MonteCarlo Tree Search with Goal-Based Heuristic

Luca Sabatucci
[1]CNR - ICAR
Via Ugo La Malfa 153, Palermo, Italy
luca.sabatucci@icar.cnr.it

Abstract

The use of a domain-driven symbolic planner may provide interesting performances, even with the most challenging planning domain. However, sometimes a domain utility-function to be maximized does not exist: there are cases in which creating such a function is difficult and error-prone. This paper investigates an alternative approach to afford deterministic planning when no utility-functions are available. In cases like these, classical planning may provide bad performances. The use of a MonteCarlo approach, in conjunction with a goal-based heuristic, has given promising results.

Introduction

Modern IoT applications and complex autonomous systems more and more exploit search algorithms. For instance, some categories of autonomous systems decide at run-time how to adapt to unexpected events by running symbolic planners (Sabatucci and Cossentino 2015). This behaviour is typically related to service composition (Venero et al. 2020) and program synthesis, i.e. deriving low-level programs from high-level specifications, is another example of research argument closely related to planning (Bodík and Jobstmann 2013).

Recent planning competitions (Cenamor and Pozanco 2019; Vallati et al. 2015) highlighted many categories of planning approach are available. Notable examples are best-first planning (descending from A*), regression planning, planning as model checking and planning based on probabilistic models. Particular attention, in this paper, is given to MonteCarlo tree search approaches (Browne et al. 2012) that recently provided impressive results in large deterministic domains. The key idea is to use a best-first search guided by random simulations to estimate states' value.

Now, one urgent research question is how to implement a domain-independent planner that can scale up concerning domain complexity and domain size (often, size is taken as a measure of the number of objects, the number of relations or even the number of characters in a problem description). Scalability is a critical point, as underlined by the research community (Long and Fox 2003; Vallati et al. 2015). In the last years, the performances of modern planners are increasing very quickly (Richter and Westphal 2010). For instance, planning as model checking allowed of dealing with problems with millions of states in timely fashion (Cimatti et al. 2003).

A successful method for solving such problems is the use of accurate domain-specific utility-function, which gives for every state a lower bound on the cost of reaching a solution state. For being useful, such a function must be efficiently computable. If planners driven by utility-functions perform very well, when domain knowledge is not available, or the function is not accurate, then performances may quickly decay. The domain complexity hampers the scalability to large-scale problems. This finding is visible, for instance, in the experimental section of (Cimatti et al. 2003). Also, MonteCarlo approaches (that demonstrated their power in domains such as game/decision theory (Ginsberg 2001)) are less effective in evaluating random walks without exploiting domain knowledge. In symbolic domains, when raising the branching factor, the probability of ending up in a goal state is very low (Nakhost and Müller 2009).

Context. We focus on the problem of planning in deterministic domains when goals are provided via some predicate, or temporal logic and no domain utility-function is available.

Here, the main problem is the lack of a function for measuring (and therefore maximizing) the utility of the explored states. This condition often makes both classical planning and MonteCarlo planning incapable of scaling up with the size of the problem, unable to manage the explosion of states. Several approaches exist to face this problem, making the planner domain independent. A common approach is to use state abstraction for reducing the number of states to be explored. Another approach is planning as heuristic search (Bonet and Geffner 2001). In consists in transforming planning into a problem of heuristic search by automatically extracting heuristics from planning domain encodings. This approach can be combined by considering a goal as a combination of several subgoals to be considered in isolation (Scala et al. 2020).

Contribution. To the best of our knowledge, the planning as heuristics and the subgoaling techniques are rarely used in combination with monte-carlo search (Francès et al. 2017).

We propose to face planning problems by combining a classical UCT (upper confidence bound) approach with blind exploration of the state space to gain information on its structure. The exploration is conducted through a heuristic function (namely R2S) that is automatically derived from the specifications of goals. R2S allows assigning a quality-value to all the states of tree.

The result of this combination is the R2S MonteCarlo tree search algorithm.

Evaluation. Finally, we set up an experimental evaluation to test and compare performances of tree search approaches in hard planning problem. The planning domain has been extracted from an IoT domain in which a branching factor of 40 makes the state explosion unmanageable for classical planning.

