Monte Carlo tree search compared to A * in airspace configuration decision problem

Preprint · May 2018

DOI: 10.13140/RG.2.2.31176.21762

2 authors, including:

Gabriel Hondet
Ecole Nationale de l'Aviation Civile
2 PUBLICATIONS **0** CITATIONS

SEE PROFILE

Monte Carlo tree search compared to A* in airspace configuration decision problem

Gabriel Hondet, Benoît Viry

May 21, 2018

Abstract

Dynamic airspace configuration is a highly combinatorial partitioning problem. Since exact methods tend to be overwhelmed by the complexity of such problems, a stochastic approach is here considered. This paper presents the Monte Carlo tree search algorithm using UCT search and adapted to one player games. The application of the algorithm to the airspace partitioning problem is then detailed. Finally the Monte Carlo tree search is compared to the A*.

Notations

- m: elementary module;
- S: air traffic control sector, group of modules;
- P: partition of the airspace, group of sectors;
- t: current time;
- $C(P, t)$: cost of a given partition P at time t;
- $C_{tr}(P_1, P_2)$: cost of transition between partitions P_1 and P_2;
- $\pi = [P_1, ..., P_n]$: a sequence of partitions (called *path* in the tree search algorithms);
- $f(\pi)$: cost associated to sequence of partitions π;
- $\mathcal{N}$: the set of nodes;
- $u, v \in \mathcal{N}^2$: nodes
- $h : \mathcal{N} \to \mathbb{R}^+$: heuristic estimating the cost from a node u to a final node;
- μ_v: mean value of outcomes of simulations run through or from a node v;
- T_v: number of simulations run through or from a node v;
- $first(\ell)$: returns the first element of ℓ.

I Introduction

The airspace is divided into sectors, themselves divided into elementary modules. Each sector is managed by a controller working position composed of two controllers. During the day, sectors are split and merged to be able to manage the varying traffic. Splitting creates smaller sectors and is therefore used when traffic gets too dense. On the opposite, merging sectors allows fewer controllers to manage the same airspace, and is therefore used when traffic becomes sparse.

Currently, configuration is mainly decided on the fly by the chief officer. This decision is based on the actual workload on each position. In this approach, future workload estimation is based on the controller's feelings and it therefore lacks a workload prediction tool. This papers aims at providing a method predicting the airspace configuration.

Several methods have been considered to solve the dynamic airspace configuration problem, for instance via genetic algorithms in [13], constraint local search in [11], integer linear programming in [14] or dynamic programming in [3].

In this paper, a temporal sequence of configurations is considered, as in [14] or [13]. Two costs are considered to create the sequence, namely the cost associated to each configuration and a transition cost. The former is based on the workload estimated for one sector given by a simple model. More com-

plex models might be used such as a neural network (see [10]).

The sequence of configurations is extracted from a tree. The resulting sequence will therefore be a path from the root to a leaf of this tree. This problem is highly combinatorial (partitioning of the airspace). Since stochastic tree search algorithms, combined with deep neural networks have proved themselves worthy by outranking the best human player of one of the most highly combinatorial game which is the game of Go, the Monte Carlo tree search [4] algorithm is used in this paper to fulfill the previously mentioned task.

II Previous related works

The dynamic airspace configuration problem requires a model of the airspace. In [13] or [14] the airspace is modelled via a graph. In those graphs, vertices represent elementary modules and an edge links two adjacent modules. In [14], to be able to produce a sequence of configurations, the graph is time dependent. An other way to model the problem is to use a constrained set of configurations as in [3]. This way any configuration will match specified requirements, which can be qualified as hard constraints.

In most cases the cost of a configuration is based on the workload. Each approach seems to give their own representation of the workload. For instance, [2] determined workload density proportionally to the time spent by aircraft in each sector. A simpler version [13] only uses the number of aircraft. On the other hand, more complex methods, involving many more inputs are also available. For instance, [10] used several indicators, such as sector volume, or vertical incoming flows in the next 15 and 60 minutes. Those indicators as inputs to a neural network forecasting the workload.

