Scheduling Borrels to maximize students attendance and social experience

Floris van der Hout
Utrecht University , 7234080, Netherlands

Thomas van Maaren
Utrecht University , 9825827, Netherlands

Rens Versnel
Utrecht University , 2693852, Netherlands

Wessel van der Ven
Utrecht University , 2828189, Netherlands

Ids de Vlas
Utrecht University , 6967396, Netherlands

Abstract

A challenge for event organizers is attracting the maximum number of attendees. In this paper we introduce the borrel scheduling problem. We prove that no deterministic algorithm can achieve a constant competitiveness for the problem in a number of online settings with different restrictions. Moreover, we show that this problem is NP-hard in the offline setting even for a restricted case. To cope with this hardness we have created a brute forcing algorithm for an offline setting. We introduce a social matrix to take social relationships into account in the borrel scheduling problem, which is also solved by our offline algoritm. Finally we evaluate our offline algorithm with a number of experiments.

2012 ACM Subject Classification Online algorithms

Keywords and phrases Borrels Scheduling, NP-completeness, competitive ratio's

Digital Object Identifier 10.4230/LIPIcs...5

1 Introduction

Image you are part of the student organization responsible for organizing "borrels" (social gatherings) for freshmen. However, the freshmen have many obligations such as deadlines, group meetings, exams, ect. These obligations are flexible and can be rescheduled or delayed, but they still need to be completed eventually. Freshmen are eager to attend the borrels and will do so as long as they don't have any conflicting obligations. However, if their schedules clash with their obligations, they will prioritize those obligations over the borrel. Borrels are typically free for students to attend, but they do incur costs for the student organization, such as renting a venue, providing snacks, and other logistical expenses. It's common for borrels to offer the first drink for free, but any additional drinks must be purchased. The revenue generated from drink sales is used to help cover the borrels costs. If students enjoy themselves, they're more likely to buy drinks. One key factor that can enhance their experience is being able to attend the borrels with their friends. Therefore, it's in the student organization's interest to not only focus on attendance but also on maximizing the happiness of the freshman by scheduling borrels in a way that allows groups of friends to participate together. This can boost the overall success of the event both socially and financially.

This paper will present two offline algorithms for scheduling borrels: one focused on optimizing student attendance, and the other designed to maximize the overall happiness of the

freshman by considering factors like social connections and shared availability. Additionally, we will provide problem competitive ratios for online algorithms which focusing on maximizing the attendance of freshmen at the borrels in multiple scenario's.

1.1 Literature review

We will visit several relevant problems in the literature, the first of which is the Social Event Scheduling Problem (SES) [**?**]. In this context, the goal is to schedule a series of events or borrels while maximizing attendance or overall utility, considering that third-party events (in our problems context obligations) are already scheduled. Utility is defined as the expected attendance across all scheduled events. The difference between Borrels Scheduling and Social Event Scheduling lies in how competing events are treated. In the SES problem, competing events are those that have already been scheduled by third parties, resulting in fixed time slots. In contrast, our constraints, which can be viewed as competing events, can often be postponed within their designated time windows. Furthermore, in the SES problem, each event is assigned a probability indicating a user's likelihood of attending. If two events overlap, the probability distribution determines which event the user will choose. However, if there are 5 events, we cannot impose a requirement for users to attend exactly 4 out of 5 third-party events (the obligations) while leaving the remaining event slots for our events (borrels). In our case, the probability of attending an event changes based on which event the user chooses to attend. There is no trivial way to translate the Borrel Scheduling problem into the SES problem.

Another relevant problem is the Optimization Of Calendar Events. Especially when optimizing the scheduling of meetings within large organizations, where the objective is to maximize attendance while considering each participant's individual schedule/calendar. Since it is not always feasible for everyone to attend, the focus shifts to maximizing the number of attendees at the meeting which is similar to our problem [**?**] [**?**]. However, solutions derived from this model are not directly applicable to our problem. In these frameworks, a participant calendars time slots are either marked available or unavailable which is different from our problem. In our case, there is a time window within which certain obligations must be fulfilled, but the exact timing has yet to be determined. This differs from the SES problem because the Optimization Of Calendar Events allows for soft constraints. Soft constraints refer to events in a user's schedule that can be rescheduled to another date for a small penalty cost. This approach resembles the Borrel Scheduling problem, where rescheduling is allowed if the event can be moved to another time within the available time slot. However, incorporating this flexibility significantly complicates the problem, making it more challenging to solve.

The final relevant problem is the Job Scheduling Problem with Deadlines [**?**], which closely parallels the obligation and borrel planning aspects of our study. In this scenario, each obligation can be viewed as a job that needs to be scheduled on a specific machine, with both a deadline and a start time. Similarly, we have borrels that also need to be scheduled on these machines. In our problems context, we want to maximize for the scheduling borrels on different machines simultaneously, the same as maximizing attendance at the events. However, this is different from the objective in the job scheduling problem with deadlines which is to minimize the makespan while ensuring that all tasks remain feasible within their respective constraints. There is no trivial way to translate the Borrel Scheduling problem to the Job Scheduling problem with deadlines.

2 Preliminaries

2.1 Formal problem definition

The timeline is partitioned into t integral time slots. The time interval from time 0 (inclusively) to time 1 (exclusively) is time slot 1, and the time interval from time $i-1$ (inclusively) to time i (exclusively) is time slot i for all $i \geq 1$. There are m borrels, and each borrel h has a length of b_h time slots. There are n students. Each student i has a set of obligations $I_i = \{(I_{i,1}, p_{i,1}), (I_{i,2}, p_{i,2}), ..., (I_{i,l_i}, p_{i,l_i})\}$, where $I_{i,j} = [r_{i,j}, d_{i,j}]$ is a time interval from time slot $r_{i,j}$ to time slot $d_{i,j}$, and $p_{i,j}$ is the time slots needed by the obligation j that has to be finished within $I_{i,j}$. A feasible schedule for a student i of obligation j is a schedule $S_{i,j}$, which is a set of time slots, such that $S_{i,j} \subseteq I_{i,j}$ and $|S_{i,j}| = p_{i,j}$. Moreover, for a student i and two of their obligations j and j', $S_{i,j} \cap S_{i,j'} = \varnothing$. Note that we assume $p_{i,j}$ are integral for all i and j, and we cannot assign a time slot partially to an obligation. As a borrels planner, you aim to schedule each borrel h starting at time slot s_h and lasting until time slot $s_h + b_h - 1$, such that as many students attend the borrels as possible. More formally, you want to find the start time s_h of each borrel h such that there exists a set of feasible schedules $S_{i,j}$ for every pair of student i and obligation j such that the conflict among the borrels and the students' schedules, $[s_h, s_h + b_h - 1] \cap S_{i,j} \neq \varnothing$ for some i and j, is minimized. We also assume that borrels start at integral time (that is, the beginning of a time slot).

