Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Antti Halme

Cooperative Heuristic Search with Software Agents

Master's Thesis
Espoo, May 13, 2014

Supervisor: Professor Pekka Orponen, Aalto University
Advisor: — " —

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Antti Halme

Title:
Cooperative Heuristic Search with Software Agents

Date: May 13, 2014 **Pages:** 88

Major: Information and Computer Science **Code:** T-79

Supervisor: Professor Pekka Orponen

Advisor: — " —

Parallel algorithms extend the notion of sequential algorithms by permitting the simultaneous execution of independent computational steps. When the independence constraint is lifted and executions can freely interact and intertwine, parallel algorithms become concurrent and may behave in a nondeterministic way. Parallelism has over the years slowly risen to be a standard feature of high-performance computing, but concurrency, being even harder to reason about, is still considered somewhat notorious and undesirable. As such, the *implicit randomness* available in concurrency is rarely made use of in algorithms.

This thesis explores concurrency as a means to facilitate algorithmic cooperation in a heuristic search setting. We use agents, cooperating software entities, to build a single-source shortest path (SSSP) search algorithm based on parallelized A*, dubbed A!. We show how asynchronous information sharing gives rise to implicit randomness, which cooperating agents use in A! to maintain a collective secondary ranking heuristic and focus search space exploration.

We experimentally show that A! consistently outperforms both vanilla A* and a noncooperative, explicitly randomized A* variant in the standard n-puzzle sliding tile problem context. The results indicate that A! performance increases with the addition of more agents, but that the returns are diminishing. A! is observed to be sensitive to heuristic improvement, but also constrained by search overhead from limited path diversity. A hybrid approach combining both implicit and explicit randomness is also evaluated and found to not be an improvement over A! alone.

The studied A! implementation based on vanilla A* is not as such competitive against state-of-the-art parallel A* algorithms, but rather a first step in applying concurrency to speed up heuristic SSSP search. The empirical results imply that concurrency and nondeterministic cooperation can successfully be harnessed in algorithm design, inviting further inquiry into algorithms of this kind.

Keywords: concurrency, parallel algorithm, nondeterminism, A*, agent, cooperation, heuristic search, 15-puzzle

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Antti Halme		
Työn nimi:	Heuristinen yhteistyöhaku ohjelmistoagenttien avulla		
Päiväys:	13. toukokuuta 2014	**Sivumäärä:**	88
Pääaine:	Tietojenkäsittelytiede	**Koodi:**	T-79
Valvoja:	Professori Pekka Orponen		
Ohjaaja:	— ” —		

Rinnakkaisalgoritmit sallivat useiden riippumattomien ohjelmakäskyjen suorittamisen samanaikaisesti. Kun riippumattomuusrajoite poistetaan ja käskyjen suorittamisen järjestystä ei hallita, rinnakkaisalgoritmit voivat käskysuoritusten samanaikaisuuden vuoksi käyttäytyä epädeterministisellä tavalla. Rinnakkaisuus on vuosien saatossa noussut tärkeään rooliin tietotekniikassa ja samalla hallitsematonta samanaikaisuutta on yleisesti alettu pitää ongelmallisena ja ei-toivottuna. Samanaikaisuudesta kumpuavaa *epäsuoraa satunnaisuutta* hyödynnetään harvoin algoritmeissa.

Tämä työ käsittelee käskysuoritusten samanaikaisuuden hyödyntämistä osana heuristista yhteistyöhakua. Työssä toteutetaan agenttien, yhteistyökykyisten ohjelmistokomponenttien, avulla uudenlainen A!-hakualgoritmi. A! perustuu rinnakkaiseen A*-algoritmiin, joka ratkaisee yhden lähteen lyhimmän polun hakuongelman. Työssä näytetään, miten ajastamaton viestintä agenttien välillä johtaa epäsuoraan satunnaisuuteen, jota A!-agentit kollektiivisesti hyödyntävät toissijaisen järjestämisheuristiikan ylläpitämisessä ja edelleen haun kohdentamisessa.

Työssä näytetään kokeellisesti, kuinka A! suoriutuu niin tavanomaista kuin satunnaistettuakin A*-algoritmia paremmin *n-puzzle* pulmapelin ratkaisemisessa. Tulokset osoittavat, että A!-algoritmin suorituskyky kasvaa lisäagenttien myötä, mutta myös sen, että hyöty on joka lisäyksen jälkeen suhteellisesti pienempi. A! osoittautuu heuristiikan hyödyntämisen osalta verrokkeja herkemmäksi, mutta myös etsintäpolkujen monimuotoisuuden kannalta vaatimattomaksi. Yksinkertaisen suoraa ja epäsuoraa satunnaisuutta yhdistävän hybridialgoritmin ei todeta tuovan lisäsuorituskykyä A!-algoritmiin verrattuna.

Empiiriset kokeet osoittavat, että hallitsematonta samanaikaisuutta ja epädeterministä yhteistyötä voi onnistuneesti hyödyntää algoritmisuunnittelussa, mikä kannustaa lisätutkimuksiin näitä soveltavan algoritmiikan parissa.

