Autonomous Agents and Multi-Agent Systems (2025) 39:7
https://doi.org/10.1007/s10458-025-09687-8

Reflexive anticipatory reasoning by BDI agents

Jomi Fred Hübner[1] · Samuele Burattini[2] · Alessandro Ricci[2] · Simon Mayer[3]

Accepted: 2 January 2025

Abstract
This paper investigates how predictions about the future behaviour of an agent can be exploited to improve its decision-making in the present. Future states are foreseen by a simulation technique, which is based on models of both the environment and the agent. Although the environment model is usually taken into account for prediction in artificial intelligence (e.g., in automated planning), the agent model receives less attention. We leverage the agent model to speed up the simulation and as a source of alternative decisions. Our proposal bases the agent model on the practical knowledge the developer has given to the agent, especially in the case of BDI agents. This knowledge is thus exploited in the proposed future-concerned reasoning mechanisms. We present a prototype implementation of our approach as well as the results from its evaluation on static and dynamic environments. This allows us to better understand the relation between the improvement in agent decisions and the quality of the knowledge provided by the developer.

Keywords Anticipatory thinking · Cognitive agents · BDI agents

1 Introduction

We illustrate the problem that motivates this research using the classical scenario where an agent has to move from its location to a target destination in a grid-like environment that contains obstacles. To implement such an agent, we consider a developer who, when reading the problem description, has imagined the scenario as shown in Fig. 1a. The developer

✉ Jomi Fred Hübner
jomi.hubner@usfc.br

Samuele Burattini
samuele.burattini@unibo.it

Alessandro Ricci
a.ricci@unibo.it

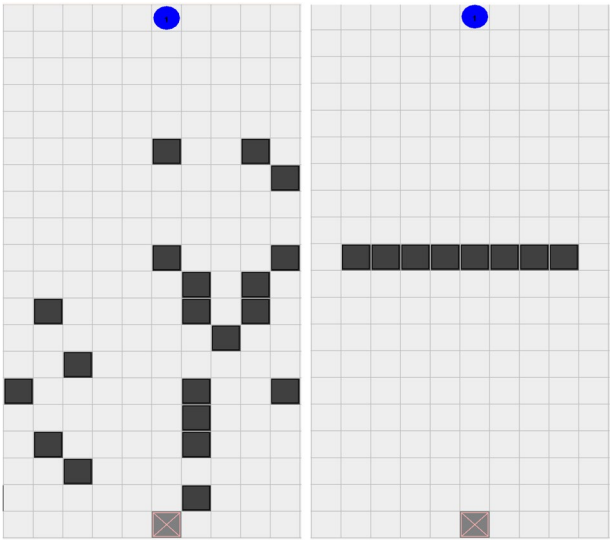
Simon Mayer
simon.mayer@unisg.ch

[1] Federal University of Santa Catarina, Florianópolis, Brasil

[2] University of Bologna, Cesena, Italy

[3] University of St.Gallen, St.Gallen, Switzerland

Published online: 23 January 2025



(a) Developer Environment Model (b) Actual Environment

Fig. 1 Grid Environment. The agent (in dark blue) departs from the top and its destination is at the bottom. Possible actions are *s*, *sw*, *se*, *w*, *e*, *n*, *nw*, *ne*. The cost of *s*, *w*, *e*, and *n* is 1 and the cost of *sw*, *se*, *nw*, and *ne* is $\sqrt{2}$ (Color figure online)

decided thus to implement a straightforward strategy: in each step, the agent moves from its current location to an adjacent location that is nearest to the destination. Now consider that, when the agent is deployed, the environment is not as imagined by the developer; but it rather resembles Fig. 1b. In this environment, the agent however fails to reach its final destination when following the strategy provided by the developer. When it arrives above the wall, the possible movement that places it near the destination is, for instance, going right. After moving to the right, the best move is going left. The agent finishes by entering a loop.

We may notice that the root cause of this problem is the developer's misconception of the environment, which leads to a non-optimal strategy (the greedy approach). Of course, in our example, the problem can be solved by revising the agent strategy as soon as we identify the misconception. However, for some applications, this misconception is not so easily identified and fixed. This is, for instance, the case when a developer is unable to obtain an accurate model of the environment. This is true not only for relatively unknown environments, but also for environments that exhibit considerable dynamics at run time. The relevance of designing agents that effectively navigate such dynamic environments —a prime example is the hypermedia environment of the World Wide Web, which is increasingly inhabited by autonomous agents [1]— has recently gained attention in the AAMAS community and beyond (e.g., [2, 3]). Moreover, this problem is faced when agents move among environments at run time. In all these cases, we cannot rely on reprogramming the agent. Rather, ideally, the agent should be able to adapt itself to different environments.

In this paper, we propose to address this problem using *anticipatory thinking* [4–7]. If the agent is able to foresee that its program will not succeed, it can either stop wasting energy on

it and give up or try to find alternative options. In the search for other options, we propose to exploit the very program the developer wrote. We assume that developers know the application and represent their expertise in that program. We are thus investigating how to exploit this knowledge while searching for an alternative behaviour when the agent foresees problems ahead. Returning to our example, consider that, based on its program, the agent has three possible actions (A, B, and C) for a situation. Action A places the agent at a distance of 10 steps from the destination, while actions B and C place it 11 and 12 steps away, respectively. Considering the developer strategy, action A is thus preferred. However, after action A the agent will be blocked. We are looking for an agent capable to decide for action B. It discovers that action A will not work and choses B instead of C because it is closer to the developer strategy.

The problem of planning ahead to find actions to achieve a goal is not a novelty, of course. Several AI techniques are proposed for that: search, planning, reinforcement learning, Monte Carlo search, etc. These techniques rely mostly on the specification of an environment model. In our research, we plan to also exploit the agent model, and specifically the knowledge a developer has equipped the agent with. We delimit this investigation to agents that have this knowledge available in a plan library, like PRS and BDI architectures [8–10]. In these architectures, the agent program is a set of plans, and these plans represent options to achieve the agent's goals. The agent is autonomous in choosing among these options to better adapt its behaviour to runtime circumstances.

In our proposal, the problem introduced in this section and that motivates this research is seen as a particular case of a more general problem: how can an agent foresee that an undesirable situation is expected to occur in the future if it keeps following its strategy; and how to adapt its present behaviour for that? An obvious undesired situation occurs if the agent becomes unable to achieve a goal, but we may consider other sorts of such situations, for example, undesired location on the path to the goal, norm violations, and missing deadlines.

We describe the background model for our proposal in Sect. 2, especially how future and time are conceived. The first contribution is presented in Sect. 3 and focuses on possible types of problems in the future and how they can be foreseen. The second contribution is presented in Sect. 4 and consists of searching for alternative options based on the agent program. The section also reports some experimental results on different strategies to implement that search. The proposal is extended and evaluated for stochastic and dynamic environments in Sect. 5. We discuss the proposal, its results, main contributions, limitations, and related work in Sect. 6. The appendix A briefly describes how the proposal is implemented.

2 Future simulation

We start our investigation in a simple context where there is only one agent and the environment changes only as a result of actions performed by this agent (a static environment). The environment is also fully observable, deterministic, and discrete [11]. The evolution of the environment is modeled by the function e based on the set of states S and actions A:

$$e : S \times A \rightarrow S$$

Given a state s and an action a, $e(s, a)$ is the next state of the environment. For example, if the state is $s_{2,1}$ (the agent is at location 2,1) and the action is w (west), $e(s_{2,1}, w) = s_{1,1}$. We assume that e models the environment perfectly. However, it is not necessarily the model that the developer has in mind when programming the agents.

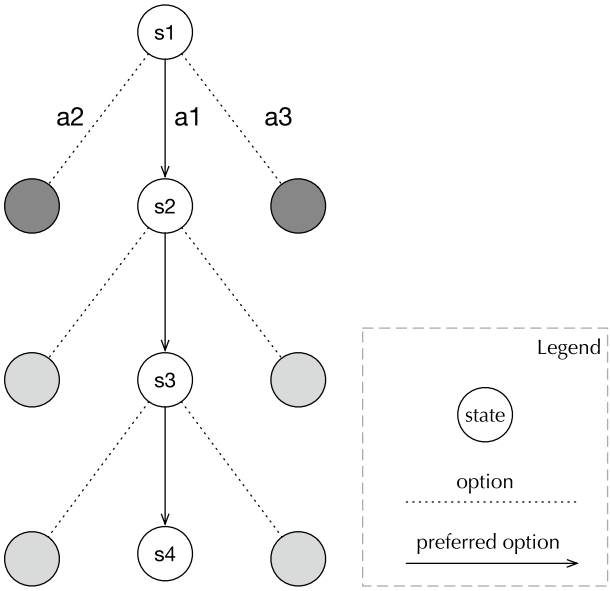


Fig. 2 Time model. Present is the state s_1. The state s_1 is transformed into another state s_2 by action a_1. Options for $s_1 = \pi(s_1) = \langle a_1, a_2, a_3 \rangle$. $e(s_1, a_1) = s_2$. The history is $\langle s_1, s_2, s_3, s_4 \rangle$

Besides a model of the environment, we also have a model of how the agent decides its action. The agent decision-making is based on its plan library and is modeled by the function π (agent *policy*):

$$\pi : S \to A^n$$

Given a state s, $\pi(s)$ is a sequence of possible *options*, ordered by their *preference*. For example, if the state is $s_{1,1}$ and $\pi(s_{1,1}) = \langle s, sw, se, w, e, n, nw, ne \rangle$, the agent prefers to go south (actions s, sw, se), then west (w), then east (e), then north (actions n, nw, ne).[1] This function represents the knowledge the developer gave to the agent: the preferred actions for every state ordered according to a strategy.