2.1.1 Offline setting

In the offline setting, all the parameters are known to the algorithm from the very beginning. That is, the algorithm knows t, m, b_h for all $h \in [1, m]$, n, and I_i for all $i \in [1, n]$. Once the algorithm receives the information, it should suggest schedules for all borrels.

2.1.2 Online setting

In the online setting, the online algorithm knows the number of time slots (t), the number of students (n), the number of borrels (m), and all b_h for each borrel h in the beginning but only learns about $I_{i,j} = [r_{i,j}, d_{i,j}]$ and $p_{i,j}$ at the beginning of time slot $r_{i,j}$ (that is, at time $r_{i,j} - 1$). The online algorithm has to decide if a borrel will start at time slot $x + 1$ right at time x (that is, the beginning of time slot $x + 1$). Formally, at time x (that is, the beginning of time slot $x + 1$), the algorithm first learns the obligations that have release time at time slot $x + 1$ and then decides if it wants to schedule a borrel starting from time slot $x + 1$.

2.1.3 Decision problem

In order to prove the complexity class of borrel scheduling it needs to be formulated as a decision problem. The aim of the original problem is to schedule each borrel h starting at time slot sh and lasting until time slot $s_h + b_h - 1$, such that as many students attend the borrels as possible. In the decision version of this problem the aim is then to decide if there exists an assignment of borrels to timeslots such that there are at least k attendances in total for some $k \in \mathbb{N}$.

2.1.4 Borrel scheduling with at most 2 concurrent obligations (BS-2CO)

We define a special case of borrel scheduling, henceforth referred to as BP-2CO, where each student has at most two obligations at every timeslot. More formally, For all students i and

any three obligations $o_1, o_2, o_3 \in I_i$ they do not span a common timeslot unless some of these obligations are the same: $(o_1 \neq o_2 \wedge o_2 \neq o_3 \wedge o_3 \neq o_1) \implies o_1 \cap o_2 \cap o_3 = \emptyset$.

The decision version of this problem is defined analogous to the decision version of the original version 2.1.3.

2.2 Set cover

In order to prove NP-hardness, we will consider the set cover decision problem with a universe $U = \{u_1, u_2, ..., u_{|U|}\}$ containing all possible elements and the set $\mathcal{S} = \{S_1, S_2, ..., S_{|\mathcal{S}|}\}$ containing subsets of the universe U and a goal $g \in \mathbb{N}$. The set $\mathcal{S}$ must contain all elements in the universe, so its union must equal U: $\bigcup_{S_a \in \mathcal{S}} S_a = U$. The aim in the set cover problem is to decide if there exists a subset A of $\mathcal{S}$ such that the union of A is equal to U and $|A| \leq g$.

3 Problem competitive ratio

3.1 Base problem

To prove the competitive ratio of the base problem we will start by taking in to consideration the following adversary cases:
$A := \{t = 2, m = 1, b_1 = 1, \forall i : 1 < i \leq n : I_i = \{([t_1, t_1], 1)\}\}$
$OPT(A) = n$ where b_1 is held at t_2
$B := \{t = 2, m = 1, b_1 = 1, I_1 = \{([t_2, t_2], 1)\}, \forall i : 1 < i \leq n : I_i = \{([t_1, t_1], 1), ([t_2, t_2]1)\}\}$
$OPT(B) = 1$ where b_1 is held at t_1

For both cases every student except for one has an obligation at timeslot t_1. Note that $A \subset B$, and the adversary can freely choose to switch after $t = 1$ between these cases depending on the behaviour of any arbitrary online algorithm.

Since there exists one borrel b_1 with length 1 every online algorithm can either plan this borrel on t_1 or not, creating two possible cases.
If some algorithm ALG_1 chooses to host the borrel on t_1 then the adversary case is A where $ALG_1(A) = 1$ and $OPT(A) = n$ giving a competitive ratio of $\frac{OPT(A)}{ALG_1(A)} = \frac{n}{1} = n$.
If some algorithm ALG_2 chooses not to host the borrel on t_1 then the adversary case is B where $ALG_2(B) = 0$ and $OPT(B) = 1$ giving a competitive ratio of $\frac{OPT(B)}{ALG_2(B)} = \frac{1}{0} = \infty$

This gives the problem a competitive ratio of n

Moreover, every algorithm hoping to attain a competitive ratio $c < \infty$ should always plan a borrel the first chance it gets to avoid having a score of 0.

3.2 Multiple borrels

Now we will investigate the competitive ratio when $m = 2$. Every algorithm must still schedule a borrel at the first possible moment where at least one person can attend for the same reasons as $m = 1$, but after that the algorithm has a lot more freedom and can't be "forced" to perform an action anymore. This still isn't enough to achieve a constant competitive ratio however.
To provide a lower bound of the competitive ratio of the problem with $m = 2$, we'll look at the following set of cases:
$A := \{t = 3, m = 2, b_1 = 1, b_2 = 1, I_1 = \{([t_2, t_3], 2)\}, \forall i : 1 < i \leq n : I_i = \{([t_1, t_1], 1), ([t_2, t_3], 2)\}\}$
$OPT(A) = 1$, where b_1 is held at t_1. b_2 can be held at any time and not affect the score.

$\forall 1 \leq p \leq n : B_p := \{t = 3, m = 2, b_1 = 1, b_2 = 1, I_1 = \{([t_2, t_2], 1)\}, \forall i : p < i \leq n : I_i = \{([t_1, t_1], 1), ([t_2, t_2], 1)\}, \forall j : 1 < j \leq p : I_j = \{([t_1, t_1], 1)\}$
$OPT(B_p) = n + p$ where b_1 is held at t_2, and b_2 is held at t_3.
$\forall 1 \leq p \leq n : C_p := \{t = 3, m = 2, b_1 = 1, b_2 = 1, I_1 = \{([t_2, t_2], 1), ([t_3, t_3], 1)\}, \forall i : p < i \leq n : I_i = \{([t_1, t_1], 1), ([t_2, t_2], 1), ([t_3, t_3], 1)\}, \forall j : 1 < j \leq p : I_j = \{([t_1, t_1], 1), ([t_3, t_3], 1)\}\}$
$OPT(C_p) = p + 1$, with b_1 held at t_1 and b_2 at time t_2.
Adversary cases B_p and C_p can be understood as cases where p people will be able to attend at t_2. To start, any arbitrary online algorithm must schedule a borrel at t_1, because if this the algorithm decides to forgo this, the adversary can switch to case A, which would result in a competitive ratio $\frac{OPT(A)}{ALG(A)} = \frac{1}{0} = \infty$. Therefore, we will assume every algorithm chooses to schedule their first borrel at timeslot t_1.