Asiasanat:	samanaikaisuus, rinnakkaisalgoritmi, epädeterministisyys, A*, agentti, yhteistyö, heuristinen haku, 15-puzzle
Kieli:	englanti

Acknowledgements

Many people have over the years demonstrated persistent and often unfounded faith in my ideas and abilities, for which I am humbled and deeply grateful. This generosity has allowed me to discover and explore a scientific and professional field that I now consider my own. The conversations I enjoyed with the wise people that crossed my path definitely marked the highlights of my study years and left me with an education I value far more than my academic achievements.

I thank Prof. Pekka Orponen for supervising and funding this thesis work. I fully acknowledge the special nature of the opportunity that I was given both here and also in my previous research project. Pekka provided with me with what many researchers can only only dream of: unwavering support, thoughtful feedback and a free hand to pursue my interests.

I thank D.Sc.(Eng.) Vesa Hirvisalo for introducing me to research very early on, completely changing the trajectory of my studies, if not my life. Vesa pulled me in to his research group, showed me how coffee is turned into science and opened so many doors for me that I'm still trying to figure out which one to go through.

I thank Prof. Stavros Tripakis for broadening my view of the academic world and Ph.D. Timo Tossavainen, my tutoring teacher, who perhaps inadvertently made me realize what university studies are all about.

I thank Sami Honkonen and the whole Reaktor family for a vitally important summer breather that I took before this thesis effort. Working in the real world gave me some much needed perspective.

I also extend my thanks to all of my teachers, from the first grade on, who always fostered my pursuit of knowledge and interest in science.

Finally, I thank my colleagues, friends and family, whom I never fail to neglect.

The experimental study presented as a part of this thesis was performed using the computing resources of the Aalto University School of Science "Science-IT" project.

Espoo, May 13, 2014

Antti Halme

Contents

Chapter 1

Introduction

No widely accepted, precise definition for an algorithm exists, but Knuth's notion [74] is at least as good as any, paraphrasing:

> An algorithm is a procedure that computes outputs with a special relation to the inputs and terminates after a *very* finite number of precisely defined basic steps.

This an uncontroversial formulation and suitably loose enough to allow not only *sequential algorithms* (or *serial algorithms*), where the presence of a single instruction processor (core) is implied, but also *parallel algorithms*, where multiple processors can work on independent computational steps simultaneously. Parallel algorithms terminate with output just like sequential ones, but aim to spread the computing effort evenly out to parallel hardware in pursuit of performance gains.

While failure to correctly isolate simultaneous executions is often problematic, maintaining execution independence is not always necessary, and in fact the independence constraint can sometimes be lifted. We adopt the term *concurrency* to refer to unconstrained parallelism, where simultaneous executions can freely interact and influence one another.

Sequential and dependency-free parallel executions are inherently deterministic, but concurrent executions are not. Rendering concurrent programs deterministic requires additional control structures that enforce an ordering between the interacting executions. As this is exactly what concurrent approaches try to do away with, we can say that concurrent executions are not strictly ordered, but rather fundamentally nondeterministic.

This nondeterministic nature of concurrency is not reflected in the practice of parallel programming today. Rather, a great deal of effort is placed in keeping concurrent programs deterministic through sequentially inspired abstractions and complex locking choreographies. At the same time many effective algorithms are

founded on the idea of relinquished determinism in the form of explicitly randomized procedures [92].

Uncontrolled interaction between distinct parts is a characteristic feature of complex systems and processes in the natural world, and yet our computations must fit this parallel model of independence or minimized communication. It is true that at the hardware level instruction execution is linearized per functional unit, but at the same time more and more of these independent execution units are made available to the programmer.

Parallel programming is hard for a number of reasons, but it is really the disconnect between sequential thinking, concurrent execution and parallel hardware that has kept the software industry back for many years now. The parallel view to instruction execution matches modern multi-core hardware better than the sequential one, but all of our algorithms, tools and programming languages were thought up under the old sequential mindset. New approaches to parallel computing and concurrency are therefore well worth exploring.

The starting point of this thesis is the unpredictable interaction and intertwining of concurrent executions. If deterministic control is not maintained, concurrent programs give different results based on the order in which instructions execute. With inconsistent results, it is easy to equate concurrency with malfunctioning parallel programs, but this view is too insular. The notorious reputation of concurrent programming is not completely unfounded, but it relies on sweeping statements about the nature of "acceptable" parallel computation.

In this work we explore a different view to parallel computing, one based on dependence and cooperation. Instead of working against the natural disorder of concurrency, we embrace it and try to use it to our advantage. We concentrate on cooperative interaction 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.

We focus on the cooperation of multiple distinct worker components, agents, executing concurrently 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. Our context is heuristic search, an important algorithmic technique with numerous real world applications from pathfinding to resource optimization.

