Optimal Task Graph Scheduling with Binary Decision Diagrams

Anders Rune Jensen, Lau Bech Lauritzen, and Ole Laursen

Department of Computer Science,
Aalborg University, Denmark,
{arj,lau,olau}@cs.aau.dk,
May 2004.

Abstract. In this paper, we represent the task graph scheduling problem with uniform processors and arbitrary task execution times with BDDs and devise a breadth-first search and an A*-based algorithm for finding optimal schedules. The representation is chosen to minimise the state BDD sizes, and the transition relations are partitioned to reduce the computation time and allow optimisations based on analysis of the task graphs. The empirical results show that guiding the search with an A* approach is difficult in practice, but that the breadth-first search works very well with graphs with many dependencies.

1 Introduction

The task graph scheduling problem considered in this paper is conceptually simple: given a directed acyclic graph of dependencies between tasks with arbirary execution times and an arbitrary number of available homogenous processors, find a schedule that minimises the total execution time. The problem and variants of it appear in many practical multiprocessor settings and also in industrial manufacturing. But since the number of possible schedules in the worst case grows as the factorial of the number of tasks and finding an optimal one among them is NP-hard [11], there is no straight-forward way of solving it.

Task graph scheduling has been extensively studied [9], however, and there are a wide range of approximative algorithms available that find good schedules within seconds even for problems with 500 tasks.

More recently the engines in verification tools such as SPIN and Uppaal have been used to solve scheduling problems optimally [1, 13, 14]. This is possible because task graph scheduling can be reformulated as a state space exploration problem, and the verification engines are capable of efficient exhaustive searching in huge state spaces.

An interesting question is that if the adaptation of the scheduling problem to these verification engines shows some promise, then how far is it possible to get by instead adapting the engine techniques to fit the scheduling problem. One widely used technique is reduced ordered binary decision diagrams (BDDs) [3]. BDDs can represent large state sets compactly and have been used succesfully for

verification of both hardware [4, 5] and software designs. Hence, in this paper we represent the task graph scheduling problem as a BDD state space exploration problem and formulate a breadth-first and an A* search algorithm for finding optimal schedules. The BDD representation must be selected with great care to be tractable in practice. Empirically, the resulting breadth-first search algorithm is very quick at solving task graphs with many dependencies.

The paper is organised as follows. First we formalise the task graph scheduling problem and briefly describe BDD-based state space exploration. We then explain the BDD algorithms that have been developed and present the results and some analysis of applying them on a standard task graph set [10].

2 Task graph scheduling

The task graph scheduling problem consists of a set of tasks $G = \{t_1, t_2, \ldots, t_n\}$ that are to be scheduled on M uniform processors. Each task $t_i \in G$ is associated with an integer execution time $\mathfrak{T}(t_i)$. Furthermore, the tasks are ordered according to a precedence relation $\rightarrow \subseteq G \times G$ where $t_j \rightarrow t_i$ if and only if task t_i depends on task t_j and cannot start before t_j has finished. Let $\mathcal{D}(t_i)$ denote the set of immediate dependencies for t_i, i.e. $\mathcal{D}(t_i) = \{t_j | t_j \rightarrow t_i\}$, and let $\mathcal{D}_c(t_i)$ be the complete set of dependencies for t_i, i.e. $\mathcal{D}_c(t_i) = \{t_j | \exists t_l, \ldots, t_k : t_j \rightarrow t_l \rightarrow \cdots \rightarrow t_k \rightarrow t_i\}$ where $k \geq 0$. Note that dependency cycles are not allowed, i.e. $\forall t_i : t_i \notin \mathcal{D}_c(t_i)$.

Hence, the tasks and the precedence relation corresponds to a directed acyclic graph with G as the set of nodes and $\rightarrow$ as the set of edges, see Figure 1. Let a path $t_l \rightarrow \cdots \rightarrow t_k$ in the task graph be weighted with the execution time associated with each node so that the length of the path is $\sum_{i=l}^{k} \mathfrak{T}(t_i)$. The critical path $CP(t_i)$ to a task t_i is the longest path from any task to t_i. The length of the critical path can be found by a simple polynomial time graph algorithm [6].

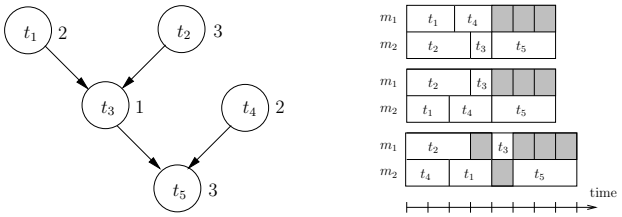

Fig. 1. An example task graph to the left where $\mathfrak{T}(t_5) = 3$, $\mathcal{D}(t_5) = \{3, 4\}$, $\mathcal{D}_c(t_5) = \{1, 2, 3, 4\}$ and $CP(t_5) = 3 + 1 + 3 = 7$. To the right three feasible schedules for the task graph with $M = 2$. The grey slots are free time slots.

An important property of the tasks graphs is the fraction of the number of complete dependencies in the graph out of the number of possible dependencies.

