Constraint Systems

Constraint Based Scheduling

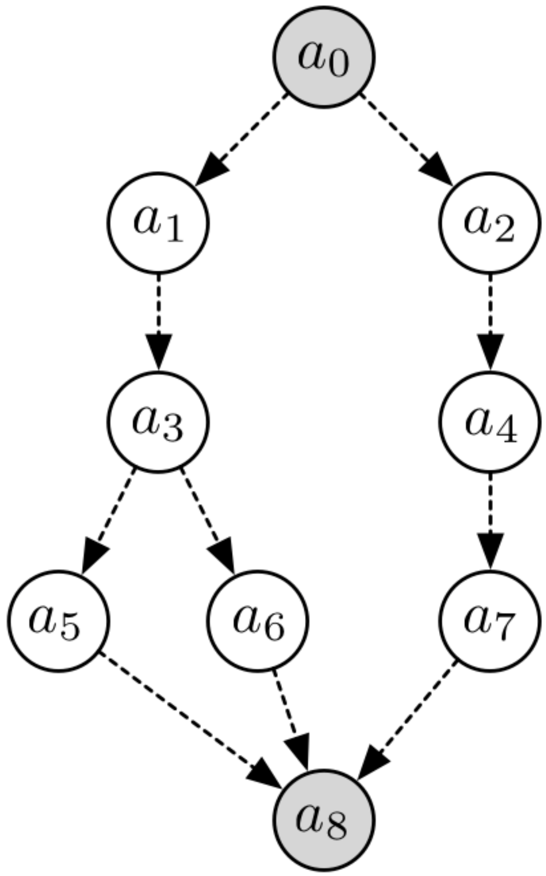
A Target Problem: RCPSP

The Resource Constrained Project Scheduling Problem:

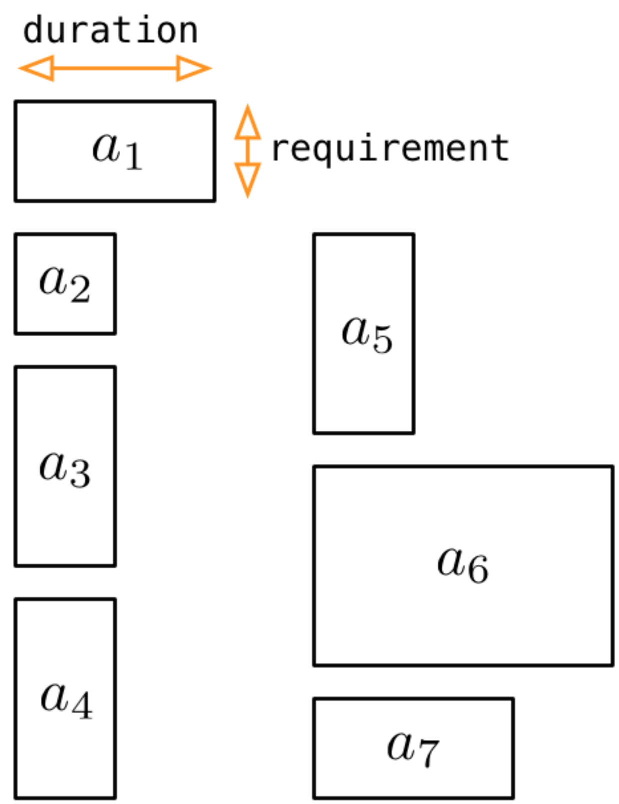
- We have a set of n activities
- Each activity i has fixed duration d_i
- Activities are connected by end-to-start precedence relations
- There are m resources
- Each resource k has fixed capacity c_k
- Each activity requires an amount $r_{i,k}$ of resource k
- requires $= r_{i,k}$ resource units are locked while the activity runs

Let's see a sample instance...

A Target Problem: RCPSP



A single resource, with $c_0 = 2$



A Target Problem: RCPSP

The network of activities/precedences is called Project Graph

- Typically: fake start/end activities
- Fake = 0 duration, 0 requirements
- They can be disregarded in CP models

Goal:

- Build a schedule
  - Assign a start time to all activities
  - Satisfy all constraints
- Minimize the project completion time (makespan)

A Target Problem: RCPSP

Many practical applications:

- Large scale construction projects
- Research/development projects
- Production planning
- Parallel software optimization
- Code optimization (compile time optimization)
- ...

Can we tackle this problem using CP?

A CP Model for the RCPSP

Which variables (i.e. how to model decisions)?

Natural approach: a start time variable for each activity

$$s_i \in \{0..eoh\}$$

- eoh is a safe "End Of Horizon":

$$eoh = \sum_{i=0}^{n-1} d_i$$

- There is always a schedule with makespan $\leq eoh$
- Unless the resource constraints are trivially infeasible

A CP Model for the RCPSP

How to model the problem objective?

- Makespan = project completion time = largest end time:

$$\min z = \max_{i=0..n-1} (s_i + d_i)$$

How to model the precedence constraints?

- If there is a precedence between activities i and j:

$$s_i + d_i \leq s_j$$

A CP Model for the RCPSP

How to model the resource constraints?

If $c_k = 1$, then activities should not overlap

- Formally, for each pair of activities i, j s.t. $r_{i,k} = r_{j,k} = 1$:

$$(s_i + d_i \leq s_j) \vee (s_j + d_j \leq s_i)$$

- A resource with unary capacity is called "disjunctive"
- We have seen this on the Job Shop Scheduling Problem

But what if $\mathbf{c_k > 1}$?

A CP Model for the RCPSP

If $c_k > 1$ finding a good model is difficult

Some possibilities

- A sum constraint for each time point
- A sum constraint for each activity start

Both are complicated and lead to weak propagation

- This is one of the reason why MILP is not good for the RCPSP
- A notable exception: the approach works for SAT based solvers

Is there an alternative? We can use a global constraint!

Constraint Systems

Constraint Based Scheduling:
The CUMULATIVE Constraint

The CUMULATIVE Constraint

We can use a new global constraint!

Basic idea: one global constraint for each resource

$$\text{CUMULATIVE}(s, d, r, c)$$

- s is a vector of start time variables s_i
- d is a vector of durations d_i
- r is a vector of requirements r_i
- c is the capacity

The durations and the requirements can be either scalars of variables

The CUMULATIVE Constraint

The cumulative constraint enforces consistency on:

$$\sum_{\substack{i=0..n-1,\\ s_i \le t < s_i + d_i}} r_i \le c, \qquad \forall t = 0..\max\{s_i + d_i\}$$

- In brief: the resource capacity is never exceeded

Which kind of consistency?

Feasibility checking is easy when all s_i are fixed (as usual):

- Check the resource usage only at the activity starts
- Rationale: resource usage can increase only at the start times

Unfortunately, filtering is NP-hard!

The CUMULATIVE Constraint

Cumulative is an NP-hard constraint

Proof (just an idea):

- If we could enforce GAC on s_i…
- …Then we could solve the decision version the bin-packing problem…
- …The bin-packing problem is NP-hard

Practical consequences:

- All filtering algorithms are suboptimal
- Typically: weak, bound-based, forms of consistency

The CUMULATIVE Constraint

Some filtering algorithms for CUMULATIVE:

- Disjunctive filtering
- Timetable filtering
- Edge-finder
- Not-first/not-last rules
- Timetable edge-finding
- Energetic reasoning
- ...

Why so many?

- Filtering is always incomplete
- The CUMULATIVE constraint is very important!

Timetable Filtering

As an example, we will describe timetable filtering

- One of the weakest algorithms
- But also one of the fastest ones

80% of the times, this is all you need

Key idea #1: rely on a minimum usage profile

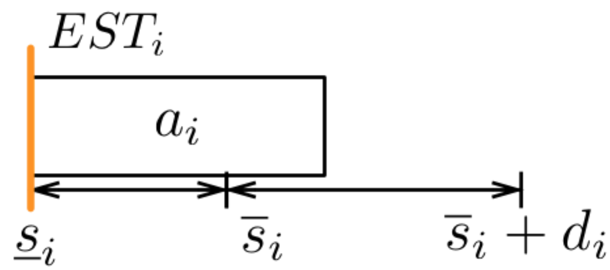
- Min. usage profile = guaranteed min. consumption per time point
- Use the profile to determine bounds for the s_i variables

Before presenting the algorithm we need some preliminary notions…

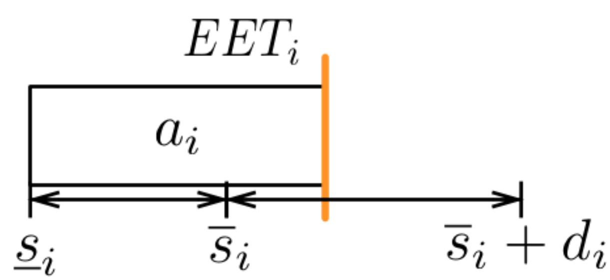
Timetable Filtering

Some notable time point for each activity:

- Earliest Start Time: $EST_i = \underline{s}_i$



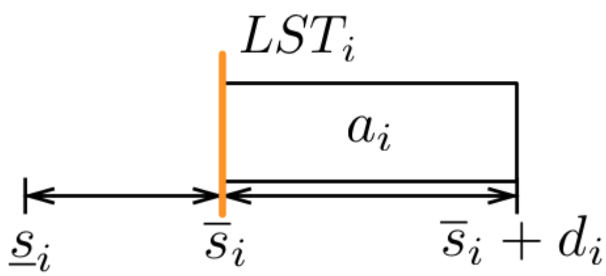
- Earliest End Time: $EET_i = \underline{s}_i + d_i$



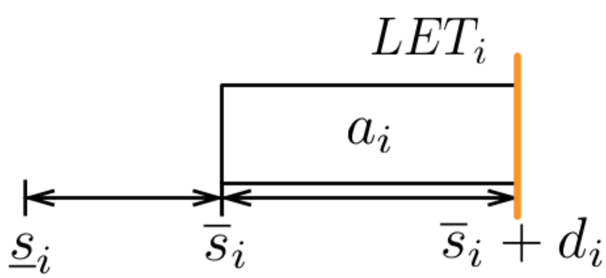
Timetable Filtering

Some notable time point for each activity:

- Latest Start Time: $LST_i = \bar{s}_i$



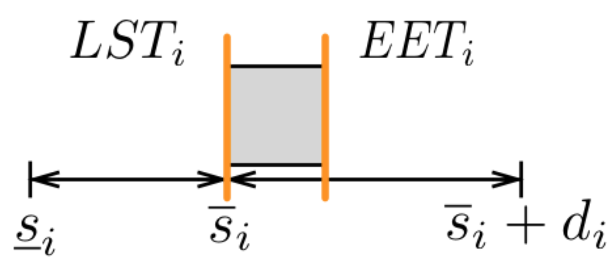
- Latest End Time: $LET_i = \bar{s}_i + d_i$



Timetable Filtering - Compulsory Parts

If we have $\mathbf{LST_i} < \mathbf{EET_i}$, then:

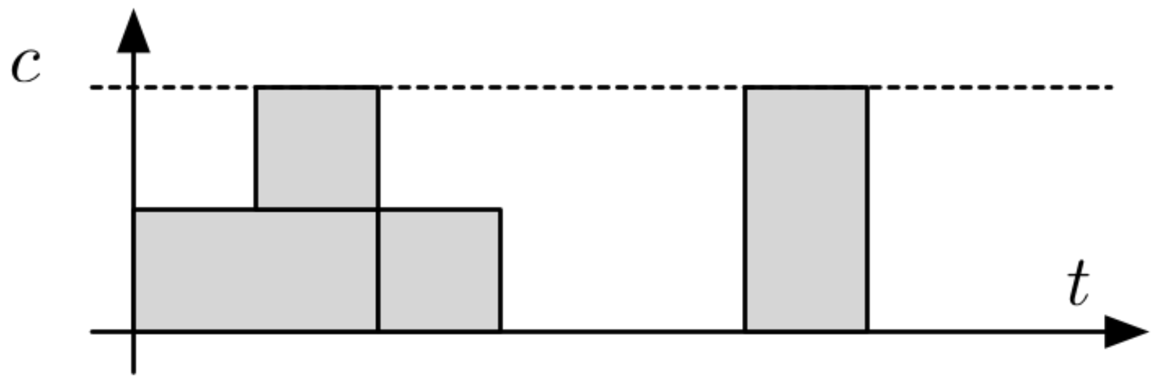
- In the interval LST_i, EET_i, activity i will certainly be executing
- Therefore, r_i units of the resource will be locked



We say that the activity has a compulsory part

Timetable Filtering - Min. Usage Profile

By aggregating all compulsory parts we get the usage profile:



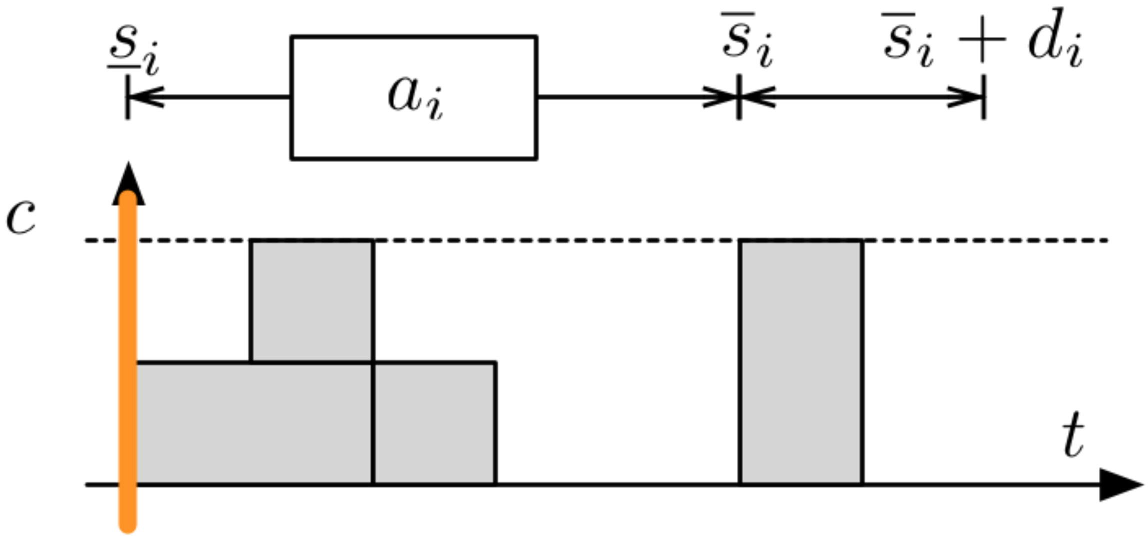
- For each time instant: minimum guaranteed resource usage

Key idea #2: For each activity i:

- Sweep the timeline (SWEEP is also the propagator name)
- Search for a suitable start time
- Update the domain of s_i accordingly

Timetable Filtering

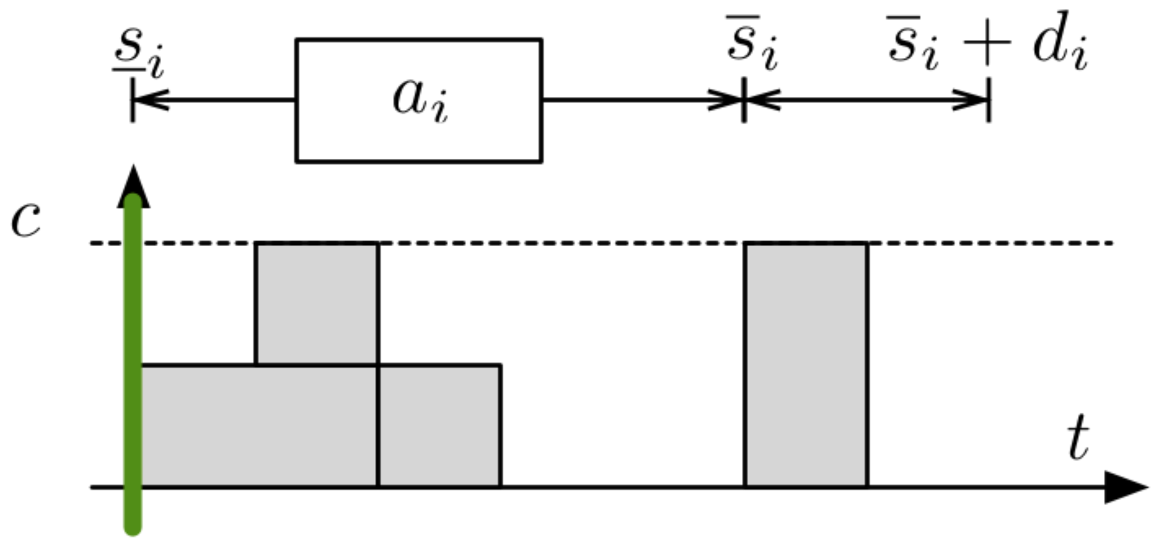
Timetable filtering for a single activity $\mathbf{a_i}$



- We keep a timeline cursor
- The initial position of the cursor is $\underline{s}_i$

Timetable Filtering

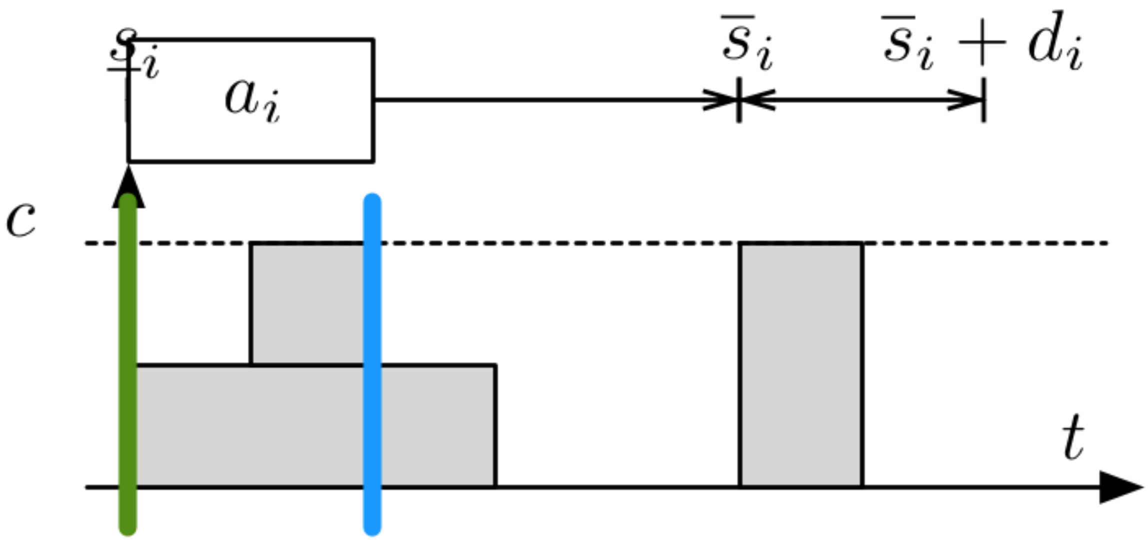
Timetable filtering for a single activity a_i



- We check whether there is enough capacity available
- In case there is, the cursor switches to checking mode
- We store the current cursor position into a variable s^*

Timetable Filtering

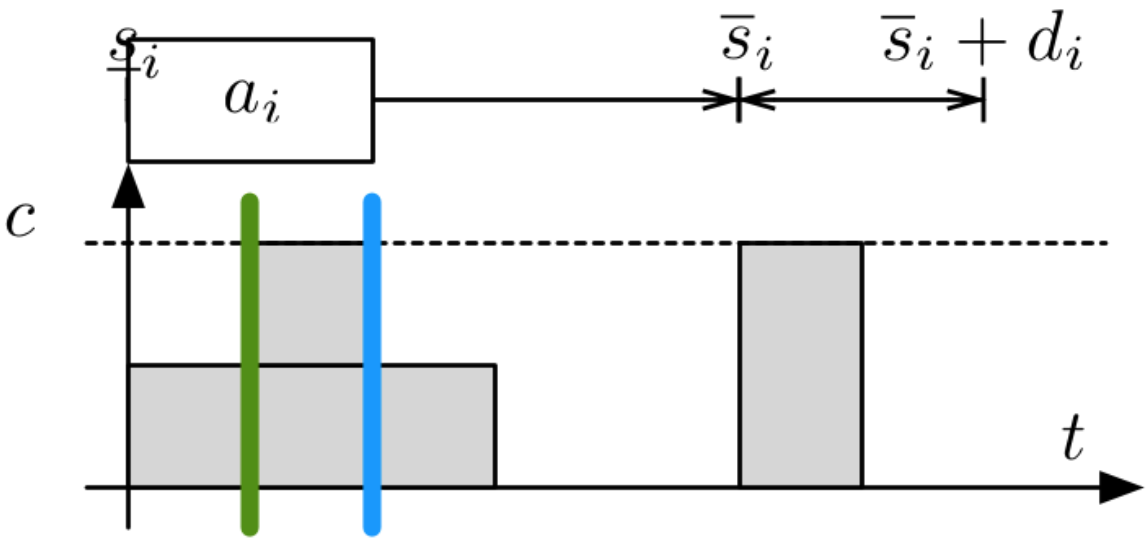
Timetable filtering for a single activity a_i



