A first approach to AGI-based Robot Task Planning

Michele Thiella[1], Elisa Tosello[1] and Enrico Pagello[1,2]

[1] *Dept. of Information Engineering, University of Padova, Via Gradenigo 6/B, 35131 Padova, Italy*
[2]*IT+Robotics srl, Contrà Valmerlara 21, 36100 Vicenza, Italy*

Abstract

Current research in robot Task Planning aims to develop solvers which quickly adapt to new assignments and scenarios. To this aim, we extend an existing proto-Artificial General Intelligence system, namely OpenCog, and give it the ability to effectively solve manipulation tasks whose domains contain four actions: *pick*, *place*, *stack*, and *unstack*. To let OpenCog solve this class of problems, we exploit its modules as the foundation of a Knowledge Base that describes and stores domains, problems, and the interactions between them. Then, we equip the system with a Breadth-First Search algorithm that finds the sequence of actions that solve the assignments. To prove the goodness of our proposal, we include and analyze a manipulation task where a manipulator robot has to interact with a human user to assemble some industrial components. Obtained results show that our system is complete and generic in terms of the domain and problem under evaluation. Future work will improve the achieved computational time and performance.

Keywords

Robot Task Planning, Artificial General Intelligence, OpenCog, Robot manipulation

1. Introduction

Advances in Artificial Intelligence (AI) allow robots to cope with an increasing task-to-task variability. However, even the most advanced robots, which exploit Neural Networks (NNs) and AI, cannot learn new tasks easily. Their architecture still has to be designed carefully.

To design a general-purpose Task Planning (TP) system, we need to choose a declarative language for formalizing the domain. Then, we should select a suitable solver which supports that language. Many different factors affect this selection process. For instance, not all languages support all types of reasoning, and solvers are typically tied to particular languages. Domains may include many objects and their properties. Finally, a language can formalize a problem in many ways. For these reasons, the selection of language and solver needs careful consideration [1]. However, this examination tends, either voluntarily or involuntarily, to restrict the possible domains or to force their description using only the features supported by the solver. Our aim is to generalise the domains and problems to be solved, and obtain a system able to solve any manipulation problem described by the actions *pick*, *place*, *stack*, and *unstack*. The following features become essential: (i) interacting with human operators and learning from them; (ii) exploiting existing Narrow-AI systems to guarantee a generalized

8th Italian Workshop on Artificial Intelligence and Robotics (AIRO 2021)
michele.thiella@studenti.unipd.it (M. Thiella); elisa.tosello@unipd.it (E. Tosello); enrico.pagello@unipd.it (E. Pagello)
0000-0003-4692-4279 (E. Tosello); 0000-0002-8152-0707 (E. Pagello)

solving process and an easy adaptability to new assignments; (iii) storing data in a Knowledge Base (KB) to facilitate the sharing of lessons learnt.

In this context, we propose to exploit an existing proto-Artificial General Intelligence (AGI) system: Open Cognition (OpenCog) [2]. It includes a flexible Knowledge Representation (KR) embodied in a scalable knowledge store, a cognitive process scheduler, and a plug-in architecture for the interaction between cognitive, perceptual, and control algorithms. Such features let OpenCog adapt to new conditions, understand them, and create behaviors based on the information learned. To make the system able to solve TP problems, we exploited its modules as the foundation of a KB that describes and stores domains, problems, and the interactions between them. Then, we equipped the system of a Breadth-First-Search (BFS) algorithm that finds the sequence of actions that solve an assignment. Our contributions follow:

- the implementation of a BFS algorithm for TP that solves manipulation problems through the combination of four actions: *pick*, *place*, *stack*, and *unstack*;
- the implementation of a supra-system that equippes OpenCog with such an algorithm and all the features necessary for TP;
- a preliminary evaluation of our proposal while solving an assembly task in conjunction with a human user. Obtained results show that our system is complete and generic in terms of problem and domain definitions. Future developments will improve the achieved computational time and performance.

