Proc. ICAPS 2010 Workshop on Planning and Scheduling under Uncertainty,
May 13, 2010, Toronto, CANADA.

Observation planning with on-line algorithms and GPU heuristic computation *

Matthieu Boussard and Jun Miura
Department of Information and Computer Sciences
Toyohashi University of Technology,
Toyohahsi, Japan

Abstract

When making an useful description of its environment, a robot has to identify both the free space and the objects location. SLAM algorithms are used for computing the free space map and an image processing algorithm is used in order to identify the objects. Those algorithms are time consuming (the time to go to the observation location and the image processing time) and are not perfect (their outcomes are stochastic). Furthermore, the agent may have multiple target to identify at the same time, and so has to build a policy for identification. We propose a Markov Decision Process (MDP)-based approach to compute those policies. Since in our application, the policy has to be computed on-line, in a limited time, optimal algorithms are too slow for those on-line purposes with a large state space. We show how on-line approaches offer solutions to the observation planning problem, and how to efficiently use those algorithms by computing an accurate admissible heuristic on a GPU.

Introduction

Building a map of a robot's environment has been well studied, and many SLAM (Simultaneous Localization and Mapping) methods have been developed (Thrun, Burgard, and Fox 2005) in order to build this map. Those methods are taking into account the uncertainty on both perception and control. They provide a geometric map, which allows the robot to move safely.

We are interested in building a cognitive map (Vasudevan et al. 2007), by adding semantic information on the objects that are present in the room. In (Masuzawa and Miura 2009) the authors present an object recognition algorithm and a planning algorithm which computes the related observation plan. In this scenario, the robot has to explore an unknown room, make the room's map while putting on this map the position of the various objects that have been detected and recognized. The number of objects and their positions are unknown at the beginning of the exploration, and only the object's types are supposed previously given.

The robot can use two kinds of image processing algorithm, for two different purposes. It has a color-based algorithm that allows the detection of several candidate objects in a wide area of the room, with a coarse localisation. Those objects have to be then identified by coming closer, and by using the second type of algorithm. Since multiple candidates are detected by the first algorithm, the agent has to compute an identifying policy for those detected objects. Nevertheless, the image processing algorithm isn't perfect, and may fail to identify an object. The object recognition probability depends on the viewpoint from where the object is observed. So after one identification fails, the robots may still want to observe the object from another viewpoint. After several identification steps, the object is ignored (the object is neither identified nor rejected).

The problem we are considering is dedicated to the planning part of the previous problem. Namely, how to compute an efficient plan to recognize several candidate objects under uncertainty. We are not dealing with the exploration aspect since we suppose the map as known, but the observation planning problem is still related to this problems, where next best viewpoint (Connolly 1985) is a classical approach. Since we suppose as known the global shape of the room, we can use planning techniques because the robots has now enough reliable information to make an efficient long term plan.

The planning algorithm proposed in (Masuzawa and Miura 2009) takes into account the uncertainty on the observation, but lacks in showing an optimality proof, and also performs an exhaustive search in the plan space, which limits the tractability of the solution.

We propose a Markov Decision Process (MDP) (Bellman 1957)-based approach to compute the observation policy. MDPs have been widely studied (Puterman 2005), and offer a strong theoretical background for action planning under uncertainty. They offer optimality proofs as well as many specialized algorithms. This paper deals also with planning in large state space problems where the classical algorithms may not be used.

This paper proposes a modelization of a classic robotic problem, the observation planning problem, as

*This work is supported by NEDO (New Energy and Industrial Technology Development Organization, Japan) Intelligent RT Software Project.

an MDP so that planning under uncertainty techniques can then be used. It also shows the suitability of two on-line algorithms UCT and LRTDP for solving MDP on that mobile robot planning problem. Furthermore, those two algorithms need an heuristic to focus their search. Hence we also propose an admissible heuristic, and since it is a different problem than the general MDP, we show how this heuristic can be efficiently solved on a GPU.

This paper is organized as follows : we first present the Markov Decision Processes and the related algorithms, then we show the MDP model for observation planning. The algorithms we are using to solve the MDP on-line, UCT and LRTDP, are then detailed, and the heuristic computation on the GPU. The results we obtain on complexity and on the quality of the solutions are presented and finally we conclude this paper.