- In checking mode, we test whether s^* is a valid start time
- This is true iff there is enough capacity in the interval $[s^*, s^* + d_i[$
- Thus, we keep on checking until we reach $s^* + d_i$

Timetable Filtering

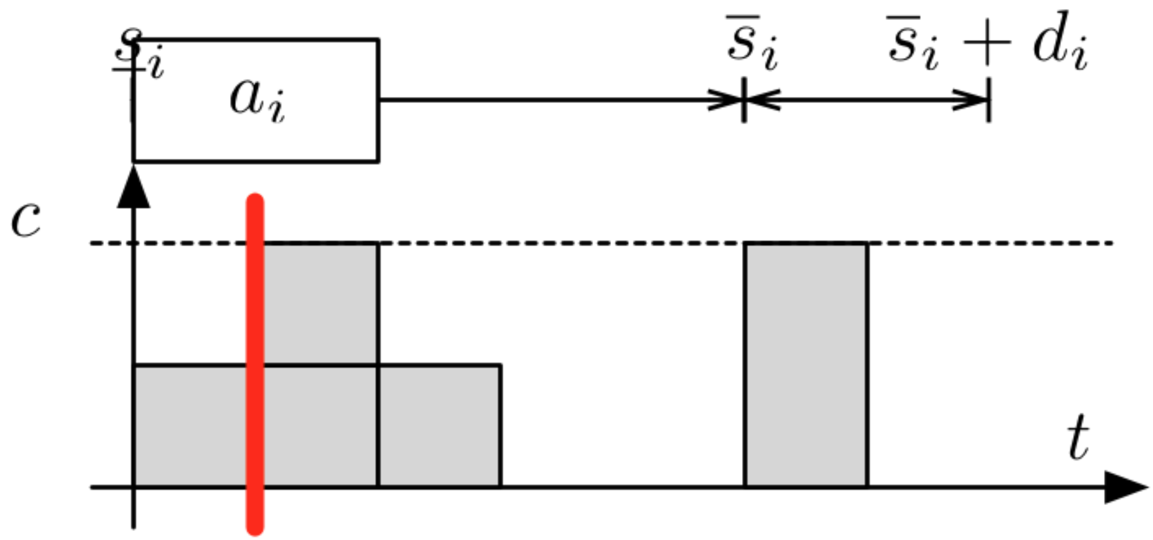
Timetable filtering for a single activity a_i



- In checking mode, we move only between Latest Start Times
- Rationale: compulsory parts begin only at LSTs
- Hence, the resource usage can increase only at LSTs

Timetable Filtering

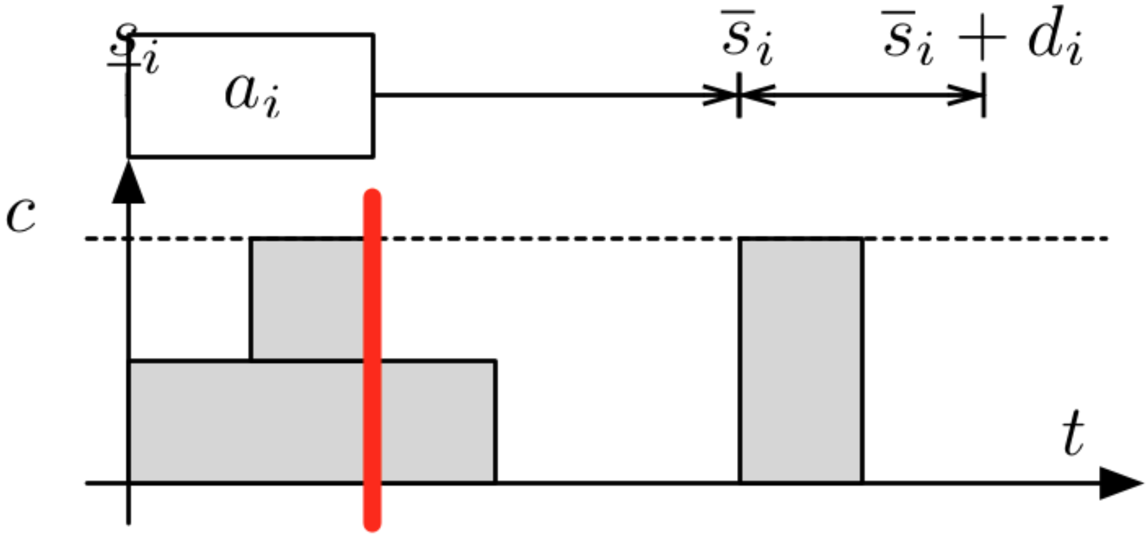
Timetable filtering for a single activity a_i



- If there is not enough capacity, we switch to seeking mode
- In seeking mode, we have concluded that s^* is not a valid start
- Hence, we search for another candidate start time

Timetable Filtering

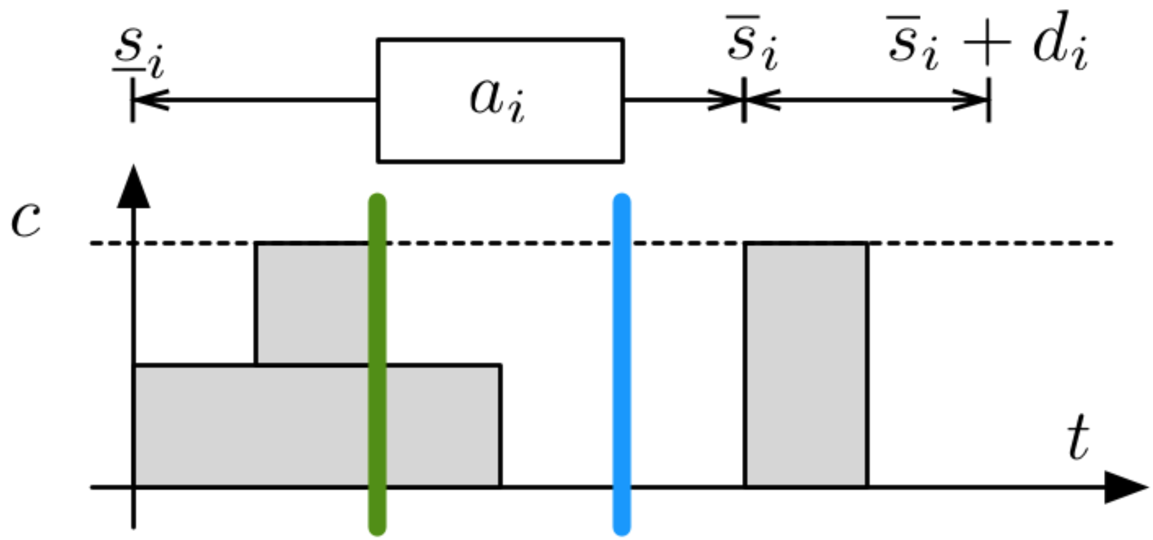
Timetable filtering for a single activity a_i



- In seeking mode, we move only between Earliest End Times
- Rationale: compulsory parts end at EETs
- Hence, the resource usage can decrease only at EETs

Timetable Filtering

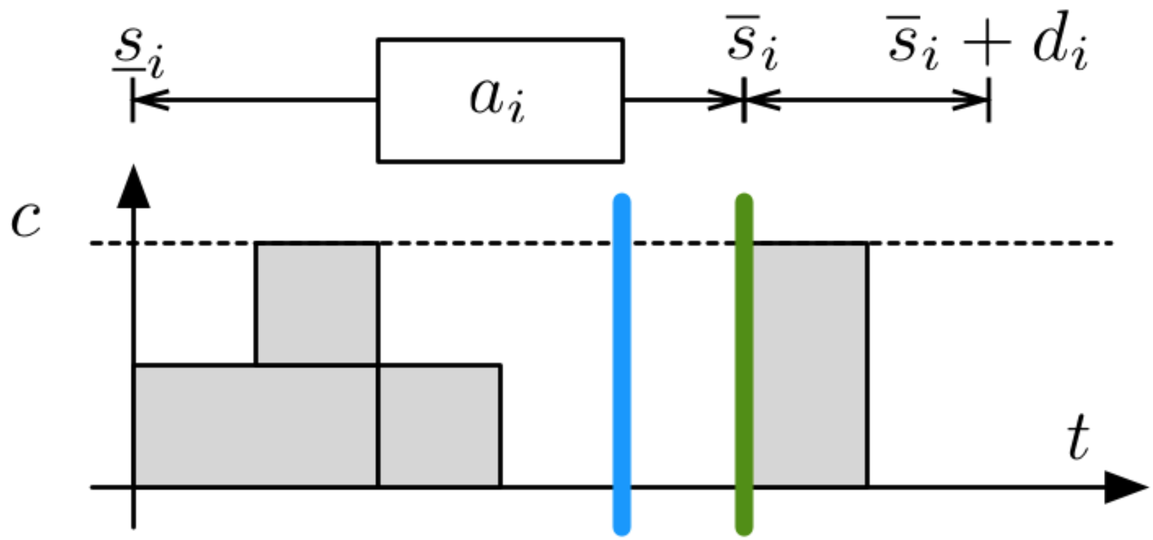
Timetable filtering for a single activity a_i



- If there is enough capacity, we switch to checking mode
- We store the current cursor position in s^*
- We start checking the interval $[s^*, s^* + d_i[$

Timetable Filtering

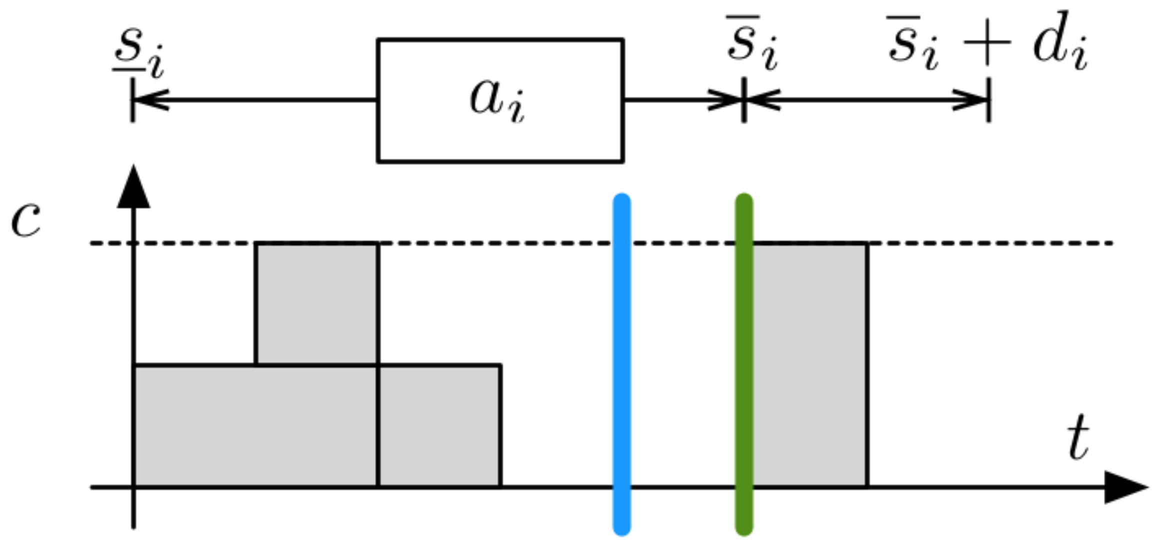
Timetable filtering for a single activity a_i



In checking mode, sweeping can proceed up to $\overline{s}_i + d_i$

Timetable Filtering

Timetable filtering for a single activity $\mathbf{a_i}$



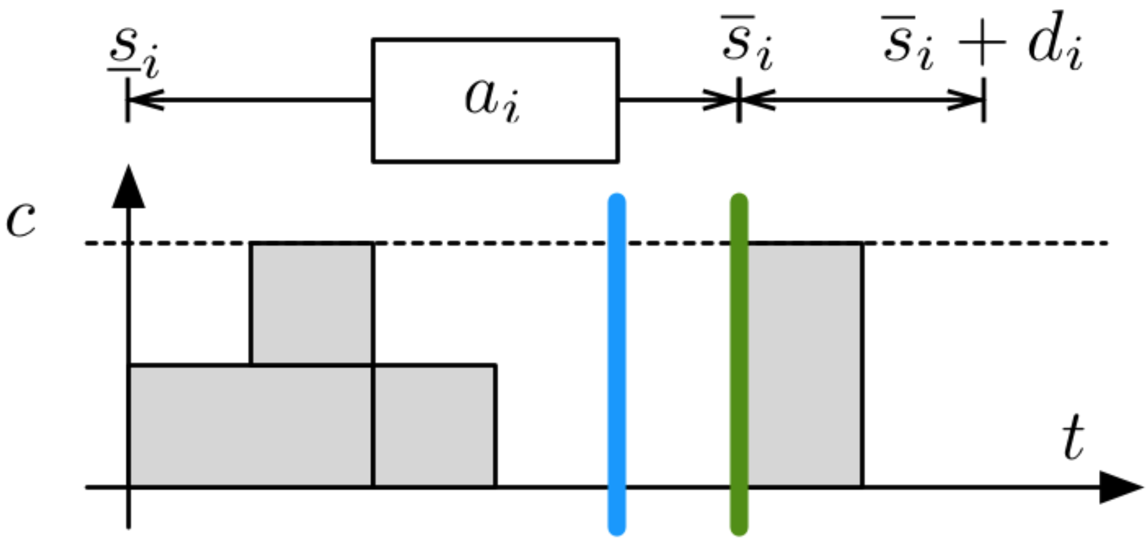
If at some point we reach $s^* + d_i$ while in checking mode...

- ...We can prune $D(s_i)$, setting s^* as the new EST_i

This is the case in our example

Timetable Filtering

Timetable filtering for a single activity $\mathbf{a_i}$



If at some point we surpass $\overline{s}_i$ while in seeking mode...

- ...We can immediately fail

Timetable Filtering

Some final considerations:

- Upper bounds on the start variables can be computed similarly
- The profile can be computed in $O(n \log n)$
  - Approach: sort and then scan
- Sweeping has complexity $O(n)$
- We need to filter n activities

Overall, the algorithm has complexity $O(n^2)$

Other Forms of Cumulative Filtering

A few other propagators for CUMULATIVE deserve a mention:

Edge Finder

- Considers pairs (Ω, i)
  - $\Omega =$ a set of activities
  - $i =$ the activities to be filtered
- Detects if activity i cannot precede any activity in Ω
- Updates $D(s_i)$ based on that information
- Complexity $O(k\, n^2)$ ($k =$ num. distinct requirements)

A very effective approach in some cases (typically: tight time windows)

- Time window $= [\underline{s}_i, \overline{s}_i]$ (in this context)

Other Forms of Cumulative Filtering

A few other propagators for CUMULATIVE deserve a mention:

Energetic Reasoning

- Energy = required resource × time
- Reason on the required energy in certain time intervals
- Detect overusage ⇒ fail
- Detect potential overusage ⇒ prune

An interesting, but seldom useful approach:

- PRO: Subsumes both timetabling at edge
- CON: Complexity $O(n^3)$ (too high in many cases)

Other Forms of Cumulative Filtering

A few other propagators for cumulative deserve a mention:

Timetable Edge Finding

- A more modern approach
- Mixes ideas from timetabling and edge finder
- Stronger than Edge Finder
- Complexity $O(n^2)$
  - Convergence is reached in multiple iterations
- Does not dominate timetabling

Constraint Systems

Constraint Based Scheduling:
Search Strategies for scheduling problems

Search Strategies for Scheduling Problems

How do we search for a solution for the RCPSP?

Several design decisions to take:

- Which variable shall we pick?
- Which value shall we assign?
- How should we backtrack?
- ...

Simple things first, so we start from...

Value Selection for the RCPSP

How should we select the values for the s_i variables?

- The objective is to minimize the makespan
- Increasing a s_i (others s_j untouched) cannot improve the makespan

Consequence: selecting $\underline{s}_i$ seems a good idea

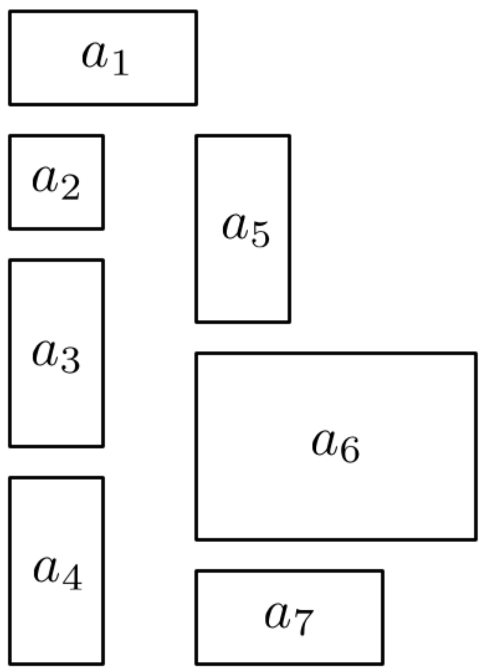
This is true not only for the RCPSP:

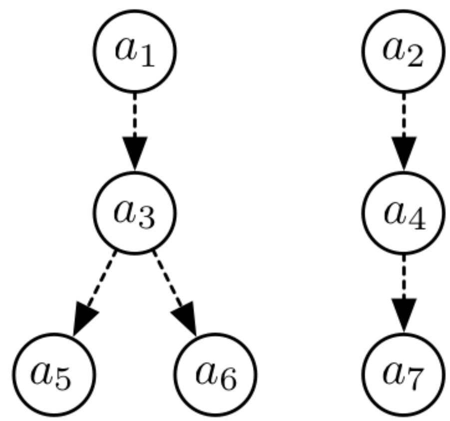
- Many scheduling problems have so-called regular cost metrics
- Regular = increasing a single s_i cannot improve the cost

Variable Selection for the RCPSP

How should we select the branching variable?