For the purpose of contradiction, we assume an online algorithm ALG with a competitive ratio of c exists. This algorithm must forgo scheduling a borrel at t_2 if there are less than $\frac{n}{c}$ students that can attend. However, this still gives a contradiction.

Consider $B_{\lfloor \frac{n}{c} \rfloor - 1}$. If ALG does schedule a borrel at timeslot t_2, the amount of people that will be able to attend is equal to $\lfloor \frac{n}{c} \rfloor - 1 + 1 = \lfloor \frac{n}{c} \rfloor$. This would result in a competitive ratio of $\frac{OPT(B_{\lfloor \frac{n}{c} \rfloor - 1})}{ALG(B_{\lfloor \frac{n}{c} \rfloor - 1})} = \frac{n + \lfloor \frac{n}{c} \rfloor - 1}{\lfloor \frac{n}{c} \rfloor} \geq c + 1 - \frac{c}{n}$. As long as $n > c$, $c + 1 - \frac{c}{n} > c$ so the performance is worse than $c \cdot OPT(B_{\lfloor \frac{n}{c} \rfloor - 1})$. Therefore an algorithm that would schedule a borrel at timeslot t_2 for $p = \lfloor \frac{n}{c} \rfloor - 1$ would not have a competitive ratio of c. This means an online algorithm that schedules a borrel at t_2 for $B_{\lfloor \frac{n}{c} \rfloor - 1}$ cannot have a constant competitive ratio c.

Now consider the case where ALG does not schedule a borrel at timeslot t_2. Looking at the case $C_{\lfloor \frac{n}{c} \rfloor - 1}$ will give us that $ALG(C_{\lfloor \frac{n}{c} \rfloor - 1}) = 1 + 0$, as the first borrel must be placed at t_1 with 1 person attending, and the second borrel must be planned at t_3 with 0 people attending. $OPT(C_{\lfloor \frac{n}{c} \rfloor - 1}) = \lfloor \frac{n}{c} \rfloor - 1 + 1 = \lfloor \frac{n}{c} \rfloor$. Therefore the competitive ratio for any algorithm that does not schedule a borrel at t_2 in the case of $C_{\lfloor \frac{n}{c} \rfloor - 1}$ is equal to $\frac{OPT(C_{\lfloor \frac{n}{c} \rfloor - 1})}{ALG(C_{\lfloor \frac{n}{c} \rfloor - 1})} = \frac{\lfloor \frac{n}{c} \rfloor}{1} = \lfloor \frac{n}{c} \rfloor$ which is greater than c for every $n > c^2 + c$.

Since ALG cannot attain a competitive ratio of c regardless of the decision it makes in the case of $p = \lfloor \frac{n}{c} \rfloor - 1$, ALG cannot have a competitive ratio of c and therefore our assumption that an online algorithm with a constant competitive ratio exists must be false.

3.3 Lookahead

Next we will look at the problem where the algorithm can look k timeslots ahead instead of 1. So in a setting with $k = 3$ the algorithm would be able to see obligation information about timeslots $\{t_1, t_2, t_3\}$ at timeslot t_1 and timeslots $\{t_2, t_3, t_4\}$ at timeslot t_2.

If $k \geq t$ the problem is simply the offline problem and thus trivial with a competitive ratio of $c = 1$.

For a smaller k the competitive ratio can be proven similarly to the base problem with adversary cases:
$A := \{t = k + 1, m = 1, b_1 = 1, \forall i : 1 < i \leq n : I_i = \{([t_1, t_k], 1)\}\}$
$OPT(A) = n$ where b_1 is held at t_{k+1}
$B := \{t = k+1, m = 1, b_1 = 1, I_1 = \{([t_{k+1}, t_{k+1}], 1)\}, \forall i : 1 < i \leq n : I_i = \{([t_1, t_k], 1), ([t_{k+1}, t_{k+1}]1)\}\}$
$OPT(B) = 1$ where b_1 is held at t_1

At timeslot t_1 any algorithm will see a timeslot with 1 free student at t_1 and 0 free students

at $\{t_2...t_k\}$ creating once again the same choice every algorithm must make at t_1 for hosting a borrel at t_1.

Then like the base problem: If some algorithm ALG_1 chooses to host the borrel on t_1 then the adversary case is A where $ALG_1(A) = 1$ and $OPT(A) = n$ giving a competitive ratio of $\frac{OPT(A)}{ALG_1(A)} = \frac{n}{1} = n$.
If some algorithm ALG_2 chooses not to host the borrel on t_1 then the adversary case is B where $ALG_2(B) = 0$ and $OPT(B) = 1$ giving a competitive ratio of $\frac{OPT(B)}{ALG_2(B)} = \frac{1}{0} = \infty$. The adversary can freely switch between these after the algorithm has chosen whether to plan a borrel at t_1, as the revealed information is exact same.

This gives this problem also a competitive ratio of n.

3.4 Combination of lookahead and multiple borrels

Combining all previous restrictions is unfortunately also uncompetitive. This can be proven with the same proof as the one used to proof multiple borrels 3.2 is uncompetive, with the addition of "stalling obligations" to make lookahead ineffective. These stalling obligations could be obligations between the obligations placed in the original proof with a lenght greater than or equal to the amount of lookahead. This would provide the algorithm with the same amount of information as in the case of multiple borrels, and would therefore achieve the same competitiveness.

4 BS-2CO is NP-complete

In this section we show that the special case BS-2CO of borrel scheduling decision problem (described in 2.1.4) is NP-complete. We start by showing that this decision problem is in NP. Then we will take the set cover decision problem, which is NP-complete [**?**], and reduce it to the BS-2CO decision problem. Then we will have shown that the decision version of BS-2CO is NP-complete. The aforementioned reduction also proves the NP-hardness of the base borrel scheduling problem since it is at least as hard as BS-2CO.

4.1 BS-2CO is in NP

In order to prove that the decision version of the special case of borrel scheduling BS-2CO 2.1.4 is in NP, we define a certificate.

Definition of certificate polynomial in size Given for each borrel a starting timeslot and a set of students attending it such that there exists some fullfilment of the obligations where the students can attend those borrels, this certificate consists of

- a list of m integers w_i where the i'th value denotes the starting timeslot of the i'th borrel. This is $O(m)$ and thus polynomial in the input size.
- for each borrel h a list of values $a_{h,i}$ where the i'th value is 1 if student i attends borrel h in the chosen schedule and 0 otherwise. This is $O(m \cdot n)$ and thus polynimal in the input size.