Related works

Markov Decision Processes (Bellman 1957; Puterman 2005) allow the formalization of a sequential decision problem under uncertainty. This process is supposed to be fully observable, i.e. the observed state is the actual state of the system. By definition of the observation planning problem, this assumption is not true, since uncertainty is on the observations. If we want to take into account precisely this uncertainty, the process became partially observable, and the related decision process is called Partially Observable Markov Decision Process (POMDP) (Cassandra, Kaelbling, and Littman 1994). In POMDP the state of an agent is represented by a probability distribution over the states and this distribution is updated according to the observation the agent receive. Even this formalism is well suited for the observation planning problem, the complexity of the related algorithms is too high for the size of the problem we are considering here. Thus we are considering a fully observable MDP and we will show it may still be suited to the observation planning problem. An MDP is a 4-tuple $\langle S, A, P, R\rangle$, where :

- S is the (finite) set of states,
- A is the (finite) set of actions,
- $P : S \times A \times S \rightarrow [0; 1]$ is the transition function,
- $R : S \times A \rightarrow \mathbb{R}$ is the reward function.

The transition function allows to model the dynamics of the process. This process is supposed Markovian, so the transition only depends on the current state, independently to the history $\forall s_t, s_{t+1} \in S, a \in A$:

$$P(s_{t+1}|s_0, a_0, \ldots, s_t, a) = P(s_{t+1}|s_t, a) \qquad (1)$$

The reward function defines the parameter we want to be optimized (here we want to minimize the time).

Once a problem is formalized as an MDP, it is then possible to compute a policy. A policy π is a function $\pi : S \rightarrow A$ that gives for every state, the action to perform. Following this policy, we can compute the value function $V^\pi : S \rightarrow \mathbb{R}$ describing the expected reward according to π, and a selected performance criterion. The one we choose (and one of the most used) is the total expected discounted reward, with a discount factor $0 \leq \gamma < 1$.

$$\forall s \in S, V^\pi_\gamma(s) = E^\pi \left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s \right] \qquad (2)$$

$V^\pi_\gamma(s)$ is the unique solution to the fixed point equation $\forall s \in S$:

$$V^\pi_\gamma(s) = r(s, \pi(s)) + \gamma \sum_{s' \in S} p(s, \pi(s), s) V^\pi_\gamma(s') \qquad (3)$$

The unique optimal value function is given by the Bellman equation (Bellman 1957). $\forall s \in S$:

$$V^*(s) = \min_{a \in A} \left(r(s, a) + \gamma \sum_{s' \in S} p(s, a, s') V^*(s') \right) \qquad (4)$$

Once the optimal value function is obtained, we can compute one optimal policy, noted π^*. It gives the best action to perform according to the selected performance criterion.$\forall s \in S$:

$$\pi^*(s) = \arg\min_{a \in A} \left(r(s, a) + \gamma \sum_{s' \in S} p(s, \pi(s), s') V(s') \right) \qquad (5)$$

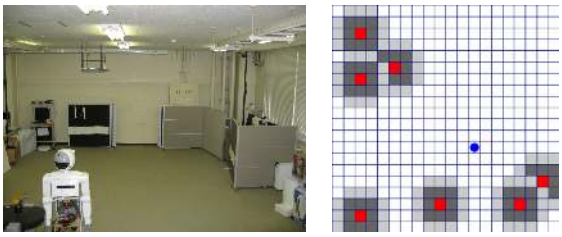
There are various algorithms to compute V^*, such as Value Iteration, or Policy Iteration (dynamic programming) (Bellman 1957). The complexity of those algorithms is polynomial ($\mathcal{O}(|S|^2 * |A|)$ per iteration, the number of iterations is bounded and polynomial).

The main problem in MDP, known as the *"curse of dimensionality"*, is the exponential state space growth due to the addition of one variable in the decision process. But for real application, we may need to define a state as a conjunction of several variables.

Researchers proposed different ways to tackle this problem. Factorized planning (Boutilier, Dean, and Hanks 1999) aims at exploiting the problem's structure to define dependencies between variables so that it may be possible to solve independent part separately avoiding the Cartesian product of variables domains.

If the problem doesn't seem to have such a structure, it may be still possible to use heuristic algorithms like $LAO*$ (Hansen and Zilberstein 2001). They suppose a goal directed MDP, with a starting state, a set of goal state and an admissible heuristic function. They proceed by successively expanding the search tree and apply a dynamic programming algorithm on that partial tree. Only accessible states are expanded, thus heuristic algorithms explore only a subset of the full state space. $LAO*$ for instance uses policy iteration after an expansion step. $LRTDP$ (Bonet and Geffner 2003), which will be detailed in the following, may be also classified in this category.

It is also possible to get an approximate solution of the optimal value function. For example, as solving an MDP can be seen as solving a set of linear equation,



(a) Actual (b) Simplified

Figure 1: Room's representation

in (Dolgov and Durfee 2006) the authors propose an algorithm to approximate the resulting linear program. But those approaches are still too slow for an embedded real application.

