# Real-Time GA-Based Probabilistic Programming in Application to Robot Control

Alexey Potapov[1,2,3(✉)], Sergey Rodionov[3,4], and Vita Potapova[2,3]

[1] ITMO University, St. Petersburg, Russia
[2] St. Petersburg State University, St. Petersburg, Russia
[3] AIDEUS, St. Petersburg, Russia
pas.aicv@gmail.com, elokkuu@gmail.com, astroseger@gmail.com
[4] Aix Marseille Université, CNRS, LAM (Laboratoire d'Astrophysique de Marseille) UMR 7326, 13388 Marseille, France

**Abstract.** Possibility to solve the problem of planning and plan recovery for robots using probabilistic programming with optimization queries, which is being developed as a framework for AGI and cognitive architectures, is considered. Planning can be done directly by introducing a generative model for plans and optimizing an objective function calculated via plan simulation. Plan recovery is achieved almost without modifying optimization queries. These queries are simply executed in parallel with plan execution by a robot meaning that they continuously optimize dynamically varying objective functions tracking their optima. Experiments with the NAO robot showed that replanning can be naturally done within this approach without developing special plan recovery methods.

**Keywords:** Probabilistic programming · Optimization queries · Genetic algorithms · Robot planning · Replanning

## 1 Introduction

It is frequently assumed that AGI systems should not only perform some abstract reasoning, but should also be able to control some body achieving goals in real environments. Even if a cognitive architecture wasn't initially developed specifically for this purpose, natural desire to try applying it for e.g. robot control can arise after its maturing.

Robot control tasks are quite interesting since they require both planning and reactive control for achieving a goal in dynamic environments. Symbolic architectures are usually good for planning, but realization of reactive behavior within them is awkward, while emergent architectures are usually better suited for reactive control. Thus, hybrid solutions are developed to solve the problem of plan recovery [1].

We are developing an approach to AGI using probabilistic programming as the starting point. In another paper [2], we explain motivation behind this approach and discuss how traditional probabilistic programming languages (PPLs) should be


B. Steunebrink et al. (Eds.): AGI 2016, LNAI 9782, pp. 95–105, 2016.
DOI: 10.1007/978-3-319-41649-6_10

extended in order to become usable as a framework for development of cognitive architectures. However, this discussion addresses questions regarding reasoning and learning, but not regarding controlling (embodied) agents. At the same time, most general-purpose PPLs support only computationally expensive queries with unpredictable execution time (so they are well-suited for planning, but not for reactive control). This issue might be called purely technical, but nevertheless it is quite important. Indeed, taking limitation of resources into account is considered essential for AGI research [3].

In this paper, we describe how PPLs with optimization queries based on genetic algorithms (GAs) can be used in robot planning and can support replanning naturally almost without modifications. To do this, we execute optimization queries from our lightweight C++ probabilistic programming engine to perform planning simultaneously with executing current best plan by the real NAO robot using its SDK functions. Experiments show that continuing optimization can track optimum of the objective function, which can considerably shift due to changes of the environment or inaccurate execution of actions by the robot.

## 2 Lightweight Implementation of GA-Based Optimization Queries in Probabilistic Programming

Conditional inference over probabilistic programs can be carried out using program traces [4] containing all made random choices, which can be modified during re-interpretation of a probabilistic program. The same approach can be used in the case of optimization queries [5]. However, it is not very fast since it requires PPL to be interpreted.

If we are using optimization quires in a somewhat restricted way considering probabilistic programs as functions of random variables, then such queries don't necessarily need access to the code of a probabilistic program. In this case, probabilistic program can be written as a function (e.g. as a virtual method of inference class) directly in the reference programming language (C++ in our case) and compiled. Such approach also simplifies integration with existing libraries, e.g. OpenCV or NAO SDK.

In our implementation, probabilistic program is a function that receives an object "rng" as a parameter from which it samples all random variables. In order to be able to perform inference (optimization) we should be able to uniquely identify all random calls. For example, we can name all random calls by unique string tags. However, it is inconvenient in the case of programs, in which some random variable is sampled several times in the same context (like in example considered below). We adopt a little bit different approach. We introduce named sources of random variables, which can be used several times. So, in our case, sources represent named sequences of random values of the same type.

Let us consider simple example.

```
double res = 0;
while(rng.flip("A")) res += rng.gaussian("B", 0, 1);
return fabs(res - 5.);
```

For the inference algorithm, this code will be a function of $A_1, A_2\ldots$, where $A_i$ is the result of $i$-th call to `rng.flip("A")`, and $B_1, B_2,\ldots$, where $B_i$ is the result of $i$-th call to `rng.gaussian("B", 0, 1)`. As easy to see, the number of requests to sources "A" and "B" can be different for different runs of the random program.