It's easier to reason on an example:

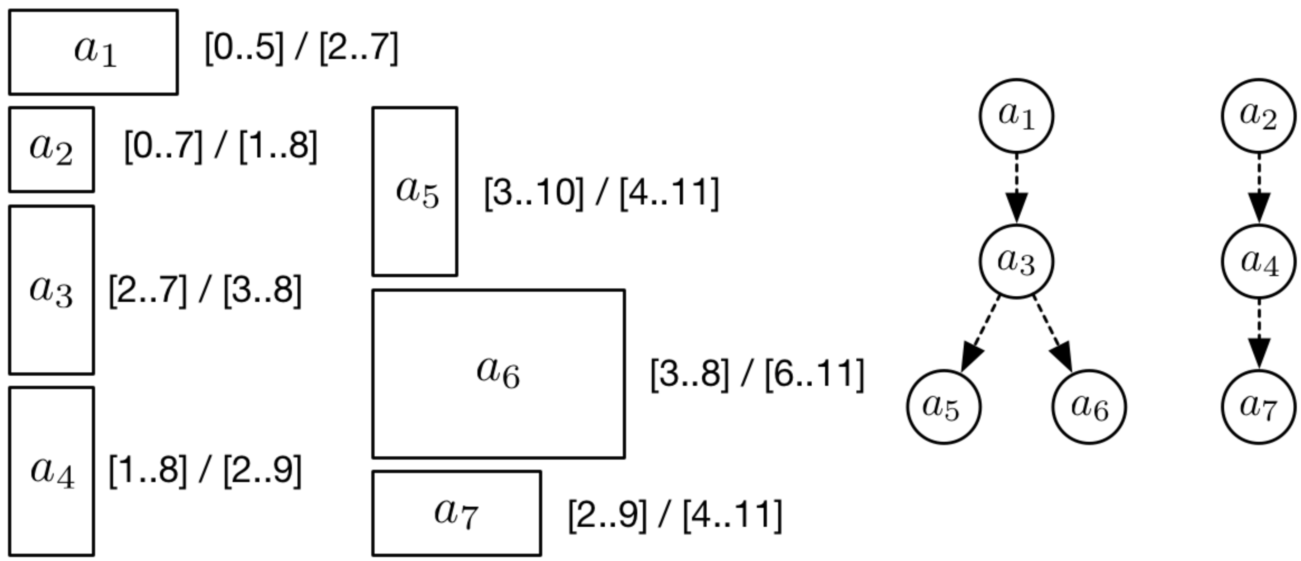




- The fake source/sink activities have been removed

Variable Selection for the RCPSP

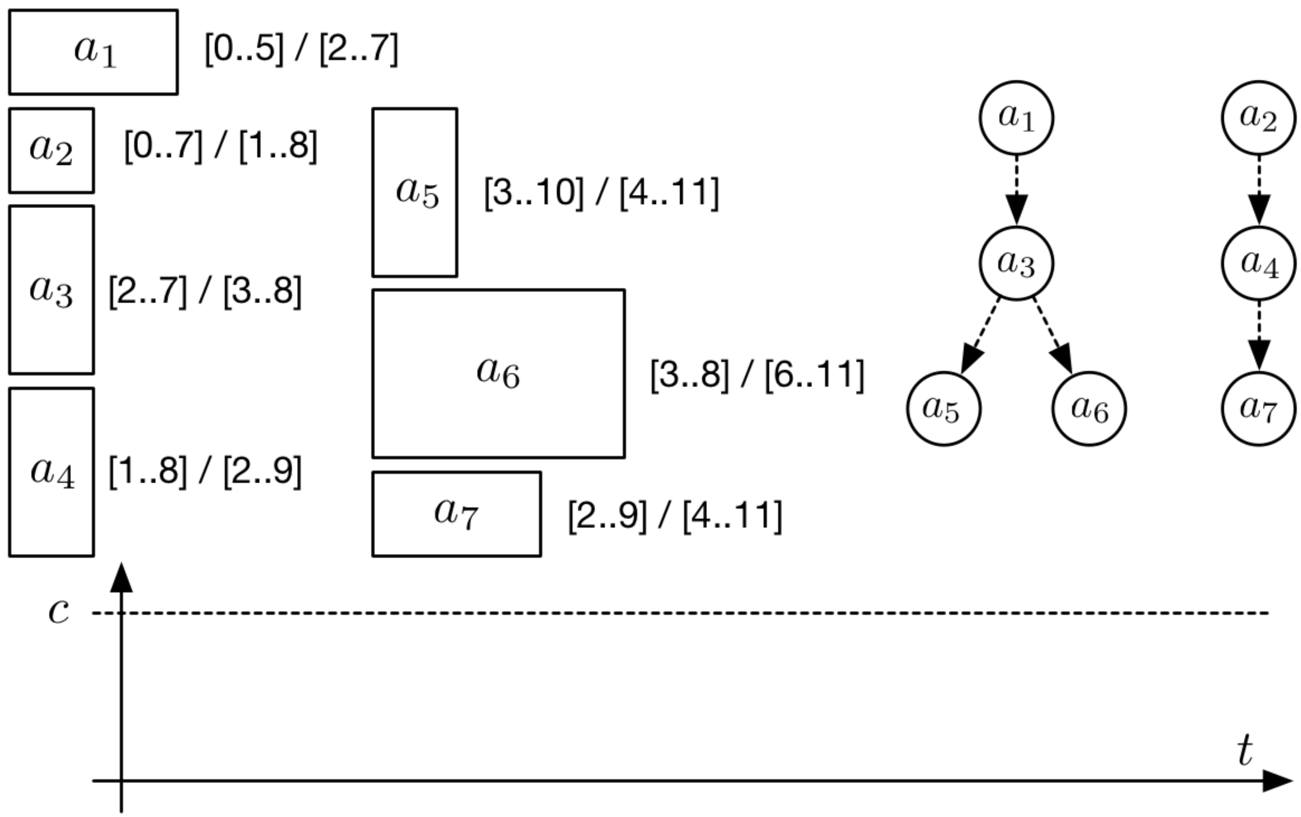
After propagating the precendence constraints, we get:



- Notation: $[EST_i..LST_i]/[EET_i..LET_i]$

Variable Selection for the RCPSP

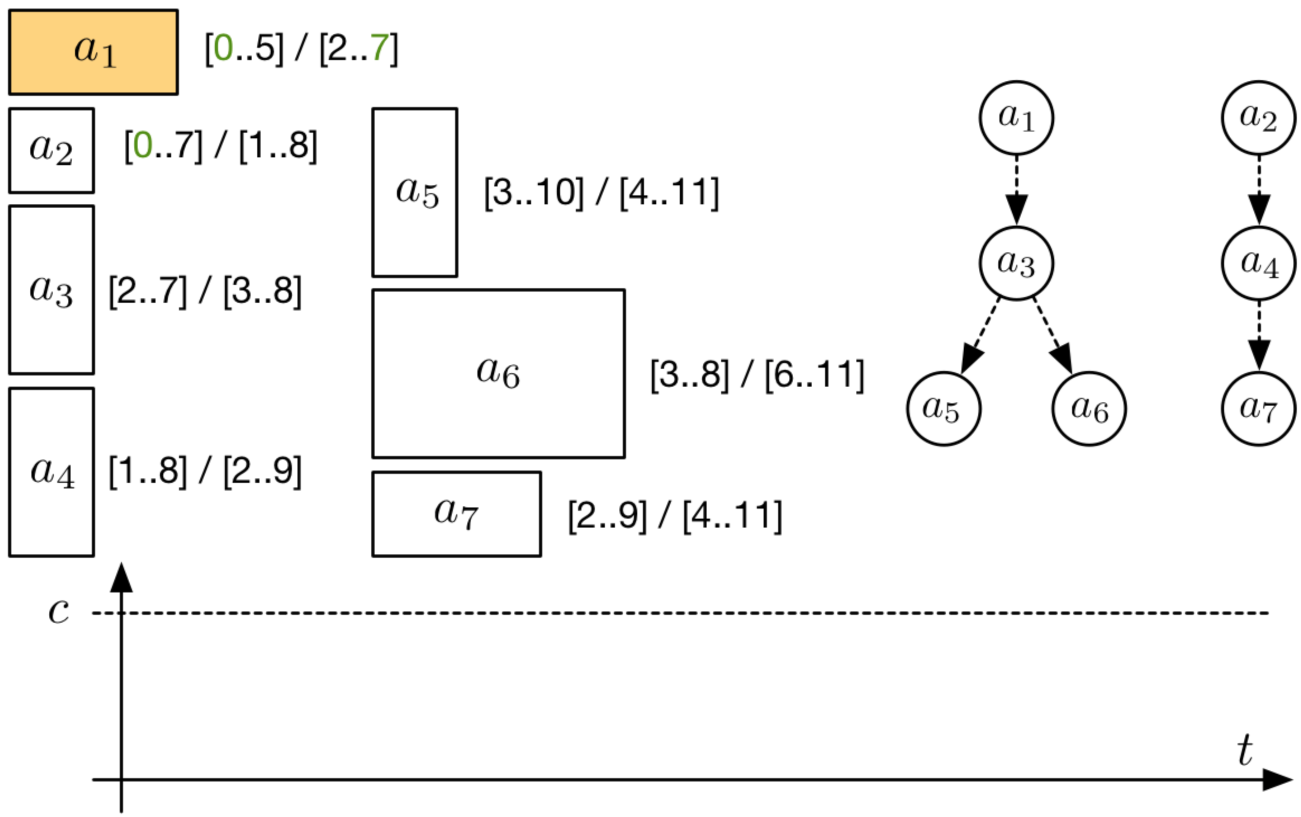
We now need to pick a variable for branching:



- A sensible criterion: minimum $\underline{s}_i$

Variable Selection for the RCPSP

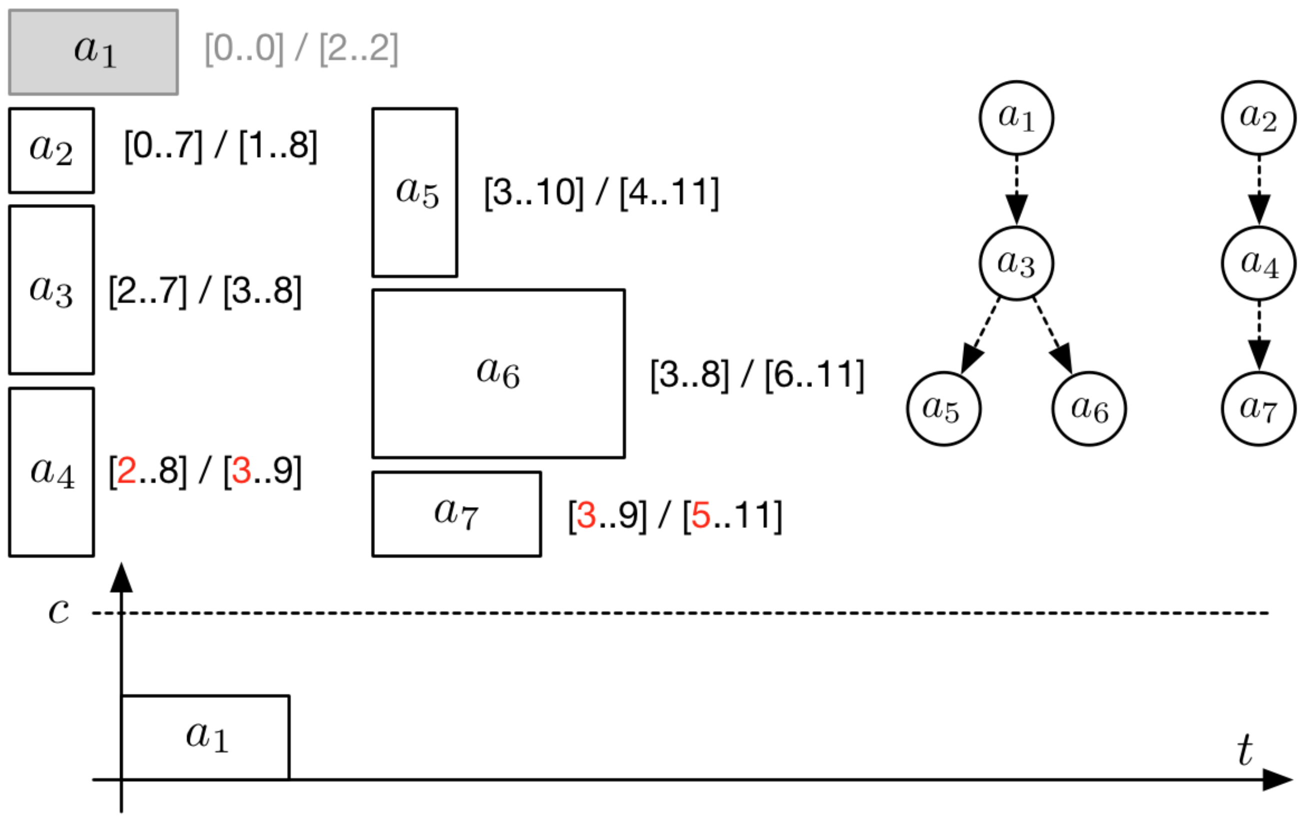
How to break ties:



- Smallest deadlines, i.e. minimum LET_i

Variable Selection for the RCPSP

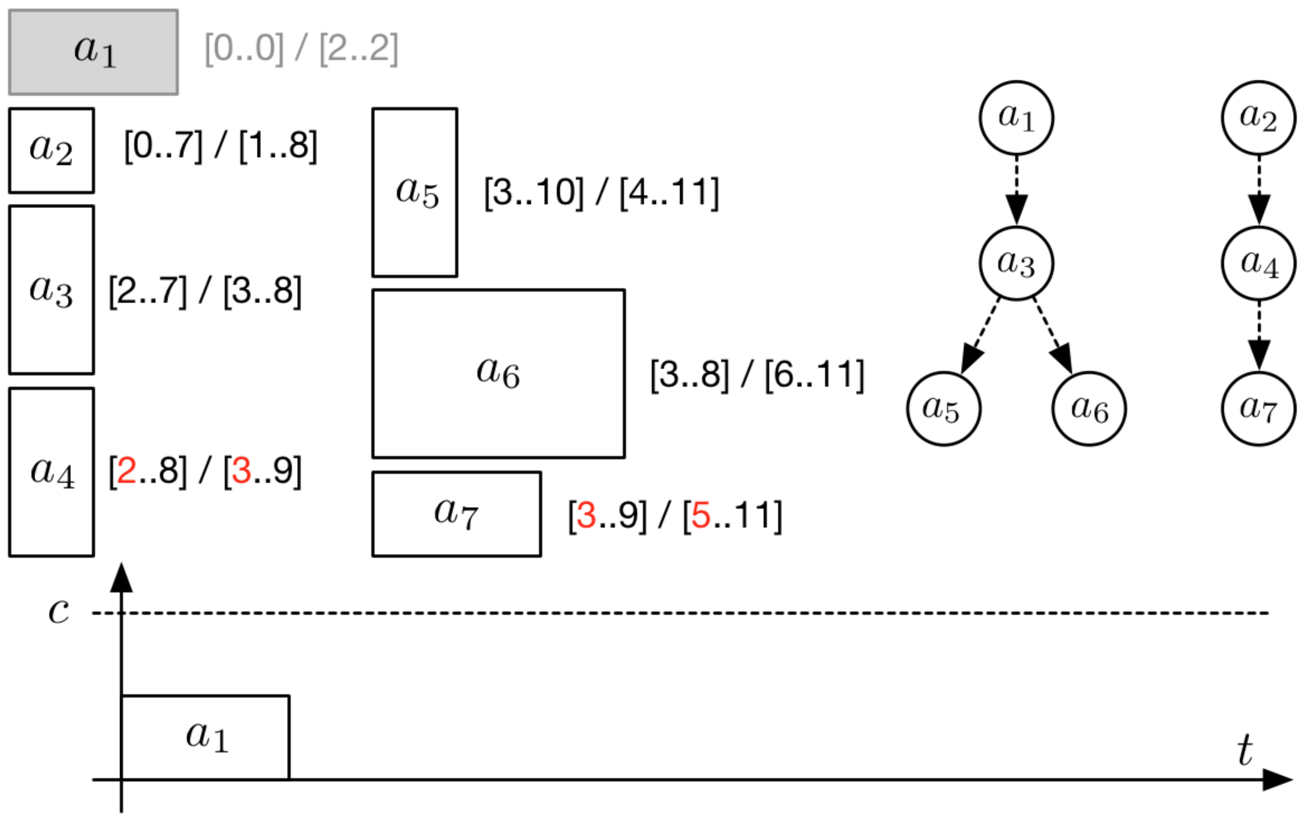
We now now schedule the selected activity at $\mathbf{EST_i}$:



- The whole duration of the activity becomes a compulsory part

Variable Selection for the RCPSP

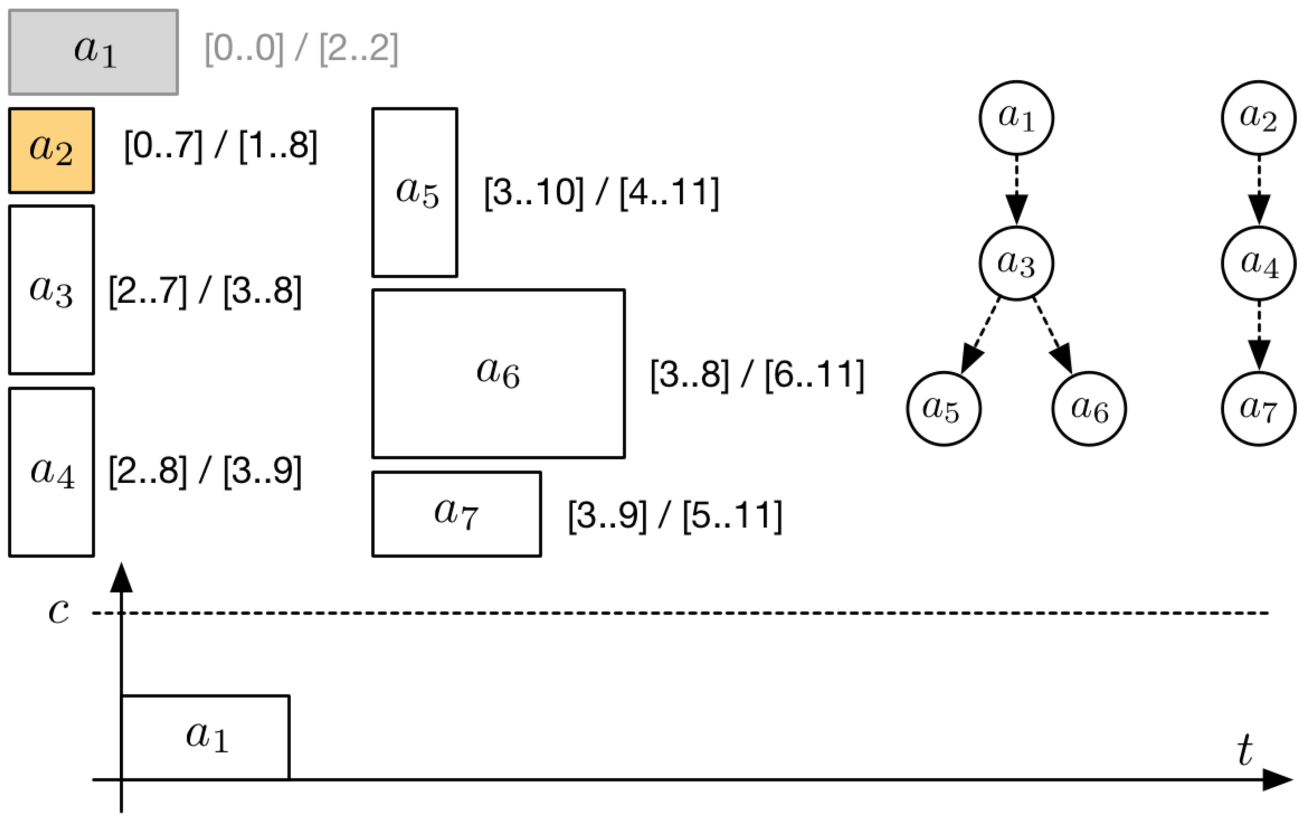
We now now schedule the selected activity at $\mathbf{EST_i}$:



- And propagation (precedences and CUMULATIVE) takes place

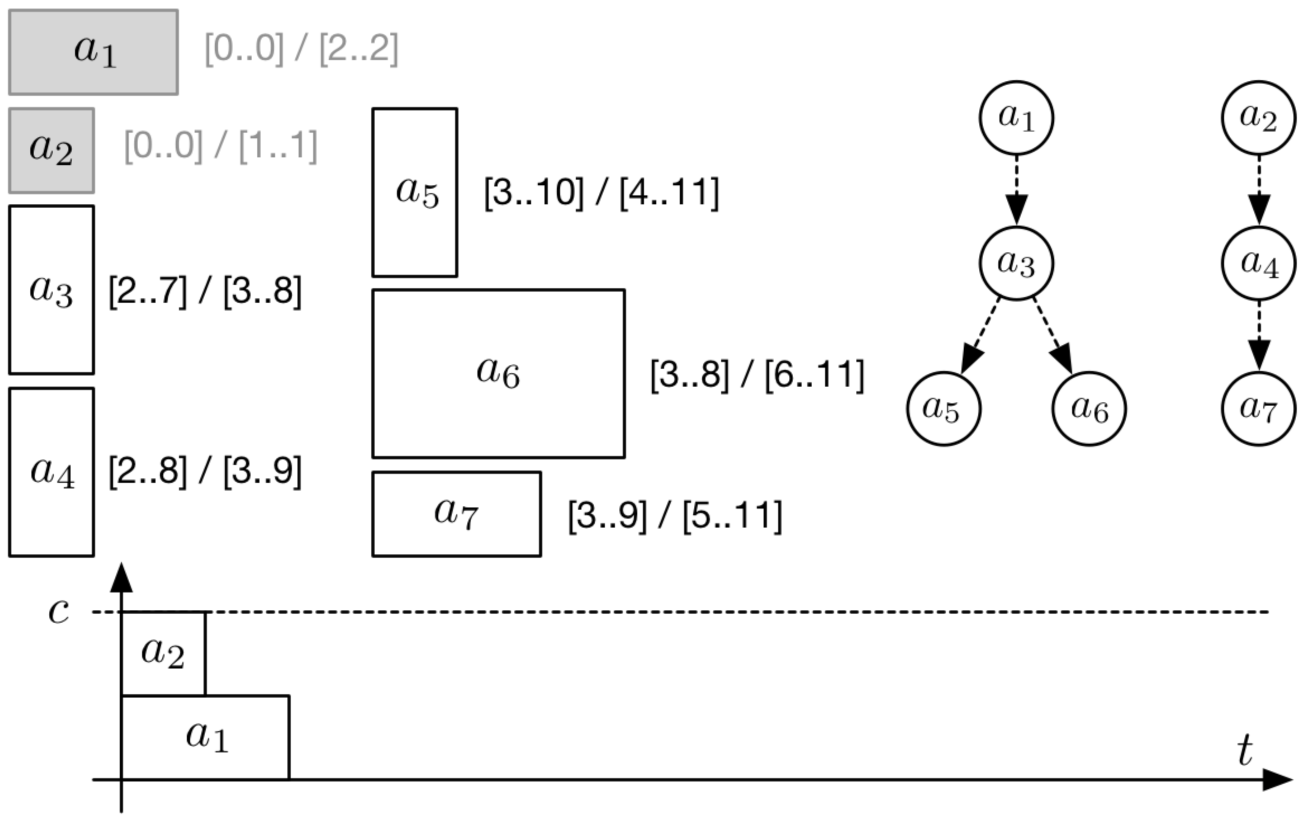
Variable Selection for the RCPSP

And then we repeat the process...