Finally, Monte-Carlo algorithms aim at solving MDP by generating simulated trajectories, where a trajectory is a succession of states and actions following a sampled policy. It has been proved in (Kearns, Mansour, and Ng 2002) that it is possible to obtain a policy arbitrary close to the optimal one, given only a simulator of the system. The first proposed algorithm is impractical due to the number of simulations needed, but many improved algorithms (Péret and Garcia 2004; Kocsis and Szepesvári 2006) have followed showing that sampling-based algorithms may also be good tools for solving MDP.

Observation Planning problem : Model

In order to use MDP to solve the observation planning problem, we need to define the four components of an MDP, namely $\langle S, A, T, R\rangle$.

States set S

Fig. 1(a) shows an example of environment where the robot has to find objects. Fig. 1(b) is the simplified view of this room, which will be used for planning. Red squares are candidate objects, the grey squares are viewpoints from where those objects can be identified by using a dedicated algorithm (observation succeed with higher probability on darker squares), and the blue dot is the current robot position. Since the transitions are independent to history, the current state has to contains all previous information needed to take the decision. Thus a state is composed by :

- the current position (x, y) of the agent ,
- the list of observed viewpoints. Since we don't want the agent to observe twice the same object from the same viewpoint, we need to keep the list of observed viewpoints $\{\{vp_1^1, vp_1^2, \ldots, vp_1^m\}, \ldots, \{vp_n^1, vp_n^2, \ldots, vp_n^m\}\}$ for all n objects, where vp_i^j is the jth observation for the ith object and m the maximum number of observations allowed.
- the information I_i whether the object i still need to be checked. Actually, an object has three states : it may be identified, rejected, or still unknown. A goal state is reached when all objects are non-unknown.

Hence, a state s is defined by :

$$s = \langle x, y, \{\{vp_1^1, \ldots vp_1^m\}, \ldots, \{vp_n^1, \ldots vp_n^m\}\}, \{I_1, \ldots I_n\}\rangle$$

It is clear that adding an object increases exponentially the size of the states space. Nevertheless it is possible to merge some states together. Since the state has to contains all information needed to take the decision, any superfluous information can be forgotten. Thus, once an object has been set as identified or rejected, there's no need to keep the information about its viewpoints anymore, since it isn't significant to take the right decision. For instance, among the four states listed below, the first three are equivalent, and not the fourth since the first object yet has not been recognized yet.

$$\begin{array}{lll} \langle 3,4, & \{\{(3,5)\},\{\emptyset\}\}, & \{Identified, Unknown\}\rangle \\ & = & \\ \langle 3,4, & \{\{(6,8)\},\{\emptyset\}\}, & \{Identified, Unknown\}\rangle \\ & = & \\ \langle 3,4, & \{\{\emptyset\},\{\emptyset\}\}, & \{Identified, Unknown\}\rangle \\ & \neq & \\ \langle 3,4, & \{\{\emptyset\},\{\emptyset\}\}, & \{Unknown, Unknown\}\rangle \end{array}$$

Actions set A

The agent has two kind of actions, move actions and observation actions. We suppose that the navigation is made by another piece of software, and that the robot reaches its destination without uncertainty. So the move actions are just given by the destination the agent wants to reach. The agent can actually only go on a viewpoint. Observation actions are applying the image recognition algorithm to a target candidate. Their behaviours are given by the transition function.

Transition function P

Since we suppose that move actions are deterministic, $P(s'|a, s) = 1$ iff s' includes the target position of a and if the remainder of the state is the same (see state definition), 0 otherwise. In order to limit the expansion of the search tree, we state also that the agent cannot move if there is something to observe and that nothing has yet been observed from this viewpoint. The optimal policy cannot be composed of succession of move actions. We avoid thus multiple unnecessary move action. The observation actions are stochastic, and the transition function is defined by the behaviour of the image recognition algorithm. An example of the effect of distance on recognition probability is given in Fig. 2. Those matches are the results given by the algorithm proposed in (Masuzawa and Miura 2009). If the number of SIFT matches is higher than a given recognition limit, the object is recognized, otherwise not. Thanks to experimentation, it is possible to compute the mean and the standard deviation of the number of SIFT matches

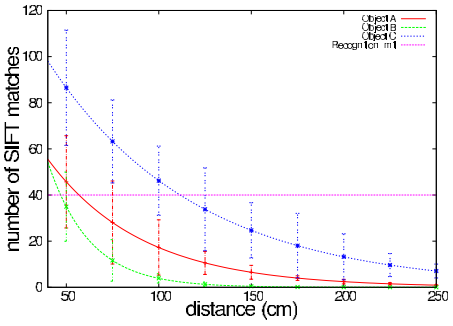


Figure 2: Effect of distance on SIFT matches

according to several distance for various objects. With those information it is then possible to compute the probability of recognition. Furthermore, we define a maximum number of observations MAX_OBS per object. Once this limit is reached or if this object has been identified (or rejected) there is no need to check this object again. Fig. 3 shows this process. It is impossible to apply an observation action on an already identified or rejected object and also for object that have been observed MAX_OBS times. If information about an object, like its pose, are collected before the object is recognized, then it is possible to modify the transition function according those new information.