We use the following types of random numbers and corresponding sampling functions:

- `flip` with parameter `p` returns true with probability `p`;
- `randint` with parameter `n` returns equally probable random integer from [0, $n$–1];
- `gaussian` with parameters mean, sigma returns normally distributed random value.

We assume that the type of the random variable is fixed and cannot be changed. On the other hand, parameters of random functions can vary in different runs of the random program (parameters of random functions can be functions of random variables). Moreover, similar to the considered simple example, the set of used random variables can vary. These features should be taken into account by the inference algorithm.

We utilize GAs to perform optimization of values of random variables as follows. A candidate solution in the population is a set of values of random variables sampled in the program. Each random value is associated with a unique tag (name of random source plus index), type of a random variable and parameters of this random variable.

GA executes the given function with random choices, which return value is interpreted as the fitness value for the candidate solution represented by specific sampled values of random variables, and control values returned by `rng` using genetic operators. `flip, randint`, etc. act as conventional pseudo-random functions while the first generation of candidate solutions is produced, but their behavior is changed for children.

Crossover is implemented in the most trivial way: random exchange of random values with the same tag. So if tag is presented in the both parents, then random value is taken randomly from one of the parents. All random values presented only in one of the parents (such a situation can happen for example in the considered simple example) are inherent by descendant. Mutations are implemented also the simplest way by varying the values of random variables. So, GA simply executes the compiled fitness-function with random choices many times controlling the values returned by basic random functions called via `rng` and trying to find the set of values that corresponds to the optimum of the fitness-function.

## 3 Planning as Probabilistic Programming

Planning can be naturally expressed in PPLs with both conditional and optimization queries. To do this, one needs to specify a generative model for plans as sequences of random possible actions with random parameters. Then, an environment model should be available for predicting outcomes of plan execution. In principle, PPLs can be used to learn this model or to make inference over stochastic models. Conditional or optimization queries with the condition of successful goal achievement or with the objective function evaluating proximity to the goal will find examples of suitable plans. There is no need to program the robot how to act in each of numerous possible situations, so to change robot's behavior we need only to specify a new goal – not to reprogram the robot.

In Turing-complete PPLs, generative models (of plans in our case) can readily include conditions, cycles, etc. Thus, PPLs provide a powerful tool for planning.

We developed a test planning system for the NAO robot using small subset of its possible actions including

- *wait* (call `qi::os::msleep(t)`, where `t` is the action parameter defining wait time in milliseconds);
- *walk* (call `ALMotionProxy::moveTo(x, y, 0)`, where `x` and `y` are parameters indicating how far in meters the robot should move);
- *turn* (call `ALMotionProxy::moveTo(0, 0, a)`, where `a` is the rotation angle);
- *posture* (pose changing using `goToPosture` command with the possible values "`StandInit`", "`Crouch`", "`Sit`", "`SitRelax`", "`Stand`", "`StandZero`").

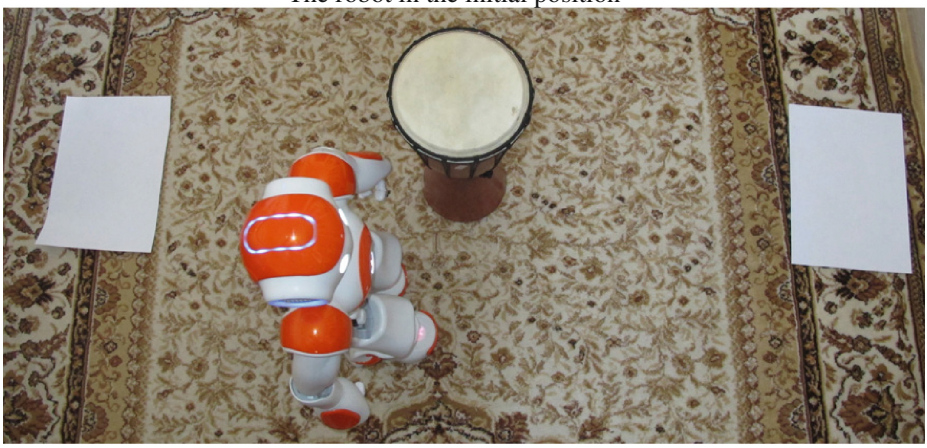
The generative model consists simply in generating a random list of random actions and random values for action parameters. Semantically different random values are generated from different random sources. For example, waiting time `t` for *wait* action is generated as `rng.randint("wait", 1000)`, and walking parameters `x` and `y` are generated as `rng.gaussian("walk", 0., 1.)`.

