OSM-S Scheduling

Bram Knippenberg
Joost Kraaijeveld
Jorg Visch

18 september 2017

Samenvatting

Planning en scheduling zijn twee belangrijke onderwerpen binnen de technische informatica. *Planning* gaat over *WAT* er gedaan moet worden. Deze opdracht gaat over *scheduling*: *WANNEER* het gedaan moet worden.

1 De theorie (individueel)

In de literatuur over scheduling wordt vaak een onderscheid gemaakt in een tweetal categoriën “scheduling policies”:

- “fair scheduling policies”
- “priority scheduling policies”

In detail verschillen deze “policies” op diverse manieren, zoals beschreven staat in (Douglass 2004):

> Determining a scheduling strategy is crucial for efficient scheduling of real-time systems. Systems no more loaded than 30 percent have failed because of poorly chosen scheduling policies. Scheduling policies may be *stable*, *optimal*, *responsive*, and/or *robust*. A *stable* policy is one in which, in an overload situation, it is possible to a priori predict which task(s) will miss their timeliness requirements. Policies may also be *optimal*[1]. A *responsive* policy is one in which incoming events are handled in a timely way. Lastly, by *robust*, we mean that the timeliness of one task is not affected by the misbehavior of another…

Beantwoord de volgende vragen over het onderwerp “scheduling policies”:

1. Beschrijf het fundamentele verschil tussen de twee “hoofd-policies”.
2. Beschrijf van elke “hoofd-policies” tenminste twee varianten, uitgezonderd het “Least Laxity” of “Minimum/Least Slack” algoritme, want dat is het onderwerp van deze opdracht.

[1] An optimal policy is one that can schedule a test set if it is possible for any other policy tot do so, (Douglass 2004)

3. Vergelijk de door jou gekozen algoritmen op systematische wijze op hun kenmerken.

Bij het bespreken van de diverse methoden moeten in ieder geval de volgende onderwerpen aan bod komen:

1. Een korte algemene beschrijving van het schedulingalgoritme,
2. De voordelen van het schedulingalgoritme, met als kernwoorden “stable”, “optimal”, “responsive” en “robust” (wat betekenen die woorden en hoe scoort het algoritme daarop). Zie quote 1.
3. De nadelen van het schedulingalgoritme, met als kernwoorden wederom “stable”, “optimal”, “responsive” en “robust” (wat betekenen die woorden en hoe scoort het algoritme daarop). Zie quote 1.
4. Voor wat voor soort problemen het algoritme bij uitstek *geschikt* is en gebruik hierbij de bovenstaande kernwoorden.
5. Voor wat voor soort problemen het algoritme bij uitstek *ongeschikt* is en gebruik hierbij de wederom de bovenstaande kernwoorden.

Beantwoord de bovenstaande vragen over “scheduling policies” kort, in maximaal 4 bladzijden. Gebruik hierbij *niet* een (letterlijke) vertaling van het eerste de beste Wikipedia-lemma of een andere, waarschijnlijk “random” via Google gevonden, webpagina. Geef voorbeelden uit het domeingebied van de programmeeropdracht (met jobs, machines en taken). Kortom, verras jezelf en doe iets wilds: *lees* een aantal verschillende webpagina’s of wellicht boeken en *denk na* om tot slot *in eigen woorden* iets op te schrijven.

2 Het programma (tweetallen)

De opdracht luidt als volgt: Schrijf een programma dat job–shop scheduling problemen oplost met behulp van het “Minimum of Least Slack” algoritme. Als inspiratie voor het algoritme kan je (Russell en Norvig 2003) (zie bijlage A) gebruiken. Ook biedt wikipedia enige hulp:

- `http://nl.wikipedia.org/wiki/Job_sequencing` (Wikipedia 2016)
- Wikipedia 2017 (Wikipedia 2017)

De output van het programma moet een overzicht per job zijn van de start– en eindtijden van de jobs in volgorde van het job–id, alles gescheiden door whitespace, e.g.:

```
0  1  288
1  8  291
2  5  258
3  5  169
4  9  100
5  0  300
```

De starttijd is hierbij het tijdstip van het starten van de eerste taak van een job waarbij de start van de allereerste taak van de alleerste job geldt als tijdstip 0. Het tijdstip van het einde van de taak is het tijdstip waarop de laatste taak van de job eindigt. Als tijdseenheid wordt dezelfde eenheid gebruikt als waar de taak–duur in is aangegeven.