Structure of the paper: Section 2 briefly introduces the R2S heuristic and reports an algorithm to automatically deriving the value of a state, given a goal. Section 3 is a state of the art section about MonteCarlo tree search. Section 4 is the core of the paper, illustrating details of the proposed approach. The algorithm is compared with other approaches in Section 5. Finally, some conclusions are drawn in Section 6.

The R2S Heuristic

The Resistance to Goal Satisfaction (R2S) adopts a correspondence with the electrical domain to provide an intuition of 'distance in the space of states' (Cossentino M., Sabatucci L. and Lopes S. 2018).

The electrical metaphor mainly involves resistors and series and parallels of resistors. The idea is that, given a conjunction of n propositional variables, $\phi = p_1 \wedge p_2 \wedge ...p_n$, each single variable value (eg: $p_i == false$) positively or negatively contributes to the overall satisfaction of the formula.

Consequently, the conjunction of predicates is similar to a series of resistors. Accepting this, a disjunction of predicates corresponds to a parallel of resistors.

To simplify computations, all resistors range from a minimum value R_{min} to a maximum value R_{max}. Actually, we choose a sufficiently great R_{max}, with the property that:

$$R_{min} = \frac{1}{R_{max}} \tag{1}$$

Given a state S, a single predicate p is associated to R_{min} or R_{max} according the basic principle:

$$R_p = \begin{cases} R_{min}, & \text{if } S \models p, \\ R_{max}, & \text{otherwise} \end{cases} \tag{2}$$

The logical NOT swaps minimum and maximum values:

$$R_{\neg\Phi} = \frac{1}{R_\Phi} \tag{3}$$

Therefore, in case of a negated predicate:

$$R_{\neg p} = \begin{cases} R_{max}, & \text{if } S \models p, \\ R_{min}, & \text{otherwise} \end{cases} \tag{4}$$

The resistance of a logical AND is computed as follows:

$$\text{if } \Phi = \bigwedge \phi_i \Rightarrow R_\Phi = \sum R_{\phi_i} \tag{5}$$

Finally, the resistance of a logical OR is computed as follows:

$$\text{if } \Phi = \bigvee \phi_i \Rightarrow \frac{1}{R_\Phi} = \sum \frac{1}{R_{\phi_i}} \tag{6}$$

Example: given a goal $G = a \wedge (b \vee \neg c)$, this allows deriving the R2S formula as

$$R2S_G = R_a + (\frac{1}{\frac{1}{R_b} + \frac{1}{R_{\neg c}}}) \tag{7}$$

Let us suppose the current state is the most far from the solution: $S_0 = \{c\}$,

$$R2S_G(S_0) = R_{max} + (\frac{1}{\frac{1}{R_{max}} + \frac{1}{R_{max}}}) = \frac{3}{2}R_{max} \tag{8}$$

Now, making a step forward the solution: $S_1 = \{b, c\}$,

$$R2S_G(S_1) = R_{max} + (\frac{1}{\frac{1}{R_{min}} + \frac{1}{R_{max}}}) \approx R_{max} \tag{9}$$

Finally, reaching the goal state $S_2 = \{a, b, c\}$

$$R2S_G(S_2) = R_{min} + (\frac{1}{\frac{1}{R_{min}} + \frac{1}{R_{max}}}) \approx 2R_{min} \tag{10}$$

The metric proved to be robust to logical transformations (i.e. De Morgan's Law).

Surely, this metric has several limitations. As an instance, it does not consider predicate interpretation. Indeed, not all predicates in a formula have the same importance. However, in a planning problem, R2S may be used when no other domain heuristics exist.

We already adopted this metric in classical planning by adopting a best-first exploration approach in the domain of service composition (Sabatucci, Cossentino, and Lopes 2019). This planner performed well, allowing a fast convergence towards the solutions. However, significant problems of scalability occurred when moved towards other applications domains with high branching factors.

In particular, this approach proved its limits in some specific situations. One of these is illustrated below.