At first, we considered such goals as approaching a position with specified relative coordinates. Objective function (or fitness-function for GAs) was calculated as the distance from the expected position after plan execution to the specified target plus time (or efforts) penalty. In order to evaluate it, plan execution should be somehow modeled.

We didn't use detailed 3D model of the robot's body, although this is possible. Instead, the model included only robot's coordinates and pose (sitting, standing, etc.), and information of how they are expected to change after executing listed actions.

Obstacles were detected using sonars, and their positions were taken into account during modeling robot's movement. That is, if the expected robot's path crossed the detected object, collision was modeled by forcing the robot pose to "`Crouch`" and the robot coordinates to those of the obstacle (the robot wasn't explicitly programmed to avoid obstacles, since either to collide with obstacles or not depends on goals). The robot was not also programmed to stand up for walking, but its expected coordinates were changed while simulating *walk* action only if its pose corresponded to standing).

The robot in the initial position



The robot moving diagonally to bypass the obstacle

Robot in the final position

**Fig. 1.** The robot executing found plan

Our GA-based optimization query managed to find plans consisting of unknown number of actions with unknown real-valued parameters. If there were no obstacles, and the initial position was sitting, than the robot guessed to stand up first and then to move directly to the goal location. If there was an obstacle, found plans bypassed it usually using the shortest route (see Fig. 1).

Of course, there are many problems with following fixed plans even in the considered simple task. Real robot's movement will never precisely correspond to the expected movement. The robot detects only the first obstacle using sonars, but there can be other obstacles encountered during bypassing the first one. Moreover, the environment can be dynamic, so obstacles or targets can change their coordinates. Agents blindly following even genial plans can act very stupid.

Having infinite computational resources, one could construct new optimal plan from scratch after performing each elementary action (as it is done in AIXI [6]). However, this works only in theory and cannot be afforded in practice. Consequently, rather complicated methods for error detection, plan recovery and replanning methods are being developed [7–9].

Here, we don't want to develop specific solutions for robot plan recovery, but consider the question what minimal modifications to the probabilistic programming framework are necessary to support plan recovery in the same sense as conventional PPLs provide a solution of the planning problem.

## 4 Simultaneous Plan Optimization and Execution

*Changing fitness-function during optimization.* What will happen if the fitness-function is changed during its optimization by GAs? If the population of candidate solutions has not yet converged or if the changes are small, GAs will find new optimum. One can also slightly modify GAs to adaptively control the mutation rate depending on changes of the population fitness, and can introduce the mechanism of recessive and dominant alleles to keep gene diversity.

At first, we tried to create separate thread for the planner and to modify data for the objective function (namely, coordinates to be reached by the robot) outside the planner while it was optimizing this function. Not very surprisingly, the best solution in the current population tracked changes in the objective function.

Consider an example. Let the robot's goal be to move one meter forward (and there is an obstacle in front), and this target is moving at the speed of 10 cm/s. Initial candidate solutions in the first population are bad. They almost don't help the robot to approach the target. However, solutions rapidly become better from population to population, and almost precise solution appears after 0.5 s. This solution will be different in different runs, but typical solution will contain three commands such as *posture* (stand), *walk* (0.1, 0.27), *walk* (0.93, −0.25). This is the plan for achieving not the original goal, but already the modified goal. Candidate solutions in consequent populations are tracking the changing goal. E.g. after 6 s the goal will be to move 1.6 m, and the typical plan will be *posture* (stand), *walk* (0.173, 0.269), *walk* (1.426, −0.268).

Consider another type of change in the environment. Let the obstacle be removed at some moment of time. The result will depend on the moment of removal. If the obstacle is removed after 3 s, the plan before removal will be like *posture* (stand), *walk* (0.036, −0.352), *walk* (0.964, 0.352). This is nearly optimal plan. However, after another 3 s of the continuing optimization process the plan will be *posture* (stand), *walk* (0.035, −0.331), *walk* (0.965, 0.331). The value of the fitness-function doesn't change after obstacle removal, but this plan becomes suboptimal. Since necessary efforts are taken into account in the objective function, the best plan should correspond to the shortest distance to the target.

Apparently, the reason of this result is convergence of population of candidate solutions after 3 s of GA execution, and small improvements of the final plan are due to mutations (which speed is not enough to achieve the optimal plan). Indeed, if the obstacle disappears after 0.5 s, the solution found before this event will be in general correct, but imprecise, e.g. *posture* (stand), *walk* (0.26, 0.32), *walk* (0.82, −0.28). However, the final solution will be *posture* (stand), *walk* (0.218, 0.002), *walk* (0.782, −0.002). The population of candidate solutions has not converged yet, so it has more capabilities of adapting to the changes of the fitness-function. The final plan is almost optimal, although it realizes forward walk in two commands. Possibly, environment changes causing large changes in the value of the fitness-function and requiring insertion of some steps in the plan will be much more difficult to track.

