STAIRS 2014
U. Endriss and J. Leite (Eds.)

doi:10.3233/978-1-61499-421-3-141

A! – A Cooperative Heuristic Search Algorithm

Antti HALME [1]
Aalto University, Finland

Abstract. We propose a new parallel search algorithm – A! – based on cooperating A* search agents, concurrency and a secondary tiebreaking heuristic. The search agents in A! share information asynchronously and trade some of their independence for additional search focus and a more global view of the search task. A! is inherently nondeterministic due to the implicit randomness of instruction scheduling, but given a consistent primary heuristic, it still finds optimal solutions for the single-source shortest path problem (SSSP). A! combines into a single cooperative search algorithm the breadth available in parallel execution and the depth-first orientation of both locally and globally informed search.

We experimentally show that A! outperforms both vanilla A* and an explicitly randomized, noncooperative parallel A* variant. We present an empirical study on cooperation benefits and scalability in the classic 15-puzzle context. The results imply that cooperation and concurrency can successfully be harnessed in algorithm design, inviting further inquiry into algorithms of this kind.

Keywords. A*, heuristic search, parallel algorithm, cooperation, nondeterminism

1. Introduction

Search features in numerous real world applications from pathfinding to resource optimization. Application of heuristic information can improve searching performance by providing focus for search space exploration. While heuristics distinguish promising areas of the search space from those less likely to lead to progress, there still remains plenty of searching to do, as there typically exist multiple equally good directions to pursue.

For the single-source shortest path problem (SSSP) the standard heuristic search method is the A* algorithm [1]. A* follows a best-first strategy that considers both known distance from the start node and an estimate for the remaining distance to a goal node. As a graph-based algorithm, A* is fundamentally hard to parallelize: best search directions depend on overall search progress. Nonetheless, there has been some success in parallelizing best-first search both locally and in a distributed fashion, mostly through insightful search space partitioning and redundancy elimination [2–6].

In this work we explore parallel heuristic search that is based on cooperation and dependence rather than on work division and independent execution. We focus on the cooperation of multiple distinct worker components, algorithmic agents, executing con-

[1] Address: Antti Halme, Aalto University, Dept. of Information and Comp. Sci., PO Box 15400, FI-00076 Aalto, Finland; E-mail: antti.halme@aalto.fi; Web: http://users.ics.aalto.fi/ahalme/coop/ .

currently and communicating asynchronously. Our main hypothesis is that cooperating agents searching together can be more effective than agents searching in isolation. We test our hypothesis in a series of computational experiments and look for some general trends. This paper summarizes the research presented in full as a master's thesis [7].

The contribution of this paper is a new kind of cooperative heuristic search algorithm, a parallel variant of A*, dubbed A!. A! features cooperating search agents that share information and collectively maintain a secondary tiebreaking heuristic. Agents in A! explore the search space in an implicitly random fashion, directed by this dynamic ranking heuristic. Indeed, we view cooperation simply as another layer of heuristic focus, a depth-first orienting sense of global search progress.

We empirically evaluate A! by solving instances of the standard 15-puzzle benchmark problem. The performance of A! is compared with vanilla A* and a randomized non-cooperative parallel A* variant. We show that A! outperforms both methods. The implementation is not as such competitive against state-of-the-art parallel 15-puzzle solvers, but rather a first step in applying cooperation as a secondary heuristic.

We begin with a discussion on cooperative search in the next section and then describe the A! algorithm. Results from computational experiments on 15-puzzles are discussed in Section 4. Related work is discussed briefly before the conclusion in Section 5.

2. Constructing cooperative search

Kishimoto et al. [4] note that combining heuristic search and parallel processing is challenging because of three kinds of overheads. *Search overhead* occurs when the parallel version expands more nodes than the sequential one, typically as a result of non-disjoint search space division between search agents. *Synchronization overhead* refers to the idle time wasted at synchronization points, such as locks on shared data. Finally, *communication overhead* occurs whenever information is exchanged.

Parallel search is usually based on search space division, the goal being the removal of search overhead [4]. This renders synchronization and communication the bottlenecks. The parallelism in heuristic search algorithms is therefore typically very simple, with search agents mostly working independently. While this approach is common in general parallel computing as well, it does not fully leverage the parallel potential of modern multi-core shared-memory systems.