Other soft constraints appear to be relevant to have a better model of the problem. For instance in [13] and [2] a coordination cost is defined. It represents the surplus of work added by flights travelling from one sector to an other. The shape of the resulting sector is considered, as the simpler is the shape, the easier it is to manage. Complex shapes are therefore avoided, thanks to the notion of compactness in [11] and balconies in [13]. To smooth the transition between two configurations, the work associated with the reallocation of one or more modules is evaluated. This transition cost is included in the cost function being minimised (in [2]).

In this paper, each partition scheme is based upon the number of aircraft in each elementary module and the cost of transition from the previous partitioning. The set of available parititions can be computed from the set of elementary modules and a context which contains all available ATC sectors.

The Monte Carlo tree search algorithm has been widely used in two-player games. Only three years after its apparition in 1990, Brügmann applies it to the Go game in [5]. While the Monte-Carlo tree search is still extensively used in two-player games (and especially the Go game), its adaptation to one player game has been worked on. Auer *et al.* propose an upper confidence bound formula in [8] which appears to be more efficient for one player games. The introduced formula uses the standard deviation of the outcome of the simulations. The latter article also uses the rapid action value estimation (RAVE) technique to quicken the convergence of the algorithm. RAVE uses the "all moves as first" heuristic, in which all moves[1] seen during simulations are considered as a first move. This allows the algorithm to update more statistics in one simulation. Gelly *et al.* use in [9] the RAVE technique coupled with several heuristics. The heuristics bias the initialisaton of a node in the search tree, pre filling its statistics using prior knowledge of the problem.

III Algorithm

The task of building an optimal sequence of partitions (lowering as much as possible the workload of each controller) is fulfilled by a stochastic tree search method, namely the Monte Carlo tree search. To assess the quality of the results, an exact method (here A^*) is used.

[1] a move is informally considered as a decision taken regarding which state to choose while descending the search tree

III.A Monte Carlo tree search

III.A.1 Overview

Monte-Carlo is a best-first search method using stochastic simulations. The algorithm is based on the computation of the reward expectancy of paths which is estimated through Monte-Carlo simulations. The algorithm actually uses two trees, an underlying tree associated to the model (e.g. a game tree) and a search tree. The latter is built incrementally, each step being composed of four phases, namely

1. selection (or tree walk): choosing successively most promising nodes from the search tree,
2. expansion: adding new nodes to the search tree,
3. simulation (or random walk): choosing successively nodes from the model tree, from the expanded node to a leaf,
4. backpropagation (or backup): applying the result of the simulation to the previously selected nodes of the search tree (phases 1 and 2).

Those phases are repeated until a stopping criterion (e.g. memory or time) is reached, resulting in algorithm 1 where the tree policy aggregates phases 1 and 2 to create a new node of the search tree and the default policy gives an evaluation of the newly added node.

Algorithm 1 General MCTS [4]

```
procedure MctsSearchTree(v_0)
    while within computational budget do
        π ← TreePolicy(v_0)
        v ← first(π)
        Δ ← DefaultPolicy(v)
        Backup(π, Δ)
    end while
end procedure
```

III.A.2 Extending the search tree

Selection (algorithm 2) The aim of the selection is to build a path from the root of the search tree by choosing successively the most promising[2] node. Given a node u that has previously been selected, the best node – according to a *tree policy* – among the children of u is chosen. This type of problem can be solved by bandits methods. Those methods consist in, given a bandit in front of several slot machines (multi armed bandit), deciding which machine will bring the highest reward knowing the past results. The objective of bandits methods is thus to maximise the reward and minimise the regret of not playing the best machine.

