Optimal Parallel Schedules

by

Michael Nidd

Department of Computer Science
University of Waterloo

Waterloo, Ontario, Canada, 1994

Prepared for professor Qiang Yang, in fulfillment of the requirements for CS 686 (Introduction to Artificial Intelligence).

Abstract

AI planning research is weak in the area of distributed applications. This paper addresses the problem of generating plans for an initially unspecified number of identical actors. The solution presented decomposes linear (single actor) plans into dependency graphs. Two algorithms which use these graphs in the construction of parallel (multiple actor) plans are presented, discussed, and compared.

Contents

1 Introduction **1**

2 The Solution **2**

2.1 Step 1: Computing the dependencies of a plan 3

2.2 Step 2: Removing Redundant Links 5

2.3 Step 3: Assigning actors 7

2.3.1 A naïve approach 7

2.3.2 A recursive approach 8

2.3.3 Comparing Solutions 10

3 Problems **13**

4 Conclusions **14**

A Subgoal solution to part 3 **15**

Bibliography **19**

1 Introduction

Over the past decade or two, the AI community has committed uncountable hours to the problem of generating plans. While many methods have been developed to utilize a piece of hardware in accomplishing a goal, the price of the hardware itself is generally lower now than it used to be; this redefines what solutions are reasonable. Designers must now ask themselves "If one robot can perform a task in ten minutes, how long would it take two or three?" There is no reason to regenerate the plan, in most cases the steps which are required will not have changed; but how should the new hardware be assigned to these steps.

The problem of assigning actors to the individual actions of a plan has not been widely addressed, yet all areas of the computer industry are moving rapidly towards distributed platforms. This paper will offer a technique by which a sequential (one actor plan) may be converted to a parallel plan which can take advantage of multiple identical actors.

While this technique was developed in isolation, the dependency diagrams (§2.1) bear a striking similarity to the relations discussed in [SR86]. In the same paper, Sandewall and Rönnquist suggest that similar techniques would be useful in "understanding narrative texts where several things are going on at the same time," since that also involves different "simultaneous" threads which may affect one another.

2 The Solution

The solution I suggest is to change the representation of the input plan into an ordered list of actions. Each action should include which resources and objects are involved. For example:
from move(b, s, d, E) (Move block b from stack s to stack d given environment E.)
to move(r, b, s, d) (Resources listed in r are involved in moving a block b from stack s to stack d; environment is implicit in where the "move" appears in the ordered list.)
With reference to the blocks world example, a resource could be a location (stack); while an object would be a block. The arm with which the block is to be moved is the "actor"; this is what is to be scheduled. This conversion would also convert any notations which use blocks (instead of stacks) as destinations, as in an "on-top-of" operator, into using stacks; this simplifies later steps.

The scheduling is done in the following three steps:

Step 1: For each operation in the list, find the most recent use of each resource, and each object required for that operation. From this, generate D a dependency list of which operations must complete before a particular operation may start.

Step 2: Remove redundant dependencies. For example (use $(2 \leftarrow 1)$ to indicate operation 2 depends on 1 ,i.e. 1 must complete before 2 is started) if $(2 \leftarrow 1)$, $(3 \leftarrow 2)$, and $(3 \leftarrow 1)$; $(3 \leftarrow 1)$ is redundant.

Step 3: Allocate actors to operations such that expected completion time is minimized.

2.1 Step 1: Computing the dependencies of a plan

This and all later steps will use the following values:

m: number of actors

n: number of actions in the plan

o: number of objects

r: number of resources

Input: $\Sigma =< \sigma_1, \sigma_2, ...\sigma_n >$

```
plan(Σ) {
    for i=0..n
        for each x = a resource or object used by σ_i
            for j=i..0
                if (σ_j uses x) add link σ_i ← σ_j, and exit j-loop
}   }   }   }
```

This algorithm adds a dependency edge linking each node to the node which most recently used each required resource or object. Notice, that double edges are possible. This algorithm assumes that an action is possible iff all objects and resources used in that action are free (i.e. there are no hidden restrictions on actions). This is a valid assumption because that is the definition of a resource. The only definition of conflicting actions is two actions which require a common resource or object. So, if two actions are conflicting for some hitherto unspecified reason, generate a new resource which is required by each of them.