We denote this the dependency fraction d. Since the graphs are directed and acyclic, a fully connected graph will have $(|G|-1)+(|G|-2)+\cdots+1 = |G|(|G|-1)/2$ dependencies. Hence

$$d = \frac{\sum_i \mathcal{D}_c(t_i)}{|G|(|G|-1)/2}$$

A schedule is an assignment of start time $\mathfrak{s}(t_i)$ to each task $t_i \in G$. A feasible schedule is a schedule where at most M tasks run concurrently at any point in time, i.e. for any set $R \subseteq G$ where $|R| > M$ it holds that $\bigcap_{t_j \in R}[\mathfrak{s}(t_j); \mathfrak{s}(t_j) + \mathfrak{T}(t_j)] = \emptyset$, and for each task t_i the dependencies $t_j \in \mathcal{D}(t_i)$ will have finished before t_i is started, i.e. $\forall t_j \in \mathcal{D}(t_i)\colon \mathfrak{s}(t_j)+\mathfrak{T}(t_j) \leq \mathfrak{s}(t_i)$. The makespan or length of a schedule is the time it takes before all tasks have finished, $\max_i(\mathfrak{s}(t_i)+\mathfrak{T}(t_i))$. An optimal schedule is a feasible schedule with the smallest possible makespan.

Figure 1 shows two optimal schedules together with a non-optimal schedule. In the non-optimal schedule t_3 has to wait an extra time slot for its dependencies to finish, thereby making the schedule one time step longer.

3 State space exploration with BDDs

A reduced ordered binary decision diagram is a rooted directed acyclic graph with internal nodes with two out-going edges and two leaf nodes labeled 1 and 0 (true and false) [2, 3, 8]. Each internal node corresponds to a Boolean variable b_i and the two out-going edges, the high branch and the low branch, correspond to the value of the variable being true or false, respectively. The nodes are arranged so that a trace of any path from the root of the BDD to a leaf node encounters variables in the same order.

Furthermore, the BDDs are reduced so that no two nodes correspond to the same variable and have the same low descendants and high descendants, and no single node has identical low and high descendant. This means that each BDD canonically represents a particular Boolean function of m variables, $f\colon \mathbb{B}^m \to \mathbb{B}$. The function value is found by following a path from the root to one of the leaf nodes, see Figure 2.

A state space exploration problem [8] consist of a state space $\mathbb{S}$, an initial state $s_{init} \in \mathbb{S}$, a set of goal states $S_{goal} \subseteq \mathbb{S}$, and a transition relation $T : \mathbb{S} \times \mathbb{S}$ where $(s, s') \in T$ if and only if the search is allowed to proceed from the state s to the next state s'. The set of successor states $S_{\tau+1}$ to a set of states S_τ is given by $S_{\tau+1} = \{s' | \exists s \in S_\tau\colon (s, s') \in T\}$. We denote this image computation operation by $S_{\tau+1} = \mathcal{I}_T(S_\tau)$ and the reverse process, the preimage computation, by $S_\tau = \mathcal{I}_T^{-1}(S_{\tau+1})$.

If a state s is encoded as a bit vector $\vec{s} = (b_1, b_2, \ldots, b_m)$, a single BDD can represent a set of states $S \subseteq \mathbb{S}$ as the characteristic function $f_S\colon \mathbb{B}^m \to \mathbb{B}$ of S:

$$f_S(b_1, b_2, \ldots, b_m) = \begin{cases} 1 & \text{if } (b_1, b_2, \ldots, b_m) \in S \\ 0 & \text{if } (b_1, b_2, \ldots, b_m) \notin S \end{cases}$$

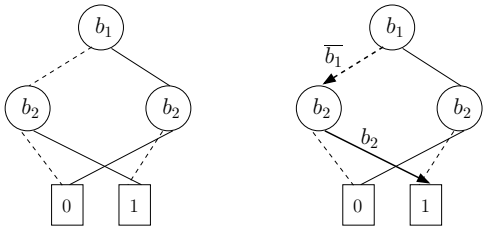

Fig. 2. The function $f(b_1, b_2) = b_1 \oplus b_2$. The solid lines are the high branches, and the dotted lines are the low branches. The function value is found by tracing a path from the root to the true or false leaf node as shown for $f(0, 1) = 1$ to the right.

For example, the set $\{(1, 0), (0, 1)\}$ can be represented as $f(b_1, b_2) = (b_1 \wedge \overline{b_2}) \vee (\overline{b_1} \wedge b_2)$ which reduces to $f(b_1, b_2) = b_1 \oplus b_2$ and is hence represented by the BDD in Figure 2. With the BDD representation it is easy to find the members of the set: simply find all paths that lead to the leaf node 1.

The point in using BDDs for representing state sets is that BDDs, in practice, are very compact. For instance, the peak BDD size of about 30 000 nodes for one of the test setups of this paper yielded sets with more than 1 000 000 states.