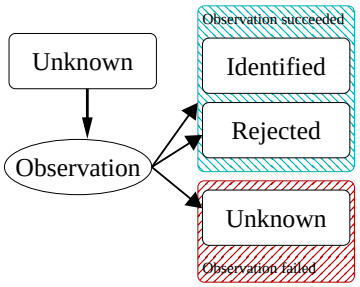


Figure 3: Observation outcomes

Reward and Cost function R

For convenience, we can express an utility function U instead of R so that $U(s,a) = R(s,a) - C(s,a)$. The cost function is given by the actual travelling time of the robot from a viewpoint to another one and by the time of the image processing algorithm. The agent should not only optimize the time spend to recognize all the objects, but also the highest number of identified objects. As discussed in the conclusion, finding a policy making a good trade-off is a hard multicriteria optimization problem. So, we are only using a Cost function, and every reward are being set to 0. In that configuration, the agent will perform the fastest plan, avoiding to make to much observations, but will not take into account the number of recognized objects.

Algorithms

The agent has to compute its policy on-line, during its mission when discovering candidate objects. It is clear that the computation time has to be low, and that the algorithm should react when the agent gets new information. In our case, we state one second as an acceptable computation time. We compare here two algorithms UCT (Kocsis and Szepesvári 2006) and LRTDP (Bonet and Geffner 2003).

The general process of the system in the observation planning context is shown in the Fig. 4. It first gathers environment's information, secondly updates if needed the MDP model of the environment, then runs a planning algorithm to select the action to perform and finally executes this action. Then the process is repeated until the end of the mission. The system is thus able to adapt to any change in the environment or with any information gathered on candidates objects before every decision step.

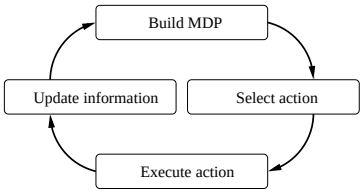


Figure 4: Global execution loop

Upper Confidence bound applied to Tree

UCB[1] applied to trees (UCT) (Kocsis and Szepesvári 2006) algorithm has shown its efficiency in board games with large state space, like Go (Gelly and Wang 2006). Alg. 1 is the main loop of planning algorithm. This part is controlling the depth of the search using H and the time (so the number of generated samples) using *timeout*. H is the number of decision steps that are being sampled. As shown in (Péret and Garcia 2004) it may be important to control this depth in order not to lower the quality of the estimated value. The algorithm has an anytime behaviour where another process may interrupt the algorithm and the current solution is returned. Otherwise, the solution is returned once a fixed number of samples have been computed. Alg. 2 is the sample's generation. When the sample reaches a goal state, the *Goalvalue* function evaluates it.

Algorithm 1: UCT Planning

Data: Starting state s_0
Result: Approximate best action a
repeat
  `search` (s_0, H);
until *timeout* ;
return `BestAction` (s_0, a)

[1]UCB stands for Upper Confidence Bounds

Algorithm 2: UCT Search

Data: Starting state s, depth h
Result: The value q of the sample
if `IsTerminal` (s) **then**
  return `goalValue` (s)
if $h = 0$ **then**
  return `heuristicValue` (s)
action $a \leftarrow$ `selectAction`(s);
state $s' \leftarrow$ `simulateAction`(s, a);
$q \leftarrow$ `cost`$(s, a) + \gamma$`search`$(s', h-1)$;
`updateValue` (s, a, q, h);
$minQ \leftarrow \min_a$ `cost`$(s, a) + \gamma \sum_{s' \in S} \widetilde{p}(s'|s, a)V(s')$;
return $minQ$