We can see that the size of this certificate is polynomial in the input size.

Existance of certificate for yes-instances We also know that this certificate exists for all yes-instances, because for every instance where it is possible to have k attendances there exists, by definition, some assignment of borrels and some schedule of obligations such that the students can attend at least k borrels. This assignment of borrels is equal to the desired values w_i in the certificate and the schedule of obligations can be translated to the booleans $a_{h,i}$ by checking if borrel h overlaps with any of the scheduled obligations of student i.

Verification Given a decision problem BS-2CO and a certificate with integers w_i and booleans $a_{h,i}$ we propose a greedy algorithm that can verify this certificate in polynomial time. Note that, the verifier can consider each student serparately, because the specific borrels that need to be attended by that students are given and the obligations of different students are independent. Therefore without loss of generality we will only consider the feasibility of the schedule for a single student.

Given the lists B_i of borrels b_h that must be attended by students i and the list A_i of obligations $I_{i,j}$ of this student, the verifier works in 3 stages:

Sorting the input and checking borrel overlap We start by sorting the list of borrels in ascending order of starting timeslot and sorting the list of obligations in ascending order of starting timeslot (start of the interval). Now for each borrel that is supposed to be attended by the student the verifier checks that there is no overlap between the attended borrels. This can easily be done in linear time since we have sorted the borrels.

Shift and shorten obligations Then the verifier can use the information that the borrels b_h in B_i must be attended, to shorten and shift obligations. This is possible without changing the feasibility of the schedule, because in a feasilble schedule no obligations can be worked on during these selected borrels. The schedule is therefore exactly as feasible after removing the invervals of these borrels from the obligations. This can be done by iterating through the start- and end times of all obligations in ascending order and subtracting the number of timeslots of attended borrels that precedes this start- or end time from it. So, for a start- or end timeslot x of an obligation, we subtract $\sum_{b_h \in P_x} \min\{b_h, b_h + (s_h - x)\}$ from it where P_x is the set of all borrels that have an end time before or equal to x. See 1.

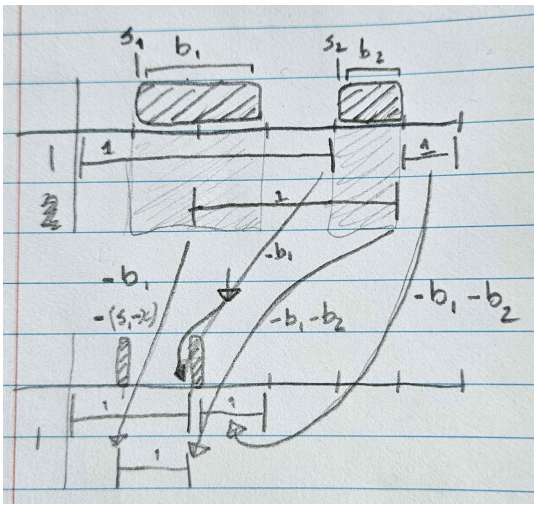

Figure 1 The shift of obligations

Fill in obligations in earliest deadline order Now since we already removed the overlap between obligations and borrels that need to be attended, we only need to find out if there exists some way to schedule all obligations within their respective intervals. This can be done greedily by prioritizing the obligations with the earliest deadline of the two concurrent deadlines. The algorithm goes through all obligation intervals in ascending order of starting timeslot and always works as much as possible on the unfulfilled obligation with the earliest deadline (until the start of a new obligation, then it re-evaluates which obligation has the earliest deadline). If this algorithm encounters an obligation that cannot be fulfilled, then we know that this schedule if infeasible. Otherwise, the schedule is shown to be feasible.

After succesfully filling in all obligations for all students, we can count all scheduled borrel attendances B_i for all students to verify that it is indeed possible to have k attendances.

Proof of optimality We can assume without loss of generality that there are exactly two concurrent obligations for every students at every timeslot after removing the intervals of the borrels (the "Shift and shorten" step). This is because we can always add an obligation (at most polynomially many) with workload of 0 to fill up any gaps.

We show the optimality of the greedy obligation checking by induction. We split the timeslots into sections by splitting on deadlines. I.E., we consinder the maximal time sections that do not contain a deadline in the middle (but start and end in a deadline) in chronological order. We show that if all obligations in the sections before a given section are scheduled optimally then after the algorithm the obligations are scheduled optimally until after this inteval as well.

Given a section $[a, b]$ of timeslots in which two concurrent obligations $I_{i,x}, I_{i,y}$ we reduce the workload of the obligation by how many timeslots it was scheduled for before the start of this section a and we set the start times of the obligation interval $r_{i,x}$ to be at least the start of the section a. Now there are two obligations of which at most one extends beyond (has a deadline later than the end of) this section. If the deadlines of both obligations are exactly at the end of the section $d_{i,x} = d_{i,y} = b$ then we can trivially say that an optimal schedule for these two obligations is completely filling in there two obligations in any order. Therefore we only need to consider the case where exactly one deadline is later than the end of the section; we will call this obligation $I_{i,x}$.

We know that obligation $I_{i,y}$ is must have a deadline equal to b (otherwise the end of the section would not be there), therefore we will need to work on this obligation for $p_{i,y}$ in the interval $[a, b]$. Then we can work on obligation $I_{i,x}$ only for the remaining time in this section $(b + 1 - a) - p_{i,y}$. We can see that our algorithms schedules the obligations in this way and that this is also (at least as feasible as) the optimal schedule up to this point, because there can be at most $b + 1 - a$ timeslots of work done and the amount of that which is worked on $I_{i,x}$ is forced.

4.2 Reduction of set cover problem to BS-2CO

In this section we reduce the set cover decision problem to the special BS-2CO case of borrel the scheduling decision problem. We do this by constructing a special instance of the borrel scheduling problem given any instance of set cover. Then we show that there exists a set cover of less than some k sets if and only if the outcome of this instance of BS-2CO is positive. This shows that the borrel scheduling decision problem is NP-hard and therefore, together with the previous proof that it is NP, shows that it is NP-complete. Additionally, since this shows that a special case of the borrel scheduling problem is NP-hard, we have shown that the base problem of borrel scheduling is also NP-hard.