With the environment and agent models, we can predict the future using *simulation* [5]. The function f models that simulation:

$$f : S \times \mathbb{N} \to S$$

Given a state s and a natural number t, $f(s, t)$ is the state of the environment t steps ahead. This function can be defined as follows:

$$f(s,t) \stackrel{\text{def}}{=} \begin{cases} s & \text{if } t = 0 \\ f(s', t-1) & \text{if } t > 0 \text{ and } s' = e(s, \pi(s)_{[1]}) \end{cases}$$

where s' is the state produced by the first option, the most preferred option, of the agent policy ($\pi(s)_{[1]}$).

[1] We use $[i]$ to denote the i-th element of a sequence and $[-1]$ to refer to its last element. For example, if $\pi(s_{1,1}) = \langle s, sw, se, w, e, n, nw, ne \rangle$, then $\pi(s_{1,1})_{[1]} = s$ and $\pi(s_{1,1})_{[-1]} = ne$.

The future time line and its states are illustrated in Fig. 2. Time is thus discrete and evolves by agent actions. By simulating the future with function f, the future is linear, as well as the computational complexity to build it. This property of our conceptualization of time is possible because we assume that both the agent policy π and the environment e are known. In other techniques, such as automated planning [12], the future is branching for all possible action because the agent policy is not considered (indeed, the objective in this case is to create a policy), which usually leads to exponential complexity. In our proposal, we are able to compute *the* future instead of possible futures.

In what follows, we use this model of the future to identify future problems (Sect. 3) and find alternative options to avoid them (Sect. 4).

3 Problem prediction

In this section, we start the development of a deliberation algorithm that improves the agent performance based on the future outcomes of its behaviour. The focus of the algorithm is on deciding the appropriate course of actions to achieve the agent's goals (practical reasoning). We depart from a quite simple algorithm (Algorithm 1) and improve it throughout this paper. This algorithm considers a single goal g, which is to bring the environment to the state g ($g \in S$). To achieve this goal, a policy π is built from the agent plan library. The agent then always decides on the most preferred option of that policy. This agent reproduces the blocking problem of the grid example presented in the introduction.

Algorithm 1 Basic agent reasoning cycle.

Input: goal g, plan library pl

$\pi \leftarrow$ a policy for g based on pl
$s \leftarrow perception()$ // s is the current state
while $s \neq g$ **do**
  $options \leftarrow \pi(s)$
  $a \leftarrow options_{[1]}$ // a is the preferred option for s
  do(a)
  $s \leftarrow perception()$

Our first improvement for this algorithm is to address future problems. The boolean function p models what a particular application considers a problem:

$$p : S^n \rightarrow \mathbb{B}$$

The domain of this function is a sequence of states representing some *history* of the environment. The function maps a history to *true* in case it configures a problem. For example, given a goal g, the problem of missing a deadline d can be modeled as

$$p^d(s_1, \ldots, s_n) \stackrel{\text{def}}{=} n > d \wedge s_i \neq g \text{ for } i = 1..n$$

any history longer than d that does not have g as one of its states is considered a problem.

Another usual kind of problem is to be in an undesirable state. For example, the agent does not want to be wet, does not want to violate some norm (when norms forbid some states), etc. Given a set U of undesired states, this problem can be modeled as

$$p^U(s_1, \ldots, s_n) \stackrel{\text{def}}{=} s_i \in U \text{ for } i = 1..n$$

For the grid example, a useful definition of problem is to be in an already visited state. If the agent comes back to a state, it will enter a loop and never achieve its goal. The problem function for this case is:

$$p^l(s_1, \ldots, s_n) \stackrel{\text{def}}{=} i \neq j \wedge s_i = s_j \text{ for } i = 1..n \text{ and } j = 1..n$$

Several problem functions can be used to compose the set P of considered problems. For example, $P = \{p^d, p^U, p^l\}$.

The problem functions (P) and the future simulation function (f) are combined into a new function *matrix* to be used in the next version of the agent reasoning algorithm. Starting from an initial state, the *matrix* function simulates the future until it finds a history that achieves the goal without problems. We represent a history h ($h \in S^n, n \geq 0$) as a sequence of states $\langle s_1, \ldots, s_n \rangle$, the last state of the history as $h_{[-1]}$, and an empty history as $\langle\rangle$. Some history h achieves the goal g if $g = s_i$ for $i = 1..n$. The concatenation of two sequences is produced by the operator $\oplus$. We have all the elements to define the *matrix* function:

$$matrix : S \times S \rightarrow S^n$$

$$matrix(s, g) \stackrel{\text{def}}{=} m(g, \langle s \rangle)$$

$$m(g, h) \stackrel{\text{def}}{=} \begin{cases} \langle\rangle & \text{if } \exists_{p \in P}\ p(h) \\ h & \text{if } h_{[-1]} = g \\ m(g, h \oplus \langle f(h_{[-1]}, 1) \rangle) & \text{otherwise} \end{cases}$$

The *matrix* function begins with the state s and builds the history incrementally using the future simulation one step ahead from the last state of a history h ($f(h_{[-1]}, 1)$ gives the next state of h when the agent's preference is followed). The function returns an empty history if a problem is found before the achievement of g.

The improved version of our agent algorithm using the matrix function is shown in Algorithm 2 (lines 4–5).[2] If we compose the problem functions ($P = \{p^d, p^U, p^l\}$), we have an agent that only commits to a goal if it foresees that the goal is achievable before the deadline d and no undesired states U are produced meanwhile. Otherwise, no energy is spent on actions for the goal.

From a broader perspective, an agent with practical reasoning as defined in Algorithm 1 puts energy into trying to achieve unachievable goals. By changing the practical reasoning to Algorithm 2, with the *same* set of plans, the agent performance is increased since no action is executed, the agent does not commit to an unachievable goal. The improvement is not the consequence of a better agent program but the result of a better practical reasoning (which considers the future). Therefore, this improvement is general for every agent using

[2] We place the future verification (lines 4–5) before the main loop because we are considering a perfect model of the environment. We can thus be sure that the goal will not be achieved. However, if the environment model is not perfect, other implementations should be considered. For instance, wait for better environment conditions where the goal can be achieved.

this new algorithm. However, the list of inputs is longer, especially the environment model might be difficult to obtain. We further discuss these aspects in Sect. 6.

Algorithm 2 Agent reasoning with problem prediction.

Input: goal g, plan library pl, environment e, problem functions P

1 $\pi \leftarrow$ a policy for g based on pl
2 $matrix \leftarrow$ a matrix simulator based on e and P
3 $s \leftarrow perception()$
4 **if** $matrix(s, g) = \langle\rangle$ **then**
5 **return** *failure*
6 **while** $s \neq g$ **do**
7 $options \leftarrow \pi(s)$
8 $a \leftarrow options_{[1]}$
9 do(a)
10 $s \leftarrow perception()$

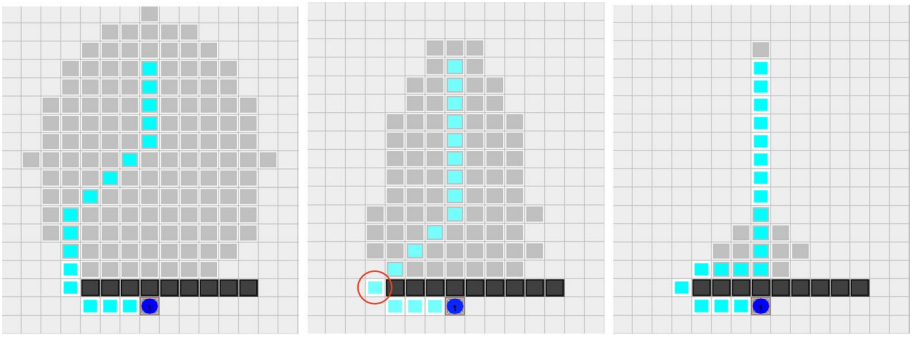
4 Reasoning about future options

The algorithm proposed in the previous section (Algorithm 2) enables an agent to avoid committing to goals it cannot achieve. However, the agent policy may offer additional options that can help the agent achieve its goal. In this section, we explore these other options. More specifically, the agent may select another option for the current state s (given by $\pi(s)_{[i]}$ with $i > 1$) instead of the most preferred option (given by $\pi(s)_{[1]}$). We are, of course, assuming that the agent developer has provided more options, once the developer is an expert on the domain application.