In calculating the complexity of this algorithm, and the others presented, consider the number of objects and resources constant. Since $1 + 2 + ... + n = \frac{n(n+1)}{2}$, that gives this step a complexity of $O(n^2)$.

In figure 1 the dependencies generated for a given plan are shown with arrows.

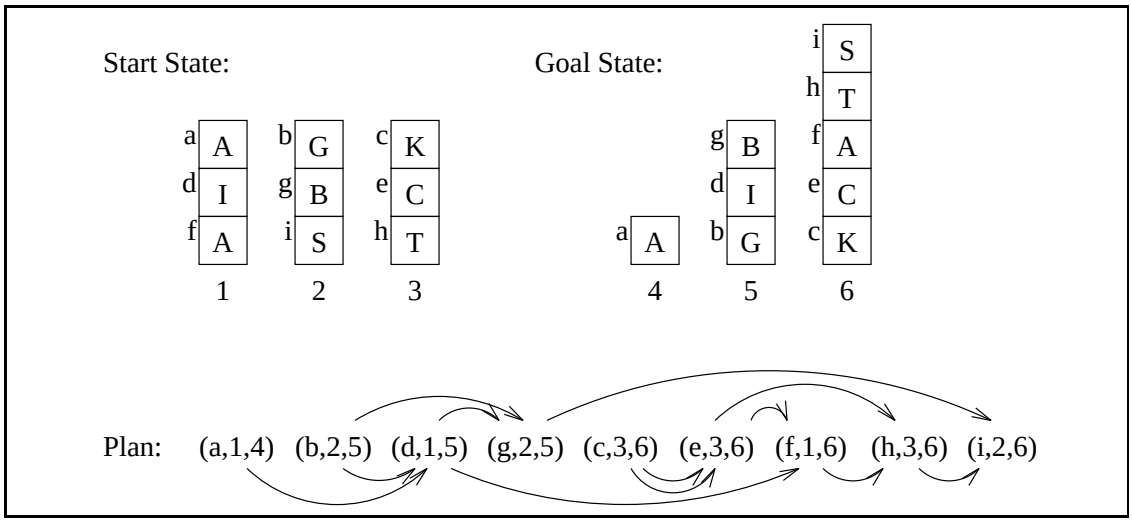

Figure 1: (Step 1) Dependencies are generated for the given plan.

2.2 Step 2: Removing Redundant Links

```
Initially set all node.visited_from = nil
check(node) {
    for i=0..(node.outdegree-1) {
        if (node.next[i].visited_from ≠ nil)
            delete link (node.next[i] ← node.next[i].visited_from)
        node.next[i].visited_from = node
    }
    for i=0..(node.outdegree-1)
        check(node.next[i])
    for i=0..(node.outdegree-1)
        node.next[i].visited_from = nil
}
```

This algorithm does a depth-first-style traversal of the dependency graph. If an action x is reached which is immediately reachable from a node y in its own path, the dependency $x \leftarrow y$ is redundant (y must have already completed for x to be reached by the current path, so the dependency of x on y is implicit in this path, and an explicit dependency is not required).

Notice the special case where $x = y$ (a case corresponding to duplicate dependencies). One of the links will be deleted, eliminating duplicates (or triplicates, etc.).

In this algorithm each node may be checked once by each incoming dependency node (maximum m times), and each check is order n; so this algorithm's complexity

is $O(mn)$.

In figure 2 the redundant dependencies have been removed from the sample dependency graph generated in step 1.

