Candidate Mini-Project 1

The problem statement is ambiguous (for example, it does not say what assigning a block to a space means), so we interpreted it in a way that makes sense:

> A block $B := (t_{\text{start}}, t_{\text{end}}, \Delta t, W)$ is an open interval of length $t_{\text{end}} - t_{\text{start}}$ that can be translated by at most Δt to each side from $(t_{\text{end}}, t_{\text{start}})$, so it must be contained in the interval $(t_{\text{start}} - \Delta t, t_{\text{end}} + \Delta t)$.
>
> Forbidden zones are intervals of the form $(t_{\text{start}}, t_{\text{end}})$.
>
> Our goal is to write a program that takes a set of blocks B and forbidden zones F, and returns a placement of some subset of the blocks $C = \{C_i\}_{i=1}^{k} \subseteq B$, satisfyint the following requirements:
>
> - Each block C_i in C has length $B_i.t_{\text{end}} - B_i.t_{\text{start}}$ to its corresponding interval in B, and is contained in the interval $(B_i.t_{\text{start}} - B_i.\Delta t, B_i.t_{\text{end}} + B_i.\Delta t)$
> - No two intervals in C intersect
> - No interval intersects a forbidden zone
> - The sum $\sum_{i=1}^{k} C_i.W$ is as high as possible, given the conditions above are satisfied.

Added Assumptions:

We made the following assumptions:

- Each block and each forbidden-zone has a positive length (i.e. $t_{\text{start}} < t_{\text{end}}$)
- All the input parameters are floating point numbers (floats), and none are excessively small, large, close together, or far apart - we did not address potential overflow and rounding errors (for simplicity)
- The intervals are open (do not include their endpoints). This is not made explicit in the instructions, so we had to choose whether they're open or closed (or neither) arbitrarily, so this is what we chose. This allows intervals to share endpoints.
  Consequently, whether forbidden zones are open or closed does not effect the solution, so we treat them as open or closed as convenient in the solution.

Our solution:

The problem, as interpreted above, is NP-hard (we will explain why as an appendix at the end). Thus, we could not hope for an efficient solution that finds the optimum in general, so we provided the following two solutions:

- **Brute-Force deterministic solution:** This solution will always provide the optimal subset, but has super-exponential runtime (as a function of the number of blocks and forbidden zones in the input). It is not feasible to use for large inputs.
- **Stochastic approximation:** This solution uses a simple Markov Chain Monte-Carlo algorithm to efficiently find a good solution - though potentially not the true maximum.
  We do not have a guarentee of how good it is, but with some probablisty analysis it is probably possible to obtain one, and anacdotically it appeared to work well on test cases.

Both solutions rely on the subroutine doesFit, which takes a collection of blocks and disjoint forbidden zones as input, and determines whether there is a legal way to place all blocks.

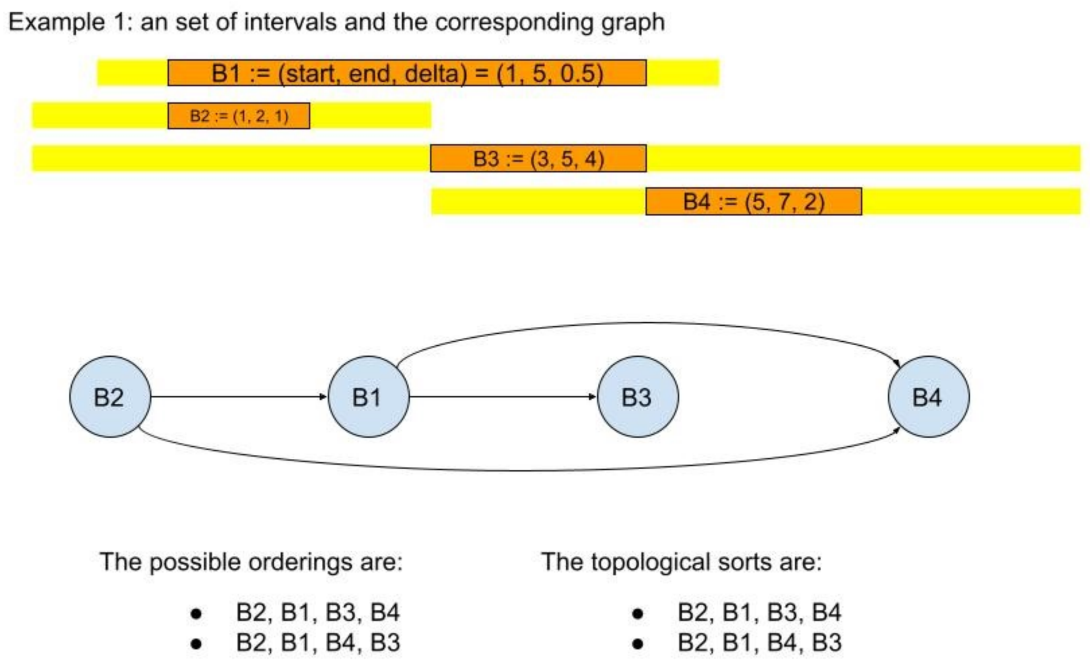
How doesFit works:

Given a collection of dijoint blocks (forbidden zones are treated as blocks with $\Delta = 0$), this method determines if all blocks fit in so they do not intersect.