We can start the development of a new algorithm by asking which is the *cause* of not achieving the goal, which is the wrong decision (in terms of selected option) that lead the future to a problematic situation. Finding the causes is a challenging task [13, 14]. Our proposal is simpler: instead of identifying the wrong decision, we search for another option that achieves the goal. For example, if the current state is s, the goal is g, the agent has three options $\pi(s) = \langle a_1, a_2, a_3 \rangle$, and the first option has problems in the future ($matrix(e(s, a_1), g) = \langle\rangle$), then the agent can try the second (a_2) or third (a_3) options. These three options are just one step ahead. Even if these alternatives have problems in the future, we can explore other options further in the future (see Fig. 2).

The search for better future options, as proposed in this section, is based on Breadth-First Search (BFS) [11]. The adaptation of the BFS for our search problem is specified in Algorithm 3.[3] This algorithm returns a sequence of actions that changes the environment

[3] It follows some details about the BFS adaptation. (*i*) A search state is a tuple (s, a, p, c) where s is a state where action a is an option, p is the plan, i.e., the sequence of actions from the initial state to s, and c is the cost of p. (*ii*) The initial states to explore in the search (line 3) are based on the options given by the policy π for a state. The first option ($i = 1$) is not considered because it leads to a future problem, the algorithm is indeed used to find another option ($i > 1$). (*iii*) The search does not end when the goal state is found as in classical search. It ends when a state with a good future is found (line 7), i.e., a state where the policy preference can be followed to achieve the goal. (*iv*) The computational complexity of BFS algorithms is exponential $O(b^d)$. In our case, the branching factor b is the number of options given by π and the search depth d is the number of actions until a state with good future. The computational complexity of the *matrix* function is linear $O(n)$, where n is the number of actions to achieve some goal using the agent preferences as defined by π.



(a) solve_p (b) solve_m (c) solve_f

Fig. 3 Search strategies

from state s to a state where the agent preferences can be followed to achieve g. For instance, considering the path shown in Fig. 3b, the search returns the actions to move the agent form the initial state to the state with a red circle. In that state, the agent preferences can be used again to achieve the goal. Different from traditional search, this proposal does not consider all possible actions for branching the tree search; only actions provided by the agent policy are taken into account. The developer's knowledge (or bias) is used to prune the tree search.

Algorithm 3 Search for alternative options.

Input: initial state s, goal g, a function δ that gives the cost of an action, environment e, problem functions P

1 $\pi \leftarrow$ a policy for g based on pl
2 $matrix \leftarrow$ a matrix simulator based on e and P
3 $open \leftarrow \{(s, a, \langle\rangle, \delta(a)) \mid i = 2..n \text{ and } a = \pi(s)_{[i]}\}$ // *open contains options to explore, n is the number of options given by $\pi(s)$*
4 **while** $open \neq \{\}$ **do**
5 $\quad (s, a, p, c) \leftarrow$ remove the search state from *open* with min cost c
6 $\quad s' \leftarrow e(s, a)$ // *s' is the next state after s and action a*
7 $\quad$ **if** $matrix(s', g) \neq \langle\rangle$ **then**
8 $\quad\quad$ **return** $p \oplus \langle a \rangle$ // *a plan p was found*
9 $\quad open \leftarrow open \cup \{(s', a, p \oplus \langle a \rangle, c + \delta(a)) \mid i = 1..n \text{ and } a = \pi(s')_{[i]}\}$
10 **return** $\langle\rangle$

With the search capability, we have a third version of the agent reasoning cycle algorithm (Algorithm 4). The added functionality is used when the current policy causes a problem in the future (line 4 of the algorithm). In that case, the search algorithm is used to compute a sequence of actions to avoid the problem (line 5). This sequence is then transformed into a policy in line 8. Figure 3a shows an example of the path to the destination found by this third algorithm. We can notice that the agent changes its decision as soon as possible; we can say "near the present". It thus *deviates* quite a lot from the path proposed

by the developer strategy that is to go south. An alternative path is shown in Fig. 3c, where the agent changes its decision as late as possible; we can say "late in the future" and near the problem. It thus deviates less from the agent strategy. We call the first search strategy solve_p and the latter solve_f (where p stands for "present" and f for "future"). To implement solve_f, we can simply change the initial set of options to explore. While solve_p uses options available for the current state to initialise the *open* set (Algorithm 3, line 3), solve_f initialises that set with all options seen by the matrix that identified the problem. Considering that the state s_4 of Fig. 2 characterises a problem, the initial search states for solve_p are the dark gray states, while these states for solve_f also includes the light gray states. We can consider also a search strategy that balance both of previous strategies. We call it solve_m and it is illustrated in Fig. 3b (where m stands for "in the middle").

Algorithm 4 Agent reasoning with search for alternatives.

Input: goal g, plan library pl, environment e, problem functions P

```
1  π ← a policy for g based on pl
2  matrix ← a matrix simulator based on e and P
3  s ← perception()
4  if matrix(s, g) = ⟨⟩ then
5  |  p ← search(s, g)
6  |  if p = ⟨⟩ then
7  |  |  return failure
8  |  π ← a new policy based on p
9  while s ≠ g do
10 |  options ← π(s)
11 |  a ← options[1]
12 |  do(a)
13 |  s ← perception()
```

4.1 Strategies evaluation

We evaluate the Algorithm 4 and the different search strategies (solve_p, solve_m, and solve_f) in three different grid scenarios, identified by "–", "U", and "H" and illustrated in Fig. 4. The figure also shows the solution found by solve_m and the states visited during the search.[4] We can notice how the search for alternative future options is used to find deviating paths and avoid failure.

The agent used in the experiments has a policy that, for every location, provides a sequence of ordered possible actions. Actions that moves the agent to a wall are not included. In some states, all 8 actions are included. The order is based on how far is the next state from the target destination. For instance, considering a destination in the south,

[4] For these experiments we implemented an extension for the interpreter of the Jason programming language [15]. This extension and instructions to reproduce the experiments are available at https://github.com/jomifred/future-bdi and briefly described in Appendix A.

if $\pi(s_{1,1}) = \langle s, sw, se, w, e, n, nw, ne \rangle$, for the location 1,1, the agent prefers to go south, then southwest, and so on.

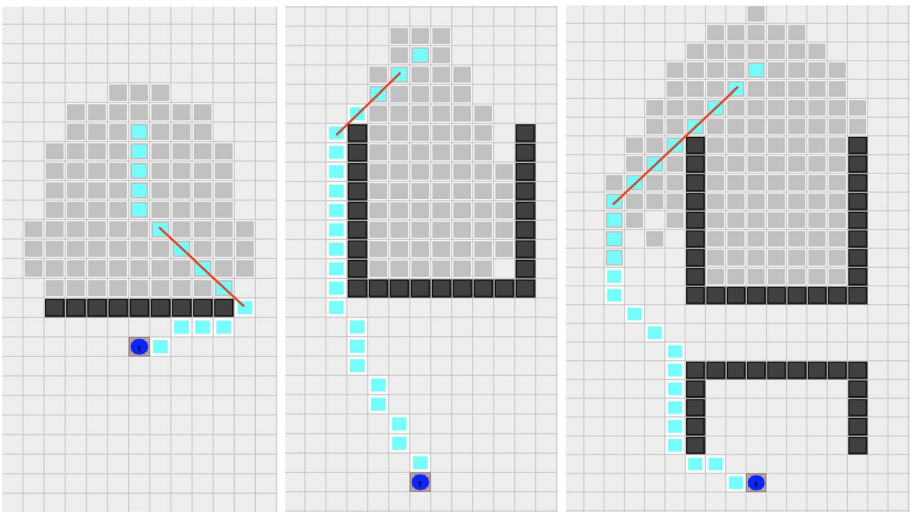
The following metrics were measured during the experiments:

- *Effectiveness*: whether the agent arrives at the destination;
- Plan *efficiency*: the number of actions performed to achieve the goal;
- Search *cost*: the number of states visited by the matrix while searching for alternative options; and
- How much the strategy *deviates* from the agent preferences: how many times the first preferred option is not taken.

Table 1 presents the results of the experiments. The evaluation also includes two new strategies: none for an agent using Algorithm 1 (i.e., the future is not considered) and one for an agent using Algorithm 2 (i.e., considers only problem prediction). As expected, the agent does not arrive at its destination when using these two strategies. However, the number of actions for one is zero; no energy is spent. We can also remember that the computational complexity of this strategy is linear. These two strategies are shown only for the first scenario, since the results for the others are similar without relevant novelties.

In the scenario "–", solve_f deviates less from the agent preference. This strategy is also the cheapest in terms of search cost. The policy specified by the developer works well in this scenario. Scenario "U" is more favorable for solve_p, which decides to deviate as soon as possible, so its search cost is better than others. In scenario "H", we do not see significant differences among the strategies. Indeed, no strategy plays well in this scenario. The agent policy is not adequate for this scenario and a lot of search is used to remediate.

