)	O(1)
Removing a task from the schedule	O(log(n)) steps	Not supported !

Θ-Tree and TimeLine comparison

Operation	Θ-Tree	Time line
Adding a task to the schedule	O(log(n))	O(1)
Computing the earliest completion time	O(1)	O(1)
Removing a task from the schedule	O(log(n)) steps	Not supported !

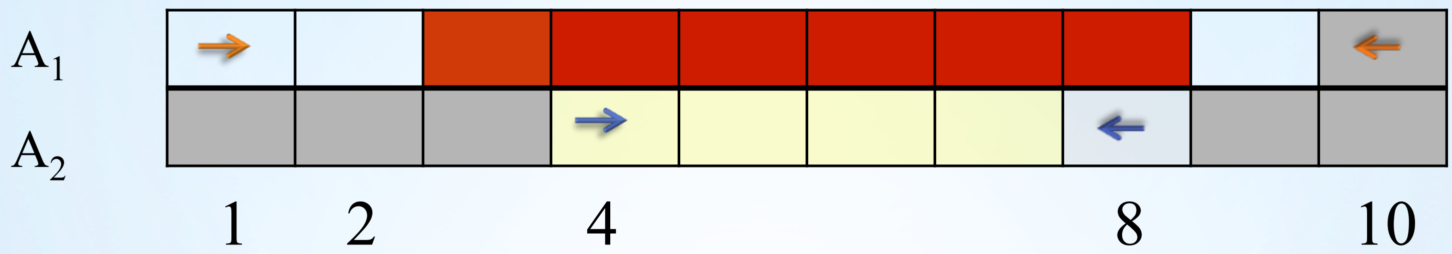
- Time line is therefore faster than a Θ–tree, but can only be used in the occasions where the removal of a task is not required.

Overload Checking



$\Theta = \{A_1, A_2\}$

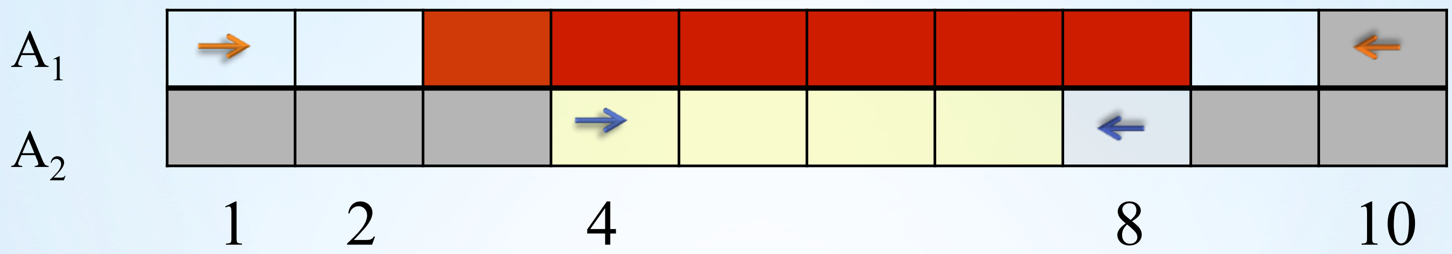
Overload Checking



$\Theta = \{A_1, A_2\}$

$d_\Theta - r_\Theta = 10-1=9 < p_\Theta = 6+4$

Overload Checking



Θ = {A_1, A_2}

$d_\Theta - r_\Theta = 10\text{-}1 = 9 < p_\Theta = 6+4$

$\Rightarrow$ There is not a valid schedule for Ω.

Overload Checking

- Overload Checking is not a filtering algorithm, as it does not propagate.

- It triggers a backtrack if the test fails.

Overload Checking

- Overload Checking is not a filtering algorithm, as it does not propagate.
- It triggers a backtrack if the test fails.

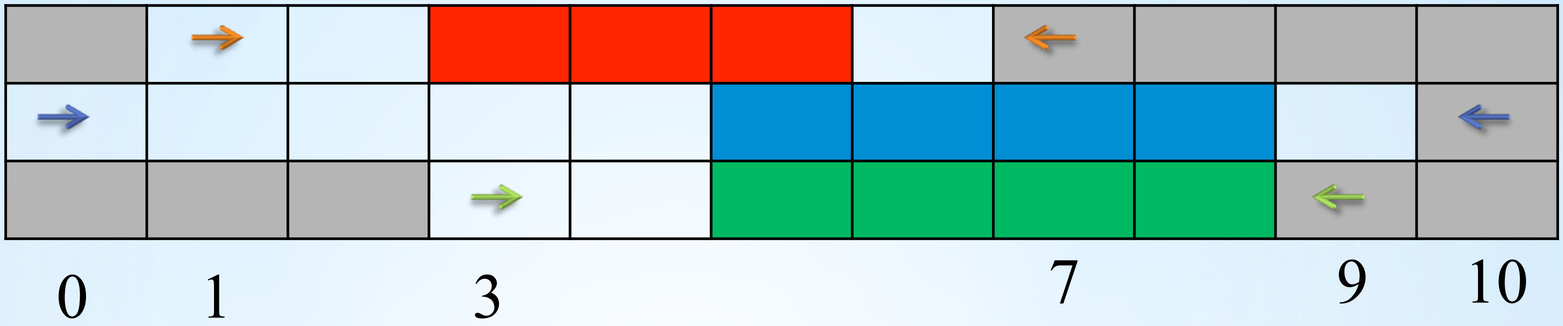
```
Θ := ∅;
for j ∈ T in non-decreasing order of lct_j do begin
   Θ := Θ ∪ {j};
   if ect_Θ > lct_j then
      fail; {No solution exists}
end;
```

The strategy of our Overload check algorithm

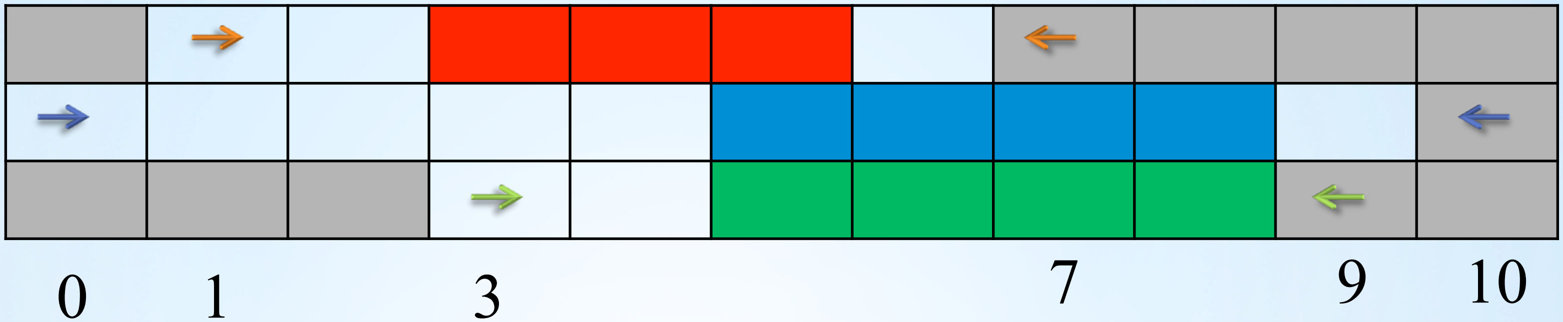
- We implement the overload check algorithm just as Vilím does. The only difference is that we simply substitute the Θ-tree with the time line.

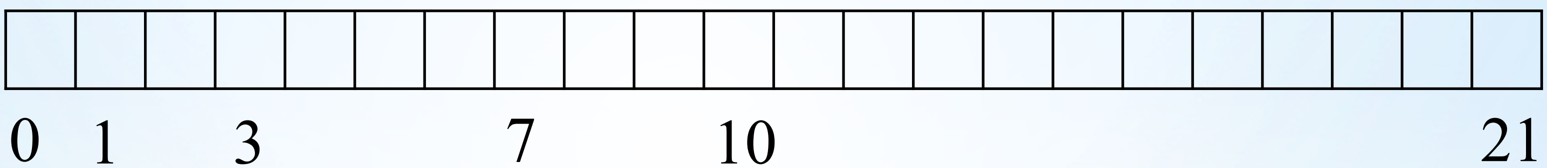
- Overload Check with implementing time line runs in linear time!

Example



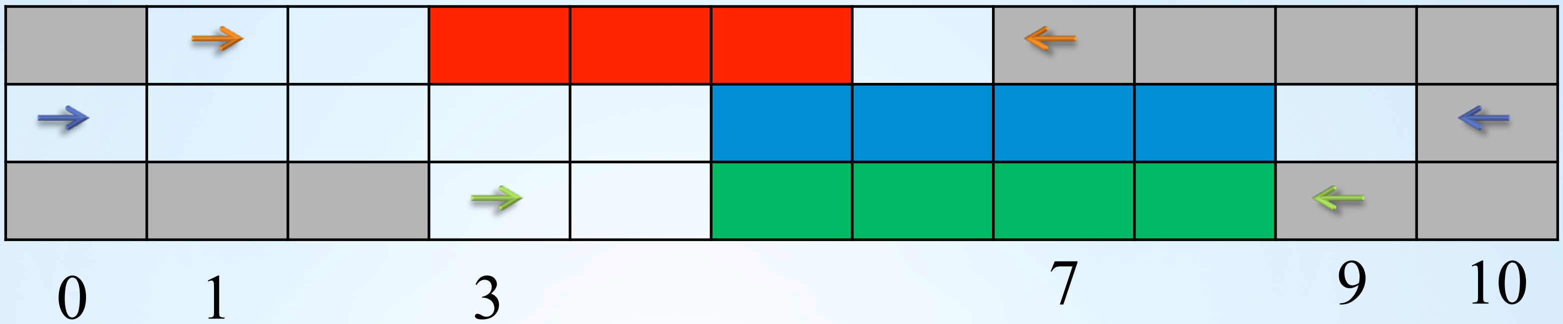
Example

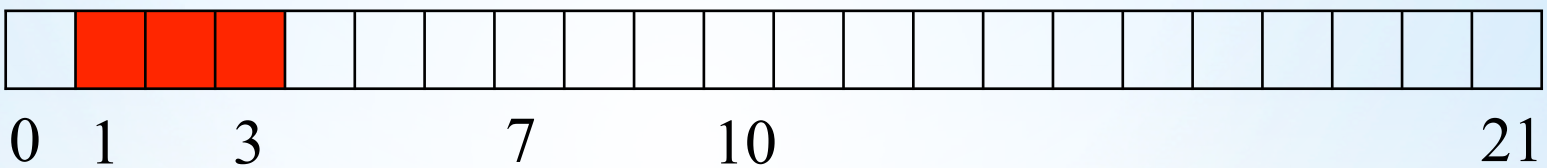




$$\{0\} \xrightarrow{1} \{1\} \xrightarrow{2} \{3\} \xrightarrow{18} \{21\}$$

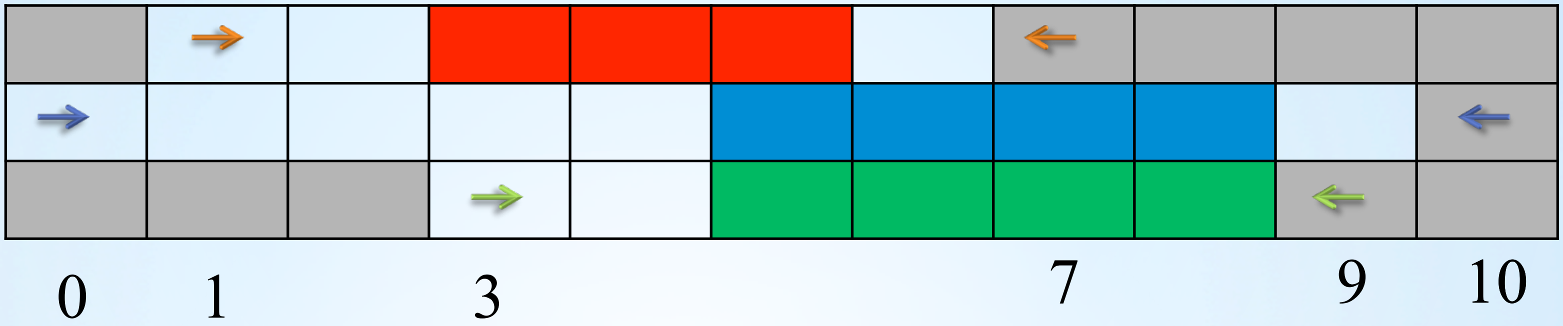
Example

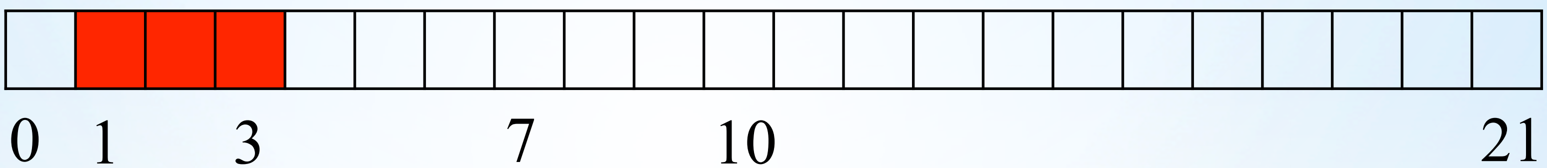




$$\{0\} \xrightarrow{1} \{1\} \xrightarrow{0} \{3\} \xrightarrow{17} \{21\}$$

Example

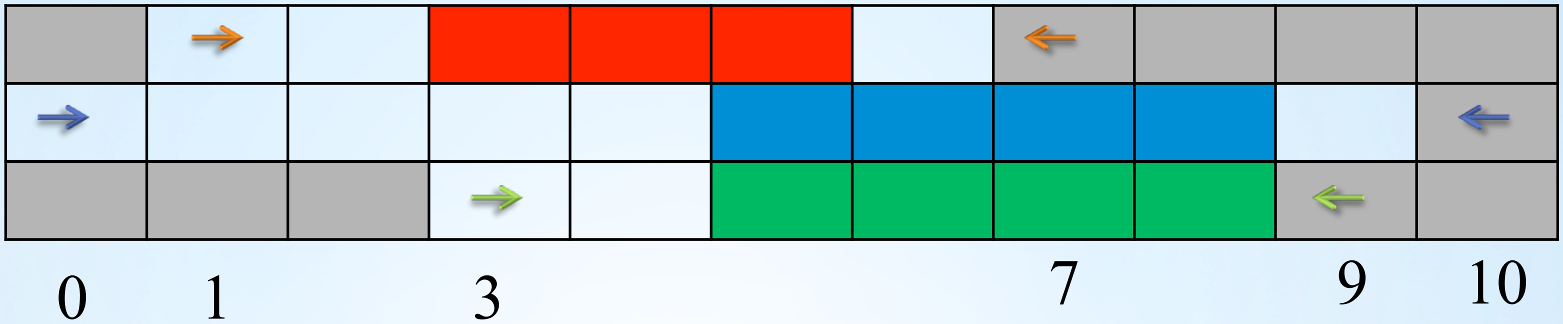


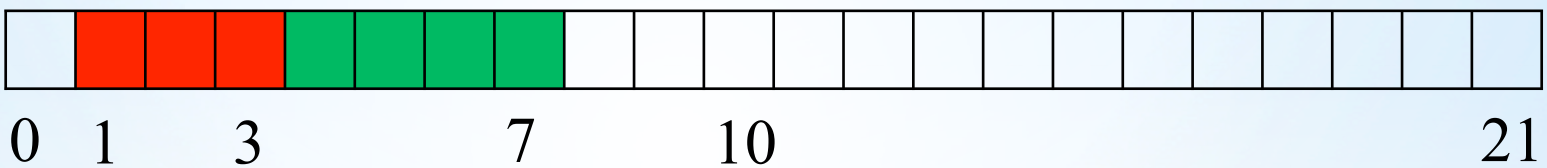


$$\{0\} \xrightarrow{1} \{1,3\} \xrightarrow{17} \{21\}$$

- Earliest completion time of Θ = 21 -17= 4.

Example

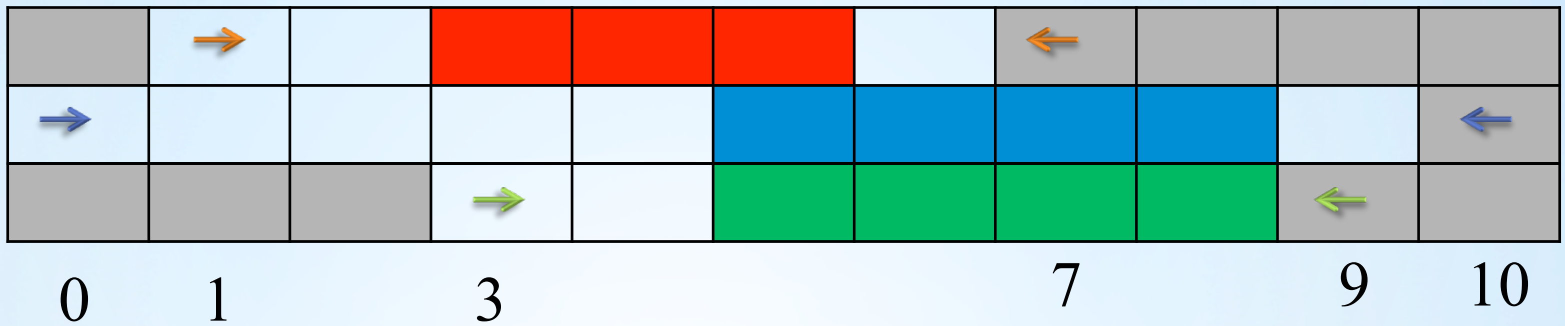


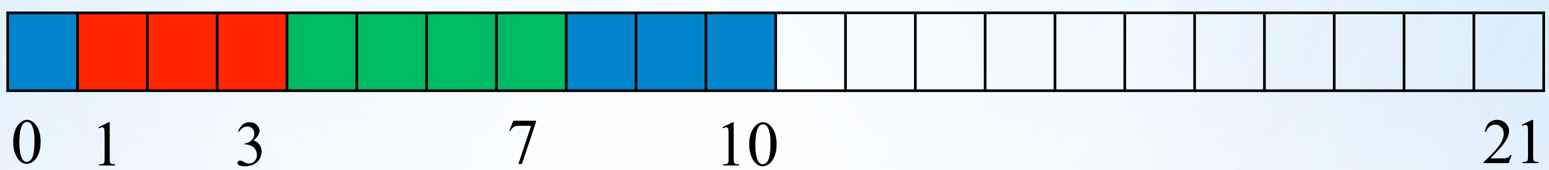


$$\{0\} \xrightarrow{1} \{1,3\} \xrightarrow{13} \{21\}$$

- Earliest completion time of Θ = 21 -13= 8.

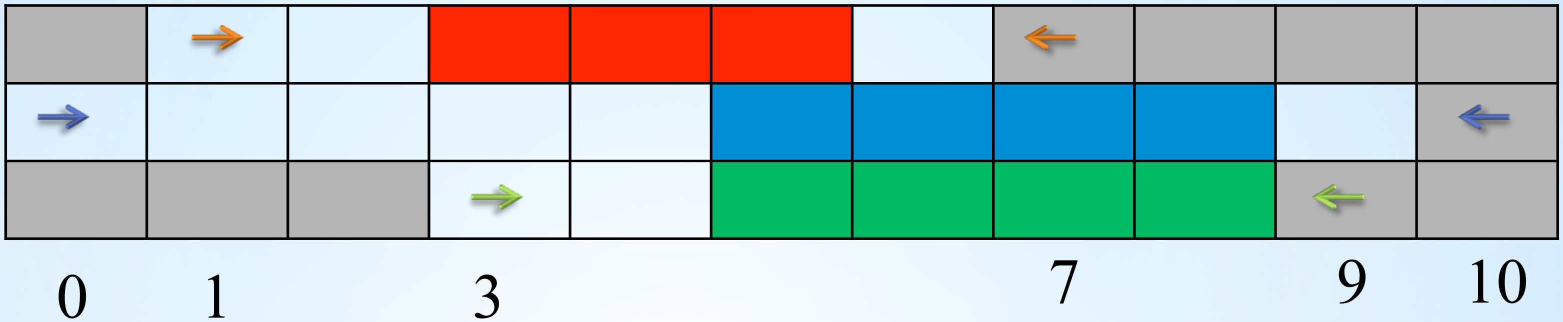
Example

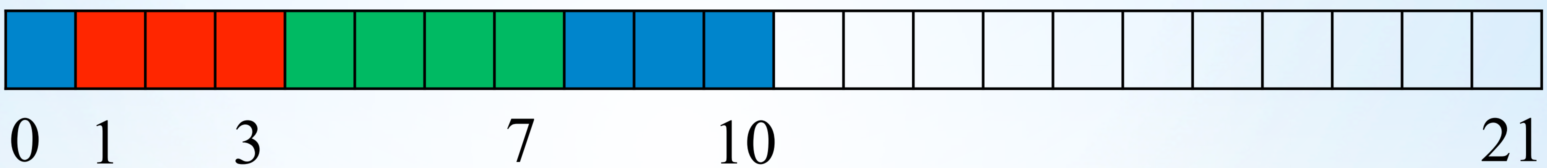




$$\{0\} \xrightarrow{0} \{1,3\} \xrightarrow{10} \{21\}$$

Example

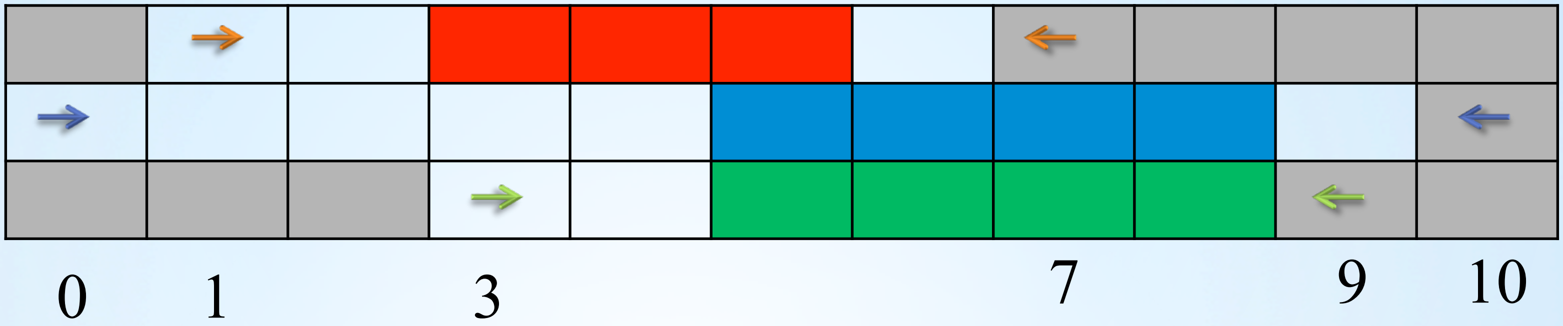


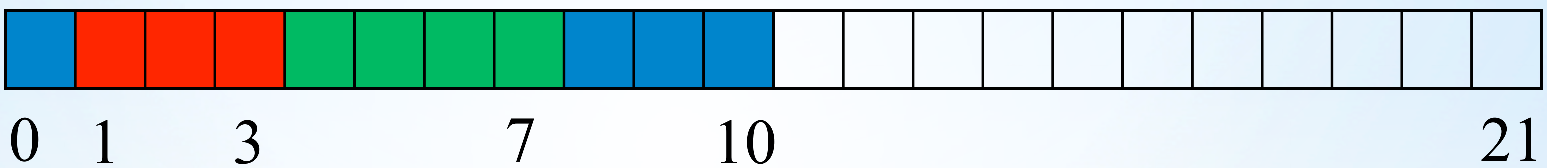


$$\{0,1,3\} \xrightarrow{10} \{21\}$$

- Earliest completion time of Θ = 21 -10=11 > 10.

Example





$$\{0,1,3\} \xrightarrow{10} \{21\}$$

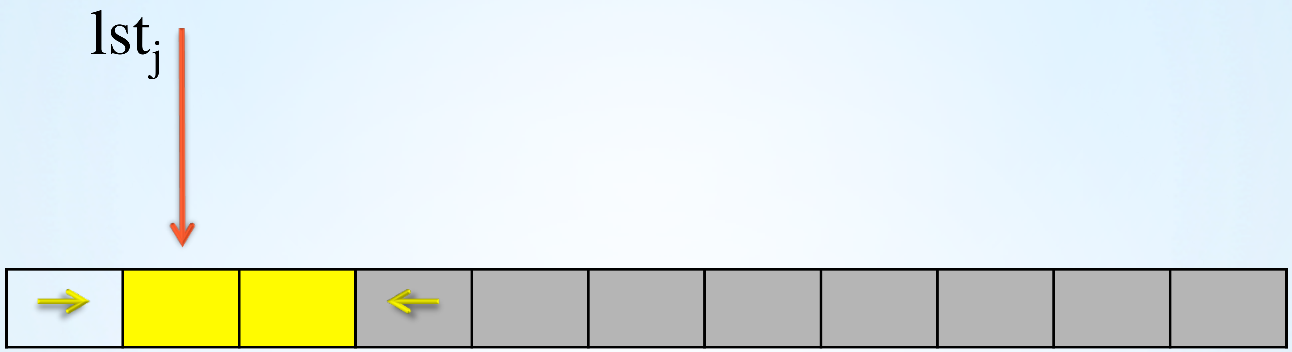
- Earliest completion time of Θ = 21 -10=11 > 10.
- Overload check fails! Thus, no valid schedule exists.

Detectable Precedences

- Let A_i and A_j be two tasks. If $ect_i > lst_j$, the precedence $A_j << A_i$ is called *detectable*.

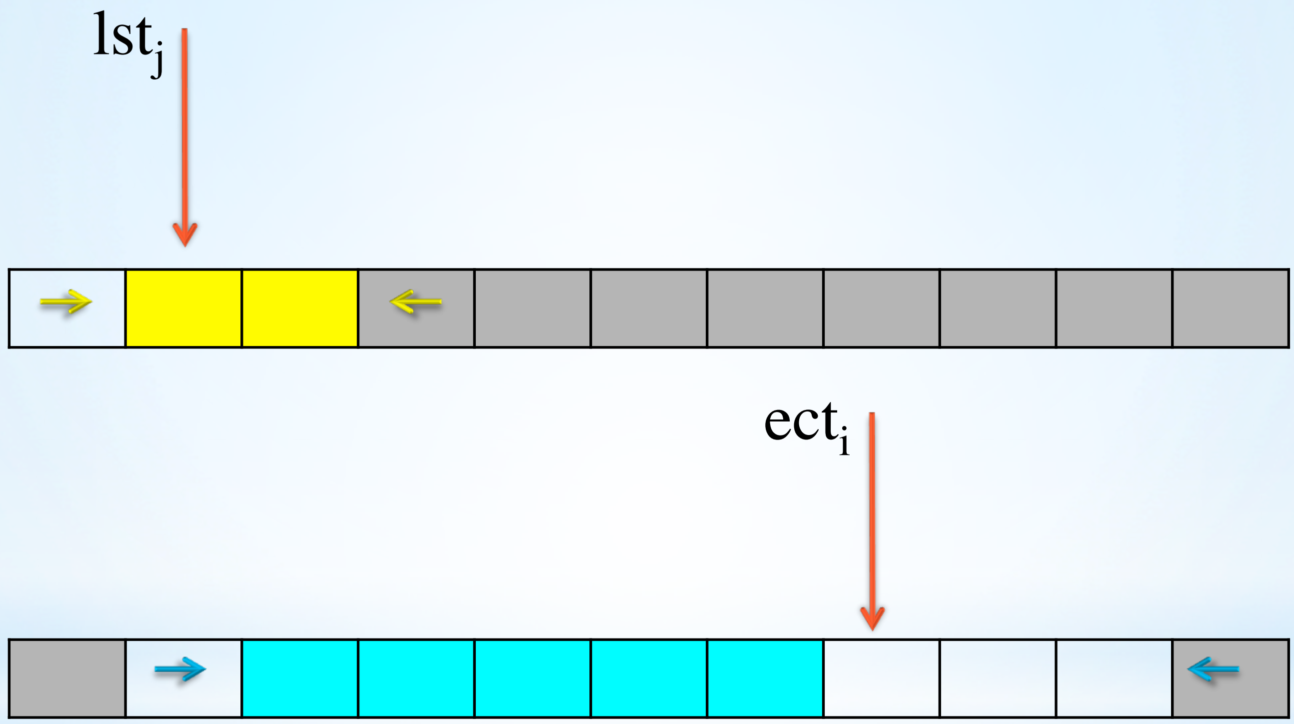
Detectable Precedences

- Let A_i and A_j be two tasks. If $ect_i > lst_j$, the precedence $A_j << A_i$ is called *detectable*.