Concatenating actions when the branching factor is high. Let us suppose that the given goal is to reach a state of the world in which the predicate b is true. The initial state $W_0 \models a$, and a couple of actions are applicable to produce state neighbours. However, we discover that, for producing a state where b is true, it is necessary to use a combination of 4 actions:

Algorithm 1: Function for calculating the R2S of propositional goals.

```
Function R2S ($State$,$\phi$):
    case $\phi = p$ is a predicate do
        if $State \models p$ then
            return $R_{min}$
        else
            return $1/R_{max}$
        end
    case $\phi = \alpha \wedge \beta$ do
        return $R2S(\alpha) + R2S(\beta)$
    case $\phi = \alpha \vee \beta$ do
        return $1/(1/R2S(\alpha) + 1/R2S(\beta))$
    case $\phi = \neg\alpha$ do
        return $1/R2S(\alpha)$
    end
```

$$(W_0) \xrightarrow{A_1} (W_1) \xrightarrow{A_2} (W_2) \xrightarrow{A_3} (W_3) \xrightarrow{A_4} (W_4)$$

where $W_4 \models b$. Now, when the planner is exploring W_0, evaluate all its neighbours, and all of them have the same R2S. Consequently, the next state to explore is selected randomly. Even switching to a breadth-first strategy does not perform well: in the worst case, the entire the tree (of depth 4) will be explored.

With a branching factor of 2, the number of states is 30. However, when the branching factor grows, there are no grants that W_4 is discovered in a timely fashion. Indeed, with a branching factor of 10, in the worst case, it could be necessary to explore $\sum_{i=1}^{4} 10^i \approx 11,000$ states before discovering W_4. Furthermore, increasing the branching factor to 40 the number of states is about 2.6 million putting in crisis almost all the state of the art planners.

Next sections illustrate how, in order to face hard-combinatorial problems, we exploited a MonteCarlo algorithm.

Montecarlo Search

Monte Carlo methods have their roots in statistical physics where they have been used to obtain approximations to intractable integrals. So far, they have been used in a wide array of domains.

The power of Monte Carlo with games and decision theory has been already demonstrated by achieving world champion level play in games such as Bridge (Ginsberg 2001) and Scrabble (Sheppard 2002).

Supported by these impressive results, recently, Monte-Carlo techniques have been employed in planning (Nakhost and Müller 2009; Silver and Veness 2010). The main principle is that, given a state, the next action may be selected by using random simulations. The simulation provides an approximate value of an action, to be used efficiently to adjust the policy towards a best-first strategy (Nakhost and Müller 2009).

The Simulation phase is critical in the algorithm; it generates random moves deeply in the tree, until a terminal position. Then it exploits some heuristics to evaluate these terminal states. This strategy is particularly suitable for games, but it may be difficult to use in other domains. For instance, in a symbolic planning problem where goals are predicates, Pellier et al. in (Pellier, Bouzy, and Métivier 2010) argued that random explorations might be inefficient. Indeed, the probability that a given simulation finds a goal state is low.

In literature, there exist many enhancements to Monte-Carlo planning that could be applied to any domain without prior knowledge about it. However, until now, these tricks typically offer useful improvements when used in a particular type of domain (Browne et al. 2012). Monte Carlo methods have their roots in statistical physics where they have been used to obtain approximations to intractable integrals. So far, they have been used in a wide array of domains.

MonteCarlo Tree Search with R2S

The ideas behind the R2SMonteCarlo tree search algorithm are: 1) during the simulation phase, stopping with an incomplete outcome when there is any change in the R2S value; 2) during the back-propagation phase, generating a reward for nodes with negative variations of R2S.

Selection: every iteration starts with node selection. This phase aims to randomly select one node among the most urgent ones to be expanded. In order to model this urgency, a $Tokens$ list contains references to some 'interesting' nodes of the Tree. Aside the root node, that is always in the list, a node is considered attractive if it offers an advancement in terms of full goal satisfaction (i.e. when $R2S(root) - R2S(node) > 0$).

The algorithm updates the $Tokens$ list at the beginning of the algorithm and after every simulation. The updating policy has been studied to balance the probability to select either the root node or a promising node.

Expansion: this phase does not present any novelties. One child node may be added to the tree according to the available actions. A node is expandable if it is not an exit node (i.e. the goal is not yet fully satisfied) and it has unvisited children.

Simulation: a simulation runs, starting from the selected node, and it stops when a new node is discovered that offers a decrement of R2S (it is marked as delta-node). Differently by traditional strategies, paths towards a delta-node are not discarded after a simulation. This phase is the essential novelty of the algorithm.