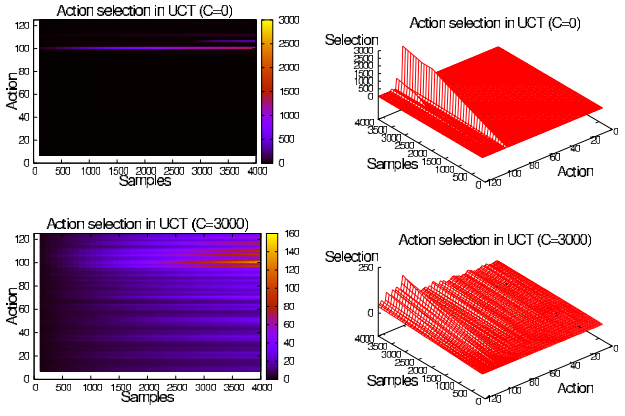


Figure 5: Action selection frequency

The *selectAction* function chooses the next action to sample using the $UCB1$, Eq. 6. It keeps tracks of the average sampled reward $\overline{Q(s,a)}$ and selects the action with the best confidence bound (here we are minimizing a cost, instead of maximizing rewards) :

$$a = \text{argmin}_{a \in A} \left\{ \overline{Q(s,a)} - C\sqrt{\frac{\ln N_s}{N_{s,a}}} \right\} \qquad (6)$$

Where N_s is the number of times for which s has been sampled, and $N_{s,a}$ the number of time action a has been selected in s. The C constant allows to tune the exploration strategy. A low value of C will give more importance to the Q-values, and thus exploit more, whereas a high value of C will give more importance to exploration. The focus of the exploration is illustrated on Fig. 5. To explain the advantages of UCB1, we introduce some definitions from the bandit theory (Auer, Cesa-Bianchi, and Fischer 2002), and we adapt them to the MDP problem. Since actions are uncertain, to sample several times an action a_i leads to the Q-values $Q_{a_{i1}}, Q_{a_{i2}} \ldots$ The expected regrets of the exploration policy after n sample is given by :

$$E\left[\sum_{j=1}^{|A|} \sum_{t=1}^{N^n_{s,a_j}} Q_{a_{jt}}\right] - \min_i E\left[\sum_{t=1}^{n} Q_{a_{it}}\right], \qquad (7)$$

where a_{it} is the action selected by the exploration strategy at time t, and N^n_{s,a_j} is the number of time action a_j has been selected up to time n. By increasing n, it has been shown that no exploration policy can have a regret that grows slower than $\mathcal{O}(\ln n)$. If an operator has a regret which grows in a constant factor of the best possible regret, it is said as solving the exploration-exploitation trade-off. In (Kocsis and Szepesvári 2006) the authors proved that $UCB1$ operator solves the exploration-exploitation trade-off.

In Alg. 2, if a state is expanded for the first time, every action have to be sampled once to apply the *selectAction* function. In Monte-Carlo planning the next state is generated only by a call to a simulator of the system, the $simulateAction(s, a)$ function. For example, in a state where an object is observable, and not yet identified, let a be the action selected by $UCB1$. The algorithm will generate, or update, a branch in the search tree corresponding of one possible outcome s' of action a executed from s. Since the actions are supposed stochastic, many different outcomes are possible. By contrast with traditional MDPs algorithms, there's no need to define the model, and only a simulator is needed. Nevertheless, it is still possible to get an approximation of the transition function $\widetilde{p}(s'|s, a)$ by computing the number of time s' has been reached from s using action a, divided by the total number of execution of a in s, $\widetilde{p}(s'|s, a) = \frac{N_{s,a,s'}}{N_{s,a}}$.

The *updateValue* function updates the Q-value of action a. This is done by computing mean value of the trajectories sampled for this action. Then the value of the state is updated using Bellman equation Eq. 5.

Labeled Real Time Dynamic Programming

The algorithm LRTDP (Bonet and Geffner 2003), Alg. 3, is an extension of $RTDP$ (Barto, Bradtke, and Singh 1995) with a labeling process. It solves an MDP by ordering the update and by using an heuristic to restrict the updates to potentially interesting states. It starts from a starting state, and selects the next state to update with the function pickNextState which simulates one outcome of an action. The Update function apply the backup operator in a state according to the complete transition function. The Checksolved is the labeling technique that improves RTDP algorithm by marking the converged states. The algorithm stops when the starting state is labelled as solved or when it reaches the timout.