The bandit problem has been applied to MCTS via the Upper Confidence Tree algorithm in [12] using the UCB1 equation 1. Let u be the node from which a child v must be selected to go deeper in the tree, T_u the number of simulations carried out from node u (which includes any simulation from nodes in any subtree of u) and β a chosen constant. The selected child v is the one maximising the *UCB1* equation

$$\mu_v + 2\beta\sqrt{\frac{2\,log\,T_u}{T_v}} \tag{1}$$

While the previous equation is well suited for two players games, it can be tweaked to improve its efficiency in one player games or sequencing problems. An alternative using the standard deviation σ_v of the outcome of the previous simulations involving node v is proposed in [8] called the *UCB1-tuned* equation

$$\mu_v + \beta\sqrt{\frac{log\,T_u}{T_v}\,min\left(\frac{1}{4}, \sigma_v^2 + \sqrt{\frac{2\,log\,T_u}{T_v}}\right)} \tag{2}$$

Exploration exploitation trade-off The constant β answers to the exploration-exploitation dilemma. In the UCB1 equations 1 and 2, the right hand term increases the UCB value of less explored nodes to consider them as still promising. A high β value will therefore make the algorithm prone to try unvisited nodes while a low value will consider almost exclusively the results obtained so far, even if other paths are better but unexplored.

Expansion (algorithm 3) If the selected node has one or more unvisited children, the selection stops and one of them is added to the search tree randomly, the latter being thus expanded by this new node.

[2] i.e. possibly leading to the best evaluation

Algorithm 2 UCT algorithm

```
function TreePolicy(π)
    v ← first(π)
    if v is terminal then
        return π
    else
        if v is not fully expanded then
            return Expand(v) ∪π
        else
            f ← BestChild(v)
            return TreePolicy(f ∪ π)
        end if
    end if
end function
function BestChild(v)
    return argmax{UCB(v′)|v′ children of v}
end function
```

Algorithm 3 Expansion

```
function Expand(u)
    ℓ ← {v|v children of u, T_v = 0}
    return random element from ℓ
end function
```

III.A.3 Simulation and backpropagation

The two steps described in this paragraph aim to guess the reward that can be expected from a path including a given node.

Simulation Once a node has been expanded, random nodes are chosen successively until a terminal state is found. The cost of the resulting path, from the root to the node is then evaluated and backpropagated to all the ancestors of the expanded node.

Heuristic One might want to bias the randomness while choosing nodes. This would imply using a heuristic which has to be able to discriminate a node between its siblings.

Backpropagation The backpropagation consists in updating the values required to carry out the tree policy. The values must be updated incrementally since the backpropagation function has the current value of the parameters and the result of the simulation as parameters. Thus, for UCB1 equation, the expected reward (mean) and the total count are updated this way

Algorithm 4 UCB1 backpropagation

```
procedure Backpropagate(u, r)
    δ ← r − μ_u
    μ_u ← μ_u + δ/(T_u+1)
    T_u ← T_u + 1
end procedure
```

For the UCB1-tuned, the standard deviation has to be computed. It results in algorithm 5 which introduces the value $m_2(u)$ associated to a node u to compute the standard deviation via the formula

$$\sigma_u^2 = \frac{m_2(u)}{T_u}. \tag{3}$$

III.B Sequence building

Once the stopping criterium mentioned in III.A.1 is matched, the sequence of states can be extracted from

Algorithm 5 UCB1-tuned backpropagation

procedure BACKPROPAGATE(u, r)
  $\delta \leftarrow r - \mu_u$
  $\mu_u \leftarrow \mu_u + \frac{\delta}{T_u+1}$
  $\delta' \leftarrow r - \mu_u$
  $m_2(u) \leftarrow m_2(u) + \delta\delta'$
  $T_u \leftarrow T_u + 1$
end procedure

the search tree.

III.B.1 Choosing nodes

To build the final sequence, nodes are chosen according to a criterion. Chaslot *et al.* in [6] propose several methods to select nodes,

- max-child: select the child with highest mean reward;
- robust child: select the most visited root child;
- secure child: select the child maximising a lower confidence bound.