2. State of the art

AGI is the ability of an intelligent agent to understand or learn any intellectual task that a human being can [3]. It is a primary goal of several artificial intelligence research.

Among others, the OpenAI project [4] applies Deep NNs (DNNs) to translate natural language into code [5], connects text to images [6], and produce human-like text [7]. Although OpenAI achieved valuable results, the NNs approach is expensive and requires an extensive amount of data to achieve generality. Moreover, it suffers from the adversarial example problem: adding an imperceptibly small but carefully designed perturbation leads the model to make a wrong prediction [8, 9, 10]. Finally, NNs are black boxes: while they can approximate any function, studying their structure won't give any insight on the function being approximated [11].

For these reasons, we investigated another AGI-oriented approach: OpenCog. This open-source framework includes a comprehensive model of human-like general intelligence. It exploits an integrative approach in which multiple AI algorithms cooperate on a common representational substrate. Such algorithms include DNNs, Probabilistic Logic Theorem Proving, Evolutionary Learning, and Concept Blending. The system is largely scalable, given the amount and diversity of data it can contain. Furthermore, current research is improving its distributivity by developing a decentralized structure composed of many KR databases and their query/reasoning engines. Agents do not need to locally solve assigned tasks. They can exploit the available knowledge and eventually augment it with new experiences.

These features make OpenCog easily extensible to the resolution of TP problems. Indeed, our own experience on TP has shown that most current planners are still extremely dependent on the problem, the domain, and the language used to define them [12, 13, 14]. Examples include the Planning Domain Definition Language (PDDL)-based approaches [15]. We aim to overcome these limitations and give generality to the TP solvers.

3. Our proposal

To provide OpenCog [2] with TP capabilities, we added a *Search Algorithm* to its modules, and we implemented a *sense-plan-act* framework whose *sense* unit includes both visual perception and Natural Language Processing (NLP). The exploited OpenCog modules follow:

- *Atom Space*. It is a KR database composed of *Atoms*. *Atoms* are hypergraphs that enclose any type of information (i.e., data, procedures, etc.) [16]. *Atoms* refer to both Nodes (vertices) and Links (edges) of an hypergraph. For example, a *Concept Node* represents any physical or abstract concept. *Inheritance Nodes* specify both intentional (is-a) and extensional (is-an-instance-of) relationships. A *Predicate Node* names the predicate of a relation, where predicates are functions that have arguments, and produce a truth value as output. An *Evaluation Link* lets specify the truth value of a predicate on a set of arguments. Finally, a *Query Link* specifies a search pattern that can be grounded, solved, or satisfied. Thus, within an industrial environment, if we want to specify that a snap ring is a movable object positioned on the top of another snap ring, and that an agent is holding a bearing sleeve, we use the following *Atoms* code:

```
    # Object Definition
    (ConceptNode "SnapRing1")

    # Movable Object Definition
    (InheritanceLink
        (ConceptNode "SnapRing1")
        (ConceptNode "object")))

    # Stack State Definition
    (EvaluationLink
      (PredicateNode "on")
      (ListLink
        (ConceptNode "SnapRing1")
        (ConceptNode "SnapRing2")))

    # In Hand State Definition
    (EvaluationLink
      (PredicateNode "in-hand")
      (ConceptNode "BearingSleeve1"))
```

- *Relex2Logic* (R2L). R2L deduces the predicate-argument structure of natural language sentences. It first produces a logical representation corresponding to hypergraphs of *Atoms* that are associated with the words of these sentences. Then, it structures the obtained descriptions to express the sentence logic in the form of a hypergraph. The following example shows some English sentences stored in the form of hypergraphs:

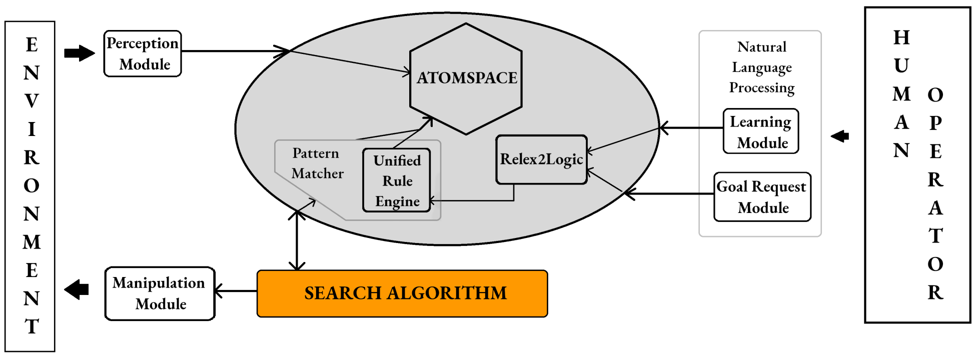


Figure 1: Our framework architecture.

```
# English sentence:
"The workbench is a fixed-object. SnapRing1 is on SnapRing2."

# Result:
(InheritanceLink
    (ConceptNode "Workbench")
    (ConceptNode "fixed-object"))

(EvaluationLink
  (PredicateNode "on")
  (ListLink
    (ConceptNode "SnapRing1")
    (ConceptNode "SnapRing2")))
```

- *Pattern Matcher* (PM). PM works with hypergraphs in the sense of fast extraction of specific data and query engine. In detail, it searches the *AtomSpace* for specific patterns of *Atoms*, i.e., hypergraphs with nodes and links of several types. If patterns have "holes" (i.e., variable locations), PM will "fill in the blanks" [17].

In this context, we store four actions in the *AtomSpace*: *pick*, *place*, *stack*, and *unstack*. Then, we add a *Search Algorithm* able to find the sequence of actions that bring a certain environment from its initial state to a desired one. The algorithm is based on BFS: each node is an *AtomSpace* that depicts the environment at a given time. Each edge out of a node represents one action executable at that state. Thus, each branch of the tree ends up describing a different sequence of actions, and the tree expansion takes place through the execution of actions using PM. We encode actions as *Query Links*, where a *Query Link* type *Atom* is defined as follows:

```
# Action Definition
(QueryLink
  (Variable_declarations)
  (Pattern_to_be_matched)  # preconditions
  (New_graph)              # effects
)
```

For example, the *pick* action is a *Query Link* that contains a pattern describing the prerequisites needed for an object to be picked up. Executing this rule will search for all atoms that satisfy these prerequisites within the *AtomSpace* at the current moment. The result is a list of *Atoms* corresponding to the objects that can actually be picked up. Finally, *Atoms* are pasted into the second pattern of the rule, resulting in a hypergraph each, describing the *pick* action performed.

We decided to encode actions as *Query Links* for the following reasons:

1. They allow us to specify a search pattern using *Variable Nodes*, which correspond to the "holes" introduced above. When we execute a *Query Link*, PM fills the "holes" assigning the *Variable Nodes* with the solution *Atoms* found by the search.
2. We can create new graphs as an effect of the *Query Link* execution. These graphs are new patterns containing the *Variable Nodes* replaced with the solution from the first point and inserted directly into the *Atom Space*.
3. *Query Links* are imperative: they actually perform the action, as opposed to declarative *Atoms* that simply describe it (which can be used in the future to reason about the action and eventually improve it with experience).

Based on these reasons, we can perform an action with or without parameters. In the former case, we can execute the *Query Link* in two steps following motivations 1 and 2. This mode allows to understand which objects are involved in the action. In the latter, we add a constraint to the *Query Link* search, limiting it to the object passed as a parameter.

