Partial-Order Support-Link Scheduling

Debdeep Banerjee and **Patrik Haslum**
Australian National University & NICTA
firstname.lastname@anu.edu.au

Abstract

Partial-order schedules are valued because they are flexible, and therefore more robust to unexpected delays. Previous work has indicated that constructing partial-order schedules by a two-stage method, in which a fixed-time schedule is first found and a partial order then lifted from it, is far more efficient than constructing them directly by a least-commitment partial-order scheduling algorithm. However, the two-stage method is limited to exploring only a fraction of the space of partial-order schedules, namely those that can be obtained from the given fixed-time schedule. We introduce a novel constraint formulation of partial-order scheduling, which establishes explicit resource-providing "links" between activities instead of detecting and eliminating potential resource conflicts. We show that this yields an algorithm that is much faster than previous (precedence constraint posting) partial-order scheduling methods, and comparable to the two-stage method in terms of the quality and robustness of the schedules it finds. This algorithm is also complete, and because it searches the entire space of partial-order schedules, can be adapted to optimising different robustness criteria.

Introduction

A partial-order schedule is one which does not specify a fixed start time for each activity, but only a set of time constraints between tasks such that any realisation that meets these time constraints is guaranteed to also respect resource constraints. Thus, a partial-order schedule represents a set of possible fixed-time schedules, all of which are feasible. The time constraints of the schedule must be efficiently checkable, and therefore are normally represented by a simple (i.e., non-disjunctive) temporal constraint network (STN). Partial-order schedules are useful because they retain more flexibility, and are therefore more robust to deviations at execution time. If, for example, an activity takes more time than expected, only future activities constrained by that activity need to be delayed, and their adjusted start times can be efficiently determined from the STN.

Recently, Policella et al. (2004; 2007; 2009) examined different algorithms for generating partial-order schedules, comparing them both in terms of the efficiency of the scheduling algorithm and the quality of the resulting schedule. In this setting, schedule quality is measured both by the traditional makespan objective and measures of how robust the schedule is to disturbances. The algorithms they compared are a least-commitment, precedence constraint-posting algorithm which generates a partial-order schedule directly (called the envelope-based algorithm, or EBA), and a two-stage procedure which constructs a fixed-time schedule that is then transformed into a partial-order schedule by a post-processing step (called the earliest-start time algorithm with chaining, or ESTA^C). Somewhat surprisingly, they found that the two-stage method outperformed the partial-order scheduling algorithm, both in terms of efficiency and the quality of schedules generated.

In this paper, we present a new, constraint-based, method for generating partial-order schedules, which we call partial-order support-link (POSL) scheduling. Like the EBA, this algorithm searches in the space of partial-order schedules directly, rather than "deordering" a fixed-time schedule like the two-stage method, ESTA^C, does. Applied to the RCPSP/max problem, the POSL scheduling algorithm produces schedules as good as those found by ESTA^C, and does so much faster than the EBA, although not as fast as ESTA^C. There are, however, advantages other than speed to constructing a partial-order schedule directly, which justify exploring such methods. For one, the search can be directed towards maximising some measure of flexibility or robustness over the entire space of partial-order schedules, whereas the two-stage method is limited to finding only the most flexible or robust partialisation of one or a few given fixed-time schedules. The POSL scheduling algorithm is also complete, and it can be easily extended to problems with more complex choices and constraints, such as multi-mode scheduling problems.

Precedence constraint-posting scheduling algorithms, like the EBA, or the methods described by Laborie (2003), work by identifying *resource conflicts* – sets of activities that may overlap in time and whose combined resource use exceeds capacity – and resolving those conflicts by adding precedence constraints between some of the conflicting activities. Partial-order support-link scheduling, in contrast, works by selecting for each activity a *support* that provides the resources it requires. In a scheduling problem with multi-capacity, non-consumed resources, such as RCPSP/max, this support will be a set of activities using the same resources: when those activities finish, they will release the

resources they were using, thus providing them to the next activity. Thus, support is indicated by links between activities, and those links imply precedence constraints.

