Monte-Carlo Fork Search for Cooperative Path-Finding

Bruno Bouzy

LIPADE, Université Paris Descartes, FRANCE,
bruno.bouzy@parisdescartes.fr

Abstract. This paper presents Monte-Carlo Fork Search (MCFS), a new algorithm that solves Cooperative Path-Finding (CPF) problems with simultaneity. The background is Monte-Carlo Tree Search (MCTS) and Nested Monte-Carlo Search (NMCS). Regarding MCTS, the key idea of MCFS is to build a tree balanced over the whole game tree. To do so, after a simulation, MCFS stores the whole sequence of actions in the tree, which enables MCFS to fork new sequences at any depth in the built tree. This idea fits CPF problems in which the branching factor is too large for MCTS or A* approaches, and in which congestion may arise at any distance from the start state. With sufficient time and memory, Nested MCFS (NMCFS) solves congestion problems in the literature finding better solutions than the state-of-the-art solutions, and it solves N-puzzles without hole near-optimally. The algorithm is anytime and complete. The scalability of the approach is shown for gridsize up to 200×200 and up to 400 agents.

1 Introduction

Cooperative pathfinding (CPF) addresses the problem of finding paths for a set of agents, for them to move to their goals. At each timestep, every agent moves to a neighbouring cell. The set of agents has to find the minimal cost for reaching its set of goals. The cost is the elapsed time. There are two families of approaches: the coupled approach and the decoupled approach. In the coupled approach, the whole set of agents is considered as one. One main bottleneck is the size of the set of joint actions which is exponential in the number of agents. In the decoupled approach, each agent is considered individually and the main obstacle is managing the collisions between the agents.

A* is the prototype of the coupled approach. A* with operator decomposition (A*+OD) [16] is a speed-up version of A*. ICTS [14] searches a solution in an Incremental Cost Tree (ICT). The weakness of the coupled approach is its inability to solve large problems or including complex coordination between agents. However, very recently, a new work, TOMPP [21], modeling multi-agent pathfinding as a network flow, contradicts this statement. Windowed Hierarchical Cooperative A* (WHCA*) [15] illustrates the decoupled approach. Push&Swap (P&S) [10] solves CPF problems on graphs with at least 2 empty cells. TASS (Tree-based Agent Swapping Strategy) is designed to solve problems with at most

4 empty cells [7, 6]. These solvers are very fast but they cannot improve their solution with more computing time.

In this paper we present Monte-Carlo Fork Search (MCFS) and its nested version Nested Monte-Carlo Fork Search (NMCFS) for CPF. The background is Monte-Carlo Tree Search (MCTS) [8] and NMCS [4]. MCTS and NMCS have been applied with success to many planning problems such as Morpion Solitaire [12]. However, their weakness lies in their inability to deal with the high branching factor of CPF problems. To solve CPF problems, our goal is to design a new algorithm that has the strength of MCTS and NMCS, and is not sensitive to the branching factor. The key idea is to make the built tree cover all the interesting parts of the game tree in a balanced way.

Like MCTS, MCFS is anytime. It has been compared to TOMPP, TASS, and Push&Swap on their test problems: specific congestion problems [9, 6], and N-puzzles without hole problems with simultaneous actions [21]. For scalability, MCFS has been assessed on grids with size up to 200×200, with obstacles (20%) and with a number of agents up to 400. Some of the results obtained with the parallel criterion are manually proved optimal, and the others are conjectured to be near-optimal. The results are translated into the sequential criterion to be compared with the numerous work using this criterion, and we show that MCFS obtains better results on this criterion.

The outline of the paper is the following. First, we give the CPF problem definition. Secondly, we relate previous work on CPF and on MCTS. Thirdly, we present the MCFS and NMCFS algorithms. Fourthly, we describe the experiments and the significant results. Fifthly, we discuss the properties of the algorithm. Finally, we conclude and present future work.

2 CPF problem definition

The CPF literature contains a lot of work. Each work uses its specific criteria to define the problem, which makes comparisons difficult. In the literature, the cells are mostly squares used with 4-connectivity or 8-connectivity. Whatever the grid connectivity, an agent can move on the neighbouring cells or stay, and respect the two following rules: No two agents can be on the same cell (**Rule 0**), No two agents can swap (**Rule 1**). With 8-connectivity, specialized rules must define whether two agents may cross diagonally or not, whether an agent can move diagonally when one obstacle is on its side, or whether it can cross two obstacles situated on a diagonal. In this paper, we use squares and 4-connectivity with rule 0 and rule 1 only.