The results on deviation depend on the scenario, but, in general, as expected, solve_p deviates more than solve_m, which deviates more than solve_f. The evaluation of



(a) "–" (b) "U" (c) "H"

Fig. 4 Grid Scenarios. The agent (in dark blue) departs from the top and its destination is at the bottom. The taken path is marked in light blue. Gray places are visited during the search. The red line marks the deviating states found by the search. Strategy is solve_m (Color figure online)

Table 1 Results for search strategies

Scenario	Strategy	Effective	Search cost	Plan efficiency	Deviation
–	none	no	∞	∞	∞
–	one	no	10	–	0
–	solve_p	yes	8414	15	7 (46%)
–	solve_m	yes	6415	15	4 (26%)
–	solve_f	yes	2578	19	3 (15%)
U	solve_p	yes	3489	23	3 (13%)
U	solve_m	yes	6032	23	3 (13%)
U	solve_f	yes	6890	26	3 (12%)
H	solve_p	yes	12353	25	11 (44%)
H	solve_m	yes	11372	25	11 (44%)
H	solve_f	yes	11053	28	11 (39%)

Table 2 Results with different policies

Scenario	Strategy	States for	States for	Improvement
H	solve_p	12353	7869	36%
H	solve_m	11372	7353	35%
H	solve_f	11053	7132	35%

efficiency indicates that solve_m and solve_p are equally good strategies. However, they have a higher search cost (in terms of visited states) than solve_f.

4.2 Pruning evaluation

The results shown in Table 1 considers an agent policy that provides 8 options. However, if the developer knows the environment and the problem enough so that actions towards the north are unnecessary to achieve the goal, the number of options can be reduced. The search space can thus be also reduced. Table 2 contains the number of visited states in the scenario "H" for the policy with 8 options and for the policy with 5 options (without north actions). The search time is reduced in 35%, which is in line with the reduction in the number of actions. For comparison, a classical BFS algorithm without pruning visits approximately $b^d = 8^{25}$ states to solve this problem (b is the number of possible actions for each state and d is the number of actions to arrive at the destination; assuming an unbounded grid scenario). Indeed, the number of visited states of our proposal is similar to an A* implementation. This is expected since the order of options in the agent policy works as a heuristic.

This experiment illustrates how the knowledge provided by the developer and implied in the agent policy can be useful to improve the search performance. This aspect is further discussed in Sect. 6.

5 Dynamic and stochastic environments

As discussed previously, the quality of the prediction depends on the environment model. In this section, we investigate the case where the environment is dynamic (it changes by other actors than our agent) and stochastic (the next state is not certainly determined by an action). In this context, an issue for the agent is how far to simulate the future, considering the uncertainty about future states. To answer that question, we revise the environment model and adapt the matrix function accordingly.

A stochastic environment can be modeled as follows [16]:

$$e : S \times A \times S \to [0, 1]$$

Given a state s, an action a, and another state s', $e(s, a, s')$ is the probability that action a transforms s into s'. Based on this new environment model, the function that predicts the state t steps ahead also provides the certainty of that future state:

$$f : S \times \mathbb{N} \to S \times [0, 1]$$

$$f(s,t) \stackrel{\text{def}}{=} \begin{cases} \langle s, 1 \rangle & \text{if } t = 0 \\ \langle g_{[1]}, c \rangle & \text{if } t > 0 \end{cases}$$
$$g = f(s', t-1)$$
$$s' = \arg\max_{s'' \in S} e(s, a, s'')$$
$$a = \pi(s)_{[1]}$$
$$c = g_{[2]} \cdot e(s, a, s')$$

where s' is the most probable state produced by the first and preferred option a of the agent policy, and c is the accumulated probability of that transition. We are thus keeping the future linear by considering the most likely next state produced by the agent's preference. The computational complexity of the matrix function is thus kept linear as well.

To handle the uncertainty related to the environment, we add a new parameter in the matrix function: the *required certainty* (rc) to continue computing the future. Its definition is thus changed to:

$$matrix(s, g) \stackrel{\text{def}}{=} m(g, \langle s \rangle, 1)$$
$$m(g,h,c) \stackrel{\text{def}}{=} \begin{cases} \langle\rangle & \text{if } c < rc \\ \langle\rangle & \text{if } \exists_{p \in P}\ p(h) \\ h & \text{if } h_{[-1]} = g \\ m(g, h \oplus \langle n_{[1]} \rangle, c \cdot n_{[2]}) & \text{otherwise} \end{cases}$$
$$n = f(h_{[-1]}, 1)$$

The agent algorithm is also adapted (Algorithm 5). The use of the matrix is moved inside the main loop (line 7). In the previous algorithms, we assumed that if the matrix does not find a history that achieves a goal, it means that such a history does not exist. This verification can thus be done before acting (in the main loop) – line 4 of Algorithm 2. This assumption is no longer valid. It might be the case that the matrix stopped due to the lack of certainty. In these cases, the agent selects a random option and waits for better circumstances (line 14 of Algorithm 5).

Algorithm 5 Agent reasoning in dynamic environments.

Input: goal g, plan library pl, environment e, problem functions P, required certainty rc

$\pi \leftarrow$ a policy for g based on pl
$matrix \leftarrow$ a matrix simulator based on e and P
$s \leftarrow perception()$
while $s \neq g$ **do**
  $options \leftarrow \pi(s)$
  $a \leftarrow options_{[1]}$ // *preferred option from* π
  if $matrix(s, g) = \langle\rangle$ **then** // *we are not* rc *sure that* g *is achieved by* π
    $p \leftarrow search(s, g)$
    if $p \neq \langle\rangle$ **then**
      $\pi \leftarrow$ a new policy based on p // *new policy*
      $options \leftarrow \pi(s)$
      $a \leftarrow options_{[1]}$
    else
      $a \leftarrow$ a random option from $options$
  do(a)
  $s \leftarrow perception()$

With the matrix function and search algorithm placed inside the loop, the agent may lose reactivity, mainly because the complexity of the search algorithm is exponential in the worth case. For example, if the search takes 10 min to finish, the agent will not do perception and react to new states for 10 min! The solution for this problem likely requires that the search runs in parallel, it is thus left for future work. In this section, we are more focused on evaluating how to exploit the future in a dynamic environment and how much time is consumed by this algorithm as we change the required certainty.

5.1 Strategy evaluation

We ran some experiments to evaluate the performance of the proposed algorithm in dynamic scenarios. We delimited the experiment to a modified version of our grid scenario where walls dynamically appear or disappear. These changes in the walls are produced randomly, i.e., they are not the consequence of an agent action. In this environment, the agent is not sure about the next state after an action. For example, the south action usually places the agent one line below in the grid; however, if a wall appears below the agent, the action fails and the agent stays put. The model of this environment assigns more probability for the agent being one line below and less probability to stay put. Figure 5 illustrates the new scenario.

The following metrics were measured during the experiments:

- *Effectiveness*: whether the agent successfully crosses the grid before a deadline;
- *Efficiency*: the traveled path length, i.e., the sum the actions' cost performed by the agent to achieve its goal; and

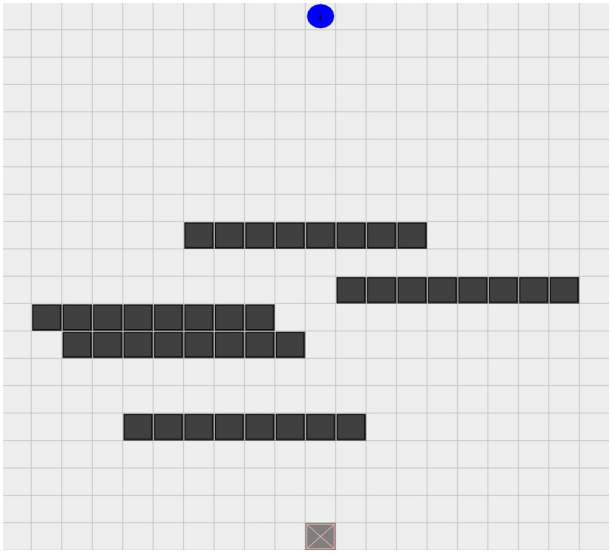


Fig. 5 Dynamic Scenario. The agent starts on the top and have to cross the grid to the bottom

- Search *cost*: the number of states visited by the matrix function while searching for alternative options.

We considered different configurations for the scenario, varying the following parameters:

- The environment's dynamicity (p), from environments that do not change ($p = 0$) to environments that change every cycle ($p = 1$).[5] A change is either the addition of a new wall or the removal of an existing wall. This parameter is used to define the function e that models the environment. Considering that pb is the probability that a block appears or disappear in a cycle as determined by p, the west action is defined as follows

$$e(s_{x,y}, w, s_{x,y}) = \begin{cases} 1 - pb & \text{if } s_{x,y} \text{ has a block} \\ pb & \text{otherwise} \end{cases}$$

$$e(s_{x,y}, w, s_{x+1,y}) = \begin{cases} pb & \text{if } s_{x+1,y} \text{ has a block} \\ 1 - pb & \text{otherwise} \end{cases}$$

$$e(s_{x,y}, w, s') = 0 \text{ for states } s' \neq s_{x,y} \wedge s' \neq s_{x+1,y}$$

$$pb = p \cdot \frac{wallsize}{gridsize} \cdot \frac{1}{2}$$

- How far the agent looks ahead in the future, defined by the required certainty (rc). For instance, if $rc = 1$, the agent only considers future states that it can be fully sure about. If $rc = 0$, the agent does not care about certainty. This value is used by the *matrix* function.
- The agent search strategy (s). Values for s are solve_m and random. The first strategy is described in Sect. 4. The random strategy consists of simply selecting a random

[5] An environment cycle starts with an environment state and ends after the execution of an action by the agent.