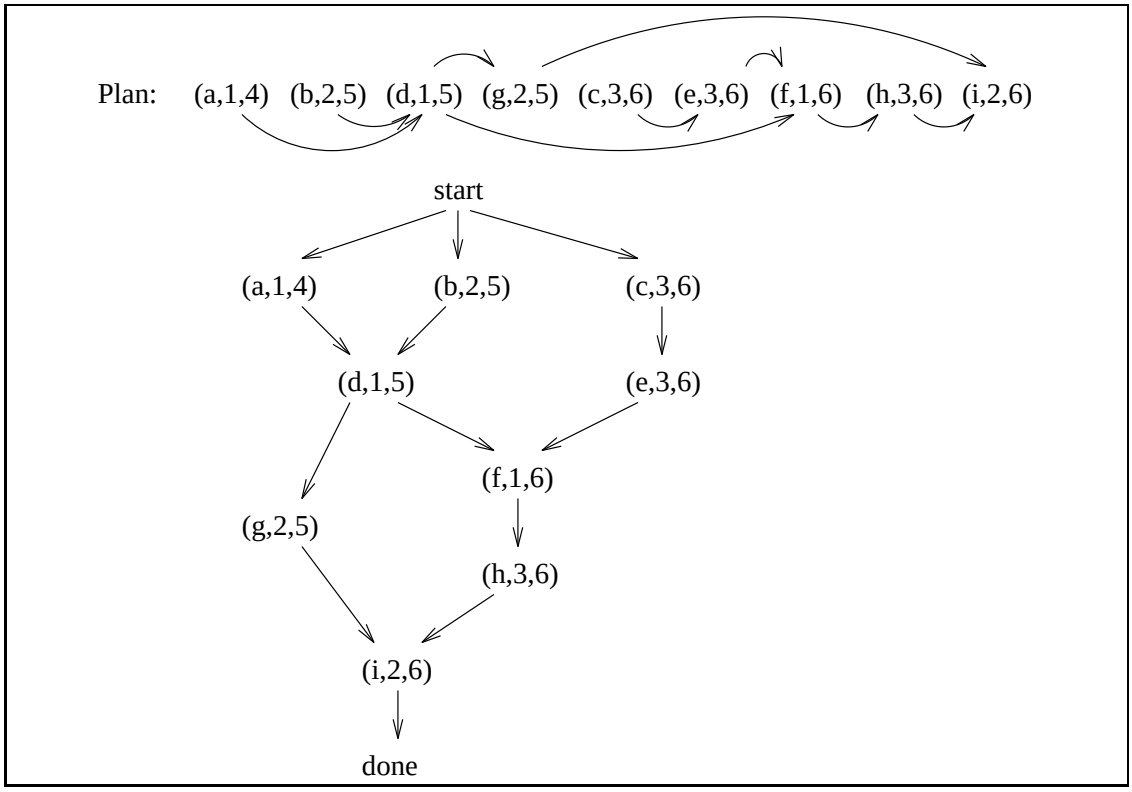

Figure 2: (Step 2) Redundant dependencies are removed (and the graph is redrawn in a more readable form).

2.3 Step 3: Assigning actors

In the implementation of step three, several techniques are possible; this section presents a special purpose solution to the problem. First, a description of the technique is given; then a pseudocode implementation is provided; then a comparison is prevented which contrasts the technique with a more general purpose AI technique, heuristic search.

2.3.1 A naïve approach

A simplistic solution to this problem is a greedy algorithm. The following example uses two queues: a node queue (nq), and an actor queue (aq).

```
schedule_1(actor_queue, root_node) {
    push(nq, root_node, 0)
    insert actor_queue into aq all with time 0
    do while (not_empty(nq)) {
        (node, ntime) ← pop(nq)
        (actor, rtime) ← pop(aq)
        time ← max(ntime, rtime)
        node.actor ← actor
        complete_time ← time + node.weight
        for each (cnode ∈ node.next)
            push(nq, cnode, complete_time)
        push(aq, actor, complete_time)
}   }
```

Pushes are done with a node and a time. They are done such that the queue is kept in time order (ordered insertion to a sorted list can use binary search, so is order $\log(n)$). In this algorithm, each node will be pushed once (order $\log(n)$) and popped once. For each node pushed and popped, one actor will be pushed and popped once (order $\log(m)$). Thus, the algorithm is order $n\log(n)\log(m)$.