In this work we focus on removing the synchronization overhead and also give up precise communication patterns. Instead of maintaining a clear separation between search agents, we emphasize their close cooperation. Specifically, we make use of asynchronous message passing and nondeterministic instruction scheduling, and build a cooperation mechanism that is powered by the indeterminacy inherent in concurrency. We explore computation that is *implicitly random* as opposed to being explicitly randomized.

The idea in cooperative search is to direct and focus the search by sharing information among concurrently operating search workers [8, 9]. The goal is to search faster as a collective, with each agent utilizing the information they obtain. The hypothesis underlying this work is that cooperating agents outperform agents in isolation: search agents make good use of shared information and the overall effort benefits from cooperation.

Fundamentally, cooperation is communication: cooperating agents consume and contribute messages, whereas isolated agents keep to themselves. Parallel search is by

default concurrent when the cooperation between the agents is unconstrained and asynchronous. Instead of enforcing determinism and working against the natural disorder of concurrency, in this work we embrace it and try to use it to our advantage.

We wish to have a cooperation mechanism that enables information sharing among concurrently operating search workers and is both effective and lightweight enough for this extra effort to be worth it. We wish to augment heuristic search in an unobtrusive way with a basic cooperation mechanism and set the stage for concurrent phenomena.

Instead of a rigid master-slave approach, we want the overall global search to be managed collegially by several processes. We do away with superfluous synchronization and rather embrace asynchronous interaction. A* is point-initialized by nature, so having a portfolio of different approaches would make sense, but to study cooperation specifically, we stick to a uniform strategy. In the terminology of Crainic and Toulouse [8], our full cooperation policy can then be classified as $pC/C/SPSS$.

3. The A! algorithm

3.1. Overview

The A! (*a-bang*) algorithm is a parallel best-first heuristic search that employs asynchronously communicating software agents as concurrently cooperating search workers. Given a graph, a start node, a goal predicate and *two* heuristic functions – a primary and secondary one, denoted $h(u)$ and $\hat{h}(u, v)$, for all u and v in the graph – A! finds a single pair shortest path from the start node to a goal node.

A! consists of N agents that each run a distinct upgraded version of A* search. Each agent performs a heuristic search starting from the start node, but also participates in the cooperation effort: the agents share information about their progress with their fellow agents. An additional message broker entity can be used to streamline communications.

The primary heuristic is the one used in an A*-like graph search itself, while the secondary heuristic serves as a tiebreaker between equally good next-to-open candidate nodes. Vanilla A* simply maintains an estimated value priority queue, but A! workers aim to discern differences between nodes valued equally interesting in the queue.

This is the crux of A!: where vanilla A* always selects the head of the estimated value priority queue, A! agents choose among up to k most promising nodes of equal value based on information they individually acquire during the search.

As information is diffused asynchronously, A! workers have varying notions of search progress. When the priority queues and candidate sets have some differences, the agents diverge, but directed by the heuristics, they also meander close together again. A! generates diversity from concurrency, but also maintains cohesion through heuristics.

For the information to share, best encountered nodes are a simple, effective choice. This directly implies a *distance-to-best*-based secondary heuristic, where each worker is directed towards the areas of the search space that have been fruitful in the past. More elaborate information sharing and utilization schemes are well worth exploring in applications, but lie outside the main focus of this work.

Note that the degenerate case of a single A! agent is *not* necessarily vanilla A*. With only a single agent passing progress information to the secondary heuristic function, the search turns into a momentum-based eager search, where candidates close to recently

opened ones are ranked high among nodes that are equal with respect to the primary heuristic. This *momentum effect* is visible later on in the experiments.

Finally, A! retains the optimality of A* in the sense that the solution paths are the shortest possible, if the primary heuristic is consistent. The secondary heuristic only selects the order in which areas of the search space get explored. The primary and secondary heuristics can be the same distance measure, but this is not necessary in general.

3.2. Algorithm details

Next we present the entire A! algorithm as a collection of pseudocode snippets. We begin with Algorithm 1, which serves as the main body of the algorithm. In short, we simply launch a collection of search workers and wait for one of them to finish. We encapsulate the cooperation communication into a message broker entity that takes care of the communication scheme following a publish-subscribe pattern.