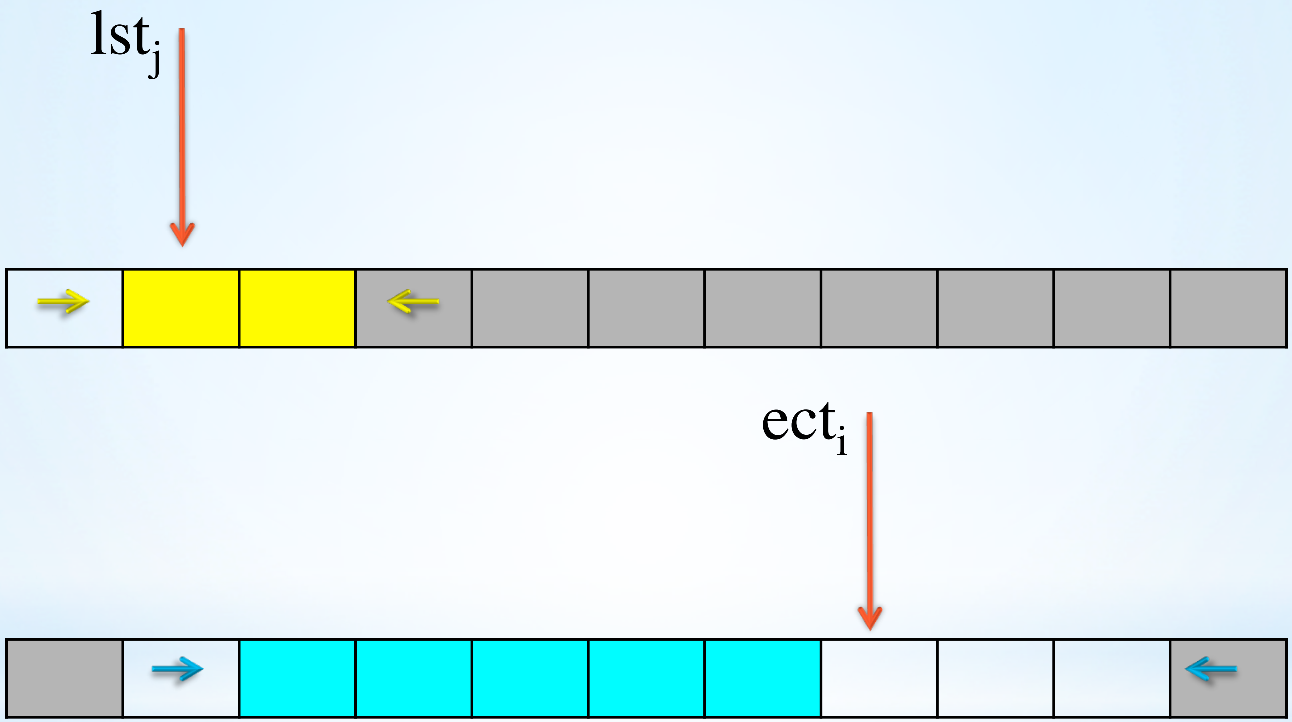
Detectable Precedences

- Let A_i and A_j be two tasks. If $ect_i > lst_j$, the precedence $A_j << A_i$ is called *detectable*.



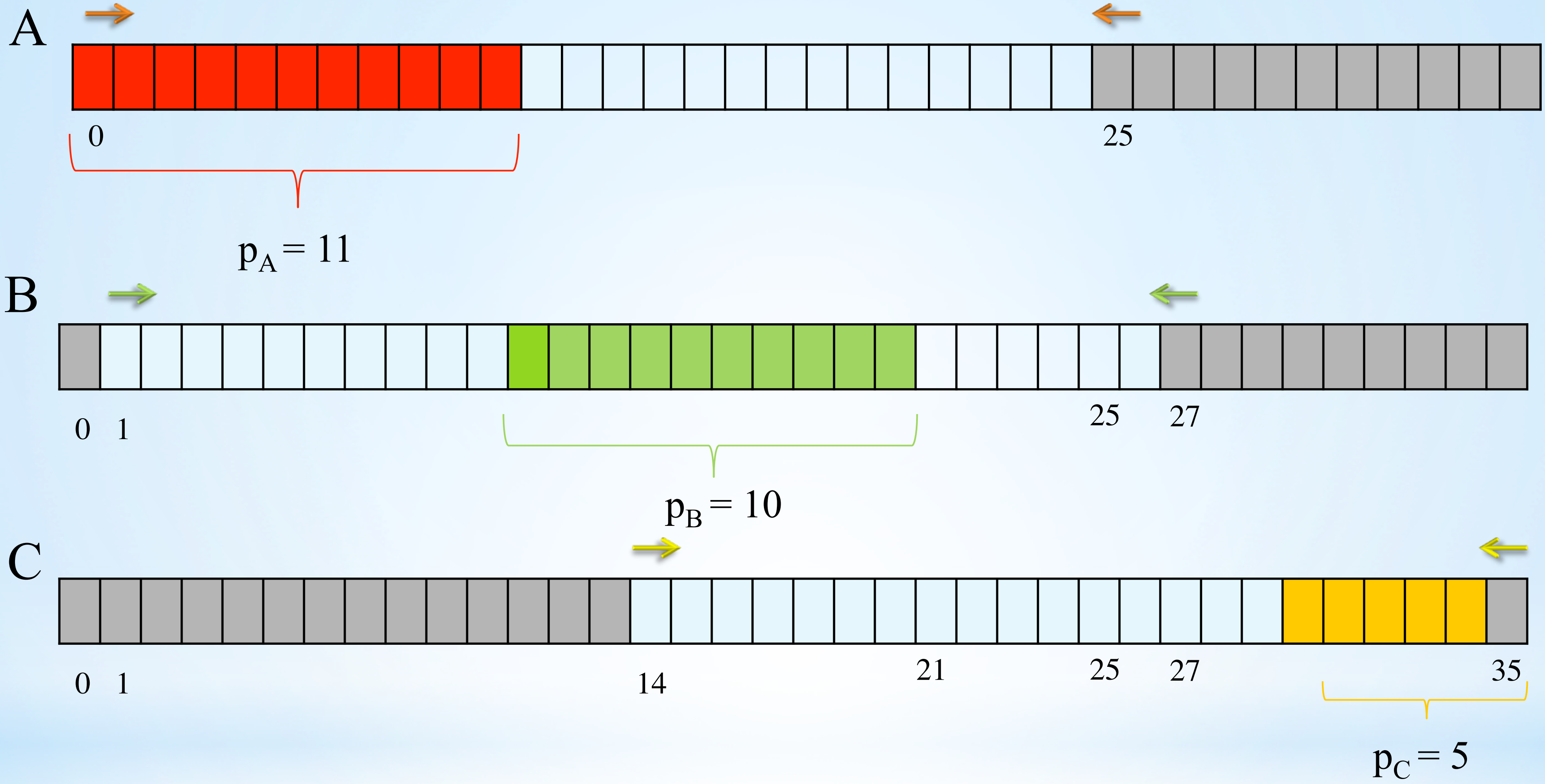
Detectable Precedences

- Let A_i and A_j be two tasks. If $ect_i > lst_j$, the precedence $A_j << A_i$ is called *detectable*.

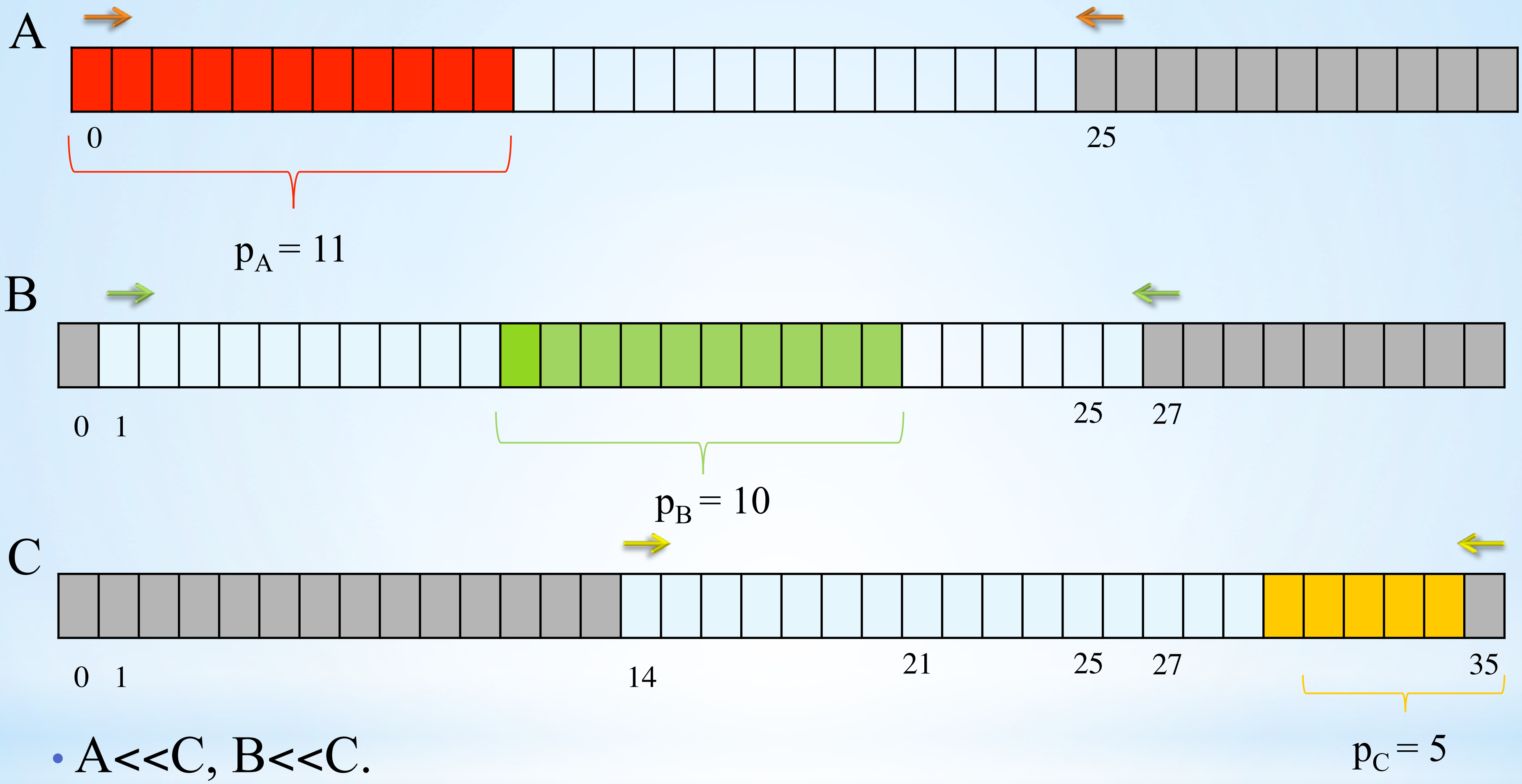


- Vilím introduced this idea and presented an algorithm in $O(n\log(n))$, using the notion of Θ-tree.

Example

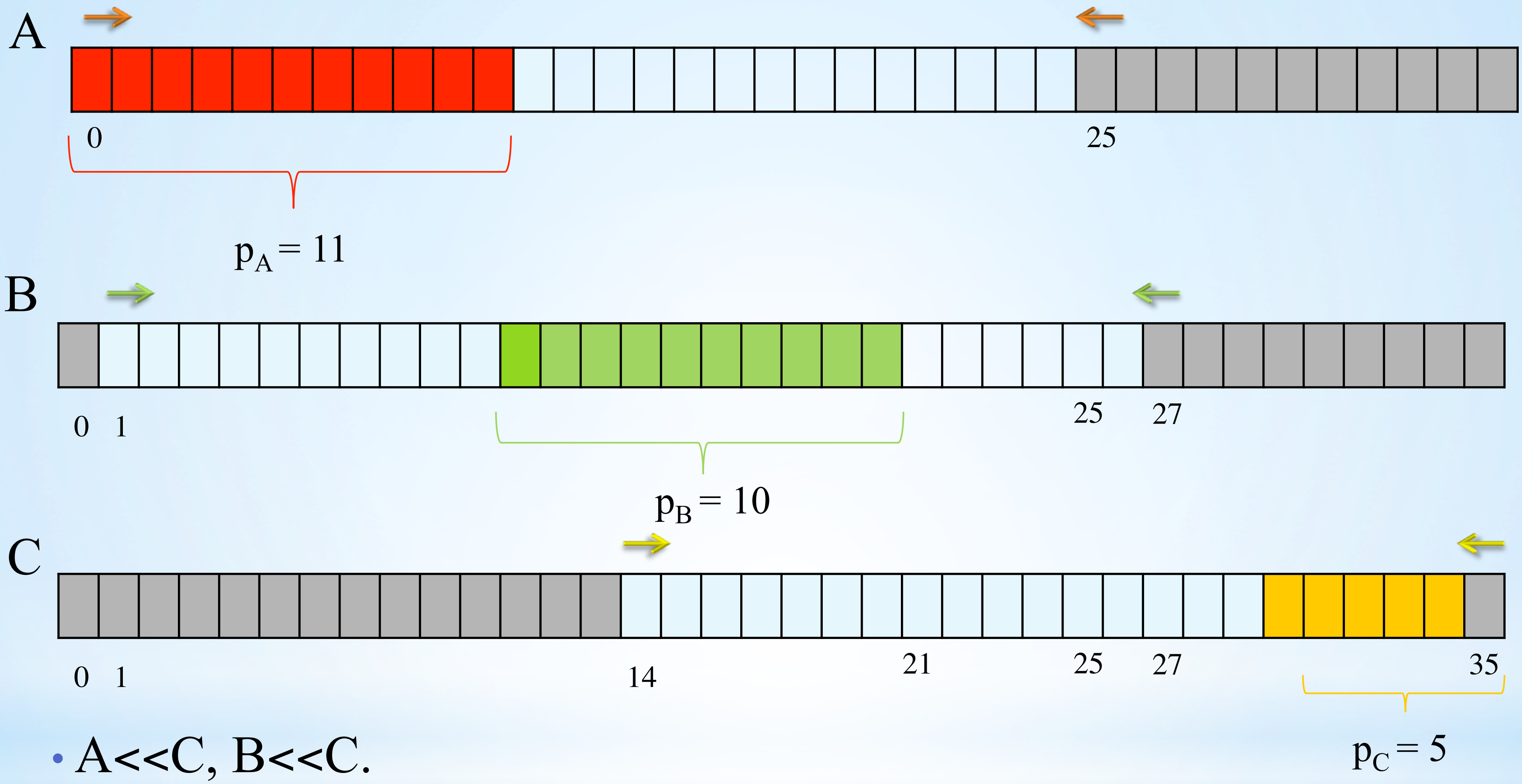


Example



- A<<C, B<<C.

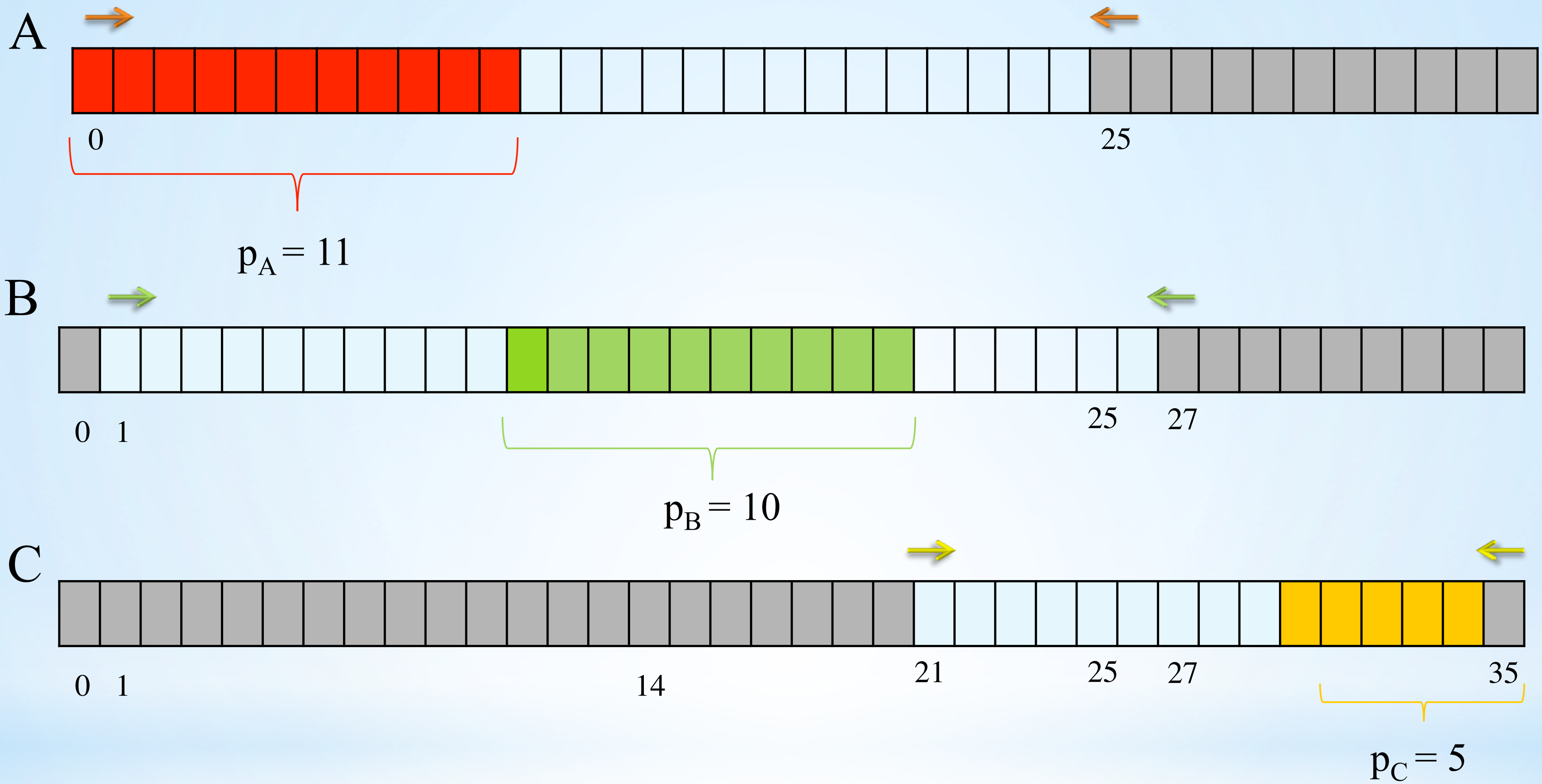
Example



- A<<C, B<<C.

- Since {A , B } << C, the domain of C will be filtered to $est_C \geq est_A + p_A + p_B = 21$.

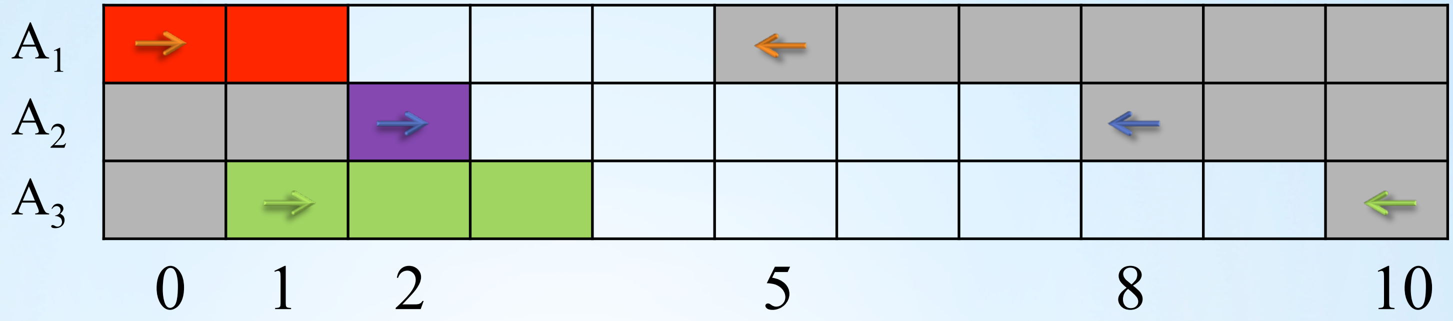
Example



- The domain of C after filtering.

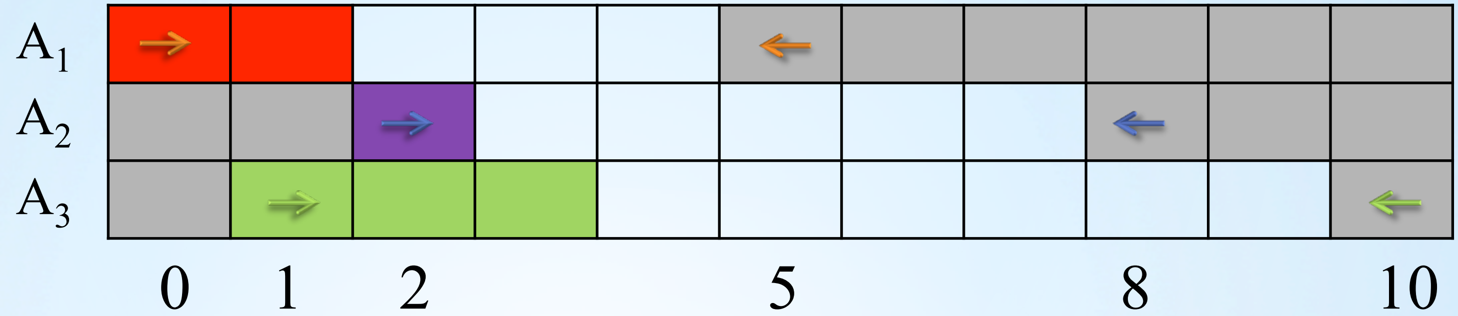
Detectable Precedences

- The tasks sorted by earliest completion times

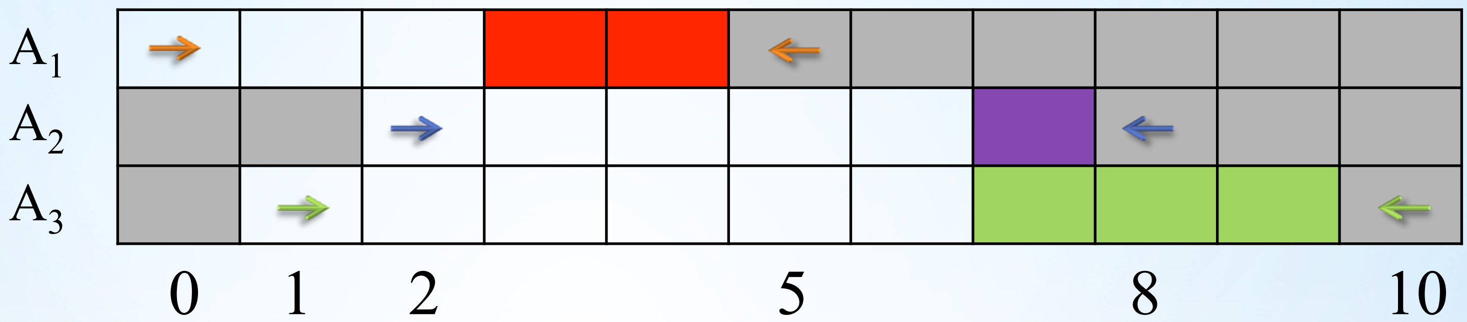


Detectable Precedences

- The tasks sorted by earliest completion times

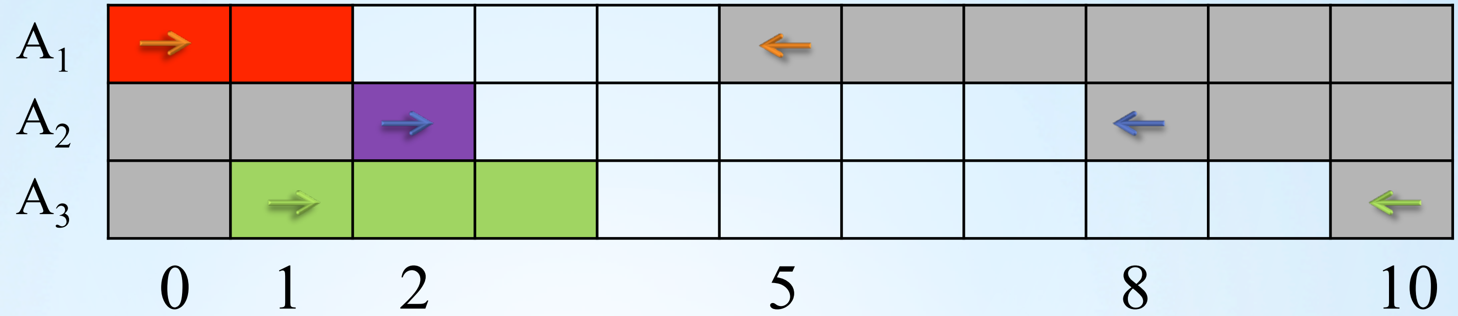


- The tasks sorted by latest starting times

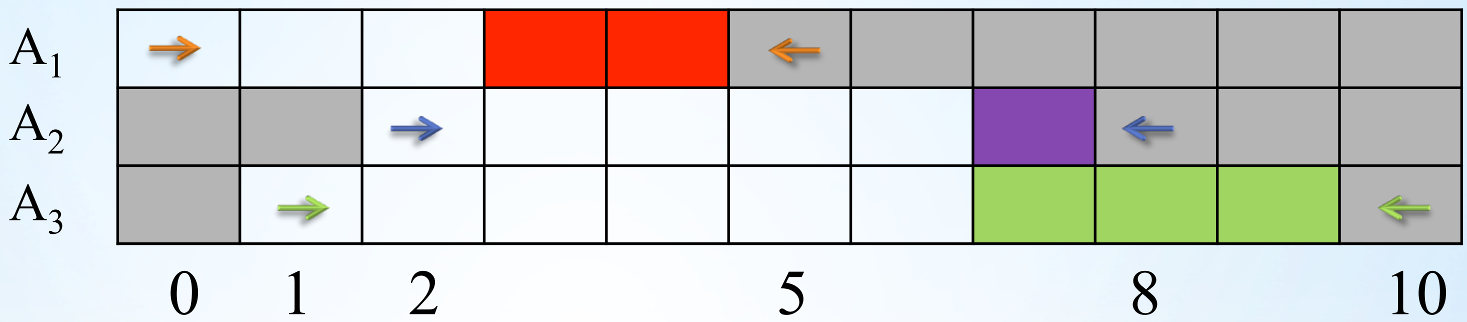


Detectable Precedences

- The tasks sorted by earliest completion times



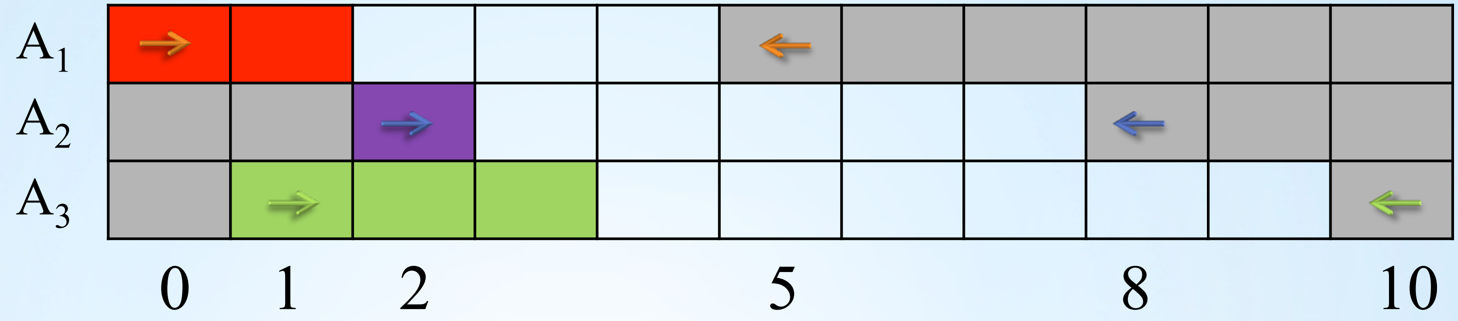
- The tasks sorted by latest starting times