2.3.2 A recursive approach

This algorithm divides the allocation problem into subgoals. Every time it finds a choice of allocating actors, it searches recursively for the optimal assignment of actors to each dependency subgraph, decides on a partitioning of the actors available, and allows a recursive call to each subgraph (with the actor list allocated) to let each subgraph take care of itself. Each instantiation is responsible for it's root node, and all children of that node whose depth [1] is greater than that of the root node. It is also responsible for nodes of equal depth, so long as its chain of control is the leftmost back-link from each node. This means that a given instantiation will follow a dependency chain until more incoming links have been seen than outgoing links, and if two siblings trace to a common node, the one to follow that link will be chosen consistently (based on the ordering of the back-link list at each node).

For each node, the following possibilities are considered:

node outdegree = 0

There are no links to follow, so return.

node outdegree = 1

There is only one next node to go to next. Go there.

[1] depth = (parent depth) + (# of children of parent) - (# of incoming links to child). Where the root is depth 0.

1 < node outdegree ≤ number of actors

Assume that the optimal allocation involves at least one different actor for each next node (rather than, for example, giving all but one to the same node, and sharing the last one among the others). This is not always optimal (see section 3), but it seems to be a reasonable assumption.

Allocate an actor to each next node; then allocate the rest, one at a time, to whichever trace returns the longest expected weight.

number of actors = 1, but node outdegree > 1

There is not much choice here; allocate the one available actor to all outgoing dependencies.

1 < number of actors < node outdegree

Create one "virtual actor" for each process, and make a recursive call to schedule that one actor. Find the two shortest schedules, and declare their actors equivalent. Keep combining actors in this manner until the number of virtual actors equals the number of actors, then do a direct substitution.

In each case, the weight of subtrees is calculated, and added to the weight of the current trace. If the next node to which the trace would like to go is too shallow (has enough incoming dependencies to cause depth to be less than zero), or if the current node is not first in the list of incoming dependencies to the next node (indicating that it is somebody else's problem), the function returns. Otherwise, it loops back for the next node.

Actual pseudocode for this algorithm is given in appendix A.

The recursive nature of this algorithm causes it to have a high order of complexity. The most expensive part of an expansion is for node.outdegree > num_actors.

This is order m^2 (not including recursive calls), since the number of links out of a node (node.outdegree) is never more than m. The most recursive calls which could be done by an expansion is m (actually max(num_actors, node.outdegree) at each node) At any particular node, an expansion might be initiated by any of the nodes above it in the dependency graph; that is on the order of $\frac{n}{3}$, which is order n. Over all n nodes, that gives this algorithm a complexity order of m^3n^2.

2.3.3 Comparing Solutions

"Why," you might ask, "would I ever use the second solution?" This would be a perceptive question. Recall that the complexity of method one (the traditional computer science approach) is $O(n\log(n)\log(m))$, while the subgoal search is $O(n^2m^3)$. The first method will give the same results as just executing the plan, and having actions ask for actors whenever they become executable. It seems logical that, knowing the entire plan, a holistic method should be able to find a better allocation than the first-come-first-served technique. It is the contention of this paper that algorithm two demonstrates this, despite the worst-case complexity.

The reason for the second algorithm's higher complexity is that a particular dependency chain might have several different allocations attempted, so individual nodes will be expanded multiple times. In practice it is unlikely that this will be a problem, unless the number of available actors is large. Multiple allocations are tried if and only if there are more actors available for use than there are outgoing dependencies from a node. After the first few nodes, it seems likely that the actors will be spread sufficiently thinly for this not to be a problem. So, if we consider the number of nodes which will try multiple allocations to be constant, the complexity drops to m^3n. This is under the assumption that $m \ll n$ anyway, so under these conditions, the two algorithms now compare: algorithm one = $O(n\log(n))$,

algorithm two = $O(n)$. Both of these will be overshadowed by step 1 which is $O(n^2)$.

Furthermore, since the second algorithm uses a subgoal technique, it is more resistant to plan changes. If a new action is added, it can be rescheduled using the actors allocated to the parent nodes. This way only the subtree from that action to the last action to depend on the new one need be rescheduled. Using the first method, the schedule cannot be modified, only remade.