Given the problem instance, including the *two* heuristic functions, the algorithm returns a path from the start node to the found goal node, if one is reached. The path is derived from a path map of successor nodes by the first worker to find a goal. Solution discovery triggers main program termination and solution path retrieval via `getPath(..)`.

The workers run `A!Solver`, outlined in Algorithm 2, which has four parts inside a loop that is repeated until program termination. The first part, lines 6–7, is the node visit, where we check whether the current node is a goal based on the `isGoal` predicate. If it is, we derive the solution path and terminate, if not, we mark the node visited.

The second part, lines 8–14, is the A* expansion. We use the primary heuristic function to estimate remaining distances for the legal neighbors of the current state and update the data structures as we discover new nodes. The priority queue `openHeap` maintains the unopened node queue ordered by estimated total cost. The `pathMap` maps nodes to one another, establishing the successor relation used in solution path derivation.

The third part, lines 15–16, is a cursory peek into the up-to-date `openHeap`. In A! we examine some interesting nodes and select among them according to the secondary heuristic, whereas in vanilla A* we simply draw one node from the top. The peek is a bounded traversal of the priority queue, where we build a list – the `peekList` – of nodes with a cost equal to the top node. If there are no nodes left, the search terminates.

The final part, lines 17–18, contains the selection routine, which for A! is given as Algorithm 3. After one of the nodes has been selected – in one way or another – it is removed from `openHeap` and turned into `current`. Removing nodes is a relatively fast operation for some priority queue implementations, including the Fibonacci heap.

Algorithm 1 : **A!Search**

Require: $N > 0$, NODE *start*, PREDICATE *isGoal*, HEURISTIC h, $\hat{h}$
Ensure: *path* from *start* to nearest node satisfying *isGoal* is shortest possible
1: $mb \leftarrow MsgBroker()$
2: **for** $i = 0$ to N **do**
3: $\quad workers[i] \leftarrow A!Solver(mb.portOut, mb.portIn, s, isGoal, h, \hat{h})$
4: **end for**
5: **for each** *worker* in *workers* **in parallel** *worker.launch*() **end for**
6: **wait** for termination
7: **return** $path \leftarrow getPath(workers)$

Algorithm 2 : **A!Solver**

Require: PORT *portIn*, *portOut*, NODE *start*, PREDICATE *isGoal*, HEURISTIC h, $\hat{h}$

```
 1: openHeap ← FibonacciHeap⟨INTEGER, NODE⟩
 2: closedSet ← Set⟨NODE⟩
 3: pathMap ← Map⟨NODE, NODE⟩
 4: current ← start
 5: repeat
 6:   if isGoal(current) then terminate(current, start, pathMap) end if
 7:   closedSet.add(current)
 8:   for each n in current.getNeighbors() do
 9:     if closedSet.contains(n) then continue end if
10:     g ← current.g + dist(current, n)
11:     f ← g + h(n)
12:     improved ← openHeap.update(n, f)
13:     if improved then pathMap.update(n, current) end if
14:   end for
15:   peekList ← openHeap.getPeekList()
16:   if isEmpty(peekList) then terminate() end if
17:   current ← A!Select(peekList, portIn, portOut, h, ĥ)
18:   openHeap.remove(current)
19: until termination
```

The selection routine `A!Select` is the real core of the A! algorithm: it contains the cooperation functionality and the application of the secondary heuristic. In the first part, lines 1–2, the cooperation routine `asyncRecv` brings new information into the agent. The read is asynchronous and nonblocking in that if there is nothing to receive, the algorithm proceeds without any delay. The routine in Algorithm 3 gives a version with best-information being shared, but other schemes can naturally be constructed here.

The second part features the inclusion of the new information, lines 3–8, as encapsulated into the secondary heuristic function, $\hat{h}$. The `peekList` is sorted on $\hat{h}$ and the highest ranking node is selected. The third part, lines 9–12, mirrors the first part: new data is sent for others to process. The listing shows a small communication overhead optimization, where the agent only informs others, if it believes that it has made progress.