However, it appears that since the goal's achievement precision is the largest term of the fitness-function, it is optimized first. Then, efforts of achieving this goal are optimized. Thus, candidate solutions corresponding to different plan sizes and to not straight routes to the target remain until strong convergence of the population, and the route bypassing appeared obstacle was found in most cases.

Of course, there can be more difficult replanning tasks, and non-conventional evolutionary computation methods natively supporting optimization of dynamic fitness-function might be necessary, but we can state that GA-base probabilistic programming even not initially design for concurrent optimization is applicable at least to some extent.

However, there are specific additional difficulties, when one tries to execute a plan simultaneously with its proceeding optimization.

*Evaluating partially executed plans.* One immediate issue with optimization of plans during their execution is that execution itself causes changes in the environment. If we simply continue the optimization process externally changing the fitness-function in accordance with the actual environment state (e.g. including position of the moving robot) than this process will need to blindly adjust the plan (e.g. from *walk* (1.0, 0.0) to *walk* (0.5, 0.0) after the robot has walked 0.5 m). Of course, adjustment can be necessary, but only because of possible difference between expected and real positions – not because of difference between initial and current positions. Is it necessary to change the inference engine to solve this problem? For example, one can try modifying all candidate solutions inside GA removing the first action from each of them after a real action has taken place. We believe that this is not necessary, and the environment model (simulation of plan execution) should mostly account for peculiarities of simultaneous plan optimization and execution.

The plan execution procedure should inform the plan simulation procedure about already completed part of the plan, and this part should simply be skipped during simulation. Since actions are non-atomic, partially executed actions should be partially skipped during simulation. This leads to main imprecision of simulation, because final position after action execution is known better than intermediate positions.

At first, we checked that the value of the objective function calculated for the plan being executed doesn't increase dramatically. We used getRobotPosition function from the NAO SDK (motion proxy) to get current robot coordinates (estimated by odometry), from which plan simulation starts. For example, initial coordinates in the robot reference system could be (2.49, −0.82, 0.23), where the first two are (x, y) in meters and the last one is the orientation angle. The goal then could be (3.49, −0.82), and the intermediate position during bypassing an obstacle could be (3.00, −1.20, 0.36), and the final position could be (3.42, −0.75, 0.36) meaning that the robot missed the target by 0.1 m. The problem is that the robot changes its orientation after execution of `Stand`, and continues to execute the plan that was designed in supposition that `Stand` action doesn't change robot coordinates and orientation.

This discrepancy is directly reflected in the value of the objective function calculated for the partially executed plan. After execution of `Stand`, it becomes 0.18 instead of 0.06 (initial non-zero value corresponds to penalty for efforts needed to perform all actions in the plan) meaning that the expected final position error is about 0.12 m. The value of the objective function evaluated during plan execution including partially executed actions doesn't go beyond 0.22 meaning that simulation of partially executed plans is generally correct. After executing the plan, its value becomes 0.16 corresponding to 0.1 m error (0.16–0.06) equal to the distance between the goal (2.49, −0.82, 0.23) and the final position (3.42, −0.75, 0.36) estimated from odometry.

Appearance of unexpected obstacle automatically leads to stepwise increase of the estimated objective function. For example, in the situation shown in Fig. 2, it increased by 0.45 meaning that further execution of the plan was expected to lead to collision and the simulated final robot position corresponded to the position of the obstacle.

*Simultaneously executing and optimizing plans.* Now, after we checked possibility of optimizing dynamically changing fitness-functions, and introduced correct evaluation of partially executed plans, we can try to optimize plans during their execution.

In our implementation, the robot waits until the best candidate solution for the plan is not optimal, but is good enough, remembers this plan and starts to execute it. During the plan execution better candidate solutions can be found, either because the first plan was not optimal or the environment has changed. Expected plan qualities estimated using the most recent and the same information about the environment should be compared, so the plan under execution should be re-simulated. The robot switches to another plan, if it is expected to improve the achieved value of the objective function by a (non-zero) certain threshold. The robot starts executing a new plan from the current moment of time. This means that it skips all actions that should have already been executed, and partially executes the action that should be executing right now. That is, continuation of executing partially executed plans corresponds to simulation of partially executed plans. It can be seen that there is some decision-making in the plan

**Fig. 2.** Appearance of unexpected obstacle

execution procedure, but it is extremely simple, and the most part of the job is done in the general-purpose optimization engine of PPL.