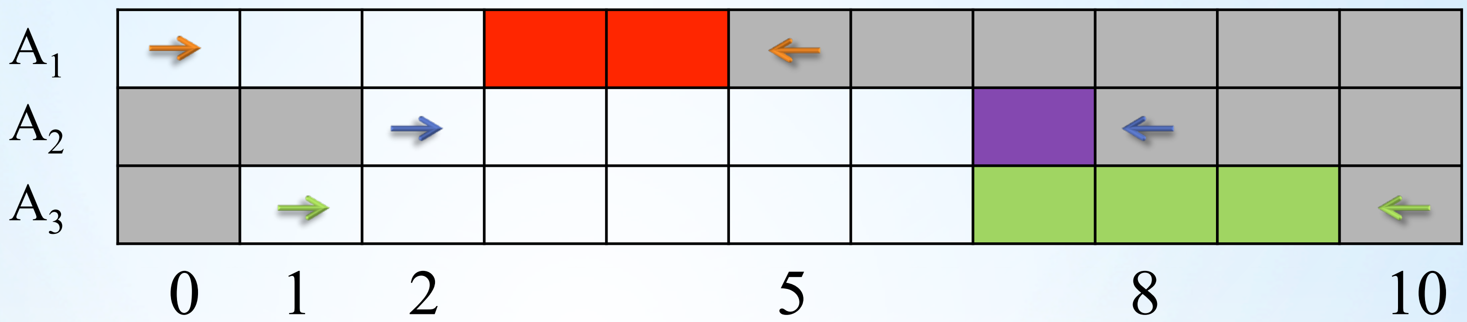
- No task has a fixed part;

Detectable Precedences

• The tasks sorted by earliest completion times



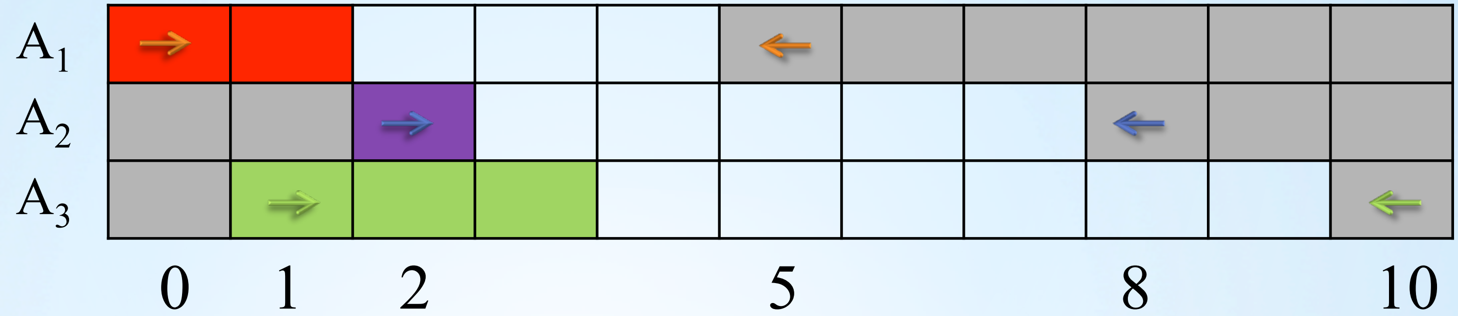
• The tasks sorted by latest starting times



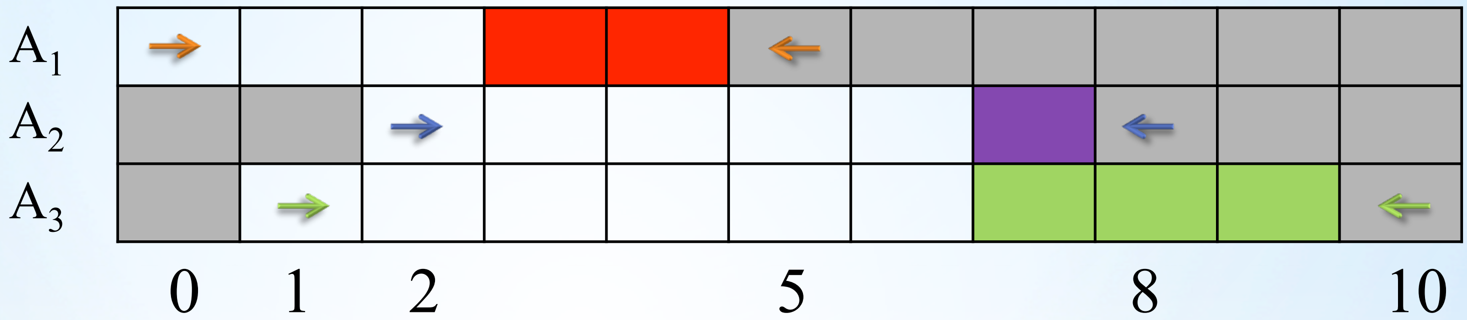
- Simultaneously iterate over all the tasks *i* from the first table and on all the tasks *k* from the second table .

Detectable Precedences

- The tasks sorted by earliest completion times



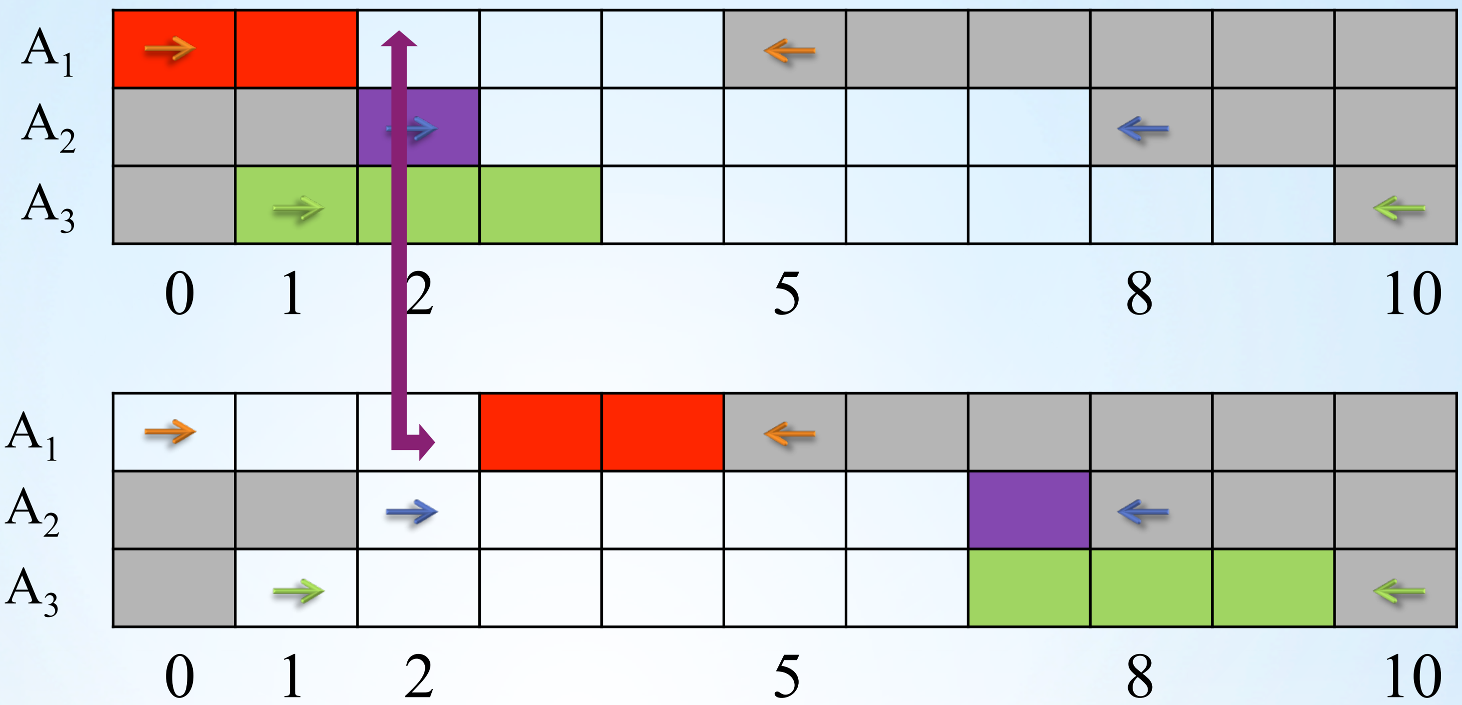
- The tasks sorted by latest starting times



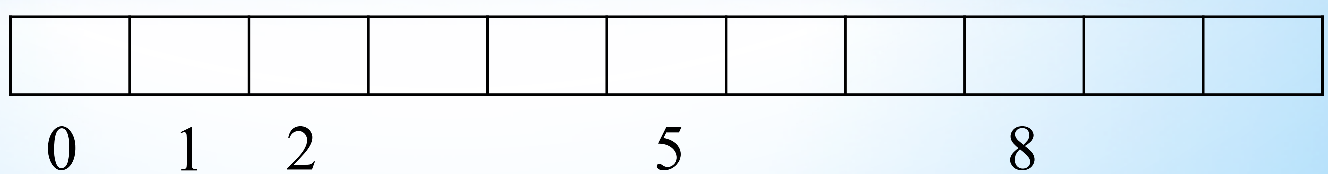
- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k << A_i$ exists, will be scheduled.

Detectable Precedences

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

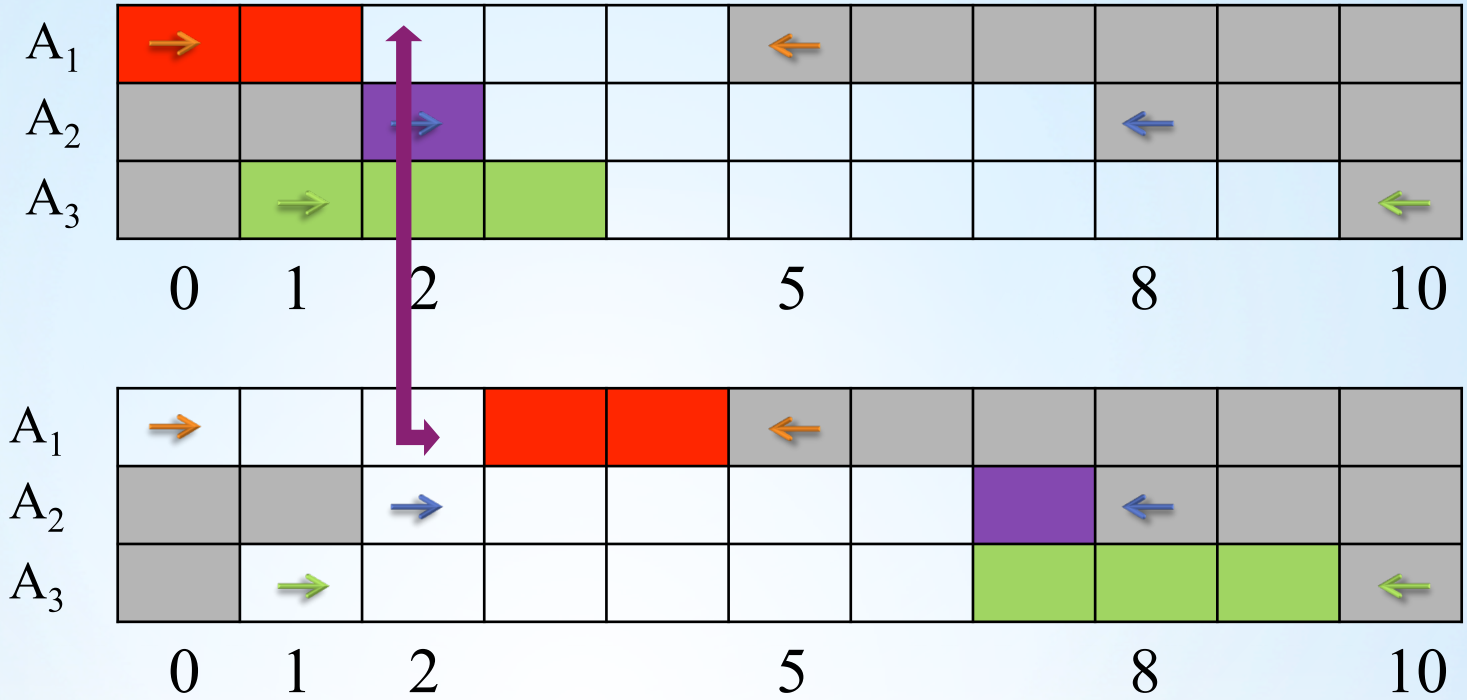


- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k \ll A_i$ exists, will be scheduled.
- Checking if $lst_1 < ect_1$?

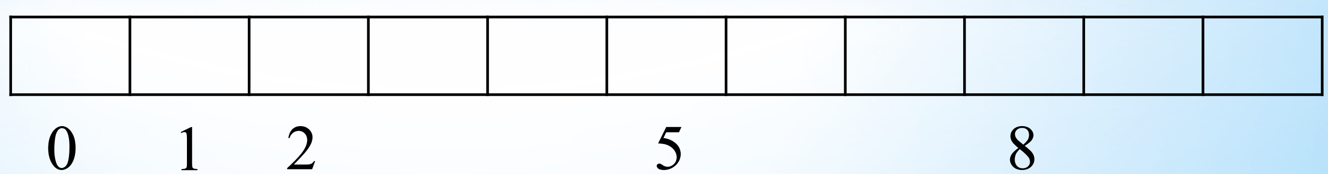


Detectable Precedences

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

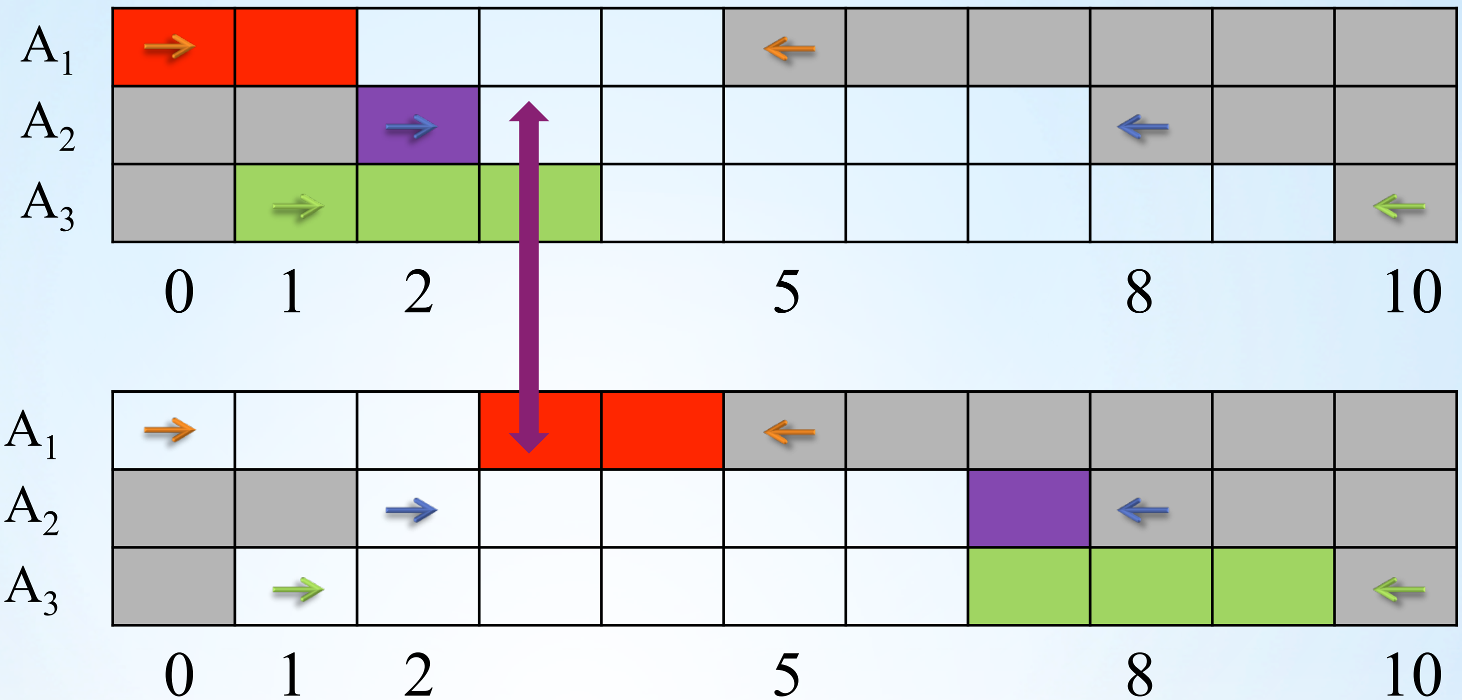


- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k << A_i$ exists, will be scheduled.
- Checking if $lst_1 < ect_1$? No!

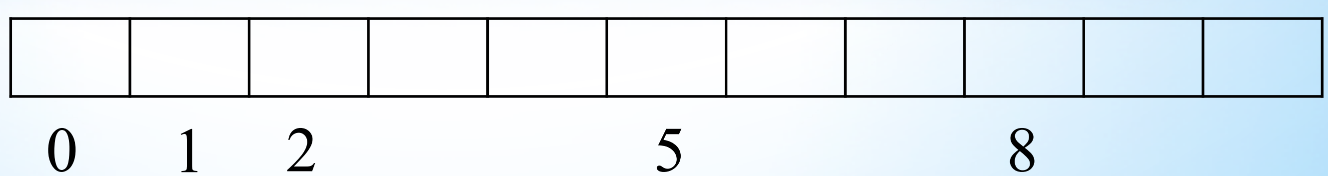


Detectable Precedences

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

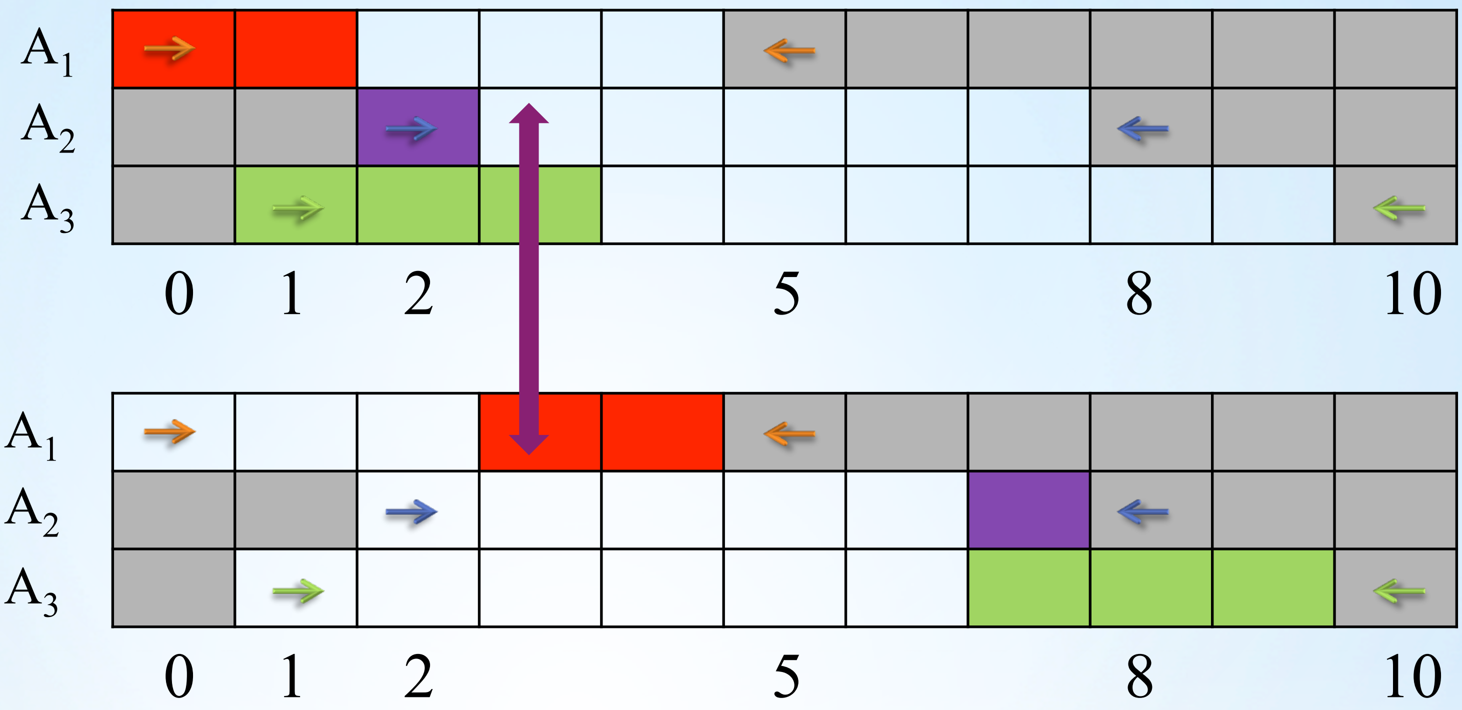


- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k << A_i$ exists, will be scheduled.
- Checking if $lst_1 < ect_2$?

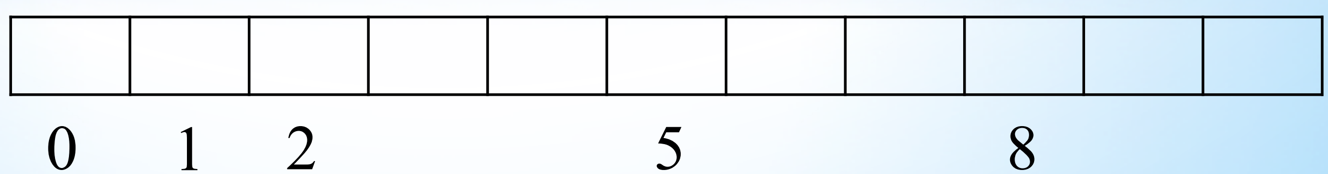


Detectable Precedences

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

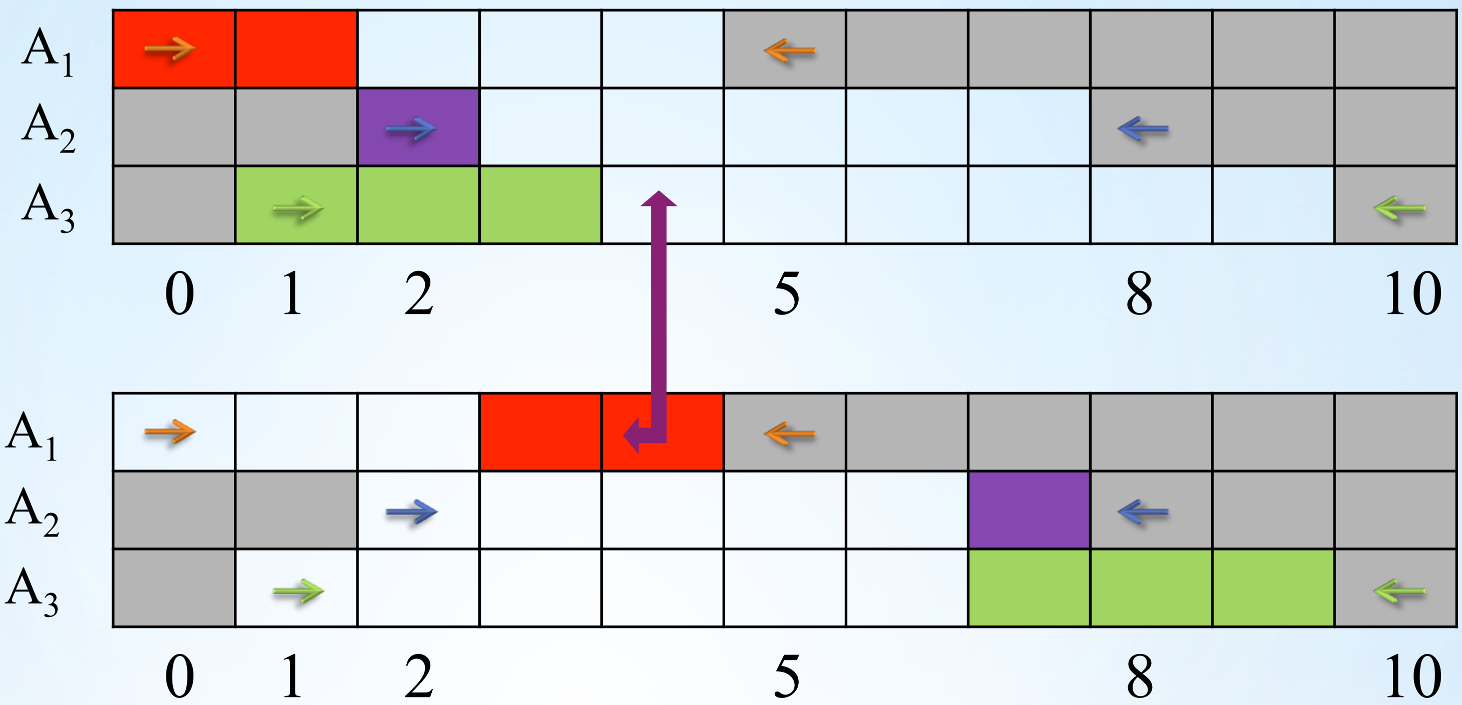


- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k << A_i$ exists, will be scheduled.
- Checking if $lst_1 < ect_2$? No!

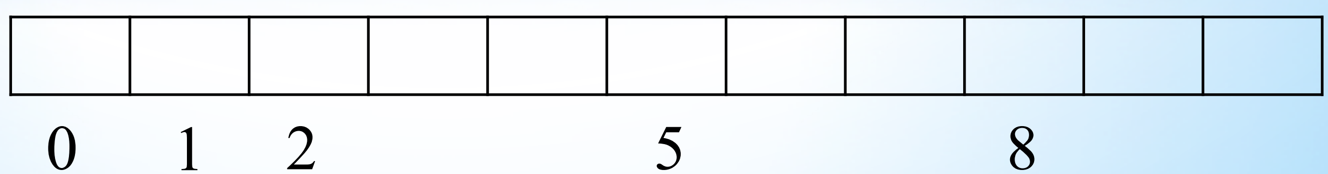


Detectable Precedences

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

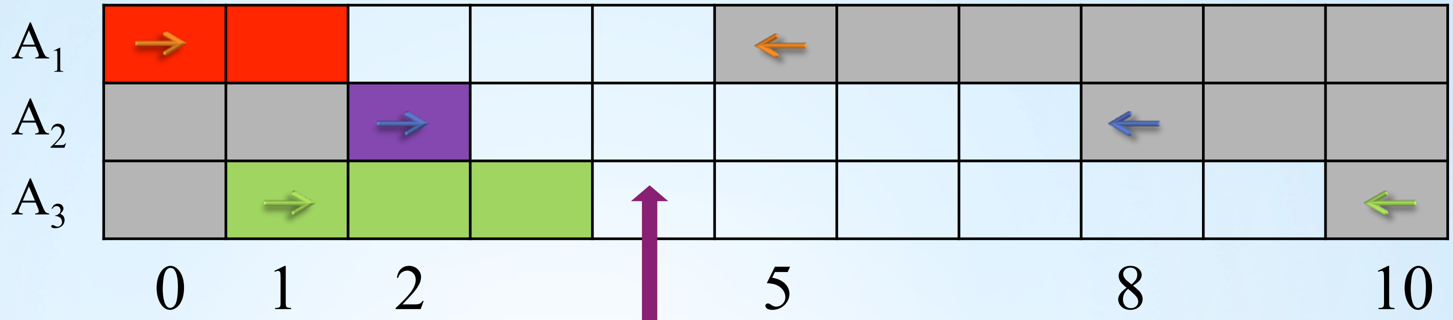


- While iterating over the next task i, all the tasks k for which the detectable precedence $A_k \ll A_i$ exists, will be scheduled.
- Checking if $lst_1 < ect_3$?