Starting from the root node, the following steps compose our BFS algorithm:

1. If the node is root:
    - If the robot is holding an object in the root *Atomspace*, then two temporary nodes are created as copies of the root related to the *place* and *stack* action.
    - Otherwise, the same nodes are created but related to the *pick* and *unstack* actions.
2. If the node is not root:
    - Consider that two of the next four actions can be excluded looking at the previous one. Starting from either *place* or *stack*, only *pick* and *unstack* can be performed, and vice versa. Consequently, by looking at the action assigned to the edge incident on this node, two temporary nodes are created as copies of this node, associating the actions following the rule just explained.
3. In each temporary node, the assigned action is executed in *Without Parameters* mode.
4. From the resulting *Atomspace* of each one, the objects on which the action is applicable are extracted.
5. For each object found, a new copy node of the root is created. Then, the assigned action is applied to this node in *With Parameters* mode.
6. These final nodes become the children of the BFS tree and the edges are labelled with the respective performed action.
7. The algorithm starts again for each new node, following the breadth-first order of the queue where nodes are stored.

This makes the algorithm and the entire system generic with respect to domains, problems, and actions. To avoid loops of actions (i.e., actions that bring the environment into a state already encountered in the same branch), we check if each new node has already been encountered within its branch. In this case, the related sequence of actions created a loop and the expansion of that branch ends.

To add extra generality to our proposal and always allow to define the assignment as a Markov Decision Process (MDP), we include a learning phase, strictly connected to R2L. In detail, human users can help the system to reconstruct the initial state of the system, from which the search algorithm will look for a sequence of actions to reach the assigned goal. To this aim, they can input additional information via English sentences and R2L accurately store this data as hypergraph. In the same way, human users can request the system to bring the environment to a certain goal. Users do not need to describe the arrangement of the whole environment: they can focus on only the states of the objects involved in the task.

Figure 1 shows the architecture of the obtained framework. A perception module detects the objects populating the robot's surroundings. A human user inputs the desired goal and the information useful to achieve it (e.g., the position of objects not visible for the robot). The *Search Algorithm* exploits all available data to find a solution. If a solution exists, the robot will perform each action of the computed sequence until achieving the assignment.

4. Experiments and Results

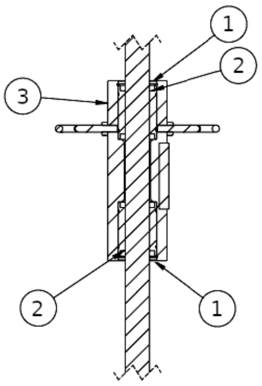
To prove the goodness of our proposal, we implemented a Robot Operating System (ROS) [18]-based setup where a human operator has to perform an assembly task. Given the task, a Franka Emika Panda manipulator robot[1] helps the human user by identifying the appropriate industrial components to be assembled, picking them in the correct order, and placing them on a pre-defined unloading position close to the human user. The help is reciprocal since the scene is not known beforehand: the human operator helps the robot by giving information about the position of non-visible objects. In this way, the robot can plan the sequence of manipulation actions optimal to achieve the task.

As shown in Figure 3, we mounted the robot on the same workbench where the human operates. Then, a free table acts as a collector and receives both the incoming pieces and the final assembly. Finally, a set of bins simulate a warehouse and group the incoming parts according to their type. We have a Microsoft Kinect One on the top of the robot end-effector, letting it detect the pieces on the table and inside the bins. To simplify both the perception and manipulation routines, industrial components are simulated as cubes with an AprilTag[2] fiducial marker attached on their tops. Each marker associates a semantic representation to each cube, i.e., the name of the industrial component it represents. To simplify collision avoidance, we assume that the human operator does not move within the robot workplace.

At the beginning, we assume that all useful components are in the scene, each inside its corresponding bin. The human helper informs the robot about the task to be performed. The robot scans its surroundings, detects the useful visible objects, and exploits the associated