The most important feature is the simultaneity or sequentiality of individual movements. A lot of work fall in the sequential category. In this work, we use simultaneous, parallel moves. When the individual actions are executed in parallel, the agents must respect rule 0: when several agents wish to move towards the same cell, only one of them can actually move to the cell. However, an agent may move to an occupied cell provided that this cell is freed by its occupant during the timestep. Therefore, it is worth noting than joint actions

including circular individual actions are possible. For instance, 4 agents situated on 4 adjacent cells can move circularly.

In the multi-agent context, two targets can be optimized: the sum of individual costs (**SUM**), the maximum of individual costs (**MAX**). In most work [16], [14], [15], [10], [7], the optimized target is SUM. However, in this paper, we optimize MAX, that is to say the global elapsed time. Some work such as [21] fall into this category. The maximum of individuals costs is also commonly called makespan. Comparing work optimizing SUM with our work is still possible provided that we estimate the SUM target by counting the individual actions of the sequences found.

At each timestep, an agent can either stay on its cell or move to an adjacent cell provided it respects rule 0 and rule 1. During a timestep, all the agents act simultaneously. The joint goal is found when all the agents have reached their individual goals. The problem is to find the minimum number of timesteps to reach the joint goal. In the following, a position refers to the set of positions of the agents. A path, a plan, an episode, a simulation refer to the same concept: a sequence of positions linked two-by-two by a joint action. The goal is the final position. A hole is an empty cell. A CPF problem may have or not have holes. Table 1 shows an instance of a CPF problem on a 4×4 grid.

Table 1: A 4×4-puzzle problem without hole. The first number in a cell represents the position of an agent, and the second number its goal. For instance, agent 0 is located on the lower right cell and its goal is the cell at the top left.

11 0	13 1	14 2	12 3
6 4	15 5	5 6	9 7
10 8	7 9	4 10	1 11
3 12	2 13	8 14	0 15

3 Related work

3.1 Cooperative Path-Finding

The starting point of CPF is centralized A*. A* works optimally on very small problems with very few agents. To avoid the exponential number of actions in the number of agents, Standley has proposed Operator Decomposition (OD) [16]. A*+OD and an admissible heuristic are optimal [16].

ICTS [14] is a two-level search: a global level and a low level. At the global level, the search proceeds on an Incremental Cost Tree (ICT) in which one node corresponds to a vector of costs. At depth δ, the sum of the costs of a node is the optimal cost plus δ. A node is a goal for ICT when there is an effective combination of individual paths without conflict and with costs corresponding to the costs of the node.

TOMPP [21], [20] optimizes the global elapsed time. It models a multi-agent path-finding problem as a network flow and shows the equivalence between the two models. It uses integer linear programming to solve the network flow problem and the multi-agent path-finding problem as a consequence. TOMPP is tested

on the N-puzzle without hole, and in many-agents-many-obstacles problems. TOMPP is time-optimal.

In video games, the optimal methods are not used because they are not scalable. Adaptations of A* are used instead. Windowed HCA* (WHCA*) decomposes the whole task into a series of single agent searches and searches m-steps plans [15]. Sub-optimal methods with some completeness guarantees on well-specified sub-classes of problems give good results: P&S [10] solves CPF problems on graphs with at least 2 empty cells. TASS (Tree-based Agent Swapping Strategy) [7] is designed to solve problems with at most 4 empty cells. [9] and [6] provide interesting test problems. MAPP (Multi-Agent Path Planning) [19] defines the Slideable class and presents a method complete on this class of problems.

3.2 MCTS

MCTS is the approach that revolutionized computer go in 2006 with the UCT method [8] and the Go playing programs Crazy Stone [5] or MoGo [13]. In the following years, MCTS was also successful in many other games and planning problems [3]. MCTS iteratively expands a tree starting a the root, and launches simulations to obtain rewards at the end of the games. An iteration has four phases: selection, expansion, simulation and propagation. In the selection phase, MCTS starts at the root of the tree and go down to a leaf node by using the UCB rule [1]. When reaching a node not fully expanded, MCTS chooses the next state with its simulation policy, and adds the corresponding node into the tree. Then MCTS starts the simulation phase. At the end of the simulation, the reward is obtained and propagated into the nodes browsed by the selection phase.

