Active Visual Search with Partially Observable Monte-Carlo Planning

Mark Baierl[*1] Jascha Hellwig[*1]

Abstract

In this work we focus on creating an environment and agent, which allow us to perform Active Visual Search (AVS) in an online setup. Therefore, we introduce an extensible environment which uses PyBullet to allow for complex simulations. Furthermore, we present an agent which is able to find objects in the environment based on observations in form of discretized images. To do so, we use a POMDP to model the problem and a modified version of the POMCP algorithm to solve it. Finally, we test our method in terms of performance and computational effort. All of this builds the foundation for a more complex approach including not only finding but also grasping an object in a known environment.

1. Introduction

In order to bring robotic performance closer to humans, an incredible number of tasks and challenges is still left to be tackled (He et al., 2017). Many of those are not even a challenge for humans, like going into the kitchen and pick up some things to cook a meal. For a human this is not a challenge, even without knowing where exactly the desired objects may be. Humans can search objects in known environments with ease and have even less problems with grasping them when found. On the contrary, teaching a robot to search and pick up an object is indeed not a trivial task. This task is even more challenging, if the exact position and pose of that object are unknown. Solving this task would make robot assistants and workers useful in real life (Chandrasekaran & Conrad, 2015; Zeng et al., 2020).

Therefore, we set the higher goal of this work to tackle the task of *grasping objects* with unknown position and pose in a known environment. For this reason, we divide this problem into two parts: *searching* and *grasping* an object. Both tasks include various problems, especially when they are performed in a *real-world like* scenario. In such a scenario the agent has no access to the *true state* of the system and therefore has to work only with *observations*. We decided to model those observations in form of images, the robot receives while operating.
In addition, the environment can change, and the agent needs to adapt in an *online* fashion. Therefore, we need to use an online planning method to compute a policy to search and then grasp an object.

Due to the complexity of the task, we decided to focus on the first part of the problem and building the foundations for the second part in the course of this work. For this reason, we introduce a new environment which allows us to model the searching and grasping task. Since we intend to simulate those tasks, before translating them to a real robot, we use *PyBullet* (Coumans & Bai, 2016–2021) to include physics in our simulation of the environment.
We combine this with *Partially Observable Markov Decision Processes* (POMDPs) (Kaelbling et al., 1998), since they have been recently shown to provide working online planning methods for the AVS problem (Wang et al., 2020; Giuliari et al., 2021). Those methods use the *Partially Observable Monte-Carlo Plannig* (POMCP) (Silver & Veness, 2010) algorithm, which combines *Monte-Carlo Tree Search* (MCTS) (Coulom, 2006) with *Monte-Carlo Belief State Updates*. They build up a *belief-map* of the environment to search an object and after each step the belief-map gets more certain about the object's true location.

We adapt the *POMP* (Wang et al., 2020) approach to our environment and introduce a new type of observations, which represents a *discretized* version of the image. POMP uses a binary flag as observation which either represent that the agent observed the object or not. We change this type of observation to a discrete mapping from an image to an *observation matrix*, containing information found in the image.

The main contributions of this work are the new environment combining PyBullet and a POMDP formulation for AVS as well as providing an agent which uses POMCP to search an object in this environment using discretized images as observations. In addition, we test our method to proof our concept and show that the task can be solved in the given environment.

[*]Equal contribution [1]Technical University of Darmstadt, Germany. Correspondence to: Mark Baierl <mark.baierl@stud.tu-darmstadt.de>, Jascha Hellwig <jascha.hellwig@stud.tu-darmstadt.de>.

2. Problem Statement

The high-level goal of this work is to develop an *online method* to find and grasp objects in a real-world scenario. This problem formulation includes various subproblems that are all non-trivial.

First of all we need to solve the *search task* in an *online* fashion. This means, that we want to find an object in an environment that is not static. Therefore, we cannot precompute an optimal plan, but must react on changes in the environment while moving through it. In addition, we intend to work with *observations* in the form of images taken by a robot. This means, our agent has no access to the *true state* of the world, but only to its observations.

When the object is found by the agent, we are still left with grasping the object. In order to do so, we will first need to *estimate the pose* of the object. This means given the object we search, we want to estimate its pose in the real-world using a *point cloud*. If we are able to estimate the pose well enough, we can use a pre-trained grasping algorithm to finish the task.

This work will focus on solving the search task and building the basis for the pose estimation. The idea is to create a *framework*, which can later be easily extended. This framework should include an adaptable partially observable environment as well as a working method to search objects in it.

3. Related Work

Since the goal of this work is to search an object in a partially observable environment, this chapter will summarize general concepts of *Active Visual Search* (AVS).
AVS is a specific group of tasks in which a robot is asked to navigate through an environment to find a specific object. More precisely, this requires the robot to solve a planning problem as well as a visual task. In recent years, two dominant approaches have been shown to solve the AVS task: *Deep-Learning Approaches* and *POMDP-Based Approaches*.