[1]See https://www.franka.de/

[2]See https://april.eecs.umich.edu/software/apriltag



Parameters	Values
Objects	① {SnapRing1, SnapRing2}, ② {BearingSleeve1, BearingSleeve2}, ③ RecirculatingBallSleeve, workbench, PurpleBin, GreenBin, RedBin, BlueBin, YellowBin
Goal	SnapRing1 is on the workbench. BearingSleeve1 is on SnapRing1. RecirculatingBallSleeve is on BearingSleeve1. BearingSleeve2 is on RecirculatingBallSleeve. SnapRing2 is on BearingSleeve2.
Additional	SnapRing1 is on SnapRing2. BearingSleeve1 is on BearingSleeve2.
Solution	(unstack (ConceptNode ”SnapRing1”) (ConceptNode ”SnapRing2”)) (stack (ConceptNode ”SnapRing1”) (ConceptNode ”workbench”)) (unstack (ConceptNode ”BearingSleeve1”) (ConceptNode ”BearingSleeve2”)) (stack (ConceptNode ”BearingSleeve1”) (ConceptNode ”SnapRing1”)) (pickup (ConceptNode ”RecirculatingBallSleeve”)) (stack (ConceptNode ”RecirculatingBallSleeve”) (ConceptNode ”BearingSleeve1”)) (pickup (ConceptNode ”BearingSleeve2”)) (stack (ConceptNode ”BearingSleeve2”) (ConceptNode ”RecirculatingBallSleeve”)) (pickup (ConceptNode ”SnapRing2”)) (stack (ConceptNode ”SnapRing2”) (ConceptNode ”BearingSleeve2”))
Time	120 sec
N. Iters.	5000

Figure 2 & Table 1: (Figure 2) A component of an orthogonal torque reaction arm; (Table 1) The list of objects to be assembled, their final configuration, the additional information give by the human helper, the solution, the computational time, and the number of iterations needed to find the solution.

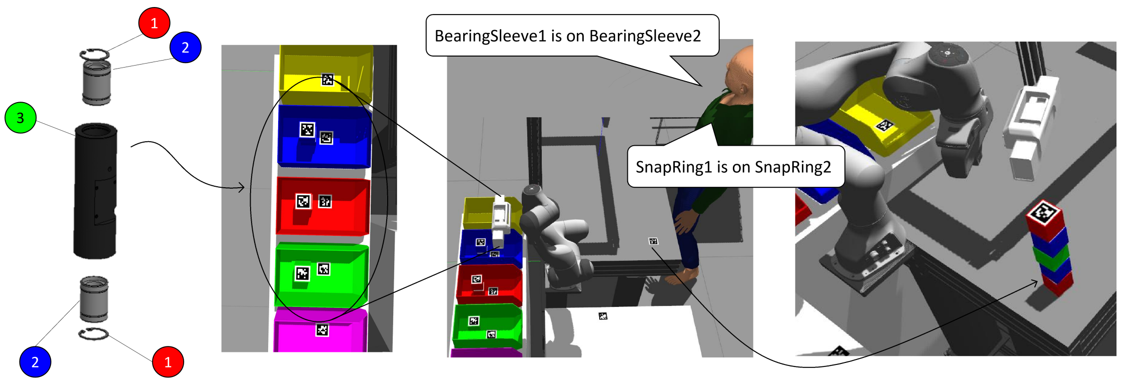

1
2
3
2
1
BearingSleeve1 is on BearingSleeve2
SnapRing1 is on SnapRing2

Figure 3: Full workflow. The product to be assembled is a piece of the orthogonal torque reaction arm of Figure 2. Each of its components is simulated by a cube. The snap rings are red cubes, the bearing sleeves are blue cubes, and the recirculating ball sleeve is a green cube. At the beginning, each cube is inside its corresponding colored bin. The human user gives the robot some additional info: ”BearingSleeve1 is on BearingSleeve2”, ”SnapRing1 is on SnapRing2”. This information, together with the retrieved visual data, lets the robot deduce the initial scene and compute an action plan that neatly brings the components from their initial location to the unloading station.

semantic representations to create a description of the environment as hypergraphs within the *AtomSpace*. Then, it interrogates the human operator to retrieve the missing information. Once known the entire environment configuration, the*Search Algorithm* looks for a solution by combining the set of preconditions and effects of the considered actions. The result is a set of actions that let the robot unload all components on the pre-defined unloading area in the correct order. If a solution exists, the robot executes the computed actions and gives the human user all pieces needed to compose the product.