4 MCFS

This section presents the rationale of MCFS compared to MCTS, the algorithm in its nested version or not, how the basic simulations are performed, and the necessary pre and post-processing. Overall, since congestion may arise at any point of a solution plan, the key idea of MCFS is to explore the game tree in a balanced manner by:

- making use of the whole simulation to build the tree,
- forking new paths at appropriate nodes of the built tree,
- not entering into the curse of the branching factor.

4.1 Similarities and differences with MCTS

Like MCTS, MCFS iteratively builds a tree with four stages: selection, expansion, simulation and propagation. The simulation stage and the propagation stage in MCFS remain identical to the corresponding stages in MCTS. Like MCTS, MCFS uses the optimism faced to uncertainty principle by using the

UCB rule [1]. The difference between MCTS and MCFS mainly lies in the manner in which the game tree is explored, the start of the next simulation is selected, and the built tree is expanded at each iteration. MCTS explores the game tree by respecting a width principle: at a node, MCTS prefers to explore an unexplored action rather than an already explored child node. Consequently, MCTS can be stuck near the root if the branching factor is high. The tree built by MCTS browses the upper part of the game tree only. Moreover, the next simulation starts from a leaf node of the built tree. Furthermore, in standard MCTS, MCTS adds one node after each simulation.

MCFS explores the game tree in a depth-first manner. First, after each iteration, *all* the states encountered during the simulation become nodes added into the MCFS tree. Secondly, at the beginning of an iteration, with the help of the UCB rule, MCFS selects the best node to fork among *all* the nodes of the built tree. This is very different from the MCTS selection stage that starts from the root node and iteratively chooses a child node with the UCB rule until it reaches a node with an unexplored action. In MCFS, the next simulation starts from this selected node which is an interior node of the built tree. In MCFS, the built tree has only one leaf node: the goal node.

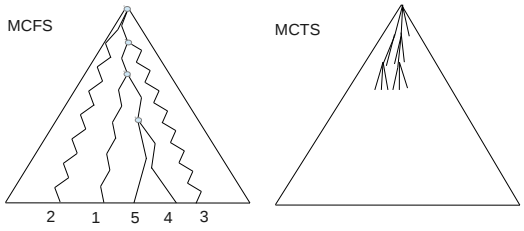

Fig. 1: The tree built by MCFS (left) and MCTS (right) within the game tree represented by a triangle. The root is at the top. The goal is the bottom line. Each action is a small straight line. The circles represent nodes whith a fork. The numbers are the iteration numbers.

Figure 1 shows an overview of how the trees built by MCFS and MCTS fill the whole game tree. As iterations are going on, the MCTS tree is deepening, and the MCFS tree is widening. At iteration 1, a length-12 sequence is found. The second iteration selects the root as a starting node, and a length-15 sequence is found. Iteration 3 selects a node situated at depth 2 on the current best sequence, and finds a length-15 sub-sequence giving a length-17 sequence. Iteration 4 selects a node at depth 4 and finds a sub-sequence of length 5 giving a length-9 sequence, the new current best. Iteration 5, selects a node at depth 6 and finds a sub-sequence of length 2 giving a sequence of length 8, the new current best. And so on until the iteration budget is exhausted. The idea to allow MCFS to fork sub-sequences anywhere in the game tree is respected.

```
1  int NMCFS(a, b, bestSeq, lev)
2  begin
3      if lev == 0 then
4          return sample(a, b, actualSeq)
5      end
6      n = 1 ; lmin = +∞ ; actualSeq ; Node root(a)
7      while n ≤ it do
8          Node nd = root.selectNode() ; pos = nd.positions
9          l = NMCFS(pos, b, actualSeq, lev − 1)
10         if l + nd.depth < lmin then
11             lmin = l + nd.depth ; bestSeq = seq(root(a), nd) + actualSeq
12         end
13         nd.backUp(l) ; nd.append(actualSeq, l, b) ; n = n + 1
14     end
15     return lmin
16 end
```