BackPropagation: as usual, the simulation result is propagated back to the root node for updating statistics of nodes and rewarding paths that conducted to delta-node.

Indeed, after a simulation ended up with a delta node, every node of the path is examined. All delta nodes in the path receive a reward, consisting in adding node's references to the $Tokens$ list.

Details about the Algorithm

Algorithm 2 shows an outline of the steps of the proposed R2S MonteCarlo Search. For the sake of simplicity, $Tokens$ is a global data, accessible from all the functions. It may be implemented as a list of references to tree nodes.

Algorithm 2: MonteCarloR2S

```
   Data: $Tokens$
1
2  Function Main(root,G,A):
3      while $CheckTerminationCond()$ do
4          $v_0 = TreePolicy(root)$
5          $v_l = Simulation(v_0, G, A)$
6          $BackPropagation(v_l, \Delta(v_l))$
7      end
8      return $ExtractSolutions()$
9
10 Function TreePolicy(root):
11     if $Tokens$ is empty then
12         $Tokens \leftarrow update\_tokens(root)$
13     end
14     $Tokens = shuffle(Tokens)$
15     $v_0 = Tokens.head$
16     $Tokens = Tokens.tail$
17     return $v_0$
18
19 Function Simulation(v,G,A):
20     $\delta = 0$
21     $v_i = v$
22     while $\delta == 0 \land v_i$ is not exit do
23         $v_i = ExpandOrChild(v_i, A, G)$
24         $\delta = R2S(v, G) - R2S(v_i, G)$
25     end
26     $\delta(v_i) = (\delta > 0)$
27     return $v_i$
28
29 Function ExpandOrChild(v,A,G):
30     $U = untried(v, A)$
31     if $U$ is not empty then
32         $a = U.head$
33         $v' = expand(State, a)$
34         if $v' \models G$ then
35             set $v'$ as exit node
36         end
37     else
38         $v' = random\_child(v)$
39     end
40     return $v'$
41
42 Function BackPropagation(v,Δ):
43     $visit(v) += 1$
44     if $\Delta > 0$ then
           /* reward                        */
45         if $\delta(State) > 0$ then
46             $Tokens \leftarrow update\_tokens(v)$
47         end
48     end
49     if $v$ is not root then
50         $BackPropagation(parent(v), \Delta)$
51     end
```

The main function runs over a number of iterations according to a termination condition. Typically the termination condition poses a temporal bound for stopping the search. Other criteria may be either a max number of iterations or a max number of visited nodes of the tree.

The main iteration consists of three standard steps (lines 4-6): TreePolicy for selecting the target node, Simulation and BackPropagation. During the iterations, the tree is explored, and nodes are marked with information like:

- $exit : Boolean$ when true, it indicates the node is an exit node of the tree, i.e. the path $\pi(root, exit)$ is a possible solution;
- $\delta : Boolean$ when true, it indicates the node is a delta node i.e. an 'interesting' node such that: $\forall p \in \pi(root, node), p \neq node : R2S(node, G) < R2S(p, G)$
- $visit : Integer$ is a counter for the number of times a node has been visited;

At the end (line 8), the algorithm simply looks at exit nodes and returns all the paths $\pi(root, exit)$.

The *TreePolicy* function (lines 10-17) selects the next node to be expanded. This consists of randomly picking one node from the $Tokens$ list. At the first iteration, when the list is empty, it is filled with references to the root node. In the beginning, most of the times, this function will return the root node. However, by increasing the tree size, the probability to explore other nodes increases (see Figure 1).

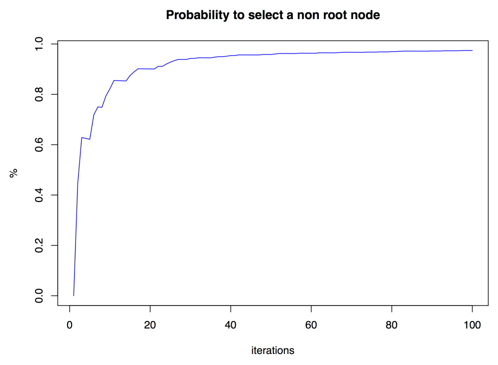

Figure 1: The probability to select a non root node converge after a number of iteration equals to the branching factor.