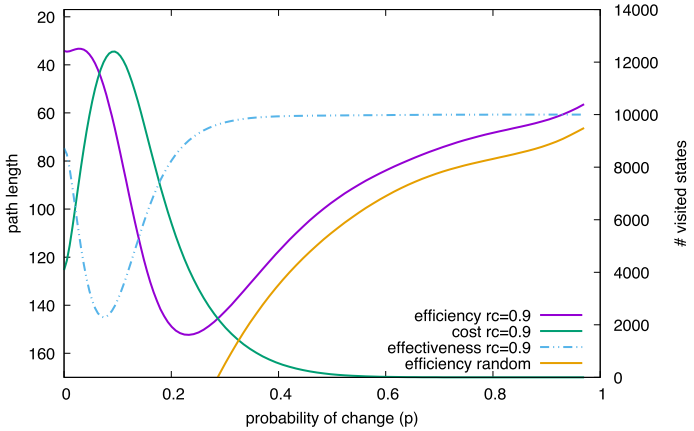


Fig. 6 Results of the search using strategy solve_m and $rc = 0.9$

option when the preferred option has problems in the future (the Algorithm 5 without lines 8–13). This strategy is always effective in this scenario and has no cost (i.e., no search).

Figure 6 presents the results of the configuration with $s =$ solve_m, $rc = 0.9$, and p varying from 0 to 1 (x-axis). The results are based on the execution of 1415 episodes. An episode starts with the agent at (15,0) and ends when either it arrives at (15,19) or the deadline is reached. The time necessary to execute an episode is proportional to the number of visited states. We thus considered that every episode that visited more than 13000 states is missing the deadline; the episode is then stopped and count as ineffective. Effectiveness is measured by the percentage of executions that the agent has crossed the grid before the deadline (to read the effectiveness in the graph, y=100000 corresponds to the fact that the agent is effective in 100% of the executions). Efficiency is measured by the mean of the length of paths used to cross the grid. Cost is measured by the mean of the number of states visited in the search (considering both successful and timeout executions). The figure also includes the efficiency of the random strategy.

Based on these results, it follows some remarks about the performance of the solve_m strategy in dynamic contexts with required certainty of 0.9.

1. The best efficiency, i.e., the shortest path to achieve the goal, is obtained when the environment is less dynamic ($p < 0.1$), since the agent can find alternative options and they remain good options throughout the episode. This efficiency obviously has a cost. The search in this configuration visits several states. Particularly when $p \simeq 0.1$, the cost may be so great that the agent is ineffective and misses the deadline.
2. When $p > 0.5$, the agent has not enough certainty about the future of its environment to properly search for alternative options, and thus few states are visited (low cost). When its first preference cannot be applied (since it leads to a problem in the future) and no alternative option can be found (for lack of certainty), the agent simply acts randomly (Algorithm 5, line 14). If it becomes stuck, either the wall blocking it may disappear or a random movement will place it in a state where its preferences can be applied again, hopefully achieving its goal. Thus, effectiveness is high in more dynamic environments.

The strategy in this type of environment is to not spend computational time looking for alternatives but to wait for a better state.

3. When $p \simeq 0.2$, we have the worst performance in terms of efficiency and cost. Effectiveness-wise, it is not that good either. Given this dynamicity and $rc = 0.9$, the agent can dedicate time to the search of options. When using these options, the agent moves out of its policy to avoid walls, but these walls will likely disappear (especially walls that are far from the agent). The deviation results in extra steps, which reduces efficiency. The alternative options (which are expensive to find) might be inefficient in the future when the walls are not there anymore or others are added.
4. When $p \simeq 0.1$, the agent crosses the grid only 30% of the episodes! Although the random strategy could be adopted, since it is very effective, its efficiency is also very low. Indeed, this configuration is quite a challenge for the agent using $rc = 0.9$. Other values of rc can perform better in this scenario, as demonstrated by the next experiments.
5. The efficiency of the random strategy gets better as the environment becomes more dynamic. With $p > 0.4$, its efficiency is similar to solve_m with no cost. Thus, in a very dynamic environment it is worthless to adopt a strategy as solve_m to search for options.
6. As expected, effectiveness and cost are correlated: if the cost increases (due to more time on search), the agent misses the deadline.

The better configuration for an agent with $rc = 0.9$ is an environment that is either more static or highly dynamic. It is particularly difficult to be effective and efficient in an environment with $p \simeq 0.2$. Although we can notice some generality in these results, the specific numbers apply only to our grid scenario. For other scenarios, the same steps can be followed to find out these numbers.

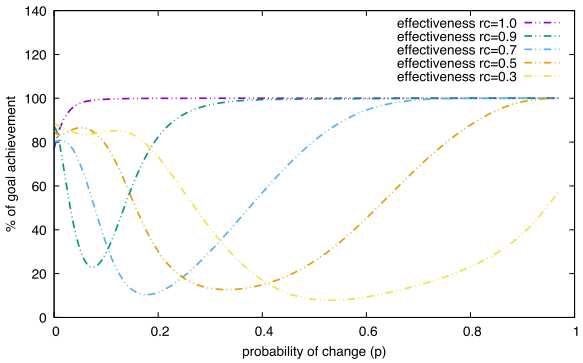
5.2 Required certainty evaluation

In this section we aim to understand how to set up the required certainty of an agent for scenarios with different properties, especially different dynamicity. We approached this objective executing some experiments where we vary the required certainty of the agent and observed its performance in different configurations of the environment.

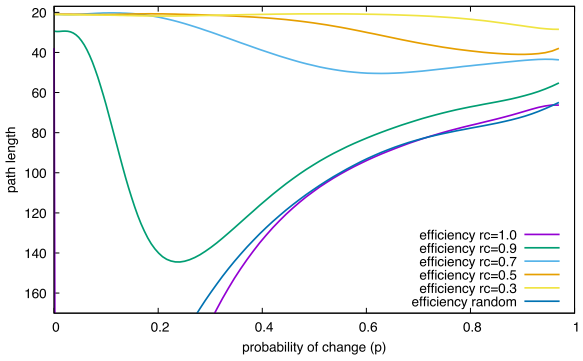
The *effectiveness* of the agent for various rc is depicted in Fig. 7a. These graphs are based on the execution of 6600 episodes. We can notice that as rc decreases, the overall effectiveness decreases. The overall effectiveness can be seen as the area above each line. The results for *efficiency* is shown is Fig. 7b. We can notice that as rc decreases, the overall efficiency increases. The results for the *cost* metric is shown in Fig. 7c. We can notice that as rc decreases, the overall cost increases.

It follows an analysis of the results considering the objective of setting up rc in a low dynamic environment ($p \simeq 0.1$):

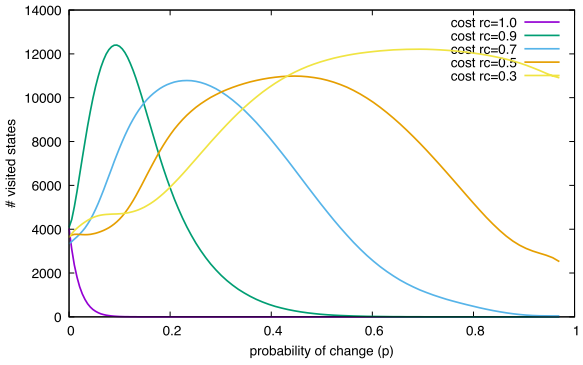
- If effectiveness is important for the application, $rc \leq 0.5$ is preferred; $rc = 1$ is also effective, but its efficiency is very low;
- If efficiency is important, $rc < 0.9$ is preferred; and
- If cost is important, $rc \leq 0.7$ is preferred. $rc = 1$ has a better cost, but it performs quite badly in the other criteria.



(a) Effectiveness



(b) Efficiency



(c) Cost

Fig. 7 Results of the search using strategy solve_m

In general, for $p \simeq 0.1$, $rc \simeq 0.3$ is a good option. For these recommendations of rc, we are considering only values for rc that we evaluate: 0.3, 0.5, 0.7, 0.9, 1.0.

Considering a medium dynamic environment ($p \simeq 0.5$):

- If effectiveness is important, $rc \geq 0.7$ is preferred;

Table 3 Required certainty (rc) for different scenarios. $\ominus$ means that lower values are better. $\oplus$ means that greater values are better

Scenario	Objective			General
	Effectiveness	Efficiency	Cost	
+static ($p \simeq 0.1$)	$rc \in [0, 0.5]$ $\ominus$	$rc \in [0, 0.9]$ $\ominus$	$rc \in [0, 0.7]$ $\ominus$	$rc \simeq 0.3$
($p \simeq 0.5$)	$rc \in [0.7, 1.0]$ $\oplus$	$rc \in [0, 0.7]$ $\ominus$	$rc \in [0.7, 1.0]$ $\oplus$	$rc \simeq 0.7$
+dynamic ($p \simeq 0.8$)	$rc \in [0.5, 1.0]$ $\oplus$	$rc \in [0, 1.0]$ $\ominus$	$rc \in [0.7, 1.0]$ $\oplus$	$rc \simeq 0.9$

- If efficiency is important, $rc \leq 0.7$ is preferred, although the differences are not significant; and
- If cost is important, $rc \geq 0.7$ is preferred.