To exploit this efficiency the transition relation must also by represented by a BDD. This is possible because the relation is also a set with a characteristic function. For example, the transition relation given by the set of pairs $\{(\langle 0, 1\rangle, \langle 1, 1\rangle); (\langle 1, 1\rangle, \langle 1, 0\rangle)\}$ can be described as

$$T(b_1, b_2, b_1', b_2') = (\overline{b_1} \wedge b_2 \wedge b_1' \wedge b_2') \vee (b_1 \wedge b_2 \wedge b_1' \wedge \overline{b_2'})$$

Note that this $T(\vec{s}, \vec{s}')$ is a function of four variables, two variables for the current state and two primed variables for the next state. In the following we append functional parantheses when we talking about the BDD representation of a set.

In BDD terms, the image can be computed as an existential quantification of the variables in $\vec{s}$ over the conjugation of $S_\tau(\vec{s})$ with $T(\vec{s}, \vec{s}')$ where finally the resulting variables in $\vec{s}'$ are renamed (denoted by $[\vec{s}/\vec{s}']$) to the corresponding values in $\vec{s}$:

$$S_{\tau+1}(\vec{s}) = \left(\exists \vec{s}(S_\tau(\vec{s}) \wedge T(\vec{s}, \vec{s}'))\right)[\vec{s}/\vec{s}']$$

In implementations a special BDD function is used to compute the conjugation and existential quantification as one operation. The simplest BDD-based search algorithms construct an initial state set S_1 and a transition relation T, and iteratively compute the images $S_2, S_3, \ldots$ until a state set S_n with a goal state is reached.

The preimage is computed as:

$$S_\tau(\vec{s}) = \left(\exists \vec{s}'(S_{\tau+1}(\vec{s}') \wedge T(\vec{s}, \vec{s}'))\right)[\vec{s}'/\vec{s}]$$

4 A breadth-first BDD algorithm for scheduling

In order to use BDDs to solve the task graph scheduling problem, it must be reformulated as a state space exploration problem. We do this by building the schedules gradually starting from a state where no tasks are started and going forward one time step at a time assigning task start times, so that in a certain state some tasks have not started, some are running and some have finished. For example, the state that corresponds to time step one for the bottom-most schedule in Figure 1 is the state where t_2 and t_4 have started and run for one time step, and the successor state to this is the state where t_4 has finished, t_2 has run for two time steps and the rest of task have not started yet.

The BDD algorithm described in this section searches these states in a breadth-first manner using three major phases per iteration. The first phase generates the states with possible combinations of tasks that can be started, the second phase run them one step and the last phase stop them if they have finished. The BDD state representation and the construction of the phases will be explained in the following. Section 5 describes an A* algorithm that reuses the state representation and transition relations but searches in a different manner.

4.1 State representation

We represent the states by associating with each task two Boolean variables, $t_{started,i}$ and $t_{finished,i}$, and an integer counter $t_{clock,i}$. When $t_{started,i}$ is true, task t_i is running and $t_{clock,i}$ keeps track of its execution time until the task has finished and $t_{finished,i}$ is set to true. Additionally an integer counter m_{free} is used to keep track of the number of free processors. The counters are encoded in binary with a number of Boolean variables.

Instead of this representation it would be possible to use only a clock counter for each task by encoding the initial state of a task as a special clock value. But with our representation it is possible to omit constraints on the clock variables of a task when it is not running, and at most M tasks will be running at the same time in each state. In practice, avoiding constraining the clock variables gave a significant reduction of the sizes of the state set BDDs. Furthermore, the transition relations are simpler when it is only necessary to check one variable to discover whether a task is started or has finished instead of all the clock variables.

The ordering of the variables is shown in Figure 3 for the graph in Figure 1. Placing the m_{free} variables first was empirically found to give better performance than placing them last, which we believe is caused by the fact that most of our transition relations depends on the value of m_{free} and hence can skip the irrelevant parts of the state set BDDs quickly if the m_{free} variables are first. The tasks are placed in a topologically sorted order so that all dependencies of a task are placed before that task. The primed next state variables are placed directly after their current state counterparts, e.g. $t_{started,i} < t'_{started,i} < t_{finished,i} < t'_{finished,i}$. Dynamic variable ordering [4] where the ordering of the BDD variables

are changed during execution was tried, but only consumed extra computation time without decreasing the state set BDD sizes.

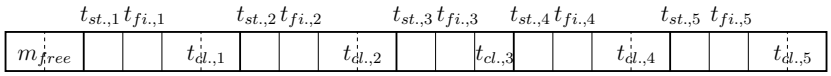

Fig. 3. Variables for the processors free counter are placed first, followed by the state and clock variables for each task.

Another way of representing the states is to have explicit processors, each with a counter, and instead of a clock on each task have an identifier for which processor it is running on. With only a few processors it is likely to use fewer state variables in total. But the transition relations must then be much more complex since they need to match tasks with processors; this makes the image computations much slower.

Even worse, with explicit processors there is the problem that two equivalent states, e.g. task 1 started on the first processor vs. task 1 started on the second, are differently represented which contributes greatly to the state explosion. Our implicit processor representation completely avoids this.

For testing for equality between binary counters and integers we will in the following use a shorthand such as $t_{clock,i} = 0$ for the requirement that the clock for task t_i is zero instead of the complete Boolean expression

$$\bigwedge_{1 \leq k \leq K} \overline{t_{clock_k,i}}, \quad K = \lceil \log_2(\mathfrak{T}(t_i) + 1) \rceil$$

where K is the number of bits needed to represent the clock counter.