3.1. Deep Learning Approaches

Many recent works use Deep Reinforcement Learning techniques (Schmid et al., 2019; Ye et al., 2019; Chaplot et al., 2020) and were able to solve the task efficiently. Those methods use visual neural embeddings for the policy training and combine this with 3D information for training. However, deep learning approaches usually require big datasets since they need to learn a model of the environment as well as a general motion policy. Recently there were some promising works using POMDPs and online planning strategies, which are in general easier to deploy, but still achieve comparable results (Wang et al., 2020; Giuliari et al., 2021).

3.2. POMDP-Based Approaches

In the last years there have been works achieving state of the art performance using POMDPs (Wang et al., 2020) (Giuliari et al., 2021) without any offline training. The first approach is *POMP*, a POMCP-based solution (Wang et al., 2020) that is only applicable to *known environments*. While this may seem limiting, it only requires a 2D *floor map* of the environment to provide efficient results. More important, there is a more recent approach *POMP++* (Giuliari et al., 2021), which extends this idea of online policy learning using POMDPs, to also address *unknown environments*.
A great benefit of POMDP-based approaches is that the true *belief state* is approximated, which can easily be visualized and allows us to better understand the decisions of the algorithm. Deep Learning techniques do in general not provide such insights into the decision making and in addition require huge amounts of training data.

According to the recent successes of POMDP-based approaches and their benefits, we decided to provide a POMDP-based solution for the AVS problem. In order to explain this solution, we will introduce the necessary foundations of POMDPs and planning in POMDPs, first.

4. Background

The focus of this work lies on solving the part of the problem related to *object search*. Therefore, the following chapter will focus on explaining foundations and methods required to search an object in an environment, without having access to the true *hidden* state of the world.

4.1. POMDPs

A MDP describes a process in which an agent decides which action $a \in A$ should be performed in a specific state $s \in S$. Performing this action results in a new state $s' \in S$ as well as in a reward $\mathcal{R}_s^a$. The difference between MDP and POMDP is that in the latter the agent is not able to observe the *true state* after performing an action but receives an observation $o \in O$. This observation is typically used to approximate the true state of the system (Kaelbling et al., 1998; Boucherie & Van Dijk, 2017).

More concretely, a MDP describes a system, in which the the environments dynamics at time t are fully determined by the corresponding state s_t at that time. Therefore, we can formulate the *transition probabilities* when performing an action a in any state s as $\mathcal{P} = P(s_{t+1} = s'|s_t = s, a_t = a)$. The expected reward for that action is determined by the *reward function* $\mathcal{R}_s^a = \mathbb{E}[(r_{t+1} = s'|s_t = s, a_t = a)]$ and the initial state of the system is determined by a probability distribution $\mathcal{I}_s(s_0 = s)$. Furthermore, the agent uses a *policy* $\pi(s,a) = P(a_{t+1} = a | s_t = s)$ to select actions (Feinberg & Shwartz, 2012).

Since we do not have access to the true state in a POMDP, our policy π needs to map observations to actions. In general, a so-called *history* is used, which is a sequence of actions and observations $h_t = \{a_1, o_1, ..., a_t, o_t\}$ (alternative formulation $h_t a_{t+1} = \{a_1, o_1, ..., a_t, o_t, a_{t+1}\}$). Observations are determined by the observation probabilities $\mathcal{Z}^a_{s'o} = P(o_{t+1} = o | s_t = s', a_t = a)$ and the policy can be formulated as $\pi(h,a) = P(a_{t+1} = a | h_t = h)$ (Silver & Veness, 2010).

The objective is to find a policy that maximizes the *value function* $V^\pi(h)$. This value function describes the expected *return* $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$ from state s when following policy π. Since we do not have access to the state in a POMDP the value function is defined as $V^\pi(h) = \mathbb{E}_\pi[R_t | h_t = h]$ and the optimal value function is then the maximum value function of all policies $V^*(h) = \max\limits_{\pi} V^\pi(h)$. The estimation of the true state can be done by using a *belief state*, which is a probability distribution over states given a history $\mathcal{B}(s,h) = P(s_t = s | h_t = h)$ (Silver & Veness, 2010).

In summary, a POMDP is formulated as a tuple $(S, A, O, T, Z, R, \gamma)$, with:

- Finite set of *partially observable states* S
- Finite set of *actions* A
- *Observation model*: $O : S \times A \leftarrow \Pi(Z)$
- *State-transition model*: $T : S \times A \leftarrow \Pi(S)$
- Z is the finite set of *observations*
- *Reward function* $\mathcal{R} : S \times A \rightarrow \mathbb{R}$
- *Discount factor*: $\gamma \in [0, 1)$

4.2. Monte-Carlo Planning in POMDPs

To find an optimal or at least near-optimal policy for a POMDP in an online fashion, one can use online POMDP planners. A successful approach is a version of *Monte-Carlo Tree Search* (MCTS) (Coulom, 2006) extended to the use with POMDPs, called *Partially Observable Monte-Carlo Planning* (POMCP). It combines MCTS with *Monte-Carlo belief state updates* and provides a computationally-efficient algorithm, which allows scalability for larger state spaces. It does so by focusing the samples to the most promising regions of the search space and therefore improves the belief state updates (Silver & Veness, 2010).