In general, for $p \simeq 0.5$, $rc \simeq 0.7$ is a good option.

Considering a high dynamic environment ($p \simeq 0.8$):

- If effectiveness is important, $rc \geq 0.5$ is preferred;
- If efficiency is important, all values have similar results; and
- If cost is important, $rc > 0.7$ is preferred.

In general, for $p \simeq 0.8$, $rc \simeq 0.9$ is a good option. Notice that with $rc \simeq 1$ and $p \simeq 1$, we have the same case of the random strategy since no matrix can run in that configuration. In the experiments, we do not identify configurations where $rc \leq 0.3$ is a good choice. Although the efficiency is good (i.e., the few times the agent succeeded in crossing the grid, it did it quite fast), this result is unlikely, as seen in the effectiveness results.

Regarding the correlation between the required certainty (rc) and the dynamicity (p) of the environment, we may conclude that as p increases, the best value for rc also increases (see Table 3). By increasing rc in more dynamic environments, we are relying mostly on the random strategy. In other words, it is not worth spending time computing the future for an uncertain environment; in these cases, random choices could be considered. This conclusion is circumscribed by the experiments we performed in just one scenario. Nonetheless, it seems generally reasonable.

6 Discussion and related work

In this paper, we are looking for improvements in agent decision-making considering the *future*. The fundamental feature of our proposal is to be built on top of the knowledge provided by the developer (hereafter represented by K). This knowledge equips an agent with options and strategies to achieve their goals, as represented by its π function. On the one hand, this knowledge enables the reductions of the search space; on the other hand, it also raises the possibility that we might be overlooking optimal behaviour.

A general view of the impact of K on the quality of agent decisions is depicted in Fig. 8. Each little circle represents a possible agent behaviour as defined by some simple policy.[6] The big square is the set of all possible behaviours. This square represents the search space

[6] A simple policy maps a state to only one action $\pi : S \rightarrow A^n$ for $n = 1$.

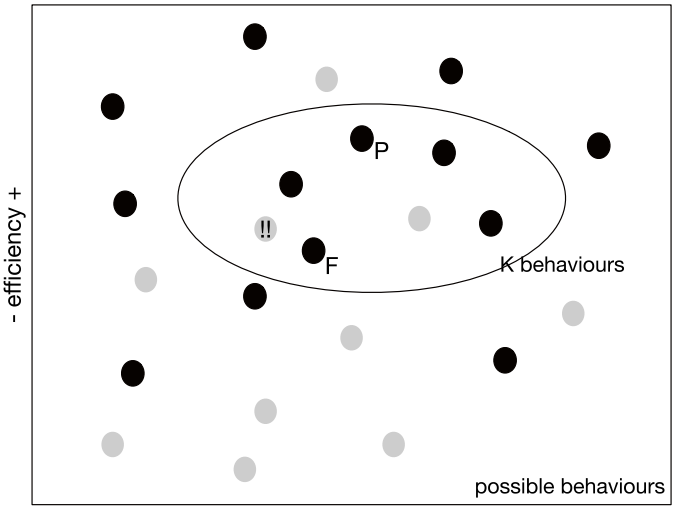


Fig. 8 Search space for behaviours

normally taken into account by techniques such as classical search and automated planning, i.e., to find a simple policy to solve a problem. The ellipse contains possible behaviours as defined by K in our proposal.[7] At the top of the figure we have better behaviours in terms of efficiency, while worst behaviours are placed towards the bottom. Black circles are effective, while gray circles represent a behaviour where the agent does not achieve its goal, even after spending energy on it. Whether a behaviour is effective or not depends on a particular configuration of the environment. Changes in the environment may imply changes in the color of the circles. Finally, the distance between the circles represents how similar their corresponding behaviours are.

The place of the ellipse in the square indicates the quality of K. If placed at the top, the developer knows the application very well and has provided valuable information. The size of the ellipse indicates how many alternatives the developer has considered and thus how much the search space can be pruned. At one extreme, K may provide a single behaviour (a unique circle). At the other extreme, it may provide options covering all possible behaviours. We can contextualise the results presented in Sect. 4.2 using Fig. 8. In that section, we evaluate two policies: the 8-policy and the 5-policy. The first provides 8 actions (and the corresponding directions) for every state; the later provides only 5 actions. The 8-policy implies an ellipse that covers the overall search space (the rectangle of Fig. 8): every behaviour is possible. The 5-policy has the ellipse as shown in Fig. 8: the ellipse area is smaller than the all behaviour space, since north actions are not possible. That experiment shows that K and the implied policies can reduce the search space of behaviours towards a goal.

Regarding the proposed search strategies (solve_p, solve_m and solve_f), suppose that the first option of the agent policy produces the behaviour marked with "!!" in the figure, an ineffective behaviour for the current environment. An agent using Algorithm 5 detects in advance that this option is ineffective and will search for alternatives. The

[7] The policy from K defines several behaviours. It is conceived as presented in Sect. 2: a policy proposes more actions for a state ($\pi : S \rightarrow A^n$ for $n > 0$). It implies many simple policies and thus many circles in the figure.

chosen alternative depends on the strategy. solve_f will likely find an alternative near "!!". solve_p deviates more and will likely find an alternative far from "!!", ideally more efficient.

Since our proposal depends on K, the behaviours defined by K (the ellipse) should ideally be small to prune the search space, but not too small. If it is too small, the agent has no space where to find alternatives for different environments, as illustrated by the problem described in the introduction (Sect. 1). Besides pruning the search space, the policy defined from K, especially the preference among options, can be used as a heuristic to speed up the search. For an example of the pruning effect, we can consider Fig. 4b. Classical BFS search visits all locations to find the path. Our search visits only part of that space (the gray area in the figure). It stops when it finds a state where the policy preference can be applied again, while classical search stops when the destination state is found. Briefly, the branch factor of the search is reduced by the options given by the policy, the depth search is reduced to a deviation from the path (the red line in Fig. 4b), and the preference among options is used as a heuristic.

Considering related work, we can start discussing the similarities between our proposal and automated planning [12], including proposals that integrate planning and BDI, as in [17, 18], for instance. Both approaches look for agents capable of finding a sequence of actions to achieve their goals, as a kind of "auto-programming." However, while planning is essentially based on a model of the environment, we also exploit a model of the agent, represented by its policy π and the derived K of Fig. 8. The agent model allows us to foresee a problem in the future with linear computational complexity. This problem may trigger the search for alternative options. Instead of considering all action options given by the environment, the options given by the agent policy are used in the search. As discussed previously in this section, we reduce the search space by doing so. Of course, these features can be seen as disadvantages: we require a model of the agent policy. Although this model can be built from the agent code/specification in the case of most BDI programming languages, in some applications the agent model is not easily obtained. Moreover, we depend on the quality of the agent model. Focusing on finding alternative options, we can also consider plan repair proposals [19]. Plan repair fixes the current plan, retaining parts of it and changing others to avoid a problem. Some proposals search for repaired plans that should be as close as possible to the original plan [20] — a metric similar to the deviation that we used in our evaluation. Our proposal does not consider to change plans, we focus on *choosing* alternative options already available. Options are usually defined by the developer, but it is not necessarily so, they could be automatically generated.

Besides classical search and planning, other techniques are related to our proposal. Model checking also deals with foreseeing problems of an agent program [21, 22]. The approach is design-time: when a problem is detected, the program is fixed. Our approach is runtime: we consider that the agent has to solve the problem without relying on the developer for upgrading its program. We are not looking for errors in the agent program; on the contrary, we are exploiting it for runtime adaptation. The proposal of Ferrando et. al. [23] also consider runtime verification. As in our proposal, theirs detects a problem and selects another option. However, they focus on problems in the present and do not consider problems in the future. In some sense, we extend that proposal considering the simulation of the future. Another approach that considers the future is to assign a probability of success to the agent's options (as proposed, for instance, in [24]). The agent then selects the option with better chances to succeed, which implies avoiding future problems. The main difficulty with this approach is how to properly assign such probabilities. Learning is a possible approach for that [25, 26].

In the context of BDI agents and their programming languages [27–30], the notion of intentions is *future*-oriented. An intention represents a desired future state that the agent has committed to pursue [31, 32]. However, few programming platforms compute the future considering the agent's intentions, their achievements, and how they impact present decisions. Only the developer considers intention as future-oriented, not the agent itself. In the direction of endowing agents with future reasoning capabilities, we found only the research on Intention Progression [33–35]. They are concerned about scheduling several intentions so that they do not negatively interfere with each other's achievements. The problem they are addressing is caused by several intentions being pursued concurrently. Problems are foreseen by computing the future of these intentions, for instance, using Monte Carlo Tree Search [36]— this work gave us initial ideas for our implementation. We follow their approach by considering that the course of actions for an intention is derived from a plan library. However, our proposal concerns a single intention and identifies whether the policy for its achievement will succeed or not. They solve the problem by improving the schedule, we solve the problem by finding better options/actions.