Het programma bevat tenminste de classes JobShop, Job en Task, waarbij de JobShop Jobs heeft, en een Job Tasks heeft. De gegevens (jobs, tasks en aantal machines) moeten vanuit een bestand worden ingelezen. De naam van het bestand moet worden meegegeven als argument aan de applicatie en mag niet hard gecodeerd in de applicatie zitten. Gebruik voor het lezen van het bestand de C++ standaard std::ifstream class. Zie (Stroustrup 2013) §8.7 en (cppreference 2016) voor uitleg in het gebruik.

Het bestand heeft een vast formaat en zie er als volgt uit:

```
6 6
2  1  0  3  1  6  3  7  5  3  4  6
1  8  2  5  4 10  5 10  0 10  3  4
2  5  3  4  5  8  0  9  1  1  4  7
1  5  0  5  2  5  3  3  4  8  5  9
2  9  1  3  4  5  5  4  0  3  3  1
1  3  3  3  5  9  0 10  4  4  2  1
```

De eerste regel geeft aan hoeveel jobs en machines er zijn (hier in beide gevallen 6, maar dat varieert per casus). De overige regels bevatten telkens de taken van een job. Jobs hebben als id het volgnummer in de lijst, i.e. de eerste job (beginnend met `2 1 0`…) is heeft 0 als id, de laatste job (beginnend met `1 3 3`…) 5. De job-regel bevat de taken. De taken staan in de volgorde van uitvoering. Taken hebben als id het volgnummer in de regel, de machine waarop deze moet worden uitgevoerd (eerste getal) en de tijdsduur van de bewerking van die machine (tweede getal), i.e. de eerste job (`2 1 0 3 1 6 3 7 5 3 4 6`) heeft als eerste taak taak 0 op machine 2 met een tijdsduur van 1 en als tweede taak taak 1 op machine 0 met een tijdsduur van 3. De taken en de tijdsduur van de bewerking worden gescheiden door een willekeurige hoeveelheid “whitespace”.

Je mag alleen gebruik maken van classes en functies die standaard bij een C++ compiler (C++11) zitten. Er mag dus geen gebruik gemaakt worden van externe libraries als Boost. Ook mag er geen platform (Windows/Linux/mac) afhankelijke code gebruikt worden. Er zijn pluspunten te verdienen voor zij die de std::regex classen gebruiken. (zie h7 Stroustrup 2013)

3 Inleveren

Het inleveren gebeurt in twee fases. Bij de eerste oplevering lever je je werk in wat je tot dusver gedaan hebt, zodat de docent je van feedback kan voorzien. Zonder begeleidend ontwerp wordt er niet naar je code gekeken. De tweede oplevering is de definitieve oplevering van het eindproduct.

- Eerste oplevering: uiterlijk vrijdag 25 september voor 12.00u, per mail. Lever in wat je op dat moment hebt. In ieder geval moet de theorie daar *in concept* bij zitten. En *in concept* betekent: in beginsel "feature compleet" er hoeft alleen nog maar wat aan de taal gesleuteld te worden.
- Definitieve oplevering: vrijdag lesweek 5, zie het toetsrooster in iSAS voor de definitieve deadline

Te laat inleveren betekent niet nakijken en dus geen voldoende cijfer.

Referenties

cppreference (2016). *Input/output library - cppreference.com*. URL: `http://en.cppreference.com/w/cpp/io`.

Douglass, Bruce Powel (2004). *Real Time UML: Advances in the UML for Real-Time Systems (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc. ISBN: 0321160762.

Russell, Stuart en Peter Norvig (2003). *Artificial Intelligence: A Modern Approach*. 2nd edition. Prentice-Hall, Englewood Cliffs, NJ. ISBN: 0130803022.

Stroustrup, B. (2013). *A Tour of C++*. C++ In-Depth Series. Pearson Education. ISBN: 9780133549003.

Wikipedia (2016). *Job sequencing - Wikipedia*. URL: `http://nl.wikipedia.org/wiki/Job_sequencing`.

Wikipedia (2017). *Least slack time scheduling - Wikipedia*. URL: `http://en.wikipedia.org/wiki/Least_slack_time_scheduling`.

