Algorithms for Intelligent Decision Making - Assignment 3

Group 28
Thalis Papakyriakou
Andrea Alfieri

March 2020

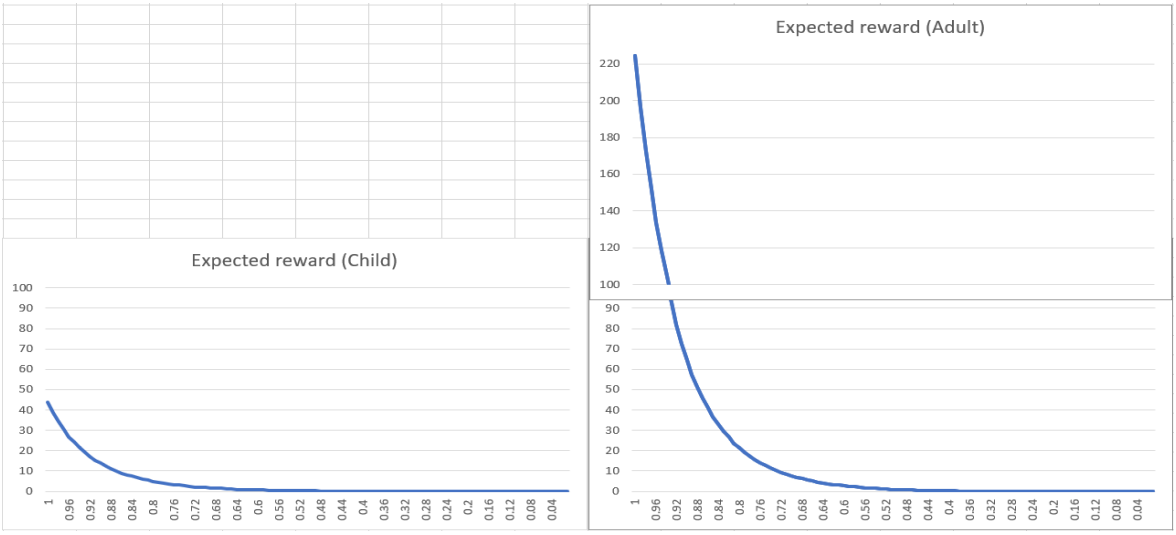


Figure 1: Expected Reward of starting state 0 after the execution of Value Iteration, for both Child and Adult CMDP's. The horizontal (x) axis represents the value of the discount factor.

1) Value iteration for single user

a – The code can be found in file PlanningAlgorithm.java, function "solveVI".

b – Child: 25.46668 - Adult: 119.1563

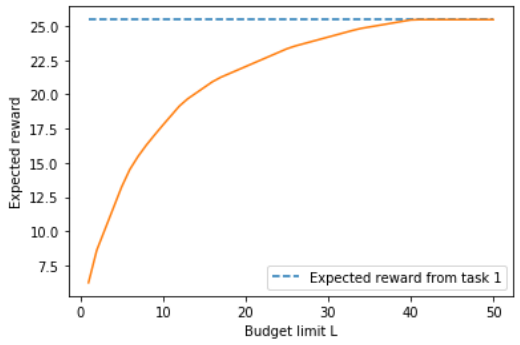
c – See Figure 1

d, e – As we can observe, the Expected Reward for the Adult is much higher than that of the Child, which was to be expected, since we calculated the optimal policies for both of them but at the same time, the potential reward of the Adult is by definition much higher than that of the Child (adults buy more expensive things).