Alternative selection policies to `A!Select` include the random (A?) and vanilla (A*) selection routines. The former selects nodes at random from the peeklist, while the latter simply takes the head of the list. We proceed with an evaluation of these selection policies in a series of computational experiments.

4. Solving 15-puzzles with A!

In this section we describe experiments based on repeated executions of three versions of heuristic search, all based on a single implementation: vanilla A*, a randomized parallel variant A?, and the cooperative A!. We focus on two dimensions that are especially interesting from a cooperative search point of view: the overall benefit from cooperation and the extent to which the methods are scalable. We first describe the experimental setting and then present the computational results. More results can be found in [7].

Algorithm 3 : **A!Select**

Require: LIST *peekList*, PORT *portIn*, *portOut*, HEURISTIC h, $\hat{h}$
Ensure: *select* is the most promising node in *peekList* according to $\hat{h}$ on *best*

```
1: update, updateH ← asyncRecv(portIn)
2: if updateH < bestH then best, bestH ← update, updateH end if
3: select ← peekList.pop()
4: selectD ← ĥ(select, best)
5: for each node in peekList do
6:    d ← ĥ(node, best)
7:    if d < selectD then select, selectD ← node, d end if
8: end for
9: if h(select) < bestH then
10:   best, bestH ← select, selectH
11:   asyncSend(portOut, {best, bestH})
12: end if
13: return select
```

4.1. Experimental setting

The implementation used in the experiments is based on an A* for n-puzzles implementation by Brian Borowski[2] (BBI). The Java program features an A* solver as well as a version of *IDA**, of which the former was extended to a cooperative version in this work. All tests were executed on a cluster comprising a mixture of blade servers featuring 2.6GHz Opteron 2435, 2.67GHz Xeon X5650, and 2.8GHz Xeon E5 2680 v2 processors.

Two standard heuristics were used in the experiments: Manhattan distance with linear collisions (LC) [10], and a 6-6-3-partitioned static disjoint pattern database for the 15-puzzle (PDB) [11]. PDB was chosen as the primary heuristic and LC-to-best as the secondary heuristic: we use the database to focus on the right areas of the search space and break ties between equal valued nodes by LC-distance to the best observed node.

The test suite is a randomly generated collection of 15-puzzle instances. Instances were solved with BBI and grouped by the length of the optimal solution path, the shortest sequence of moves from the start position to the goal position. Randomly generated impossible instances – off-parity with respect to the target goal state – were discarded.

The instances within a given optimal length group proved to differ greatly in their difficulty: the average number of nodes examined during the search for instances in any group covers a range of several orders of magnitude. Further, as the methods under evaluation have stochastic and nondeterministic properties, runs on a given instance are themselves subject to nontrivial variation. The grouping is still justified, as with enough instance in each group, some general trends become apparent.

The experiments focused on *the number of nodes opened by the winning agent*, which was found to be a reasonably good metric. The goal in this work was not to build a competitive 15-puzzle solver, but to study cooperation effects in A!, so runtime is not considered here. Still, the winning agent measure reflects both the total work done by *all* agents – including repetition – and, to some extent, the total runtime as well. This follows from agents exploring states at roughly the same rate given a core per agent.