Heuristic computation

We use as an heuristic the values of the same MDP where the observation always succeed. This is an admissible heuristic and underestimates the actual value func-

Algorithm 3: LRTDP

Data: Starting state s_0, ϵ
Result: Optimal action a
while $\neg$`Solved`$(s_0) \wedge \neg timout$ **do**
  `LRTDPtrial` (s_0, ϵ);
return `Best` (s_0, a)

Algorithm 4: LRTDPtrial

Data: Start state s,ϵ
$visited$ =EMPTYSTACK;
while $\neg$`Solved`(s) **do**
  `Push`$(visited, s)$;
  if `isGoal` (s) **then**
    break;
  a$\leftarrow$`GreedyAction` (s);
  `Update` (s);
  s$\leftarrow$`PickNextState` (s, a);
while $visited \neq EMPTY\_STACK$ **do**
  s$\leftarrow$`Pop` $(visited)$;
  if $\neg$`CheckSolved`(s, ϵ) **then**
    break;

tion. This Deterministic MDP can be represented by a directed acyclic graph, we organize the states by level according to the number of observations needed and perform a bottom-up backup. This exhaustive backup can be massively parallelized and has been developed in order to be computed on a GPU, each thread computing its value. GPU are taking advantages of data locality, hence we represents the states by a simple 1D array, Fig. 6, where the first elements are the states with 0 observation, then the states with 1 etc. We encode all state information on a 32 bits integers, 10 bits for the coordinates (x, y), and one bit for the status of each object (observed or not). It is sufficient since in that case there is no uncertainty on the observations.

To compute the shortest path, we use a simple bottom-up parallel algorithm, see Fig. 7 and Alg. 5. Its has been coded in OpenCL. Implementation details have been omitted like memory alignment, local memory management or synchronizations. Many useless computations are made since every states of the level $l+1$ are evaluated to compute one state of level l and lots of them have a transition probability of

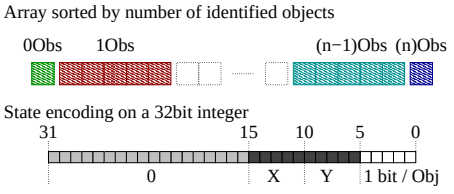


Figure 6: Data structure for planning on GPU (5 obj)

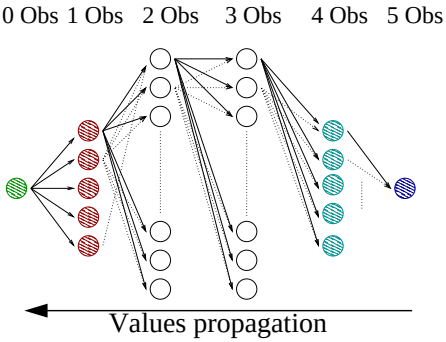


Figure 7: Bottom-up shortest path (5 Obj)

0. The number of operation needed for one level l is $\binom{NB\_OBJ}{l} * \binom{NB\_OBJ}{l+1} * |Viewpoints|^2$. This algorithm is work-efficient since it doesn't perform more backup than the non-parallel one ($\mathcal{O}(|S|^2)$). Despite of that quite inefficient algorithm, the design fits perfectly of the requirement for good performances on GPU and allows to find the optimal heuristic value function.

Algorithm 5: Parallel bottom-up heuristic

Data: states[],nb_obj,nb_vp
Result: V*
end$\leftarrow$ nb_vp$*2^{\text{NB_OBJ}} - 1$;
start$\leftarrow$ end-nb_vp+1;
for *level=nb_obj-1 down to 0* **do**
  target_start $\leftarrow$ start ;
  target_end $\leftarrow$ end ;
  end $\leftarrow$ start-1;
  start $\leftarrow$ start $- \binom{\text{nb_obj}}{\text{level}} *$nb_vp;
  forall $k \in$ *[start;end] in parallel* **do**
    s$\leftarrow$states[k];
    min$\leftarrow +\infty$;
    for *target=target_start to target_end* **do**
      s'$\leftarrow$states[target];
      val$\leftarrow$ V*[target];
      val$\leftarrow$ val+cost(s,s')+obs_cost;
      val$\leftarrow$ val*transition(s,s');
      if *val<min* **then**
        min$\leftarrow$ val;
    V*[k]$\leftarrow$ min;
return *V**

Computing this heuristic is time consuming (see Tab. 1) and the complete plan must be computed under 1s. So, given the current heuristic implementation, we can't deal with more than 8 objects and 21 viewpoints per objects (on a GeForce 8600GTS). The CPU algorithm performs the same bottom-up algorithm. LRTDP can't use an good heuristic so it almost has to explore the complete state space. The setup time for executing the kernel on the GPU is around 0.15s (load the program from a file, compile it and build the

#object	4	5	6	7	8	9
GPU	0.36	0.35	0.39	0.43	0.6	1.24
CPU	0.01	0.03	0.1	0.33	1.36	7.14
LRTDP	4.56	13.27	33.94	19.59	98.27	134.1

Table 1: Deterministic shortest path computation time

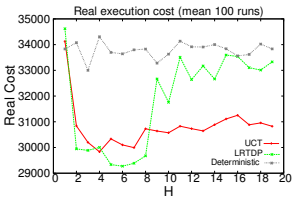
executable), so we need a certain size of problem to see the efficiency of the GPU algorithm. Once build, the kernel can be reused without this extra time.