A Planning and acting in the real world

Zie de volgende pagina voor het artikel ’Planning and acting in the real world’ (Douglass 2004)

11 PLANNING AND ACTING IN THE REAL WORLD

In which we see how more expressive representations and more interactive agent architectures lead to planners that are useful in the real world.

The previous chapter introduced the most basic concepts, representations, and algorithms for planning. Planners that are are used in the real world for planning and scheduling the operations of spacecraft, factories, and military campaigns are more complex; they extend both the representation language and the way the planner interacts with the environment. This chapter shows how. Section 11.1 extends the classical language for planning to talk about actions with durations and resource constraints. Section 11.2 describes methods for constructing plans that are organized hierarchically. This allows human experts to communicate to the planner what they know about how to solve the problem. Hierarchy also lends itself to efficient plan construction because the planner can solve a problem at an abstract level before delving into details. Section 11.3 presents agent architectures that can handle uncertain environments and interleave deliberation with execution, and gives some examples of real-world systems. Section 11.4 shows how to plan when the environment contains other agents.

11.1 TIME, SCHEDULES, AND RESOURCES

The classical planning representation talks about *what to do*, and in *what order*, but the representation cannot talk about time: *how long* an action takes and *when* it occurs. For example, the planners of Chapter 10 could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of **scheduling**. The real world also imposes many **resource constraints**; for example, an airline has a limited number of staff—and staff who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.

The approach we take in this section is "plan first, schedule later": that is, we divide the overall problem into a *planning* phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later *scheduling* phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

$Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\},$
$\{AddEngine2 \prec AddWheels2 \prec Inspect2\})$

$Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))$

$Action(AddEngine1,$ DURATION:30,
USE:$EngineHoists(1))$
$Action(AddEngine2,$ DURATION:60,
USE:$EngineHoists(1))$
$Action(AddWheels1,$ DURATION:30,
CONSUME:$LugNuts(20)$, USE: $WheelStations(1))$
$Action(AddWheels2,$ DURATION:15,
CONSUME:$LugNuts(20)$, USE: $WheelStations(1))$
$Action(Inspect_i,$ DURATION:10,
USE:$Inspectors(1))$

Figure 11.1 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B.

This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. The automated methods of Chapter 10 can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness. GRAPHPLAN (Section 10.3), SATPLAN (Section 10.4.1), and partial-order planners (Section 10.4.4) can do this; search-based methods (Section 10.2) produce totally ordered plans, but these can easily be converted to plans with minimal ordering constraints.

11.1.1 Representing temporal and resource constraints

JOB
DURATION
CONSUMABLE
REUSABLE
MAKESPAN

A typical **job-shop scheduling problem**, as first introduced in Section 6.1.2, consists of a set of **jobs**, each of which consists a collection of **actions** with ordering constraints among them. Each action has a **duration** and a set of resource constraints required by the action. Each constraint specifies a *type* of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is **consumable** (e.g., the bolts are no longer available for use) or **reusable** (e.g., a pilot is occupied during a flight but is available again when the flight is over). Resources can also be *produced* by actions with negative consumption, including manufacturing, growing, and resupply actions. A solution to a job-shop scheduling problem must specify the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the **makespan**.

Figure 11.1 shows a simple example: a problem involving the assembly of two cars. The problem consists of two jobs, each of the form $[AddEngine, AddWheels, Inspect]$. Then the

$Resources$ statement declares that there are four types of resources, and gives the number of each type available at the start: 1 engine hoist, 1 wheel station, 2 inspectors, and 500 lug nuts. The action schemas give the duration and resource needs of each action. The lug nuts are *consumed* as wheels are added to the car, whereas the other resources are "borrowed" at the start of an action and released at the action's end.

The representation of resources as numerical quantities, such as $Inspectors(2)$, rather than as named entities, such as $Inspector(I_1)$ and $Inspector(I_2)$, is an example of a very general technique called **aggregation**. The central idea of aggregation is to group individual objects into quantities when the objects are all indistinguishable with respect to the purpose at hand. In our assembly problem, it does not matter *which* inspector inspects the car, so there is no need to make the distinction. (The same idea works in the missionaries-and-cannibals problem in Exercise 3.9.) Aggregation is essential for reducing complexity. Consider what happens when a proposed schedule has 10 concurrent $Inspect$ actions but only 9 inspectors are available. With inspectors represented as quantities, a failure is detected immediately and the algorithm backtracks to try another schedule. With inspectors represented as individuals, the algorithm backtracks to try all 10! ways of assigning inspectors to actions.