Construction of the reduction Given an instance of the set cover decision problem with a universe $U = \{u_1, u_2, ..., u_{|U|}\}$, a set $\mathcal{S} = \{S_1, S_2, ..., S_{|\mathcal{S}|}\}$ containing subsets of the universe U, and a goal $g \in \mathbb{N}$. We create an instance of borrel scheduling with the following parameters:

$t = \|\mathcal{S}\|$	the number of timeslots is equal to the number of sets in $\mathcal{S}$.
$n = \|U\|$	the number of students is equal to the size of the universe.
$m = g$	the number of borrels is equal to the set cover goal.
$b_h = 1$	the length of each borrel h is 1.

Furthermore, for all students i and subsets $S_a \in \mathcal{S}$ we create an obligation j with $I_{i,j} = [a, a]$ and $p_{i,j} = 1$ for student i if $i \notin S_a$. Additionally we add an obligation x to each student i with $I_{i,x} = [1, t]$ and $p_{i,x} = |\{a : i \in S_a\}| - 1$. The borrel decision problem is to determine if there exists a schedule such that there are at least $|U|$ attendances. See 2

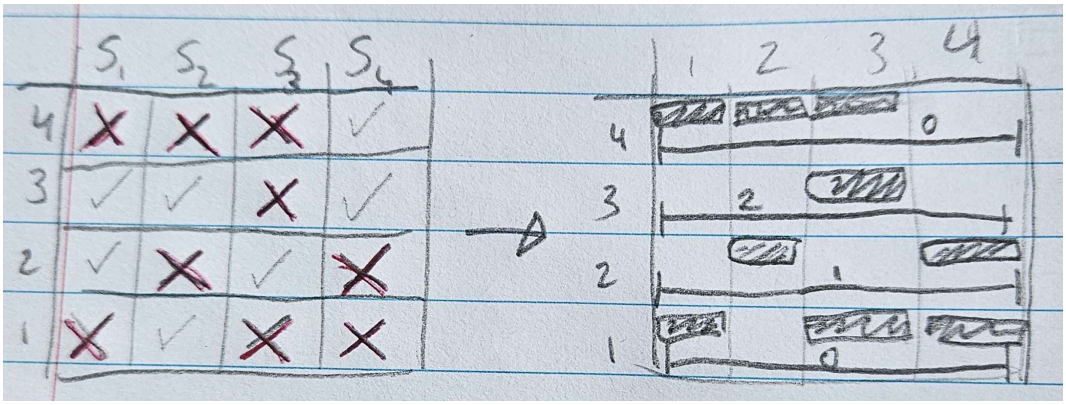

Figure 2 The reduction from set cover to borrel planning

Proof of the reduction We will show that this instance of the borrel scheduling problem is equivalent to the original set cover problem.

Note, first, that each student can attend at most one borrel, since there is one timeslot for each set in $\mathcal{S}$ and for every student i there is an obligation of length 1 for every set in $\mathcal{S}$ that contains i and there is an obligation of length one less that the number of sets in $\mathcal{S}$ that do not contain i. We can also see that a student i can only attend a borrel at timeslot j if i is in the j'th set S_j.

Then if the borrel scheduling problem decided that there is a feasible borrel schedule such that there are at least $|U|$ attendances with k borrels, then these must be $|U|$ different students and therefore all elements in the universe.

This borrel schedule must also use at most k borrels corresponding to sets in $\mathcal{S}$ to cover this universe. This shows if there exists a borrel schedule such that at least $|U|$ students can attend, then there must exist a set cover for the original problem of size at most g.

Then, conversely, if the original set cover instance can be covered in at most g sets, we can show that there exists some assignment of the borrels such that there are at least $|U|$ attendances. Given the g sets that cover the universe, the g borrels can be scheduled at the timeslots corresponding to those sets. Then each student can attend at least one borrel since the sets cover the entire universe, resulting in at least $|U|$ attendances.

Since we have shown that a yes-instance of the special instance of borrel scheduling implies a

yes-instance of the original set cover, and we have shown that a yes-instance of the original set cover implies a yes-instance of the borrel scheduling (and by contraposition a no-instance of borrel scheduling implies a no-instance of the set cover) it follows that the constructed special instance of borrel scheduling problem and the set cover problem are equivalent and thus set cover can be reduced to borrel scheduling.

Lastly we define s as the total number of elements in subsets of $\mathcal{S}$, $s = \sum_{S_a \in \mathcal{S}} |S_a|$, and then we can see that this reduction can be done in polynomial time because we can

- Define the number of students and timeslots and borrels trivially in polynomial time in s.
- Define the length of all borrels in $O(|\mathcal{S}|) = O(s)$ and thus polynomial time in s.
- Define all $O(|U|^2)$ obligations in $O(s)$ time each and thus polynomial time in s.

Since the input size is at least s and all parts of the reduction can be done in time polynomial in s, we have shown that the reduction can be done in polynomial time. This means that BS-2CO is NP-hard. And since the base problem of borrel scheduling at least as hard, it is also NP-hard. Moreover, since the base problem (and thus BS-2CO) is a special case of the borrel scheduling with social matrix (i.e., the case with the identity matrix), we have shown that borrel scheduling with a social matrix is also NP-hard.

5 Offline Algorithms

In this section we will provide an algorithm for the base case as well as an algorithm for solving a variation of the base case. The source code can be found at `https://github.com/tvmaaren/ADS-Borrels/tree/main/offline%20algoritm`. As we will see in section 4 the problem is NP-hard and therefore providing a polynomial algorithm for this problem would prove that NP=P. It is therefore not worthwhile to try and find a polynomial algorithm for this problem and therefore all the algorithms in this section will have exponential time. Even though this is quite bad, it is still possible to make an algorithm better than going through all possibilities.

5.1 Iterators

Our algorithms will be a brute force algorithm for a great part. This makes it necessary to iterate over all different possibilities. We will see that in some cases our algorithm can be made faster by choosing a good ordering in our iteration.

5.1.1 Borrel positions

First of all we want to iterate over all possible borrel positions. If for example the amount of borrels is 2 ($m = 2$), the first borrel is of length 3 and the second of length 1 ($b = \{3, 1\}$) and the amount of timeslots is 5 ($t = 5$) we see that starting time of the borrels have (s) have 15 possibilities:

$$\{1,1\},\{2,1\},\{3,1\},\{1,2\},\{2,2\},\{3,2\},\{1,3\},\{2,3\},$$
$$\{3,3\},\{1,4\},\{2,4\},\{3,4\},\{1,5\},\{2,5\},\{3,5\}.$$

Note that $b_h + s_h - 1 \leq t$ always has to hold to ensure that the borrel is completely inside of the time interval. We can construct a general iterator, by increasing the s_1 until it hits the end of the time interval. After this it increases s_2 by 1 and it can start over again with s_1. It does this until s_2 hit the end of the time interval and it increments s_3 and so on. Our borrel iterator is defined in the appendix (see algorithm 1).