Algorithm 1: NMCFS

4.2 MCFS and NMCFS Algorithms

Since the pseudo-code of NMCFS is very similar to that of MCFS, Algorithm 1 directly shows the pseudo-code for NMCFS. (The MCFS pseudo-code corresponds to the NMCFS pseudo-code with $lev = 1$). NMCFS takes the starting position (a), the goal position (b) and the nesting level (lev) as inputs. NMCFS returns the length of the best plan found ($bestSeq$) (line 1). If the level is zero, NMCFS calls *sample* the basic simulation function. n is the number of simulations performed so far. $lmin$ contains the current best length. $actualSeq$ is the sequence of positions played at each iteration. $root$ is the root node initialized with the starting position a. NMCFS is a while loop (lines 7-14). it is the number of iterations to perform. During an iteration, the best node of the tree is selected (line 8) with equation 1. NMCFS is called, starting on this node with the nesting level minus one (line 9). The length of the simulation plus the depth of the node in the tree is compared to $lmin$ (line 10). If the comparaison holds, $lmin$ and $bestSeq$ are updated (line 11). The nodes between the selected node and the root are updated with l, and the tree is expanded by adding $actualSeq$ to it ($backUp$ and $append$ line 13).

In equation 1, it is essential to notice that $argmin$ is applied *over the whole tree*, whis is very different from the MCTS selection. $nd.lmin + nd.depth$ is used as an exploitation term to focus the search near the current best sequence. Concerning the exploration term, C is a parameter set experimentally. var is the variance over the lengths of the sequences going through the node. It enables MCFS to prefer nodes with high variance. $nForks$ is the number of times the node has been selected so far. $log(n)$ makes it possible to forget no node for n sufficiently large.

```
1 int sample(p, b, seq) begin
2     l = 0
3     while ((p ≠ b) and (l < ls)) do
4         action = pseudoRandomChoice(b)
5         p = play(p, action) ; seq[l] = p ; l = l + 1
6     end
7     return l
8 end
```

Algorithm 2: Sample

$$nd = \underset{builttree}{\arg\min}(lmin + depth - C\sqrt{\frac{var\log(n)}{1 + nForks}}) \quad (1)$$

4.3 Basic simulations

Algorithm 2 shows function *sample* that executes a basic simulation starting on position p. seq contains the actual sequence played out. l is the current length of the simulation. It executes a loop while the goal is not reached and l does not exceed ls the maximal length of sequences. The joint action is determined by *pseudoRandomChoice*. It returns a joint action according to a pseudo-random policy. *pseudoRandomChoice* does not enumerate all the joint actions, which could be tricky when the number of agents is large. Instead, each agent says which cell it wants to move on: its wish. If all the wishes are compatible with Rule 0 and Rule 1, then the joint action is valid and returned. When two wishes are in conflict, the conflict is solved by prioritizing one agent over the other at random. When all the conflicts are solved, the wishes become the actual elementary actions, and the joint action is returned. If some conflicts cannot be solved after a given number of tries, then the agents relax their wish, and the wishes are formulated again. Without relaxing the wishes, the joint action contains optimal individual actions only. With relaxing, the joint action may also contain non optimal elementary actions. The function *play* transforms the position according to the effects of the joint action.

4.4 Pre and Post processing

Before launching NMCFS, all the distances between two cells assuming the obstacles and no agent are computed, and stored in a table to be used in the simulations. After completing a best simulation, NMCFS counts the number of elementary actions used.

5 Experiments

5.1 Set of problems

To assess our approach, we have taken three kinds of problems. First, we addressed the congestion problems of TASS and Push&Swap. [6] contains six in-

teresting problems which we named from 515 up to 520. Problem 5xy refers to the problem defined by figure 5.xy pages 58–61 in [6]. For instance, 515 contains 10 agents with 19 cells and 520 contains 4 agents with 8 cells. [10] contains eight specific problems: Tree, Corners, Tunnel, String, Loopchain, Connector, Rotation, and Stacks. The first six problems contain between 3 and 7 agents on small graphs with at most 18 nodes. The last two problems have 16 agents with 24 or 25 nodes.

Secondly, we addressed some N-puzzle problems with one hole ($N = 8, 15, 24$) or no hole ($N = 9, 16, 25$). The N-puzzle problem with one hole in which SUM is optimized is known to be NP-hard [11]. The N-puzzle problem without hole in which MAX is optimized is very representative of the CPF class of problems.