III.B.2 Iterative pathfinding

The path is built iteratively by calling successive Monte Carlo tree searches. Say an MCTS has been called on a node u. Once a stopping criterium is matched, a node v among the ones reachable from u is selected via one of the previously mentioned policies. Then the MCTS algorithm is called back with node v as the root node. The final path is composed of the successive roots. The algorithm is described in 6.

III.C A*

To evaluate the exactness of the paths given by the MCTS algoithm, the A* algorithm is used. A* is an exact best first search pathfinding algorithm which uses a heuristic function to guide its search. The algorithm is given in 7 where u_0 is the initial state, T the set of terminal nodes, h a heuristic function estimating the cost from a state u given as argument to a final state, $k\colon \mathcal{N}^2 \to \mathbb{R}$. The function $first(G)$ returns the first element of G and $f-\text{insert}(G, v)$ inserts v in G ordering first by f increasing then by g decreasing.

Algorithm 6 Path building. The NEXTNODE (here max-child) function is one among those in III.B.1.

function MCTSSEARCH(u_0, n)
  $u \leftarrow u_0$
  $\pi \leftarrow \{u\}$
  for $i = 1$ to n **do**
    MCTSSEARCHTREE(u)
    $u \leftarrow$NEXTNODE(u)
    $\pi \leftarrow \pi \cup \{u\}$
  end for
  return π
end function
function NEXTNODE(u)
  return $argmax\{\mu_v | v \text{ children of } u\}$
end function

Heuristic To be sure to have the optimal solution, the heuristic h has to be *minimal* i.e. let h^* be the optimal heuristic (the one giving the true distance from a node to a final state), then for any node u, $h(u) \leq h^*(u)$.

III.D Theoretical performances

In this paper, a high branching factor problem is considered.

Let H be the height of the tree, K the branching factor. Then, the complexity of the A* is expected to be, in the worst case, exponential in H (see [1]).

Since the Monte carlo tree is a stochastic method, one must wait for a satisfactorily converging solution rather than the exact solution. It has however been proven in [12] that the bias of the expected payoff tends to zero.

IV MODEL

In this section, we discuss the model established to approach this configuration problem. First we define a structure for a controlled sector and all possible

Algorithm 7 A* algorithm [1]

procedure A*(u_0)
  $G \leftarrow u_0; D \leftarrow \emptyset; g(u_0) \leftarrow 0; f(u_0) \leftarrow 0$
  father(u_0) $\leftarrow \emptyset$
  while $G \neq \emptyset$ **do**
    $u \leftarrow first(G); G \leftarrow G \backslash \{u\}$
    $D \leftarrow D \cup \{u\}$
    if $u \in T$ **then**
      return father
    end if
    for v in childen of u **do**
      if $v \notin D \cup G$ or $[g(v) > g(u) + k(u,v)]$ **then**
        $g(v) \leftarrow g(u) + k(u,v)$
        $f(v) \leftarrow g(v) + h(v)$
        father(v) $\leftarrow u$
        $f-$insert(G, v)
      end if
    end for
  end while
end procedure

transitions through time. Then a workload model is defined. And finally we define a cost per partition and a cost function aimed to be minimized.

IV.A State

IV.A.1 Time step

Since the overall goal is to compute a temporal sequence of partitions over a day, the day has to be sampled. In the model tree, a timestep separates a node from its children. Practically, a timestep is one minute.

IV.A.2 Partitions

In the configuration problem, the airspace needs to be divided into sectors. Each sector represents an area controlled by two controllers and is composed of one or more elementary modules. A partition P is a set of non-overlapping sectors covering all the airspace.

A sector is a group of elementary modules, but in practice only a collection of sectors are allowed in this model. This collection is referred to as the context. It is the list of operation ATC sectors actually used in ATC centres.