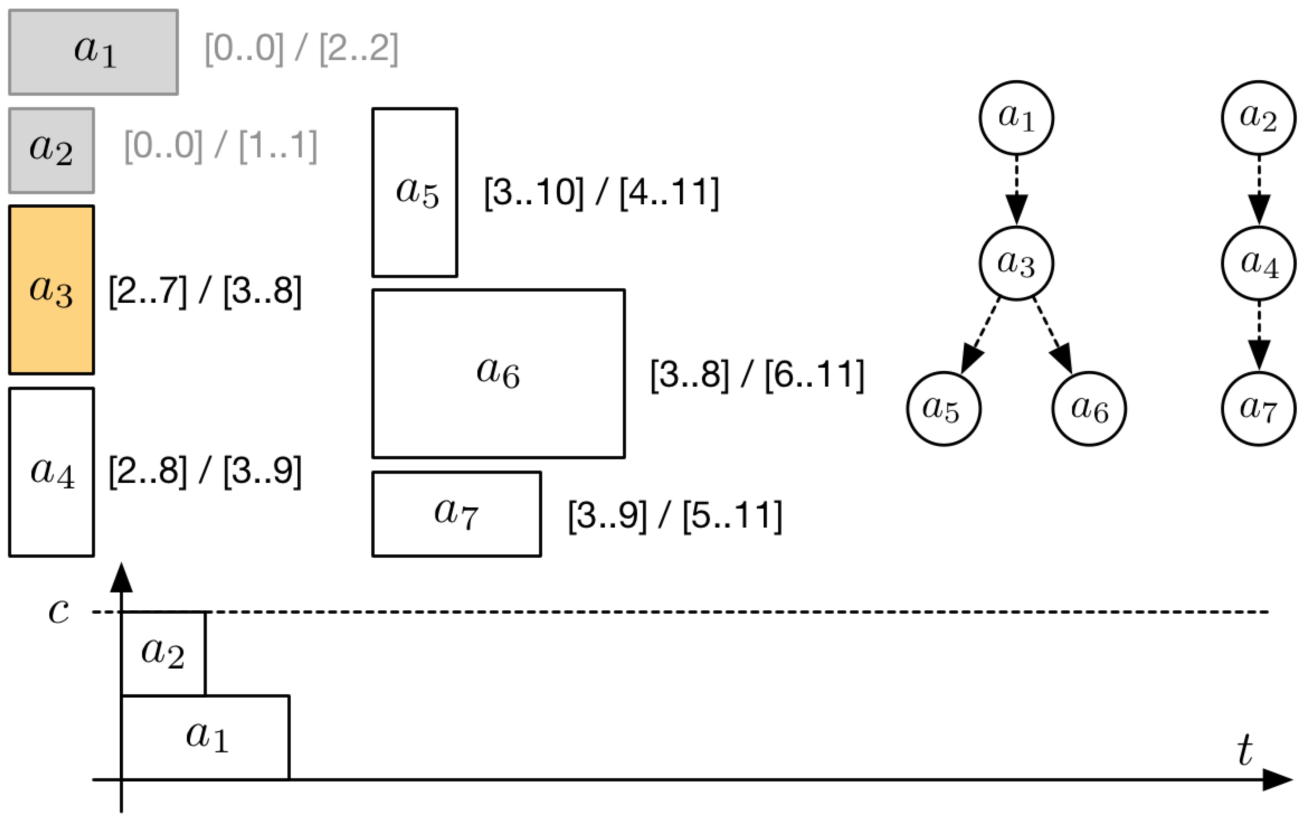
Variable Selection for the RCPSP

And then we repeat the process...



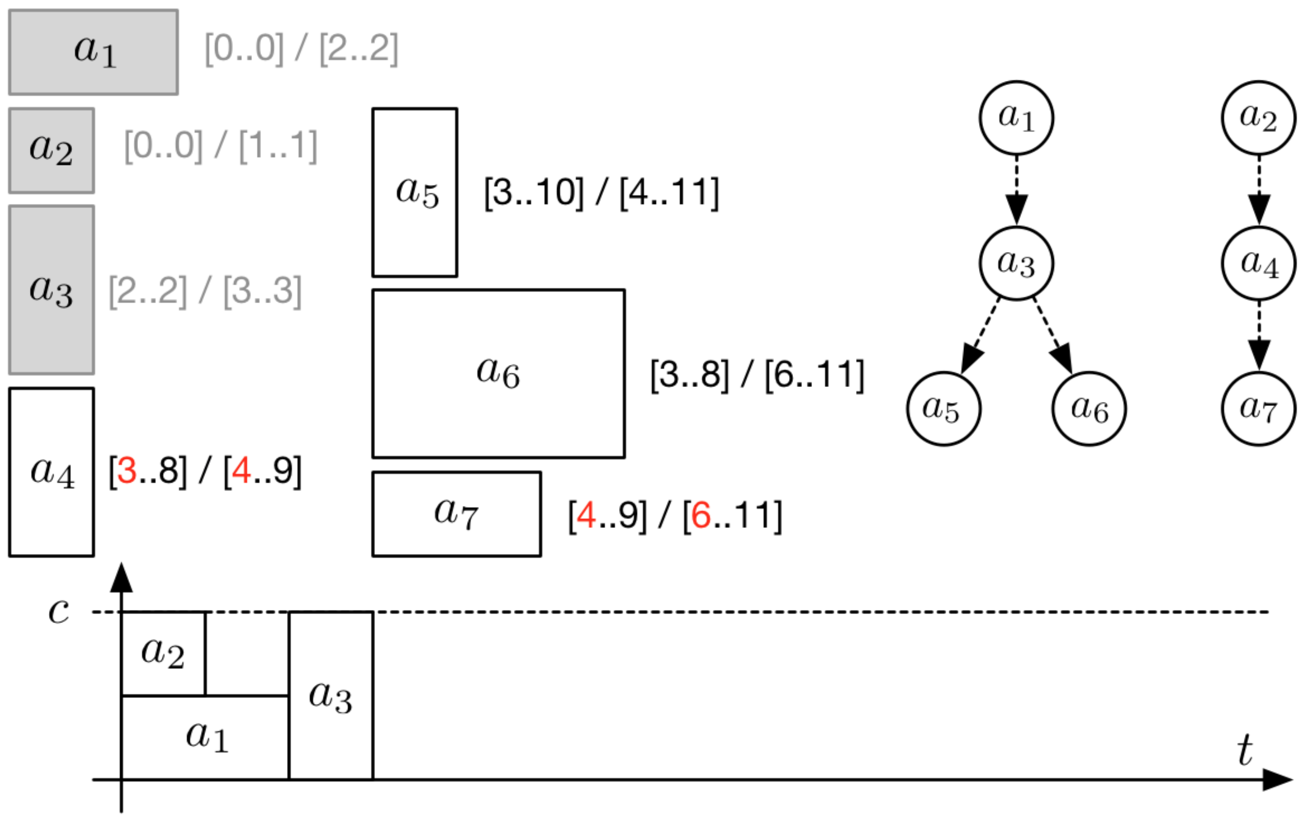
Variable Selection for the RCPSP

And then we repeat the process...



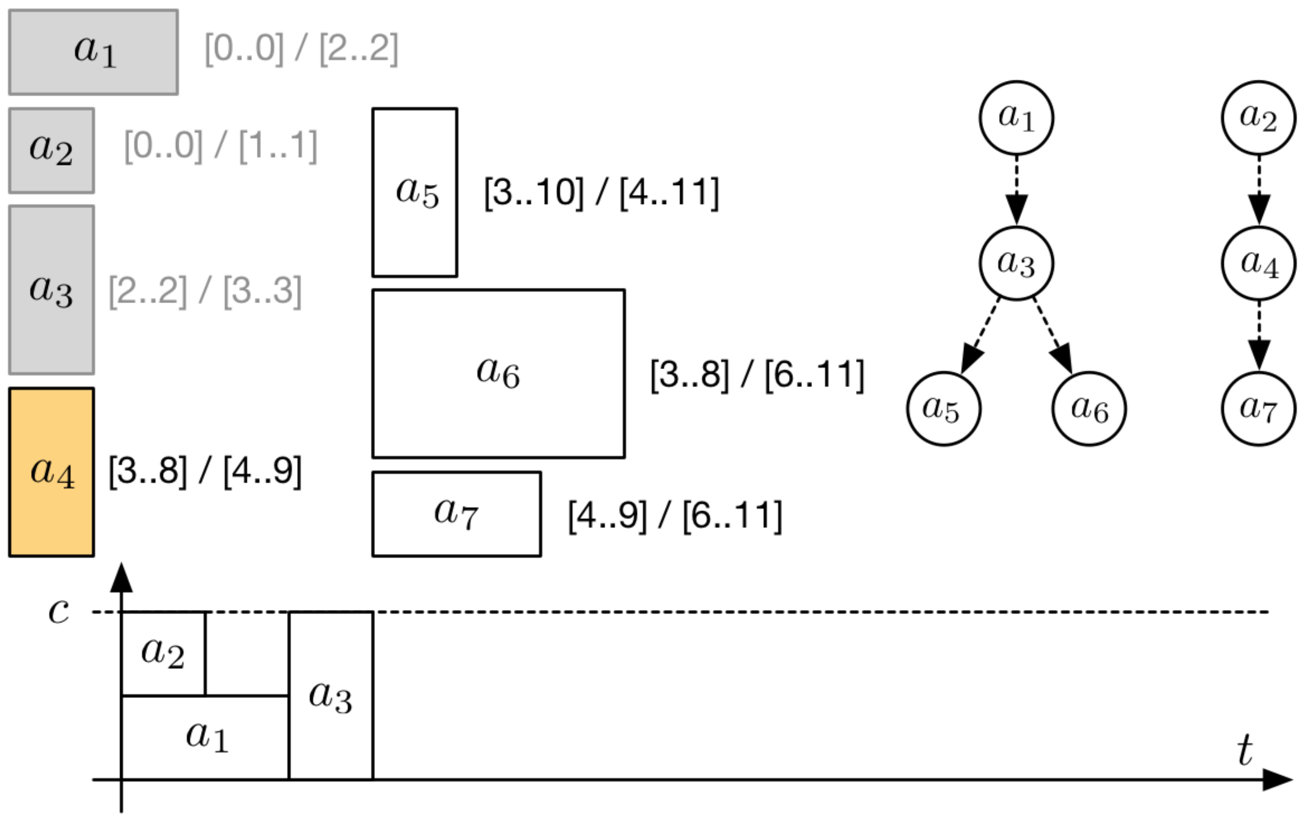
Variable Selection for the RCPSP

And then we repeat the process...



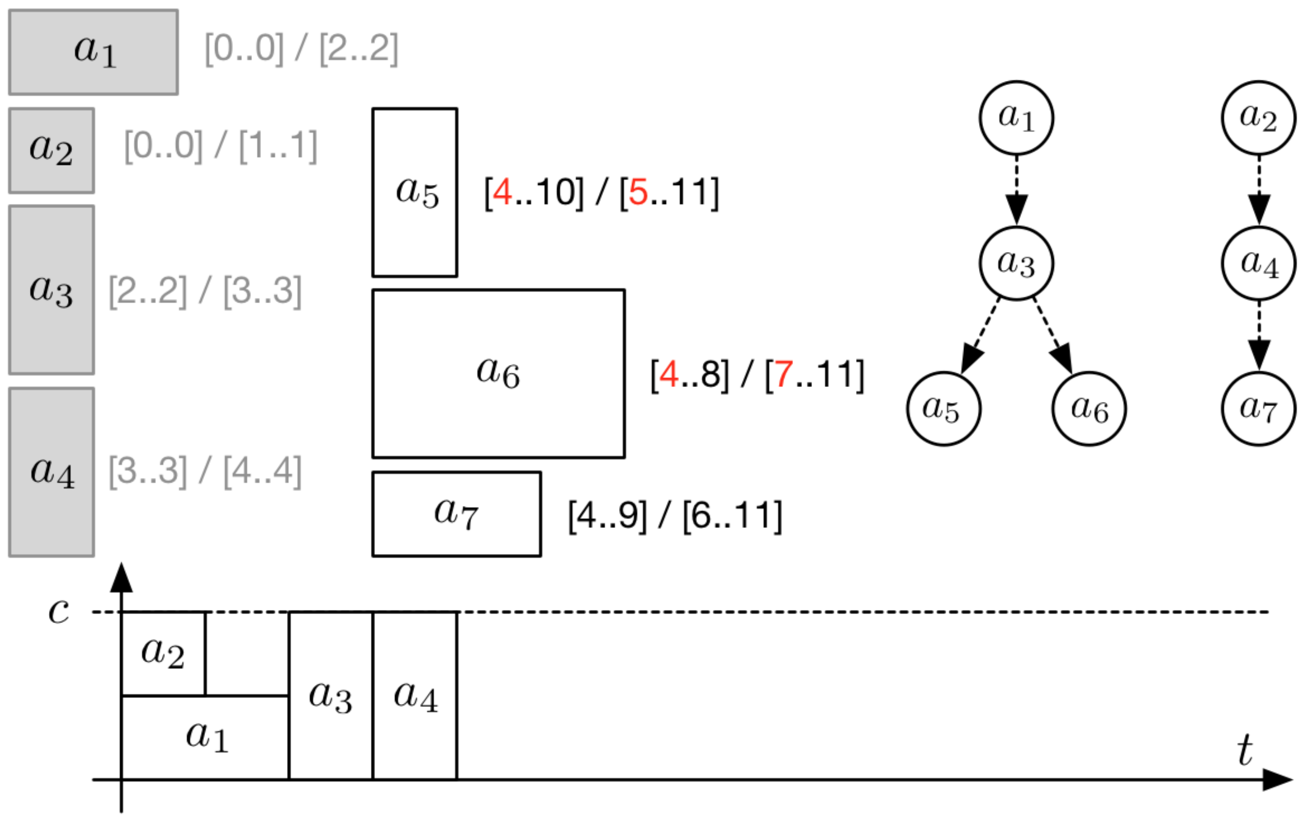
Variable Selection for the RCPSP

And then we repeat the process...



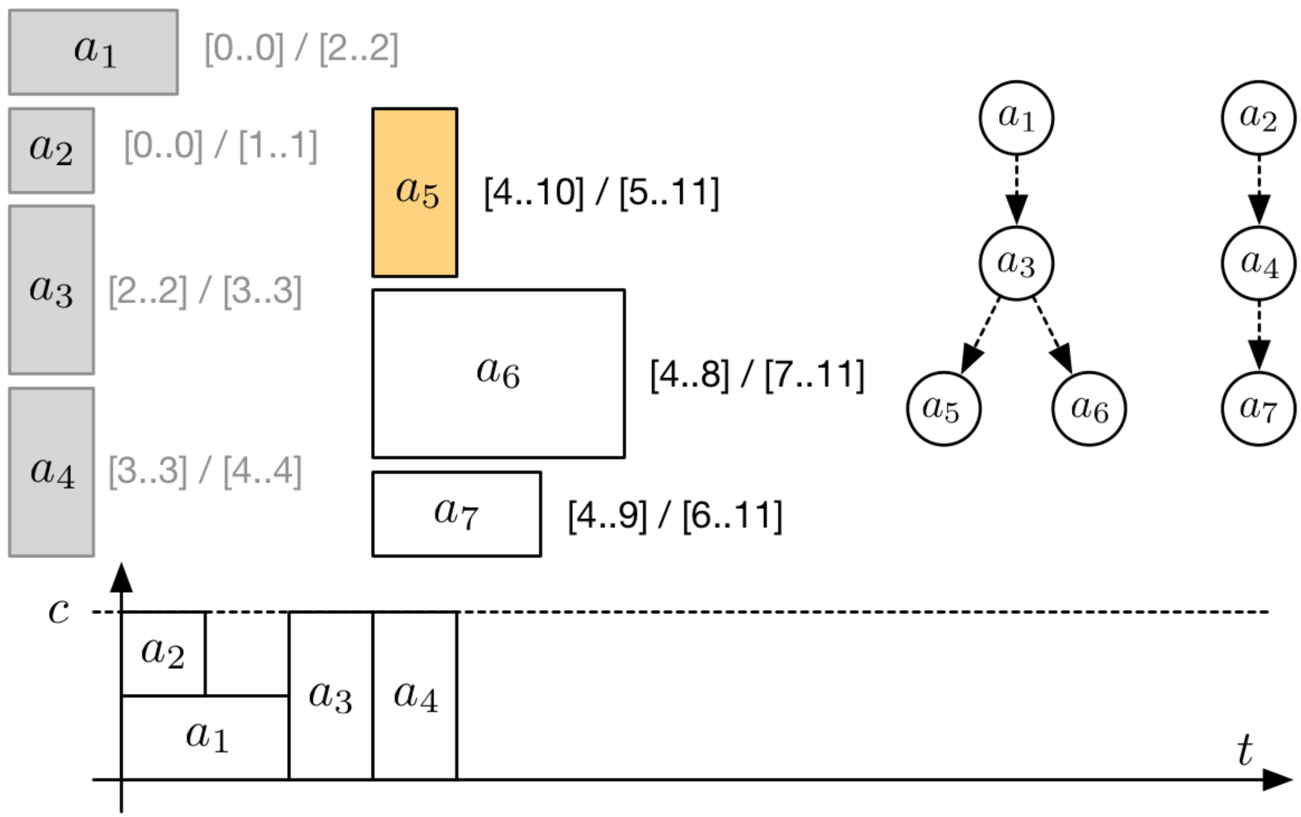
Variable Selection for the RCPSP

And then we repeat the process...



Variable Selection for the RCPSP

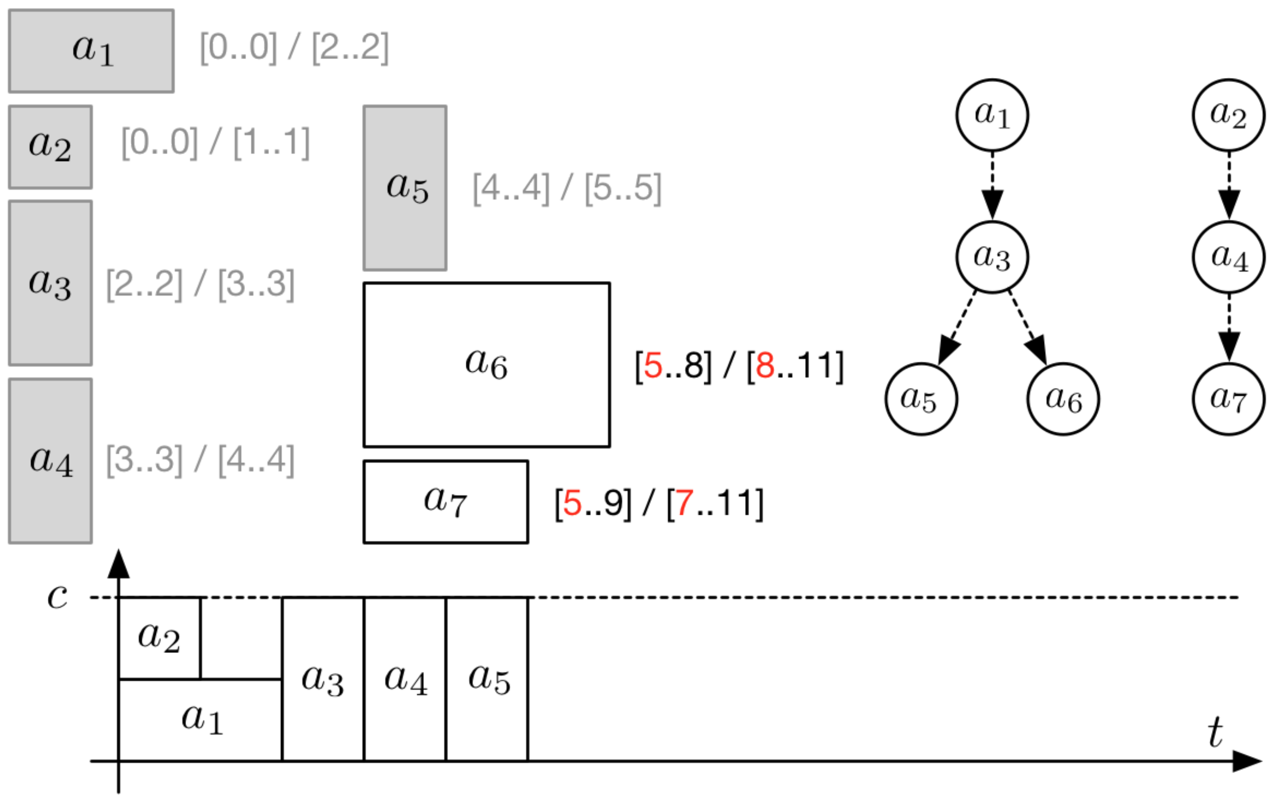
And then we repeat the process...