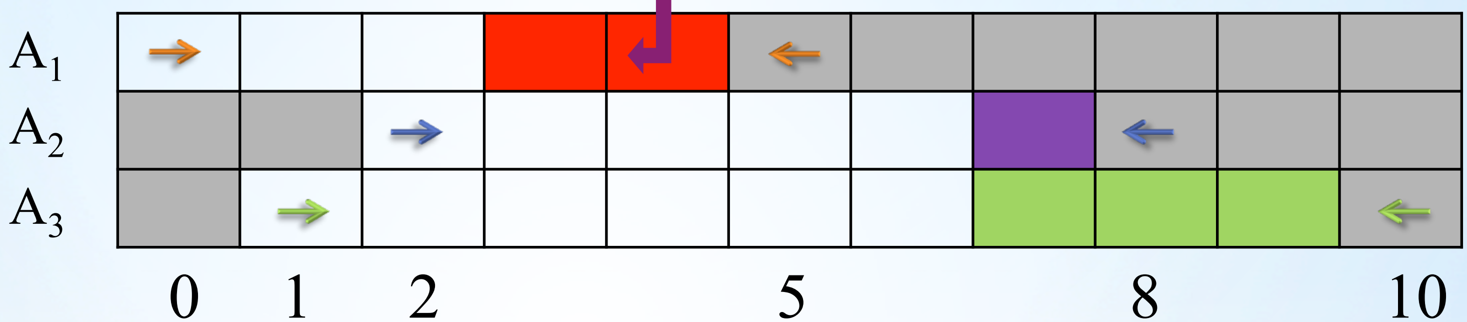


Detectable Precedences

- The tasks sorted by earliest completion times

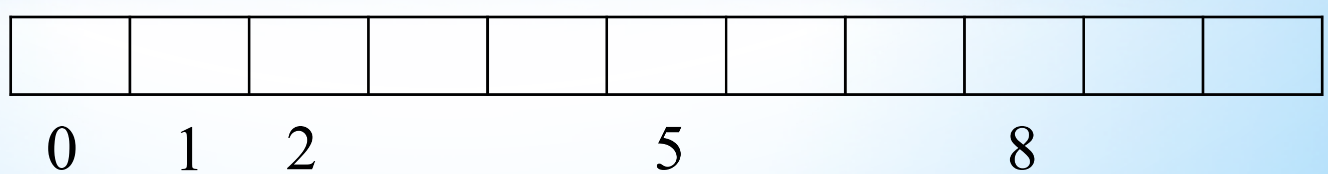


- The tasks sorted by latest starting times



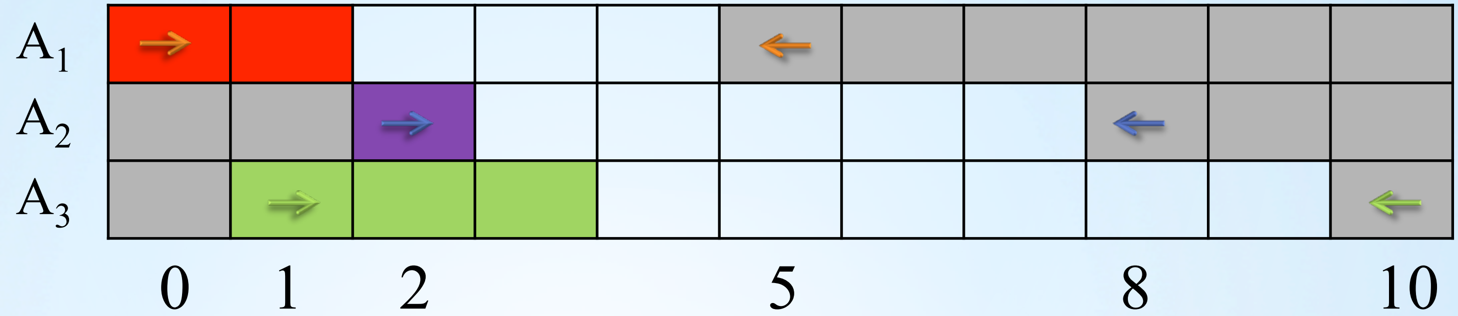
- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k << A_i$ exists, will be scheduled.

- Checking if $lst_1 < ect_3$? Yes!

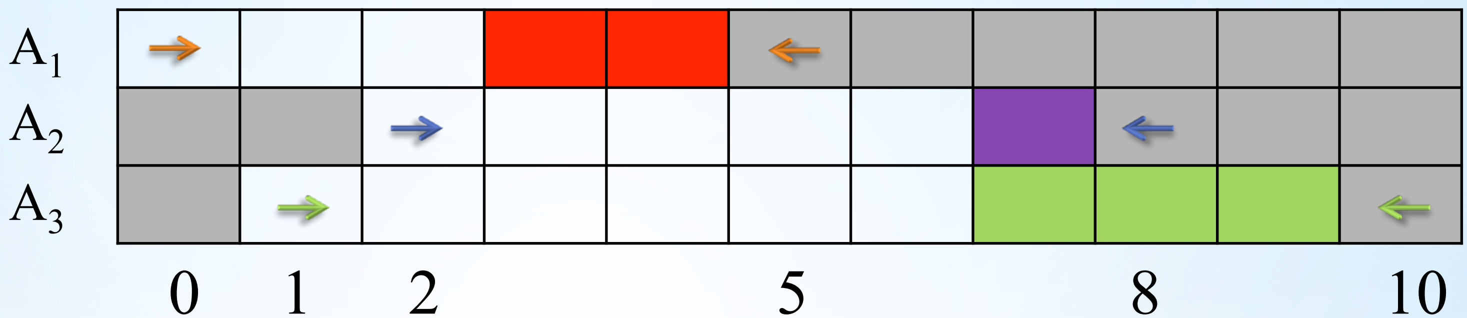


Detectable Precedences

- The tasks sorted by earliest completion times

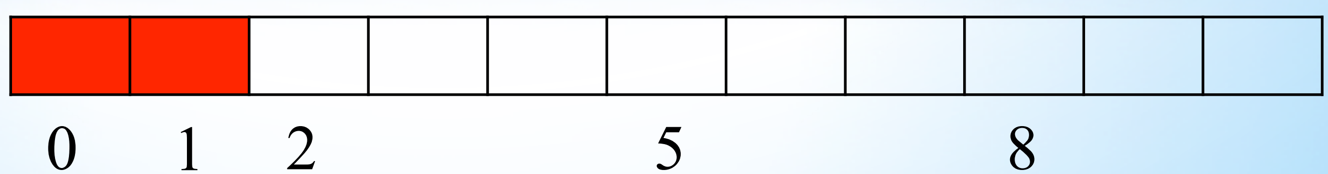


- The tasks sorted by latest starting times



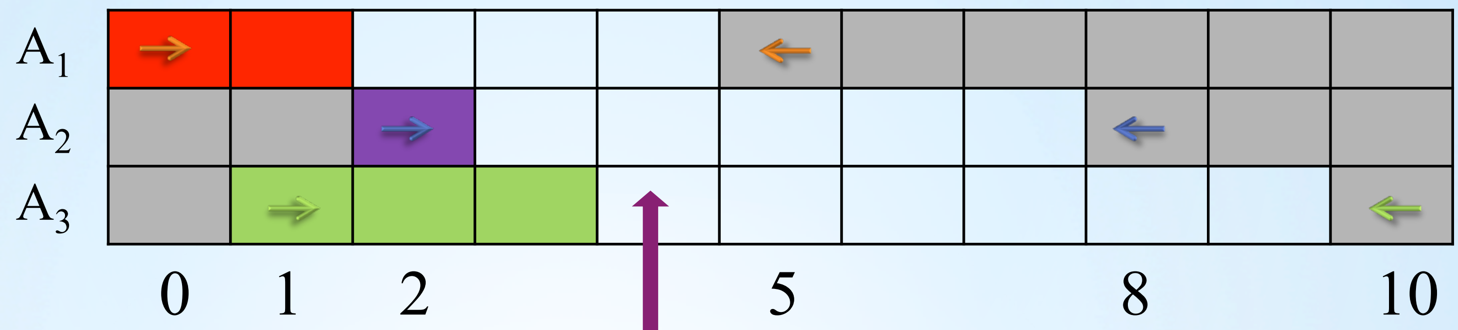
- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k \ll A_i$ exists, will be scheduled.

- Checking if $lst_1 < ect_3$? Yes!
- The red task will be scheduled on the time line.

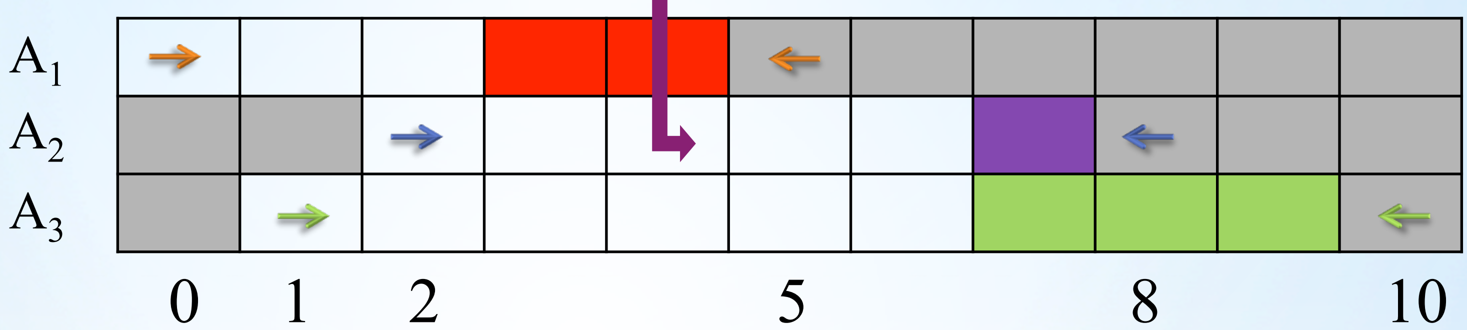


Detectable Precedences

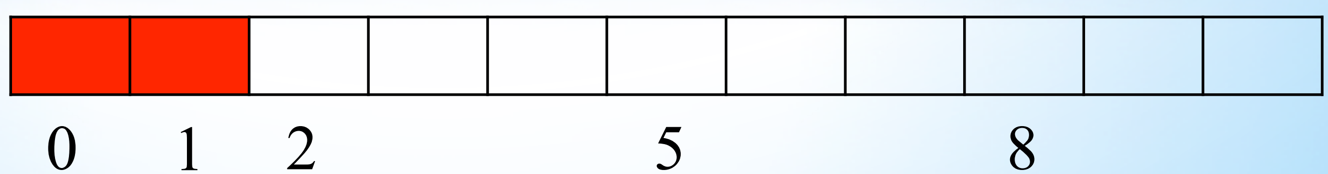
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

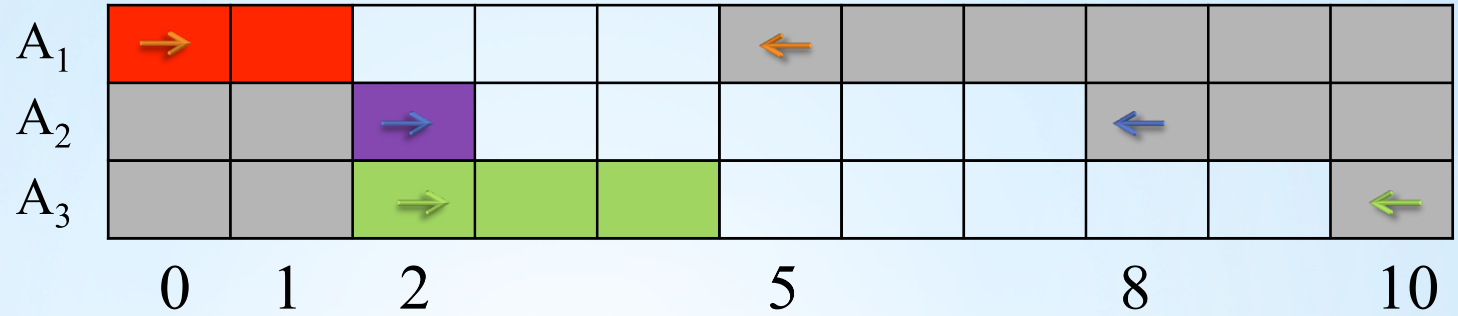


- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k \ll A_i$ exists, will be scheduled.
- Checking if $lst_2 < ect_3$? No!

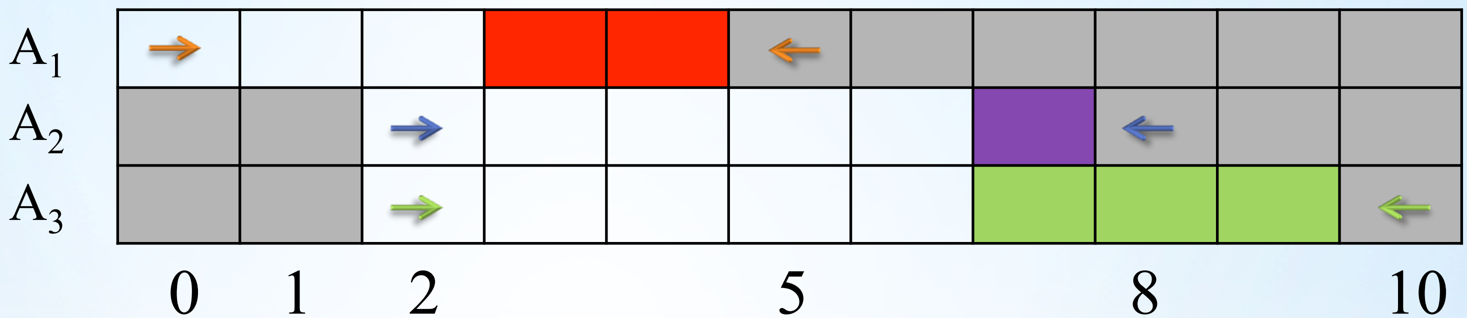


Detectable Precedences

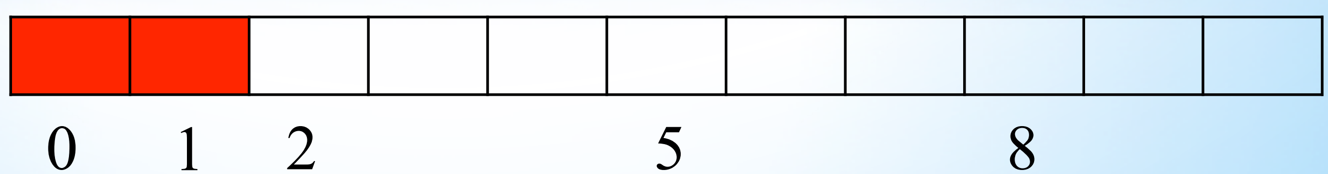
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

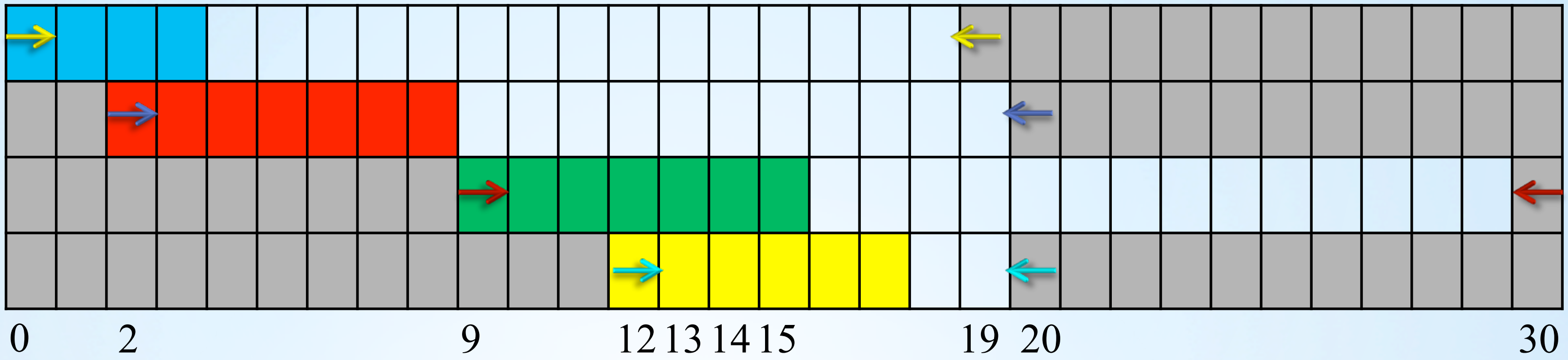


- While iterating over the next task *i*, all the tasks *k* for which the detectable precedence $A_k << A_i$ exists, will be scheduled.
- The detectable precedence rule prunes the earliest starting time of the green task up to the earliest completion time of the time line.



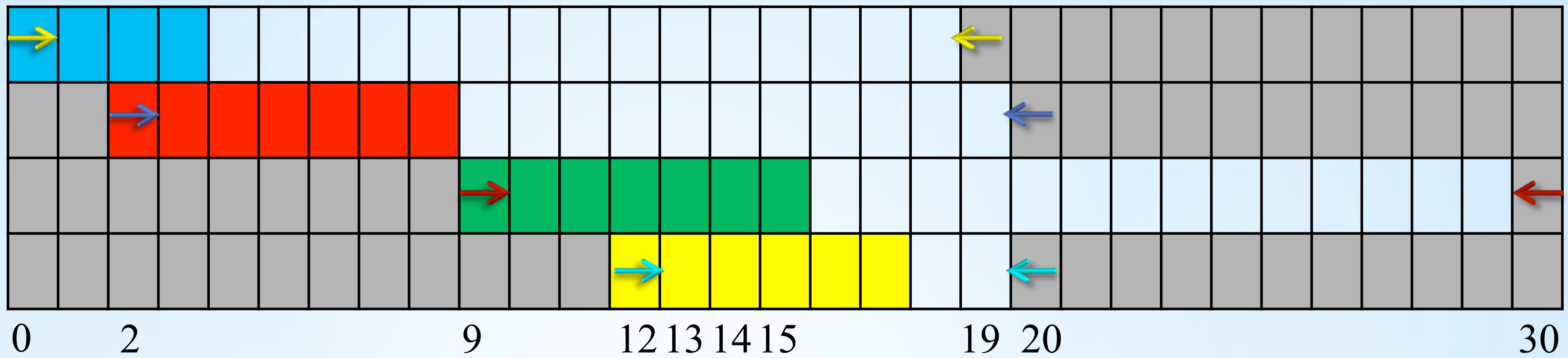
Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times

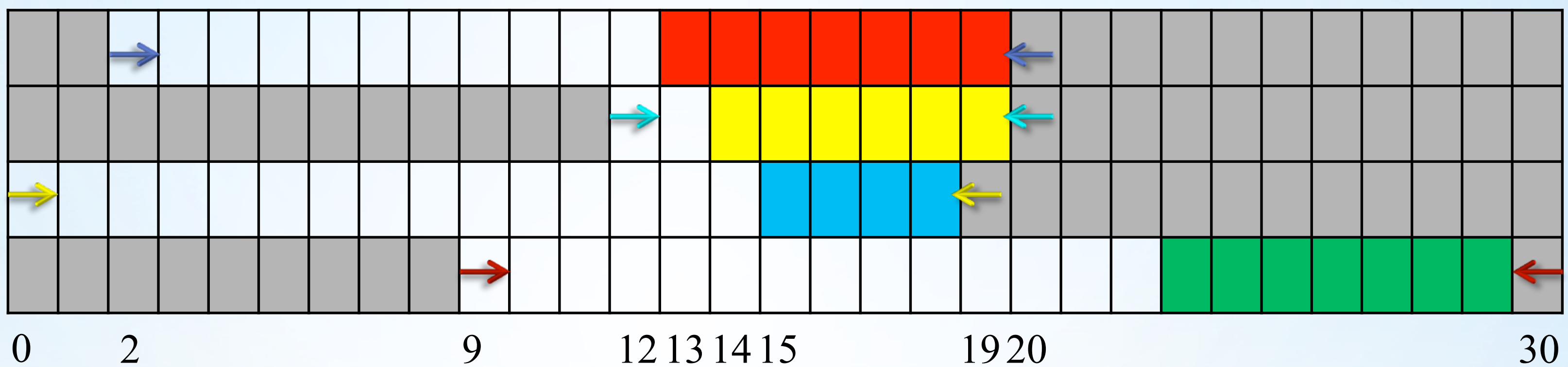


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times

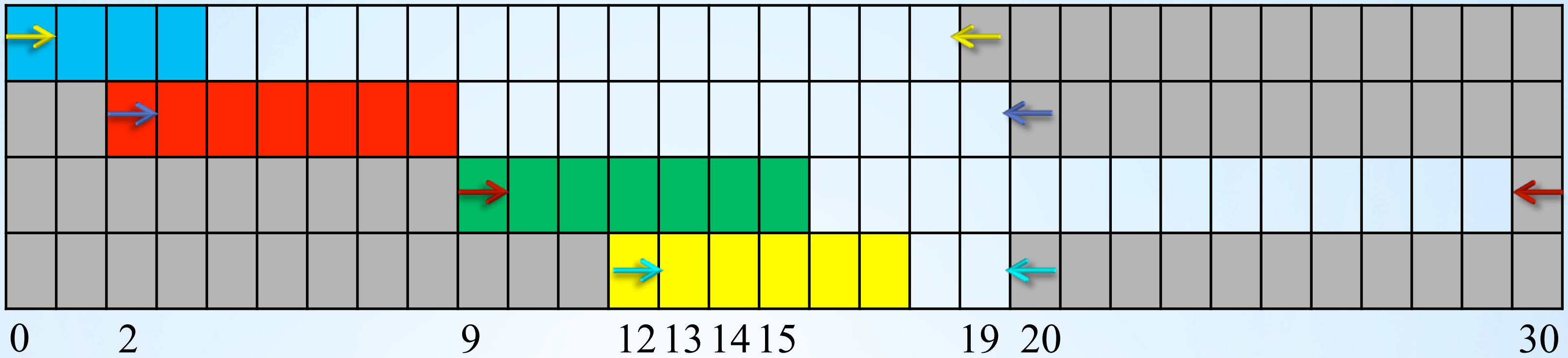


- The tasks sorted by latest starting times

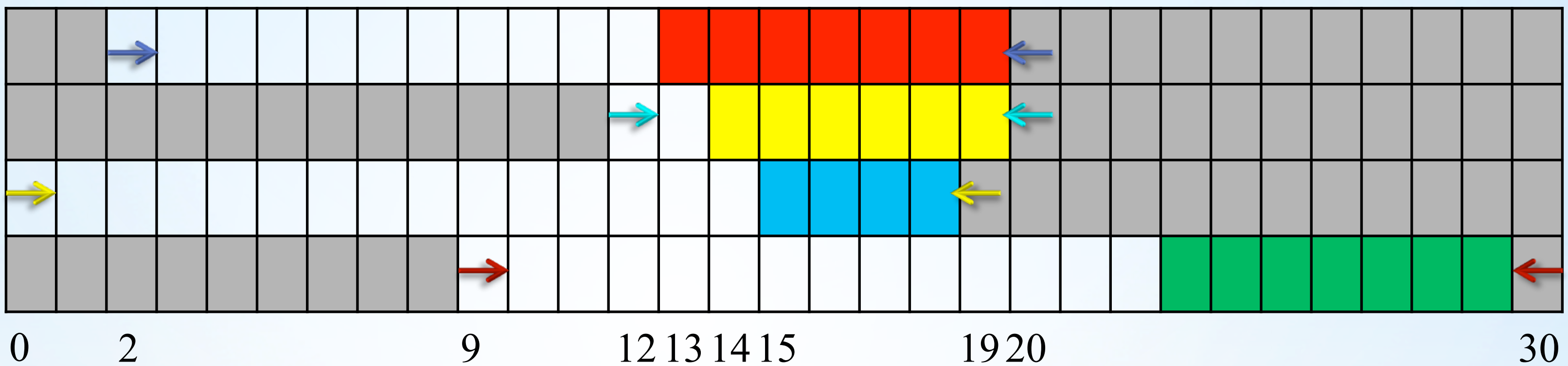


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times



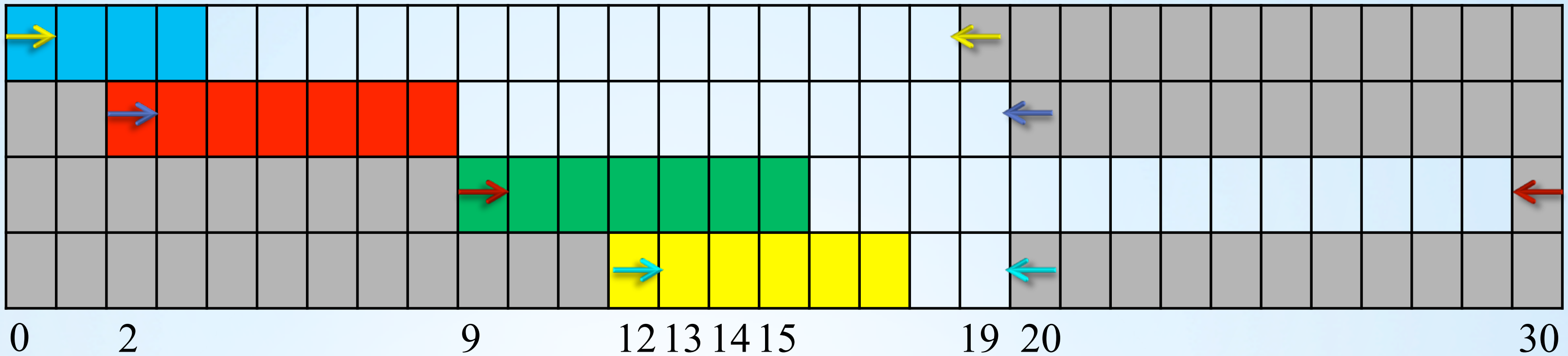
- The tasks sorted by latest starting times



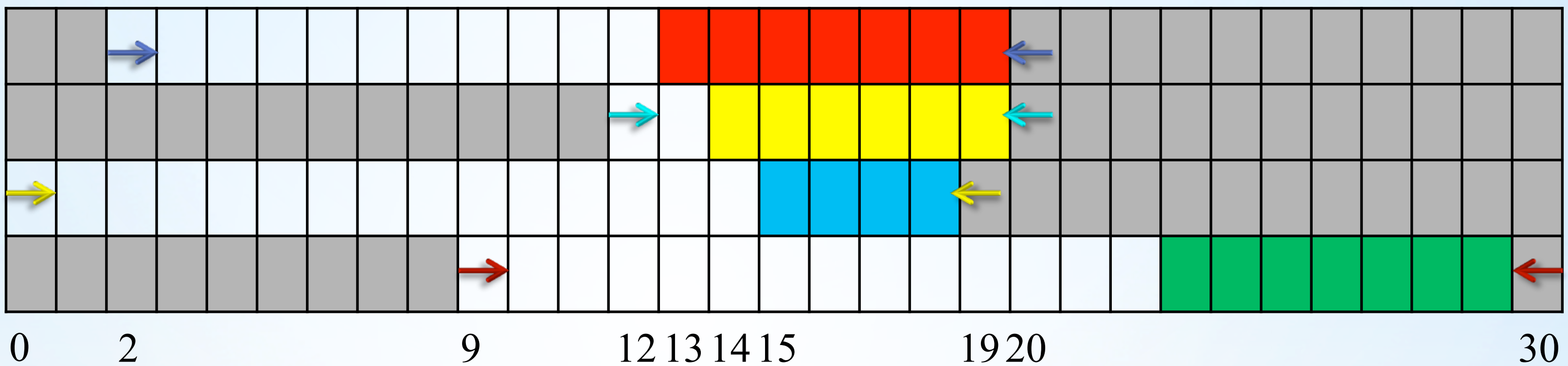
- The yellow task has a fixed part;