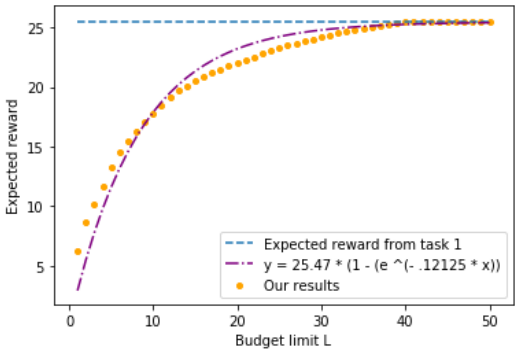
The effect of the discount factor γ is much greater for the Adult than for the Child; namely, we can observe that, as the discount factor decreases (which means the discount grows), the drop-rate of the Expected Reward is much higher for the Adult. Again, this was to expected, since by definition the Adult needs more time to be persuaded, meaning more steps, meaning less effective rewards as the step number grows. By receiving the reward at a later time than the Child, a small discount factor harms the received reward a lot more, and that explains the sudden drop in Expected Value.

Based on the above observations, we can also realise why the Expected Value approaches zero in both CMDPs (MDPs for now), when the discount factor is around 0.6: Even if the number of steps needed is not very big, a small discount factor (big discount) will quickly minimize any future reward by the time we receive it. This occurs because, per step, the exponent of the discount factor is increased by one. A small discount factor, raised to a high enough power (the step), will result in a multiplier so close to zero, that the reward of that step will also end up being zero, no matter its initial value. Here is the relationship between the Expected Value of the current and next step, for more clarity:

$$Q_{t+1}(s,a) = R(s,a) + \gamma * \sum_{s' \in S} P(s'|s,a) * \max_{a' \in A} Q_t(s',a')$$

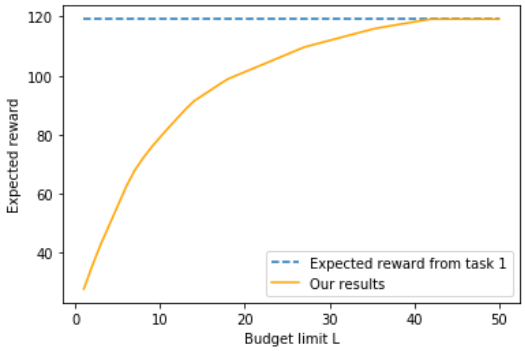


(a) Influence of the available budget L on the expected value - Child

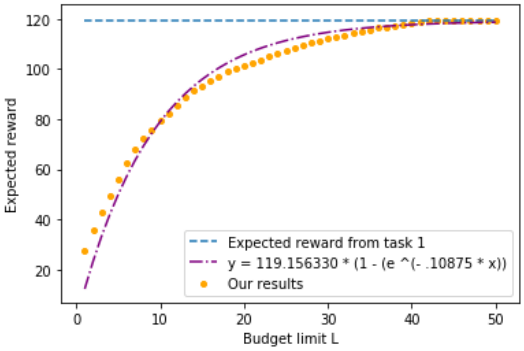


(b) Our attempt at modeling the curve - Child

Figure 2



(a) Influence of the available budget L on the expected value - Adult



(b) Our attempt at modeling the curve - Adult

Figure 3

2) Constrained planning for single user

a – To assign the cost to the CMDP we simply replaced *cmdp.assignCost(0, 0, 0)* with:

```
// Assign cost
for(int s = 0; s < cmdp.getNumStates(); s++)
        for(int a = 0; a < cmdp.getNumActions(); a++)
                cmdp.assignCost(s, a, 2*a);
```

b – For the optimal unconstrained policy, the expected discounted cost was:

Child: 40.233734 - Adult: 41.97915

c, d – As shown in Figures 2a and 3a, we can see how the expected reward for the constrained problem approaches the two asymptotes as the budget limit is increased. In fact, this reward-limit is the same as the expected reward we found for the unconstrained problem in task 1.b and is explainable by the fact that, as budget $L \to +\infty$, the constrained algorithm policy approximates the unconstrained one, since big limits is equivalent to the limits being lifted and the problem becoming an actual unconstrained one. Keep in mind, both algorithms calculate the optimal expected reward/policy.