Consider the example from sections 2.1 and 2.2. Imagine that every event has an expected duration of 1 time unit, except c, which is expected to take 6 time units (because this block is very heavy). Figure 3 shows the actor allocations [2] which will result from each of the two methods. Notice that the first algorithm allocated the same actor to the heavy block and another, while the second algorithm was able to foresee the problem and dedicate an actor to that block only. This reflects the fundamental difference between the algorithms: the first allocates whatever it can, and later corrects for earlier mistakes; while the second tries to avoid mistakes.

[2] Allocations from algorithm two with multiple actors identified for the same action mean that either will do, so choices may be made arbitrarily, or by any external means which fits the particular problem.

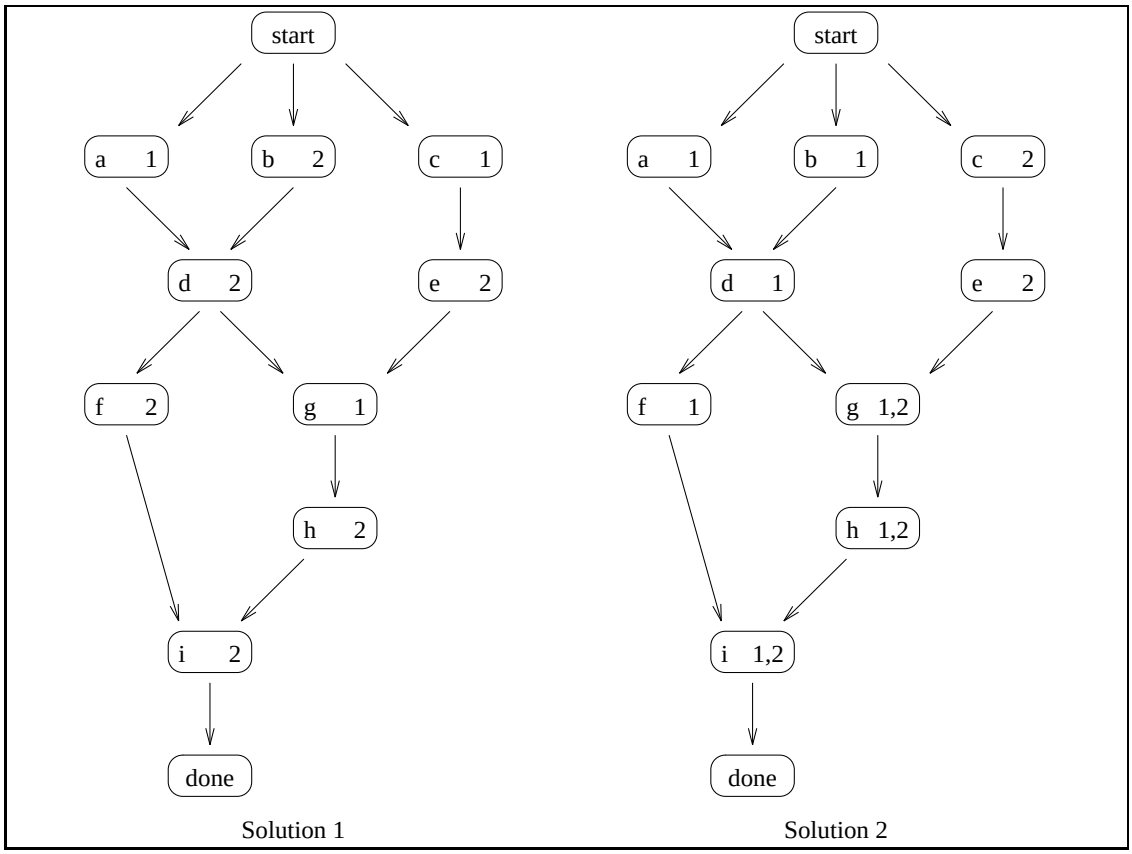

Figure 3: (Step 3) Resources are allocated by one of two techniques (block c is heavy)

3 Problems

One problem with the actor allocation (step 3) is the assumption that if a node has enough actors for all outgoing links, each link should get at least one. An example of this is given in figure 4, where an optimal allocation of two actors is $r_1 = \{1,3,5,7,8\}$ and $r_2 = \{2,4,6\}$. What the current algorithm will actually generate is an allocation equivalent to $r_1 = \{1,3,4,5,6,7,8\}$ and $r_2 = \{2\}$ which is less than optimal. The most obvious solution to this problem is to perform an exhaustive search of allocations at each node, but this would almost certainly increase the time complexity of the algorithm.