Having defined the states and some notation the initial state of the algorithm is simply

$$\vec{s}_{init} = (m_{free} = M) \wedge \bigwedge_i \overline{t_{start,i}} \wedge \overline{t_{finished,i}}$$

where M processors are free and all tasks are not started and not finished. Note that we do not constrain the clock variables. For the graph in Figure 1 the initial state is:

$$\vec{s}_{init} = \overline{m_{free_1}} \wedge m_{free_2} \wedge \overline{t_{started,1}} \wedge \overline{t_{finished,1}} \wedge \overline{t_{started,2}} \wedge \overline{t_{finished,2}} \wedge \cdots$$

The goal states are those states where all tasks have finished, i.e. $S_{goal} = \bigwedge_i t_{finished,i}$.

4.2 Start transition relation

The first phase in each iteration is the image computations of a series of transition relations $T_{b_i}(\vec{s}, \vec{s}')$ to generate the states with all possible combinations of tasks

started, i.e. the state where no tasks are attempted started and the states where task one is started if possible, where task two is started if possible, where task one and two are started if possible, etc.

The reason we are using a number of transition relations for this purpose instead of a single large one is that it is more flexible and makes it possible to avoid a large number of constraints for keeping variables unchanged. Since a large relation would be a disjunction of the individual $T_{b_i}(\vec{s}, \vec{s}')$, the splitted relation is what is referred to as a disjunctive partioning [5].

Each $T_{b_i}(\vec{s}, \vec{s}')$ allows a transition that sets $t_{started,i}$ to true and decrements the number of free processors if task t_i has not been started yet, there is a free processor and the immediate dependencies of task t_i have finished:

$$T_{b_i}(\vec{s}, \vec{s}') = (m_{free} \neq 0) \wedge \overline{t_{started,i}} \wedge \bigwedge_{d \in \mathcal{D}(t_i)} t_{finished,d} \\ \wedge t'_{started,i} \wedge \overline{t'_{finished,i}} \wedge (t'_{clock,i} = 0) \wedge (m'_{free} = m_{free} - 1)$$

As an example, the transition relation for task 3 in the graph in Figure 1 is:

$$T_{b_3}(\vec{s}, \vec{s}') = (m_{free_1} \vee m_{free_2}) \wedge \overline{t_{started,3}} \wedge t_{finished,1} \wedge t_{finished,2} \\ \wedge t'_{started,3} \wedge \overline{t'_{finished,3}} \wedge \overline{t'_{clock,1}} \wedge m'_{free} = m_{free} - 1)$$

We avoid constraints for keeping the variables for the other tasks unchanged by letting the image computation quantify and rename only the variables that can change from the current state to the next state in the above expression. This implies that the BDD nodes for current state variables for the other tasks are not affected and hence stay the same. The omitted constraints reduce the transition relations and thereby the computation time greatly.

The relations $T_{b_i}(\vec{s}, \vec{s}')$ can be used to compute all the possible combinations of started tasks with a fixed point algorithm that at each iteration starts either of the tasks or none. But in practice a more efficient way of generating the combinations is the recurrence:

$$S_{i+1} = \mathcal{I}_{T_{b_i}}(S_i) \cup S_i \ , \ 1 \leq i \leq |G|$$

At the first step of the recurrence the current state set consists of the original state set and the states where task 1 is allowed to start. At the second step the current state set consists of the original state set, the states where task 1 is allowed to start, where task 2 may start and where both task 1 and 2 may start. And so forth.

4.3 Run transition relation

The next phase in each iteration ensures for each task that either the task has not started, it has finished or the clock of the task is incremented by one:

$$T_r(\vec{s}, \vec{s}') = \bigwedge_i \overline{t_{started,i}} \vee t_{finished,i} \vee t'_{clock,i} = t_{clock,i} + 1$$

Again, the image computation only quantifies and renames variables that are supposed to change, here the clock variables, so that we avoid having to set all other variables equal to their successor. For the graph in Figure 1 the transition relation will be:

$$T_r(\vec{s},\vec{s}') = \overline{t_{started,1}} \vee t_{finished,1} \vee t'_{clock,i} = t_{clock,i} + 1 \cdots$$

4.4 End transition relation

The last phase in each iteration ensures that all tasks that have run their entire time are marked as finished. This is done by computing the image where the first task is stopped if possible, then computing the image where the second task is stopped if possible, etc. The relation for task t_i is a disjunction of the two possibilities that either the task is not running and the state should remain unchanged, or the task is running, in which case it has finished if the clock of the task is $\mathfrak{T}(t_i)$ or else remain unchanged:

$$\begin{aligned} T_{e_i}(\vec{s},\vec{s}') = \big(& (\overline{t_{started,i}} \vee t_{finished,i}) \\ & \wedge t'_{started,i} \leftrightarrow t_{started,i} \wedge t'_{finished,i} \leftrightarrow t_{finished,i} \wedge m'_{free} = m_{free}\big) \\ \vee \big(& t_{started,i} \wedge \overline{t_{finished,i}} \\ & \wedge (t_{clock,i} = \mathfrak{T}(t_i) \wedge t'_{started,i} \wedge t'_{finished,i} \wedge m'_{free} = m_{free} + 1 \\ & \vee t_{clock,i} \neq \mathfrak{T}(t_i) \wedge t'_{started,i} \wedge \overline{t'_{finished,i}} \wedge t'_{clock,i} = t_{clock,i} \\ & \wedge m'_{free} = m_{free})\big) \end{aligned}$$

The image computation then only quantifies and renames all the variables of task t_i and m_{free}. Note that there are no requirements for the clock variables if task t_i has not started or has finished.