AGGREGATION

11.1.2 Solving scheduling problems

We begin by considering just the temporal scheduling problem, ignoring resource constraints. To minimize makespan (plan duration), we must find the earliest start times for all the actions consistent with the ordering constraints supplied with the problem. It is helpful to view these ordering constraints as a directed graph relating the actions, as shown in Figure 11.2. We can apply the **critical path method** (CPM) to this graph to determine the possible start and end times of each action. A **path** through a graph representing a partial-order plan is a linearly ordered sequence of actions beginning with $Start$ and ending with $Finish$. (For example, there are two paths in the partial-order plan in Figure 11.2.)

CRITICAL PATH METHOD

The **critical path** is that path whose total duration is longest; the path is "critical" because it determines the duration of the entire plan—shortening other paths doesn't shorten the plan as a whole, but delaying the start of any action on the critical path slows down the whole plan. Actions that are off the critical path have a window of time in which they can be executed. The window is specified in terms of an earliest possible start time, ES, and a latest possible start time, LS. The quantity $LS - ES$ is known as the **slack** of an action. We can see in Figure 11.2 that the whole plan will take 85 minutes, that each action in the top job has 15 minutes of slack, and that each action on the critical path has no slack (by definition). Together the ES and LS times for all the actions constitute a **schedule** for the problem.

CRITICAL PATH

SLACK

SCHEDULE

The following formulas serve as a definition for ES and LS and also as the outline of a dynamic-programming algorithm to compute them. A and B are actions, and $A \prec B$ means that A comes before B:

$$
\begin{aligned}
&ES(Start) = 0 \\
&ES(B) = \max_{A \prec B} ES(A) + Duration(A) \\
&LS(Finish) = ES(Finish) \\
&LS(A) = \min_{B \succ A} LS(B) - Duration(A) \ .
\end{aligned}
$$

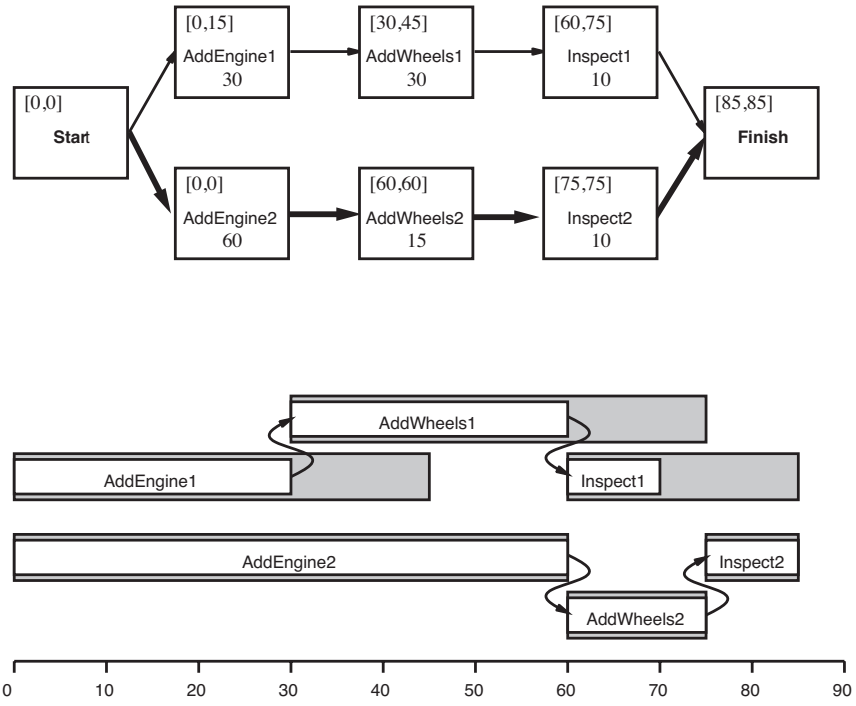

Figure 11.2 Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.1. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair $[ES, LS]$, displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Grey rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a gray rectangle indicates the slack.