Detectable Precedences (with fixed part)

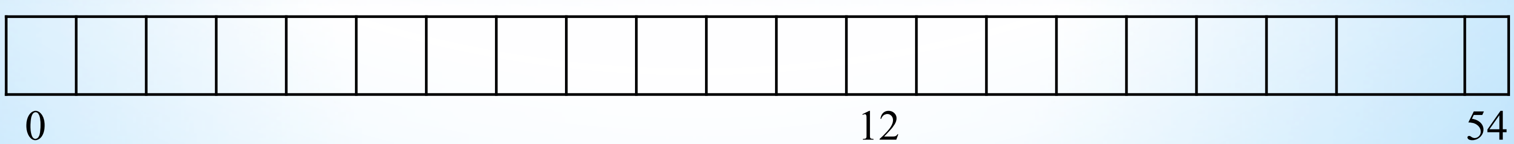
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

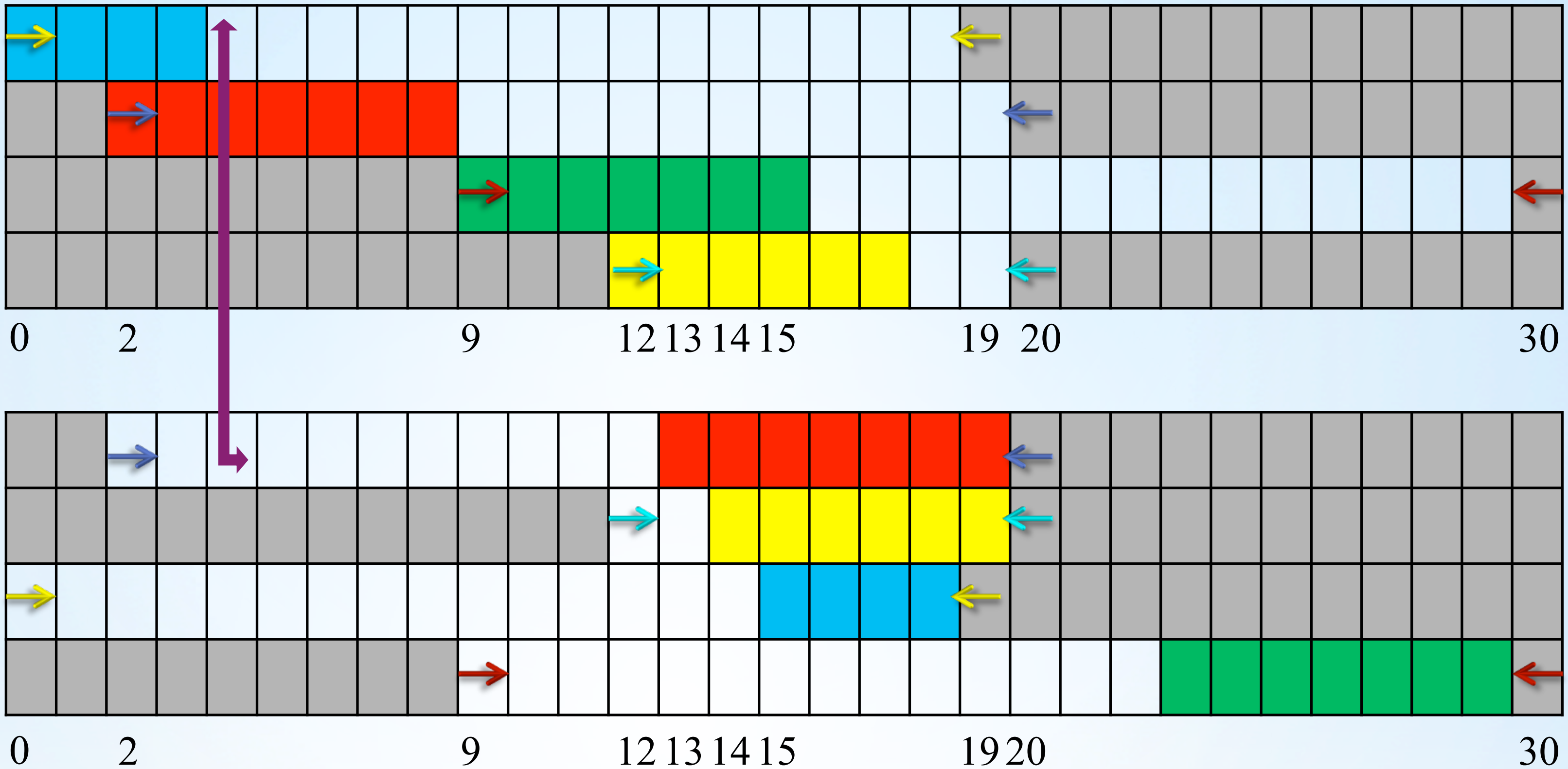


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .

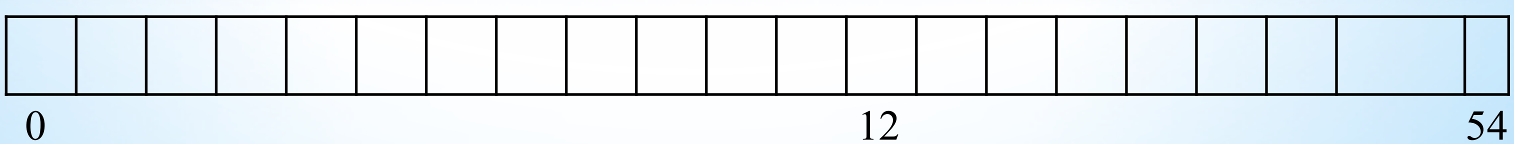


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

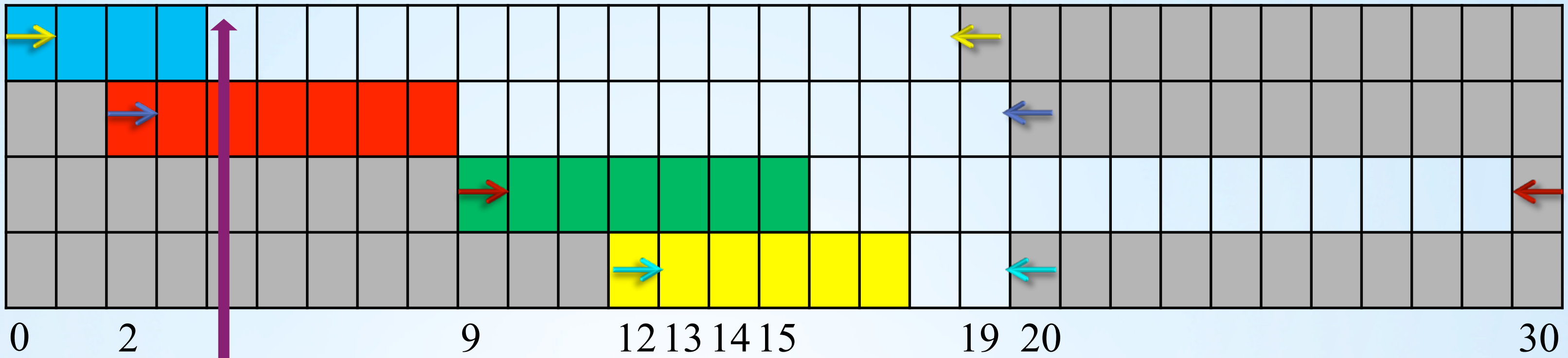


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_1 < ect_1$?

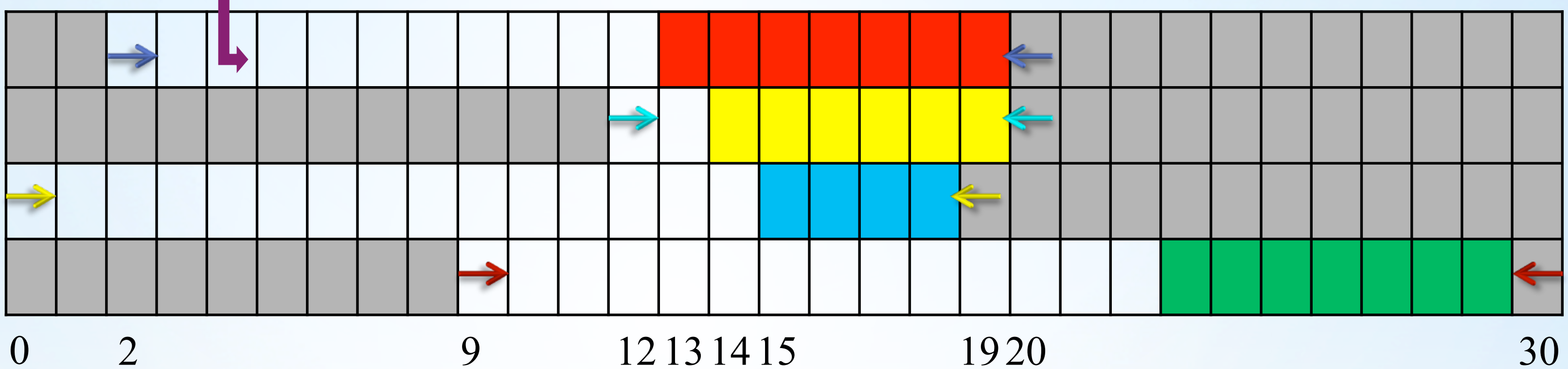


Detectable Precedences (with fixed part)

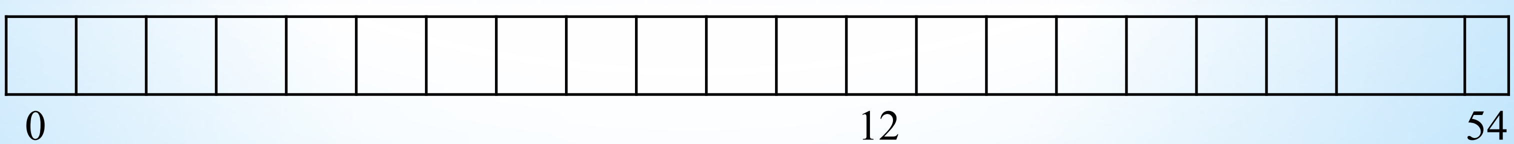
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

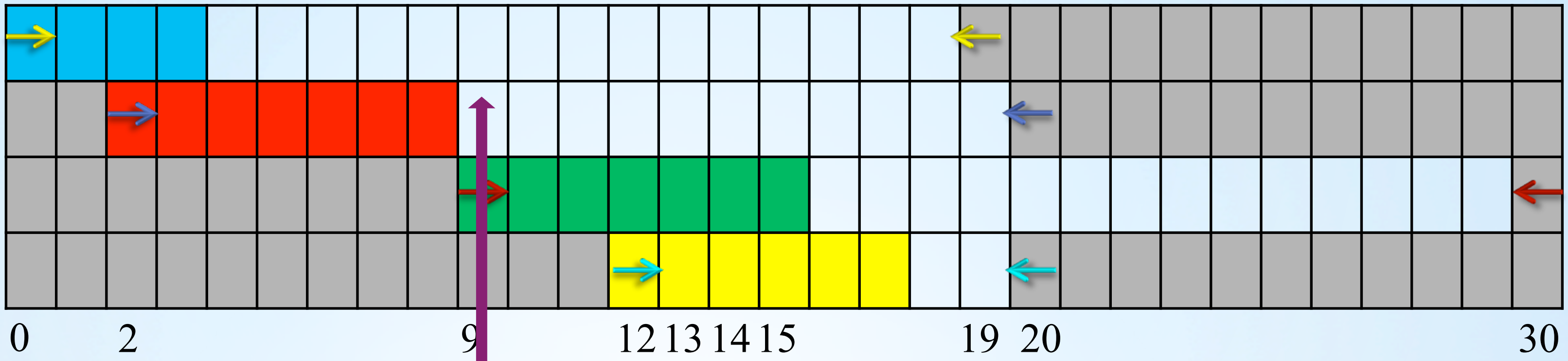


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_1 < ect_1$? No!

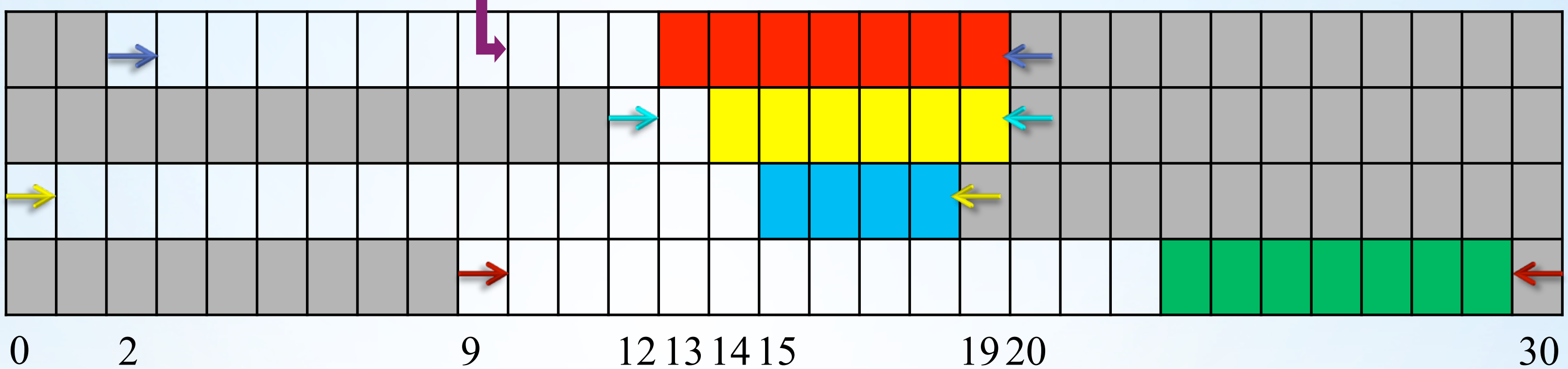


Detectable Precedences (with fixed part)

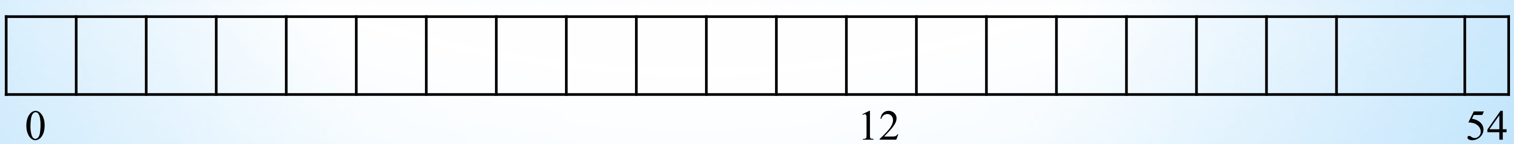
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

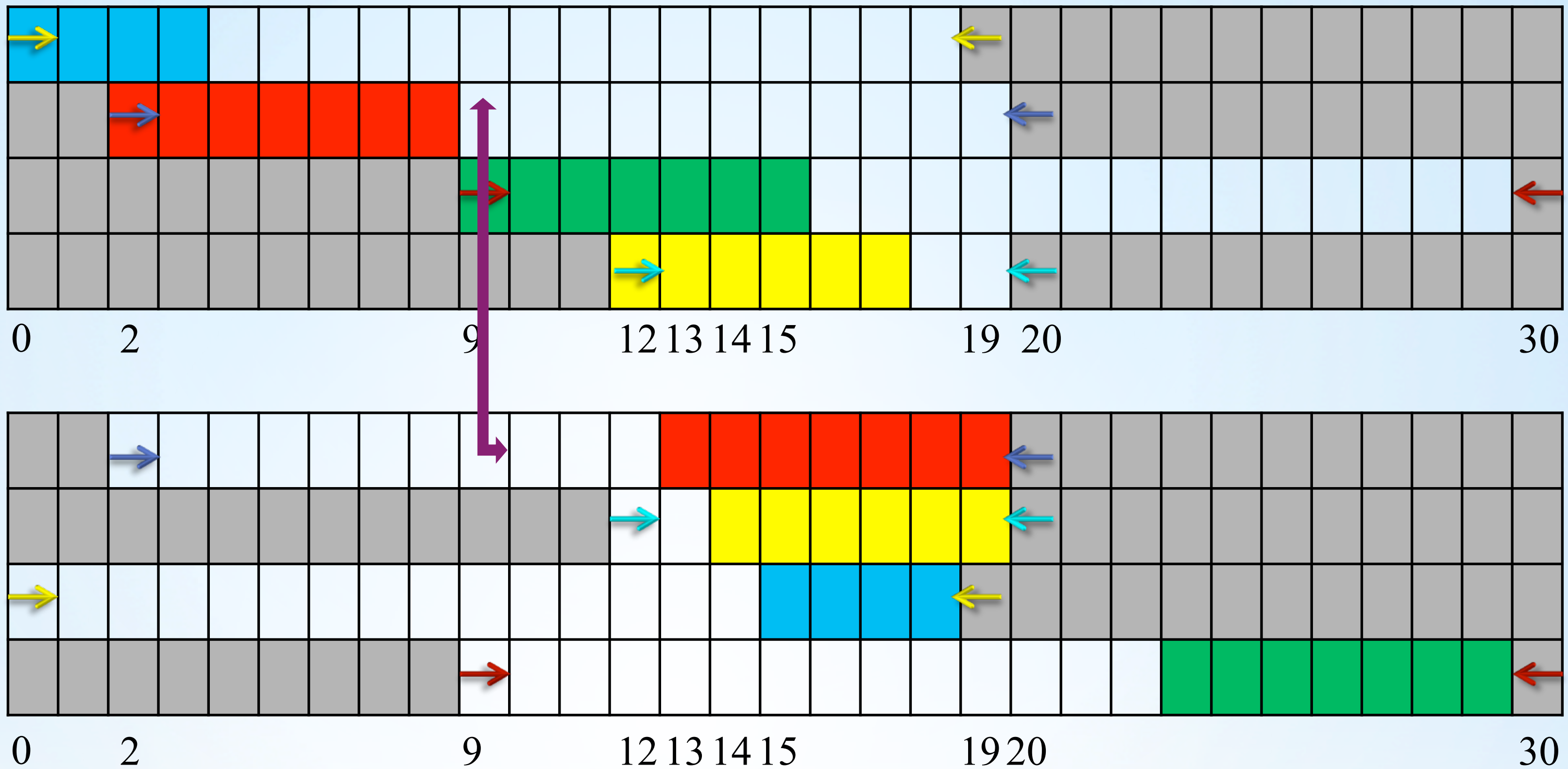


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_1 < ect_2$?

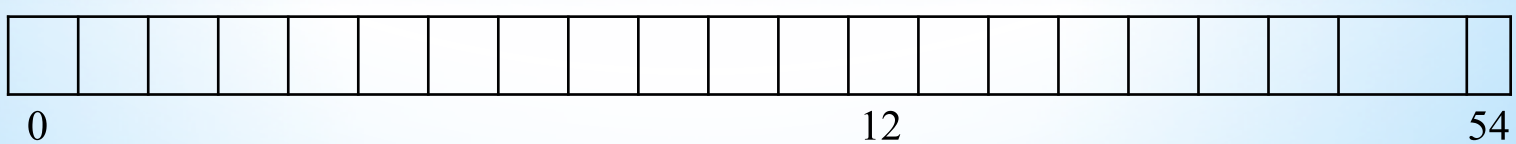


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

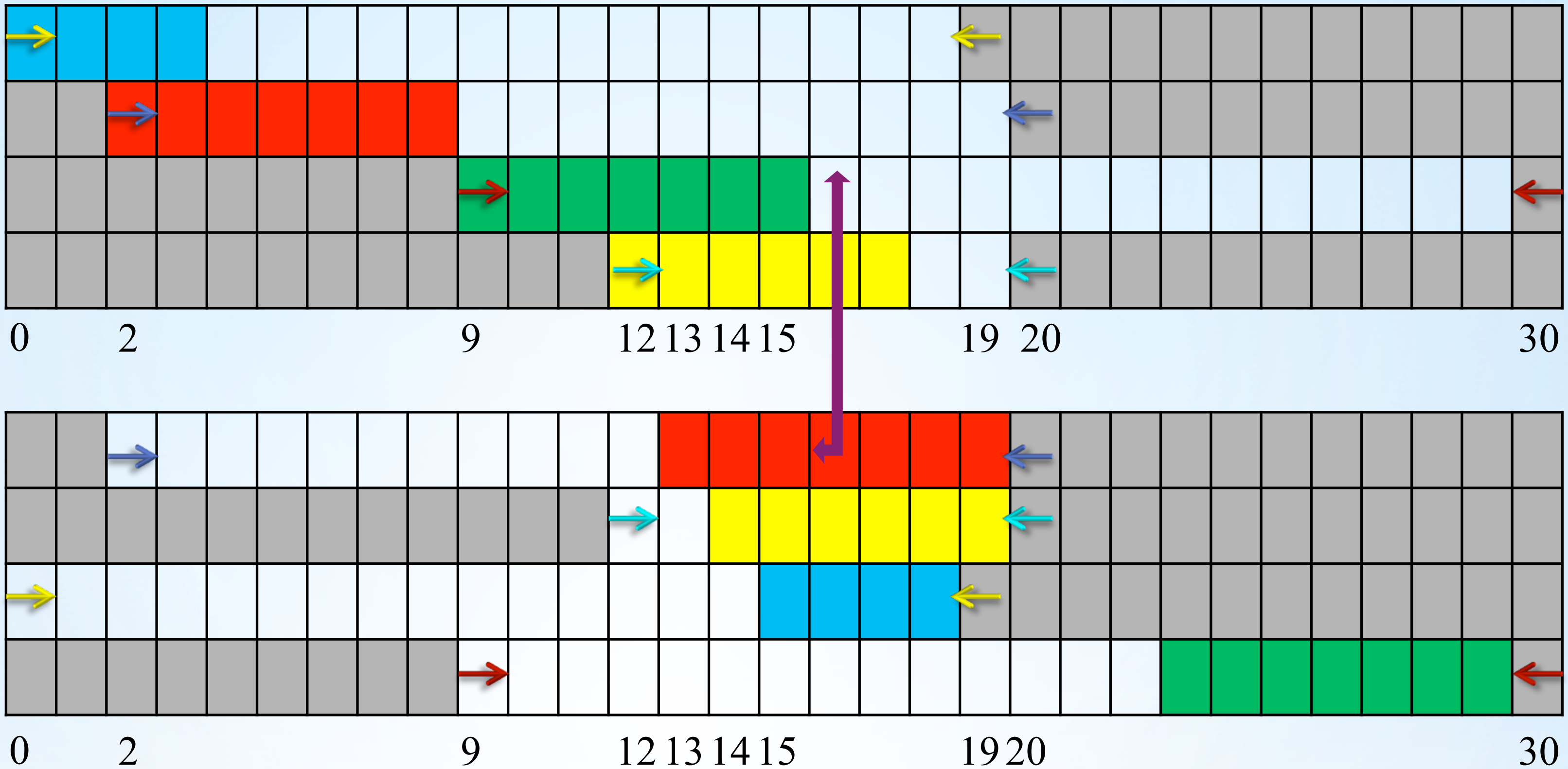


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_1 < ect_2$? No!

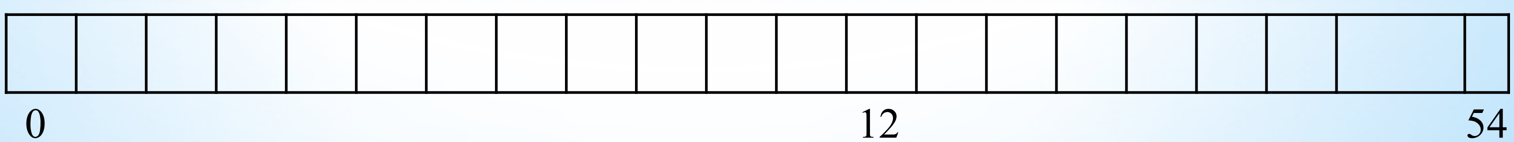


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

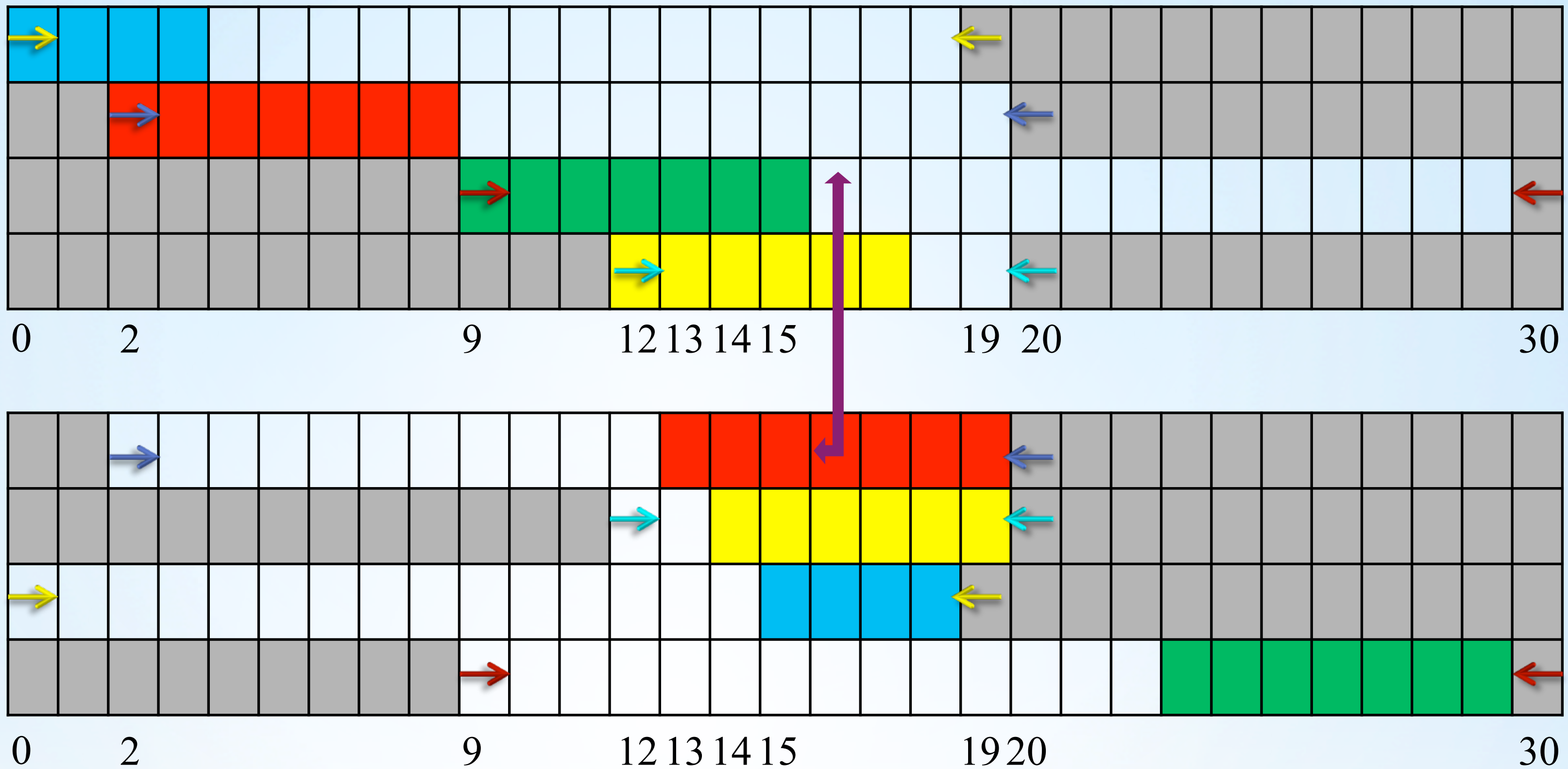


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_1 < ect_3$?