4.5 Bounding the possible task running times

Because of the dependencies in the task graphs, most tasks cannot start until after a certain point in time, the earliest possible start time $\mathfrak{E}(t_i)$. Furthermore most tasks must have started before a certain other point in time, the latest possible start time $\mathfrak{L}(t_i)$, in order to yield an optimal schedule.

A task t_i cannot possibly start before its complete set of dependencies $\mathcal{D}_c(t_i)$ have run so $\mathfrak{E}(t_i)$ is the length of an optimal schedule for the subgraph with the tasks in $\mathcal{D}_c(t_i)$. This is of course the very problem we are trying to solve, but we can obtain a safe estimate quickly by observing that the start time is bounded from below by the length of the critical path to the task and also by the sum of the execution times of the tasks in $\mathcal{D}_c(t_i)$ divided by the number of processors:

$$\mathfrak{E}(t_i) \geq \max(CP(t_i), \lceil \sum_{d \in \mathcal{D}_c(t_i)} \mathfrak{T}(t_d)/M \rceil)$$

For the example in Figure 1, $CP(t_3) = 3$ and $\sum_{d \in \mathcal{D}_c(t_3)}(2+3)/2 = 3$ which gives $\mathfrak{E}(t_3) = 3$.

Now consider a task t_i and the subgraph G with t_i and all the tasks that depend on t_i (i.e. $\{t_j | t_i \in \mathcal{D}_c(t_j)\}$). Task t_i must have started and run to completion some time before the schedule is complete – else the other tasks in G would not have time to run. More precisely, t_i must have finished in time for the longest path from t_i to some leaf node in G to execute. Hence, $\mathfrak{L}(t_i)$ is the length of the critical path in G subtracted from the length O of an optimal schedule, $\mathfrak{L}(t_i) = O - CP(G)$. For the example in Figure 1, $G_{t_3} = \{t_4, t_5\}$, $CP(G_{t_3}) = 3$, $O = 3 + 1 + 3 = 7$ resulting in $\mathfrak{L}(t_3) = 6 - 3 = 3$.

O is of course not known, but an overestimate can be computed with a fast heuristic algorithm that generates good schedules. One such simple algorithm is a slight modification of this breadth-first search algorithm that at each step only picks the schedules that starts the most tasks.

At each step in the algorithm, the bounds $\mathfrak{E}(t_i)$ and $\mathfrak{L}(t_i)$ can be used to avoid computing images over transition relations that cannot possibly start or end a task at that step – in practice, this was found to save at least 50–60% of the start and end image computations. Furthermore, $\mathfrak{L}(t_i)$ can be used to prune the state sets from states where a task that would certainly have been completed in an optimal schedule is not finished yet – refer to Section 4.6 for details. This was also found to speed up the algorithm (by more than 10%).

A further possible optimisation is to keep track of already reached states and prune the current state set from these at each iteration. This will reduce the number of states, but empirically it turns out that the BDDs for the state sets actually end up larger so that the algorithm becomes slower (about twice the running time). An explanation for this could be that the extra states can make variables in some paths irrelevant which makes it possible to reduce the BDDs more.

4.6 The complete algorithm

The complete algorithm appears in Listing 1.1. Line 3–5 put the current state set through the start transition relations of the tasks that are able to start at this point, line 6 advances the clocks one step with the run transition relation and line 7–9 finishes tasks with the end transition relations of those tasks that can possibly finish. Line 10–11 cuts of states with unfinished tasks that certainly would have had finished in an optimal schedule.

When the algorithm terminates, the length of an optimal schedule is simply the number of iterations in the loop. To actually obtain an optimal schedule the algorithm must be modified to save the current state S at each iteration. Denote these by $S_1, \ldots, S_n$ and let for the sake of the explanation T denote a combined abstract transition relation that encompasses both the start, run and end transition relations. Then a sequence of states $s_1, \ldots, s_n$ leading to an optimal schedule, i.e. $s_1 = s_{init}$, $s_n \in S_{goal}$ and $(s_\tau, s_{\tau+1}) \in T$ for $1 \leq \tau < n$, can be obtained by setting s_n to a solution in S_n and picking $s_{n-1}, s_{n-2}, \ldots, s_1$ with the recurrence $s_\tau \in \mathcal{I}_T^{-1}(s_{\tau+1}) \cap S_\tau$. With the states $s_1, \ldots, s_n$, a schedule can be constructed by examining the states in turn to discover when each task is started (by conjugation with the constraint $t_{started,i}$).

```
1  S = {s_init}
2  τ = 0
3  while (S ∩ S_goal) = ∅
4    for each task t_i
5      if 𝔈(t_i) ≤ τ ≤ 𝔏(t_i)
6        S = ℐ_{T_{b_i}}(S) ∪ S
7    S = ℐ_{T_r}(S)
8    for each task t_i
9      if 𝔈(t_i) + 𝔗(t_i) ≤ τ ≤ 𝔏(t_i) + 𝔗(t_i)
10       S = ℐ_{T_{e_i}}(S)
11     if = 𝔏(t_i) + 𝔗(t_i)
12       S = S ∩ t_{finished,i}
13   τ = τ + 1
```

Listing 1.1. The breadth-first based algorithm; S is the current state set.