- In case of ties on LET_i, we look simply at the activity index

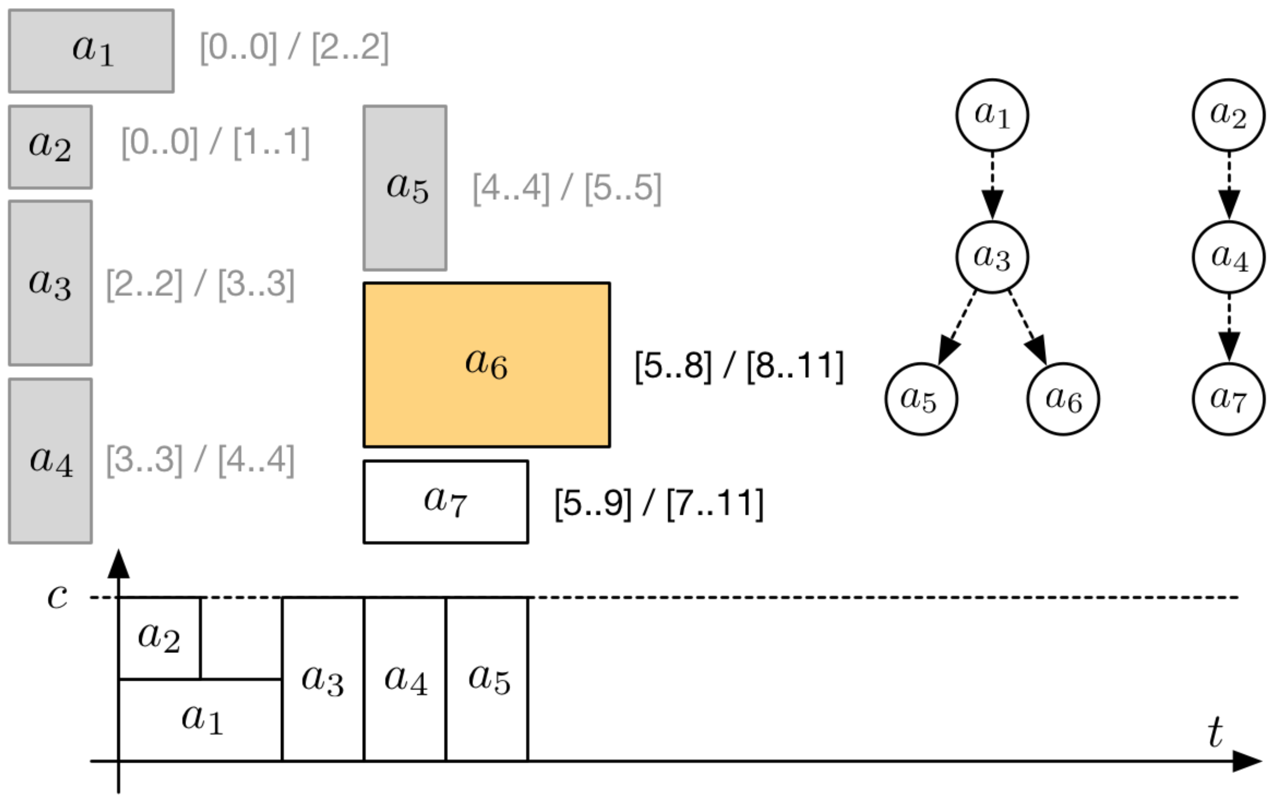
Variable Selection for the RCPSP

And then we repeat the process...



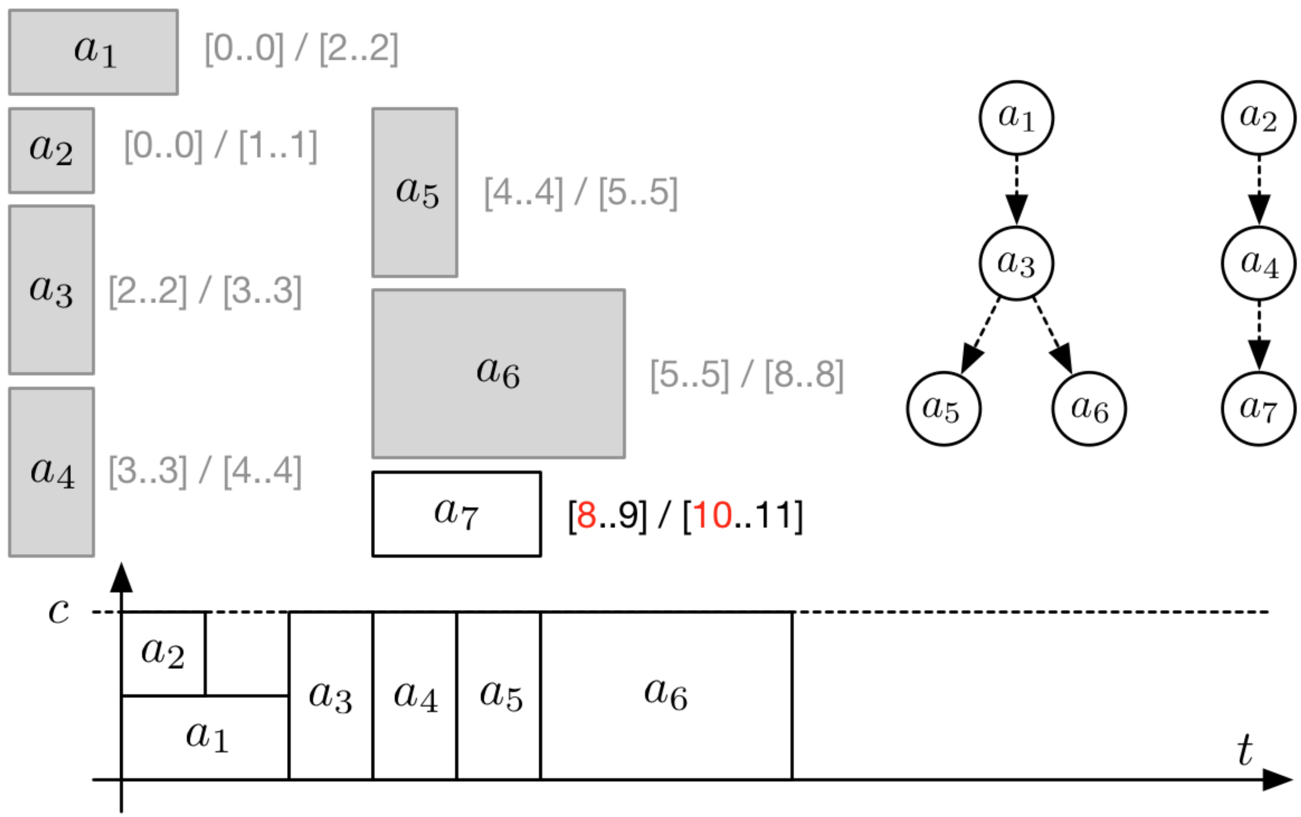
Variable Selection for the RCPSP

And then we repeat the process...



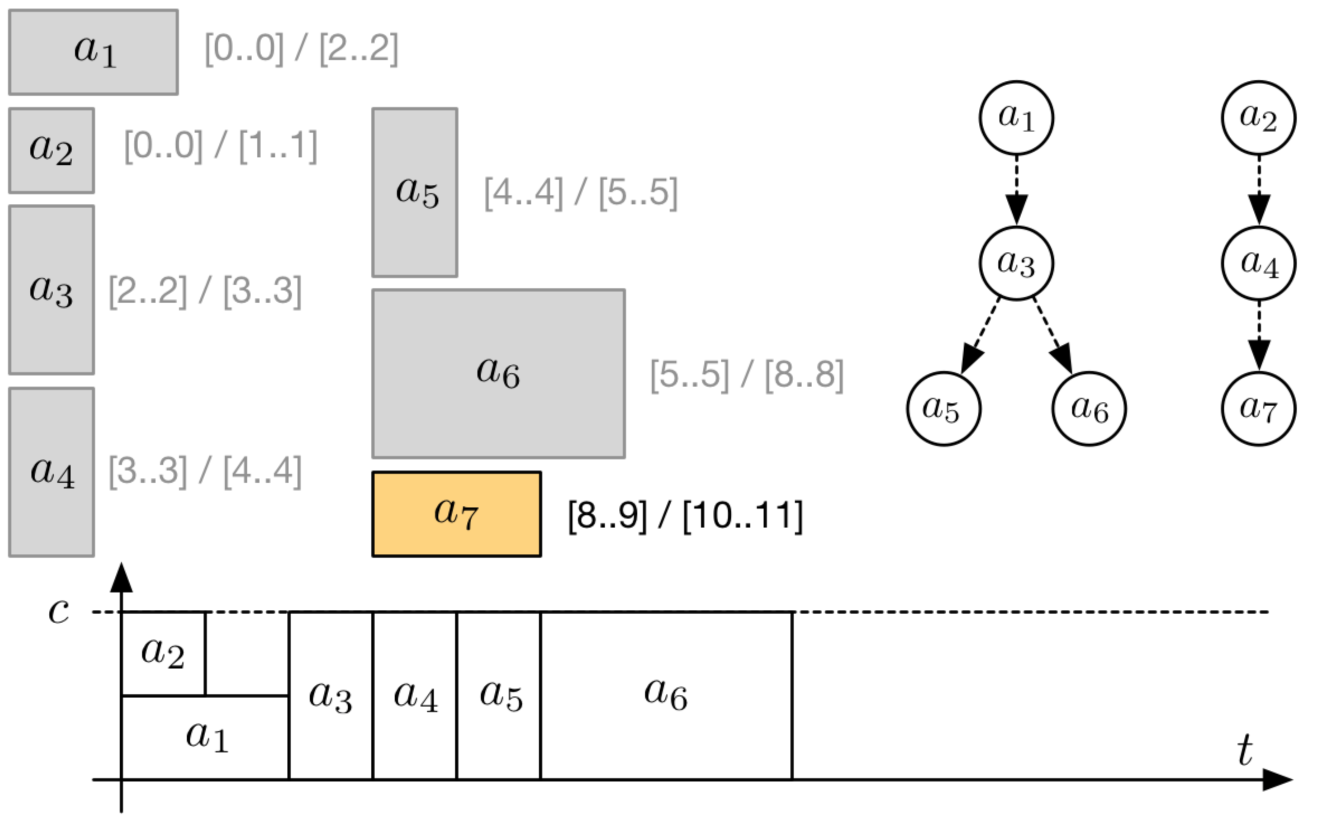
Variable Selection for the RCPSP

And then we repeat the process...



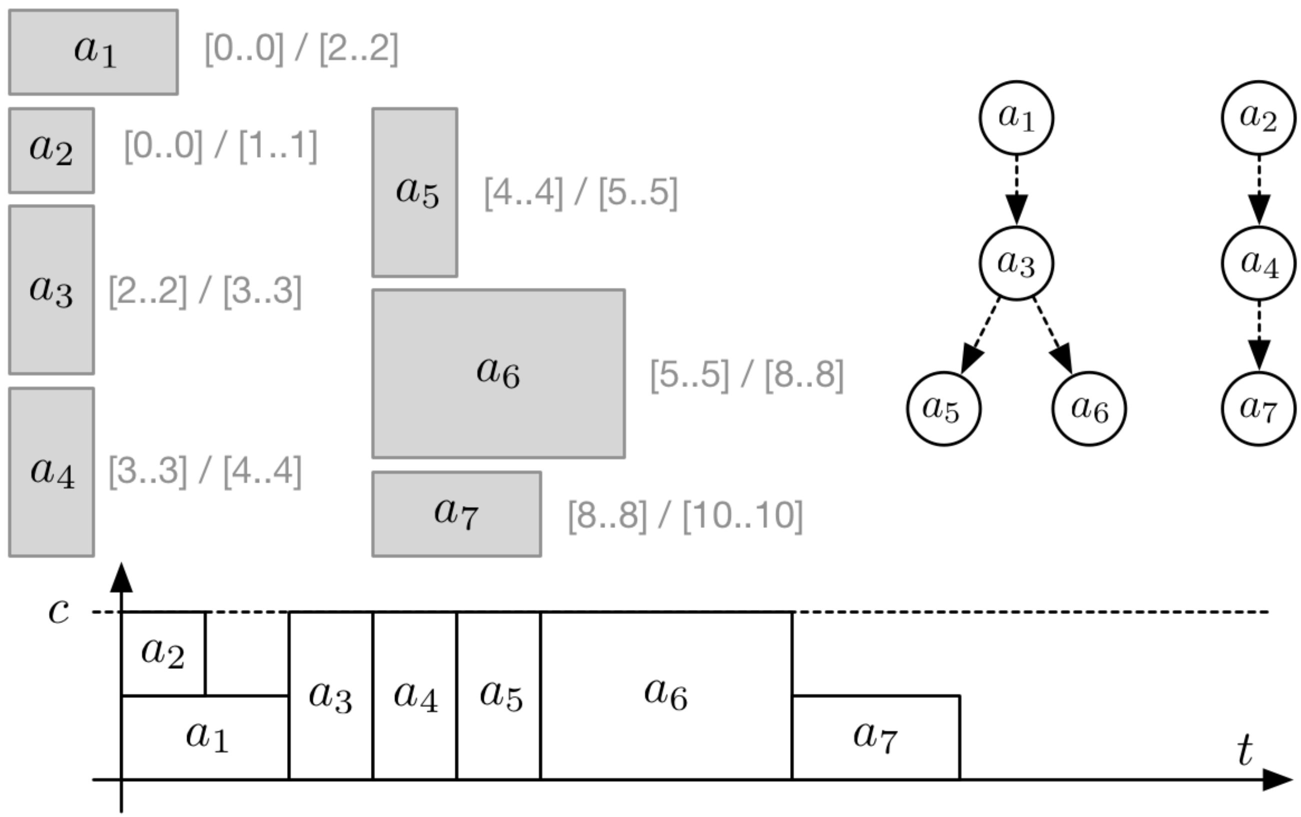
Variable Selection for the RCPSP

And then we repeat the process...



Variable Selection for the RCPSP

When all variables are assigned, we have a schedule



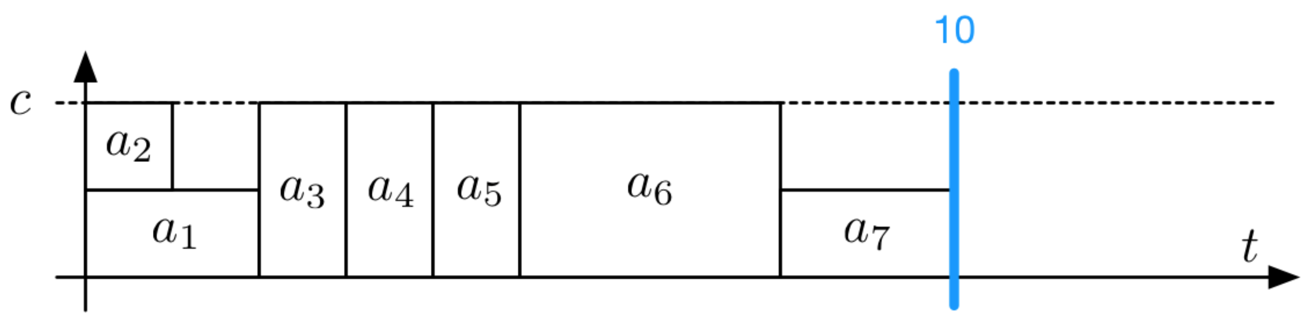
An Link with PRB Scheduling

This process for constructing an RCPSP solution has name!

Corresponds to a famous heuristic, greedy, solution approach

- A form of so-called Priority Rule Based Scheduling
- Works well in many cases

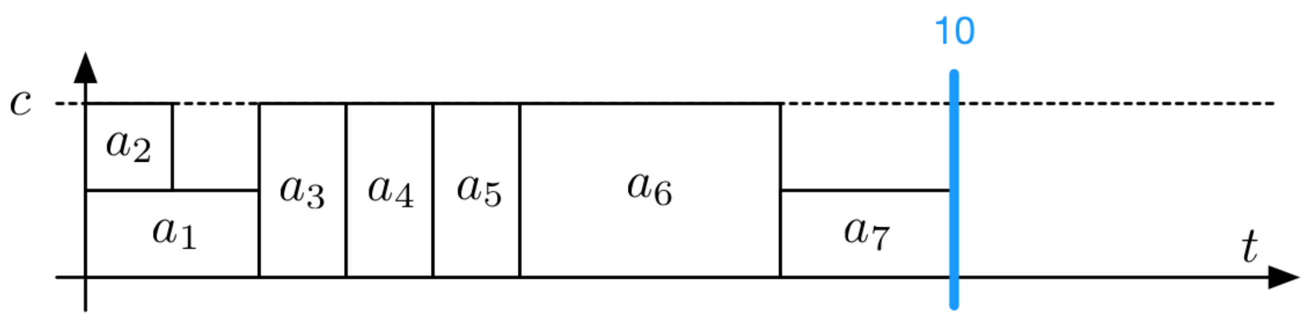
For our example the final makespan is 10



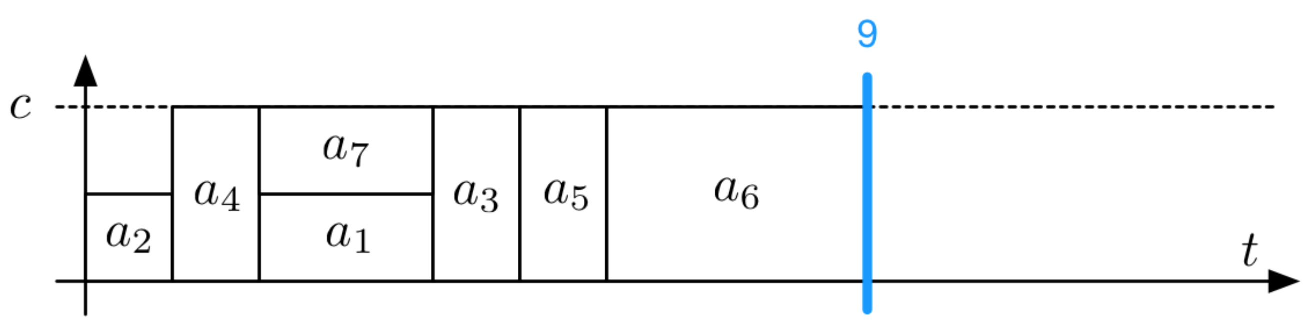
An Link with PRB Scheduling

However, PRB scheduling mat be sub-optimal:

- Here's our final schedule

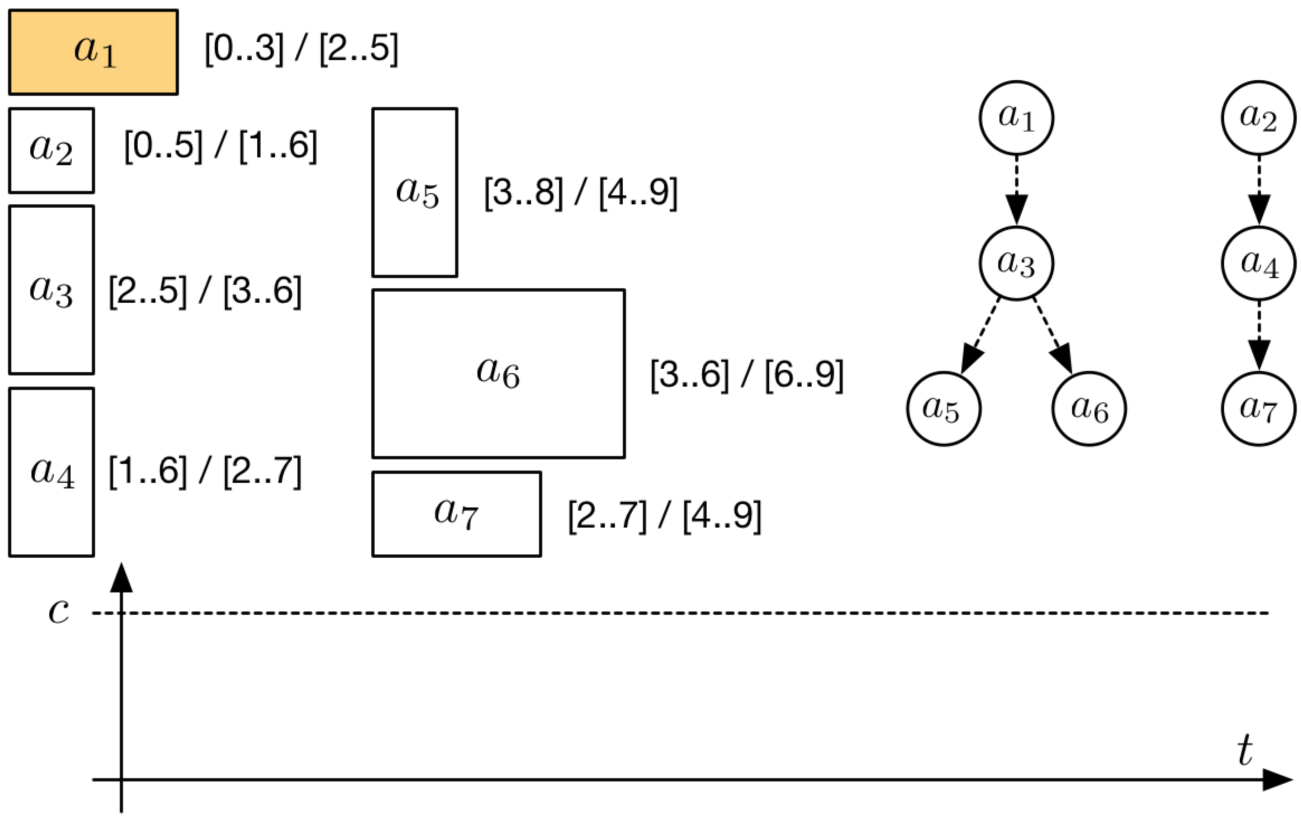


- Here's an optimal one:



Backtracking in CP Search for Scheduling

As usual, proving optimality requires to search and backtrack



- Let's go back to the root node

Backtracking in CP Search for Scheduling

As usual, proving optimality requires to search and backtrack

We could post $s_1 \neq 0$, which would ensure complete search

- But it is weak, since s_i domains tend to be very large

A strange alternative: we mark activity i as postponed

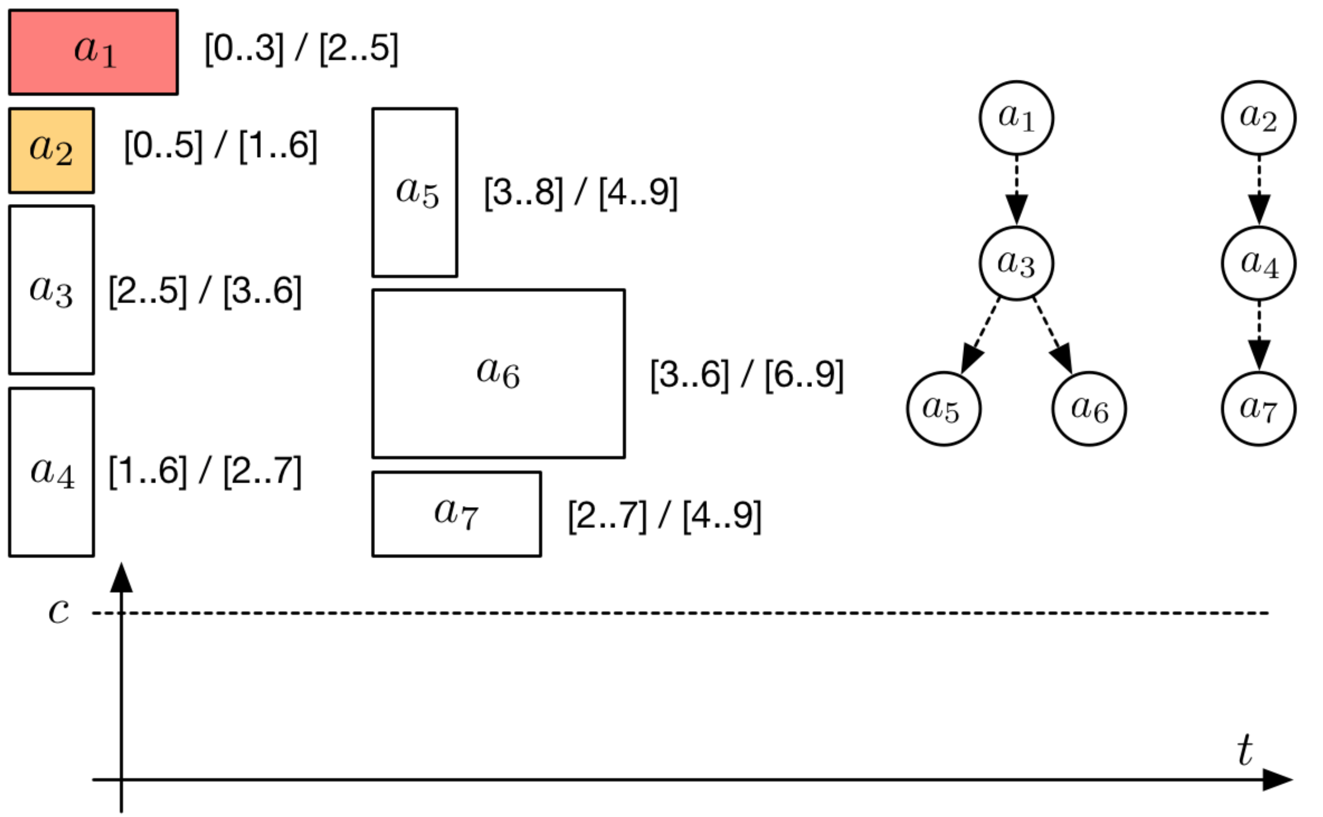
- A postponed activity cannot be selected for branching…
- …Until its $\underline{s}_i$ value changes

Rationale: we want to explore a different branching decision

- We always schedule activities at their $\underline{s}_i$
- Hence, the scheduling decision changes when $\underline{s}_i$

Backtracking in CP Search for Scheduling

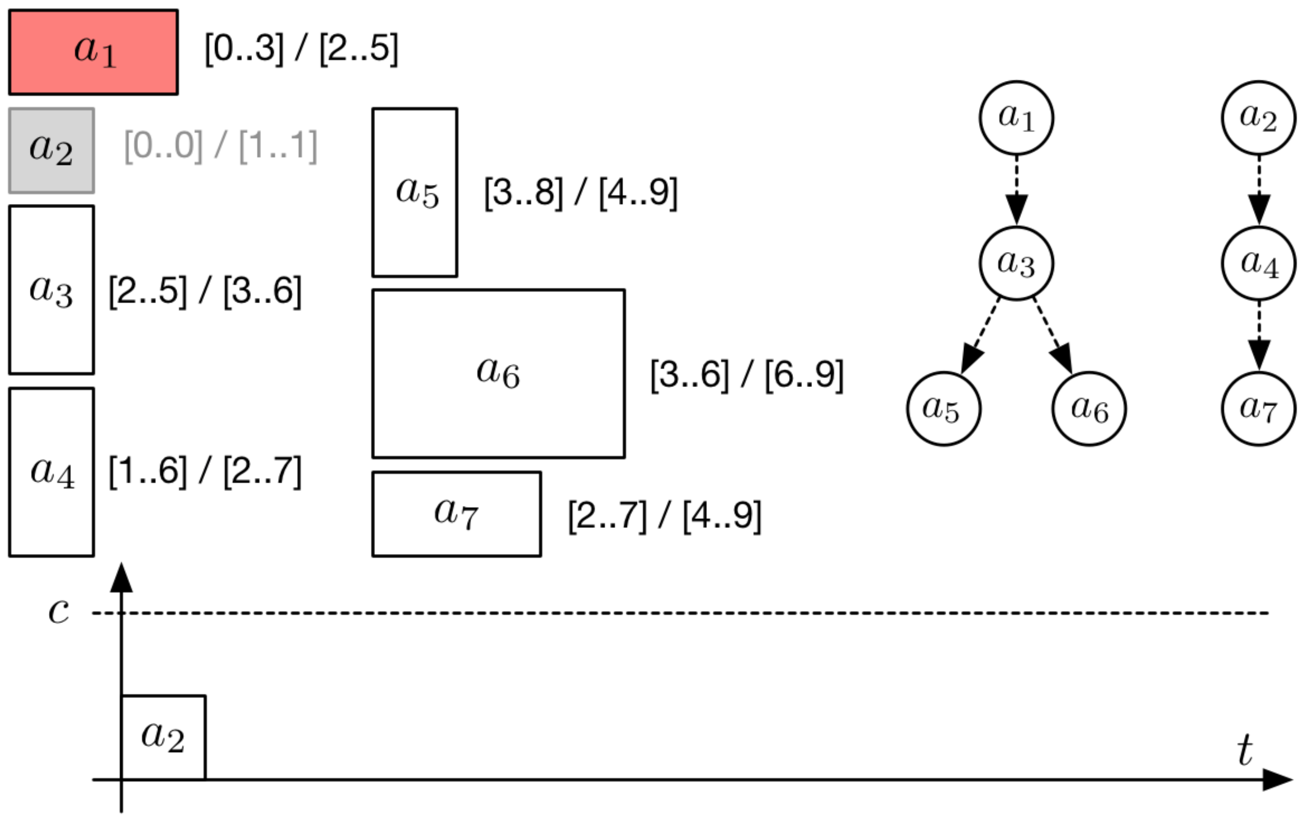
As usual, proving optimality requires to search and backtrack



- So s_1 is postponed, forcing us to pick s_2 for scheduling

Backtracking in CP Search for Scheduling

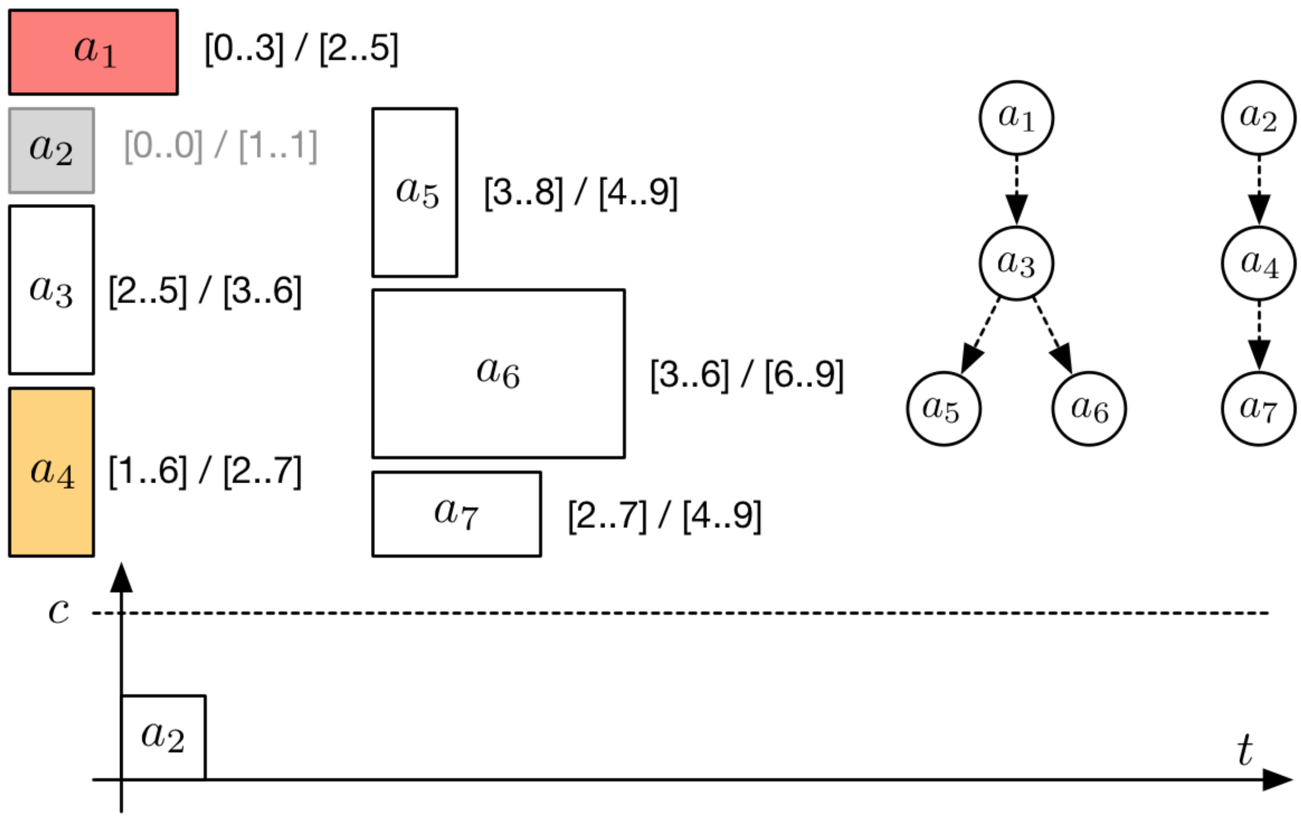
As usual, proving optimality requires to search and backtrack



- We proceed as usual…

Backtracking in CP Search for Scheduling

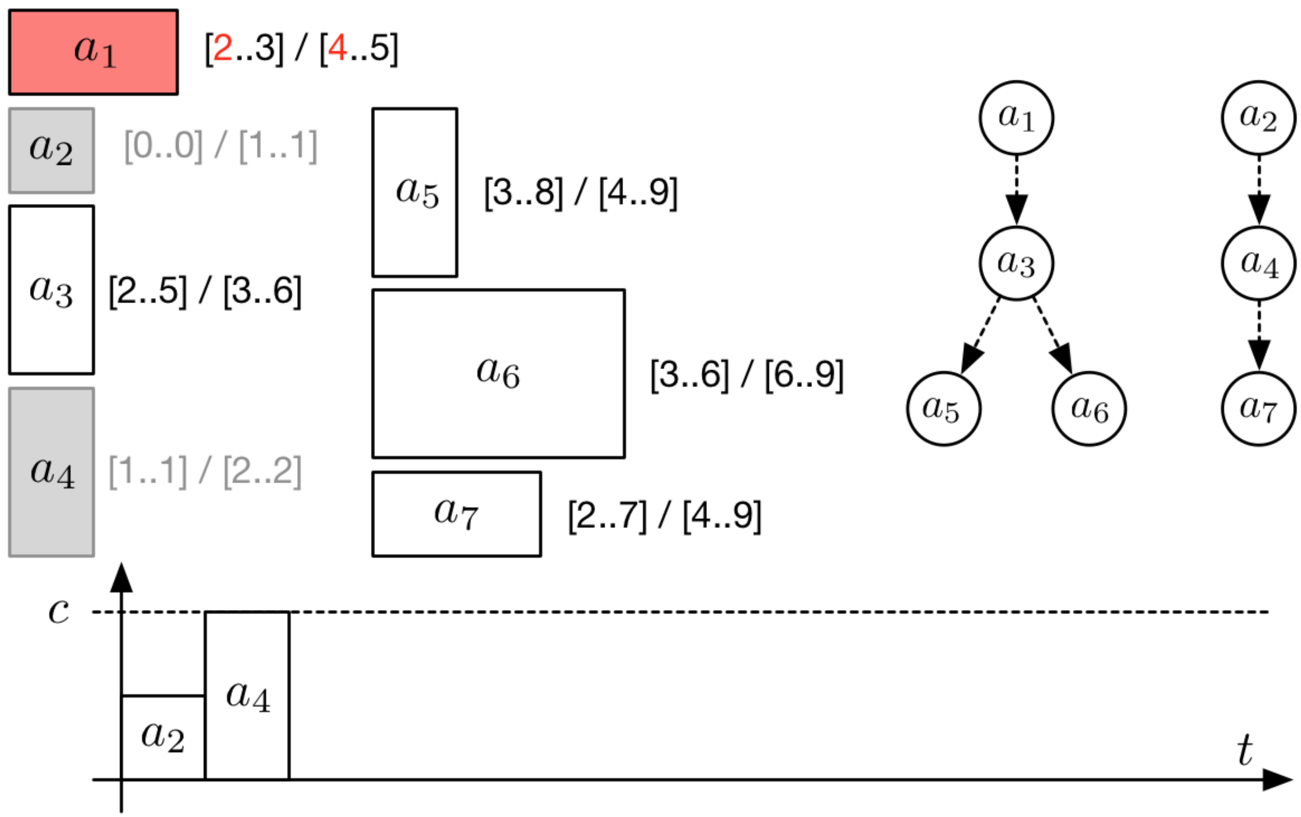
As usual, proving optimality requires to search and backtrack



- We proceed as usual...

Backtracking in CP Search for Scheduling

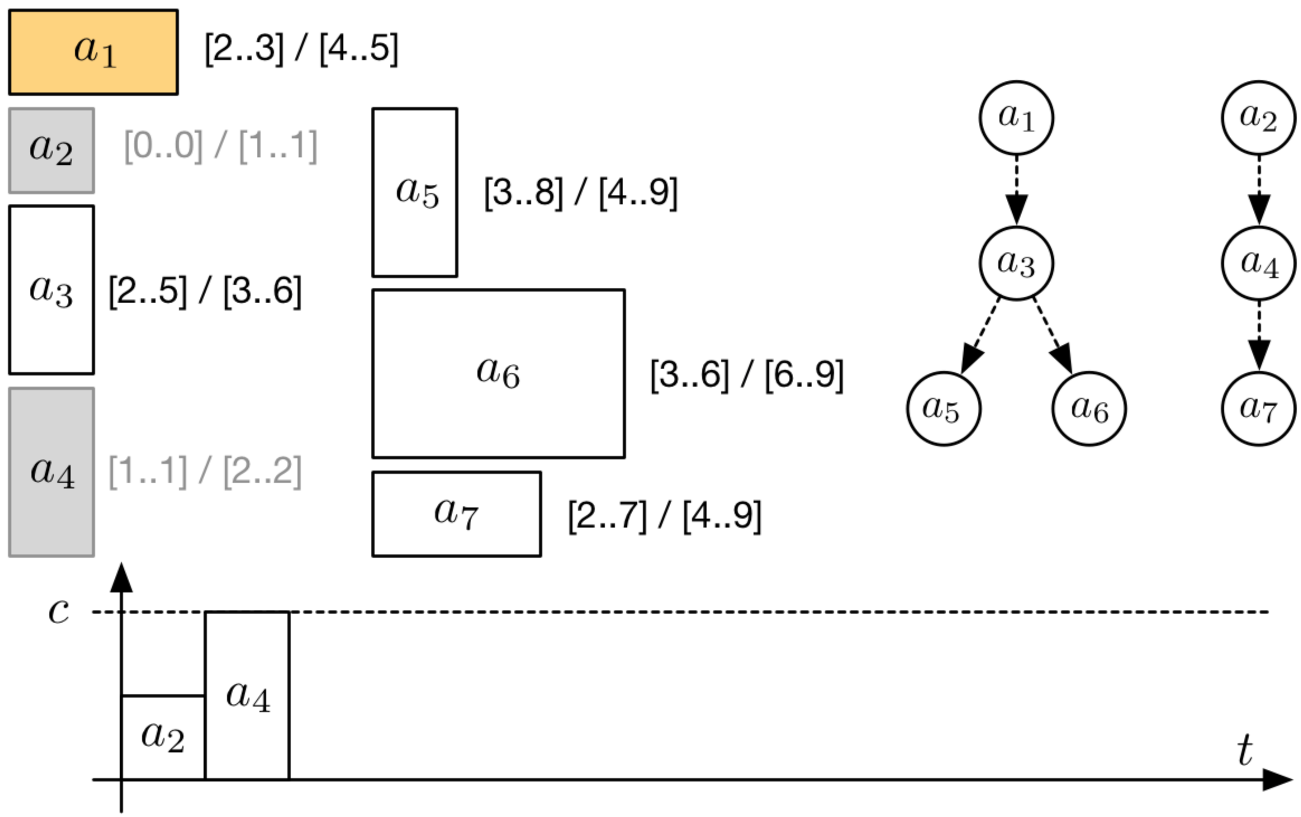
As usual, proving optimality requires to search and backtrack



- Until the value of $\underline{s}_1$ is updated by propagation

Backtracking in CP Search for Scheduling

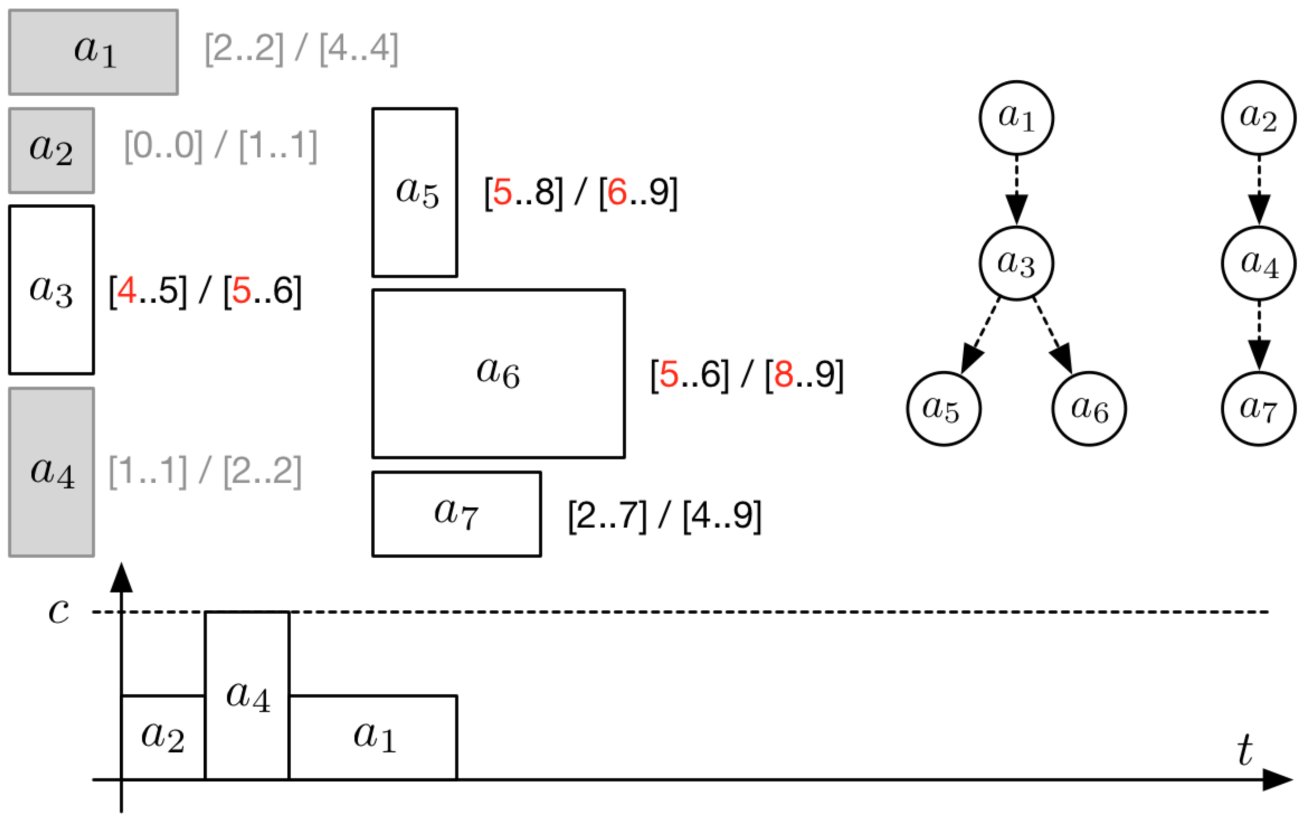
As usual, proving optimality requires to search and backtrack



- At this point, s_1 becomes eligible for branching

Backtracking in CP Search for Scheduling

As usual, proving optimality requires to search and backtrack



- By proceeding along this branch, we will find the optimal schedule

SetTimes Search Strategy

This scheduling strategy is often called SetTimes

Main ideas (a summary):

- Schedule-or-postpone decisions
- Always pick an activity with minimum $\underline{s}_i$
- Schedule activities at their $\underline{s}_i$

SetTimes is typically a very effective strategy

- Based on PRB scheduling: finds good solutions early
- Effective branching choices (much better than posting $s_i \neq v$)
- The strategy implicitly makes ordering decisions

SetTimes Search Strategy

Some caveats

Technically, SetTimes is an incomplete search strategy

- At choice points, we do not partition the search space
- Either we schedule an activity at $\underline{s}_i$, or we make it wait

Why does it work? SetTimes is based on a dominance rule

- The cost function is regular
- Hence, there is no point in not scheduling activities at their $\underline{s}_i$…
- …Unless they are delayed by previous activities

SetTimes Search Strategy

When is SetTimes not working?