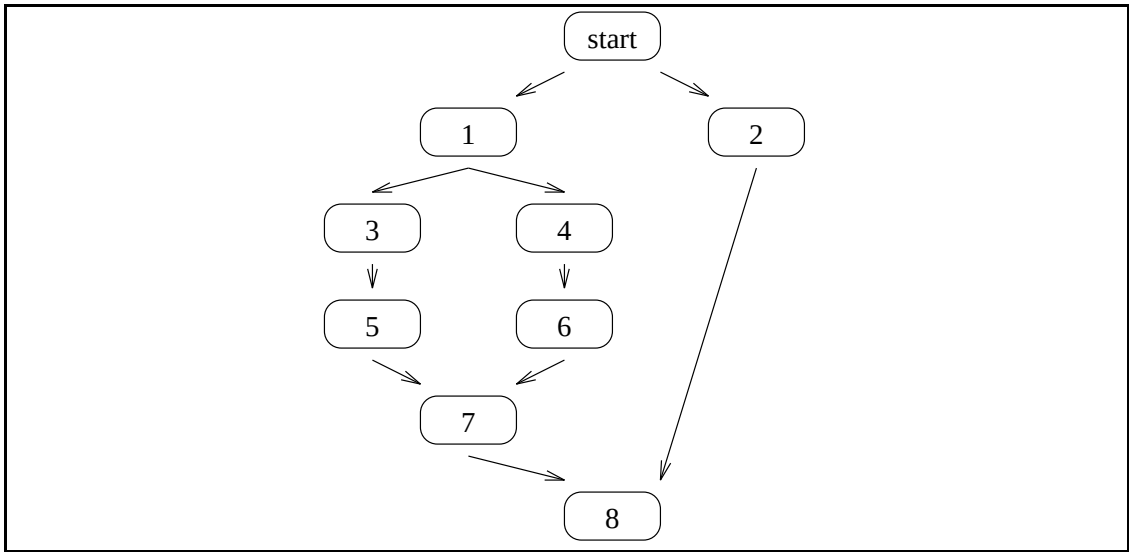

Figure 4: A pathological situation for two-actor allocation

Another problem is that of deciding which actor to use when the algorithm does not express a preference (i.e. a set of valid actors is listed for a node). There will often be at least one bad choice of actors. This might be eliminated by a two-pass algorithm, but at the time of this writing, I have no good solution.

4 Conclusions

This is a good solution to the problem, although it is not without drawbacks. The use of an approximate algorithm for step 3 allows poor allocations to be made, but this algorithm has speed advantages for the average case, and also for efficient recovery from small changes in the plan. In short, it is similar to most AI solutions to problems; it's worst case performance is not as good as a traditional solution, and it is sometimes wrong, but most of the time it is reasonably fast, and reasonably correct.

In future work I intend to incorporate the language proposed by Allen in [All84] to describe the dependency diagrams. This would help to integrate developments with the results of Rutten [Rut91], who is also working with temporal considerations in planning (and uses Allen's notation). Furthermore, a better algorithm for part three almost certainly exists (with respect to both complexity and the frequency with which the optimal allocation is reached). I intend to work towards this goal, starting by addressing the problems identified in section 3.

Further questions involve extentions to the problem. Suppose the user wants to specify which actor is used for some particular action. The algorithm would currently have difficulty dealing with this restriction. The algorithm may, however, generalize to multiple types of actors; or to similar actors which have different qualities (such as speed or cost). There seems to be a great deal of room for research in this area.