Results

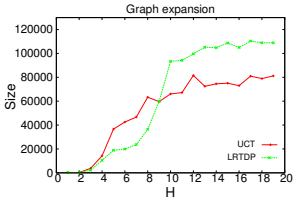
When UCT don't explore, its main difference with LRTDP is that the Q-values aren't computed from the complete model, but only sampled from executions. We fixed the *timout* to 0.5s, since it takes around 0.5s to computes the heuristic. Because the heuristic value is a good estimation of the real cost, and because the robot will receive new information between two planning steps, we are taking into account only the uncertainty on a limited horizon. After that horizon H, we are using the heuristic value.

Fig. 8(a) shows the impact of H on the performances. This is the mean on 100 runs values on the situation depicted Fig.11. The plan alternatively selects one move action and one observation action. Furthermore the move actions are deterministic, hence the uncertainty arises only on observation steps. The deterministic execution computes a plan without uncertainty (i.e. follows the heuristic). With $H = 1$ the robot are just following the heuristic value, it also gives the deterministic plan value. With a limited H LRTDP manages to find the optimal value. When H is very high, ($H > 8$), UCT is still able to find a good plan, while LRTDP performances drop. This drop can be explained by the time spent for labeling the states while UCT just samples trajectories with no further work. The fact that UCT only samples the Q-value impacts on the graph expansion, limiting it, see Fig. 8(b).

The evolution of the value functions are presented Fig. 9 and Fig. 10. It shows that the expected value of LRTDP is more accurate ($V(s_0)$ is higher) than UCT and that LRTDP is able to converge for $H \leq 6$. It also shows that increasing H first increases the precision of



(a) Mean execution cost



(b) Size of search Graph

Figure 8: Comparison between UCT and LRTDP

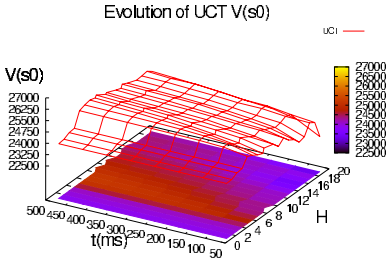


Figure 9: Evolution of UCT starting state's value

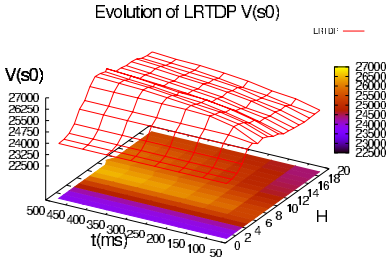


Figure 10: Evolution of LRTDP starting state's value

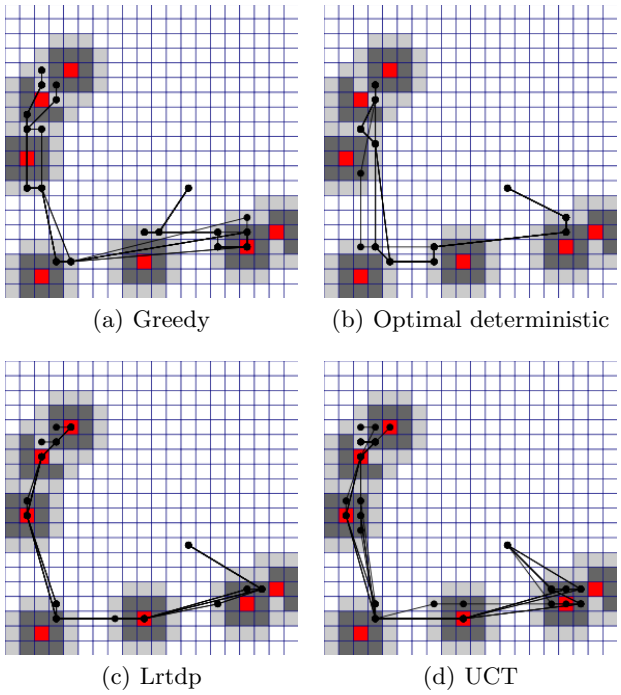
the value function and then, due to insufficient computation time, decrease. The mean cost of 100 runs is in the best case 29000 and the higher value of the starting state is 26567. So it shows that both algorithms are underestimating the real value. The optimal value, without any restriction on the horizon, is 28366 and has been computed using LRTDP in 10 minutes.