Now we will prove that the borrel iterator is correct. This means that the borrel iterator iterator should at some point reach every possible case. Let X be the set of all possible cases

$$X := \{s \in \{1, \ldots, t\}^n \mid s_h + b_h - 1 \leq t\}. \tag{1}$$

Given that we have a fixed amount of borrels m, a fixed length of borrels b and a fixed amount of time slots t. We define $f : \{1, \ldots, t\}^n \to \{1, \ldots, t\}^n$ as $f(s) = (\pi_2 \circ \text{borrelIt})(t, m, b, s)$ where π_2 is the second projection.

▶ **Theorem 1.** $\{f^n((1, \ldots, 1)) \mid n \in \mathbb{N}\} = X$.

Proof. We can give every element of $s \in X$ the following index.

$$i(s) = \sum_{h=1}^{m} \left((s_h - 1) \cdot \prod_{h'=1}^{h-1} (t + 1 - b_h) \right)$$

This index will range from 0 to $\prod_{h=1}^{m}(t+1-b_h)-1 = \#X-1$. We will see that $i \circ f \circ i^{-1}(n) = (n+1) \text{mod} \#X$. Let $n \in \{0, \ldots, \#X - 1\}$ and $s = i^{-1}(n)$.

Assume $n = \#X - 1$. We see that $n = \#X - 1$ if and only if $s_h = t + 1 - b_h$ for all h. We therefore see that $(s_h + 1) + b_h - 1 \leq t$ is not true, hence s_h will be set to 0 fore every h, hence $i(f(i^{-1}(n))) = i(f(s)) = i(0, \ldots, 0) = 0 = (n + 1) \mod \#x$.

Assume $n < \#X - 1$. Let k be the smallest values such that $s_k \neq t + 1 - b_h$. We see that s_h will be zero for all $h < k$ and s_k will be incremented by one and s_h for $h > k$ will remain the same. We see that $f(s) = (0, \ldots, 0, s_k + 1, s_{k+1}, \ldots, s_m)$, hence

$$\begin{aligned} i \circ f \circ i^{-1}(n) &= i(0, \ldots, 0, s_k + 1, s_{k+1}, \ldots, s_m) \\ &= i(t + 1 - b_1, \ldots, t + 1 - b_{k-1}, s_k, \ldots, s_m) + 1 \\ &= i(s) + 1 = n + 1 = (n + 1) \mod \#X - 1. \end{aligned}$$

We therefore see that $\{i \circ f^n \circ i^{-1}(0) \mid n \in \mathbb{N}\} = \{0, \ldots, \#X - 1\}$, hence

$$\{f^n(1, \ldots, 1) \mid n \in \mathbb{N}\} = X$$

◀

5.1.2 Subsets

For each student we need to make a decision on which borrels they should attend. Again we need to go through all possibilities to see which choice is the best. The easiest way of doing this is by using the binary counter method. Given a set T we make a binary number b that has the same amount of bits as T has elements. A 1 in the i-th digit indicates that the i-th element of T is part of S and a 0 indicates that it is not part of S. By incrementing this number until all bits are set to 1 we get all possible subsets. For example if $T = \{12, 2, 4\}$ we get the following iteration as seen in table 1.

b	000	001	010	011	100	101	110	111
S	$\emptyset$	{12}	{2}	{2, 12}	{4}	{12, 4}	{2, 4}	{12,2,4}

Table 1 Subset iteration when using the binary counter method.

The downside of using this method in our case is that we would be doing a lot of unnecessary work. Assume that it is possible for a student to go to a set of borrels B. If

$C \subset B$ we know that C can not contain more borrels and therefore it will always be able to go to the borrels in C. It is therefore much better to use an iteration where bigger subsets come first (See table 2). The subset iterator will use a separate procedure called the fixed subset iterator which will iterate over subsets of the same size (see algorithm 2). The subset iterator will keep lowering the size of the sets until there are no elements left (see algorithm 3).

S	$\{12,2,4\}$	$\{12,2\}$	$\{12,4\}$	$\{2,4\}$	$\{12\}$	$\{2\}$	$\{4\}$	$\emptyset$

Table 2 Subset iteration when asserting that bigger subsets should come first.

We will first prove that the fixed subset iterator is correct, by proving that it will eventually pass through all subsets of the given size.

For a given finite set $C = \{c_1, \ldots, c_{\#C}\}$ we will define $f : \bigvee(C) \to \bigvee(C)$ as $f(A) = \pi_2 \circ \text{fSubIt}(A, C)$ where π_2 is the second projection.

▶ **Lemma 2.** *Define*

$$i^* := \max\{1 \leq i \leq \#C \mid c_i \in A \mid c_i \in A, \#(\{c_i, \ldots, c_{\#C}\} \cap A) \leq \#C - i\}$$

. Then

$$f(A) = C \backslash \{c_i\} \cup \{c_i, \ldots, c_{i+j}\} \backslash \{c_{i+j+1}, \ldots, c_{\#C}\}$$

for an $A \subset C$ unequal to $A \neq \{c_1, \ldots c_{\#A}\}$

Proof. We see that once the program has reached line 8 of algorithm 2 that $c_i \in A$ and that $j \leq \#C - i$. We also see that that j was inceremented every time $c_i \in A$, hence $j = \#(\{c_i, \ldots, c_{\#C}\} \cap A)$, so $\#(\{c_i, \ldots, c_{\#C}\} \cap A) \leq \#C - i$. We also that the i at line 8 is the biggest with this property, because the algorithm has started with $f = \#C$, and has decremented i since. Therefore we can say that $i = i^*$ at line 8.

Now we see that

$$f(A) = C \backslash \{c_i\} \cup \{c_i, \ldots, c_{i+j}\} \backslash \{c_{i+j+1}, \ldots, c_{\#C}\}.$$

◀

We will now give every $A \subset C$ the following index $i(A) := \sum_{i=1}^{\#C} 2^{i-1} \cdot 1_{c_i \in A}$. For any number $n \in \mathbb{N}$ we define $\#n$ as the amount of 1's in n's binary notation. For example $\#4 = 1$ and $\#10 = 2$. Note that $\#i(A) = \#A$ for any set subset A.