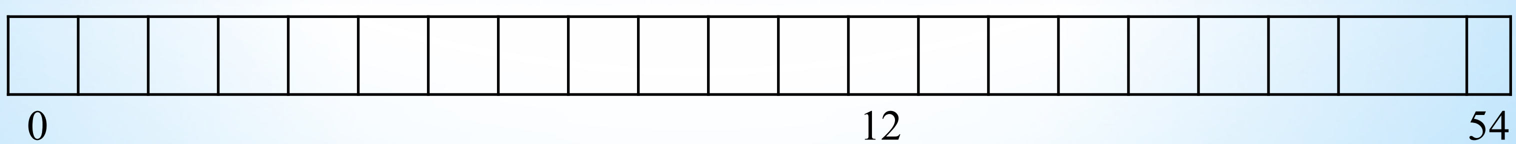
Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times



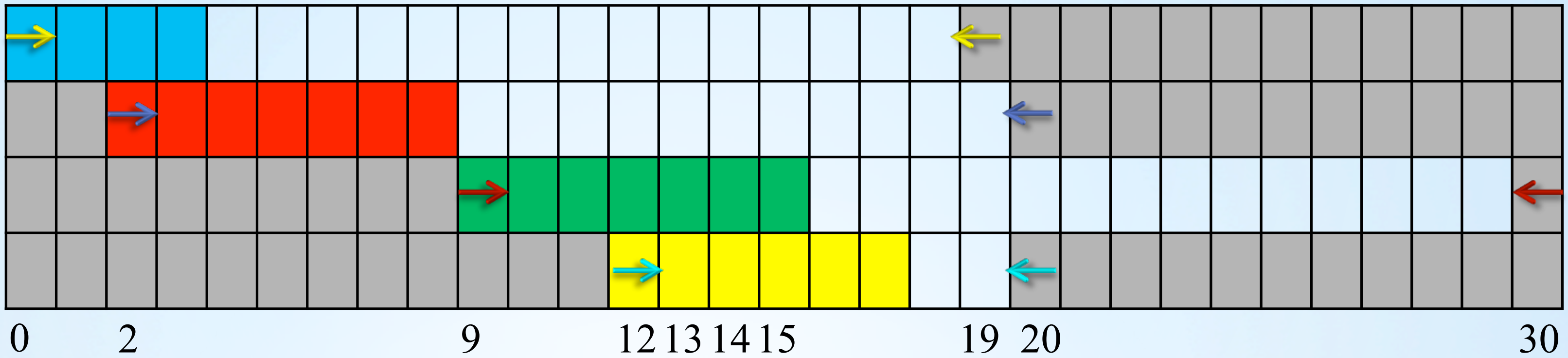
- The tasks sorted by latest starting times

- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_1 < ect_3$? Yes!

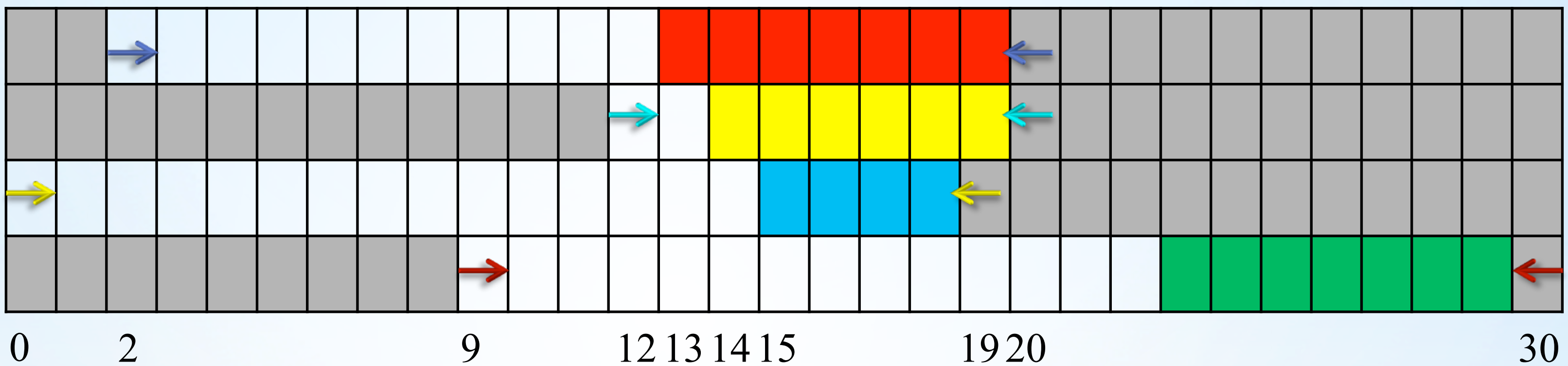


Detectable Precedences (with fixed part)

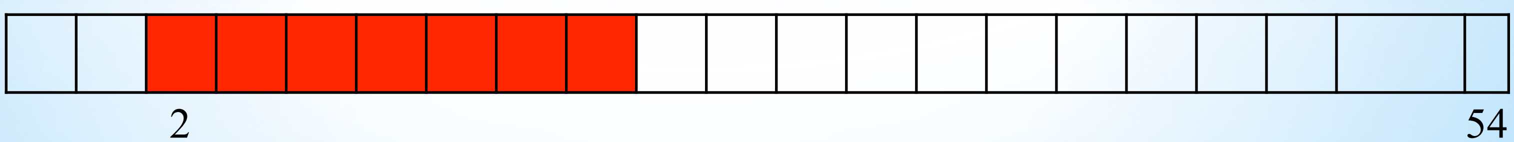
- The tasks sorted by earliest completio n times



- The tasks sorted by latest starting times

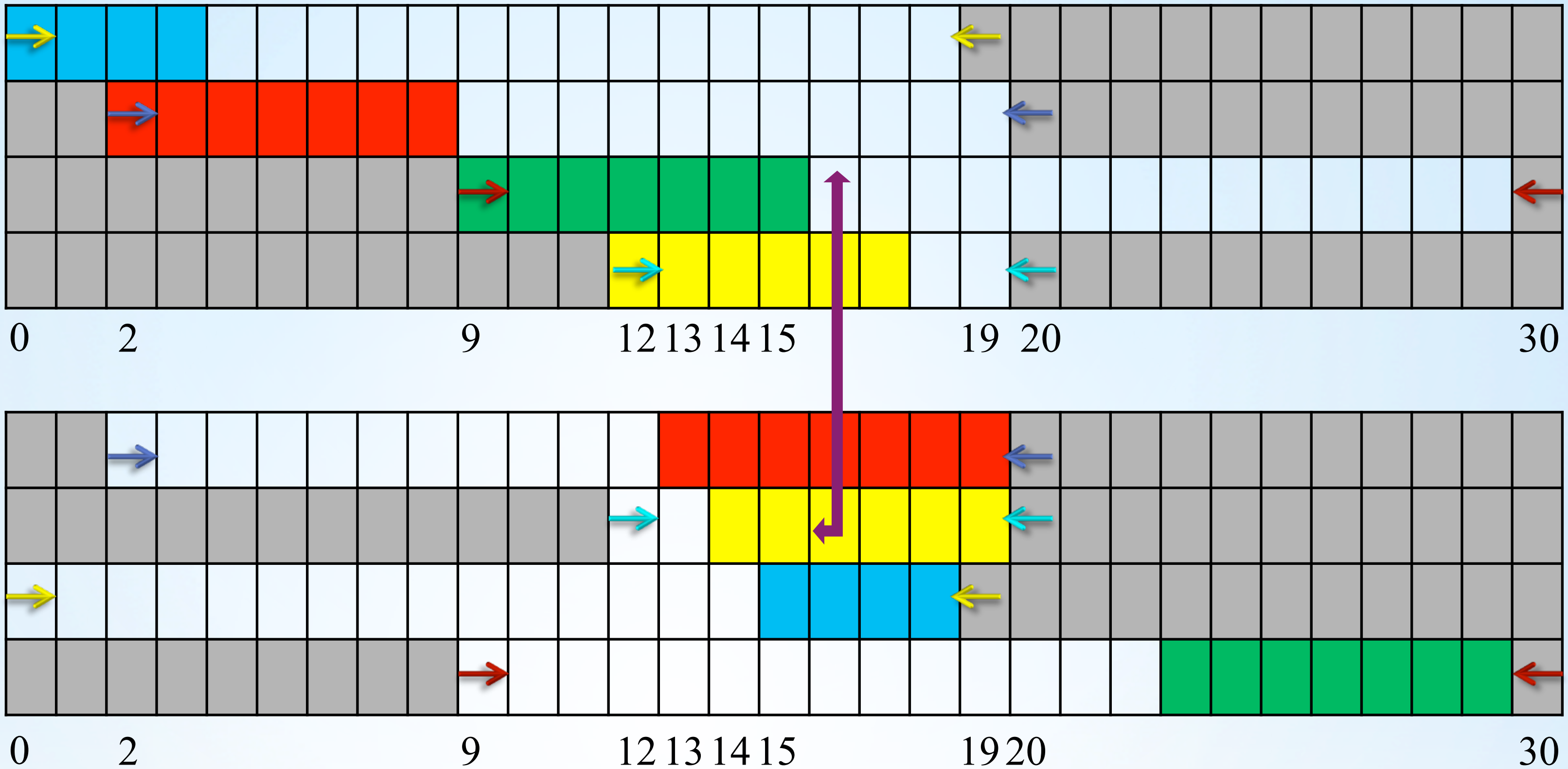


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_1 < ect_3$? Yes!
- The red task will be scheduled on the time line.

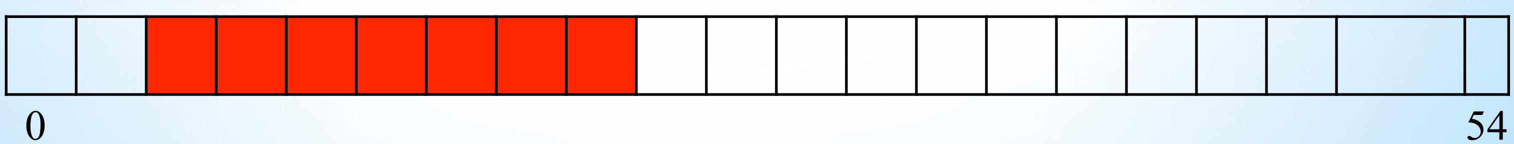


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

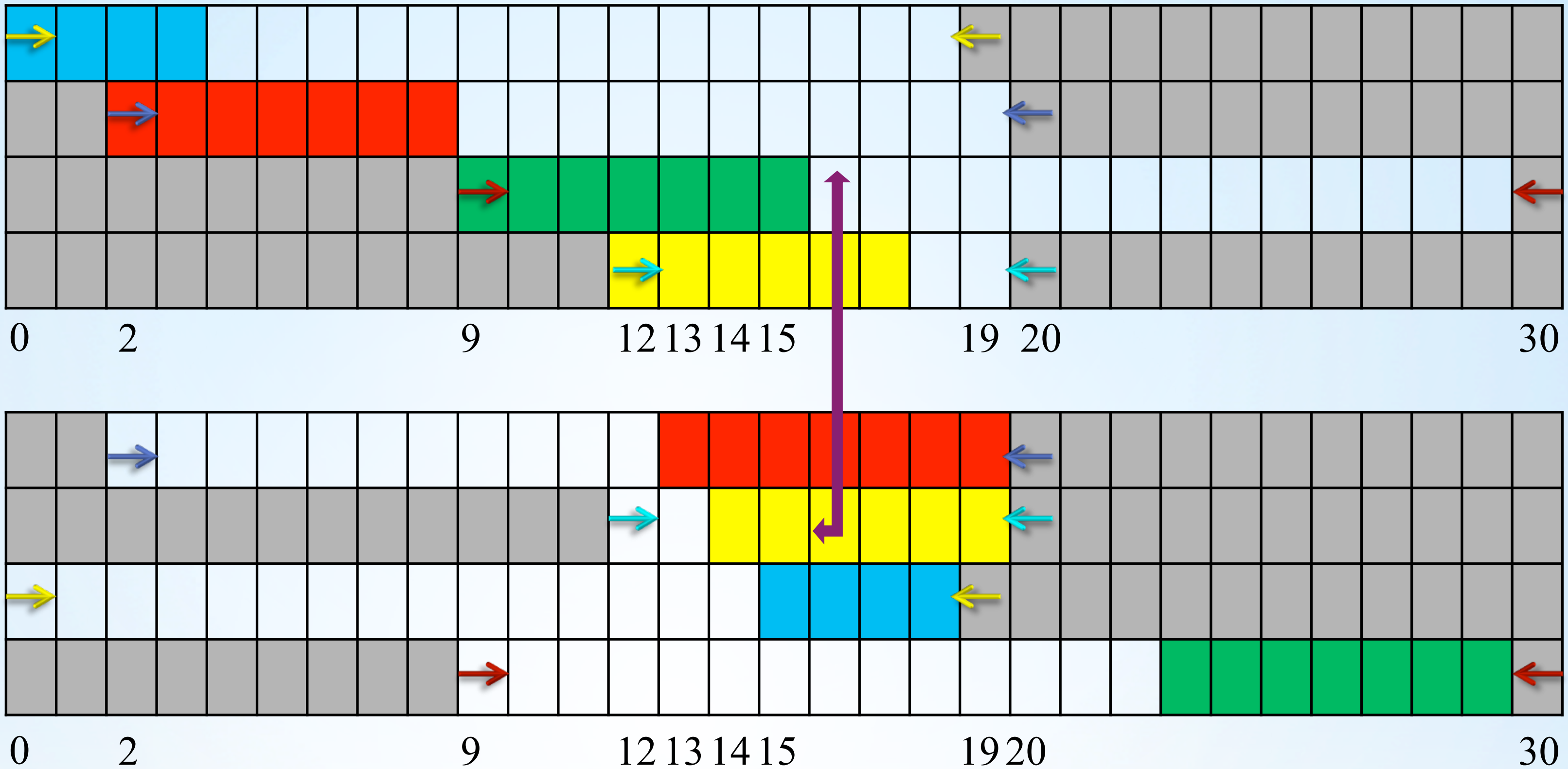


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_2 < ect_3$?

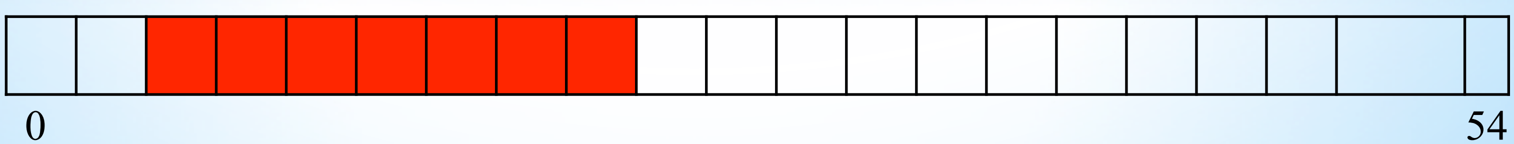


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

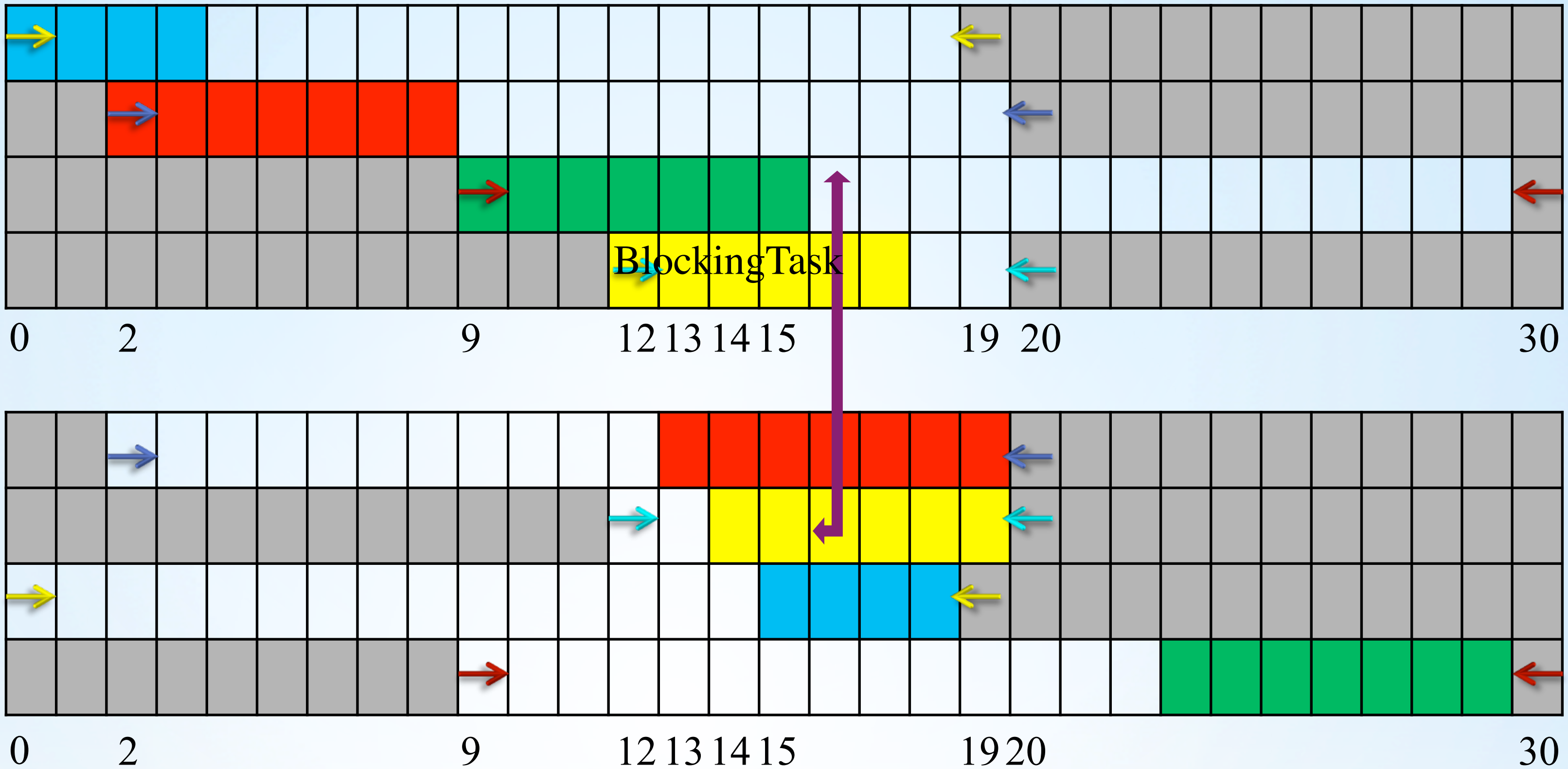


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_2 < ect_3$? Yes!

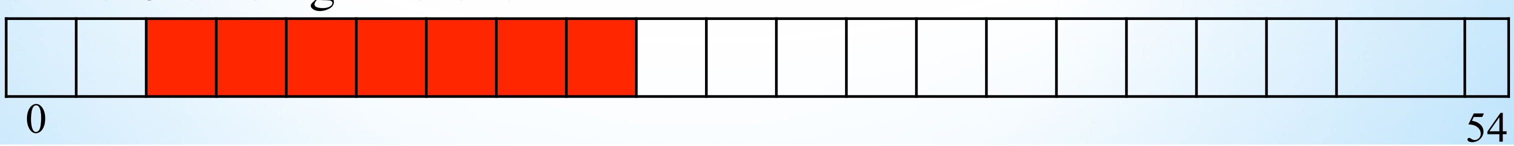


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times



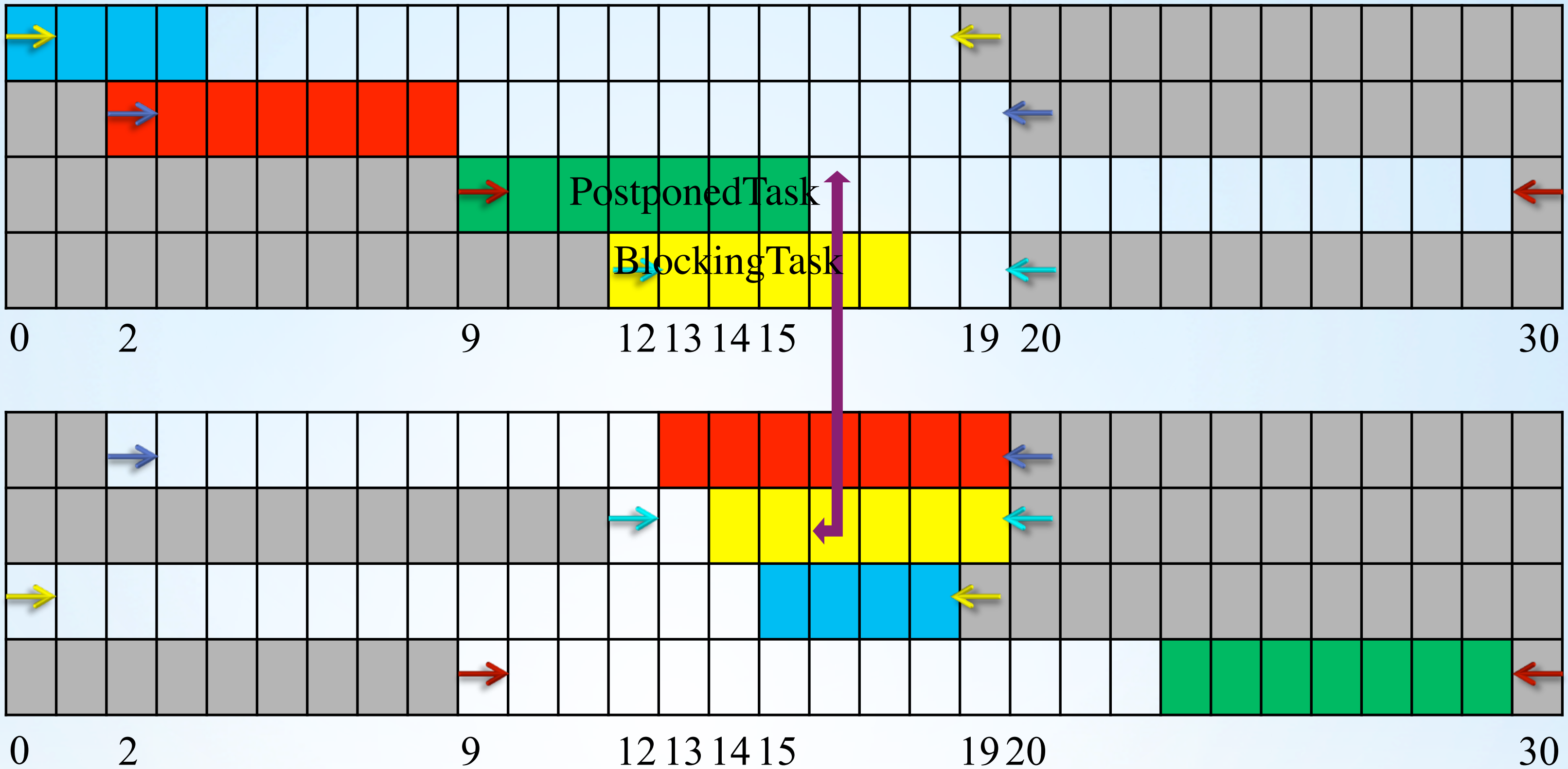
- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_2 < ect_3$? Yes!
- The yellow task has a fixed part. We call it the *blocking task*. It will not be scheduled before being filtered.



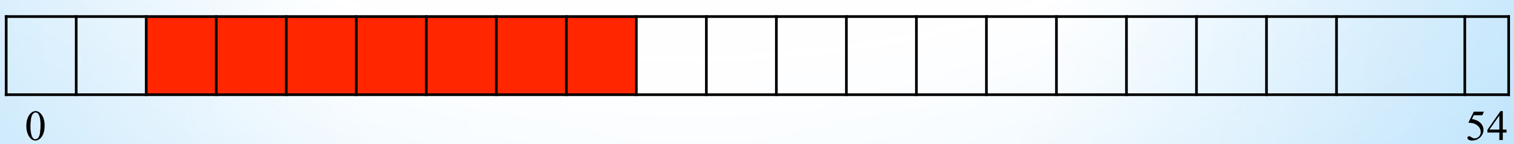
Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times

- The tasks sorted by latest starting times



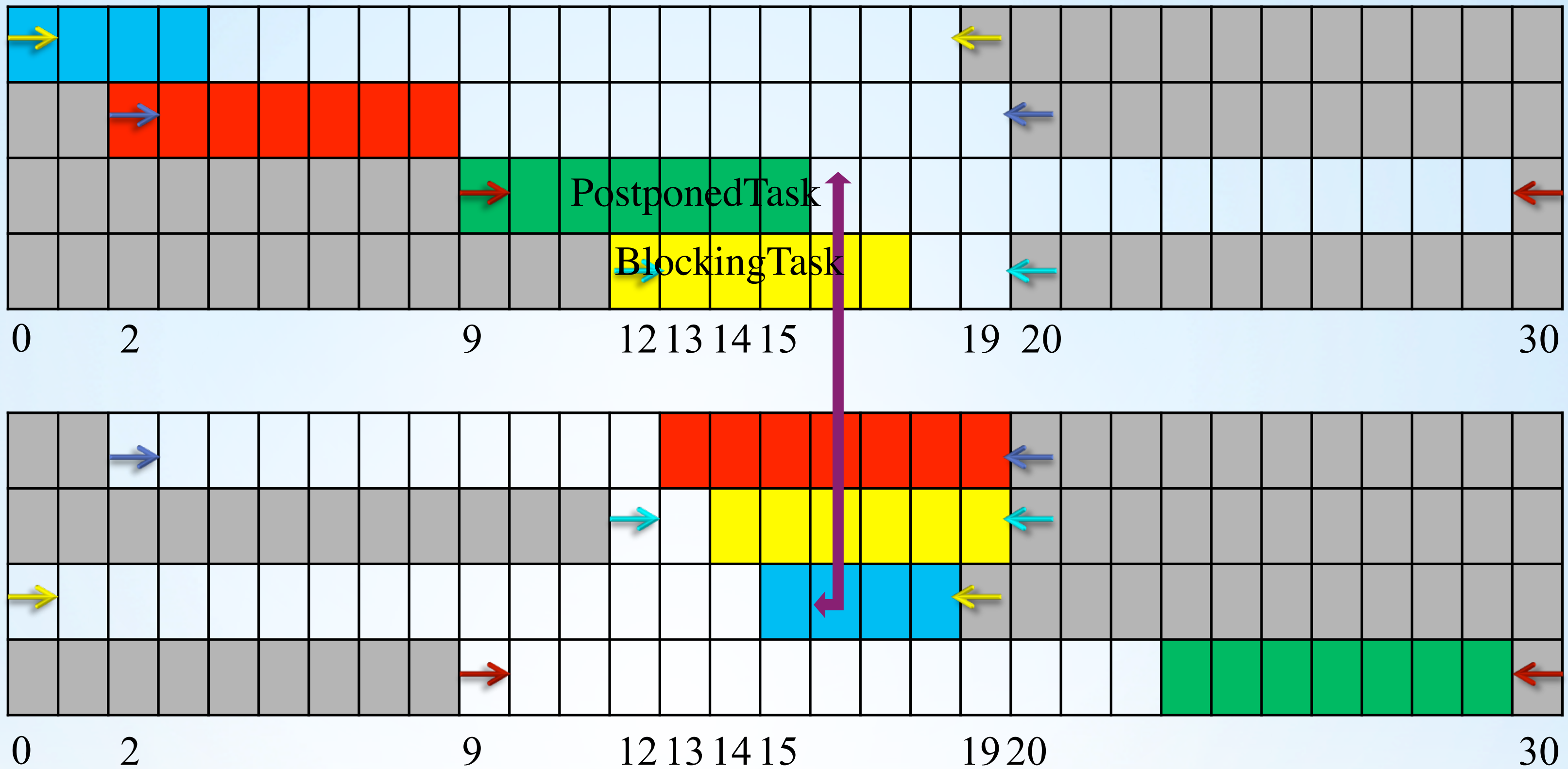
- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_2 < ect_3$? Yes!
- Filtering of the current task (green) will be postponed!