Moreover, we tried to fit a negative exponential model to our curve and the best one we could find was $y = A(1 - e^{-\tau x})$, as shown in Figures 2b and 3b. This may suggest that the relation is, in fact, exponential. The model is considered by physicists to converge to the asymptote in $\approx \frac{5}{\tau}$ (some say 3 instead of 5, but we are not willing to participate in that debate), which also applies to both or our models with $40 < \frac{5}{\tau} < 48$.

3) Constrained planning for multiple users

In the Appendix, we provide a description of the algorithm executed in function "solve", which reformulates the problem into a Linear Programming model and then solves it using the Simplex Method. We figured it would be important to outline the main idea, as we understood it, since we could use arguments based on the algorithm's functionality to back up our answers in different questions. You do not have to read it, but we might refer to it in our answers.

Additionally, we eliminated a time-consuming redundancy in the algorithm, in the "construct policy" section, where the calculation of the divisor for normalization does not have to be included in the deepest for-loop, but can be moved outside of it.

a – The code can be found in file Homework.java, function "task4".

b, d – .

	Child	Adult
Expected Reward	17.7554	79.1637
Expected Cost	10.0000	9.9999
Total Expected Reward	96.9191	
Total Expected Cost	≈ 20	

(a) Separate planning

	Child	Adult
Expected Reward	8.8269	98.7113
Expected Cost	2.1385	17.8614
Total Expected Reward	107.5382	
Total Expected Cost	≈ 20	

(b) Multi-agent planning

c – The total expected cost for the Separate planning does not exceed the budget limit, and frankly, it couldn't: each agent gets his own model and half the original budget as his limit. Since the model strictly enforces, through a constraint, that the total cost of actions per agent may not surpass the personal budget limit, there is no way for the cost to exceed the agent's personal limit, and thus no way for the total cost to exceed the total limit.

e – The total expected cost for the Multi-agent planning does not exceed the budget limit, and frankly, it couldn't: both agents are merged into a model that no longer distinguishes between agents, but enforces a single cost constraint such that there is no way the total cost of all actions can surpass the budget limit in the resulting solution.

f, g – First off, we notice that both total expected costs are equal to the budget limit. This indicates that the budget limit is low, since, by directly correlating with the availability of effective actions, the solver cannot achieve any higher reward without breaking the constraint.

Like we said before, Separate planning creates a different model per agent, and enforces half the original budget limit to each. This, implicitly, enforces some kind of fairness amongst resource distribution per agent, but that is not necessarily a good thing: both Child and Adult will be exposed to a similar amount of advertising, but the Adult has much more potential to achieve high rewards, thus we may be hindering the Adult from achieving higher rewards by capping his budget limit to half the original. Additionally, the Adult needs more exposure to advertising to achieve the reward, and in this scenario, where there is a single, big reward achievable only after some steps, there is the danger of not reaching it at all with a small budget limit. Multi-planning, on the other hand, mixes both agents into a single model, and thus doesn't consider

any individual agent, only the total reward and cost, but is much slower when there are a lot of agents.

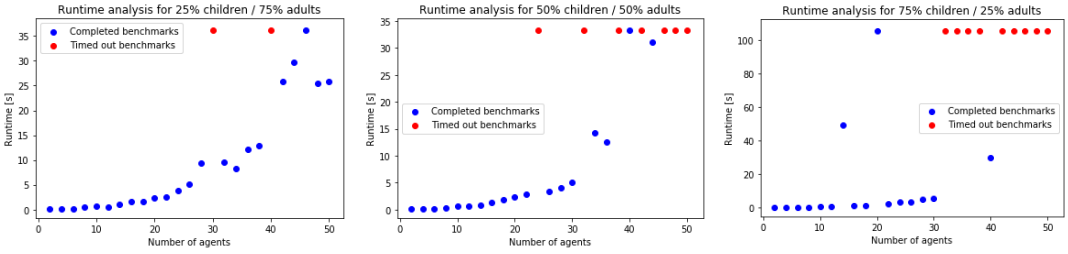
Given the above reasons, there is no absolute best planning for every situation: In cases like ours, of single rewards in the future, separate planning could have devastating effects for the hard-to-achieve-reward agents. Even in cases of frequent rewards, separate planning will hinder the agents with the most potential. Thus, we conclude that, in those cases, Multi-agent planning seems more beneficial, and that can also be seen by our experimental data. Separate planning could be really helpful in a scenario where we care more about fair resource distribution than the total reward, or in a scenario where the Multi-agent algorithm cannot scale for a lot of agents (since it creates a single, big model), and we are forced to do the planning in parallel through Separate planning and custom budget-limits per agent with testing.