Time is taken into account by more generic cognitive agent architectures. For example, the episodic memory of SOAR and its planning capabilities [37]. Regarding the future, they are more focused on finding a policy to achieve a goal, i.e., self-building a program. Our proposal departs from an existing agent program, usually provided by the developer. Currently, we are focused on evaluating whether this program will succeed or produce undesirable outcomes. It enables the agent to give up the goal prematurely or find alternatives.

Regarding the work on anticipatory thinking cited in the introduction and that served as inspiration for our research, our contribution relies on focusing on BDI agents and exploiting their particular features. As illustrated by the algorithms presented throughout the paper, we present a precise and operational proposal about how to anticipate problems in the future. As well as proposed in [6], we propose a metacognitive process, and, as proposed by [4], the nature of this process is similar to normal agent reasoning. Our agent places a clone of itself in a "matrix" and "sees" the outcomes (metacognition). The clone decides and executes actions in the same way as the original agent (similar process), of course with their actions being simulated. Simulation is a type of prediction tool proposed in [5]. Another difference in the matrix is that the clone is not able to foresee the future.

7 Conclusions and future work

The main conclusion of this paper is that the knowledge provided by the developer can help an agent to exploit its future and autonomously improve its performance. The improvement depends on the properties of this knowledge and the quality of the environment model given to the agent. Considering a general cognitive agent architecture, our contribution is the reduction of the search space promoted by that knowledge. Considering agent architectures based on plans, the contribution is to foresee problems in the application of these plans.

In future work, we also plan to address some delimitations initially considered for the scope of the research presented in this paper.

- The requirement of an environment model. This requirement is quite common in techniques that simulate the future. In some application however, it is difficult to obtain this

model [38]. One alternative is to learn an environment model, for example, by trial-and-error experimentation, as in reinforcement learning [39].

- The multi-agent case. How to face the case where more agents are changing the environment? Could we consider other agents as simply part of the environment? How can we exploit the K of other agents in the simulation of the future?
- The multi-goal case. As introduced by the Intention Progression Problem cited in Sect. 6, once the agent has several goals, the execution of their interleaving plans can produce a problem in the future. The detection of a future problem is quite straightforward in our proposal: simply simulate all plans. The difficulty is deciding for which goal alternative options are taken; including the option of dropping a goal.
- The past. In some cases, we are not able to find alternative options in the future because the wrong option was selected in the past. In other cases, better options in the past allows the agent to deviate less from its preferences. In these cases, the agent should be able to undo some actions to return to previous states. Of course, to undo actions may be difficult to implement. In some scenarios however, undoing actions is feasible; for instance, in our grid scenario, undoing actions can be seen as returning to some location.
- The causes. As commented in Sect. 4, when a problem is foreseen, our proposal looks for any other option without problems. However, if we could identify the *causes* of the problem [14], we might identify when the wrong decision was taken and change the course of actions at that point. This approach may have better results than the search strategies we investigate (solve_p, solve_m, and solve_f).
- Background process. As discussed in the end of Sect. 4, the proposed algorithm may reduce agent reactivity. One possible solution is to implement the future simulator as a concurrent process that potentially interferes in the main reasoning cycle of the agent.
- Future for the programmer. The current proposal uses the simulation of the future to improve the decisions of the agent. The agent developer does not handle time. Agent programming languages could be extended to allow the programmer to write plans like "if I will be wet in the future, then take an umbrela". This kind of plan is domain-dependent, and thus it is up to the developer to specify. As we have intentions as future oriented mental attitudes, we may have beliefs as well.

A Implementation

This appendix briefly describes the implementation of our proposal in Jason [15]. The implementation and the experiments are available at https://github.com/jomifred/future-bdi.

Regarding the environment model required by our proposal, a simulator should be provided. Several Jason applications, including the examples provided in the distribution, provide an environment simulator. In these cases, the environment model is easy to obtain.

Regarding the agent model. Since Jason agents do not decide their actions based only on the state of the environment, but consider all their mental state, the argument for the function π is the entire agent state (beliefs, plans, intentions,...). The environment state is indirectly considered because it is represented in the agents' beliefs. Plans play an important role in the agent model, they define the set K of Fig. 8. In the case of the grid scenario, the agent goal is to be at some location and the plans for this goal (represented by `pos(X,Y)`) are the following:

```
// plans for goal pos(X,Y)
//
@  [preference(0),cost(0.0)] +!pos(X,Y) : pos(X,Y). // nothing to do
@s [preference(D),cost(1.0)] +!pos(X,Y) : ok(s) & distance(s ,D) <- s;  !pos(X,Y).
@sw[preference(D),cost(1.4)] +!pos(X,Y) : ok(sw)& distance(sw,D) <- sw; !pos(X,Y).
@se[preference(D),cost(1.4)] +!pos(X,Y) : ok(se)& distance(se,D) <- se; !pos(X,Y).
@w [preference(D),cost(1.0)] +!pos(X,Y) : ok(w) & distance(w ,D) <- w;  !pos(X,Y).
@e [preference(D),cost(1.0)] +!pos(X,Y) : ok(e) & distance(e ,D) <- e;  !pos(X,Y).
@n [preference(D),cost(1.0)] +!pos(X,Y) : ok(n) & distance(n ,D) <- n;  !pos(X,Y).
@nw[preference(D),cost(1.4)] +!pos(X,Y) : ok(nw)& distance(nw,D) <- nw; !pos(X,Y).
@ne[preference(D),cost(1.4)] +!pos(X,Y) : ok(ne)& distance(ne,D) <- ne; !pos(X,Y).
@id[preference(D),cost(1.9)] +!pos(X,Y) :     distance(idle,D) <- idle; !pos(X,Y).

// checks if going to some direction is possible (free cell)
ok(D) :- next(D,X,Y) & free(X,Y).

next(s ,X  ,Y+1) :- pos(X,Y). // my next location if doing south
next(sw,X-1,Y+1) :- pos(X,Y).
next(se,X+1,Y+1) :- pos(X,Y).
next(w ,X-1,Y  ) :- pos(X,Y).
next(e ,X+1,Y  ) :- pos(X,Y).
next(n ,X  ,Y-1) :- pos(X,Y).
next(nw,X-1,Y-1) :- pos(X,Y).
next(ne,X+1,Y-1) :- pos(X,Y).
next(idle,X,Y)   :- pos(X,Y).

free(X,Y) :- X >= 0 & Y >= 0 & w_size(W,H) & X < W & Y < H
             & not obstacle(X,Y)
             & not agent(_,X,Y).
distance(Dir,Dist) :- next(Dir,X,Y) & destination(GX,GY) &
                      Dist = math.sqrt( (X-GX)**2 + (Y-GY)**2 ).
```

All initial plans (marked with `@`) are options for the goal `pos(X,Y)`. They are annotated with preference and cost. The preference annotation is used to order the options. The cost annotation is used to evaluated the chosen options.

The Jason agent reasoning cycle was customised so that whenever the agent has to select a plan for a goal, it creates a matrix (as introduced in Sect. 3) to foresee if the first option will produce a problem in the future. To build the matrix, we take the environment simulator and a clone of the agent. This clone has a customised architecture that redirects its perception and actions towards the simulator instead of the real environment. Although the environment is simulated in the matrix, the decisions of the agent are not simulated, its reasoning cycle is executed as usual in Jason.

The matrix runs until a problem is identified, a timeout is reached, or the goal is achieved. The notion of problem is: the environment returns to an already seen state, entering thus in a loop (the problem represented by p^l in Sect. 3).

In the case that a problem is identified, a failure event `-!pos(X,Y)` is produced for the goal and the developer can then decide how to handle it. In the experiments, we used the following code:

```
-!pos(X,Y)[error(future_issue(FI)),error_msg(M)]
    : pos(CX,CY) // my location
   <- .print("Future problem for goal pos(",X,",",Y,"): ",FI,", ",M);
      jason.future.plan_for(
          pos(X,Y),
          { @[cost(0), preference(0)]+!pos(X,Y) : pos(CX,CY) },
          Plan, solve_m);
      .add_plan(Plan);
      !pos(X,Y). // try the goal again
```

This plan can be read as follows: if there is failure in goal `pos(X,Y)` due to a problem `FI` in the future, use the internal action `jason.future.plan_for` to build a new plan for this goal using strategy solve_m. The internal action implements the search for alternative options and returns (in variable `Plan`) a Jason plan that avoids the future problem. This plan is added in the Plan Library as the preferred option for the goal. The goal is then retried.

In our implementation, the developer provides the environment and agent models, which are domain dependent. The matrix simulator, problem detection, and the search for alternative options are general and provided as a Jason customised agent architectures and internal actions.

Acknowledgements This work was produced at the research group headed by Professor Alessandro Ricci at the University of Bologna during the sabbatical of Professor Jomi F. Hübner. He expresses gratitude to the team for the inspiring conversations that took place at that time, which results include this paper. The sabbatical year was partially funded by Brazilian Agencies for Higher Education and Research: CNPq and CAPES, under the project PrInt CAPES-UFSC "Automation 4.0".