5 A guided BDD algorithm

Instead of searching exhaustively for an optimal schedule in a breadth-first manner, it is possible to guide the search with an adaptation of A* for BDDs [7, 8]. The cost of reaching each set of states is maintained and combined with an underestimate of the remaining cost to reach a goal state. This gives a lower bound on the total cost of a set of states, which means that promising states with a lower cost can be examined first.

When applying the A* algorithm to a problem, a measure of cost and a heuristic to estimate the remaining cost must be established. An intuitive approach for the task graph scheduling problem is to use the length of the schedule as the cost measure, but unfortunately it is very difficult to compute a useful underestimate of the remaining schedule length.

Instead we let the cost be the number of free time slots in the schedule and use the number of free slots at the next step of the schedule as an estimate of the remaining free slots. These choices will give optimal schedules since the number of free slots at the next step is a lower bound on the total remaining free slots, and a schedule with the fewest possible free time slots must be an optimal schedule. To see this, consider the schedules in Figure 1 where the two optimal schedules both have three free time slots, whereas the schedule with five free slots is one step longer – more free time slots necessarily means a longer schedule.

The algorithm, given in Listings 1.2, works by maintaining a priority queue of visited states and their costs. When a state set has been through the start transition (line 7-8), it is (on line 9-15) split according to how many free slots the transition creates, and the new state sets are then enqueued with their new cost. States with equal cost are merged.

When the algorithm has finished, the length of an optimal schedule is the sum of the execution times for all tasks and the number of free slots, divided by the number of processors.

```
1  Q.insert(s_init)
2  while (Q.top() ∩ S_goal) = ∅
3    (h, S) = Q.pop()
4    S = I_{T_r}(S)
5    for each transition T_{e_i}
6      S = I_{T_{e_i}}(S)
7    for each transition T_{b_i}
8      S = I_{T_{b_i}}(S) ∪ S
9    for i = 0 to M
10     Z_i = (S ∩ m_free = i)
11     h' = h + i
12     if Q contains a state p with h' free slots
13       p = p ∪ Z_i
14     else
15       Q.insert(h', Z_i)
```

Listing 1.2. The A* algorithm; h is the number of free slots associated with a state set in Q.

Unfortunately, the algorithm can only compute the length of an optimal schedule and is not capable of actually finding one since it does not keep track of the time for each state. It is possible to split the state sets in the queue even further so that states with different times are separated; this also has the benefit of making it possible to search the most complete schedules first. Unfortunately, the performance turns out to be abysmal in practice because the queue is splitted into so many states that the benefit of using BDDs vanishes.

6 Experimental evaluation

The breadth-first search and A* algorithms have been implemented in C++ with the BDD package BuDDy [12]. Artificially generated task graphs from the Standard Task Graph Set [10] were used to check the algorithms and evaluate their performance.

The experiments were carried out on a SUN Solaris 9 machine with eight 900 MHz UltraSPARC III CPUs (the implementation was, however, single-threaded) and 32 GB RAM.

6.1 Results for graphs with 50 tasks

The breadth-first algorithm and the A* algorithm were run on the first 120 of the graphs with 50 tasks from the Standard Task Graph Set, with 2, 4 and 8 processors. Each process was allowed to run for at most one hour and was given about 2 GB RAM. The results appear in Figure 4 and 5.

Overall, the results show that the breadth-first algorithm can solve many of the problems optimally within one hour. But there are large differences in how much time it takes to solve the different task graphs. Also, some graphs

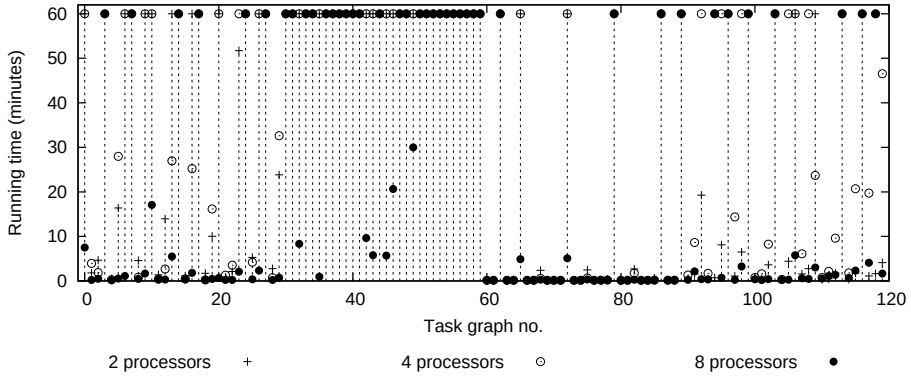

Fig. 4. Breadth-first search results with 2, 4 and 8 processors for the 120 first graphs with 50 tasks. 52.5% were solved with two processors, 48.3% with four and 66.7% with eight.