The *Simulation* function (lines 19-27) is invoked with the selected node v as parameter. This function performs a random walk, of variable depth, starting from v. At each step, it updates v_i with one of its child, obtained by selecting a random action (line 23). The simulation terminates when reaching a terminal node of the tree (a node that cannot be further expanded, for instance, an exit goal), or when the new v_i improves or decreases the goal satisfaction, respect to the initial node v (line 24). The last step before returning the node is marking the delta-value of the node to true when it improves goal satisfaction (line 26).

The expansion strategy is in charge of the *ExpandOrChild* function (lines 29-40) that is a standard policy, reported for completeness. Given a node v it looks at untried actions for that node. If all the actions have already been tried, then it randomly selects one of the children (line 38). Otherwise, it selects an action a (line 32) and applies a to v thus generating a new node v' (line 33). It also checks whether the new node is an exit node (lines 34-36).

The *BackPropagation* function, basically, updates the $Tokens$ list when the parameters Δ is greater than zero, i.e. when the last simulation improved goal satisfaction. This is a recursive function: the parameter v is initially set to that terminal node, and successively, it assumes the values of all the parent nodes until the root. Every time, a node is marked as 'δ', the algorithm adds node references to the $Tokens$ list according to a reward strategy.

Different reward policies may be tuned according to different strategies: 1) a reference for each child, 2) proportional to the R2S increment, 3) inversely proportional to the depth in the tree. In order to avoid the problem of local minima and plateau, the root node always has the highest probability to be selected than any other node.

Example and Some Property

Here we illustrate an example of tree construction, showing the contribution of the three main steps.

Intending to illustrate the approach, we suppose a tree with branching factor of 3, and a reward policy of type I: 1) root node is a delta node by default and 2) rewarding consist of one reference for each node's child. Therefore, in this example, the reward is constant: 3 references.

At the very first iteration (see row n.1 in Figure 2), the $Tokens$ list only contains three references to the root node. Clearly, the only possible selectable node is the root node. In this case, the Simulation, starting from $v0$ creates a path $v0 - v8$ where the last node, $v8$, has a better value of goal satisfaction. Given that $\Delta > 0$, the path is not removed. On the contrary, the leaf node is rewarded by introducing three references to $v8$ in the $Tokens$ list. Back propagating to the root (that is a delta node for construction), also $v0$ is rewarded. At the end of the iteration, the $Tokens$ list contains five references of $v0$ (the remaining 2 plus the new 3) and three references of $v8$.

The second iteration is an instance of reduction of goal satisfaction. The root node (with higher probability) is selected again. However, the Simulation produces a path with a $\Delta < 0$. For this reason, new nodes of the path may be ignored. There is no reward, and therefore the $Tokens$ list remains the same, except than $v0$ has one reference fewer.

The third iteration selects $v8$ to be further expanded. A new Simulation generates a path $v8 - v12$ where the last node is again a delta node ($\Delta > 0$). This time BackPropagation rewards $v12$ as well as $v8$ and $v0$ (all the delta nodes until the root). At the end of the iteration, the $Tokens$ list contains seven references of $v0$, five to $v8$ (the remaining 2 plus the new 3) and three references to the new $v12$ node.

In the last shown iteration, the root node is again selected, and this time the Simulation produces a new path $v0 - v15$ in which the leaf node is delta, and therefore path nodes are maintained. BackPropagation rewards node $v15$ and the root node.

The example is useful to illustrate some property of the R2S MonteCarlo Search.

The first interesting property is the alternating node selection. The algorithm preserves a chance in choosing both the root node and any delta nodes. This is important for two reasons: when there are not delta nodes, the algorithm becomes similar to a general MCTS approach; when there are delta nodes, these vary the probability distribution, creating dynamism in tree exploration and potentially improving search performances. Next section presents an experimental evaluation that supports this claim.