Given an ordering of the blocks, it is easy to verify whether the blocks can fit in that order (place each block as early as possible without intersecting previous blocks - and see whether this can be completed for all blocks). However, iterating over all orders and check whether each one works is computationally infeasible. This method atempts to rule out most orders, so it only has to check a small set of orders.

How its done: We will build a directed graph that has each block as a vertex. If the latest possible start point of a block 1 is before the earliest possible end point of block 2, then block 1 must come before block 2. In this case, we will put an edge from block 1 to block 2 on the graph. For every pair of blocks, we will check whether we can add an edge from one to the other. Now, every ordering of the blocks that is possible must be a topological sorting of the graph. Now, to determine whether all the blocks can fit, we only have to iterate over all the topological ordering to see if one of those orderings work.

As a toy example, consider the set of intervals below:


Example 1: an set of intervals and the corresponding graph
B1 := (start, end, delta) = (1, 5, 0.5)
B2 := (1, 2, 1)
B3 := (3, 5, 4)
B4 := (5, 7, 2)
B2
B1
B3
B4
The possible orderings are:
B2, B1, B3, B4
B2, B1, B4, B3
The topological sorts are:
B2, B1, B3, B4
B2, B1, B4, B3

If we're lucky, there are only a few topological orderings and this function will run in polynomial time (and pretty eficiently!). If we're unlucky, there may be very many topological sortings. Thus, the function has an optional parameter to control its maximum number of iterations (used in the approximation solution). We conjecture that in non-adversarial cases, if one topological sort works than there are many topological sorts that work, and we'd likely find one with a reasonable number of iterations.

The Brute-Force Solution

The brute force solution iterates through every subset of the blocks and check whether thit fits using doesFit. It returns the maximum value achieved by a subsets that "fits" (i.e. can be legally placed without intervals intersecting each other, nor the forbidden blocks).

In particular, note that this solution calls doesFit without the optional parameter, so doesFit is guarenteed to return the correct conclusion on whether the intervals fit (though it is not guarenteed to be efficient...). Consequently, this solution is guarenteed to return the correct solution - though it is infeasable for anything but very small inputs. Also note that doesFit requires the forbidden-zones to be disjoint - so this method "disjointifies" the forbidden zones first thing.

The Approximation Algorithm[1]

The brute-force solution is infeasible for anything but the tyniest inputs. Thus, we use the following algorithm to approximate the solution, which takes in a set of blocks and forbidden zones as inputs, as well as the parameters NUM_ITERS, ITERS_PER_SUBSET, and TEMPRATURE for optimization of the algorithm (all can be controlled, but have reasonable default value in our implementation).

[1]This solution is a variation of a classis approximation algorithm for the Knapsack problem, which is taught in a course I TAed (task 5 in this HW assignment `https://courses.cs.washington.edu/courses/cse312/22au/homework/pset9.pdf`, for example)

Algorithm 1 MCMC for Blocks Maximization Problem

```
1: $n \leftarrow$ number of blocks in input
2: fz $\leftarrow$ disjointify(Forbidden Zones)
3: subset $\leftarrow$ vector of $n$ zeros, where subset is always a binary vector in $\{0,1\}^n$ which represents
   whether or not we have each block. (Initially, start with an empty subset of blocks).
4: best_subset $\leftarrow$ vector of $n$ zeros
5: for $t = 1, \ldots,$ NUM_ITER do
6:     $k \leftarrow$ a random uniform integer in $\{1, 2, \ldots, n\}$.
7:     new_subset $\leftarrow$ subset but with subset[$k$] flipped ($0 \to 1$ or $1 \to 0$).
8:     $\Delta \leftarrow$ value(new_subset) $-$ value(subset)
9:     if $\Delta > 0$ OR ($T > 0$ AND Unif(0, 1) $< e^{\Delta/\text{TEMPRATURE}}$) then
10:        if doesFit(new_subset $\cup$ fz, ITERS_PER_SUBSET) then
11:            subset $\leftarrow$ new_subset
12:    if value(subset) $>$ value(best_subset) then
13:        best_subset $\leftarrow$ subset
```

Informally, the algorithm works as follows:

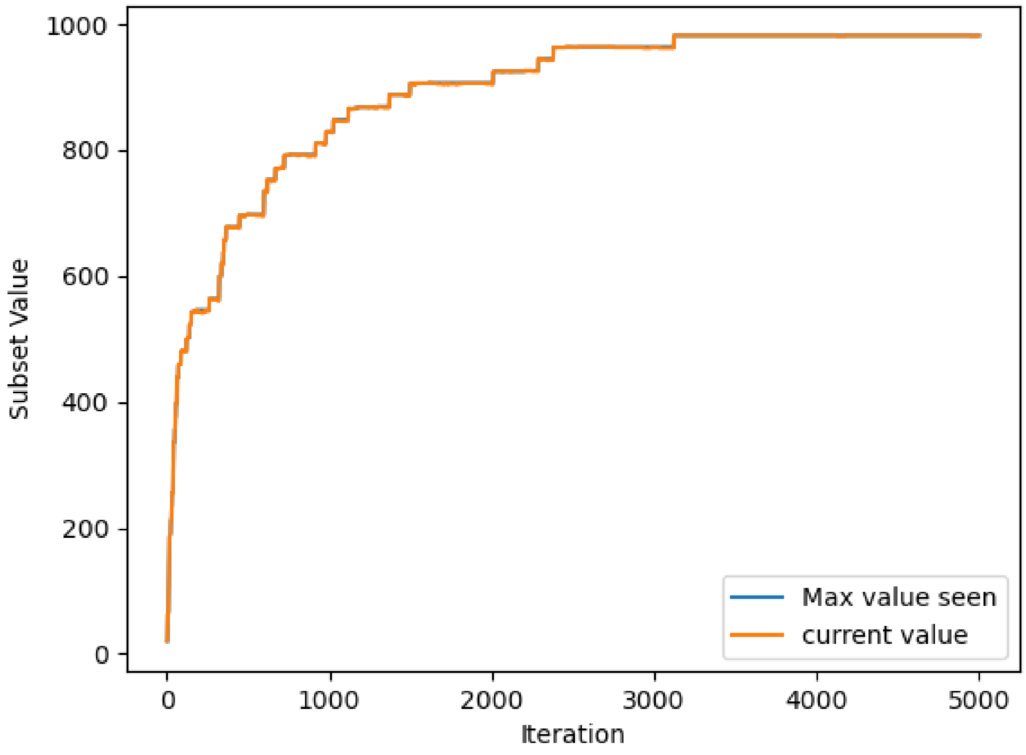
- start with an empty subset of the blocks.
- At each iteration, for NUM_ITER iterations, we create a new sucset that differs by 1 from the current set (either join an interval or remove an interval from the subset). If the new subset does not satisfy the requirements (i.e. it cannot fit without intervals intersecting themselves or forbidden zones), we discard it and cary on. If the new interval satisfies the fitting requirement, and has equal or higher value ($\sum W_i$) to our current interval, we switch our current interval to the new interval. Lastly, if the new interval fits but has less value than ours, we switch to it with a certain probability, which is dependent on how much we lose by switching and the TEMPERATURE parameter.
- Meanwhile, we keep track of the best subset we have seen so far, and the output is the best subset seen after all iterations are complete.

Thus, in essence, the algorithm tests various subsets, each one derived from the one before, and is more likely to reach subsets that have a higher value. Empirically, with enough iterations and a good choice of TEMPRATURE, the algorithm is pretty successful for reasonable runtime.

The graph below shows the progress of the algorithm on one test case. The input in this case is the set of 100 blocks (represented as $(t_{\text{start}}, t_{\text{end}}, \Delta t, W)$)

$$B := \{(i, i+1, 1, 1)\}_{i=0}^{49} \cup \{(i, i+1, 1, 20)\}_{i=0}^{48}$$

with parameters NUM_ITERS = 5000, ITERS_PER_SUBSET = 10, and the default TEMPERATURE value. After running for about 20 seconds, the final approximation was 982, and the true maximum is 1002.



Justification that the original problem is NP-hard

We will show that this problem is no easier than the 0-1 Knapsack Problem (https://en.wikipedia.org/wiki/Knapsack_problem), or in other words the Knapsack problem can be reduced to the problem we're trying to solve.

Assume that there exist an efficient solution to the problem we are trying to solve. And consider an arbitrary instance of the Knapsack problem: a maximum total weight W and a set of n items each one with value $0 \leq v_i \leq W$ and weight $w_i > 0$. Now, we can construct a set of n blocks each one with value v_i and parameters $(t_{\text{start}}, t_{\text{end}}, \Delta t)_i$ such that the block has length w_i and can fit anywhere in the interval $(0, W)$. In other words, we find $(t_{\text{start}}, t_{\text{end}}, \Delta t)_i$ such that

$$\begin{aligned} t_{\text{end}} - t_{\text{start}} &= w_i \\ t_{\text{start}} - \Delta t &= 0 \\ t_{\text{end}} + \Delta t &= W \end{aligned}$$

or equivalently

$$\begin{aligned} \Delta t &= \frac{1}{2}(W - w_i) \\ t_{\text{start}} &= \frac{1}{2}(W - w_i) \\ t_{\text{end}} &= \frac{1}{2}(W + w_i) \end{aligned}$$

The solution to the Knapsack problem is the same as the solution to our problem with the n constructed blocks as input (and no forbidden zones). Thus, if there exist an efficient solution to our problem there exist an efficient solution to the Knapsack problem (which is NP-hard), and therefore our problem is also NP-hard.