Finding the Needle in the Haystack with Heuristically Guided Swarm Tree Search

Stefan Edelkamp[1], Peter Kissmann[2],
Damian Sulewski[1], Hartmut Messerschmidt[1]

[1]Technologie-Zentrum Informatik, Universität Bremen
[2]Fakultät für Informatik, TU Dortmund

Abstract. In this paper we consider the search in large state spaces with high branching factors and an objective function to be maximized. Our method portfolio, which we refer to as heuristically guided swarm tree search, is randomized, as it consists of several Monte-Carlo runs, and guided, as it relies on fitness selection. We apply different search enhancement such as UCT, look-aheads, multiple runs, symmetry detection and parallel search to increase coverage and solution quality. Theoretically, we show that UCT, which trades exploration for exploitation, can be more successful on several runs than on only one. We look at two case studies. For the *Same Game* we devise efficient node evaluation functions and tabu color lists. For *Morpion Solitaire* the graph to be searched is reduced to a tree. We also adapt the search to the graphics processing unit.

1 Introduction

Morpion Solitaire (see Figure 1) is a pen-and-paper longest path state space problem played on an infinite grid with some set of marked intersections taken as the initial state. In each move k − 1 intersections are covered and a new one is produced by placing a (horizontal, vertical, or diagonal) line having $k - 1$ edges. (The usual setting is a Greek cross with 36 marked intersections and $k = 5$.) Edges of every two lines must not overlap. In the disjoint model two lines on the same alignment must not touch each other, in the touching model they can.

Same Game[1] (see Figure 2) is an interactive computer puzzle played on an n x m grid covered with nm balls in k colors. (Usually, $n = m = 15$ and $k = 5$). Balls can be removed, if they form a connected group of $l > 1$ elements. The reward of the move is $(l-2)^2$. If a group of balls is removed, the ones on top of these fall down. If a column becomes empty, those to the right move to the left, so that all non-empty columns are aligned. Clearing the complete board yields a bonus reward of 1,000 points. The objective is to maximize the total reward. Both optimization

[1] See http://de.wikipedia.org/wiki/SameGame

problems are known to be NP-hard (Kendal et al. 2008, p. 31-34; Demaine et al. 2006, p. 439-453).

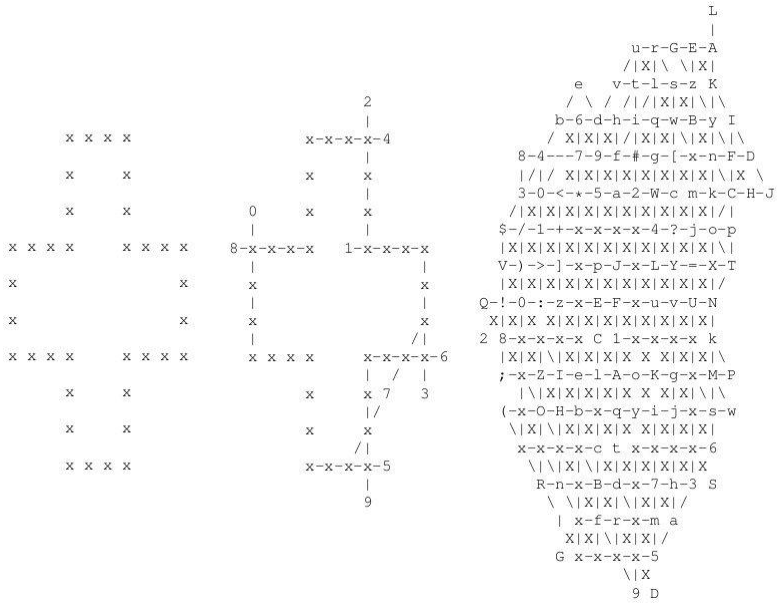

Figure 1: *Morpion* Solitaire: initial, intermediate and final position.

In order to find optimized solutions to these two games, we sped-up search by using what we call *Heuristically Guided Swarm Tree Search* integrating search heuristics, duplicate and symmetry detection, tabu and look-ahead strategies as well as single root and set-based UCT (short for upper confidence bounds applied to trees) (Kocsis and Szepesvari 2006, p. 282-293). We exploited parallel hardware as available in multi-core CPUs and graphics cards on current personal computer systems.