A significant difference with most of MonteCarlo approaches is that Simulation is not only used to estimate a node value, but it may generate useful nodes to explore further. In other words, whereas traditionally the tree is built incrementally by evaluating node's payout via simulations, the R2S approach, giving importance to some regions than

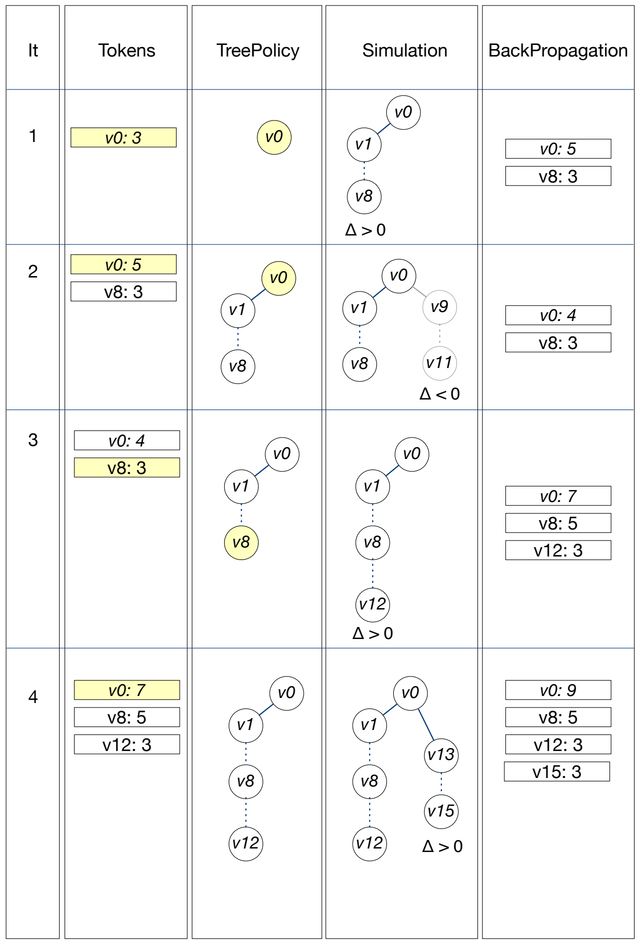

Figure 2: Example of execution of the R2S MonteCarlo Tree Search. The first four iterations are shown in rows. The first column shows the iteration number. The second column shows the $Tokens$ list at the beginning of the iteration; blocks show the number of tokens for each node. The third column reports the tree at the beginning of the iteration. The Yellow colour is used to highlight the selected token and tree node. The Fourth column shows how the Simulation works. Dotted lines indicate there are omitted nodes, not shown for compactness. The last node is annotated with positive or negative Δ. Nodes in Grey colour will not be added to the tree. Finally, the last column shows how the $Tokens$ list is updated consequence of the BackPropagation.

others, it allows a very asymmetric tree growth that better suits for trees with high depth. Figure 3 provides an instance of asymmetric exploration of a tree with a branching factor of 40.

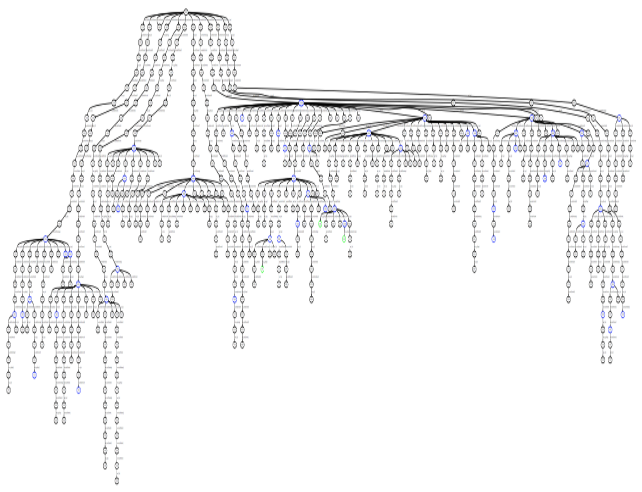

Figure 3: Example of Asymmetric exploration of the tree. After 208 iterations and 700 visited nodes, the R2S Monte-Carlo search returned with 3 solutions.

Another property concerns nodes that initially seems promising but that lead to dead points. When a new delta node is discovered, the reward policy is studied to grant the possibility to expand all the node's child. Indeed, the number of references is equal, at least, to the node branching factor. However, selecting a node in which the simulation phase leads to a non-delta node, it decreases the probability to select that node again. The probability goes down if all its children have been already explored without results. However, whatever node, the probability is never zero, because there is always a chance that a parent root is selected. This is similar to the pruning of unpromising paths, with a chance to recover them in future.

Evaluation

We used the following an application domain from Smart Ships for conducting some preliminary experiments.