Before presenting the algorithm itself, we will first introduce three important concepts:

1. Evaluate a *state* using *Monte-Carlo simulation*.
2. Determine, which *action* to perform with *Monte-Carlo tree search*.
3. Update the *belief state* using *Monte-Carlo belief state updates*.

4.2.1. Monte-Carlo Simulation

Monte-Carlo simulation can be used to evaluate a state s in a MDP by using *rollouts*. A rollout from state s uses a *MDP simulator* and a random *rollout policy* to select and perform actions until either a terminal state or the discount horizon is reached. Then, we can estimate the value of the state s by the average return of N simulations, $V(s) = 1/N \sum_{i=1}^{N} R^i$, where R^i is the return of corresponding to the i-th simulation (Mooney, 1997; Tesauro & Galperin, 1997).

In order to extend this technique to the use with POMDPs, a POMDP simulator is required to generate observations, as well as a history based random rollout policy $\pi_{rollout}(h,a)$. This allows to estimate the value of a history ha by the average return of N simulations starting in ha (Bertsekas & Castanon, 1998).

4.2.2. Monte-Carlo Tree Search

Since Monte-Carlo simulation allows us to evaluate a state, we can now use it in combination with Monte-Carlo tree search (Coulom, 2006) to detect promising states. The general idea is to sequentially evaluate the nodes of a search tree in a best-first order using Monte-Carlo simulation. The main concept is that for each state s there is a node in the tree, which contains an action-value $Q(s,a)$ as well as a visitation count $N(s,a)$ for each action and an additional count $N(s) = \sum_a N(s,a)$. The action-value of a state s and an action a can then be estimated by the *average return* of all simulations in which the action was performed in that specific state (Silver & Veness, 2010).

In order to select the best action, we need to build and search the tree. For this purpose, two different policies are used:

1. A *tree policy* that is used to navigate through the search tree.
2. A *rollout policy* that is used when the scope of the tree is exceeded and new nodes are created.

In general, the rollout policy is *uniform random*, which provides the *exploration* aspect of the search. The tree policy can be implemented greedily, however it has been shown that it can be improved by using the *UCT algorithm* (Kocsis & Szepesvári, 2006). The *UCB1 algorithm* (Auer et al., 2002) is used to determine the value of an action by $Q^{\oplus}(s,a) = Q(s,a) + c\dfrac{\log N(s)}{N(s,a)}$. Important to notice is that the constant c controls the exploration and exploitation trade-off, if $c = 0$ the tree policy acts greedy. While navigating through the search tree, actions are selected by $\operatorname{argmax}_a Q^{\oplus}(s, a)$, if all actions from the current state s have been selected previously. If not all actions were already selected in a state, the rollout policy is used to select an action.

Monte-Carlo tree search can be extended to the use with POMDPs (Silver & Veness, 2010). In order to do so, states s are replaced by histories h and the search tree then contains a node $T(h) = \langle N(h), V(h) \rangle$ for each seen history h. $N(h)$ again represents the number of times a history h has been visited and $V(h)$ is its value, but now defined by the average return of all simulations starting with h. A simulation at time t starts in an initial state that is sampled from the current belief state $\mathcal{B}(\cdot, h_t)$. The purpose of the two policies remains the same but the UCB1 algorithm is now history based $V^{\oplus}(ha) = V(ha) + c\frac{\log N(h)}{N(ha)}$ (Silver & Veness, 2010).

4.2.3. Monte-Carlo Belief State Updates

In order to handle large state spaces of a POMDP, the POMCP algorithm uses an *unweighted particle filter* to approximate the belief state. In addition, the particles are updated based on sample observations, rewards and state transitions using a *Monte-Carlo procedure* (Silver & Veness, 2010).

The belief state for history h_t is approximated by K particles $B_t^i \in S, 1 \leq i \leq K$. This allows us to formulate the belief state $\hat{\mathcal{B}}(s, h_t)$ as the sum of all particles. The benefit of using an unweighted representation is that it can be implemented efficiently with a *black box simulator* of the POMDP, which does not require an explicit model of the POMDP (Silver & Veness, 2010).

In order to update our particles and therefore the belief state after the agent has performed an action a_t and received an observation o, Monte-Carlo simulation is used. At first, a particle is selected from B_t which represents a state s sampled from our current belief state. Then a black box simulator $\mathcal{G}(s, a_t)$ is used to generate the next state s' and the corresponding observation o. In order to approximate the true belief state, only particles that result in the real observation $o = o_t$ are added to the next set of particles B_{t+1}. Those steps are performed until K particles have been added and the true belief state is approximated when using sufficient many particles K (Silver & Veness, 2010).