The idea is that we start by assigning $ES(Start)$ to be 0. Then, as soon as we get an action B such that all the actions that come immediately before B have ES values assigned, we set $ES(B)$ to be the maximum of the earliest finish times of those immediately preceding actions, where the earliest finish time of an action is defined as the earliest start time plus the duration. This process repeats until every action has been assigned an ES value. The LS values are computed in a similar manner, working backward from the $Finish$ action.

The complexity of the critical path algorithm is just $O(Nb)$, where N is the number of actions and b is the maximum branching factor into or out of an action. (To see this, note that the LS and ES computations are done once for each action, and each computation iterates over at most b other actions.) Therefore, finding a minimum-duration schedule, given a partial ordering on the actions and no resource constraints, is quite easy.

Mathematically speaking, critical-path problems are easy to solve because they are defined as a *conjunction* of *linear* inequalities on the start and end times. When we introduce resource constraints, the resulting constraints on start and end times become more complicated. For example, the $AddEngine$ actions, which begin at the same time in Figure 11.2,

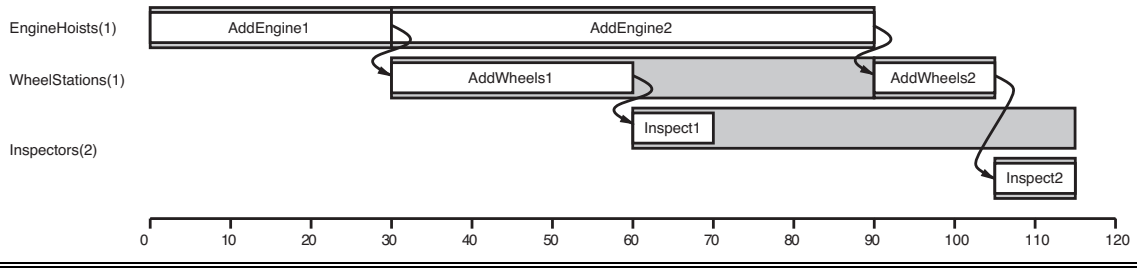

Figure 11.3 A solution to the job-shop scheduling problem from Figure 11.1, taking into account resource constraints. The left-hand margin lists the three reusable resources, and actions are shown aligned horizontally with the resources they use. There are two possible schedules, depending on which assembly uses the engine hoist first; we've shown the shortest-duration solution, which takes 115 minutes.

require the same $EngineHoist$ and so cannot overlap. The "cannot overlap" constraint is a *disjunction* of two linear inequalities, one for each possible ordering. The introduction of disjunctions turns out to make scheduling with resource constraints NP-hard.

Figure 11.3 shows the solution with the fastest completion time, 115 minutes. This is 30 minutes longer than the 85 minutes required for a schedule without resource constraints. Notice that there is no time at which both inspectors are required, so we can immediately move one of our two inspectors to a more productive position.

The complexity of scheduling with resource constraints is often seen in practice as well as in theory. A challenge problem posed in 1963—to find the optimal schedule for a problem involving just 10 machines and 10 jobs of 100 actions each—went unsolved for 23 years (Lawler *et al.*, 1993). Many approaches have been tried, including branch-and-bound, simulated annealing, tabu search, constraint satisfaction, and other techniques from Chapters 3 and 4. One simple but popular heuristic is the **minimum slack** algorithm: on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the ES and LS times for each affected action and repeat. The heuristic resembles the minimum-remaining-values (MRV) heuristic in constraint satisfaction. It often works well in practice, but for our assembly problem it yields a 130–minute solution, not the 115–minute solution of Figure 11.3.

MINIMUM SLACK

Up to this point, we have assumed that the set of actions and ordering constraints is fixed. Under these assumptions, every scheduling problem can be solved by a nonoverlapping sequence that avoids all resource conflicts, provided that each action is feasible by itself. If a scheduling problem is proving very difficult, however, it may not be a good idea to solve it this way—it may be better to reconsider the actions and constraints, in case that leads to a much easier scheduling problem. Thus, it makes sense to *integrate* planning and scheduling by taking into account durations and overlaps during the construction of a partial-order plan. Several of the planning algorithms in Chapter 10 can be augmented to handle this information. For example, partial-order planners can detect resource constraint violations in much the same way they detect conflicts with causal links. Heuristics can be devised to estimate the total completion time of a plan. This is currently an active area of research.