References

1. Ciortea, A., Mayer, S., Gandon, F., Boissier, O., Ricci, A., & Zimmermann, A. (2019). A decade in hindsight: the missing bridge between multi-agent systems and the world wide web. In: AAMAS 2019-18th International Conference on Autonomous Agents and Multiagent Systems, p. 5.
2. Calbimonte, J.-P., Ciortea, A., Kampik, T., Mayer, S., Payne, T. R., Tamma, V., & Zimmermann, A. (2023). Autonomy in the age of knowledge graphs: Vision and challenges. *Transactions on Graph Data and Knowledge (TGDK)*. https://doi.org/10.4230/tgdk.1.1.13
3. Vachtsevanou, D., Ciortea, A., Mayer, S., & Lemée, J. (2023). Signifiers as a first-class abstraction in hypermedia multi-agent systems. In: Proc. of the 2023 International Conference on Autonomous Agents and Multiagent Systems, pp. 1200–1208.
4. Jones, S.J., & Laird, J.E. (2023) A cognitive architecture theory of anticipatory thinking. AI Magazine (June), 155–164. https://doi.org/10.1002/aaai.12102.
5. Szpunar, K. K., Spreng, R. N., & Schacter, D. L. (2014). A taxonomy of prospection: Introducing an organizational framework for future-oriented cognition. *Proceedings of the National Academy of Sciences, 111*(52), 18414–18421. https://doi.org/10.1073/pnas.1417144111,https://arxiv.org/abs/www.pnas.org/doi/pdf/10.1073/pnas.1417144111
6. Amos-Binks, A., & Dannenhauer, D. (2019). Anticipatory thinking: A metacognitive capability. In: Proceedings of the Workshop on Cognitive Systems for Anticipatory Thinking. https://doi.org/10.48550/arXiv.1906.12249.
7. Amos-Binks, A., Dannenhauer, D., & Gilpin, L. H. (2023). The anticipatory paradigm. *AI Magazine, 44*(2), 133–143.
8. Georgeff, M.P., & Lansky, A.L. (1987) Reactive reasoning and planning. In: Proc. of the Sixth National Conference on Artificial Intelligence (AAAI 87), pp. 677–682.

9. Rao, A.S., & Georgeff, M.P. (1995). BDI agents: from theory to practice. In: Lesser, V. (ed.) Proceedings of the First International Conference on MultiAgent Systems (ICMAS'95), pp. 312–319. AAAI Pess, San Francisco, USA.
10. Silva, L.d., Meneguzzi, F., & Logan, B. (2020). BDI agent architectures: A survey. In: Bessiere, C. (ed.) Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20, pp. 4914–4921 https://doi.org/10.24963/ijcai.2020/684.
11. Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3rd ed.). New Jersey: Prentice Hall.
12. Ghallab, M., Nau, D., & Traverso, P. (2016). *Automated Planning and Acting*. Cambridge: Cambridge University Press.
13. Sgaier, S. K., Huang, V., & Charles, G. (2020). The case for causal AI. *Stanford Social Innovation Review, 18*(3), 50–55. https://doi.org/10.48558/KT81-SN73
14. Pearl, J., & Mackenzie, D. (2018). *The Book of Why: The New Science of Cause and Effect*. London: Basic Books.
15. Bordini, R.H., Hübner, J.F., & Wooldrige, M. (2007). Programming Multi-Agent Systems in AgentSpeak Using *Jason*. Wiley Series in Agent Technology. John Wiley & Sons, England https://doi.org/10.1002/9780470061848.
16. Garcia, F., & Rachelson, E. (2013). 1. Markov Decision Processes, pp. 1–38. John Wiley & Sons, Ltd, UK https://doi.org/10.1002/9781118557426.ch1.
17. Sardina, S., & Padgham, L. (2011). A BDI agent programming language with failure handling, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems, 23*(1), 18–70. https://doi.org/10.1007/s10458-010-9130-9
18. Meneguzzi, F., & De Silva, L. (2015). Planning in BDI agents: A survey of the integration of planning algorithms and agent reasoning. *The Knowledge Engineering Review, 30*(1), 1–44. https://doi.org/10.1017/S0269888913000337
19. Nebel, B., & Koehler, J. (1995). Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence, 76*(1), 427–454. https://doi.org/10.1016/0004-3702(94)00082-C
20. Babli, M., Sapena, Ó., & Onaindia, E. (2023). Plan commitment: Replanning versus plan repair. *Engineering Applications of Artificial Intelligence, 123*, 106275. https://doi.org/10.1016/j.engappai.2023.106275
21. Bordini, R.H., Visser, W., Fisher, M., Pardavila, C., & Wooldridge, M. (2003). Model checking multi-agent programs with CASP. In: Pre-proceedings of Third Workshop on Automated Verification of Critical Systems (AVoCS 2003), 2–3 April, Southampton, U.K.
22. Kamali, M., Dennis, L. A., McAree, O., Fisher, M., & Veres, S. M. (2017). Formal verification of autonomous vehicle platooning. *Science of Computer Programming, 148*, 88–106. https://doi.org/10.1016/j.scico.2017.05.006
23. Ferrando, A., & Cardoso, R. C. (2023). Failure handling in BDI plans via runtime enforcement. In: Gal, K., Nowé, A., Nalepa, G.J., Fairstein, R., Radulescu, R. (eds.) ECAI 2023 - 26th European Conference on Artificial Intelligence, September 30 - October 4, 2023, Kraków, Poland - Including 12th Conference on Prestigious Applications of Intelligent Systems (PAIS 2023). Frontiers in Artificial Intelligence and Applications, vol. 372, pp. 716–723. IOS Press, ??? https://doi.org/10.3233/FAIA230336.
24. Baitiche, H., Bouzenada, M., & Saïdouni, D.E. (2017). Towards a generic predictive-based plan selection approach for bdi agents. Procedia Computer Science 113, 41–48 https://doi.org/10.1016/j.procs.2017.08.283 . The 8th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2017) / The 7th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2017) / Affiliated Workshops.
25. Singh, D., Sardina, S., & Padgham, L. (2010). Extending BDI plan selection to incorporate learning from experience. *Robotics and Autonomous Systems, 58*(9), 1067–1075. https://doi.org/10.1016/j.robot.2010.05.008
26. Singh, D., Sardina, S., Padgham, L., & James, G. (2011). Integrating learning into a BDI agent for environments with changing dynamics, pp. 2525–2530. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-420.
27. Boissier, O., Bordini, R., Hübner, J. F., Ricci, A., & Santi, A. (2013). Multi-agent oriented programming with JaCaMo. *Science of Computer Programming, 78*(6), 747–761. https://doi.org/10.1016/j.scico.2011.10.004
28. Pokahr, A., Braubach, L., & Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In: Bordini, R.H., Dastani, M., Dix, J., El Fallah Seghrouchni, A. (eds.) Multi-Agent Programming: Languages, Platforms, and Applications. Multiagent Systems, Artificial Societies, and Simulated Organizations, pp. 149–174. Springer, Cham.

29. Hindriks, K. V. (2009). Programming rational agents in GOAL. In R. H. Bordini, M. Dastani, J. Dix, & A. E. F. Seghrouchni (Eds.), *Multi-Agent Programming* (pp. 119–157). Cham: Springer.
30. Dastani, M. (2008). 2APL: A practical agent programming language. *Autonomous Agent and Multi-Agent Systems, 16*, 241–248.
31. Bratman, M. E. (1987). *Intention, Plans, and Practical Reason*. Cambridge: Harvard University Press.
32. Cohen, P. R., & Levesque, H. J. (1987). Intention = choice + commitment. In: Proceedings of the 6th National Conference on Artificial Intelligence, pp. 410–415. Morgan Kaufmann, Cambridge.
33. Logan, B., Thangarajah, J., & Yorke-Smith, N. (2017). Progressing intention progression: A call for a goal-plan tree contest. In: Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS'17, pp. 768–772. https://doi.org/10.5555/3091125.3091234.
34. Thangarajah, J., Padgham, L., & Winikoff, M. (2003). Detecting & Avoiding Interference Between Goals in Intelligent Agents. In: IJCAI'03: Proceedings of the 18th International Joint Conference on Artificial Intelligence, pp. 721–726.
35. Dann, M., Thangarajah, J., & Li, M. (2023). Feedback-guided intention scheduling for bdi agents. In: Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, AAMAS'23, pp. 1173–1181.
36. Yao, Y., Logan, B., & Thangarajah, J. (2014). Sp-mcts-based intention scheduling for bdi agents. In: Proceedings of the Twenty-First European Conference on Artificial Intelligence. ECAI'14, pp. 1133–1134. IOS Press, NLD https://doi.org/10.5555/3006652.3006903
37. Laird, J. E. (2019). *The SOAR Cognitive Architecture*. Cambridge: The MIT Press.
38. Söderström, T., & Stoica, P. (1989). *System Identification*. Saddle River, New Jersey: Prentice Hall.
39. Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Cambridge: Bradford.