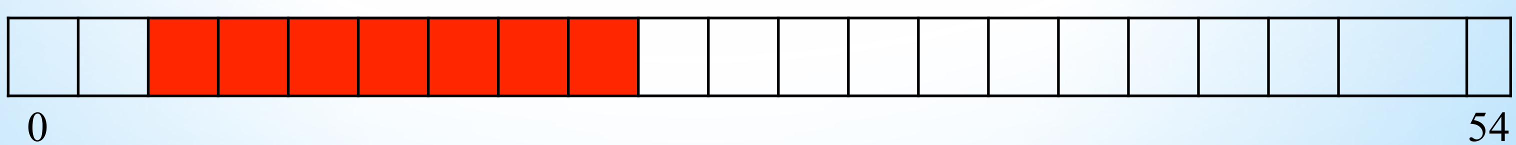
Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times

- The tasks sorted by latest starting times

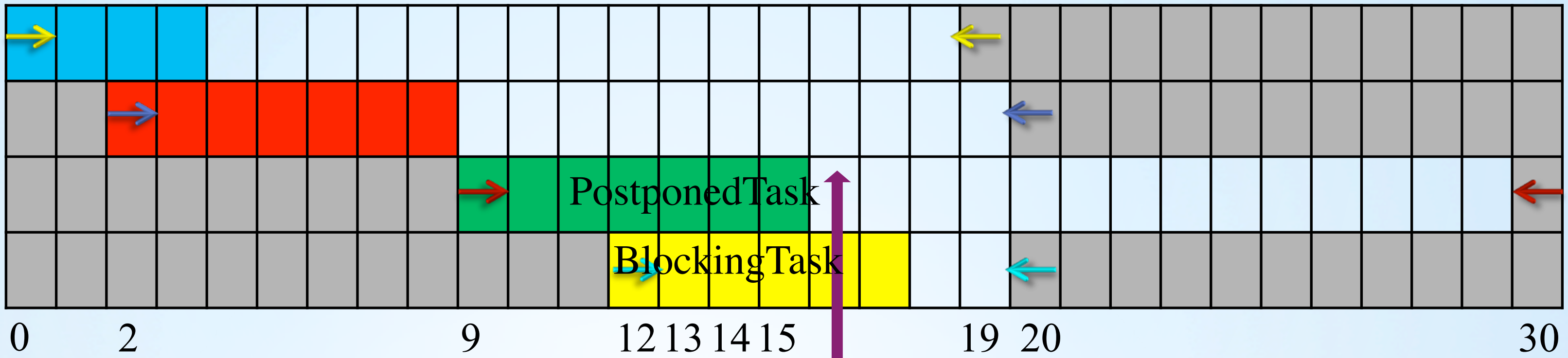


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_3 < ect_3$?

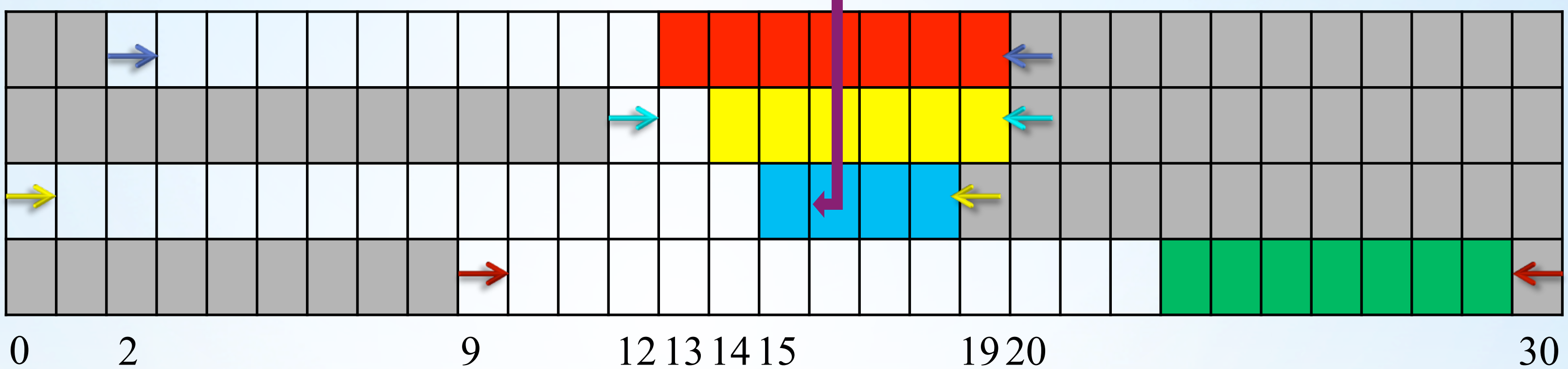


Detectable Precedences (with fixed part)

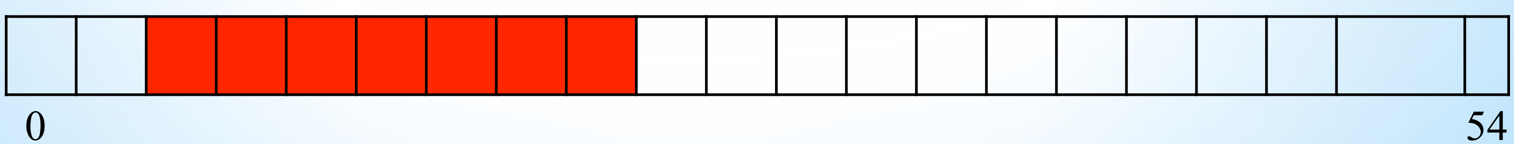
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times



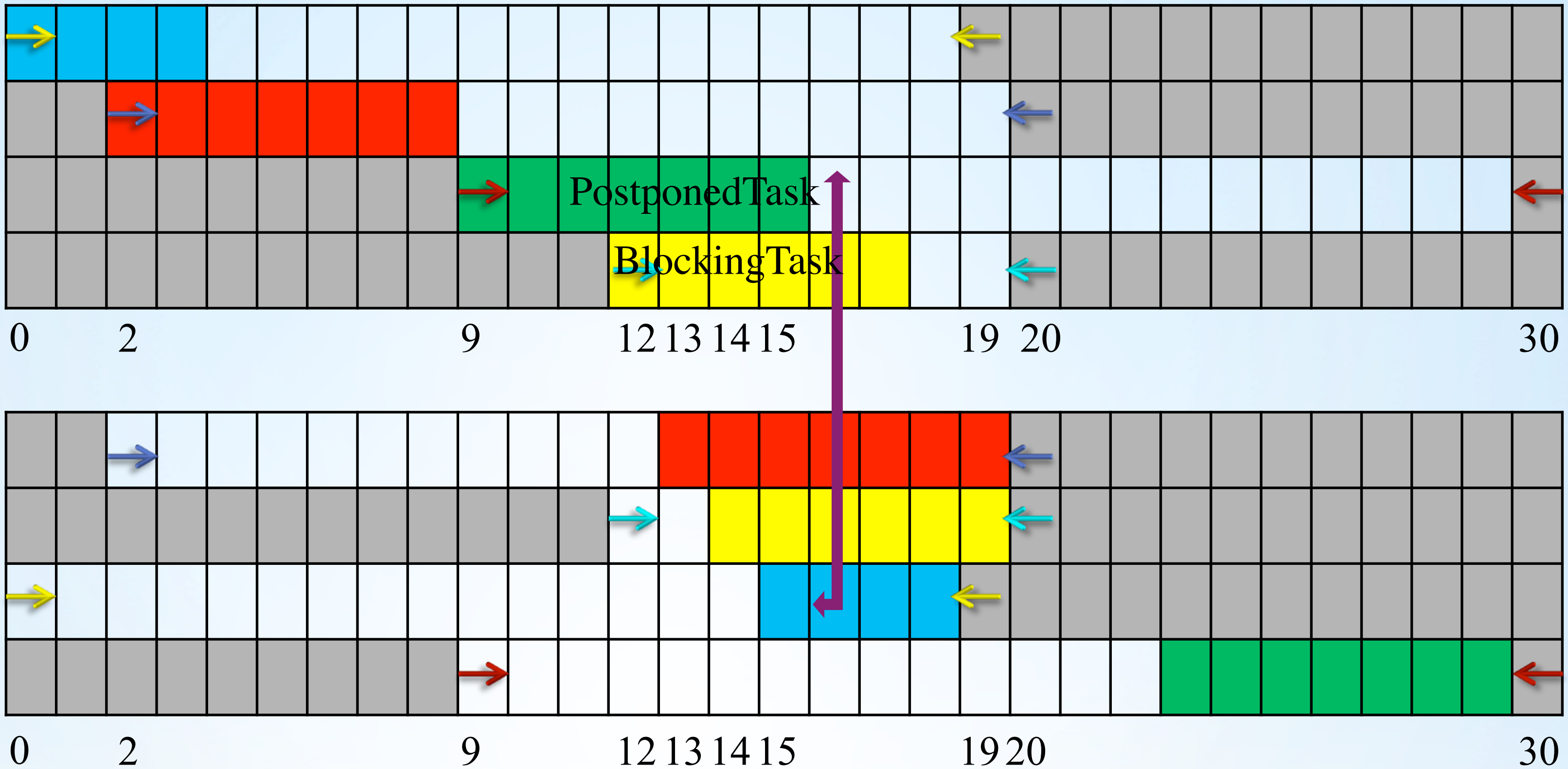
- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_3 < ect_3$? Yes!



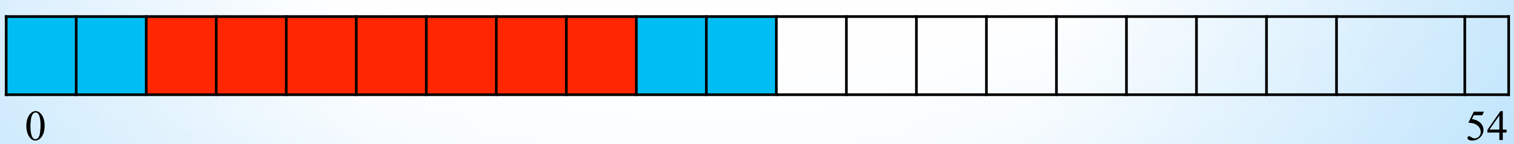
Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times

- The tasks sorted by latest starting times

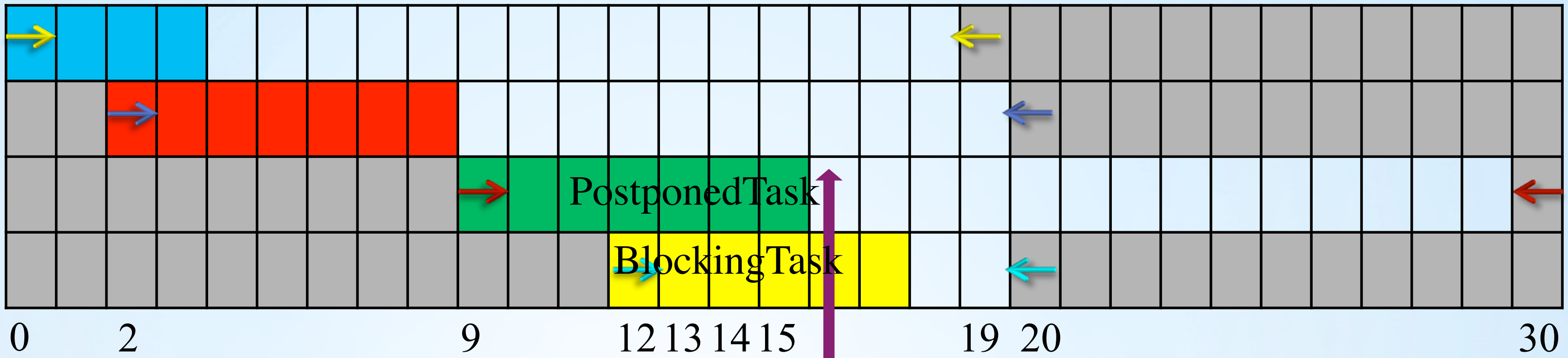


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_3 < ect_3$? Yes!
- The blue task will be scheduled on the time line.

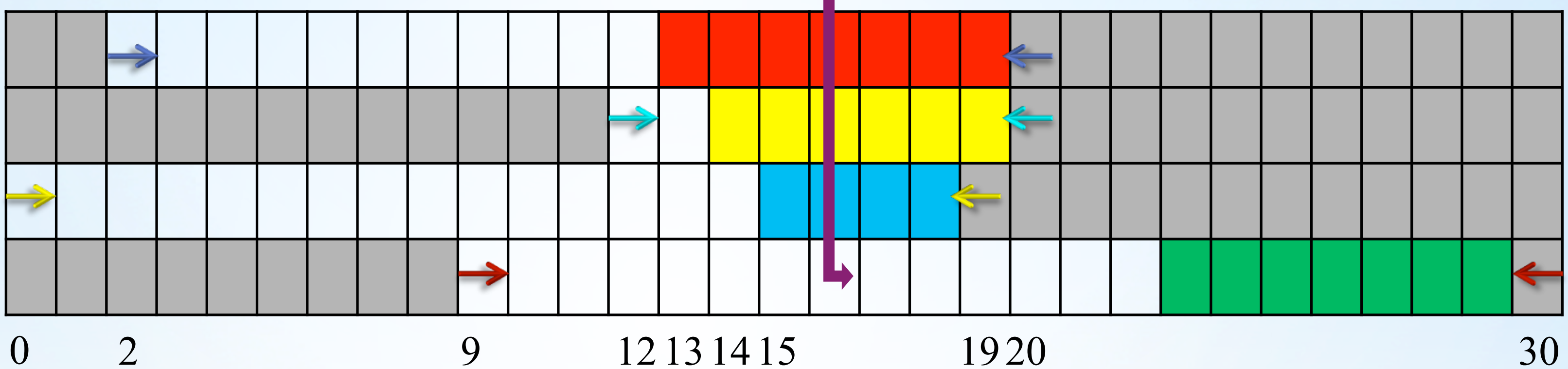


Detectable Precedences (with fixed part)

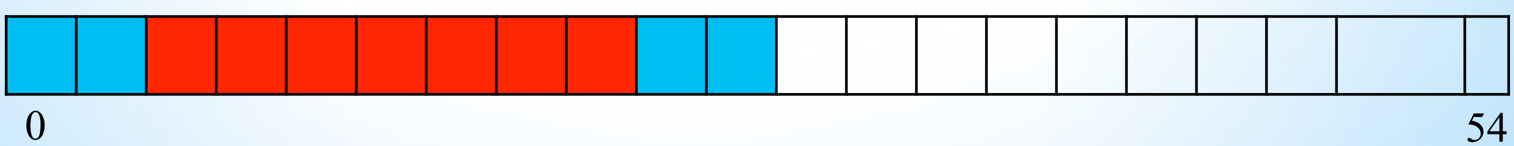
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

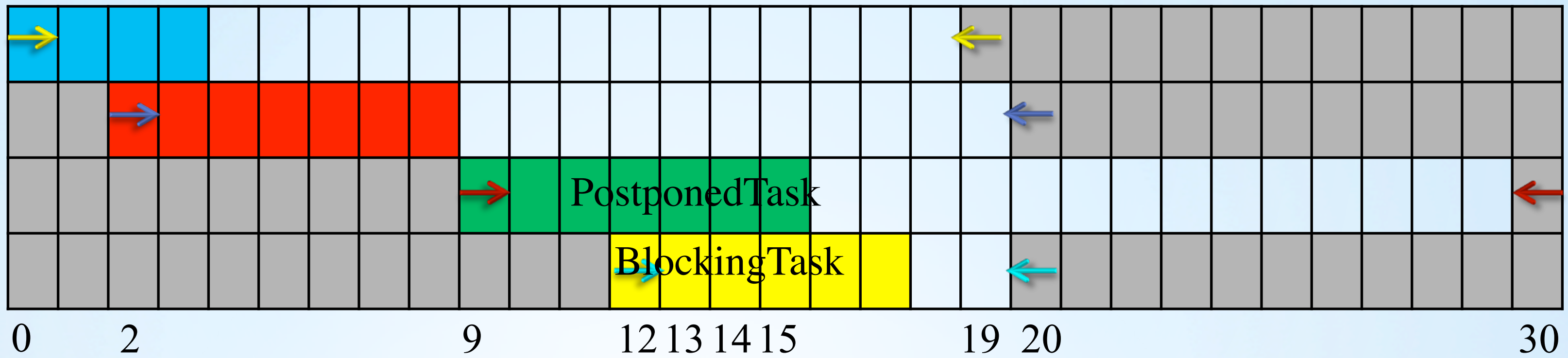


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_4 < ect_3$? No!

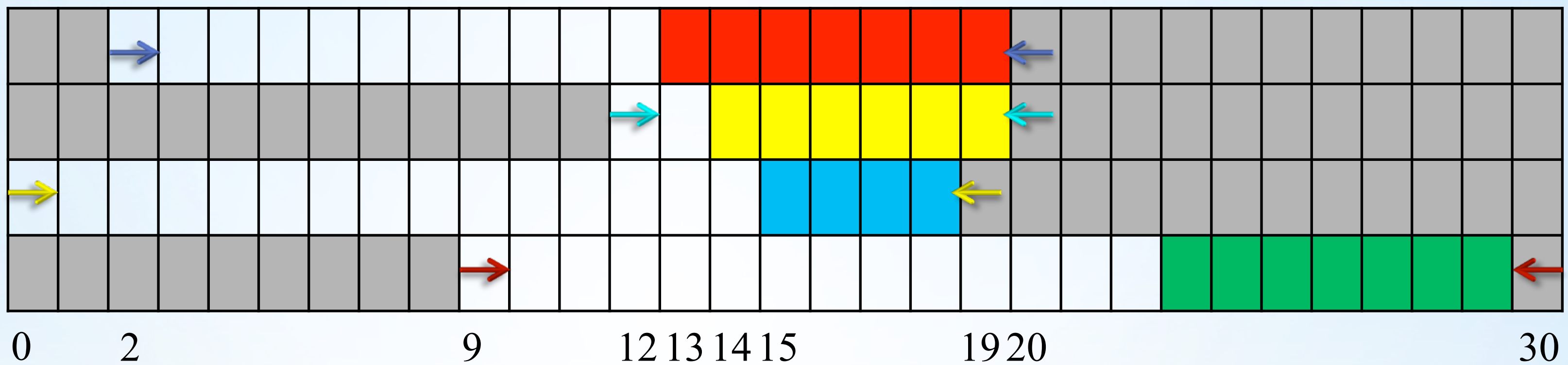


Detectable Precedences (with fixed part)

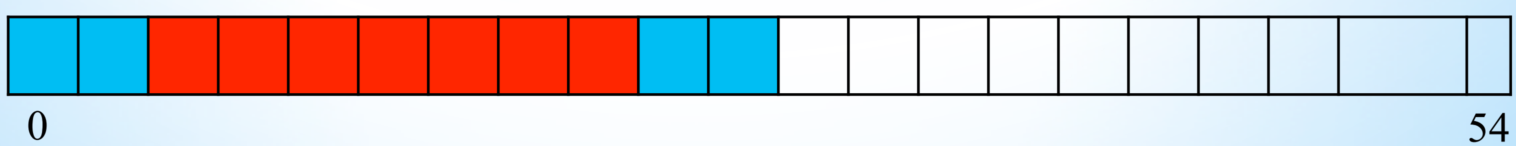
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

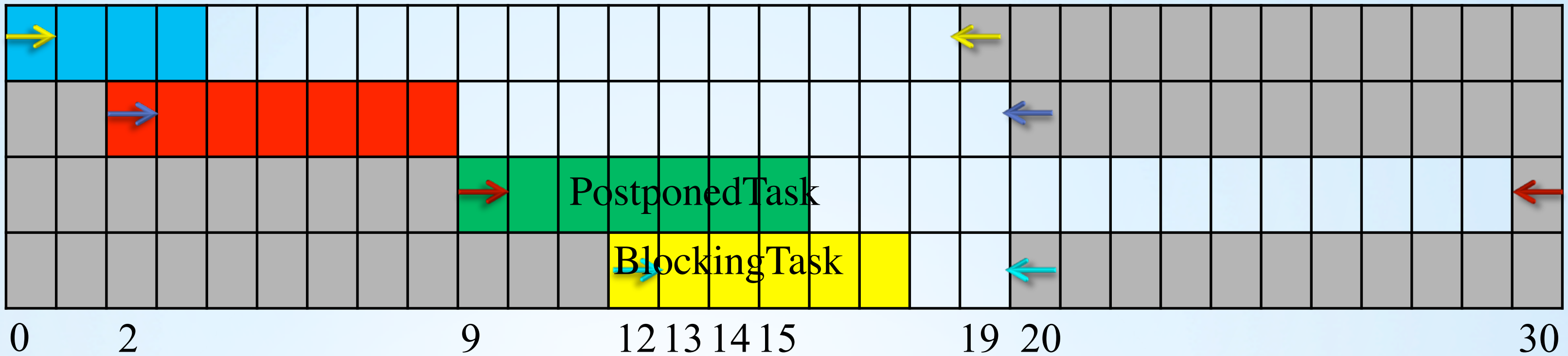


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Processing of the green task is over! Note that it is not filtered yet, since there exists a blocking task which has not been scheduled yet.

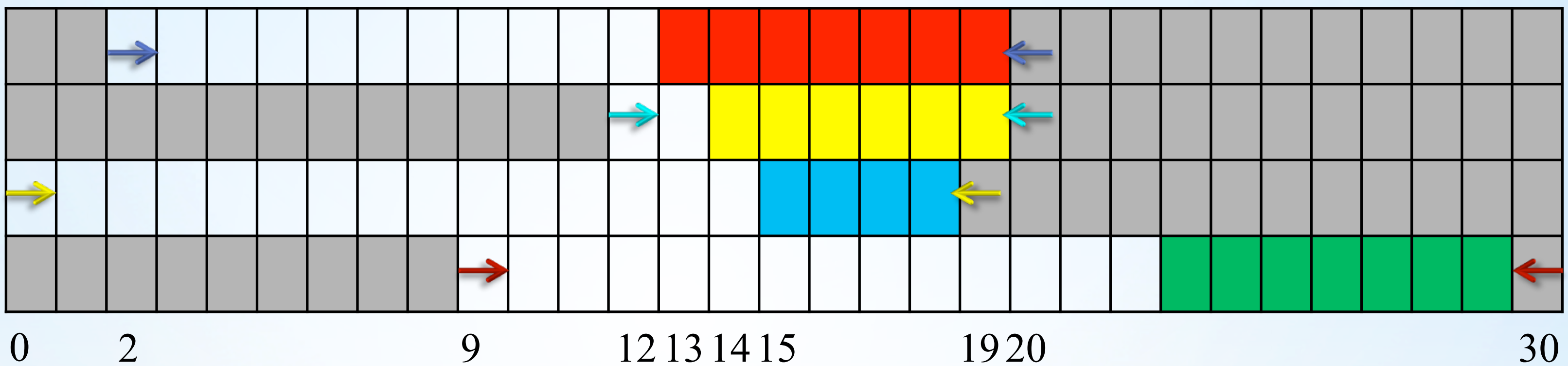


Detectable Precedences (with fixed part)

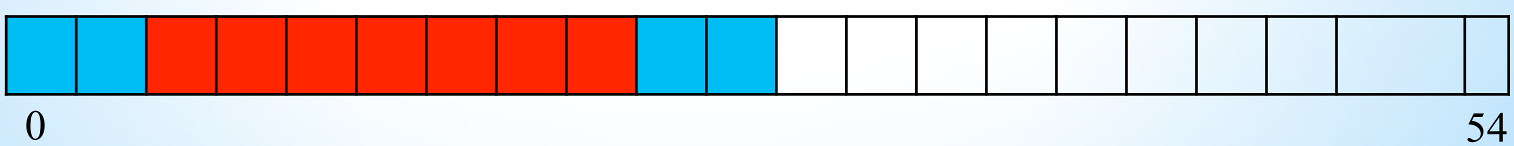
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

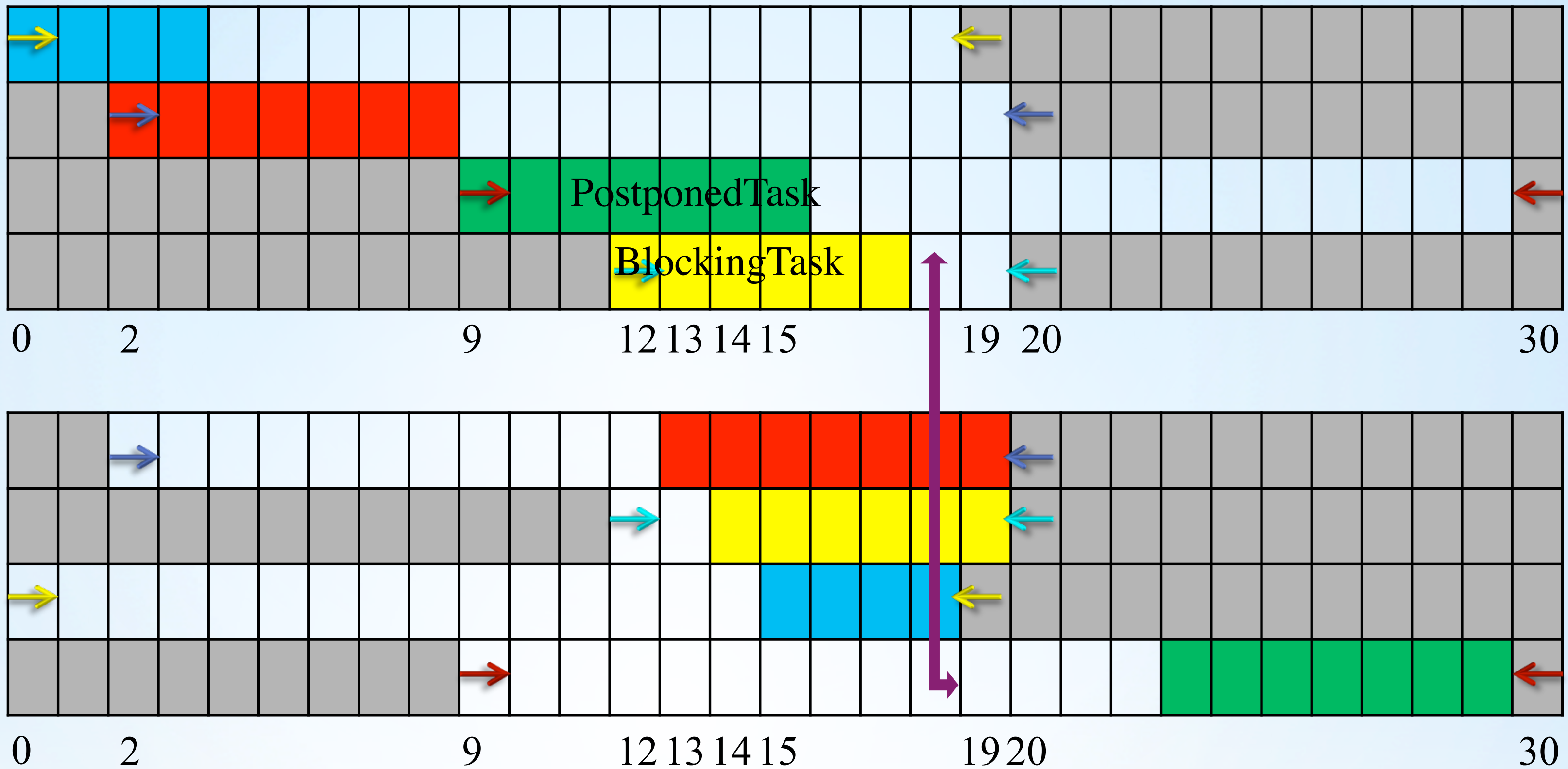


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- It will be filtered after the blocking task is processed.