[2] http://www.brian-borowski.com/Software/Puzzle/

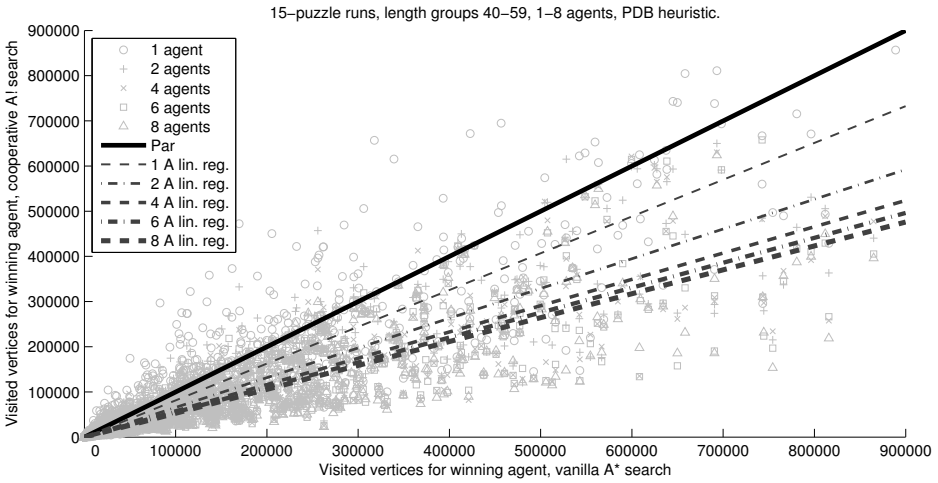

Figure 1. Relative performance of A* (x-axis) and A! (y-axis). The solid line is par, so data points below it represent instances for which A! performs better than A*. Agent count is evaluated in five batches – 1, 2, 4, 6, and 8 agents – with the respective trend lines showing how the configurations compare. With trend lines approaching $\frac{1}{2}$, the winning agents in multi-agent A! open roughly half the states of a vanilla A* run.

4.2. Computational results

The experiments give an overview of the performance of A! in comparison with vanilla A* and a non-cooperative random selection variant A?. We first show that A!, featuring cooperation and the secondary heuristic based ranking, overcomes both vanilla A* and A?. Second, we show that while the returns are diminishing, having more agents improves the overall performance of both A? and A!, but that A! clearly outperforms A?, making a preliminary case in favor of cooperation.

To see how A! fares against the competition, we set the algorithms to solve a suite of instances and observed how many search graph vertices the winning agents open. Figure 1 shows 100 15-puzzles from each of the 40–59 optimal path length groups run on A* and A! in five agent configurations: 1, 2, 4, 6, and 8 agents. We use the median of five runs: this was found to be a reasonable compromise between result quality and available computing resources.

We see the majority of the points falling under the solid par-line, indicating that search using A* is more sluggish than with A!: more states get visited before the optimal solution path is found. Some instances above the par-line – especially for the one agent case – show the vanilla algorithm outperforming A!, likely due to the secondary heuristic being misinformed about the best direction. This is the cost from going depth-first over breadth: all heuristics can be fooled. The trend lines still validate A!. With eight agents, the slope approaches $\frac{1}{2}$, indicating that on average A! needs to see only half the states as vanilla A*. Similar results for A! vs. A? are found in [7].

Searching efficiently in parallel requires a scalable algorithm. Figure 2 shows that A! outperforms A* and A?, and that adding new agents to A! increases its performance. The figure shows runs in optimal length groups, with means of the 100 instances in each group forming a trend line for each of the agent configurations. The group trends are normalized with respect to vanilla A* performance. Finally, the trend lines themselves