IV.A.3 Transitions

During the day, and with the evolution of the traffic, the workload (rigorously defined in IV.B) varies. To help controllers to maintain a homogeneous level of workload, the airspace partitioning needs to be adapted through successive transitions starting from an inital partition. This maneuver requires either coordination or the opening of a new position and hence increases controller's workload. This is the reason why a full reconfiguration isn't feasible, and only a subset of all available transitions can be operated.

In the model considered here, three transitions are possible namely a merge, a transfer or a split. For an airspace composed of three modules A, B, and C, those actions represent:

- merge: $\{\{A,B\},\{C\}\} \quad \rightarrow \quad \{\{A,B,C\}\}$
- split: $\{\{A,B,C\}\} \quad \rightarrow \quad \{\{A,B\},\{C\}\}$
- transfer: $\{\{A,B\},\{C\}\} \quad \rightarrow \quad \{\{A\},\{B,C\}\}$

Each transition is composed of at most one merge, split or transfer; more would result in too complex reconfigurations.

Tree translation

In the tree used in the algorithm exposed earlier, each node has an embedded state. Considering a node u, its children are the elements of the set of nodes having as states the possible transitions from the state of u. Going down in the tree by selecting nodes is then equivalent to moving forward through time, each step in the tree being a time step.

IV.B Workload

Let us now precise the model established to evaluate a workload metric. The workload depends on external criteria such as sector and traffic complexity or human factors (e.g. controller health or stress). In the

model assumed here, the workload felt by a controller is directly and only related to the traffic complexity.

The simplest approch considers only the number of aircraft per sector N_{aircraft}. We define two arbitary thresholds, namely th_{low} and th_{high}, defining three zones where the workload is low (resp. normal and high) if $N_{\text{aircraft}} < th_{\text{low}}$ (resp. $th_{\text{low}} \leq N_{\text{aircraft}} \leq th_{\text{high}}$ and $th_{\text{high}} < N_{\text{aircraft}}$). Those workload levels are translated to probabilities p_{low}, p_{normal} and p_{high}, where $p_{\text{low}}(S)$ is the probability that the sector S is underloaded (resp. balanced and overloaded). In short, for a sector S at time t and N_{aircraft}:

$$\begin{aligned} p_{\text{low}}^{S,t} &= \mathbb{1}_{[0,th_{\text{low}}[}(N_{\text{aircraft}}) \\ p_{\text{normal}}^{S,t} &= \mathbb{1}_{[th_{\text{low}},th_{\text{high}}]}(N_{\text{aircraft}}) \\ p_{\text{high}}^{S,t} &= \mathbb{1}_{]th_{\text{low}},\infty[}(N_{\text{aircraft}}) \end{aligned} \tag{4}$$

where $\mathbb{1}$ is the indicator function defined by:

$$\mathbb{1}_I(x) = \begin{cases} 1 & \text{if} \quad x \in I \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

This expression of the workload, expressed in terms of probabilities, allows us to use a more complex model for the workload such as a neural network (see [10]). This method learns the probabilities p_{low}, p_{normal} and p_{high} based on parameters including traffic complexity and the complexity of the sector (e.g. airspace volume).

IV.C Partition cost and cost function

IV.C.1 Partition cost

In order to evaluate a given partition P at time t, a cost $C(P,t)$ needs to be defined. In this paper, this cost depends on the workload in each sector. The definitions given in [7] are used to represent a high (respectively normal and low) cost for partition P:

$$\begin{aligned} c_+(P,t) &= \sum_{S \in P_t} \delta_h(S,t) \cdot p_{\text{high}}^{S,t} \cdot |S|^2 \\ c_=(P,t) &= \left(\sum_{S \in P_t} \delta_n(S,t) \cdot p_{norm}^{S} \cdot |S|^{-2} \right)^{-1} \\ c_-(P,t) &= \sum_{S \in P_t} \delta_l(S,t) \cdot p_{\text{low}}^{S,t} \cdot |S|^{-2} \end{aligned} \tag{6}$$

with $\delta_h(P,t)$ (resp. $\delta_n(P_t)$ and $\delta_l(P_t)$) equals 1 if the probability p_{high} (resp. p_{normal} and p_{low}) is superior to the two others, and 0 otherwise.