4.2.4. POMCP Algorithm

Combining all previously mentioned methods results in the POMCP algorithm (Silver & Veness, 2010). This algorithm consists of two major steps

1. Search the best action given the current history.
2. Update the belief state after the action has been performed.

Algorithm 1 POMCP (Silver & Veness, 2010)

```
procedure SEARCH($h$)
    repeat
        if $h = empty$ then
            $s \sim \mathcal{I}$
        else
            $s \sim B(h)$
        end if
        SIMULATE($s, h, 0$)
    until TIMEOUT
    return $\operatorname{argmax}_b V(hb)$
end procedure

procedure SIMULATE($s, h, depth$)
    if $\gamma^{depth} < \epsilon$ then
        return 0
    end if
    if $h \notin T$ then
        for all $a \in A$ do
            $T(ha) \leftarrow (N_{init}(ha), V_{init}(ha), \emptyset)$
        end for
        return ROLLOUT($s, h, depth$)
    end if
    $a \leftarrow \operatorname{argmax}_b V(hb) + c\sqrt{\frac{\log N(h)}{N(hb)}}$
    $(s', o, r) \sim \mathcal{G}(s, a)$
    $R \leftarrow r + \gamma \cdot$ SIMULATE($s', hao, depth + 1$)
    $B(h) \leftarrow B(h) \cup \{s\}$
    $N(h) \leftarrow N(h) + 1$
    $N(ha) \leftarrow N(ha) + 1$
    $V(ha) \leftarrow V(ha) + \frac{R - V(ha)}{N(ha)}$
    return $R$
end procedure

procedure ROLLOUT($s, h, depth$)
    if $\gamma^{depth} < \epsilon$ then
        return 0
    end if
    $a \sim \pi_{rollout}(h, \cdot)$
    $(s', o, r) \sim \mathcal{G}(s, a)$
    return $r + \gamma \cdot$ ROLLOUT($s', hao, depth + 1$)
end procedure
```

In order to search the best action for the current history, an implementation of Monte Carlo Tree search for POMDPs

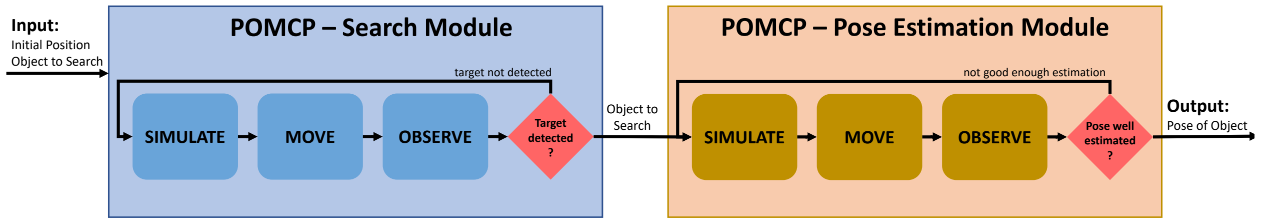


Figure 1. Combination of the Search Module and Pose Estimation Module.

(Algorithm 1) is used. The start for the search is the given history h, which allows to sample a start state $s \sim B(h)$, however if h is empty we sample a state from the initial state distribution $\mathcal{I}$.

When the search is complete, the action is performed by the agent and a true observation o_t is received. We can now use the POMDP simulator again to update the belief state by sampling states from the current belief state and passing them to the POMDP simulator.

Iteratively performing search and belief updates provides an efficient algorithm to handle large POMDPs. However, it is still necessary to provide a POMDP simulator for the problem specific POMDP to simulate steps and observation.

5. Method

In order to search and grasp objects in real-world scenarios, we need to find an object and estimate its pose. We intend to handle those two tasks independently from each other. Therefore, we introduce a *POMCP search module* as well as the concept and idea for the *POMCP pose estimation module*, which is left for future work.

The main concept can be seen in Figure 1. First, the search module (described in 5.2) uses the POMCP algorithm to find an object. It receives the initial position in the environment and the object it should search as an input. Once it believes to have found the object it terminates, and the pose estimation module starts. It receives the object to search as an input from the search module and again uses the POMCP algorithm. It terminates once it believes to have estimated the pose of the object ”well enough”.

5.1. Environment

Due to the intention for solving the task in real-world scenarios, we decided to use PyBullet (Coumans & Bai, 2016–2021) as basis for the simulation of our environment. PyBullet is a powerful library, which not only allows to include physics and image rendering for a simulated environment, but also provides tools for complex simulations of robots. For this reason, PyBullet can be used to simulate real-world scenarios in a reasonable way. This allows us to define our environment and test different approaches before deploying it to a real robot.

Since in a real-world task, we do not know the true position of the object, the only information an agent has access to are observations. We define those observations as images from a camera for the search task. Therefore, our agent can move in the environment and after each movement he observes a new image. One can imagine the agent like a camera that is flying over large table on which objects are placed.

The objects the agent has to find are simple geometric forms like an ’L’ build out of similar colored cubes. However, there might be other objects, which could make the search more difficult by occlusions, distractions or even by blocking the path.