The Shipboard Power System (SPS) is the component of a ship that is responsible for granting energy to navigation, communication, and many other operational systems. It is consists of various electric and electronic equipment, such as generators, cables, switchboards, circuit breakers, fuses, buses, and many kinds of loads. In modern ships, this component is more and more similar to an IoT application.

The Reconfiguration of a SPS (Sabatucci L., Cossentino M., De Simone G. and Lopes S. 2018) is a critical operation necessary to grant the continuity of services in unexpected situations, such as in the case of severe or major faults. A software manager drives the reconfiguration procedure. It controls generators, switchers and loads in order to configure the electrical schema according to a mission to accomplish. In this way, the manager is able: to isolate faults, to

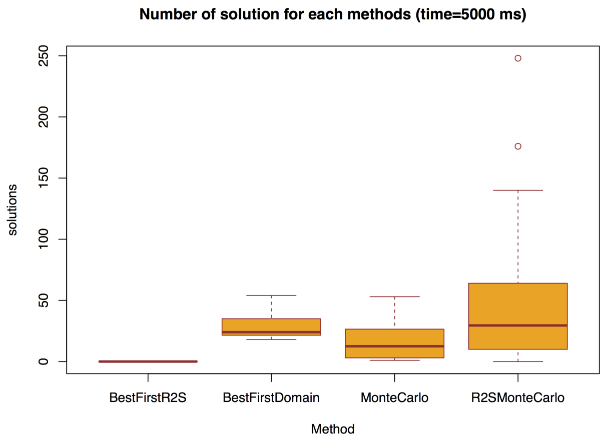

Figure 4: Box-plot of the outcome of four different algorithms in 20 runs (a different goal characterizes each run, automatically generated). Results are obtained, giving each algorithm the same temporal cut-off of 5000ms, time in which they must found the highest number of solutions.

restore/transfer power to vital loads, but also, more generally, to optimise the management of electrical and electronic equipment by improving efficiency.

In some previous works (Sabatucci, Cossentino, and Lopes 2019; Sabatucci L., Cossentino M., De Simone G. and Lopes S. 2018), we adopted a symbolic approach to implementing the manager as a planner where ship missions are encoded as goals. The plan represents the actions to operate in order to switch from the current configuration to the desired one.

The problem is combinatorial with a branching factor equal to the number of switcher in the circuit (in a small/middle size ship they are in the order of 30-50). Moreover, in order to accomplish some goals, sometimes it is necessary to operate a sequence of four/five actions (as the example described in Section 2). For these reasons, a classical planning approach degrades by increasing the scale.

In the following, we adopted the R2S MonteCarlo Search with a mid-size problem of branching factor 40. We selected the most 'difficult' initial configuration, i.e. all is switched off. We also built a random mission generator, i.e. a function that create random goals.

The experiment consisted of generating 20 different problem statements (initial state, actions and goals). For each of them, a script executes four algorithms, setting a temporal cutoff of 5000 ms. In this time, they must find the highest number of solutions. The algorithms in this comparison are: 1) a best-first planner using the R2S metric for the next node, 2) a best-first planner customized with a utility function specific for the SPS reconfiguration domain, 3) a MonteCarlo tree search with upper confidence bound, and 4) the R2S MonteCarlo tree search.

The algorithm is implemented in the Scala language. We used a 2.7 GHz Intel Core i5 and 16 Gb RAM to run the experiment. Figure 4 graphically reports the results of the experimental phase. Each box plot describes the performances measured for the four methods.

At a first analysis, it is quite clear that if a domain heuristic is available, a best-first planner can find solutions in a timely fashion: it represents the best approach dominating all the other three ones.

However, when the domain heuristic is not available (or it contains errors), the performances of the same planner decades quickly. Even by using the R2S metrics, the planner did not discover any solution. Conversely, the MonteCarlo method did provide much more solutions, even if the planner with domain heuristics dominates its best performance.

Conversely, the R2S MonteCarlo approach, despite the greater dispersion (the interquartile range) is characterized by a median value that is close to the optimal result and an upper quartile that is even better, respect to the best result of the best-first planner with domain heuristics.