- Non-regular cost functions
  - E.g. costs for starting activities too early
- Side-constraints that alter the structure of the problem
  - Many possible cases!

A consequence:

Other search strategies are becoming mode popular

- In particular, domain splitting
- Remember FDS? The experimentation was doing domain splitting

Constraint Systems

Constraint Based Scheduling: An Example Application

An Example Application

Target Problem: run parallel code on a HW accelerator

- The code is split into a tasks
- Some tasks communicate data to others
- The accelerator contains many cores, grouped in clusters
- Intra-cluster communications are fast
- Inter-cluster communications are slow
- Inter-cluster communications use a local comm. port
- No-preemption: each core runs a single process at any time
- Task durations are approximately known

Objective: complete the execution as fast as possible

An Example Application

Modeling (just an intuition):

HP: tasks have been pre-assigned to clusters

- Each task = an activity
- Approximate duration = activity durations
- Quick communications = precedence constraints
- Slow communications = precedences + extra activities
- Clusters = cumulative resources
  - Capacity = number of cores per cluster
- Communication ports = other resources

In other words: **we model the problem as an RCPSP**

Dealing With Flexible Durations

One tricky point: durations are approximately known

- The actual durations become known only at run time
- Consequence: fixed start times are not a good idea!

A first solution: use an on-line heuristic instead of CP

E.g. a queue-based scheduler:

- First-In Firs-Out scheduler
- Fixed Priority scheduler (possibly optimized priorities)

This is what most people do in the embedded system community

But there is an alternative...

Partial Order Schedules

We can turn our fixed-start schedule into a flexible schedule

A possible approach:

- Instead of fixing the start times
- We prevent resource conflicts by adding new precedences
- Result: an augmented project graph

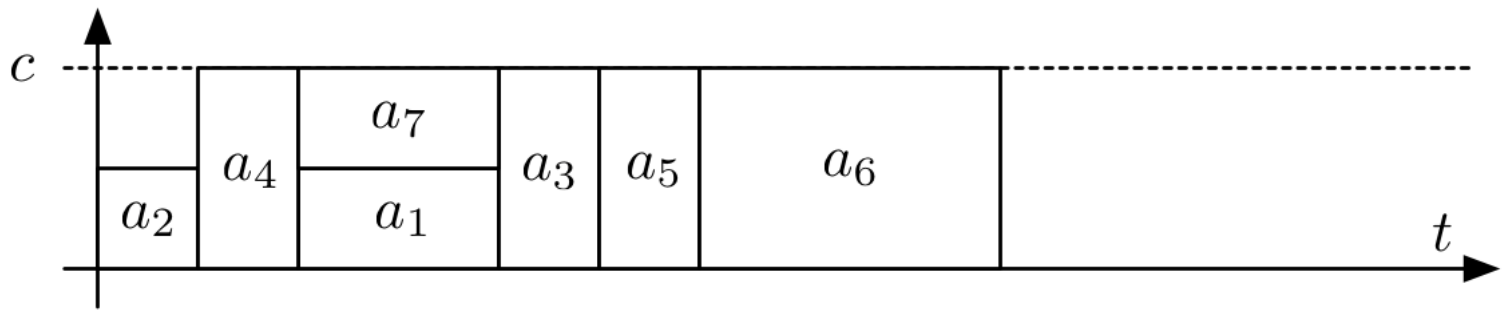
This graph is called a Partial Order Schedule

Fundamental property:

- As long as all precedences are respected...
  - Both original and added
- ...No resource constraint is violated

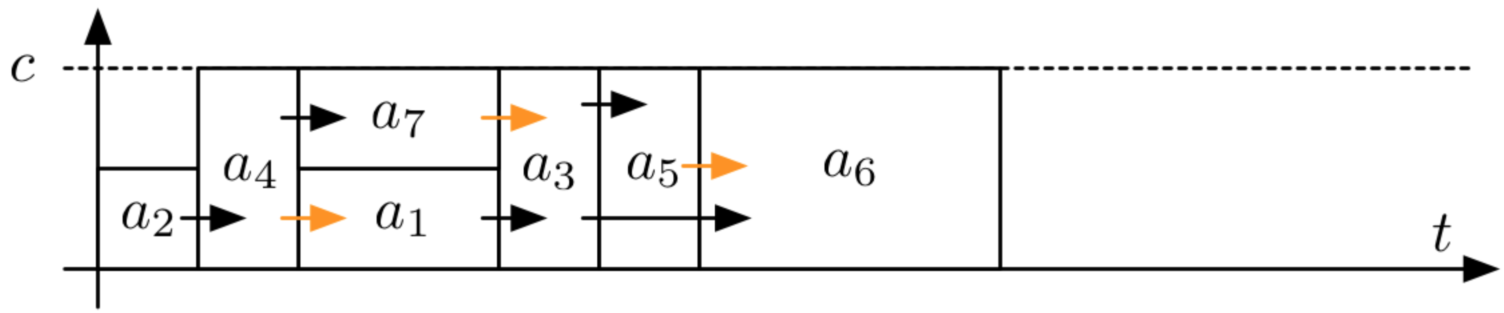
Partial Order Schedules

Here's our optimal schedule



Partial Order Schedules

And here's a possible POS



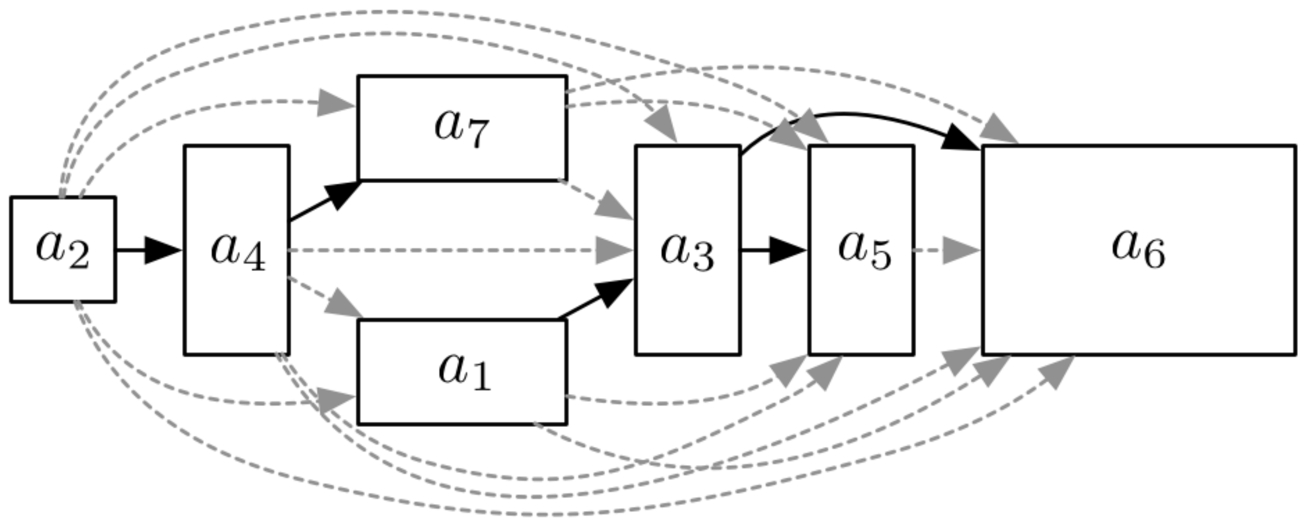
The POS is defined by the superimposed arcs:

- The black arcs correspond to existing precedences
- The orange arcs are implicit
- They exist to prevent overusage of the resource

How do we detect the arcs to be added?

Converting a Schedule into a POS

A possible approach: convert a schedule in a POS

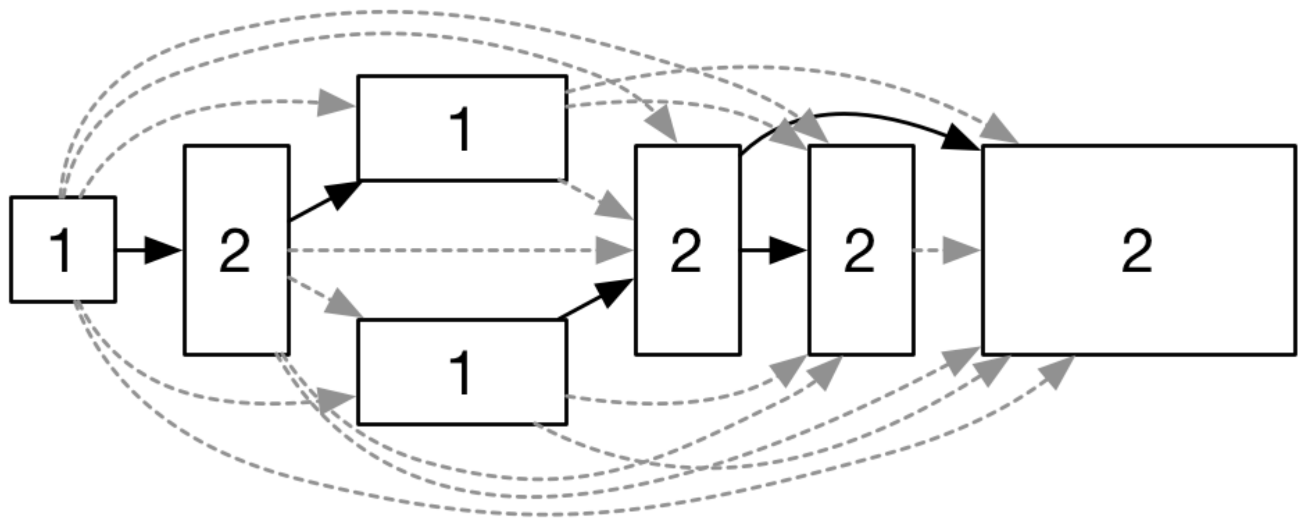


First, we detect all precedence constraints

- The original ones
- All the implicit precedences

Converting a Schedule into a POS

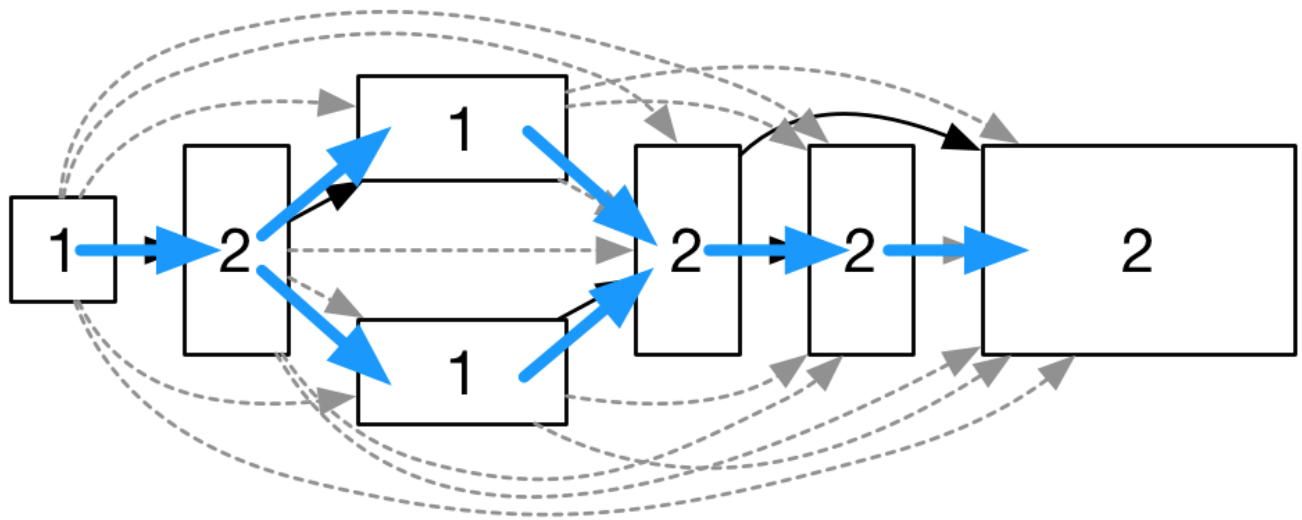
A possible approach: convert a schedule in a POS



We view the activities as arcs, with a flow requirement

Converting a Schedule into a POS

A possible approach: convert a schedule in a POS



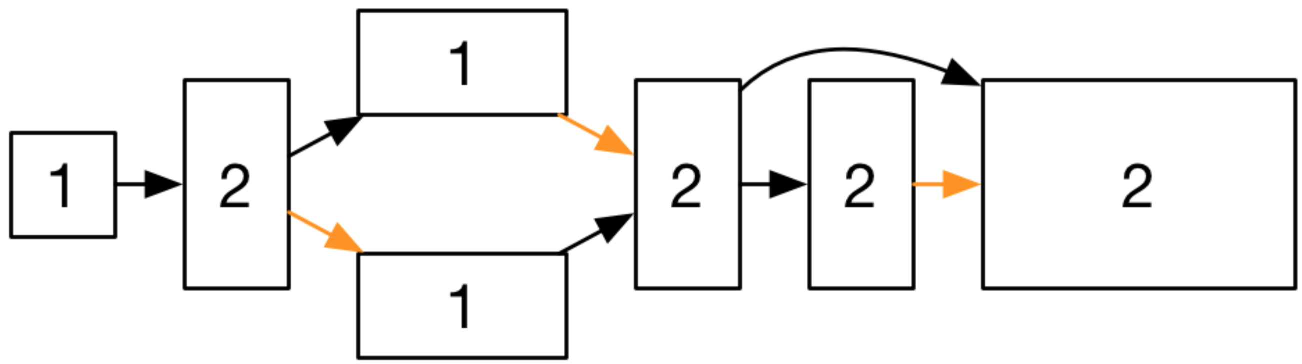
Then we build a feasible flow

- We can use the same algorithm as the GCC propagator
- Blue arcs = non-zero flow

All the arcs with non-zero flow are part of the POS

Converting a Schedule into a POS

A possible approach: convert a schedule in a POS



In this case, we get our example POS

Some Experiments

Benchmarks

- 110 "realistic" instances
- 16 clusters (1 core per cluster)

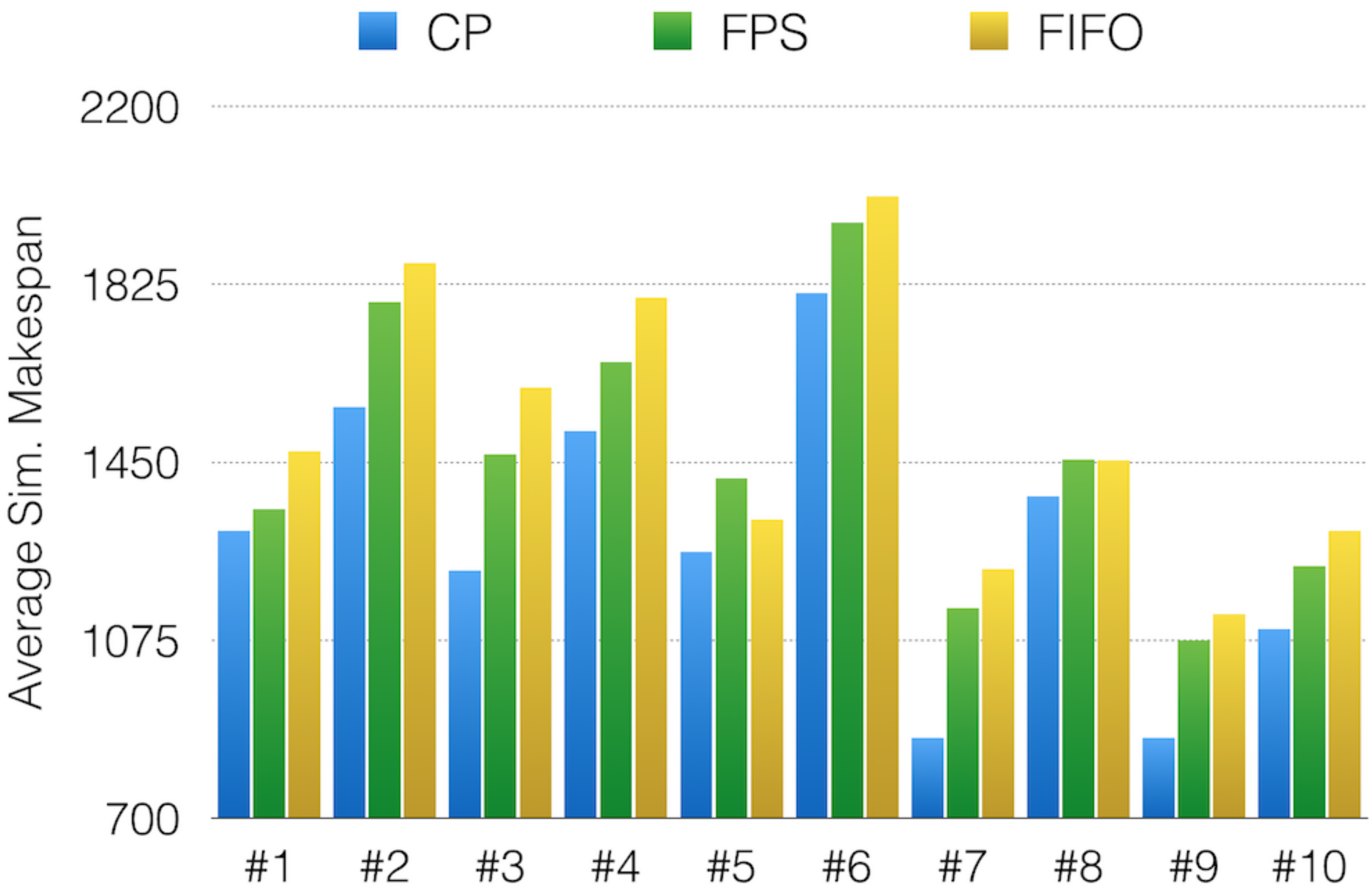
The competitors

- First-In First-Out scheduler
- CP solver + POS conversion
- Fixed Priority Scheduling (Tabu search to optimize priorities)