Thirdly, we addressed medium-sized-to-large-grid-many-agents-with-obstacles problems with a low level of congestion. [16] and [17] contain two such examples which we call $s10$ and $s11$. In video games, the problems encountered may have one thousand agents or more [19]. Furthermore, they may have large grids, like 512×512 in reference benchmarks [18]. Since our pre-processing stage that computes all the distances between two cells of the grid is currently limited to 200×200 grids, our approach cannot deal with such benchmarks. Instead, we generated random problems on 25×25 grids, with 125 (20%) obstacles and 100 agents, on 100×100 grids with 400 agents and 2000 (20%) obstacles, and on 200×200 grids with 400 agents and 4000 (10%) obstacles.

5.2 Experimental settings

The experiments were performed with elementary actions played simultaneously, with 4-connectivity, and the global elapsed time as target of optimization. $maxpl$ is the heuristic value using the maximum over the individual path lengths. $C = 1$. $ls = 500$. For each problem and each algorithm, we give the number of time steps used (nts) and the number of elementary actions used (nea) to solve the problem. Furthermore, we mention the computing time spent, the nesting level (lv) and the number of iterations for each level (il). We used a 3.2 Ghz computer with 6 Gb to perform the experiments. We compare the results of NMCFS to those of TASS [6], P&S [10, 9] and TOMPP [21].

Table 2: Results on Khorshid's congestion problems.

	nts		nea					
	M	Op	TS	M	Op	t	lv	il
520	6	6	34	17	17	0.1s	2	5
519	8	8	30	18	12	0.03s	2	5
518	10	10	58	26	≤ 26	2m	2	30
517	13	13	170	31	≤ 35	3m	2	30
515	15	≤ 15	459	71	≤ 71	30m	3	40
516	19	≤ 19	234	86	≤ 86	72h	3	30

5.3 Results

Congestion problems Table 2 shows the results on Khorshid's congestion problems. The results in terms of nts are new. For the first four problems,

we found the optimal value (Op) with paper and pen, and NMCFS found the optimal values for these problems. For problems 516 and 515, we do not know if the solutions are optimal or not. In terms of nea, NMCFS (M) outperforms TASS (TS). The harder the problem, the larger the difference. TASS solves all the problems in less than one second. NMCFS solved the easy problems in less than one second, but used three days for problem 516 to find out the value in the table. With less than 3 days of time, NMCFS was not able to solve the problems of [6] which are more complex than problems 515 and 516.

Table 3: Results on Luna's congestion problems.

	nts		nea					
	M	Op	P&S	M	Op	t	lv	il
Rot.	1	1	18	16	16	0.01s	1	1
Tree	6	6	18	12	12	0.01s	1	5
Str.	8	8	26	20	≤ 20	0.02s	1	5
Corn.	8	8	50	32	≤ 32	1s	2	5
Conn.	16	≤ 16	86	70	≤ 70	1m	2	20
Tunn.	15	≤ 15	81	49	≤ 49	1h	3	30
Loop.	19	≤ 17	350	106	≤ 98	12h	2	200

Table 3 shows the results on Luna's congestion problems. Again, the results in terms of nts are new. For the problems Rotation, Tree, String, Corners and Loopchain, we found the Op value with paper and pen. NMCFS found the optimal value for Rotation, Tree, String, and Corners but not for Loopchain. For Tunnel and Connector, we do not know if the results are optimal or not. In terms of nea, NMCFS outperforms Push&Swap. For Loopchain, the difference is large. Push&Swap solves all the problems in a few seconds [10]. NMCFS solves the easy problems in less than one second, but used several hours for Loopchain.

N-puzzle problems Table 4 shows the results obtained by NMCFS on puzzle problems. For the 8-Puzzle, NMCFS found the optimal solution quickly. For the 25-puzzle, NMCFS is slightly sub-optimal because the optimal length is 7 [20]. For the other puzzles, NMCFS found solutions for which the optimality remains not known. For each problem, table 4 also gives the branching factor to show that NMCFS is not constrained by this feature.