It is now possible to assign a cost to each partition. This cost is linear regarding c_+, $c_=$, c_- and n (cardinal of the partition P). The partition cost is defined as follows:

$$C(P,t) = \alpha c_+ + \beta c_= + \gamma c_- + \lambda n \tag{7}$$

The parameters α, β, γ and λ (all positive) determine a priority on which parameter to optimize. For instance, a high value α help to minimize c_+, hence the overall number of sectors with too much traffic. In a real application, it maybe interesting to order those parameters as follow: $\alpha > \gamma > \beta > \lambda$.

IV.C.2 Transition cost

In an operational context, each reconfiguration increases the workload for controllers. This needs to be considered via the definition of a transition cost. A transition is the difference of two partitions separated by only on time step. The transition cost is then

$$C_{tr}(P_1,P_2) = \begin{cases} 0 & \text{if} \quad P_1 = P_2 \\ 1 & \text{otherwise} \end{cases} \tag{8}$$

IV.C.3 Objective function

Having defined a cost for partitions and transitions, it is now possible to aggregate everything in order to build a cost function over an entire path. For a path $\pi = [P_0, \dots, P_n]$ with a time from t_0 to t_n, the cost function is given by:

$$\begin{aligned} f(\pi) = C(P_0,t_0) + \sum_{i=1}^{n} [&C(P_i,t_i) + \\ &\theta \cdot C_{tr}(P_{i-1},P_i)] \end{aligned} \tag{9}$$

with $\theta > 0$ a parameter to determine. This is the objective function to minimize in the A* algorithm.

The Monte Carlo Tree search algorithm maximizes an objective function. This function can be interpreted as the reward of a path. Given the loss function defined previously, the target function can be

constructed as:

$$Q(\pi) = \frac{1}{f(\pi)} \tag{10}$$

IV.D Heuristic

To use effectively the A* algorithm, a heuristic is needed. Using the null heuristic ($\forall u \in \mathcal{N}, h(u) = 0$) – and thus using Dijkstra algorithm – revealed to be inefficient considering the branching factor. Let $k \in \mathbb{N}$, u a node selected at time k. The heuristic $h(u)$ gives the cost of the path starting from u to a leaf which is reached at a time t_f. The heuristic is built without considering transitions (transition cost and IV.A.3). Let P_i^* be the partition with minimal cost at time i, the heuristic is then

$$h(u) = \sum_{i=k+1}^{f} C(P_i^*, t_i) \tag{11}$$

This allows to generate the sequence of the best partitions among all possible at each time step.

V Results

V.A Experimental setup

For the tests, a fictitious area is used. It contains 5 elementary modules and 12 control sectors. The resulting model tree has a mean branching factor of 6. Regarding the branching, the smallest French area – which is Paris West – has a branching factor of 16 for 20 control sectors and 12 elementary modules. Considering the already important branching factor, a stochastic tree search method seems appropriate.

The traffic is simulated the following way, each elementary module has an associted list which, at index i has the number of aircraft in it at time step i. Those lists have been written by hand.

V.B Accuracy

The graph 1 asserts the correct behaviour of both algorithms A* and MCTS regarding grouping and degrouping. On the y axis is indicated the number of sectors. The trafic activity is the following,