4) Scalability of the constrained planning

a – As shown in Figure 4, the runtime analysis of the implementation suggests an exponential relationship with the number of agents, making the algorithm hard to run for a bigger size of the agent set. Moreover, some particular configurations (30 agents in figure 4a or 24 agents in figure 4b) seem to be harder to solve for the program. Overall, benchmarks with a higher portion of adults seem to be easier to solve and many solutions are found within the time limit (600 seconds), while many benchmarks with more than 30 agents go into timeout when children are the majority (figure 4c). This came in contrast with our initial expectations: we expected that more adults would equal more runtime, since adults need more steps to reach their reward, and they can utilize a bigger budget, which would mean more budget allocations possible towards finding the optimal one. We were wrong.

We suspect that the trick lies within the way the Simplex Method solver dismisses/rejects possible solutions: An adult's optimal reward is only found when he is allocated a lot of budget. A scenario with many adults means the total budget will be cut off fast with each new adult, and, if the solver is able to realise that a small budget for an adult will not lead to an optimal solution, then it can only concern itself with big allocations per adult. On the other hand, we can observe that the optimal reward is achieved when child agents receive a very small budget compared to the adults of the scenario. A scenario with many children would mean many combinations of small budget allocations are available for the child agents, and they are all possible contestants for being the optimal allocation since they approximate the actual optimal reward. This is our theory. The code for this analysis can be found in Homework.java, "task5".

We decided not to exclude the outliers in the plots, which might have been caused by the scheduling of our laptops' processors, in order to provide a more statistically significant analysis.



(a) Majority of adults (b) Equal sizes (c) Majority of children

Figure 4

b, c – The goal of this algorithm is to allocate a useful budget to each child/adult agent, and then determine a good policy per agent.

We presume the reward function is the same per agent in a group, and the main difference would be the transition-probability function.

We notice that, if we were to evenly distribute the budget to everyone, every agent would have a budget of 10. By running a few experiments using the solver, we can receive the budget-limit graphs for one random agent per group, like we did in section 2c. In the graphs, we notice that the growth-rate of the expected reward is at its highest up-until about budget=15 for both the child and the adult. These peaks can give us an intuition on whether the initial budget per agent, 10, is already more than enough for a certain agent to approximate his max reward. If so, we set the initial budget of that group of agents to their growth-rate peak. If not, we keep the initial value for now (10).

The next step is to repeatedly use Separate planning on all agents with the initial budgets we set for each. Each agent can run in parallel to the other, so this is not a time-consuming operation. For every run of the algorithm, we will remove some budget (proportional to the total available) from group 1, and evenly distribute it to group 2. If we notice that, per run, the total reward increases, we continue moving in this direction until we approximate a reward-limit. If we notice that, per run, the reward decreases, we start the same procedure in reverse, moving budget from group 2 to group 1. Obviously, the budget distribution per run won't be uniform, but that can easily be handled throughout the runs with a bit of sorting.