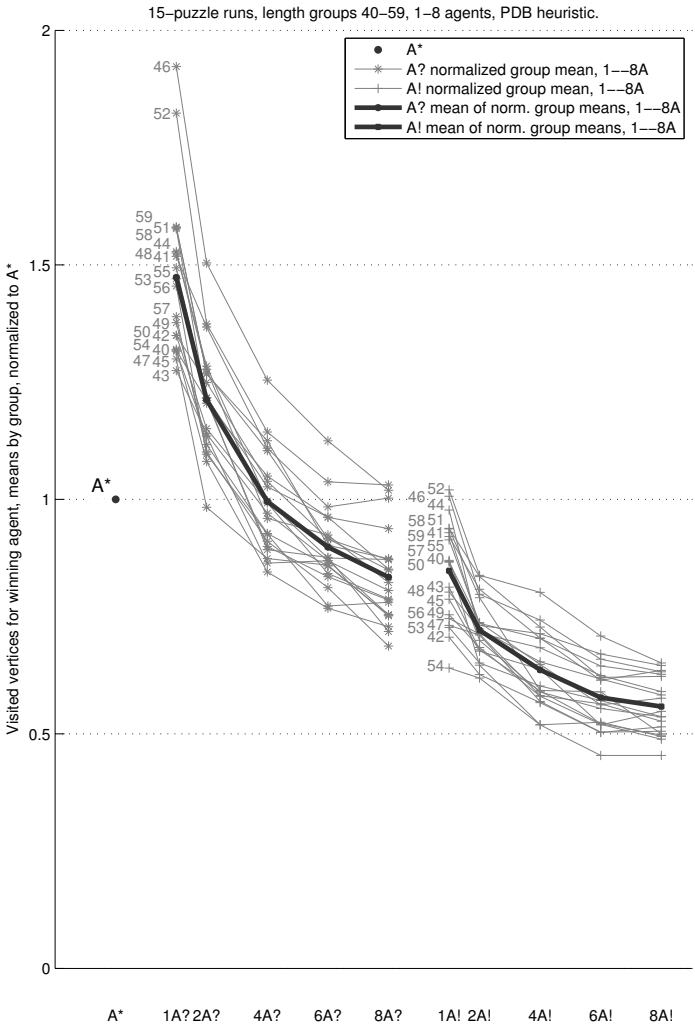

Figure 2. A*-normalized length group means demonstrating scaling benefit from adding more agents. Each line represents a set of instances run over several configurations and repeats, for each method and with the performance scaled to the vanilla A* case. We can see the relative benefit of adding more agents, but also diminishing returns for each expansion. The trend lines overlap to a moderate extent – further illustrated by the mean – suggesting rudimentary asymptotic bounds for the approaches.

are averaged over for a pair of thick mean-of-means curves that summarize over the thousands of data points drawn evenly from the $40-59$ optimal path length range.

We see that A?, perhaps in the spirit of *random restarting* (repeated local search), initially performs worse than A*, as the random selection policy is inferior to a systematic approach, but then with more agents the probabilities turn in its favor. In contrast, A! already starts off well, due to the momentum effect, and gains more power as more agents begin to cooperate.

For some of the groups in Figure 2, A! gets close to the $\frac{1}{2}$ threshold in A*-relative expansion, which also appears to be a plateau for general scalability with regards to

this implementation if not the approach itself. While individual groups exhibit erratic behavior, the overall trend is quite clear: A! outperforms A* and A?, and scales to at least a few agents, but with diminishing returns.

5. Related work

Cooperation has been an theme in AI research for years [12, 13]. The taxonomy work of Crainic and Toulouse [8, 14] is a good place to start exploring the literature on cooperative search. Alba et al. [15] offer extensive surveys on a closely related field of parallel metaheuristics. A! is perhaps best categorized as *distributed search* [5], but the approach is motivated by a *multi-agent system* view of algorithmic cooperation [7].

Barbucha [16] and Ouelhadj and Petrovic [9] present ideas that are similar to A!, but feature cooperation more in terms of traditional parallelism. In A! we propose viewing cooperation as a dynamic heuristic, and present unconstrained concurrency and close interaction as a source of search diversity and performance.

Classical planning tasks make good benchmarks and are often featured in parallel A* papers to give the methods credibility beyond puzzles [2, 4]. However, in these contexts, cooperation effects are rarely studied directly. Information exchange, mostly considered from a search space partitioning and distributed load balancing angle, appears to not have been considered from the heuristic point of view taken in this work. Most A* parallelizations are deterministic and the rest explicitly randomized.

Without search space partitioning, parallelism can be achieved in search through parallelizing node processing in a heavy graph, as in the chess machine *Deep Blue* [17]. Algorithm portfolios and hybrids are another easy way to exploit parallelism, an example being the ManySAT solver [18]. Machine learning methods have also been successfully applied to the discovery of parallel configurations for search [19]. Load balancing through duplicate detection [2], hashing [3, 4], or transposition tables [5] might well be useful in improving exploration diversity also in A!.

6. Conclusion

The 15-puzzle experiments show that the cooperative A! algorithm outperforms both vanilla A* and the non-cooperative random parallel variant A? in this context. Adding more agents to A! clearly improves performance, but the returns are diminishing. Search overhead – the lack of explored path diversity – appears to be a limiting factor in A! performance and an issue worth addressing in future work.

A* expands the search broadly in all directions and in an orderly fashion, in a sense being forgetful about search history. A!, in contrast, prefers depth and emphasizes progress and search momentum. A? is forgetful as well, but through *explicit* randomization, the agents can stumble on the right path faster than in systematic browsing – given enough agents. A! embraces concurrency and *implicit* randomization: the parallel agents cooperate in a nondeterministic way in focusing the search effort in areas that have been found promising. The secondary heuristic serves as a global compass that augments the search when the primary heuristic fails to disambiguate between candidates.