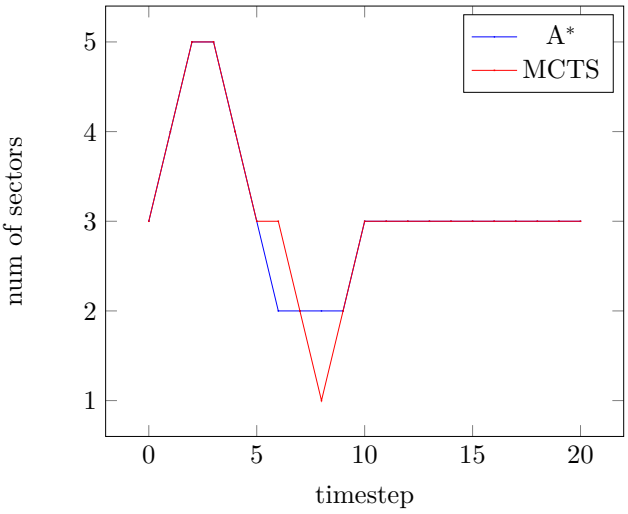

Figure 1: Grouping profile

1. The first part of the scenario shows a busy situation, all sectors are overloaded;
2. then follows a quiet moment with few aircraft, much of the sectors are underloaded;
3. the end of the scenario has two overload modules and the rest under normal or under load.

The expected behaviour is a tendency to increase the number of sectors through splits to cope with part 1 followed by many merges to reduce the number of sectors in response to phase 2. The algorithms should end with a specific sector created, the one overloaded in 3 and the rest grouped. Seeing 1, both algorithm behave as expected.

Accuracy can be quantitatively assessed by comparing the outcome of the MCTS algorithm against the A* cost and a greedy algorithm cost. The time allowed to the MCTS to run is considered long enough to allow the MCTS to converge to its best solution. The results are summarised in table 1.

The graph 2 shows the evolution of the difference of costs with varying depth. The error rate is expected to grow as the depth increase since the MCTS might fail to explore the best nodes among the exponen-

Alg.	Cost	Err.
A*	77.31	0%
MCTS	82.48	6.7%
Greedy	90.50	17.1%

Table 1: Comparison of several methods on the generated scenario ($\theta = 3$).

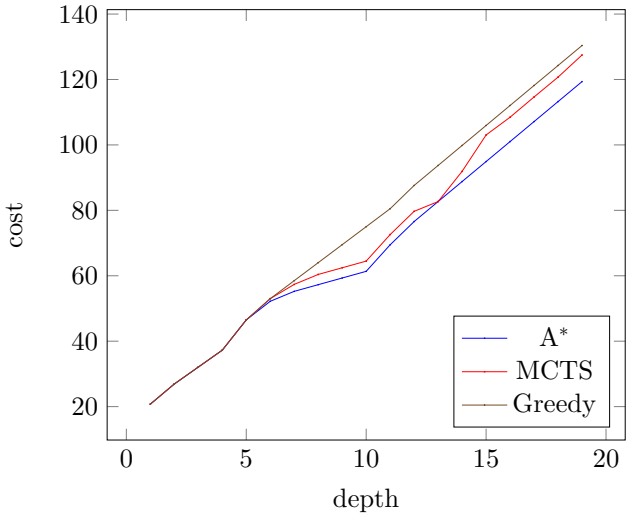

Figure 2: Cost difference for varying depth with 0.5 seconds allowed to MCTS and $\theta = 3$.

tially growing amount of them. The figure 2 does not show a constant growth of error rate between MCTS and A*. As hoped, the MCTS remains always better than the greedy algorithm.

V.C Converging speed

Since the MCTS converges to a solution, one expects to know the time required to be close enough to the optimal solution. To have an idea of this lapse, several MCTS are run with different computing time. This results in a graph mapping the time allowed to compute each step of the MCTS to the cost of the final path. To get rid of the undesired varations due to the stochastic nature of the algorithm, the latter process is run several times; which allows one to compute the mean over all computed costs.