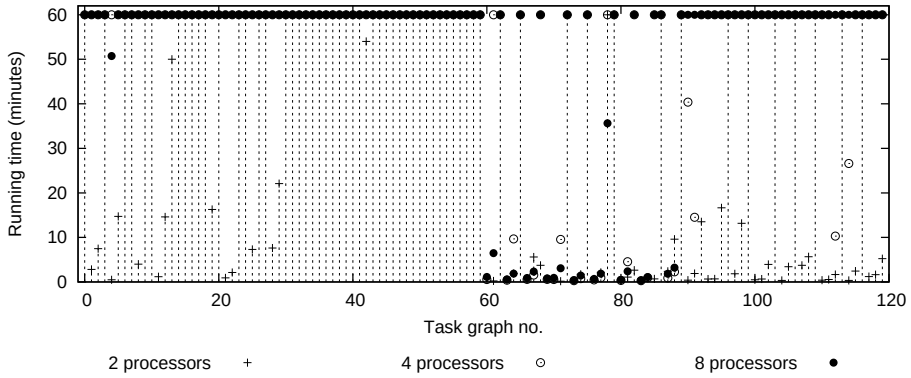

Fig. 5. A* search results with 2, 4 and 8 processors for the 120 first graphs with 50 tasks. 50% were solved with two processors, 18.3% with four and 17.5% with eight.

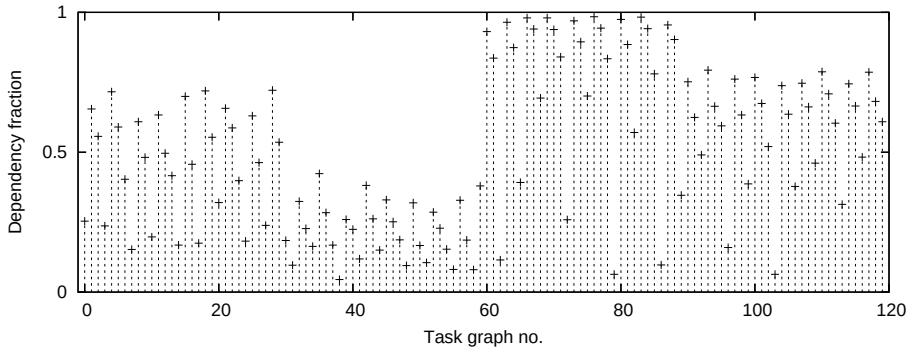

Fig. 6. The dependency fraction for the first 120 graphs. The dependencies are generated using different algorithms in intervals of 30.

are solved fastest with two processors and some with four, although clearly the algorithm is best with eight processors. The differences seem to have to do with the number of states in the state sets. More processors result in more states since there are more possible schedules. The reason that eight processors then was faster seems to be that the state set BDDs exhibited more sharing. Also the $\mathfrak{E}$ and $\mathfrak{L}$ estimates are more accurate with eight processors, which allows the algorithm to cut off a lot of states. And finally the algorithm runs for fewer iterations since the length of the schedule is shorter.

In comparison, the A* algorithm performed worse on almost all instances. There are two explanations for this. First, the A* algorithm will search all schedules with no free slots first, then those with one free slot, etc. If there are no free slots in the optimal schedules this reduces the number of states to search, but if the optimal schedules contain x free slots then the search has to fully explore x non-optimal branches, whereas the breadth-first algorithm examines all of these at the same time. Second, since the A* algorithm does not keep track of the elapsed time it is not possible to use the $\mathfrak{E}$ and $\mathfrak{L}$ optimisations. Indeed, experiments with earlier versions of the breadth-first algorithm without the $\mathfrak{E}$ and $\mathfrak{L}$ optimisations showed that the A* algorithm then was faster on graphs where the optimal schedules did not have any free slots.

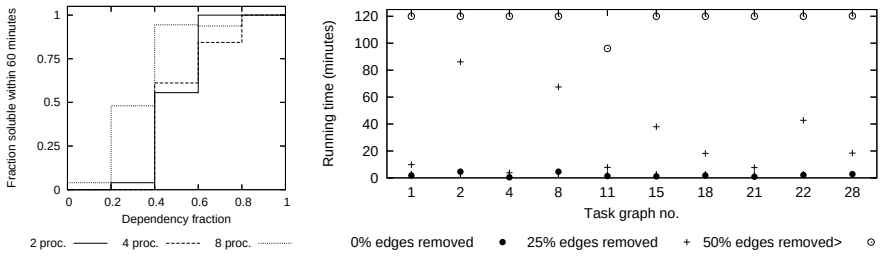

Fig. 7. To the left, the fraction of task graphs that were soluble with the breadth-first algorithm within 60 minutes in intervals of dependency fractions. To the right, the breadth-first algorithm run on selected task graphs with 0%, 25% and 50% edges removed at random.