The paper is structured as follows. First, we introduce to the essence of guided runs and further search enhancements, including UCT. We look at different search heuristics and analyze the possible impact of having several independent runs. Next, we present a simple implementation of Parallel UCT along with some pseudo code. This parallelization is extended to work on sets, which are seeded with exploration results obtained on the GPU. Finally, we review related work, show experimental results, and conclude.

2 Heuristically Guided Swarm Tree Search

In random depth-first trials, also known as Monte-Carlo runs, successors are drawn at random. The game is played to the end, and the achieved result is record-

ed. If it is better than the current high-score, both the score and the stack contents are stored for keeping track of the best solution found so far.

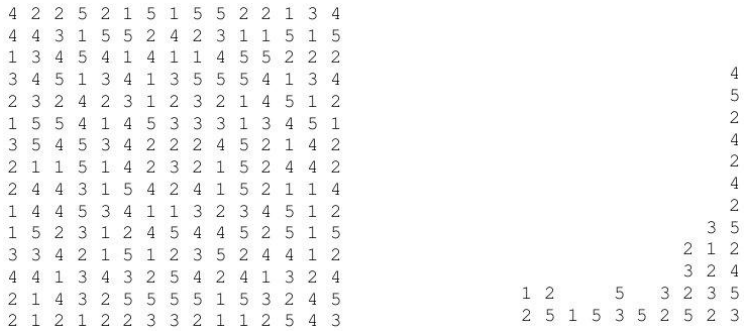

Figure 2: Same Game: initial and final positions (not yielding final reward of 1,000).

Algorithm 1 Selecting the next candidate using guided swarm tree search.

```
next(Succs) {
  htotal = 0;
  for (i = 0; i < succs; i++) htotal += Succs[i].h;
  random = rand[0..htotal - 1]; index = -1; ntotal = 0;
  while (ntotal <= random) {index++; ntotal += Succs[index].h;}
  return Succs[index];}
```

In uninformed swarm tree search, the probability of a successor to be selected is 1 divided by the number of successors, yielding a uniform distribution. Hence, at every sampled state, we determine the number of successors *succs* and a random number in {0,...,*succs* − 1}. Unfortunately, in state spaces with large branching factors, uniformed random runs often end up with inferior solution paths, so that improving them over time requires large amounts of time and space.

When problem-specific knowledge is encoded into the random sampling process, much better solutions can be obtained. Such knowledge can be thought of as the result of an evaluation (a.k.a. fitness or heuristic) function h from states to some positive numbers. In Heuristically Guided Swarm Tree Search we, therefore, apply non-uniform sampling (see Alg. 1), which is best thought of as a fitness selection according to the relative strength of the successors' evaluation.

Algorithm 2 Generating and evaluating successors in the Same Game.

```
hull(i, j, c, r) {
 if (i < 0 || i > n - 1 || j < 0 || j > m - 1) return 0;
 if (board[i][j] != c || vis[i][j] == r) return 0;
 vis[i][j] = r;
 return 1 + hull(i+1,j,c,r) + hull(i,j+1,c,r) +
            hull(i-,j,c,r) + hull(i,j-1,c,r); }
void populate(i, j, val, iter) {
  if (i < 0 || i > n - 1 || j < 0 || j > m - 1) return;
  if (vis[i][j] != r) return; vis[i][j] = 0;
  if (count[i][j] == 0) count[i][j] = v;
  pop(i+1,j,v,r);pop(i,j+1,v,r);pop(i-1,j,v,r);pop(i,j-1,v,r);}
computesuccs() {
  succs = 0; vis = value = count = 0;
  for (j = 0; j < n; j++) for (k = 0; k < m; k++)
      if (board[j][k] != '0') {
        if (count[j][k] == 0) {
          count[j][k]=hull(j,k,board[j][k],succs);
          pop(j,k,count[j][k],succs); }
      succx[succs] = j; succy[succs++] = k; }
  for (j = 0; j < n; j++)
      for (k = 0; k < m; k++) value[j][k] = score[count[j][k]];
  return succs; }
```

3 Search Heuristics

Evaluation Functions In both games, we combine the computation of successors with their evaluation. In *Morpion Solitaire* we use a simple function that prefers dense line arrangements and touching lines. As rewards in the *Same Game* are immediate, we apply an evaluation function that takes into account the current and the remaining score (see Alg. 2). The algorithm to compute all successors obviously requires time $O(nm)$, and has been fine-tuned by using a stack instead of a recursive implementation.