▶ **Lemma 3.** *Let $0 \leq n < 2^{\#C}$. Assume that there exists a smallest $0 \leq k < n$ such that $\#k = \#n$, then $i \circ f \circ i^{-1}(n) = k$*

Let $n = i(A)$ for a $A \subset C$ and $k = i(D)$ for a $D \subset C$. Take $k < l < n$. If $l < i^{-1}(A \cap \{c_1, \ldots, c_{i^*}) \}$ we see that $\#l > \#k = \#n$. We know that $\#(\{c_{i^*}, \ldots, c_{\#C}\} \cap A) \leq \#C - i^*$ and $\#(\{c_{i^*+1}, \ldots, c_{\#C}\} \cap A) \geq \#C - i^* - 1$, because i^* is the maximum. Therefore $\#(\{c_{i^*}, \ldots, c_{\#C}\} \cap A) = 1 + \#(\{c_{i^*+1}, \ldots, c_{\#C}\} \cap A) \leq \#C - i^*$, hence $\#(\{c_{i^*}, \ldots, c_{\#C}\} \cap A) = \#C - i^*$. If $l \geq i^{-1}(A \cap \{c_1, \ldots, c_{i^*}) \}$, there has to be a $i > i^*$ such that $c_i \in A$. Because i^* is the maximum we know that $\#(\{c_i, \ldots, c_{\#C}\} \cap A) > \#C - i^*$, hence $\{c_i, \ldots, c_{\#C}\} \subset A$ and therefore $\#l < \#n$. We conclude that l cannot have the property that $\#k = \#n$, hence k is the smallest value such that $\#k = \#n$.

Proof. We leave this as an exercise for the reader. ◀

▶ **Theorem 4.** *The fSubIt(A, C) procedure will eventually pass through all subsets of C that have the same size as A.*

Proof. Let $0 \leq m \leq \#C$. Define $X := \{0 \leq 1 < n < 2^{\#C} \mid \#n = m\}$ Take the set $A := \{c_{\#C}, \ldots, c_{\#C-m+1}\}$. We see that $i(A) = \max(X)$. Because X is finite and because of lemma , we see that $\{(i \circ f \circ i^{-1})^n(A) \mid n \in \mathbb{N}\} = X$. This means that $\{f^n(A) \mid n \in \mathbb{N}\} = i^{-1}(X) = \{A' \subset C \mid \#A' = t\}$. Hence we see that all subsets of length t are passed by the fixed subset iterator. ◀

5.2 Base case

Now we will provide an algorithm for the base case. You can find the algorithm in the appendix (See algorithm 6).

The algorithm will return s, o and v. s_h for every borrel h is the time where borrel h starts. For every student i and for every obligation j of student i, we define $o_{i,j}$ as the set of timeslots that obligation j should be performed. v is the the value of this case, which is the sum of the amount borrels that every student is able to attend. The s^*, o^* and v^* variables represent the best case found so far.

The algorithm works by iterating of all possible borrel positions as discussed in section 5.1.1. For every student it will then iterate over borrel subsets as discussed in section 5.1.2. The ChooseObl procedure is used to check if it can attend all borrels and to provide the times the student will be working on the obligation. Whenever it finds a borrel subset it can attend, the loop ends, because all subsets that come afterwards will not have a greater size.

The choose obligation procedure works by having a set U of all the taken timeslots. It first adds all the borrels and returns false if there is an overlap. It will then use the fixed subset iteration to look at all the possible times for the obligations and seeing if it can find a case where there are no overlaps.

5.3 Social Matrix algorithm

In the original problem every borrel attendance had the same value. Now we will introduce a social matrix. This social matrix will tell us how much students like each other. The value is calculated by taking the vector V_h defined as

$$V_{h,i} = \begin{cases} 1 & \text{if student } i \text{ is present at borrel } h \\ 0 & \text{if student} i \text{ is not present at borrel h} \end{cases}$$

and multiplying it with the social matrix:

$$v = \sum_{h=1}^{m} V_h^T \cdot M \cdot V_h.$$

Algorithm 8 is the algorithm for the case with social matrix.

6 Complexity

We will now calculate the time complexity of the borrel algorithm. We see that we first iterate over all possible borrel positions. We know that there are $\prod_{h=1}^{m}(t - b_h + 1)$ possible borrel positions. We then loop over the n students. Within the student loop we loop over al borrel subsets. In the worst case we will need to loop 2^m times. For every borrel subset we will need to check if the student can attend these borrels. In the chooseObl procedure we first loop over all the borrels wich takes m time and afterwards we will be looping over all the fixed size subsets of the obligations. We know that for any student i and obligation j of student i, there are ${}_{(d_{i,j}-r_{i,j}+1)}\mathrm{C}_{p_{ij}}$ subsets of size $p_{i,j}$. Within this loop it will need to go throug all obligations to check if there is a overlap. The time complexity becomes

$$\mathcal{O}\left(\left(\prod_{h=1}^{m} t - b_h + 1\right) \cdot n \cdot 2^m \left(m + \sum_{i=1}^{n}\prod_{j=1}^{\ell_i} {}_{(d_{i,j}-r_{i,j}+1)}\mathrm{C}_{p_{ij}} \cdot \ell_i\right)\right).$$

7 Experiments

In this section, we present computational results from our algorithm, which has optimally solved the base problem. Our primary focus is on the efficiency of our algorithm, as detailed in the algorithm section, where we demonstrate that it consistently delivers optimal solutions. Our approach utilizes a brute-force method enhanced with strategic shortcuts, making it a robust benchmark for other researchers. These results will help determine whether alternative strategies can outperform our exhaustive search for the optimal solution.

In table 3 you see the runtime of every instance.

Name of instance	Time in seconds
Example	0.007611
Group a	0.016793
Group b	0.707175
Group c	0.000965
Group d (1)	0.000245
Group d (2)	0.112592
Group d (3)	0.827414
Group e	0.357478
Group f (many)	0.323208
Group f (overlap)	0.003717
Group g	0.000751
Group h	0.000779
Group i (Overlapping Borrels)	0.003077
Group i (Online Destroyer 1)	0.000045
Group i (Online Destroyer 2)	0.000050
Group i (Simple pre-emption)	0.000142

Table 3 Runtime of the algorithm on all the instances. **CPU:** Intel i5-7200U (4) 3.100GHz, **OS:** Manjaro Linux x86_64

8 Conclusion

In this paper we have established the NP-hardness of the borrel scheduling (with social matrix) problem. This result shows the intrinsic difficulty of solving the problem even in an offline setting. We have also provided a brute force approach to solving the offline problem, with experiments showing the performance of this algorithm. These performance statistics may be used as benchmarks for researchers looking to create a more refined offline algorithm. For the online variant of the problem we have proven that no algoritm can achieve constant competitiveness in a number of different settings. With this we have shown that even with multiple restrictions to the problem, online algorithms will still provide poor attendence.