We implemented the above algorithm for the above instance (C:1388, A:2429). The "proof of concept" code can be found in Homework.java, "task6". The expected reward per run is presented below, along with the amount of agents of each group with a certain budget:

- All agents have budget 10 → 216933
- Child (budget 9: 1388), Adult (budget 10: 1041, budget 11: 1388) → 220432
- Child (budget 8: 1388), Adult (budget 11: 2082, budget 12: 347) → 223852
- Child (budget 7: 1388), Adult (budget 11: 694, budget 12: 1735) → 227023
- Child (budget 6: 1388), Adult (budget 12: 1735, budget 13: 694) → 230014
- Child (budget 5: 1388), Adult (budget 12: 347, budget 13: 2082) → 232535
- Child (budget 4: 1388), Adult (budget 13: 1388, budget 14: 1041) → 234374
- Child (budget 3: 1388), Adult (budget 14: 2429) → 236096
- Child (budget 2: 1388), Adult (budget 14: 1041, budget 15: 1388) → 236593
- Child (budget 1: 1388), Adult (budget 15: 2082, budget 16: 347) → 235934
- Child (budget 0: 1388), Adult (budget 15: 694, budget 16: 1735) → 234002

As we can observe, the optimal reward is achieved when all Child agents have a budget of 2, when 1041 Adults have a budget of 14, and when 1388 Adults have a budget of 15. Thus, **the expected budget to be allocated to the child group is 2776, and the expected budget to be allocated to the adult group is 35394**.

Now that we have decided the budget per agent, we can also determine the corresponding stochastic policy of each agent through the already calculated value functions. A simple last step would be to combine the policies into one joint policy as our solution.

An alternative, more complex, and optimal method to the Budget Allocation Problem (BAP) would be the one described in the paper "Budget Allocation using Weakly Coupled, Constrained Markov Decision Processes"[1]: The authors suggest formulating the LP relaxation of the IP model

[1] Craig Boutilier and Tyler Lu. "Budget Allocation using Weakly Coupled, Constrained Markov Decision Processes". In: *Proceedings of the 32nd Conference on Uncertainty in Artificial Intelligence* (2016), pp. 52–61.

of the multiple-choice knapsack problem (MCKP); it is easy to adapt the BAP problem to the equivalent MCKP problem. Using the simple, greedy algorithm of Sinha and Zoltners to solve the relaxation, the produced solution is actually an optimal one to the original BAP.

A Appendix - LP model algorithm

A. First, the objective function is created. There are a total of x = n * s_i * a_i decision variables in the objective function, with n:numOfAgents, s_i:numOfStatesOfAgent#i, and a_i:numOfActionsOfAgent#i. Each decision variable represents a state-action pair of an agent. Each decision variable has a coefficient equal to the reward that the corresponding agent 'i' would receive if he executed action a_i in state s_i. The objective is to maximize this function.

B. There are n * s_i flow-constraints, each representing the state of an agent. The constraints are equality ones, with a right-hand side value of 1 if the state represented is the initial one, and zero otherwise. Per constraint, there are 'x' decision variables, with their coefficients being complex enough for us to not understand the intuition behind them.

C. There is one cost-constraint, which has the cost of each state-action pair as the coefficient of the respective decision variable, forcing the 'x' decision variables to take values that result in a total cost lower than the budget.

D. There are 'x' non-negative-constraints, one for each decision variable, with a coefficient of 1 for the single represented variable. This forces all the decision variables to have non-negative values.

E. The Simplex Method is used to solve the above model, which results in a list of point-value pairs, as many as the initial decision variables we had. The value of a point, when normalized against the values of other points representing the same state and agent, represents the stochastic probability of that agent taking that action when in that state (the stochastic policy).

   Furthermore, the value of a point can act as a coefficient of the reward/cost of the corresponding decision variable. When added together, the reward/cost state-action pairs of an agent, along with their point-value coefficients, form the expected reward/cost of that agent.

B Appendix - Executing the program

After compilation, run "java Homework 1", where argument '1' is an example, and indicates the task you would like to be executed.