Duplicate and Symmetry Detection In BFS and UCT we incorporated hashing for duplicate detection, avoiding redundant work at a tree node that has already been explored. Simple hash functions map the core of the state. For *Morpion Solitaire* the bounding box can be included. Given that we are interested in some good solutions, for parallel UCT we use a bit-state hash table without any collision resolution

strategy. Symmetrical states, which can be detected by reflecting the board horizontally, vertically and diagonally, are also omitted from the UCT/BFS tree.

Tabu Colors and Lookahead For the *Same Game* we perform a one step lookahead once every fixed number of iterations. If larger groups are eliminated, the changes in the game due to gravity and alignment are considerable. Not performing the actual change in the successors looses too much information.

Even though our state evaluation procedure and look-ahead procedure are efficient, the invocation of the look-ahead after each step led to a program with small progress. The mixture of information gathering every *l th* iteration seemed was a key to success. (We chose *l* to be between 5 and 20.)

In the *Same Game* we additionally implement the strategy proposed in (Schadd et al. 2008, p. 1-12) to prevent the tabu color to be selected in order to build large groups as a hard constraint, so that successors on unforced selection get a fitness value and selection probability 0. In most cases, the tabu color with the largest number of balls gave the best results, while in some cases the choice of other colors was more effective.

UCT (Kocsis and Szepesvari 2006, p. 282-293) is a value-based reinforcement learning algorithm. The action value function is approximated by a table, containing a subset of all state-action pairs. A distinct value is estimated for each state and action in the tree by Monte-Carlo simulation. The policy used by UCT balances exploration with exploitation according to the formula $v + C * \sqrt{(\log(N)/n_i)}$, where v is the value stored at each node, C is a constant to be adapted to the problem at hand, n_i the visits to the actual node s to be evaluated, and n is the number of visits to the parent of s.

UCT is the algorithm of choice in playing two-player games with hardly accessible evaluation functions such as *Go*[2] , and in general game playing (Genesereth et al. 2005, p. 62-72). It also applies to single-agent state space maximization problems with large branching factors (Schadd et al. 2008, p. 1-12) by propagating the maximum leaf value to the root, like in our implementation. UCT has two phases. In the beginning of each run it selects actions according to knowledge contained within the UCT tree. Once it leaves the scope of this search tree it has no knowledge and behaves randomly. Thus, each state in the tree estimates its value by Monte-Carlo simulation. As more information propagates up the tree, the policy improves, and the Monte-Carlo estimates are based on more accurate returns.

Multiple Runs For *Morpion Solitaire* we experimented with different number of runs in UCT. Let us consider $\sqrt{\log(n+k)/n}$ and $\sqrt{\log(n+k)/k}$ for $k = 1$:

$$\sqrt{\log(2)/1} = 0.8325 \qquad \sqrt{\log(2)/1} = 0.8325$$

$$\sqrt{\log(3)/2} = 0.7411 \qquad \sqrt{\log(1)/1} = 1.0481$$

[2] See, e.g., http://www.lri.fr/~teytaud/mogo.html

…	…
$\sqrt{\log(101)/100} = 0.2148$	$\sqrt{\log(101)/1} = 2.1482$
$\sqrt{\log(10001)/10000} = 0.0303$	$\sqrt{\log(10001)/1} = 3.0348$
$\sqrt{\log(000001)/1000000} = 0.0037$	$\sqrt{\log(000001)/1} = 3.7169$

Let us assume an artificial example with a complete binary tree of depth 30. At all leaves of the left subtree of the root we find a value 10, and in the right subtree all leaves have value 0, except one with the optimum value 100. One random run finds the maximum with probability $1/2^{30}$. Subsequently, we find the sole optimum after an expected number of 2^{30} runs.