Conclusions

We have described a new planning algorithm based on a combination of MonteCarlo tree search and a heuristic for measuring the distance of a state to the desired goal. The algorithm is based on a best-first search for exploring the space state. When evaluating the next node, a simulation phase tries to estimate the relative distance towards the goal by using the R2S metric. This function is automatically generated from goal specification and allows estimating this distance. To evaluate the algorithm, we used an experimental setting based on a combinatorial problem from the Smart Ship domain. Results proved the algorithm performances are close to those of an informed search (which provides the upper bound).

This is a preliminary work; a deeper experimental analysis is required to understand performance characteristics. We are considering to adopt a wider-range of comparative benchmark experiments to gain a better representation of how well the algorithm performs, specially against other domain-independent heuristic approaches.

Furthermore, as future work, we are studying how to exploit the algorithm for reporting partial solutions when full solutions do not exist or are not discovered in the time cutoff. This may be very useful in critical scenarios (emergency management, safety-critical systems) in which the partial satisfaction of a goal is better than no actions.

References

Bodík, R., and Jobstmann, B. 2013. Algorithmic program synthesis: introduction.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence. 2001 Jun; 129 (1-2): 5-33.*

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Cenamor, I., and Pozanco, A. 2019. Insights from the 2018 ipc benchmarks. In *Proc. of the Workshop of the IPC WIPC*.

Cimatti, A.; Pistore, M.; Roveri, M.; and Traverso, P. 2003. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence* 147(1-2):35–84.

Cossentino M., Sabatucci L. and Lopes S. 2018. Partial and full goal satisfaction in the musa middleware. In *Multi-Agent Systems - Prof. of the European Conference on Multi-Agent Systems (EUMAS) 2018*, volume 11450 of *Lecture Notes in Computer Science book series (LNCS)*, 15–29. Springer.

Francès, G.; Ramírez Jávega, M.; Lipovetzky, N.; and Geffner, H. 2017. Purely declarative action descriptions are overrated: Classical planning with simulators. In *IJCAI 2017. Twenty-Sixth International Joint Conference on Artificial Intelligence; 2017 Aug 19-25; Melbourne, Australia.[California]: IJCAI; 2017. p. 4294-301.* International Joint Conferences on Artificial Intelligence Organization (IJCAI).

Ginsberg, M. L. 2001. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research* 14:303–358.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

Nakhost, H., and Müller, M. 2009. Monte-carlo exploration for deterministic planning. In *Twenty-First International Joint Conference on Artificial Intelligence*. Citeseer.

Pellier, D.; Bouzy, B.; and Métivier, M. 2010. An uct approach for anytime agent-based planning. In *Advances in Practical Applications of Agents and Multiagent Systems*. Springer. 211–220.

Richter, S., and Westphal, M. 2010. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Sabatucci, L., and Cossentino, M. 2015. From means-end analysis to proactive means-end reasoning. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2–12. IEEE Press.

Sabatucci, L.; Cossentino, M.; and Lopes, S. 2019. Service composition with partial goal satisfaction. In *Proceedings of the 1st Workshop on Artificial Intelligence and Internet of Things co-located with the 18th International Conference of the Italian Association for Artificial Intelligence (AIxIA 2019).*

Sabatucci L., Cossentino M., De Simone G. and Lopes S. 2018. Self-reconfiguration of shipboard power systems. In IEEE., ed., *Proc. of the 3rd eCAS Workshop on Engineering Collective Adaptive Systems, Trento (Italy).*

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramirez, M. 2020. Subgoaling techniques for satisficing and optimal numeric planning. *Journal of Artificial Intelligence Research* 68:691–752.

Sheppard, B. 2002. World-championship-caliber scrabble. *Artificial Intelligence* 134(1-2):241–275.

Silver, D., and Veness, J. 2010. Monte-carlo planning in large pomdps. In *Advances in neural information processing systems*, 2164–2172.

Vallati, M.; Chrpa, L.; Grześ, M.; McCluskey, T. L.; Roberts, M.; Sanner, S.; et al. 2015. The 2014 international planning competition: Progress and trends. *Ai Magazine* 36(3):90–98.

Venero, S. K.; Schmerl, B.; Montecchi, L.; Dos Reis, J. C.; and Rubira, C. M. F. 2020. Automated planning for supporting knowledge-intensive processes. In *Enterprise, Business-Process and Information Systems Modeling*. Springer. 101–116.