This simple but expressive environment allows us to model and test real-world like scenarios in a simplified way, as it is simple to understand and to observe for a human. More important, this allows us to understand the way an agent acts.

5.2. POMCP Object Search

In order to apply POMCP to AVS, we use the concept and formulation provided in Wang et al. (2020). In addition, we introduce a new type of observations and use the belief state to determine when the search is successful. We expect the floor map of the environment (e.g., the possible positions inside the environment) to be known.
This allows the agent to have a map of the environment, to collect information while moving through it. This map is initialized without any information about the object’s position, initially everything is marked as a *candidate* position. While exploring, the agents marks positions as *empty*, *desired object* or *other object*. This map is part of a *state* in the POMDP as well as an expected object position, which may be at every *candidate* position.

In the following we will first present the POMDP formulation itself, to then explain how the POMCP agent simulates, moves, and observes. Finally, we will present our approach of using the belief state to determine when an object is found.

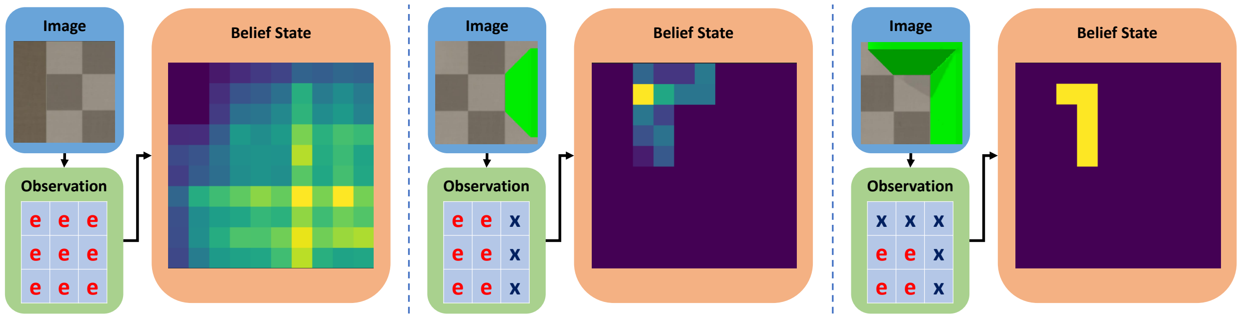


Figure 2. Visualisation of belief state updates performed while the agent moves in the environment.

5.2.1. POMDP for Object Search

In our POMDP formulation for the AVS task, the set of partially observable states S contains states, which include the position of the agent, the expected object position, and a discrete map of the environment containing already collected information. The actions A describe possible movements of the agent in the environment and after each action a reward is received. The reward function R provides a positive reward signal if the object is observed and a negative reward signal for moving as well as reobserving empty positions.

5.2.2. Simulation

In order to plan, the POMCP algorithm uses simulations. During those simulations no real observations are received, but simulated ones. This means, that for each simulation the object is placed at some candidate location according to the state's environment map. Then steps and observations are simulated just as if the object would really be at that location until the simulated object is found. While doing so, the simulator renders images according to its state of the world which act as observations during the simulation.
It is important to notice that in simulation the agent still has no knowledge of the object's true position in simulation, just the environment returns different observations for each simulation, since the object positions may differ in simulation. Using such simulations allows to express a belief in all possible objects positions and how to move optimally while considering this belief.

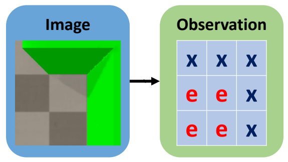
5.2.3. Actions

There are four different possible actions $a \in A$ in the POMDP. Each action corresponds to a movement in a two-dimensional space, in our case moving north, east, south or west, $A = \{NORTH, EAST, SOUTH, WEST\}$.
As already mentioned, the search space is discretized, and each action corresponds to a movement to the next discrete position in that direction. It is important to notice that not every action is possible in every state of the POMDP, for example if some obstacle blocks the way or at the boundaries of the environment.
In a POMDP it is in general possible to define actions that do not directly result in an observation. For our purpose such actions are not necessary and therefore each action directly results in a new observation.

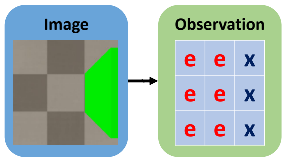
5.2.4. Observation

In previous works the observations were quite simple and a single observation was either finding the object or not (Wang et al., 2020; Giuliari et al., 2021). This reduces the complexity of observations, but it requires some reasoning about seeing the whole object. This may be beneficial in some case, still we want to leave the reasoning about having seen the whole object to the agent.

