CS 170 Project Reflection

Araav Patel, Sarthak Kamat, Vincent Lim

December 2021

1 Algorithm Description

Our final solution incorporated greediness, randomization, dynamic programming, reduction via approximation, and local search heuristics.

First we used a Dynamic Programming approach (details listed in section 2) to fill out a 2D table that answers the sub-problem: "Using only tasks 1..i and finishing by time T, what is the optimal ordering of igloos to visit?"

This sub-problem framing exhibits a bias towards tasks with earlier indices being completed first, since the recurrence relation only swaps out old tasks instead of pushing them back forward. To deal with this issue, we did the following:

- Sort by deadline, to get a reasonable heuristic on general ordering of tasks.
- Before sorting by deadline, try sorting by profit/unit time as a tie breaker.
- Add in dummy tasks that are due later and worth as much as their penalty to value, to approximate the exponential decay. We added dummy tasks $(p_i \cdot e^{-0.0170n}, d_i, t_i)$, where $n \in \{1, \ldots, 1440 - t_i\}$, and enforced that only one such task from a group of dummy tasks associated with a real task can be picked at once in the dynamic programming.
- Finally, we applied local-search to optimize our DP solution. While the profit of a neighboring solution (created by swapping two igloos in the ordering) is better than the original solution, perform a swap. Keep track of all neighboring solutions using a stack.

This sort of mixed problem-framing was superior to using any one approach, because it exploited the benefits of each approach, while limiting the drawbacks. Specifically, we made the dynamic programming approach the core logic of the algorithm, with modifications that always lead to strictly better solutions. The greedy tie-breaking always leads to a strictly better solution, but also holds intact the original deadline ordering. Dummy tasks help neutralize bias towards earlier tasks, by giving us the opportunity to add them later on, and create a discrete approximation for the penalty mechanism. And using local-search only at the end helped prevent the risk of exploting too much before exploring the solution space.

2 Other Approaches

2.1 Adhoc/Dynamic Programming Formulations

For the ad hoc algorithms, we tried algorithms in the following order to achieve our final outputs. Each algorithm listed below did better than the last, leading up to our final algorithm.

1. Greedy: we first tried a greedy algorithm that greedily picked the task with the highest profit/duration ratio. We also tried to take the best solution for each task with other greedy variables, such as deadline and duration. This did not perform well at all, landing us in the bottom 50% of the leaderboard.

2. Dynamic Programming: we simplified the problem to remove the exponential decay and formulated an optimal algorithm to the simplified problem via dynamic programming. Let $f(i,t) = max(f(i-1,t), f(i-1,t')$+tasks[i].benefit), where i represents task i, t represents the time at which the task i would end (0 through 1440), and t' is when the task i would start. If the task cannot start at a $t' > 0$, it means that the task duration was larger than the current time t or the task deadline. Then we had $f(i,t) = f(i-1,t)$. All elements of the 2-d dp table would be initialized to 0. This did reasonably well, but since it did not account for the profit decay (and instead was replaced 100% decay if completed after the deadline), this landed us around the top 30% of the leaderboard at the time of submission.

3. DP with exponential decay: Instead of giving full penalty to tasks done after the deadline, we incorporated the exponential decay. In the previous solution, we checked whether our task duration was longer than our current time t and the task duration. Now, if the task duration is longer than the task duration but still smaller than the current time t (meaning it can still be completed), we add in the penalty to create the new task benefit. Then, the recurrence relation from the previous part still holds and we can reuse that code. All other edge cases and initializations from the previous part hold true for this part.

2.2 LP Formulations

We also attempted formulating the problem as an ILP, solving with an LP solver and post-processing (via rounding) to reconstruct an integer solution. For all formulations below, let n represent the number of tasks, p_i, d_i, and t_i be the maximum profit, duration, and deadline of task i respectively. We attempted two LP formulations:

2.2.1 LP Formulation 1

Let x_i represent the time we finish task i. The LP is formulated as follows:

$$\begin{aligned} \max_x \ & \sum_{i=1}^{n} \min(p_i \cdot -0.0170(x_i - t_i), p_i) \\ \text{s.t. } & x_i - d_i \geq 0 && \forall i \\ & \max(x_i, x_j) - \min(x_i - d_i, x_j - d_j) \geq d_i + d_j && \forall i \neq j \end{aligned}$$

Intuitively, $\min(p_i \cdot -0.0170(x_i - t_i), p_i)$ is a linear approximation of the *worst case* decay, as the original exponential decay function is not concave. The constraint $x_i - d_i$ enforces that all tasks are completed after time $t = 0$, and the constraint $\max(x_i, x_j) - \min(x_i - d_i, x_j - d_j) \geq d_i + d_j$ enforces that the tasks do not overlap.

Unfortunately the LP did not work out well in practice, both in terms of compute time and the quality of the approximation, so we abandoned the idea, instead choosing to optimize our DP approach further.

3 Compute

We used free Google Cloud credits that are offered to all users to run a high-memory (64GB RAM) and multi-CPU (16-core) instance for multiprocessing solutions. The high compute power wasn't really necessary for our solution, which computer pretty quickly even on our local machines (under an hour). But it helped create a quick turn-around time for getting feedback on algorithm performance and working on improving it.