Table 4: Results and branching factors (bf) on N-puzzle problems.

n	bf	nts	nea	t	lv	il
8	123	4	26	0.1s	1	10
9	27	6	38	5s	1	10
15	3815	7	84	20m	3	50
16	951	8	90	4h	3	50
24	$\approx 10^5$	7	141	8h	3	30
25	$\approx 3 \times 10^4$	8	120	30h	3	30

Many-agents-large-grid problems Table 5 shows the results achieved by NMCFS on large grids with many agents. na is the number of agents, $nobs$ the number of obstacles, pb the problem, and h the $maxpl$ value. On 25×25 gridsize problems with 20% of obstacles and 100 agents, the values are obtained in about

6 hours at level 2, and *nea* ranges in the interval 2350, 2570 with 2470 on average. For a 30×20 grid with 100 agents and 17% of obstacles, [10] mentions the average solution quality for WHCA* and P&S of 2700 and 2300 respectively. Table 5 also shows the results achieved by NMCFS on 100×100 grids with up to 400 agents and 2000 obstacles. With such level of congestion (20% of obstacles and 4% of agents), NMCFS at level 2 finds "good" solutions - not to say near-optimal. On 200×200 grids with 400 agents and 4000 obstacles, since the congestion is very low (10% of obstacles and 1% of agents), NMCFS finds the optimal solution despite the size of the grid. We are currently investigating problems with more agents (up to 2000) and larger grids (512×512 grids of [18]). To do so, NMCFS is not a bottleneck, instead we need to improve the pre-computation of distances with a hierarchical approach [2].

Table 5: Results on large grids with obstacles and many agents.

gridsize	na	nobs	pb	nts	h	nea	t	lv	il
8×8	11	15	10	12	12	63	0.1s	1	5
			11	11	11	48	0.1s	1	5
25×25	100	125	1	43	41	2351	6h	2	50
			2	43	43	2508	6h	2	50
			3	44	41	2460	6h	2	50
			4	44	40	2569	6h	2	50
100×100	100	1000	1	164	164	6879	1m	1	1
	200	2000	1	165	165	14203	5m	1	1
	400	2000	1	171	171	33286	15m	1	1
200×200	400	4000	1	344	344	55620	1h	1	1

6 Discussion

This section discusses some properties of MCFS: the anytime property, memory use, completeness, optimality and computation time. Let it be the number of iterations, na the number of agents, gs the gridsize, ls the length of the simulations, nlv the number of nesting levels, and nnd the number of nodes. Figure 2 illustrates the anytime property of MCFS.

Concerning memory use, most of the memory used by MCFS concerns nnd. nnd is linear in it, and linear in ls (which mainly depends on gs). The size of a node depends linearly on na. Memory used by MCFS is in $O(na \times it \times ls)$. Nesting MCFS permits to build smaller trees at each nesting level and to save memory. The memory used by NMCFS is in $O(na \times nlv \times it^{1/nlv} \times ls)$. In our experiments, the memory use was an obstacle for large grids for pre-processing the individual distances between cells and storing them. In practice, this last point is a current limitation of our approach.

To discuss the completeness of the approach, we have to prove that at least one simulation will reach the goal. For the bad cases, MCFS use pseudo-random simulations that are not biased: all the elementary actions are drawn with the uniform distribution of probability. The set of agents executes a random walk on a finite graph of the problem. If a problem is solvable, there is a sequence linking the starting position to the goal, and given a sufficiently large number of timesteps, a random walk on this graph encounters the goal position. However, if

the number of timesteps is finite, which is the case in practice, then some random walks do not encounter the goal. If the number of timesteps is large enough, a random walk reaches the goal with a probability $p > 0$. With a sufficiently large number of random walks, the goal will be reached at least once. Provided we set up ls and it at sufficiently large values, a least one simulation succeeds. Therefore, MCFS, with ls and it sufficiently large, is complete on the set of solvable CPF problems whose graph size is less than a given threshold.

The computation time is linear in it. One iteration time is mainly linear in ls. At each timestep, the time to perform the choice of the joint action and to process the effects of the joint action is at least linear in na. When collisions have to be managed, the time is longer. Therefore, in the good case only, the computation time of MCFS is in $O(it \times ls \times na)$.

We have no proof of optimality, but we observed near-optimality in practice. Asymptotically, we believe that all nodes will be visited infinitely, and the best sequence found. The problem is that MCFS tackles the problem by lowering its current upper bound progressively without using any measure of near-optimality. The relevant question is which value of it garantees MCFS to find the optimal solution.