In general, we want to be able to search for objects that might not fit into a single observation and still be sure to have found them. For this reason, we have chosen a new format for observations, which is a discretized version of the image the agent receives after each action. Therefore, an observation $o \in O$ is a $N \times N$ sized grid, where N defines the level of discretization. Each element $z_{i,j} \in Z$ represent a part of the image. For example $z_{0,0}$ is the upper-left corner of the image. In order to assign such values to an element in the observation grid, a simple object detection can be used. In this version of the environment, a color-based decision is enough. This means, if the object we search is green, everything green is marked as the desired object, as can be seen in Figure 3.



(a) Example 1



(b) Example 2

Figure 3. Transforming an image to a 3×3 observation.

This type of transforming images to observations of course is not accurate, as one can see in Figure 3a and 3b. It introduces some flaws in certain situation, but this is intended. In real-world scenarios a more complex transformation has to be used, but the agent should never rely on it to be exact. This mistrust in the observation and the ability to handle it motivated these kinds of observations and is the reason why we have not chosen to stick to observations like the ones in previous methods.

The power and robustness of our agent lies in the combination of observations and the belief state. The termination of the search does not depend on the observation but on the belief state. Therefore, we will first explain the belief state of our POMDP to then explain how it is updated when receiving an observation in order to determine if the desired object has been found.

5.2.5. Belief State Update

After each simulation, the agent performs a real action and receives a real observation. As explained earlier, the POMCP algorithm then performs a Monte-Carlo belief state update. This update results in particles, which cause the same observation as the real observation received.
Therefore, if the observation shows an object, all particles will also have an object at that position in their internal environment map (see Figure 2). This means that the internal representations of the environment of all particles converges more and more to the true object position.

6. Experiments

In order to test our approach, we have implemented our environment and POMDP in a way that it can be integrated in the *POMDPy* framework (Emami et al., 2015). This framework provides an efficient POMCP solver, which we used to test our approach. To show that our method is working, we compare our approach with a random walk algorithm. In addition, we test the effect of different number of Monte-Carlo simulations on the performance and computational effort.
First, we compare how many times the algorithm has found the object (e.g., reached the terminal state) in 100 episodes, seen in Figure 4.

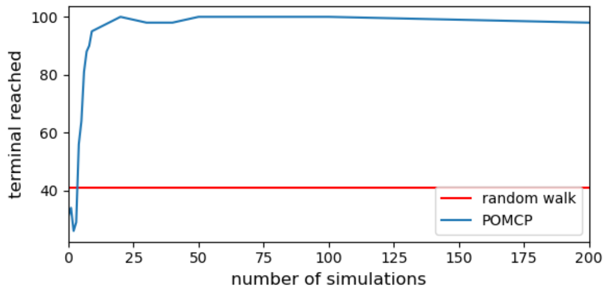


Figure 4. Comparison of different number of simulations and how often they reach the terminal state.

The random walk does on average find the object in 41 of 100 episodes before the maximum number of steps has been reached. Our approach does already surpass this performance using only four simulations, which results in an average of 56 objects found in 100 episodes. One can see that the performance settles around 50 simulations since the current environment is still quite simple and not many simulations are required to find the object reliably.
Next, we compare the average steps required to find the object using different numbers of simulations, depicted in Figure 5.

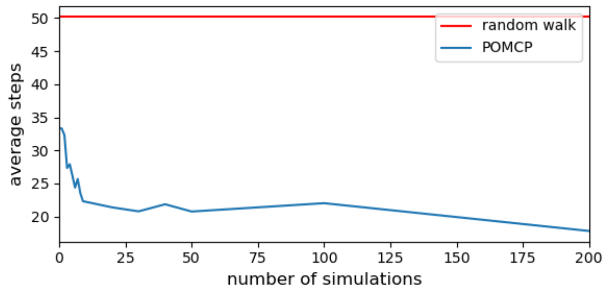


Figure 5. Comparison of different number of simulations and the average steps required to find the desired object.

As expected, the number of steps required to find the object decrease when using more simulations. In comparison to the random walk approach, all number of simulations perform better. It is to notice that these values only describe executions, in which the object was found and not those, in which the algorithm was not able to find the object. This explains why even the previously worse performing number of simulations outperform the random walk approach, since if they find the object, they require fewer steps.
Since more simulations tend to lead to fewer steps required to find the object, it is left to evaluate the trade-off in terms of computational effort when using more simulations, seen in Figure 6.

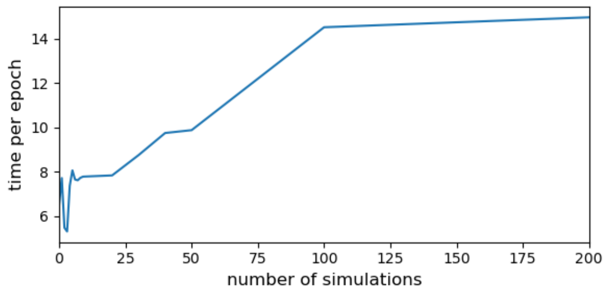


Figure 6. Comparison of different number of simulations and the required time for an epoch in seconds.

As expected, the more simulations are used, the more time is required to run an epoch. However, there are several options to increase performance in terms of time since the evaluation was completely non-optimized and performed on a simple CPU.