As the name suggests, POSL scheduling is inspired by partial-order causal-link (POCL) planning methods (*e.g.* McAllester and Rosenblitt 1991; Vidal and Geffner 2004). Readers familiar with POCL planning may find this analogy helpful: In planning, to execute an action one must ensure that its preconditions are satisfied, and a POCL planner does this by posting, to each precondition, an explicit link from an earlier action that establishes the truth of the condition. In scheduling, to execute an activity one must ensure that its required resources are available, and the POSL scheduler does this by posting explicit links from earlier activities that provide those resources.

The RCPSP/max Problem

Following Policella et al., we use the single-mode RCPSP/max (*e.g.* Kolisch and Padman 2001) problem to evaluate the POSL scheduling algorithm. A RCPSP/max problem instance consists of a set of activities, $A = \{0, \ldots, n+1\}$, where 0 and $n+1$ are "dummy" activities marking the start and end, and a set of resources, $R = \{1, \ldots, m\}$, each with a given capacity $\mathrm{cap}(r)$. Each task i has a fixed processing time, d_i, and a requirement $\mathrm{req}(i, r)$ for each resource (which may be zero, if the task does not use resource r). The processing time of the start and end activities is zero. Additionally, there is a set of given time constraints, in the form of differences between activity start times: $S_j - S_i \geq \delta_{ij}$. That $S_i - S_0 \geq 0$ and $S_{n+1} - S_i \geq d_i$, for $i = 1, \ldots, n$, i.e., that each real activity takes place in between the start and end, is given. The objective is to schedule all tasks, respecting time and resource constraints, so as to minimise makespan.

The RCPSP/max problem is very flexible. For example, an ordinary precedence constraint, $i \prec j$, is expressed as $S_j - S_i \geq d_i$, while a maximum delay, i.e., that j must start no later than t time units after the start of i, can be expressed as $S_i - S_j \geq -t$. For this reason, even deciding if a problem has a feasible solution is NP-hard.

The POSL Scheduling Algorithm

The POSL scheduling algorithm is constraint-based, i.e., it consists of a constraint model of the problem, a collection of propagation rules and a branching/search scheme.

Constraint Model

Each activity is associated with a start time S_i and an end time E_i, where $E_i = S_i + d_i$. The given time constraints map directly to constraints on the start time variables, as shown above. For each resource r, the model contains a *support matrix*, $P_r(\cdot, \cdot)$, where $P_r(i, j)$ is the amount of resource r that activity i will provide to activity j. The resource constraints of the problem translate into the following:

$$\sum_{j=0,\ldots,n} P_r(j, i) = \mathrm{req}(i, r) = \sum_{j=1,\ldots,n+1} P_r(i, j)$$

for each $i = 1, \ldots, n$, i.e., for each real activity, the amount of the resource provided to it equals the amount it requires, and this also equals the amount it provides to other activities. For the dummy start and end activties we have

$$\sum_{j=1,\ldots,n+1} P_r(0, j) = \mathrm{cap}(r) = \sum_{j=0,\ldots,n} P_r(j, n+1).$$

and $\sum_j P_r(j, 0) = \sum_j P_r(n+1, j) = 0$. As can be seen, these are essentially flow constraints: the start activity is the only source, the end activity the only sink, and all real activities preserve flow. This is because resource use is non-consuming. If $P_r(i, j) > 0$, we say there is a *support link* from i to j. This implies a precedence constraint, i.e., $(P_r(i, j) > 0) \rightarrow (S_j - S_i \geq d_i)$.

Propagation and Search

During search, we maintain bounds on the differences between activity start times, similar to an STN. Let $l_{i,j}$ and $u_{i,j}$ denote the lower and upper bounds on the difference $S_j - S_i$, respectively. (We do not perform complete propagation of upper bounds, as explained below.)

Implicit in the time bounds is a precedence relation, defined by $i \prec j$ iff $l_{i,j} \geq d_i$, i.e., j must start an amount of time after the start of i that is at least the duration of activity i, or, equivalently, j must start after the end of i. Likewise, we can define an "anti-precedence" relation, meaning that i cannot end before the start of j, as $i \bar{\prec} j$ iff $u_{i,j} < d_i$.[1] Note that $\prec$ and $\bar{\prec}$ are not each others negation: if the start times of i and j are loosely constrained, we can have $i \not\prec j$ and $i \not\bar{\prec} j$. However, the relations are mutually exclusive, i.e., it will never be the case that both $i \prec j$ and $i \bar{\prec} j$ hold.