We performed multiple tests with different assembled products and various quantities of pieces to be assembled. For the sake of brevity, we only report the results obtained when assembling the 5 items of Table I: two snap rings, a recirculating ball sleeve, and two bearing sleeves. They should form the component of Figure 2: a piece of an orthogonal torque reaction arm. Thus, the following goal configuration should be true: *"SnapRing1 is on the workbench. BearingSleeve1 is on SnapRing1. RecirculatingBallSleeve is on BearingSleeve1. BearingSleeve2 is on RecirculatingBallSleeve. SnapRing2 is on BearingSleeve2."* As shown in the table, as additional information, the human informs the robot that some parts are not visible because they are behind other items: *"SnapRing1 is on SnapRing2"*, *"BearingSleeve1 is on BearingSleeve2"*. Once the system has learned the configuration of its surroundings and the final goal, it process the information as hypergraphs within the *AtomSpace* until producing a sequence of *pick*, *place*, *stack*, and *unstack* actions that solves the human request. The solution follows (see Table I):

1. (unstack (ConceptNode "SnapRing1") (ConceptNode "SnapRing2"))
2. (stack (ConceptNode "SnapRing1") (ConceptNode "workbench"))
3. (unstack (ConceptNode "BearingSleeve1") (ConceptNode "BearingSleeve2"))
4. (stack (ConceptNode "BearingSleeve1") (ConceptNode "SnapRing1"))
5. (pickup (ConceptNode "RecirculatingBallSleeve"))
6. (stack (ConceptNode "RecirculatingBallSleeve") (ConceptNode "BearingSleeve1"))
7. (pickup (ConceptNode "BearingSleeve2"))
8. (stack (ConceptNode "BearingSleeve2") (ConceptNode "RecirculatingBallSleeve"))
9. (pickup (ConceptNode "SnapRing2"))
10. (stack (ConceptNode "SnapRing2") (ConceptNode "BearingSleeve2"))

Once the *Search Algorithm* has found the sequence of actions to achieve the assignment, the robot executes the computed actions, as shown in Figure 3. To find a solution, the system takes 120 seconds on a laptop with Intel(R) Core(TM) i5-9300H CPU, NVIDIA GeForce GTX 1650 Max-Q graphics card and 8GB of RAM. The operating system is installed on an external SSD connected with the USB 3.1 standard. The computational time decreases exponentially as the number of objects to be assembled decreases.

We compared our proposal with Fast-Forward (FF) [19] and SMTPlan [20]. The former takes 0.006 sec to find a solution, the latter takes 18.141 sec. This result may seem discouraging. Otherwise, we should account that both the algorithms take as input a problem and a domain formulated using PDDL. Such formulations should faithfully represent the state of the world in its entirety. In our case, instead, the *Search Algorithm* is complete, and the overall framework is independent of the formulation of initial and final configurations. Indeed, the human operator can inform the system about the observability of the problem and can help deduce the start configuration of the environment while exploiting the natural language.

5. Conclusions and Future Work

In this paper, we extended OpenCog to let the system solve TP problems. Without loss of generality, we focused on robot manipulation problems solvable through a combination of four actions: *pick*, *place*, *stack*, and *unstack*. To let OpenCog solve these problems, we exploited its modules as the foundation of a KB that describes and stores domains, problems, and the interactions between them. Then, we equipped the system of a BFS algorithm that finds the sequence of actions that solve the assignments. We included a preliminary evaluation of our proposal that asked a manipulator robot to interaction with a human operator to solve an assembly task. Obtained results prove that our system is complete and generic in terms of the considered problem and domain.

Future work includes the integration of additional actions to improve our system's generality. Moreover, we will focus on increasing its performance via temporal reasoning. Finally, we will test our proposal in dynamic environments populated by movable objects.