Let us consider how it works. If the robot starts at the position, e.g. (−4.12, −0.13, −3.09), and its goal is the point (−5.12, −0.18) located in one meter in front of the robot, the first acceptable plan will typically be something like *posture* (stand), *wait* (720), *walk* (−0.69, −0.43), *walk* (0.74, −1.13), *walk* (0.91, 1.52), which is neither precise nor most efficient. Also, robot's orientation changes after standing up, e.g. to −2.96, and the plan becomes even less precise. Better plan will be found while the robot is doing its first action (standing up), e.g. *posture* (stand), *walk* (−0.72, 0.02), *walk* (−0.29, −0.04). If an obstacle suddenly appears in front of the robot even if it has started to walk leading to instant decrease of current plan quality, good plan with obstacle avoidance and account for the current real coordinates can be found before collision, e.g. *posture* (stand), *walk* (0.57, 0.51), *walk* (0.42, −0.65). If nothing else happens, no plan change will occur and the final robot position and orientation can be (−5.12, −0.16, −2.95) that is quite close to the goal.

Of course, success highly depends on sensors and actuators. In particular, usage of NAO sonars allows rough estimation of distance to obstacles, but not their coordinates or sizes. Interestingly, if the robot does collide and fall down, and this is reflected in the current robot pose and position (that can be known from sensors), it will find a plan including stand up action and further walk to the goal. That is, simultaneous plan optimization and execution works well for plan recovery.

## 5 Conclusion

Our study showed that possible applications of probabilistic programming in intelligent agents go beyond offline inductive and deductive reasoning and also include real-time robot control. Latter can be achieved, because execution of optimization queries for dynamically varying objective functions leads to tracking of their current optima by current population of candidate solutions. Thus, simultaneous plan optimization and execution automatically, without introducing special plan recovery methods, yields adaptation of the plan to changes in the environment or to imprecise execution of actions.

Of course, time scales of plan optimization and action execution should be similar. We were lucky, and special alignment of these scales was not necessary in our experiments, otherwise more complex coordination between modules would be necessary. More specific optimization methods, which explicitly take variations of objective functions with time into account, might be necessary to develop in order to avoid convergence to degraded populations of candidate solutions that cannot adapt to changes.

Nevertheless, all specific information concerning plan recovery task is contained in the procedures of plan generation and simulation, which are necessary also for planning itself and should be the part of the agent's knowledge, while optimization queries can be considered as a basic cognitive function. Deep interactions of this function with the agent's knowledge might be necessary in advanced AGI systems, but for other reasons discussed in another paper.

**Acknowledgements.** This work was supported by Ministry of Education and Science of the Russian Federation, and by Government of Russian Federation, Grant 074-U01.

## References

1. Ould Ouali, L., Rich, Ch., Sabouret, N.: Plan recovery in reactive HTNs using symbolic planning. In: Bieger, J., Goertzel, B., Potapov, A. (eds.) AGI 2015. LNCS, vol. 9205, pp. 320–330. Springer, Heidelberg (2015)
2. Potapov, A.: A Step from Probabilistic Programming to Cognitive Architectures (in print)
3. Wang, P.: The Logic of intelligence. In: Goertzel, B., Pennachin, C. (eds.) Artificial General Intelligence. Cognitive Technologies, pp. 31–62. Springer, Heidelberg (2007)
4. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: a language for generative models. arXiv:1206.3255 [cs.PL] (2008)
5. Batishcheva, V., Potapov, A.: Genetic programming on program traces as an inference engine for probabilistic languages. In: Bieger, J., Goertzel, B., Potapov, A. (eds.) AGI 2015. LNCS (LNAI), vol. 9205, pp. 14–24. Springer, Heidelberg (2015)
6. Hutter, M.: Universal Artificial Intelligence: Sequential Decisions Based on Algorithmic Probability. Springer, New York (2005)
7. Boella, G., Damiano, R.: A replanning algorithm for a reactive agent architecture. In: Scott, D. (ed.) AIMSA 2002. LNCS (LNAI), vol. 2443, pp. 183–192. Springer, Heidelberg (2002)

8. Ayan, N.F., Kuter, U., Yaman, F., Goldman, R.P.: HOTRiDE: hierarchical ordered task replanning in dynamic environments. In: ICAPS Workshop, Providence, RI (2007)
9. Karapinar, S., Altan, D., Sariel-Talay, S.: A robust planning framework for cognitive robots. AAAI Technical report WS-12-06, pp. 102–108 (2012)