A Subgoal solution to part 3

Initial Call: schedule_2(root, num_actors, actor_list, 0)

```
schedule_2(from_node, num_actors, actor_list, initial_depth) {
    ret.weight = 0
    ret.hungry = FALSE
    ret.deepest = nil
    node = from_node
    depth = initial_depth
    while(depth > 0) {
        node.actor = actor_list
        if node.outdegree = 0
            done; exit loop
        else if node.outdegree = 1
            next = node.next[0]
            ret.weight = ret.weight + node.weight
        else if node.outdegree ≤ num_actors {
            allocate rv = array of ret[node.outdegree]
            allocate rl = array of actor_list[node.outdegree]
            next = nil
            depth_change = 0
            for i = 0..(node.outdegree-1) {
                rv[i].weight = ∞
                rv[i].hungry = TRUE
            }
```

```
        for j = 0..num_actors {
            w=-1; i=-1;
            for(x=0..(node.outdegree-1)) {
                if (rv[x].weight > w) and (rv[x].hungry = TRUE)
                    w=rv[x].weight; i=x
            }   }
            if (i ≥ 0) append(rl[i], actor_list[j])
            else append(rl[0], actor_list[j..(num_actors - 1)]; exit loop
            rv[i] = schedule_2(node.next[i], lengthof(rl[i]), rl[i], node.outdegree)
            if (rv[i].final_node ≠ nil)
                next = rv[i].final_node
                depth_change = rv[i].final_depth
        }   }
        w = rv[0].weight;
        for(x=1..(node.outdegree-1))
            if (rv[x].weight > w) w=rv[x].weight
        ret.weight = ret.weight + node.weight + w
        depth = depth + depth_change
    }
    else if num_actors = 1
        ret.hungry = TRUE
        create rv = ret
        ret.weight = ret.weight + node.weight
        for i=0..(node_outdegree - 1)
            rv = schedule_2(node.next[i], 1, actor_list[0])
            ret.weight = ret.weight + rv.weight
```

```
            if (rv[i].final_node ≠ nil)
                next = rv[i].final_node
                depth_change = rv[i].final_depth
        }   }
        depth = depth + depth_change
    }
    else { /* node.outdegree > num_actors > 1 */
        ret.hungry = TRUE
        create tr = array of actor_list[node.outdegree] (temporary actor identifiers)
        allocate rv = array of ret[node.outdegree]
        for i = 0..(node.outdegree-1)
            rv[i] = schedule_2(node.next[i], 1, tr[i])
        for j = 0..(node.outdegree - num_actors) {
            w1=w2=∞
            for(x=0..(node.outdegree-1)) {
                if(rv[x].weight ≤ w1) {
                    w2 = w1
                    i2 = i1
                    w1 = rv[x].weight
                    i1 = x
            }   }
            replace all occurrences of tr[i1] with tr[i2]
            rv[i2].weight = w1 + w2
            tr[i1] = nil
            rv[i1].weight = 0
        }
```

```
            /* Now: num_actors = number of non_nil temporary actor identifiers */
            i = 0;
            for j = 0..(node.outdegree-1) {
                if (tr[j] ≠ nil)
                    replace all occurrences of tr[j] with actor_list[i]
                    i = i+1
            }   }
            w = 0
            for (i=0..(num_actors - 1))
                if (rv[i].weight > w) w = rv[i].weight
            ret.weight = ret.weight + node.weight + w
        }
        if (next.backlink[0] ≠ node) exit loop
        node = next;
        depth = depth - (node.indegree - 1)
        actor_list = Union(node.backlink[all].actor_list)
    }
    return ret
}
```

Bibliography

[All84] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2), July 1984.

[Rut91] Eric Rutten. A temporal representation for imperatively structured plans of actions. In *EPIA 91, fifth Portuguese Conference on Artificial Intelligence*, volume 541 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1991.

[SR86] Erik Sandewall and Ralph Rönnquist. A representation of action structures. In *aaai-86, Proceedings of the fifth national conference on artificial intelligence*, Los Altos, California, United States of America, August 1986. The American association for artificial intelligence in cooperation with the University of Pennsylvania, Morgan Kaufmann Publishers, Inc.