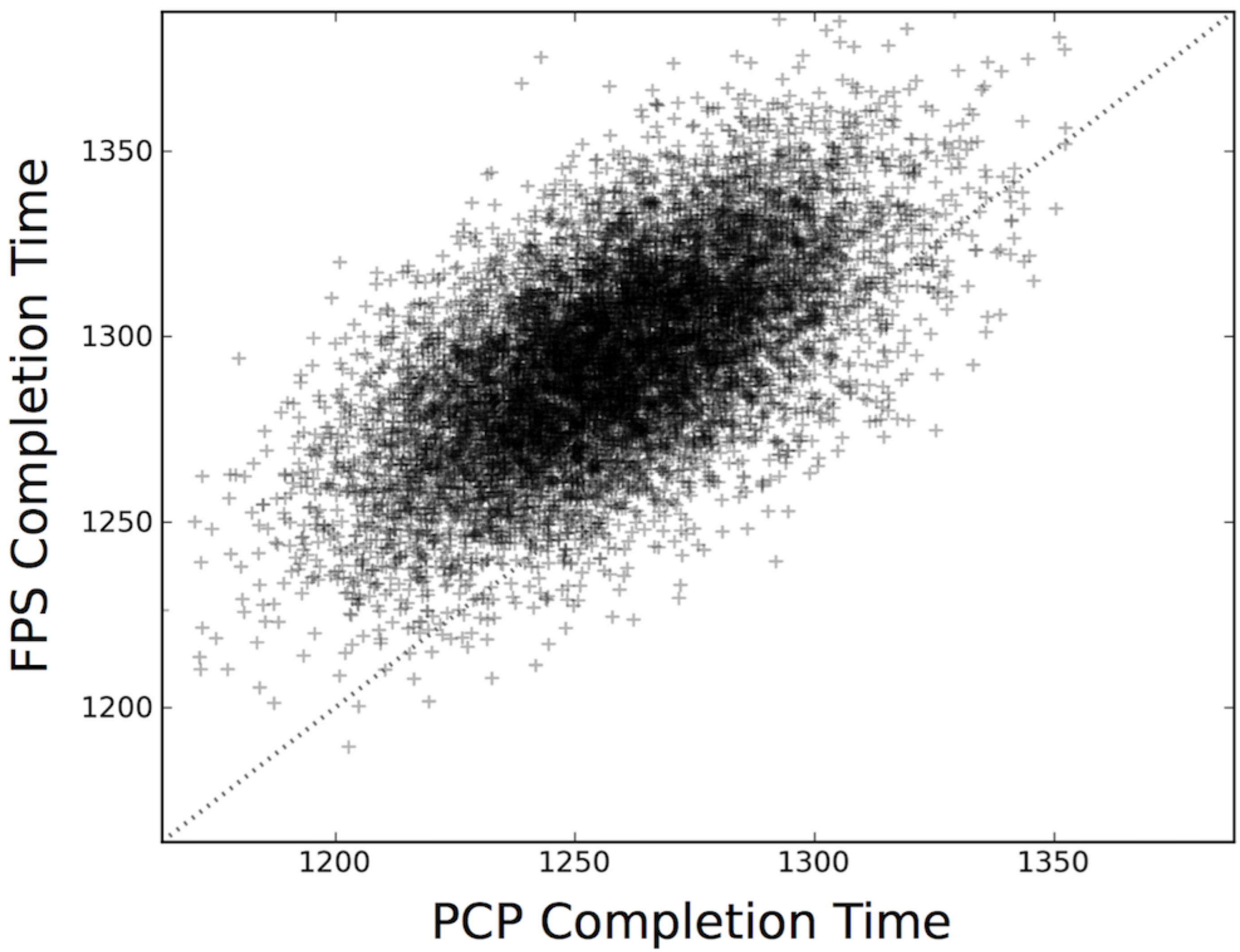
Solution time (off-line)

- FIFO: no off-line part
- CP: < 1 sec to find a very good solution, long opt. proof
- FPS: 4 hours

Some Experiments



Some Experiments



Constraint Systems

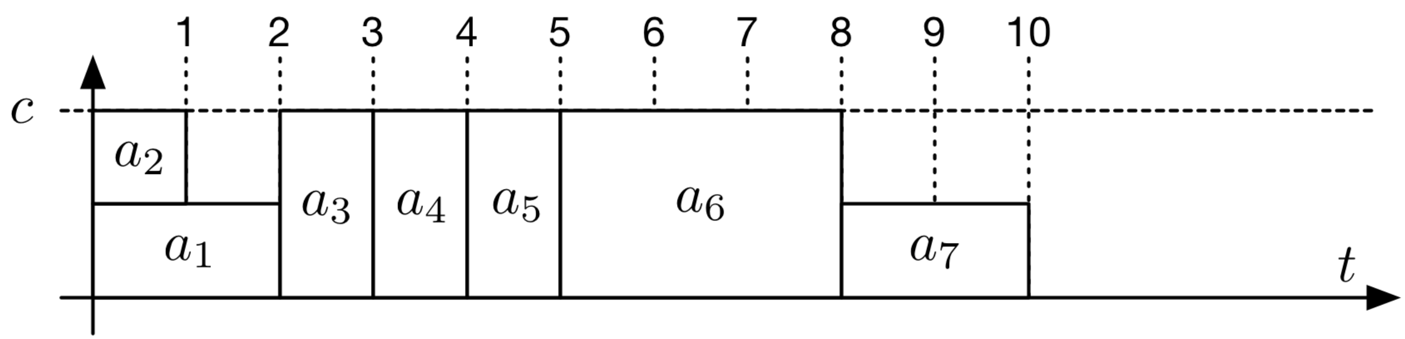
Constraint Based Scheduling
Large Neighbourhood Search

Large Scale Scheduling Problems

What if we have a large scale problem?

- We know that we can use LNS to improve the scalability
- Let's try to apply "textbook" LNS to this problem

We start from the initial solution of our RCPSP

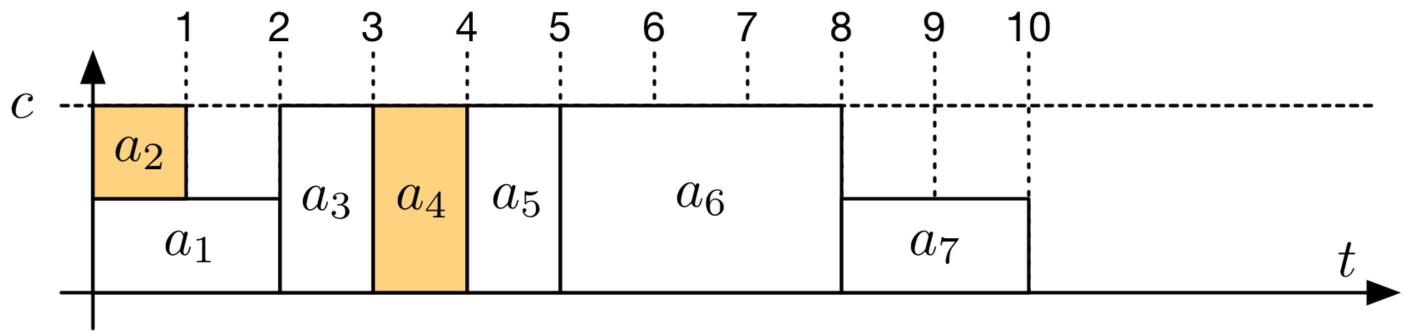


Large Scale Scheduling Problems

What if we have a large scale problem?

- We know that we can use LNS to improve the scalability
- Let's try to apply "textbook" LNS to this problem

We select a few s_i variables (i.e. two activities)...

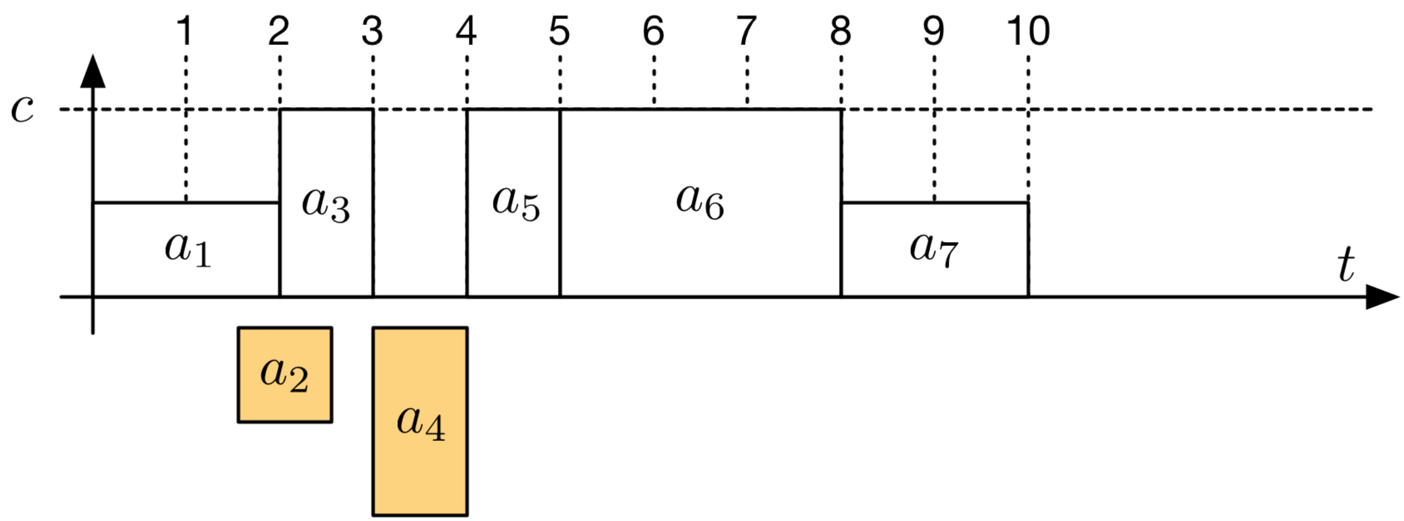


Large Scale Scheduling Problems

What if we have a large scale problem?

- We know that we can use LNS to improve the scalability
- Let's try to apply "textbook" LNS to this problem

...We relax them... And we are stuck :-(



We can obtain no improvement by scheduling a_2 and a_4

Large Scale Scheduling Problems

What if we have a large scale problem?

- We know that we can use LNS to improve the scalability
- Let's try to apply "textbook" LNS to this problem

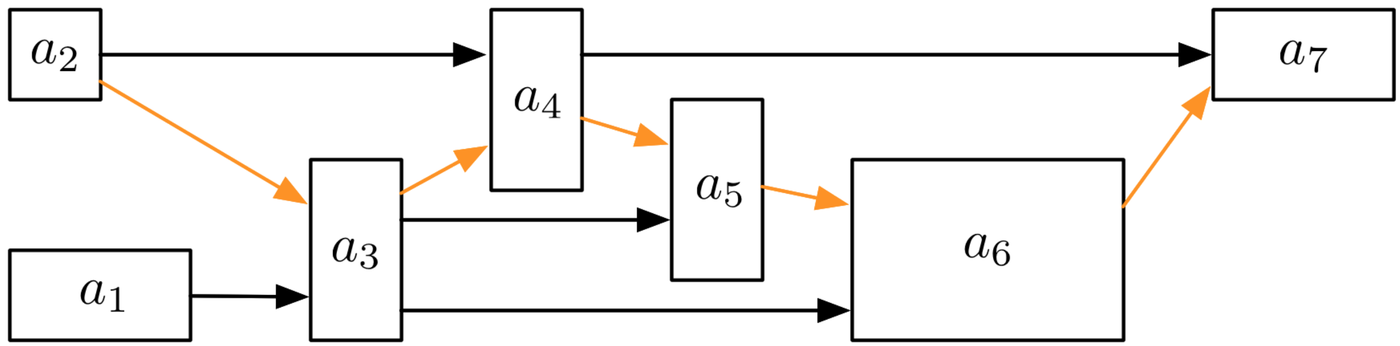
Classical LNS does not work well on many scheduling problems

- The problem is that, by freezing all s_i variables except a few...
- ...We are left with too little flexibility!
- E.g. if we don't relax the last scheduled activity...
- ...We cannot improve the makespan

How can we deal with this issue?

LNS for Scheduling Problems

For example, **we can try to freeze/relax** <u>ordering decisions</u>



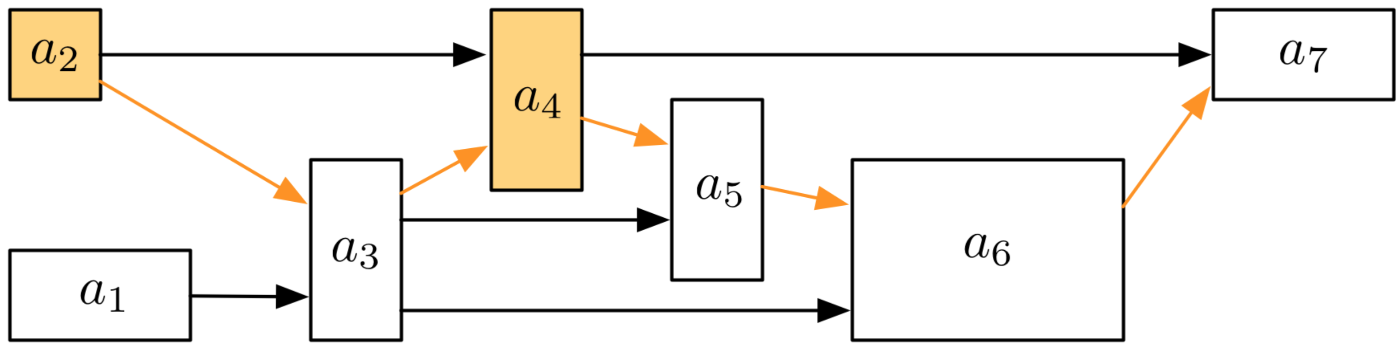
First, we obtain a POS using the method previously described

- Black arcs: original precedences
- Orange arcs: added precedences

The POS encodes all ordering decision in the schedule

LNS for Scheduling Problems

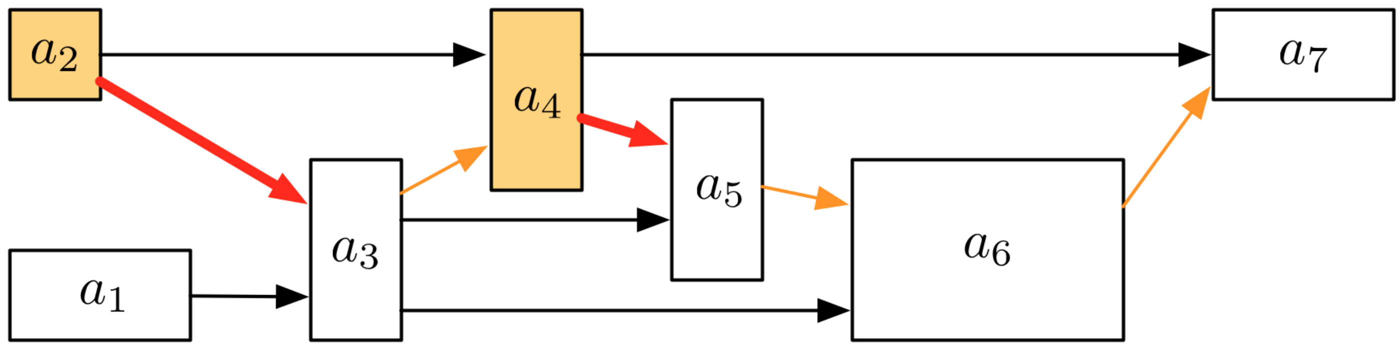
For example, **we can try to freeze/relax** ordering decisions



Then we select some activities to be relaxed (e.g. again a_2 and a_4)

LNS for Scheduling Problems

For example, **we can try to freeze/relax** ordering decisions

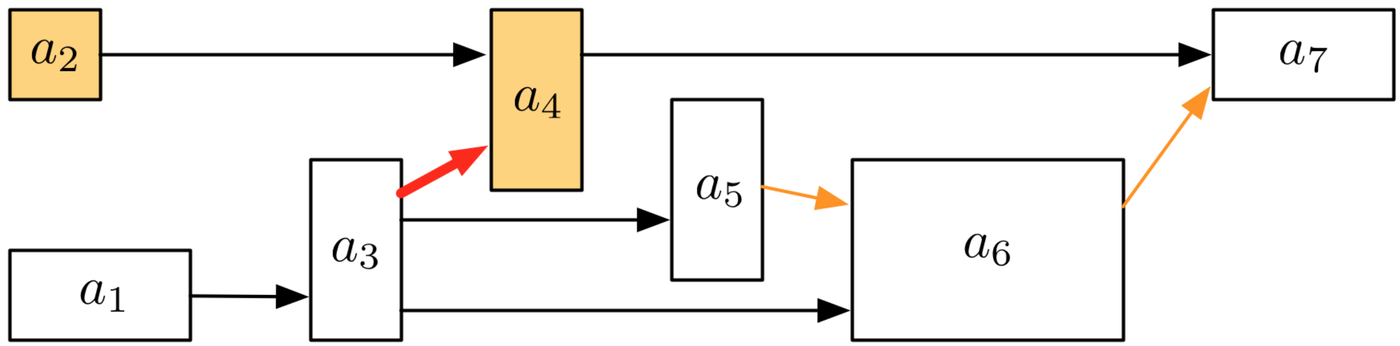


Then we select some activities to be relaxed (e.g. again a_2 and a_4)

- We remove all added arcs that end on selected activities

LNS for Scheduling Problems

For example, **we can try to freeze/relax ordering decisions**

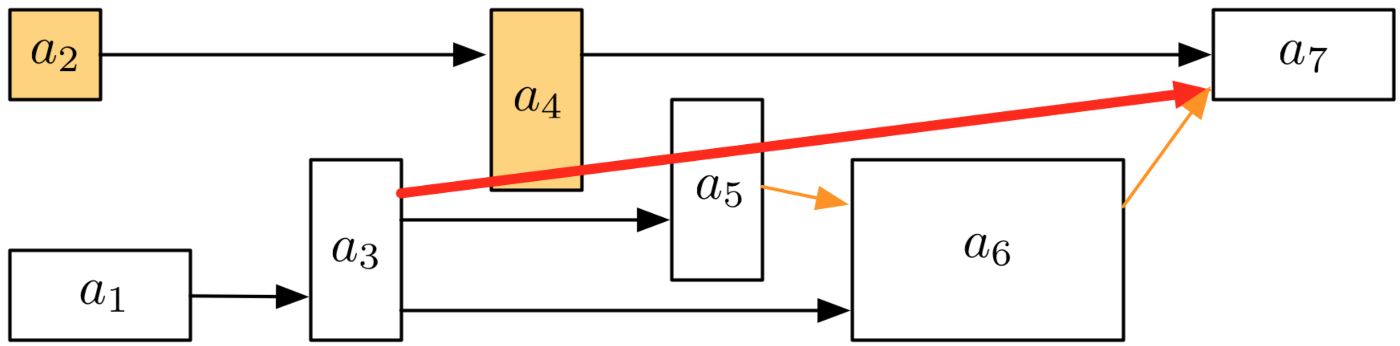


Then we select some activities to be relaxed (e.g. again a_2 and a_4)

- We remove all added arcs that start from selected activities
- We reroute all added arcs that end from non-selected activities…

LNS for Scheduling Problems

For example, **we can try to freeze/relax ordering decisions**



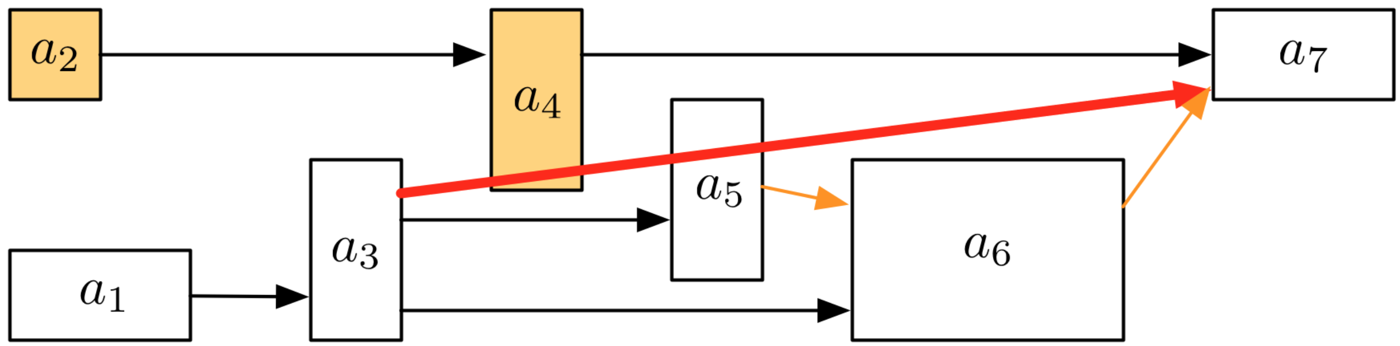
Then we select some activities to be relaxed (e.g. again a_2 and a_4)

- We remove all added arcs that start from selected activities
- We reroute all added arcs that end from non-selected activities...
- ...So that they end on non-selected activities

At the end of the process, we have a new RCPSP

LNS for Scheduling Problems

For example, **we can try to freeze/relax** ordering decisions



- We may choose to branch only on the relaxed variables…
- …Or even on all the variables

In both cases, the problem is much simpler!

- The additional arcs cause a lot of constraint propagation…
- …And provide a very good makespan bound

In this case, by reinserting a_2 and a_4 we easily find the best solution