UCT, however, will first make two runs, which establish the value 10 on the left branch with probability 1 and value 0 on the right branch with probability $1/2^{29}$. Afterwards, the optimum can no longer be achieved. This example indicates that critical decisions at the root are rarely withdrawn, so that drawing more random samples at a node can indeed be advantageous.

4 Simple Parallel UCT

Parallelizing UCT to multiple cores is a hot research topic. Different approaches have been proposed (Enzenberger and Müller 2009; Cezenave and Jouandeau 2009, p. 1-6), all with their individual pros and cons. We propose a simple novel Parallel UCT variant (using pthreads), that achieves a close-to-optimal speed on multi-core machines. It only uses a single lock on the variable for increasing the number of nodes in the UCT tree. The main routine for calling different threads is displayed in Alg. 3. It also initializes the search to the start state, allocates space for the node array node, the hash table. The procedure called by each thread is an infinite loop, initializing the search and the UCT sub-procedures.

Algorithm 3 Invocation of the threads in Parallel UCT.

```
uctmontecarlo(thread) {
  while (1) {
    init(thread);
    index[thread] = uct(depth[thread], thread);
    if (index[thread] > 0) montecarlo(thread);}
main() {
  init(); highscore = 0;
  for (i = 0; i < THREADS; i++) // spawn light-weight processes
    thread_create(threads[i], uctmontecarlo);}
```

Algorithm 4 Parallel UCT algorithm.

```
uct(depth, thread) {
 j = 0; depth = 0; expandleaf[thread] = 1;
 while (node[j].leaf == 0) {
  maxv = 0; maxs = -1; succs = 0;
  for (i = 0; i < node[j].numberofsuccs; i++) {
   s = node[j].successors[i];
   if (node[s].count == 0) {
    maxv = infinity; maxs = s; expandleaf[thread] = 0;
    Succs[thread][succs++] = s;}
   else {
    v=node[s].value+C*sqrt(log(node[j].count)/node[s].count);
    if (node[s].value == 0) v = 0;
    else if (v > maxv) {maxv = v; maxs = s;}}}
    if (maxs == -1) {node[j].value = 0; return -1;}
    if (succs > 0) {maxs = next(Succs[thread],succs);}
    node[j].count++;
    j=maxs; stack[thread][depth] = node[maxs];
    doMove(node[maxs],thread); depth++;}
  node[j].count++;
  if (expandleaf[thread]) {
    if (search(sol[thread])) {node[j].value = 0; return -1;}
    else {expandnode(j,thread); insert(sol[thread]);}}
  return j;}
```

The growth of the search tree in UCT is shown in Alg. 4 and the according Monte-Carlo search is displayed in Alg. 5.

At a leaf node the UCT tree is enlarged by generating the successors of the encountered leaf node. Note that once all successors have been expanded, the selection of successors in the top-down phase of the algorithm is deterministic. If there are still successors left, one is chosen randomly. If a node has been fully explored, its value is set to 0 and is omitted from further processing.

For parallelization, the algorithm increases the number of node visits when walking down the tree, applying the UCT formula at every successor node. This is in contrast to the usual sequential implementation where node counts are updated bottom-up. The advantage for an early increase is that different threads likely lead to different leaves.

Algorithm 5 Parallel Monte-Carlo search.

```
void montecarlo(thread) {
  succs = 0;
  while (1) {
    Succs = expand();
    for (i = 0; i < |Succs|; i++)
    if(canMove(Succs[i],thread))succ[thread][succs++]=Succs[i];
    if (succs == 0) break;
    r = next(Succs);
    stack[thread][depth[thread]] = succ[thread][r];
    doMove(thread); depth[thread]++;}
  if (depth[thread] > highscore) highscore = depth[thread];
  j = index[thread];
  while (j != -1) {
    if(node[j].value<depth[thread])node[j].value=depth[thread];
    j = node[j].parent;}}
```

At each node in the Monte-Carlo search, we always generate all successors. When a run is finished, we check whether a new highscore has been obtained, so that we can backup the according stack. At the end of the procedure, we store the obtained value at the search tree leaf where the Monte-Carlo run has started and propagate the outcome bottom-up to the root of the UCT search tree, so that the root value always reflects the optimal value found in its leaves. We checked that race conditions leading to inconsistencies (not harming the validity of UCT) at inner nodes did not appear.