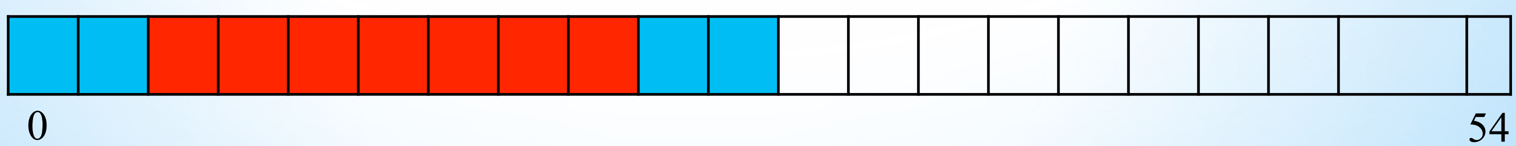


Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times
- The tasks sorted by latest starting times

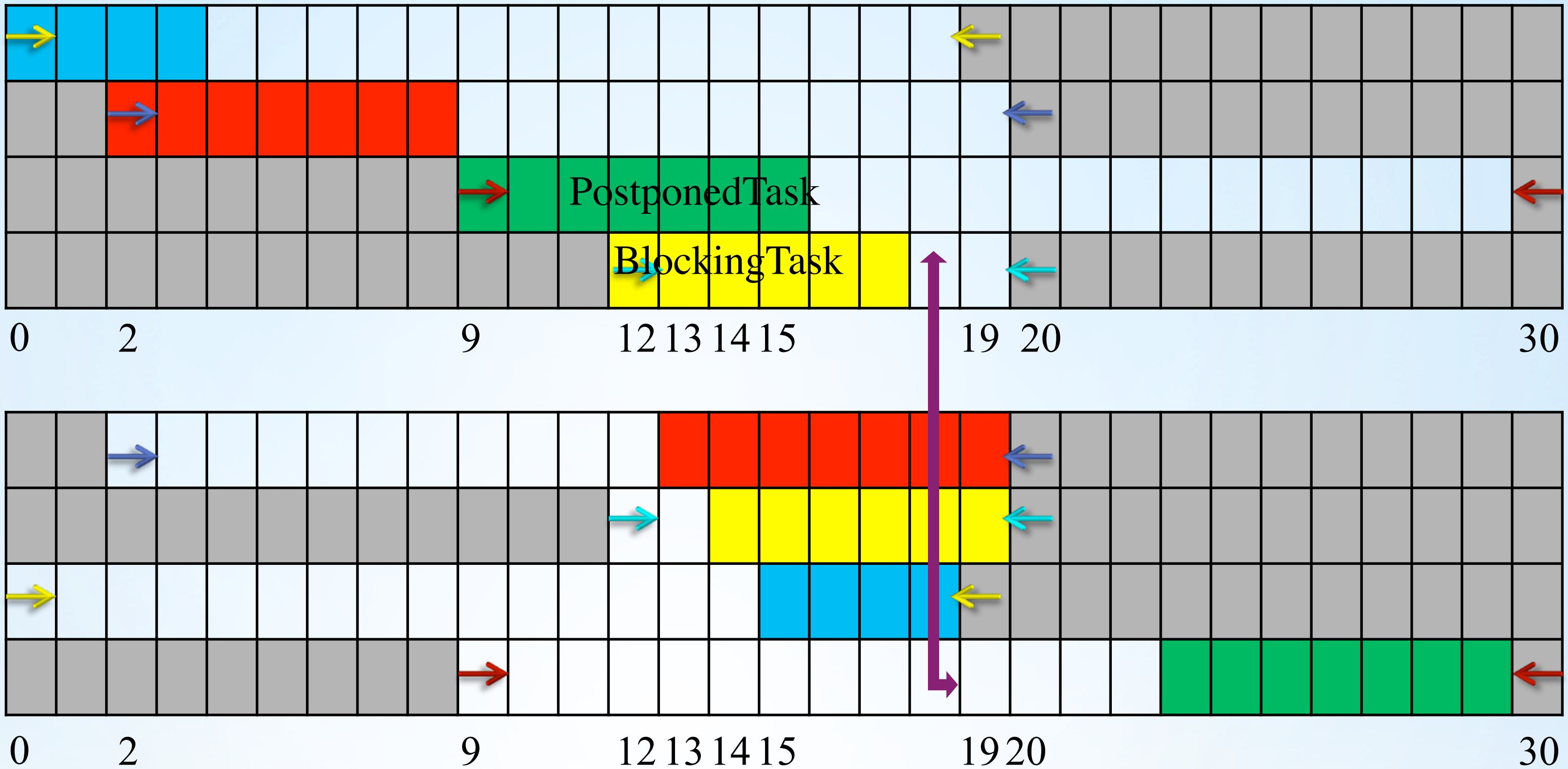


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_4 < ect_4$?



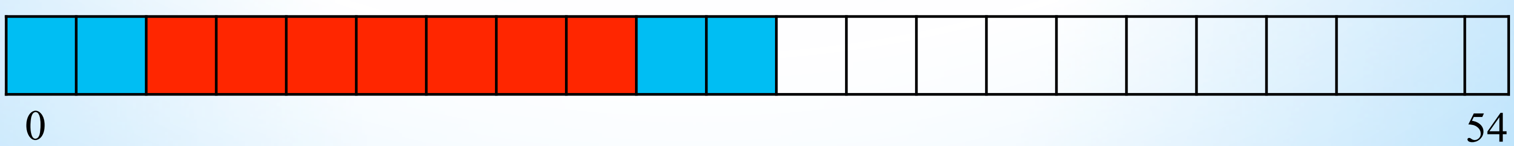
Detectable Precedences (with fixed part)

- The tasks sorted by earliest completion times



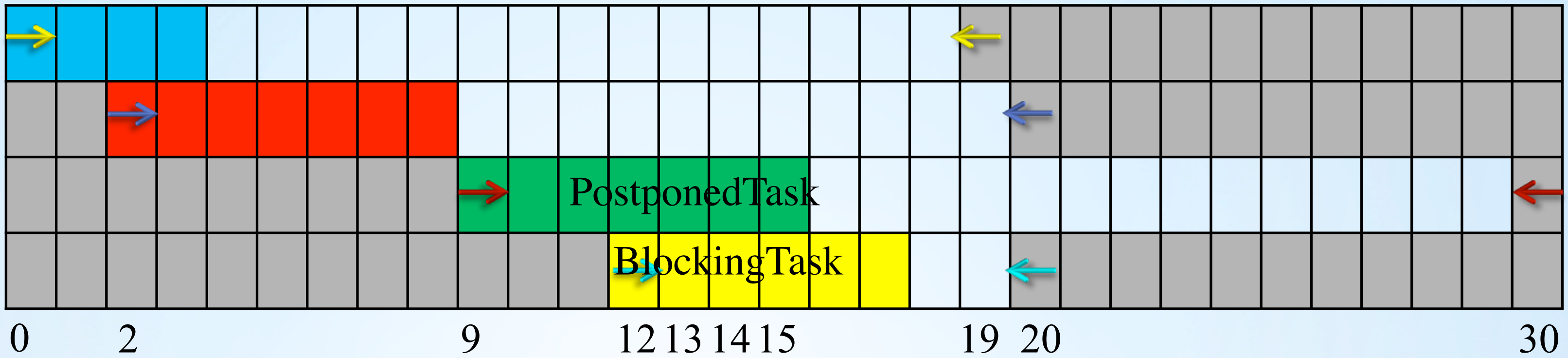
- The tasks sorted by latest starting times

- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Checking if $lst_4 < ect_4$? No!

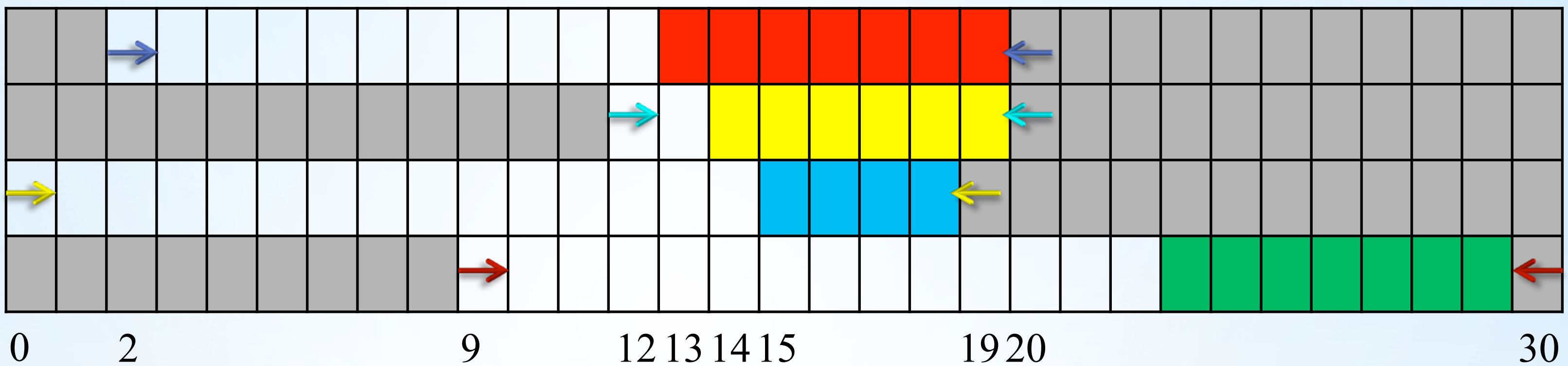


Detectable Precedences (with fixed part)

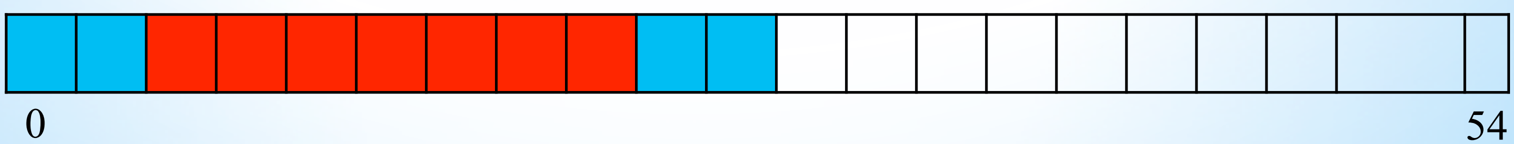
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

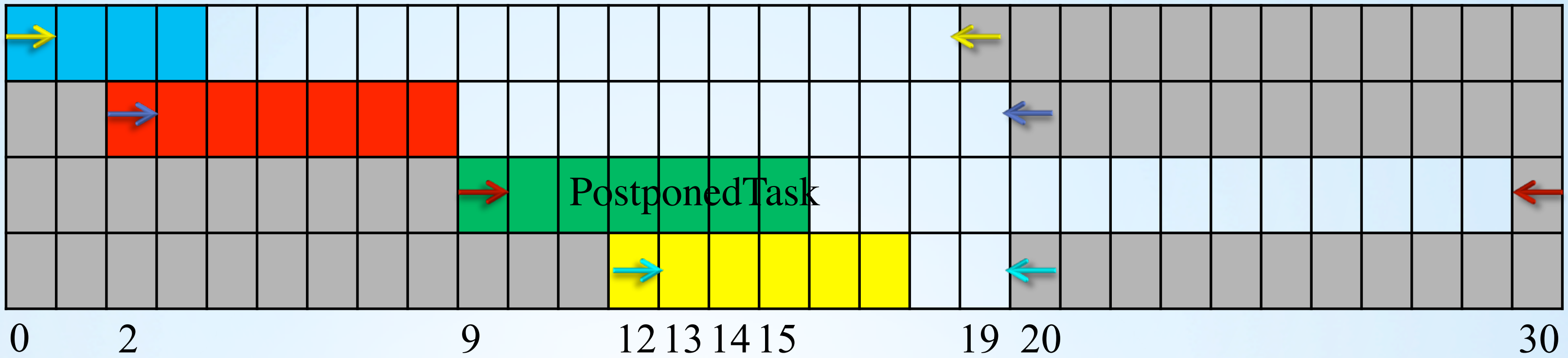


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- The yellow task is the blocking task. It will be first filtered to the earliest completion time of time line.

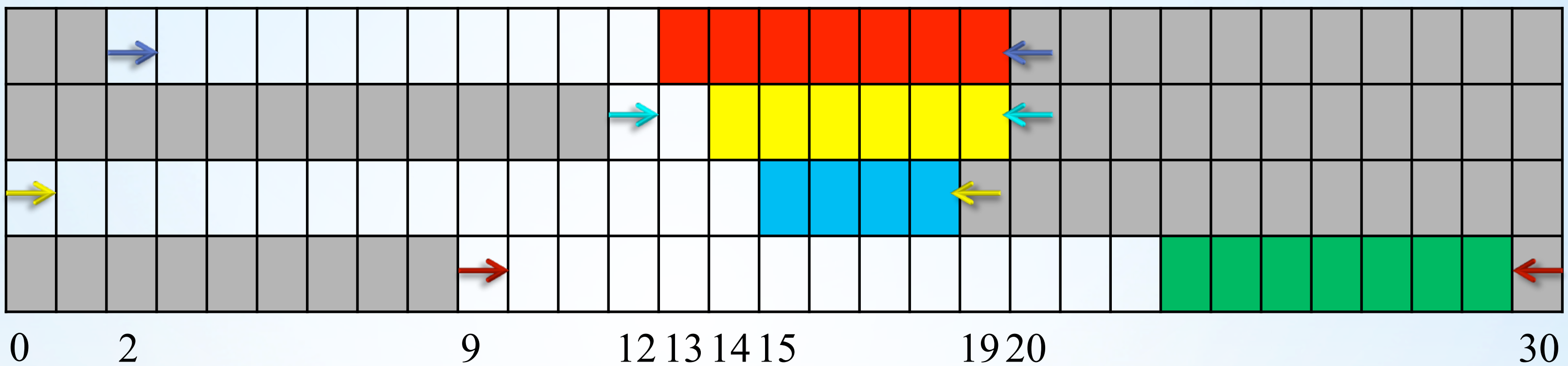


Detectable Precedences (with fixed part)

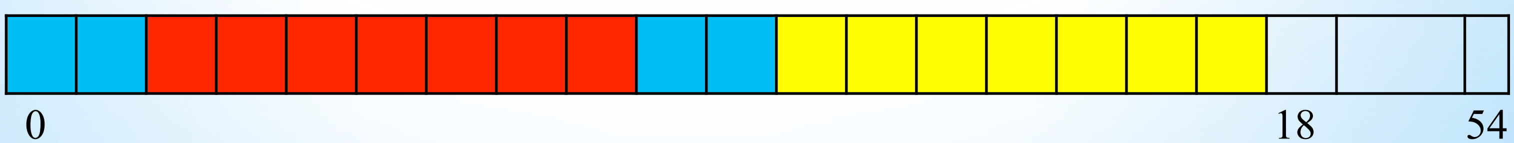
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times

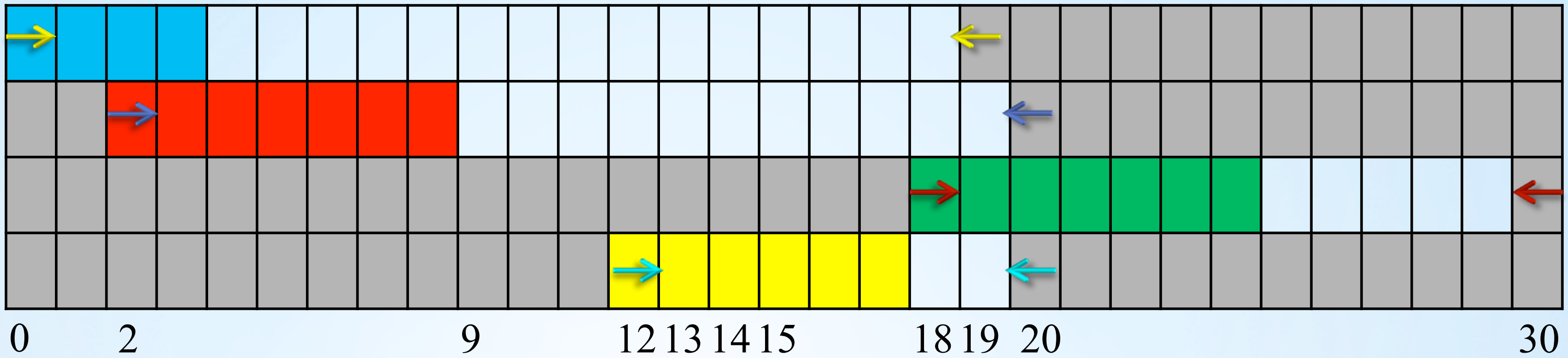


- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- The yellow task is then scheduled on the time line.

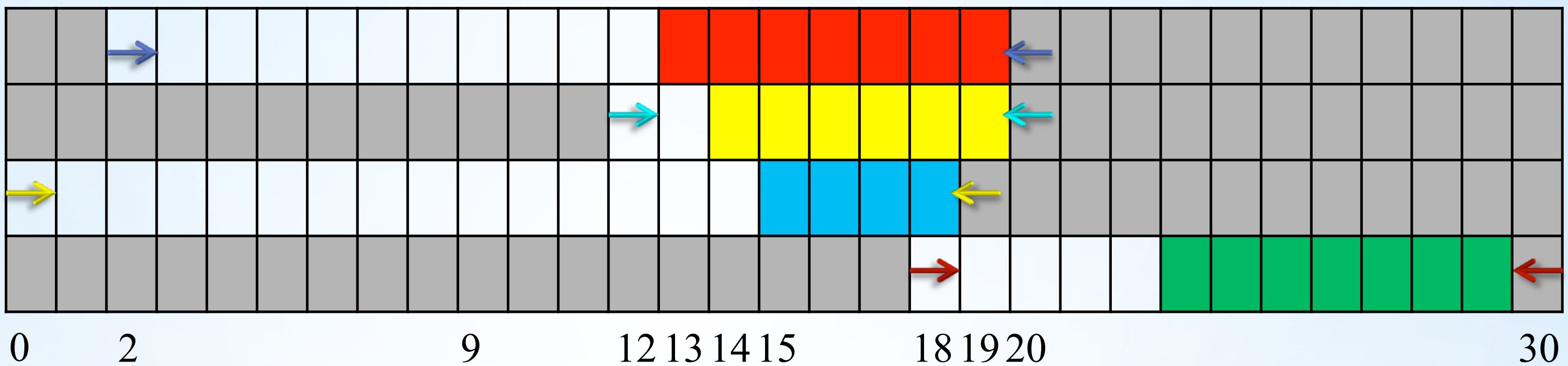


Detectable Precedences (with fixed part)

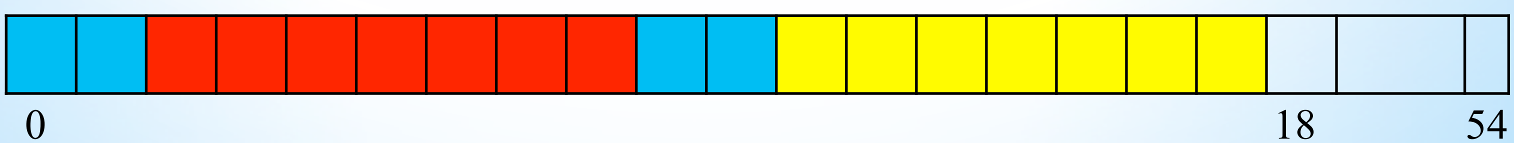
- The tasks sorted by earliest completion times



- The tasks sorted by latest starting times



- Simultaneously iterate over all the tasks i from the first table and on all the tasks k from the second table .
- Now, the postponed task (green) is filtered to the earliest completion time of time line.



OUTLINE

- SCHEDULING
- CONSTRAINT PROGRAMMING
- PRELIMINARIES
- PROPAGATION OF DISJUNCTIVE CONSTRAINT
- EXPERIMENTAL RESULTS
- CONCLUSION

Problem definitions

- To compare the linear algorithm with their counterparts, we ran the experiments on job-shop and open-shop scheduling problems.

- In these problems, *n* jobs consisting of a set of non-preemptive tasks, execute on *m* machines. Each task executes on a predetermined machine with a given processing time.

- In the job-shop problem, the tasks belonging to the same job execute in a predetermined order. In the open-shop problem, the number of tasks per job is fixed to *m* and the order in which the tasks of a job are processed is immaterial.

- In both problems, the goal is to minimize the makespan, *i.e.* the time when the last task completes.

Modeling the problems

- We model the problems with one starting time variable $S_{i;j}$ for each task j of job i.

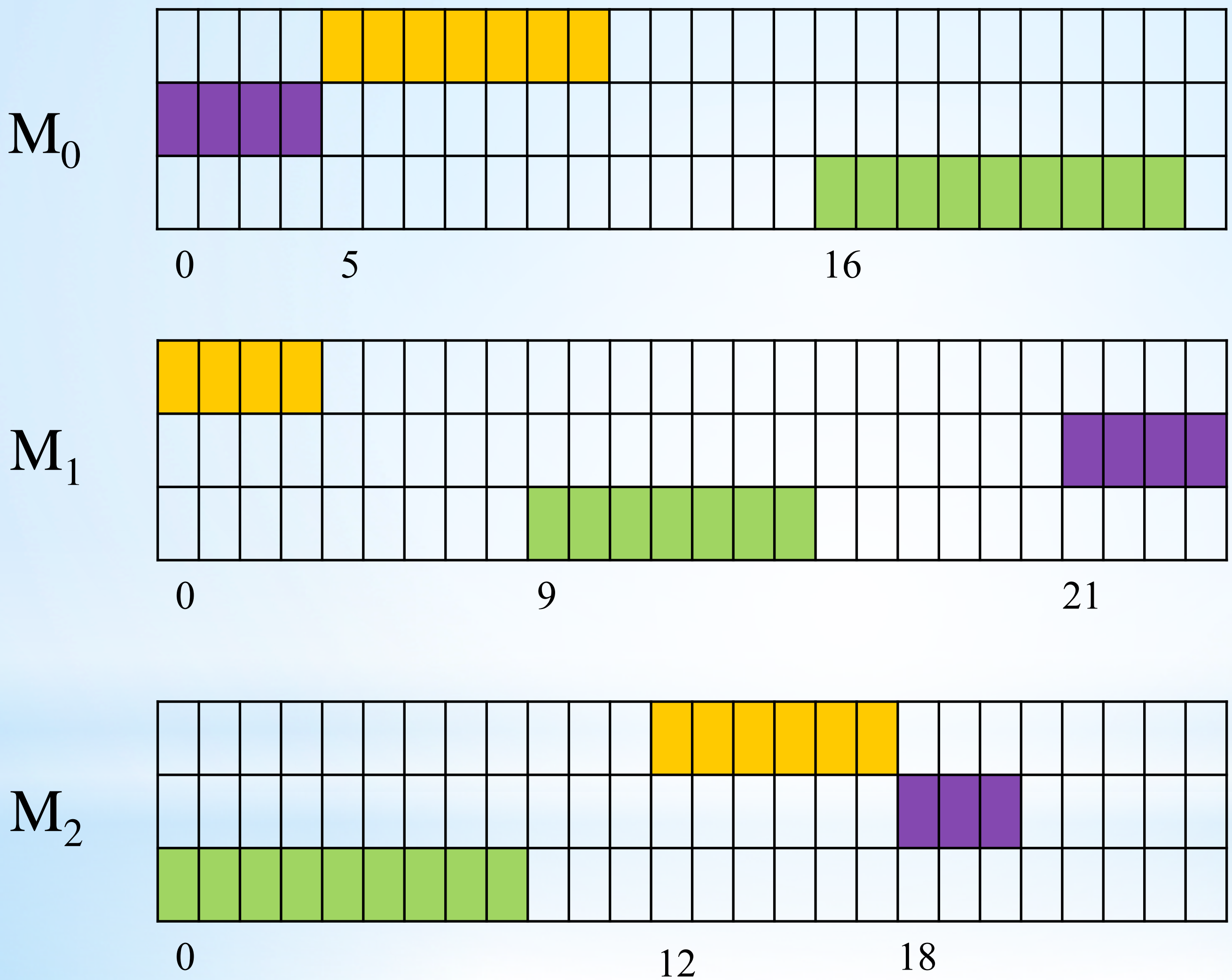
- We post a DISJUNCTIVE constraint over all starting time variables of tasks running on the same machine.

- For the job-shop scheduling problem, we add the precedence constraints $S_{i,j} + p_{i,j} \leq S_{i,j+1}$.

- For the open-shop scheduling problem, we add a DISJUNCTIVE constraint among all tasks belonging to the same job.

- For both problems, there is also a constraint posted to minimize the makespan.

Example of a Job-shop scheduling problem



Experiments

- After 10 minutes of computations, the program halts.
- The problems are not solved to optimality.
- The number of backtracks that occur will be counted.
- We compare two algorithms which explore the same tree in the same order.

Experiments

- A larger portion of the search tree will be traversed within 10 minutes with the faster algorithm.
- The bigger the portion of the search tree which has been explored, the more the number of backtracks, the faster the algorithm!
- Normally, we should notice that our algorithms get faster as the number of tasks increases.
- This expectation was verified by running the experiments on two benchmark problems!

Results for open shop problem

$n \times m$	OverloadCheck	Detectable Precedences	Time Tabling
4×4	0.96	1.00	1.00
5×5	1.03	1.12	1.75
7×7	1.02	1.16	2.09
10×10	1.06	1.33	2.14
15×15	1.03	1.39	2.15
20×20	1.06	1.56	2.17

Results for open shop problem

$n \times m$	OverloadCheck	Detectable Precedences	Time Tabling
4×4	0.96	1.00	1.00
5×5	1.03	1.12	1.75
7×7	1.02	1.16	2.09
10×10	1.06	1.33	2.14
15×15	1.03	1.39	2.15
20×20	1.06	1.56	2.17

- The results of three methods on open-shop benchmark problem with *n* jobs and *m* tasks per job. The numbers indicate the ratio of the cumulative number of backtracks between all instances of size *nm* after 10 minutes of computations.

Results for job shop problem

$n \times m$	OverloadCheck	Detectable Precedences	Time Tabling
10×5	1.07	1.27	2.11
15×5	1.02	1.35	2.27
20×5	1.00	1.55	2.12
10×10	1.01	1.25	2.18
15×10	1.26	1.42	1.97
20×10	1.00	1.47	2.14
30×10	1.08	1.56	2.36
50×10	1.05	1.48	3.18
15×15	0.95	1.48	2.16
20×15	1.04	1.61	2.13
20×20	1.09	1.46	1.71

Results for job shop problem

$n \times m$	OverloadCheck	Detectable Precedences	Time Tabling
10×5	1.07	1.27	2.11
15×5	1.02	1.35	2.27
20×5	1.00	1.55	2.12
10×10	1.01	1.25	2.18
15×10	1.26	1.42	1.97
20×10	1.00	1.47	2.14
30×10	1.08	1.56	2.36
50×10	1.05	1.48	3.18
15×15	0.95	1.48	2.16
20×15	1.04	1.61	2.13
20×20	1.09	1.46	1.71

- The results of three methods on job-shop benchmark problem with *n* jobs and *m* tasks per job. The numbers indicate the ratio of the cumulative number of backtracks between all instances of size *nm* after 10 minutes of computations.

OUTLINE

- SCHEDULING
- CONSTRAINT PROGRAMMING
- PRELIMINARIES
- PROPAGATION OF DISJUNCTIVE CONSTRAINT
- EXPERIMENTAL RESULTS
- CONCLUSION

Conclusion

- Thanks to the constant time operation of the Union-Find data structure, we designed a new data structure, called time line, to speed up filtering algorithms for the Disjunctive constraint.

Conclusion

- Thanks to the constant time operation of the Union-Find data structure, we designed a new data structure, called time line, to speed up filtering algorithms for the Disjunctive constraint.
- We came up with three faster algorithms to filter the disjunctive constraint.

Algorithm	**Previous complexity**	**Now complexity**
Time-Tabling	$O(n\log(n))$ (Ouellet & Quimper)	$O(n)$ (Fahimi & Quimper)
Overload check	$O(n\log(n))$ Vilím	$O(n)$ (Fahimi & Quimper)
Detectable precedences	$O(n\log(n))$ Vilím	$O(n)$ (Fahimi & Quimper)


Thank you!