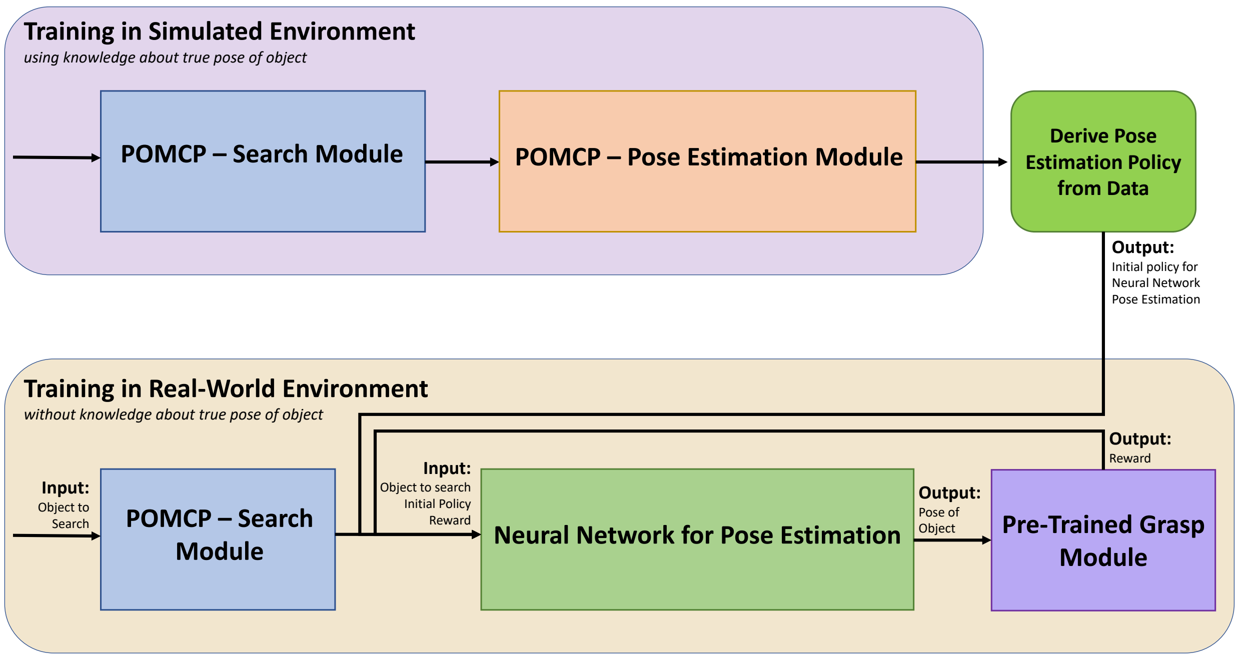

Training in Simulated Environment
using knowledge about true pose of object
POMCP – Search Module
POMCP – Pose Estimation Module
Derive Pose Estimation Policy from Data
Output:
Initial policy for Neural Network Pose Estimation
Training in Real-World Environment
without knowledge about true pose of object
Input:
Object to Search
POMCP – Search Module
Input:
Object to search
Initial Policy
Reward
Neural Network for Pose Estimation
Output:
Pose of Object
Pre-Trained Grasp Module
Output:
Reward

Figure 7. Idea for a sim-to-real model using POMCP search and pose estimation during simulation to quickstart training in the real-world.

In general, our experiments have shown that our approach works as expected. The next steps in terms of experiments can now focus on evaluating different scenarios like occlusions and other distractions.

7. Future Work

In order to extend this work to provide a complete module for searching and grasping an object, there are multiple topics of interest. In general, the pose estimation part has to be build and tested and in addition the search module can be extended.

During this work, several extensions for the POMCP search module came to our mind. First of all, the algorithm can be extended by including actions, which rotate the camera. This allows to not only look directly on the table, but to look at positions far away. This makes observations harder to process and probably noisier, however this may be handled by adding the measurements of a range scanner.
Another extension is the change from a two-dimensional search space mapping to a three-dimensional. This would allow to search objects that are placed on top of other objects for example. Since this enlarges the state space, the effect on the performance and run time has to be evaluated.
As a last possible extension, we have in mind to work with an unknown state space (Giuliari et al., 2021). It has been shown to work and this allows to remove the requirement of knowing the state space.

In order to build the pose estimation part, the main idea is to develop a simple POMCP pose estimation module as seen in Figure 1. This will first work only in simulated environments since it probably requires the true pose in order to define a reasonable reward function. However, using this in simulation allows us to extract a simple policy for pose estimation. This policy may then be used as an initial policy for a neural network for pose estimation. Then, a pre-trained grasp module like DexNet (Mahler et al., 2017) could be used to generate a reward in the real-world, depending on how good the grasping was performed. This reward can then be used to train the pose estimation network, as can be seen in Figure 7. This concept, if realisable, would provide a module to search and grasp an object and improve over time.