5 Set-based Parallelization

An alternative to the parallelization given above is to seed the search with a larger set of root nodes and to use UCT to explore the k best of them for a fixed number of nodes.

To achieve this, we use an implementation of a priority queue Q based on *weak-heaps* (Edelkamp and Wegener 2000) (for worst-case efficient priority queues, see (Edelkamp 2009)). The queue contains the indices of the nodes in the UCT tree along with their corresponding number of expansions, the maximal depth reached, and the resulting UCT value. The queue is organized according to these UCT values (higher values are better).

To come up with a set of states to insert into the queue, we initially all perform a complete BFS up to a certain layer. For each state in this layer, we create one node in the UCT tree (and one corresponding element in the queue).
The removal of the maximal element is done by swapping it with the last one in the queue, decrementing the queue's size and re-establishing the correct order. This can be done in logarithmic worst-case time. We repeat this removal of the maximal element, until we come up with the k best ones.

Typically, we assume k/n being very small, with n being the number of threads we use. For the parallelization, each thread takes the first unused element and performs a normal UCT run starting at this element. The result will be re-inserted into the queue. For this, at first only the maximal depth and the number of expansions are updated, the corresponding element is swapped with the one at position size and the size of the queue is incremented. At this time, the order of the queue is not correct. As we need to update the UCT values of all elements in the queue when the total number of expansions is changed, we delay this step until all the k nodes are expanded. Then, we update all UCT values and need to re-organize Q. In total, extracting the k-best elements requires at most $|Q| - 1 + k \cdot \lceil log(|Q| + 1) \rceil$ comparisons.

6 GPU-based State Space Exploration

Given that the number of cores and the processing power on the graphics card in form of a GPU is much larger than on the CPU, we used general-purpose GPU programming to generate the large state space efficiently. The computational model is very different from the one on the CPU. Programming the GPU requires a special compiler, which translates the code to native GPU instructions. The GPU architecture executes the same instructions running on all processors and supports different layers for memory access. It allows concurrent reads to one memory cell, but forbids simultaneous writes.

Progress in state space search on the GPU (Edelkampand Sulewski 2009) shows that using the GPU, speed-ups achieved in exploring permutation games can be up to 30. Recent findings (Edelkamp et al. 2009) suggest that significant speed-ups on the GPU are available also for the complete exploration of other combinatorial games. For our case of solving games with a large branching behavior, we aim at both fast successor generation and large-scale search. So far, we have applied GPU search only to *Morpion Solitaire*, but most design options are available for *Same Game* as well.

Based on different time and space considerations, a state is represented by its stack. Two states are the same if their sorted stacks match. Bounding boxes for each stack can be maintained and state symmetries and hash functions can also easily be applied to the entire stack to eliminate duplicates efficiently.

In the BFS parallelization the GPU extends all stacks in parallel one step at a time. Given the small memory requirements of stacks especially in low search depth, millions of stacks can be extended in parallel. As the GPU has a memory limitation and BFS levels naturally grow rapidly, the search has been externalized to store data on the hard disk. Duplicate detection is delayed and performed similar to the setting in I/O efficient model checking proposed in (Barnat et al. 2007, p. 281-293), where the generated state set stored in RAM is checked against the explored one on disk while avoiding additional external sorting efforts.

This way, we constructed a complete search engine, combining breadth-first and depth-first search, eventually exploring the entire search space. Facing the size of the problems, however, even with large amounts of time and space, the problem of fully exploring *Morpion Solitaire* practically remains unsolved. Therefore, we used parallel search on the GPU mainly to give rise to good seeds for the UCT mechanisms in form of large breadth-first layers in set-based UCT.

7 Related Work

The best known solution for *Morpion Solitaire* in the touching model has 170 moves. It has been found by Charles-Henri Bruneau and was published by Pierre Berloquin in April 1976 in *Science & Vie*.[3] For the disjoint version the current record has been found with the help of a computer.[4] In 2009 Tristan Cazenave documented a solution to the disjoint problem with 80 moves using Nested Monte-Carlo Search (Cazenave 2009, p.456-461; Cazenave and Jouandeau 2009, p. 1-6).