Let $U_r(i, j)$ and $L_r(i, j)$ denote upper and lower bounds on $P_r(i, j)$. The remaining, i.e., currently unsatisfied, demand (for r) of activity i is $\mathrm{req}(i, r) - \sum_j L_r(j, i)$. Similarly, the remaining, i.e., currently unallocated, support (of r) of i is $\mathrm{req}(i, r) - \sum_j L_r(i, j)$. We say that i is a possible predecessor of j (on r) if $(U_r(i, j) - L_r(i, j)) > 0$ and $i \not\bar{\prec} j$. This means that it is possible to establish a support link from i to j, but that this has not yet been done. (It could also mean it is possible increase the flow along an existing link. However, when establishing a new support link from i to j we always assign it the maximum flow, so this case does not arise.)

Initial Bounds Clearly, we have $P_r(i, j) = 0$ whenever $i \bar{\prec} j$, since a support link from i to j implies $i \prec j$. If $i \not\bar{\prec} j$, we have $P_r(i, j) \leq \min(\mathrm{req}(i, r), \mathrm{req}(j, r))$.

Branching Rule and Strategy The branching rule is a binary choice between $P_r(i, j) = U_r(i, j)$ and $P_r(i, j) < U_r(i, j)$, i.e., between establishing a support link with maximum flow and lowering the upper bound on maximum flow by 1. To choose a pair of activities to branch on, we take a j with minimum earliest start time, and a resource r such that the remaining demand of j for r is non-zero: if there are

[1] This is not the same as the anti-precedence graph introduced by Muscettola (2002), which is a relation over time points rather than activities (i.e., intervals). Note that $\bar{\prec}$ is not transitive.

several, we take the j and r such that the smallest $U_r(i,j)$ over all possible predecessors i of j on r is minimum. (This aims to choose a most constrained activity that is not fully supported.) Then we choose a possible predecessor i of j (on r) such that the remaining demand of i (for r) is zero, and the earliest end time of i is minimum.

Additional Propagation The basic model constraints yield some obvious propagation rules, e.g., from bounds on the temporal constraints we can infer anti-precendence relations, from which we can infer impossible supports. Apart from these, we use the following special propagation rules.

1. For each resource r and activity i that uses r, we calculate the *reserved flow*, which is the amount of the resource provided to i that is accounted for by complete chains of activities from the start. This is given by the fixpoint of

$$\mathrm{RF}(i,r) = \sum_j \min(L_r(j,i), \mathrm{RF}(j,r))$$

and $\mathrm{RF}(0,r) = \mathrm{cap}(r)$. Then, for any activity j with remaining demand for r greater than $\mathrm{cap}(r) - \mathrm{RF}(i,r)$, we can infer that $i \prec j$. This is because the amount $\mathrm{RF}(i,r)$ is already "allocated", to either an activity or a support link, at every point from the begining to the start of i, and thus not available to support j until after i has completed.

2. For any activity i with non-zero remaining demand for resource r, let W be any subset of the possible predecessors of i (on r) such that the total support that activities in W can provide to i, i.e., $\sum_{j \in W} U_r(j,i)$, is at least equal to the remaining demand of i, and let $\mathrm{eet}(W)$ be the latest earliest end time among activities in W. The earliest start time of i is lowerbounded by $\min_W \mathrm{eet}(W)$, because some such set W must eventually be chosen to support i. To find the lower bound, we sort the possible predecessors of i by increasing earliest end time and add their maximum support until the point where it is sufficient.

Incremental Maintenance Although the precedence and anti-precedence relations, sets of possible predecessors, etc., are defined in terms of the basic model constraints, we do not compute them that way. Instead, they, and the bounds on support matrices, are maintained by incremental updates following each branching choice.