Acknowledgments

Part of this work was supported by MIUR (Italian Minister for Education), under the initiative *Departments of Excellence* (Law 232/2016), and by Fondazione Cariverona, under the project *Collaborazione Uomo-Robot per Assemblaggi Manuali Intelligenti* (CURAMI).

References

[1] Y. Jiang, S. Zhang, P. Khandelwal, P. Stone, An empirical comparison of pddl-based and asp-based task planners, CoRR abs/1804.08229 (2018). URL: http://arxiv.org/abs/1804.08229. arXiv:1804.08229.

[2] D. Hart, B. Goertzel, Opencog: A software framework for integrative artificial general intelligence., in: AGI, volume 171 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2008, pp. 468–472.

[3] B. Goertzel, C. Pennachin (Eds.), Artificial General Intelligence, Cognitive Technologies, Springer, 2007. URL: http://dblp.uni-trier.de/db/series/cogtech/354023733.html.

[4] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, P. Zhokhov, Openai baselines, https://github.com/openai/baselines, 2017.

[5] M. C. et al., Evaluating large language models trained on code, CoRR abs/2107.03374 (2021). URL: https://arxiv.org/abs/2107.03374. arXiv:2107.03374.

[6] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, I. Sutskever, Learning transferable visual models from natural language supervision, CoRR abs/2103.00020 (2021).

[7] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, D. Amodei, Language models are few-shot learners, CoRR abs/2005.14165 (2020).

[8] A. Gleave, M. Dennis, N. Kant, C. Wild, S. Levine, S. Russell, Adversarial policies: Attacking deep reinforcement learning, CoRR abs/1905.10615 (2019). URL: http://arxiv.org/abs/1905.10615. arXiv:1905.10615.

[9] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, A. Swami, The limitations of deep learning in adversarial settings, in: 2016 IEEE European Symposium on Security and Privacy (EuroS P), 2016, pp. 372–387. doi:10.1109/EuroSP.2016.36.

[10] I. Goodfellow, Attacking machine learning with adversarial examples, 2020. URL: https://openai.com/blog/adversarial-example-research/.

[11] V. Buhrmester, D. Münch, M. Arens, Analysis of explainers of black box deep neural networks for computer vision: A survey, CoRR abs/1911.12116 (2019). URL: http://arxiv.org/abs/1911.12116. arXiv:1911.12116.

[12] F. Ceola, E. Tosello, L. Tagliapietra, G. Nicola, S. Ghidoni, Robot task planning via deep reinforcement learning: a tabletop object sorting application, in: 2019 IEEE International Conference on Systems, Man and Cybernetics (SMC), 2019, pp. 486–492. doi:10.1109/SMC.2019.8914278.

[13] G. Nicola, L. Tagliapietra, E. Tosello, N. Navarin, S. Ghidoni, E. Menegatti, Robotic object sorting via deep reinforcement learning: a generalized approach, in: 2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN), 2020, pp. 1266–1273. doi:10.1109/RO-MAN47096.2020.9223484.

[14] L. Tagliapietra, E. Tosello, E. Menegatti, A planning domain definition language interpreter and knowledge base for efficient automated planning, in: The 16th International Conference on Intelligent Autonomous Systems (IAS-16)), 2021.

[15] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, D. Wilkins, PDDL—The Planning Domain Definition Language, 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212.

[16] B. Goertzel, Folding and unfolding on metagraphs, CoRR abs/2012.01759 (2020). URL: https://arxiv.org/abs/2012.01759. arXiv:2012.01759.

[17] F. Baader, T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998. doi:10.1017/CBO9781139172752.

[18] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, Ros: an open-source robot operating system, in: ICRA Workshop on Open Source Software, 2009.

[19] J. Hoffmann, Ff: The fast-forward planning system, AI Magazine 22 (2001) 57–62.

[20] M. Cashmore, M. Fox, D. Long, D. Magazzeni, A compilation of the full pddl+ language into smt, Proceedings of the International Conference on Automated Planning and Scheduling 26 (2016) 79–87. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/13755.