Fig. 11 shows the path followed by the various algorithms. Greedy policy goes to the closest viewpoint, optimal deterministic is the shortest path with deterministic observation, the two others are LRTDP and UCT computed with $H = 6$.

Conclusion and future works

This paper shows the suitability of UCT and LRTDP, two on-line algorithms, for observation planning. The main advantage is their anytime behaviour, allowing the robot's main controller to stop the algorithm when action is needed or let the quality of the solution increases. The other advantages comes with the MDP formalisation, allowing a good description of the variable to optimize. We have shown several algorithms, and each of them has advantages and drawbacks. Selecting the good algorithm according to the problem to solve is also an interesting decision problem.

Planning with limited lookahead using an heuristic has yet some issues. With a very small horizon, the policy tends to be greedy. We can't avoid that issue, and we have to set a minimum horizon so that this phenomenon will not occur. The second issue is the labelling process of LRTDP which may labelled some states as solved when they are not (observing twice an object, or immediately succeed the observation, may leads to the same state). In that case the value of s_0



(a) Greedy (b) Optimal deterministic

(c) Lrtdp (d) UCT

Figure 11: Execution of different policies, 10 runs

depends on which trajectory has been first sampled. We want to introduce a backward re-labelling process to fix that issue and study the impact on the quality.

Lots of work has been done in multi-criteria MDP (White 1982). Since the observation planning problem has two different and opposite variables to optimize, namely the time and the number of identified objects, it is hard to merge everything in a single real valued reward function. Adapting UCT or LRTDP in a multi-criteria context is a challenging work. We are more especially interested in finding one pareto-optimal policy instead of all non dominated policies, which is often the case in multi-objective optimisation.

We manage to compute the heuristic efficiently on the GPU. The current implementation is very simple and, even the algorithm is work-efficient, it still computes many unnecessary null transition. There lots of space for further optimization. Since the MDP are P-complete, we don't want to solve every MDP on the GPU, but we want to define a subclass of MDP which could efficiently be solved with that kind of algorithm, or, like in the present paper, how a part of the computation can be efficiently deported on the GPU.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.* 47(2-3):235–256.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to act using real-time dynamic programming. *Artif. Intell.* 72(1-2):81–138.

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bonet, B., and Geffner, H. 2003. Labeled rtdp: Improving the convergence of real-time dynamic programming. In Giunchiglia, E.; Muscettola, N.; and Nau, D. S., eds., *ICAPS*, 12–31. AAAI.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Cassandra, A. R.; Kaelbling, L. P.; and Littman, M. L. 1994. Acting optimally in partially observable stochastic domains. In *AAAI'94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, 1023–1028. Menlo Park, CA, USA: American Association for Artificial Intelligence.

Connolly, C. 1985. The determination of next best views. In *IEEE International Conference on Robotics and Automation.*, 432–435.

Dolgov, D. A., and Durfee, E. H. 2006. Symmetric primal-dual approximate linear programming for factored MDPs. In *Proceedings of the Ninth International Symposiums on Artificial Intelligence and Mathematics (AI&M 2006)*.

Gelly, S., and Wang, Y. 2006. Exploration exploitation in Go: UCT for Monte-Carlo Go.

Hansen, E. A., and Zilberstein, S. 2001. Lao*: A heuristic search algorithm that finds solutions with loops. *Artif. Intell.* 129(1-2):35–62.

Kearns, M. J.; Mansour, Y.; and Ng, A. Y. 2002. A sparse sampling algorithm for near-optimal planning in large markov decision processes. *Machine Learning* 49(2-3):193–208.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In Fürnkranz, J.; Scheffer, T.; and Spiliopoulou, M., eds., *ECML*, volume 4212 of *Lecture Notes in Computer Science*, 282–293. Springer.

Masuzawa, H., and Miura, J. 2009. Observation planning for efficient environment information summarization. In *IROS*. IEEE.

Péret, L., and Garcia, F. 2004. On-line search for solving markov decision processes via heuristic sampling. In de Mántaras, R. L., and Saitta, L., eds., *ECAI*, 530–534. IOS Press.

Puterman, M. L. 2005. *Markov Decision Processes : Discrete Stochastic Dynamic Programming*. Wiley.

Thrun, S.; Burgard, W.; and Fox, D. 2005. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press.

Vasudevan, S.; Gächter, S.; Nguyen, V.; and Siegwart, R. 2007. Cognitive maps for mobile robots-an object based approach. *Robot. Auton. Syst.* 55(5):359–371.

White, D. J. 1982. Multi-objective infinite-horizon discounted markov decision processes. *Journal of mathematical analysis and applications* 89:639–647.