An interesting question is what characterises the task graphs that the algorithms have trouble solving. The dependencies in the graphs are generated with four different algorithms. The graphs 0–29 and 60–89 are generated with random dependencies, while 30–59 and 90–119 are generated by placing tasks in layers without internal dependencies. Moreover 0–59 use fixed probabilities whereas 60–119 use a fixed number of predecessors which results in relatively fewer dependencies on larger graphs; Figure 6 shows the dependency fraction for the first 120 graphs.

Consider the plot to the left in Figure 7 with the fraction of soluble graphs in the dependency fraction intervals 0–20%, 20–40%, 40–60%, 60–80% and 80–100%. Clearly, the number of dependencies are crucial for the solubility; in the interval 80–100, all graphs were soluble. This is confirmed by the plot to the right

in Figure 7 where the breadth-first algorithm experiment has been repeated on eight quickly solved graphs with 25% and 50% of the edges removed at random. The reason that more dependencies yield better performance is that the dependencies constrain the problem and thus in general lead to fewer feasible schedules to examine.

6.2 Results for graphs with more than 50 tasks

The results of running the breadth-first search algorithm on the first ten 100, 300 and 500 task graphs appear in Table 1. Since the breadth-first search algorithm was faster than the A* algorithm, we did not try the latter on the large graphs.

	0	1	2	3	4	5	6	7	8	9
100	10h25m	–	6m	38m	–	7m3s	55m	–	–	1h21m
d (%)	58.2	41.8	80.0	72.2	54.0	83.6	75.3	58.1	34.6	76.4
300	4h8m	–	–	52m	23h7m	–	59m	24h46m	–	–
d (%)	91.8	85.8	77.8	93.6	89.6	81.5	94.8	90.3	85.9	72.0
500	–	18h10m	–	–	11h24m	139h16m	–	–	46h54m	–
d (%)	79.3	94.9	92.5	84.8	95.5	93.6	88.9	78.0	95.1	91.2

Table 1. The breadth-first search algorithm on the first ten graphs with 100, 300 and 500 tasks. For the graphs without a result the state set BDDs grew so large that the algorithm would not have finished in reasonable time.

Unfortunately, the algorithm does not seem to scale well. With twice as many task, it is an order of magnitude slower. The extra tasks enlarge the state representation and increases the number of transition relations. As expected the performance is also correlated with the dependency fraction for the large graphs.

7 Conclusions and future work

We have represented the task graph scheduling problem as a BDD state space search problem and explored a number of design alternatives. The state representation was chosen so that any temporarily unused BDD variables could be ignored, and the transition relation was split into three semantic phases of which the first and last phase were further partitioned into a relation per task. These two ideas made the algorithms tractable in practice. Furthermore, the bounds for the time frame in which a given task could run were estimated and used to save more than half of the image computations, and also allowed pruning of some non-optimal schedules.

Some other techniques from the literature [4] turned out to have a negative impact on the performance. Dynamic variable ordering was tried but did not reduce the state set BDDs, and pruning of already visited states enlarged the state set BDDs instead of reducing their size.

But overall, a breadth-first search algorithm was found to be able to solve 50% of the task graphs with 50 tasks and at least 40% of the possible dependencies within 60 minutes. Larger task graphs with many dependencies could also be solved, although it then takes more than ten hours to find an optimal schedule. A guided search strategy with a BDD adaptation of the A* algorithm was also tried, but was found to be inefficient. We attribute this to the fundamental problem of finding a heuristic for the scheduling problem that is able to compete with the brute-force breadth-first search.

Future work could be to extend the BDD representation to model more complex scheduling problems, such as scheduling with communication costs or job-shop problems.

References

1. Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. In *Theoretical Computer Science (to appear)*, 2004.
2. Henrik Reif Andersen. An introduction to binary decision diagrams. Technical report, IT University of Copenhagen, 1999.
3. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
4. Randal E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD, San Jose/CA*, pages 236–243, 1995.
5. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.
6. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
7. Stefan Edelkamp and Frank Reffel. OBDDs in heuristic search. In *KI - Kunstliche Intelligenz*, pages 81–92, 1998.
8. R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA*: An efficient BDD-based heuristic search algorithm. In *Proceedings of 18th National Conference on Artificial Intelligence (AAAI'02)*, pages 668–673, 2002.
9. Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.
10. Kasahara Laboratory. Standard Task Graph Set. `http://www.kasahara.elec.waseda.ac.jp/schedule/index.html`.
11. E. L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 1978.
12. Jørn Lind-Nielsen. *BuDDy: Binary Decision Diagram package*, 2002. `http://www.itu.dk/research/buddy/`.
13. J. Rasmussen, K. G. Larsen, and K. Subramani. Scheduling using priced timed automata. In *Proc. 10th Int. Conf. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2004)*, pages 220–235, 2004.
14. Theo Ruys. Optimal Scheduling Using Branch and Bound with SPIN 4.0. In *Model Checking Software – Proceedings of the 10th International SPIN Workshop (SPIN 2003)*, pages 1–17, Portland, OR, USA, May 2003.