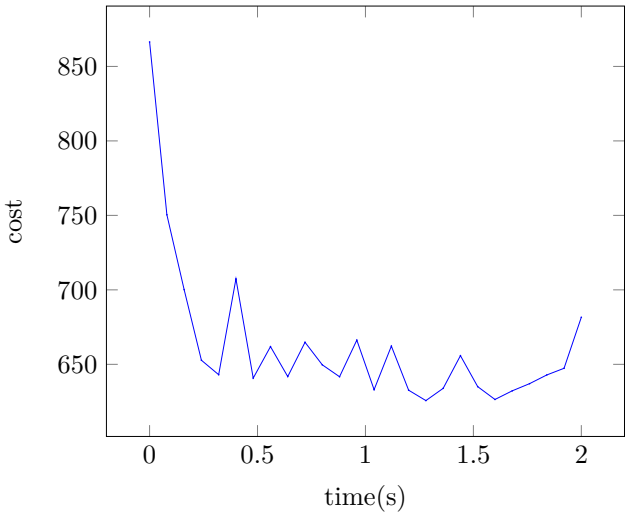

Figure 3: Monte Carlo tree search called with varying time allowed to compute each step, with a depth of 40 and $\theta = 11$. The cost seems to be approximately stable from $t = 0.5$s.

The resulting graphs are shown in figure 3 in which it can be seen that the algorithm reaches its best cost when it runs for 1 second per step, but the cost is approximately stable from 0.5 seconds.

VI Conclusion

The presented work provides the basis of a method able to create dynamic airspace overture schemes. The model is built around a quantification of the workload felt by a air trafic controller. This quantification is then used to create a cost function to be minimised. The highly combinatorial nature of the problem makes it prone to be solved with stochastic methods. The Monte Carlo tree search algorithm is then introduced since it has been used in some of the most combinatorial problem (game of Go). In this paper, the algorithm is modified to take into account not only a result but all the path leading to this leaf. It is also adapted to the one-player game nature of

the problem.

Further work Since all the tests has been run using the same fictitious area with the same trafic, neither the influence of the complexity of the airspace nor of the trafic have been assessed. It would be thus relevant to carry further tests on bigger areas with heavier and differing trafic. This can be emphasised by [7], in which the time needed to compute the cost with A* depends heavily on the trafic itself.

More complex workload models can also be used such as neural networks in [10].

References

[1] Jean-Marc Alliot, Thomas Schiex, Pascal Brisset, and Frédérick Garcia. *Intelligence artificielle et informatique théorique*. Cépadues, 2nd edition, 2002.

[2] Judicaël Bedouet, Thomas Dubot, and Luis Basora. Towards an operational sectorisation based on deterministic and stochastic partitioning algorithms. In *The Sixth SESAR Innovation Days*, 2016.

[3] Michael Bloem and Pramod Gupta. Configuring airspace sectors with approximate dynamic programming. 2010.

[4] Edward Browne, Cameron Band Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactionson Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[5] Bernd Brügmann. Monte carlo go. October 1993.

[6] Guillaume Chaslot, Mark Winands, H Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. 04:343–357, 11 2008.

[7] Etienne Ferrari. Prévision des ouvertures de secteurs de contrôle aérien, 2017.

[8] Romaric Gaudel and Michèle Sebag. Feature selection as a one-player game. 06 2010.

[9] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michele Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[10] David Gianazza. Forecasting workload and airspace configuration with neural networks and tree search methods. *Artificial intelligence*, 174(7-8):530–549, 2010.

[11] Peter Jägare, Pierre Flener, and Justin Pearson. Airspace sectorisation using constraint-based local search. In *ATM 2013, June 10-13, Chicago, IL*. Federal Aviation Administration, 2013.

[12] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, volume 6, pages 282–293. Springer, 2006.

[13] Marina Sergeeva, Daniel Delahaye, Catherine Mancel, and Andrija Vidosavljevic. Dynamic airspace configuration by genetic algorithm. *Journal of traffic and transportation engineering (English edition)*, 4(3):300–314, 2017.

[14] Tambet Treimuth, Daniel Delahaye, and Sandra Ulrich Ngueveu. A branch-and-price algorithm for dynamic sector configuration. In *8th International Conferenceon Applied Operational Research (ICAOR)*, volume 8, pages 47–53, 2016.