A! combines into a single approach multiple unfinished algorithmic ideas: concurrent execution, asynchronous interaction, implicit randomization, globally informed

depth-first orientation, and the use of a secondary heuristic. A far more detailed study is needed for explicating the exact contribution of each of these factors in A! performance. The approach should also be validated in other contexts.

Still, the experiments indicate that nondeterministic cooperation emerging from asynchronous message exchange *can* be beneficial in heuristic search. Next steps could also include combining A! and cooperation ideas with other parallel A* techniques.

Acknowledgements

I am very grateful to Pekka Orponen for supporting and funding this work. I also thank the reviewers for insightful comments. The computational experiments in this work were performed using the computer resources provided by the Aalto University School of Science *Science-IT* project.

References

[1] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[2] Ethan Burns, Sofia Lemons, Wheeler Ruml, and Rong Zhou. Best-First Heuristic Search for Multicore Machines. *Journal of Artificial Intelligence Research*, 39:689–743, 2010.

[3] Matthew Evett, James Hendler, Ambuj Mahanti, and Dana Nau. PRA*: Massively Parallel Heuristic Search. *Journal of Parallel and Distributed Computing*, 25(2):133–143, 1995.

[4] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a Simple, Scalable, Parallel Best-First Search Strategy. *Artificial Intelligence*, 195(0):222–248, February 2013.

[5] John W. Romein, Aske Plaat, Henri E. Bal, and Jonathan Schaeffer. Transposition Table Driven Work Scheduling in Distributed Search. In *Proc. of the 15th Nat. Conf. A. I. (AAAI '99)*, pages 725–731, 1999.

[6] Rong Zhou and Eric A. Hansen. Structured Duplicate Detection in External-Memory Graph Search. In *Proc. of the 19th National Conference on Artificial Intelligence (AAAI '04)*, pages 683–688, 2004.

[7] Antti Halme. *Cooperative Heuristic Search with Software Agents*. Master's thesis, Aalto University, 2014.

[8] Teodor G. Crainic and Michel Toulouse. Explicit and Emergent Cooperation Schemes for Search Algorithms. In *Proc. of the 2nd Intl. Conf. on Learning and Intel. Optim. (LION '07)*, pages 95–109, 2008.

[9] Djamila Ouelhadj and Sanja Petrovic. A Cooperative Hyper-Heuristic Search Framework. *Journal of Heuristics*, 16(6):835–857, December 2009.

[10] Othar Hansson, Andrew Mayer, and Moti Yung. Criticizing Solutions to Relaxed Models Yields Powerful Admissible Heuristics. *Information Sciences*, 63(3):207–227, September 1992.

[11] Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive Pattern Database Heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[12] Tad Hogg and Bernardo A. Huberman. Better Than the Best: The Power of Cooperation. In *1992 Lectures in Complex Systems*, volume V, pages 165–184. Addison-Wesley, 1993.

[13] William A. Kornfeld. The Use of Parallelism to Implement a Heuristic Search. In *Proc. of the 7th Intl. Joint Conference on Artificial Intelligence (IJCAI '81)*, volume 1, pages 575–580, 1981.

[14] Teodor G. Crainic and Michel Toulouse. Parallel Meta-heuristics. In *Handbook of Metaheuristics*, chapter 17, pages 497–541. Springer, 2010.

[15] Enrique Alba, Gabriel Luque, and Sergio Nesmachnow. Parallel Metaheuristics: Recent Advances and New Trends. *International Transactions in Operational Research*, 20(1):1–48, 2013.

[16] Dariusz Barbucha. Search Modes for the Cooperative Multi-agent System Solving the Vehicle Routing Problem. *Neurocomputing*, 88:13–23, July 2012.

[17] Murray Campbell, A. Joseph Hoane Jr., and Feng-hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134 (1-2):57–83, January 2002.

[18] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: A Parallel SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009.

[19] Diane J. Cook and R. Craig Varnell. Adaptive Parallel Iterative Deepening Search. *Journal of Artificial Intelligence Research*, 9(1):139–166, August 1998.