For the Same Game, Takes and Kosters (2009, p. 249-256) use Monte-Carlo search to compute a solution of 76,764, improving some previous result of 73,998.[5] Tristan Cazenave reports solutions on Same Game as well (Cezenave 2009, p. 456-461). His record of 77,934 points was obtained with Nested Monte-Carlo Search. The overall best solution we could find in the Internet are the 82,604 points submitted by a player called *spurious ai*[6] .

There are a number of different parallelizations of Monte-Carlo Search/UCT. We highlight the ones, which we found most related.

- According to a personal conversation with one of the authors of the general game player (Genesereth et al. 2005, p. 62-72) MALIGNE, they use a large number of processors, without any communication between them.

[3] The solution we coded is available at http://euler.free.fr/morpion.htm and is dated 1982 by J.B.Bontè.

[4] According to http://www.morpionsolitaire.com the currently best computer solution for the touching version reported by Tristan Cazenave has 144 moves.

[5] See http://www.js-games.de/eng/games/samegame/lx/play.

[6] See http://www.js-games.de/eng/highscores/samegame/lx.

To prevent each processor from doing the same, they start one instance of their player on each and provide it with a different UCT value.

- The approach in our general game playing agent GAMER is similar to the single-run parallelization presented in (Cezenave and Jouandeau 2007). We start one master UCT process and several slave processes. Each slave develops its own UCT tree. In contrast to MALIGNE, all these processes have the same UCT value. To prevent all the processes from creating the same trees, we use different seeds for the random number generators. The master manages only the root of a UCT tree (representing the current state). After each UCT run, each process informs the master process of the chosen move and the achieved result and the master updates its root correspondingly. The move that has achieved the best average result during all the runs is performed.
- Tristan Cezenave has ported Nested Monte-Carlo Search (Cezenave 2009, p.456-461; Cezenave and Jouandeau 2009, p.1-6) to a shared memory multiple core scenario. Recall that Nested Monte-Carlo search is a limited-depth BFS with Monte-Carlo runs invoked at the leaf and best values propagated to the root. In contrast to UCT the search tree does not adapt to the solutions found. Besides parallelizing Monte-Carlo search but not UCT, the parallelization mechanisms are different to the ones that we have considered.
- In his *epsilon-trick*[7], Lukasz Lew forces the threads to use an edge that has been selected by the UCT formula in one node more than once. This is implemented by an additional counter that releases the edge preference once exceeding the threshold. The idea has similarities to our approach of enforcing multiple runs.
- In the context of their Go-player, Müller and Enzenberger (2009) provide lock-free implementation of UCT based on a processor dependent implementation trick that has been used in chess beforehand. The trick involves inline assembler and relies on the order of instruction execution in Intel's x86 architecture.

[7] http://computer-go.org/pipermail/computer-go/2007-January/008057.html

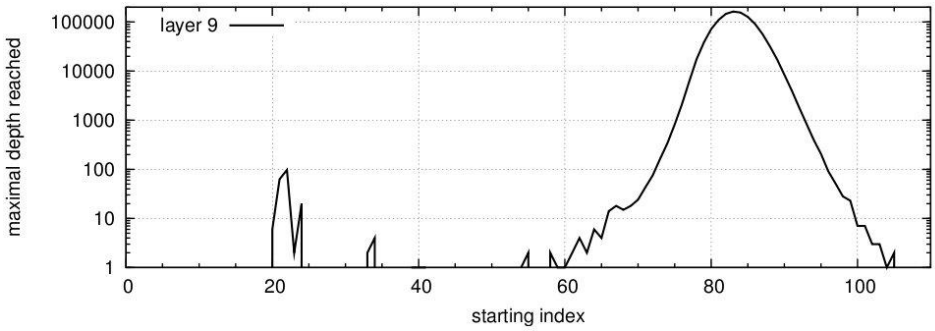

Figure 3: States distribution, BFS-layer 9 after 2,000 initial Monte-Carlo evaluations

8 Experiments

We ran experiments using an Intel i7 processor with 4 hyper-threaded cores, 12 GB of RAM, and two Nvidia GTX 280 graphics cards. The speed-ups obtained on the CPU are considerable (we recorded 780% – 800% CPU usage).