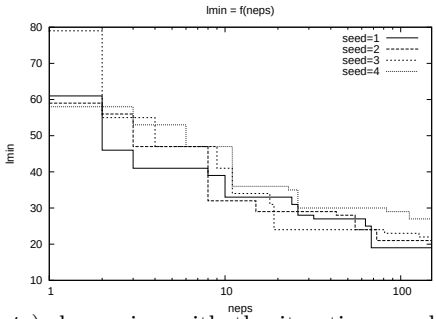

Fig. 2: *lmin* (i.e. *nts*) decreasing with the iteration number (*neps*) on four runs of MCFS on the loopchain problem at level 2.

7 Conclusion

In this paper, we have described both MCFS, a new algorithm inspired from MCTS, that solves CPF problems, and NMCFS its nested version. MCFS is one of the first approaches that deals with CPF problems by optimizing time. NMCFS near-optimally solves the 16-puzzle and the 25-puzzle, but it is still surpassed by TOMPP. For congestion problems, NMCFS outperforms TASS and P&S. For large problems with many agents, large grids, and obstacles, our approach gives results with up to 400 agents and 200×200 gridsize. The cost of obtaining such results is computation time. MCFS is anytime and provides approximate solutions in limited time. Conversely to MCTS and A*, MCFS is not constrained by the huge branching factor of CPF problems. With sufficient time and memory, MCFS is complete. In finite time and with a finite memory, MCFS is not complete, and near-optimal only.

The current work can be investigated further in several directions. First, we want to handle benchmarks [18]. Secondly, studying the speed of convergence to optimality is an important perspective. Thirdly, we aim at investigating the effects of nesting levels. Fourthly, assessing MCFS on other planning problems would be informative.

References

1. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multi-armed Bandit Problem. *Machine Learning*, 47(2-3):235–256, 2002.
2. A. Botea, M. Muller, and J. Schaeffer. Near Optimal Hierarchical Path-Finding. *J. of Game Dev.*, 1(1):7–28, 2004.
3. C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte-Carlo Tree Search Methods. *IEEE TCIAIG*, 4(1):1–43, 2012.
4. T. Cazenave. Nested Monte-Carlo Search. In *IJCAI*, 2011.
5. R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games*, volume 4630 of *LNCS*, pages 72–83, 2006.
6. M.M. Khorshid. Solving Multi-agent Pathfinding Problems in Polynomial Time using Tree Decomposition. Master's thesis, University of Alberta, 2011.
7. M.M. Khorshid, R.C. Holte, and N.R. Sturtevant. A Polynomial-Time Algorithm for Non-Optimal Multi-Agent Pathfinding. In *SoCS*, pages 76–83, 2011.
8. L. Kocsis and C. Szepesvari. Bandit-based Monte-Carlo Planning. In *ECML*, pages 282–293, 2006.
9. R. Luna and K.E. Bekris. Efficient and complete centralized multi-robot path planning. In *IROS*, 2011.
10. R. Luna and K.E. Bekris. Push and Swap: Fast Cooperative Path-Finding with Completeness Guarantees. In *IJCAI*, pages 294–300, 2011.
11. D. Ratner and M. Warmuth. Finding a shortest solution for the NxN-extension of the 15-puzzle is intractable. *Journal of Symbolic Computations*, 10:111–137, 1990.
12. C. Rosin. Nested Rollout Policy Adaptation for Monte Carlo-Tree Search. In *IJCAI*, pages 649–654, 2011.
13. S. Gelly and D. Silver. Achieving master level play in 9x9 computer go. In *AAAI*, pages 1537–1540, 2008.
14. G. Sharon, R. Stern, M. Goldenberg, and A. Felner. The Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding. In *IJCAI*, pages 662–667, 2011.
15. D. Silver. Cooperative Pathfinding. *AI Programming Wisdom*, 2006.
16. T.S. Standley. Finding Optimal Solutions to Cooperative Pathfinding Problems. In *AAAI*, 2010.
17. T.S. Standley and R. Korf. Complete Algorithms for Cooperative Pathfinding Problems. In *IJCAI*, pages 668–673, 2011.
18. N. Sturtevant. Benchmarks for Grid-Based Pathfinding. *IEEE TCIAIG*, 4(2):144 – 148, 2012.
19. K.-H. C. Wang and A. Botea. MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *JAIR*, 42:55–90, 2011.
20. J. Yu and S. LaValle. Planning Optimal Paths for Multiple Agents on Graphs. arXiv:1204.3830, 2012.
21. J. Yu and S. LaValle. Time Optimal Multi-agent Path Planning on Graphs. In *WoMP*, 2012.