8. Conclusion

In this work, we have provided a new and extensible environment for AVS using POMCP. This environment allows to simulate real-world like scenarios and can easily be adapted to create harder and more complex environments.
Moreover, we have provided a POMCP based solution in order to find a desired object in this environment. In addition, we introduced a new type of grid-like observations, which let the agent reason when an object is found or not based on what has been observed.
The environment provided also allows for additional tasks like pose estimation and grasping and therefore it builds the foundation for future works, in which a robot may find and grasp an object.

References

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

Bertsekas, D. and Castanon, D. Rollout algorithms for stochastic scheduling problems. In *Proceedings of the 37th IEEE Conference on Decision and Control (Cat. No.98CH36171)*, volume 2, pp. 2143–2148 vol.2, 1998. doi: 10.1109/CDC.1998.758655.

Boucherie, R. J. and Van Dijk, N. M. *Markov decision processes in practice*. Springer, 2017.

Chandrasekaran, B. and Conrad, J. M. Human-robot collaboration: A survey. In *SoutheastCon 2015*, pp. 1–8, 2015. doi: 10.1109/SECON.2015.7132964.

Chaplot, D. S., Gandhi, D. P., Gupta, A., and Salakhutdinov, R. R. Object goal navigation using goal-oriented semantic exploration. *Advances in Neural Information Processing Systems*, 33, 2020.

Coulom, R. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, pp. 72–83, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540755373.

Coumans, E. and Bai, Y. Pybullet, a python module for physics simulation for games, robotics and machine learning. http://pybullet.org, 2016–2021.

Emami, P., Hamlet, A. J., and Crane, C. Pomdpy: An extensible framework for implementing pomdps in python. 2015.

Feinberg, E. and Shwartz, A. *Handbook of Markov Decision Processes: Methods and Applications*. International Series in Operations Research & Management Science. Springer US, 2012. ISBN 9781461508052. URL https://books.google.de/books?id=TpwKCAAAQBAJ.

Giuliari, F., Castellini, A., Berra, R., Bue, A. D., Farinelli, A., Cristani, M., Setti, F., and Wang, Y. POMP++: pomcp-based active visual search in unknown indoor environments. *CoRR*, abs/2107.00914, 2021. URL https://arxiv.org/abs/2107.00914.

He, W., Li, Z., and Chen, C. L. P. A survey of human-centered intelligent robots: issues and challenges. *IEEE/CAA Journal of Automatica Sinica*, 4(4):602–609, 2017. doi: 10.1109/JAS.2017.7510604.

Kaelbling, L. P., Littman, M. L., and Cassandra, A. R. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134, 1998. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(98)00023-X. URL https://www.sciencedirect.com/science/article/pii/S000437029800023X.

Kocsis, L. and Szepesvári, C. Bandit based monte-carlo planning. volume 2006, pp. 282–293, 09 2006. ISBN 978-3-540-45375-8. doi: 10.1007/11871842_29.

Mahler, J., Liang, J., Niyaz, S., Laskey, M., Doan, R., Liu, X., Ojea, J. A., and Goldberg, K. Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics. *CoRR*, abs/1703.09312, 2017. URL http://arxiv.org/abs/1703.09312.

Mooney, C. Z. *Monte carlo simulation*. Number 116. Sage, 1997.

Schmid, J. F., Lauri, M., and Frintrop, S. Explore, approach, and terminate: Evaluating subtasks in active visual object search based on deep reinforcement learning. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5008–5013. IEEE, 2019.

Silver, D. and Veness, J. Monte-carlo planning in large pomdps. In Lafferty, J., Williams, C., Shawe-Taylor, J., Zemel, R., and Culotta, A. (eds.), *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010. URL https://proceedings.neurips.cc/paper/2010/file/edfbe1afcf9246bb0d40eb4d8027d90f-Paper.pdf.

Tesauro, G. and Galperin, G. On-line policy improvement using monte-carlo search. In Mozer, M. C., Jordan, M., and Petsche, T. (eds.), *Advances in Neural Information Processing Systems*, volume 9. MIT Press, 1997. URL https://proceedings.neurips.cc/paper/1996/file/996009f2374006606f4c0b0fda878af1-Paper.pdf.

Wang, Y., Giuliari, F., Berra, R., Castellini, A., Bue, A. D., Farinelli, A., Cristani, M., and Setti, F. POMP: pomcp-based online motion planning for active visual search in indoor environments. *CoRR*, abs/2009.08140, 2020. URL https://arxiv.org/abs/2009.08140.

Ye, X., Lin, Z., Lee, J.-Y., Zhang, J., Zheng, S., and Yang, Y. Gaple: Generalizable approaching policy learning for robotic object searching in indoor environment. *IEEE Robotics and Automation Letters*, 4(4):4003–4010, 2019.

Zeng, R., Wen, Y., Zhao, W., and Liu, Y.-J. View planning in robot active vision: A survey of systems, algorithms, and applications. *Computational Visual Media*, pp. 1–21, 2020.