Morpion Solitaire We first generated the state space with BFS on the GPU, which was truncated, when the GPU space was exhausted. Then each state in BFS-layer 9 was evaluated by running 2,000 Monte-Carlo simulations. The obtained distribution of maximal solution depths is shown in Fig. 3.

The advantage of the GPU search is that it is complete, i.e., it eventually will explore the entire search space. The current best solution on the GPU is 116. Our current best solution length, obtained with parallel UCT (15 threads and 100 runs) is 128; one solution is displayed in Fig 1. Seeded with the first 15 steps of Bonté's solution, the best solution we could obtain has a length of 130=115 + 15. Seeded with 111 steps, we could reproduce a solution of 170 moves.

Same Game Our single-threaded version already beats the reported maximum of 82,604 found in the Internet (see Table 1). Hence, for parallelization, we only applied an *embarrassing parallel* approach, in which the program is invoked with different UCT values and tabu-colors. For a fixed UCT value and sufficient RAM available we subsequently obtain a 5-fold speed-up.

9 Conclusion

We considered advances to random search procedures like UCT yielding some new strategy that we refer to as *Heuristically Guided Swarm Tree Search*. We applied a flexible *random choice model* based on the fitness of the successors. For *Morpion Solitaire* the results are close and in the *Same Game* the results are state-of-the-art.

Table 1: Results for the *Same Game*.

Game	Points	Iteration	Tabu Color	Game	Points	Iteration	Tabu Color
0	2,561	188,633	1	10	2,796	130,496	2
1	4,995	319,991	Max	11	3,710	425,892	Max
2	2,858	1,342,750	Max	12	3,271	259,438	3
3	4,051	1,289,432	Max	13	2,432	632,190	Max
4	4,633	1,264,406	Max	14	3,877	240,061	2
5	5,003	302,745	Max	15	6,074	787,383	Max
6	2,717	534,920	Max	16	5,166	905,760	Max
7	4,622	28,433	Max	17	6,044	1,126,290	Max
8	6,086	688,276	max	18	5,019	22,732	Max
9	3,628	767,966	3	19	5,175	477,067	Max
				Sum	84,718		

References

Barnat J, Brim L, and Simecek P (2007) I/O efficient accepting cycle detection. In CAV.

Cazenave T (2009) Nested monte-carlo search. In IJCAI.

Cazenave T and Jouandeau N (2007) On the parallelization of uct. In Computer Games Workshop (CGW).

Cazenave T and Jouandeau N (2009) Parallel nested Monte-Carlo search. In IPDPS.

Demaine E, Demaine M, Langerman A and Langerman S (2006) Morpion solitaire. Theoretical Computer Science, 39(3).

Edelkamp S (2009) Rank-relaxed weak queues: Faster than pairing and Fibonacci heaps? Technical Report 54, TZI Universität Bremen.

Edelkamp S, Messerschmidt H, Sulewski D and Yücel C (2009) Solving games in parallel with linear-time perfect hash functions. Technical Report 52, TZI Universität Bremen.

Edelkamp S and Sulewski D (2009) Parallel state space search on the gpu. In Symposium on Combinatorial Search.

Edelkamp S and Wegener I (2000) On the performance of weak-heapsort. In STACS.

Enzenberger M and Müller M (2009) A lock-free multithreaded Monte-Carlo tree search algorithm. In Advances in Computer Games (ACG).

Genesereth M, Love N and Pell B (2005) General game playing: Overview of the AAAI competition. AI Magazine, 26(2).

Kendall G, Parker A and Spoerer K (2008) A survey of NP-complete puzzles. International Computer Games Association Journal (ICGA).

Kocsis L and Szepesvari C (2006) Bandit based Monte-Carlo planning. In Machine Learning: ECML 2006.

Schadd MPD, Winands MHM , van den Herik HJ, Chaslot G and Uiterwijk JWHM (2008) Single-player monte-carlo tree search. In Computers and Games.

Takes FW and Kosters WA (2009) Solving samegame and its chessboard variant. In T. Calders, K. Tuyls, and M. Pechenizkiy, editors, 21th Benelux Conference on Artificial Intelligence (BNAIC 2009).