We also do not use complete propagation of temporal constraints, because this is computationally expensive. We compute only transitive updates to the lower bounds (based on minimum time lags and posted precedences), and check them against the maximum time lags given in the problem instance. This is sufficient to detect any inconsistency, since during the search we never post new maximum time lag constraints. However, it does mean that some anti-precedence relations may not be detected, and thus that we may overestimate the sets of possible predecessors and therefore explore branches that could have been cut off earlier.

Evaluation

We evaluated the POSL scheduling algorithm on the J10, J20, J30 and C100 RCPSP/max problem sets.[2]

[2]Available from PSPLib, `http://129.187.106.231/psplib/`.

	% solved	|*flex*|	|*fldt*|	Time	Makespan
J10					
POSL	97.86				
first		0.235	0.656	0.56	46.91
best (mksp)		0.237	0.660	0.85	45.75
best (fldt)		0.266	0.674	3.29	46.89
EBA	97.78	0.16	0.65	0.19	55.47
ESTA^C					
(2007)	98.15	0.20	0.68	0.03	46.70
(2009)	96.3	0.19	0.68	0.02	49.49
J20					
POSL	88.58				
first		0.242	0.600	0.28	78.17
best (mksp)		0.235	0.588	1.05	76.65
best (fldt)		0.255	0.613	4.15	78.19
EBA	89.63	0.13	0.58	1.99	94.03
ESTA^C					
(2007)	96.67	0.19	0.64	0.19	72.75
(2009)	95.6	0.16	0.65	0.12	83.97
J30					
POSL	81.08				
first		0.250	0.532	1.21	96.46
best (mksp)		0.243	0.528	2.42	94.38
best (fldt)		0.257	0.545	6.93	96.60
EBA	82.22	0.16	0.56	10.94	116.10
ESTA^C					
(2007)	97.04	0.25	0.64	0.83	78.55
(2009)	96.3	0.25	0.59	0.41	107.08
C100					
POSL	60.15				
first		0.144	0.560	26.84	403.37
best (mksp)		0.138	0.545	36.53	400.31
best (fldt)		0.145	0.561	47.17	403.32
EBA	27.04	0.11	0.56	183.79	632.27
ESTA^C					
(2007)	99.26	0.07	0.50	0.48	374.35
(2009)	99.3	0.05	0.50	2.07	440.79

Table 1: Average results achieved by different scheduling algorithms. Results for EBA and ESTA^C are taken from Policella et al., 2007; 2009. Note that averages are taken over different sets of problems for each algorithm, and thus not directly comparable.

Like Policella et al., we use two measures of the temporal robustness of a schedule: *flexibility*, defined as the fraction of pairs of activities that are not sequenced, and *fluidity*, which aims to measure the ability of the schedule to absorb delays. Fluidity is defined as the slack between activities i and j (i.e., the difference between their maximum and minimum separation) as a percentage of the schedule horizon, averaged over all distinct pairs i, j. This measure depends on the horizon, which is taken to be the sum of all activity durations and the sum of all minimal time lags in the problem. (This ensures that there is a valid schedule with a makespan less than or equal to the horizon, if any valid schedule exists.) The flexibility and fluidity of a schedule are a function of the time constraints, both those given as part of the problem specification and those added by the scheduler

to ensure resource feasibility. Therefore, Policella et al. report normalised values, $|\mu(S)| = {}^{\mu(S)}/_{\mu(P)}$, where $\mu(S)$ is the measure (flexibility or fluidity) computed for the solution, and $\mu(P)$ is the same measure computed taking into account only the original problem time constraints. To facilitate comparison, we follow the same formula.

Table 1 summarises our results. For reference, we include the results achieved by EBA and ESTA^C (from Policella et al., 2007; 2009; both are for the quadratic MCS variant). Our results, and those of Policella et al. 2009, are averages over solved instances in each problem set, while the results reported by Policella et al. in 2007 (for EBA and ESTA^C) are averages over the set of instances solved by *all* algorithms they compared, which is in each case a much smaller set. Also, our tests were run on a roughly twice as fast computer. Thus, the average values are not directly comparable. Nevertheless, they indicate that POSL is faster than EBA, and solves more problems, while achieving schedule robustness comparable to that of ESTA^C. However, the iterated chaining methods introduced by Policella et al. in 2009 still yield higher robustness measures.