Our findings provide significant theoretical insights into the landscape of the borrel scheduling problem. Future work may investigate heuristic based offline algorithms to solve the offline problem more efficiently or seek specialized algorithms that perform well under specific conditions of the problem.

9 Appendix

Algorithm 1 Borrel iteration

```
1: procedure BORRELIT(t, m, b, s)
2:     for h ← 1, m do
3:         s_h ← s_h + 1
4:         if s_h + b_h − 1 ≤ t then
5:             return (False,s)
6:         else
7:             s_h ← 0
8:         end if
9:     end for
10:    return (True,s)
11: end procedure
```

Algorithm 2 Fixed Subset iterator.

```
1: procedure FSUBIT(sub,set)
2:     i ← #set
3:     for j ← 1,#sub do
4:         while set_i ∉ sub do
5:             i ← i − 1
6:         end while
7:         if j ≤ #set − i then
8:             sub ← sub\{set_i} ∪ {set_i, ..., set_{i+j}}\{set_{i+j+1}, ..., set_{#set}}
9:             return(False,sub)
10:        end if
11:        i ← i − 1
12:    end for
13:    return(True,{set_1, ..., set_{#sub})
14: end procedure
```

Algorithm 3 Subset iterator.

```
1: procedure SUBIT(sub,set)
2:     if #sub=0 then
3:         return(True,set)
4:     end if
5:     if ( thenend)
6:         return(False,{set_1, ..., set_{#sub-1})
7:     else
8:         return(False,sub)
9:     end if
10: end procedure
```

Algorithm 4 Checks if the current obligation times are valid.

```
procedure OBLIGATIONOVERLAP(t,obl,k,r,p,d,U):
    V ← ∅
    for j ← 1, k do
        if (U ∪ V) ∩ obl_j = ∅ then
            V ← V ∪ obl_j
        else
            return True
        end if
    end for
    return False
end procedure
```

Algorithm 5 Choose obligation times.

```
1: procedure CHOOSEOBL(t,borSub,b,s,r,p,d,k)
2:     U ← ∅
3:     for j ← 1, k do obl_j ← {r_j, ..., r_j + p_j − 1}
4:     end for
5:     for h ← 1, #b do
6:         A ← s_h, ..., s_h + b_h − 1
7:         if A ∩ U ≠ ∅ then
8:             return(False, obl)
9:         end if
10:        U ← U ∪ {s_h, ..., s_h + b_h − 1}
11:    end for
12:    loop ← True
13:    while loop do
14:        if ¬OBLIGATIONOVERLAP(t,obl,k,r,p,d,U)) then
15:            return(True, obl)
16:        end if
17:        for i ← 1, k do
18:            (end,obl_j) ← FSUBSIT(obl_j, {r_j, ..., d_j})
19:            loop ← ¬end
20:            if end then
21:                obl_j ← {r_j, ..., r_j + p_j − 1}
22:            else
23:                break
24:            end if
25:        end for
26:    end while
27: end procedure
```

Algorithm 6 Finds the optimum borrel times.

```
procedure INITIALSTATE(t, m, b, n, I, p)
    for h ← 1, m do
        s_h ← 0
    end for
    for i ← 1, n do
        for j ← 1, ℓ_i do
            o_{i,j} ← {r_{i,j}, ..., r_{i,j} + p_{i,j} − 1}
        end for
    end for
    v ← 1
    return(s,o,v)
end procedure

procedure BORREL(t, m, b, n, I, p, ℓ)
    (s, o, v) ← initialState(t, m, b, n, I, p, ℓ)
    (s*, o*, v*) ← initialState(t, m, b, n, I, p, ℓ)
    while True do
        v ← 0
        for i ← 1, n do
            borSub ← {1, ..., m}
            vS ← 0
            while True do
                (canAttend,obl) ← CHOOSEOBL(t, borSub, b, s, r_i, d_i, p_i, ℓ_i)
                if canAttend then:
                    vS ← #borSub
                    o*_i < −obl
                    break
                end if
                (end,borSub) ← SUBSIT(borSub,1,...,m)
                if subsIt(borSub) then
                    break
                end if
            end while
            v ← v + vS
        end for
        if v* < v then
            (s*, o*, v*) ← (s, o, v)
        end if
        (end,s) ← BORRELIT(t, m, b, s)
        if end then
            break
        end if
        return(s,o,v);
    end while
end procedure
```

Algorithm 7 Calculate the value using the social matrix

```
procedure CALCVALUE(borSub, M, n, m)
    v ← 0
    for k ← 1, m do
        for i ← 1, n do
            if k ∈ borSub_i then
                for j ← 1, n do
                    if k ∈ borSub_j then
                        v ← v + M_{i,j}
                    end if
                end for
            end if
        end for
    end for
    return(False)
end procedure
```

Algorithm 8 Borrel with social matrix

procedure BORREL($t, m, b, n, I, p, M, \ell$)
  $(s^*, o^*, v^*) \leftarrow$ INITIALSTATE(t, m, b, n, I, p, ℓ)
  $(s^{**}, o^{**}, v^{**}) \leftarrow$ INITIALSTATE(t, m, b, n, I, p, ℓ)
  while True **do**
    $v^* \leftarrow 0$
    for $i \leftarrow 1, n$ **do**
      borSub$_i \leftarrow \{1, ..., m\}$
    end for
    end $\leftarrow$ False
    while ¬end **do**
      validSubsets $\leftarrow$ True
      for $i \leftarrow 1, n$ **do** (canAttend,o_i) $\leftarrow$ CHOOSEOBL(t, borSub, $b, s, r_i, p_i, d_i, \ell_i$)
        if ¬canAttend **then**
          validSubsets $\leftarrow$ False
          break
        end if
      end for
      if validSubsets **then**
        $v \leftarrow$ CALCVALUE(borSub, M, n, m)
        if $v > v^*$ **then**
          $o^* \leftarrow o$
          $v^* \leftarrow v$
        end if
      end if
      for $i \leftarrow 1, n$ **do**
        (end,borSub$_i$) $\leftarrow$ SUBSIT(borSub$_i$, $\{1, ..., m\}$)
        if end **then**
          borSub$_i \leftarrow \{1, ..., m\}$
        end if
      end for
    end while
    if $v^{**} < v^*$ **then**
      $(s^{**}, o^{**}, v^{**}) \leftarrow (s^*, o^*, v^*)$
    end if
    (end,s^*) <- BORRELIT(t, m, b, s^*)
    if end **then**
      break
    end if
    return(s, o, v);
  end while
end procedure