Integrated into a systematic search, the POSL scheduling algorithm is complete. For the purpose of experimental evaluation, we count an instance as unsolved if no solution was found in 60 seconds. EBA and ESTA are both incomplete algorithms.

Optimising Makespan or Robustness

Because it is a complete search scheme, it is easy to turn POSL scheduling into an optimising algorithm by using it in a branch-and-bound search. We implemented two variants: one optimises makespan, while the other aims to optimise schedule robustness, as measured by flexibility and fluidity.

In table 1, we show results for the first solution found, which is unaffected by the optimisation objective and thus the same for both, and for each of the two variants the best solution found within the 60 second time limit.

Flexibility is simply the size of the precedence relation, which is straightforward to put a bound on. Fluidity, however, is a more complex measure to optimise, particularly as we do not perform complete temporal propagation. Therefore, we use a surrogate objective, which we believe correlates reasonably well with fluidity. This is given by

$$\sum_{i=1,\ldots,n} \frac{\mathrm{lst}(i) - \mathrm{est}(i)}{|\{j \mid i \prec j\}|}$$

i.e., summing the slack of each activity divided by the number of activities ordered after it. The branch-and-bound search works on both robustness measures at the same time, i.e., when a solution is found, constraints are added to ensure that the next solution improves both flexibility and our surrogate fluidity measure.

The makespan-optimising search is quite good: 48% of schedules found are optimal w.r.t. makespan, and on average makespan is only 1.5% above best known. The robustness-optimising search is not as good, yielding only a small, though steady, increase in flexibility and fluidity. (Note, however, that this is at the cost of only a very small increase in makespan.) The likely reason for this is that we do not implement any effective propagation of the bounding constraints, which thus serve mainly to cut off search branches when the bounds are exceeded, and not to direct the search towards better solutions.

Conclusions

Previous work has indicated that constructing partial-order schedules by a two-stage method, in which a fixed-time schedule is first found and a partial order then lifted from it, is far more efficient than constructing them directly, by a least-commitment partial-order scheduling algorithm (Policella et al. 2004; 2007). We have demonstrated that searching in the space of partial-order schedules is not as hopeless as previous results may have suggested. The key innovation is a change of problem formulation, from detecting and eliminating potential resource conflicts to ensuring activities' resource requirements are met by establishing explicit support links. The resulting POSL scheduling algorithm convincingly outperforms EBA, the least-commitment algorithm introduced by Policella et al. (2004), but does not yet reach the performance of the two-stage method. However, these results are preliminary, and there is scope for developing the POSL method further. In particular, improving propagation of bounds on various measures of schedule robustness, such as flexibility and fluidity, has the potential to effectively direct the search to the most robust partial-order schedules across the entire space of solutions.

References

Kolisch, R., and Padman, R. 2001. An integrated survey of project scheduling. *OMEGA International Journal of Management Science* 29(3):249–272.

Laborie, P. 2003. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence* 143:151–188.

McAllester, D., and Rosenblitt, D. 1991. Systematic nonlinear planning. In *Proc. 9th National Conference on Artificial Intelligence*.

Muscettola, N. 2002. Computing the envelope for stepwise-constant resource allocations. In *Principles and Practice of Constraint Programming (CP'02)*, 139–154.

Policella, N.; Smith, S.; Cesta, A.; and Oddi, A. 2004. Generating robust schedules through temporal flexibility. In *Proc. 14th International Conference on Automated Planning & Scheduling (ICAPS'04)*, 209–218.

Policella, N.; Cesta, A.; Oddi, A.; and Smith, S. 2007. From precedence constraint posting to partial order schedules. *AI Communications* 20(3):163–180.

Policella, N.; Cesta, A.; Oddi, A.; and Smith, S. 2009. Solve-and-robustify. *Journal of Scheduling* 12:299–314.

Vidal, V., and Geffner, H. 2004. Branching and pruning: An optimal temporal POCL planner based on constraint programming. In *Proc. 19th National Conference on Artificial Intelligence (AAAI'04)*, 570–577.