The main contribution of this thesis is a new kind of a cooperative heuristic search algorithm, a parallel variant of the A* algorithm [53, 115], dubbed A!. The algorithm features cooperating search agents that communicate asynchronously and share information in order to collectively maintain a secondary tiebreaking heuristic. The A! agents explore the search space in an implicitly random fashion, directed by this dynamic ranking heuristic.

We empirically evaluate A! by solving instances from the standard benchmark problem family of n-puzzles [115, 122]. The performance of A! is compared with vanilla A* and a randomized non-cooperative parallel A* variant. The main observations from the A! experiments are:

- *Cooperation benefit*: A! consistently outperforms the parallel version of vanilla A* and the explicitly randomized parallel A* variant, as measured by fewer explored states by the winning agent.
- *Scaling benefit*: A! performance improves with more cooperating agents, but the returns are clearly diminishing.
- *Path diversity*: A simple search space exploration visualization, together with execution data, suggests that the observed performance gain in favor of A! is due to the more focused nature of the search effort.
- *Heuristic impact*: A better search heuristic improves A! performance relatively more than that of the competition.
- *Hybrid performance*: Combining the implicit randomness of A! with explicitly random search space exploration does not yield a better search algorithm.

For the n-puzzle problem, the studied A! implementation is not as such competitive against state-of-the-art parallel A* algorithms. Rather, the algorithm is a first step in applying concurrency to speed up heuristic search. Still, the results imply that concurrency, nondeterministic cooperation and implicit randomness can successfully be harnessed in algorithm design.

We begin with a background review and discuss related work more in chapter 5. Chapter 3 outlines the A! algorithm and then chapter 4 presents results from the experimental evaluation. Final thoughts on A!, cooperation, concurrency and implicit randomness are offered in the concluding chapter 6.

Chapter 2

Background

This chapter gives an overview of the topics that underlie the A! algorithm and specifically our approach to cooperation. We begin with the concepts of parallelism and concurrency and aim to elucidate their difference. We then consider the various dimensions of parallel computing, followed by a brief discussion on parallel speedup and performance issues. A section on parallel programming presents the broader context of this work: understanding some of the main issues helps in thinking about parallelism in a new way.

Cooperation, with its many tradeoffs and challenges, is explored after that. We first consider cooperation in general and then focus on cooperative search specifically. We finish off the background chapter with a brief discussion on agents and multi-agent systems, emphasizing the general idea of reasoning about computation through objects with identity.

2.1 Parallelism and concurrency

Parallelism is about performance, the goal being execution speedup. Today, computers have multiple processing units and various kinds of specialized parallel hardware that make the parallel execution of instructions possible. Solving problems efficiently on these parallel computers, with as much of the processing capability in use as possible, typically requires the design of algorithms that specify multiple operations for each execution step, i.e., parallel algorithms [13].

Parallelism comes in many shapes and forms. On a parallel computer, the operations of a parallel algorithm can be performed simultaneously by different processors, but a single-processor computer can also run parallel algorithms. Further, even a single-processor computer can exploit algorithmic parallelism using the low-level parallel functionality found in modern processors. Matching the parallelism described in an algorithm to the parallelism available in the hardware

remains an open challenge in modern computing.

In contrast to the physical reality of parallelism in hardware, *concurrency* is a more abstract, logical notion of actions happening at the same time. Concurrency is fundamentally a software notion, a property of a system with simultaneously active, interacting parts. While the difference between parallelism and concurrency is sometimes subtle or non-essential, understanding the distinction enables unambiguous discussion. Sadly, however, the terms continue to be used carelessly and even interchangeably.

Unsatisfied with the lack of precision in specifying these concepts, Buhr and Harji [18] offer an opinionated view on the matter. They find that having two distinct terms – one founded on hardware phenomena and the other on software – is essential, but at the same time in practice there is a strong dependence between the two: programs are often tailored to exploit only one kind of parallelism. They arrive in their treatise to a central conclusion: "There are no parallel programs; there are only concurrent programs that happen to execute in parallel given appropriate hardware."

We adopt a policy of using the term 'parallel' to refer to independent simultaneous executions and related hardware phenomena, and reserve 'concurrent' for situations where the executions are free to interact and typically represent distinct logical units in software. We emphasize the deterministic nature of parallelism in contrast to the nondeterministic behavior exhibited by concurrent executions.

Making good use of either parallelism or concurrency is still a kind of a novelty today, but both are slowly finding their place as programming languages, tools and techniques develop [9, 84, 90, 91]. Parallel programming is closely associated with high-performance computing, while more concurrent programs are found, for example, in the various back-end processes that web servers run. Locks and other synchronization primitives have of course been used for a long time to manage software systems with interacting parts, but there the effort has traditionally been in keeping independent things independent, rather than in using concurrent interaction for gains in an algorithmic sense.

Opportunities for deterministic parallelism have been extracted and exploited for decades, but concurrency has traditionally garnered much less interest. One could even say that optimizing hardware for specific flavors of parallelism – and vice versa – has cast a shadow over parallel programming. Without powerful abstractions, programmers are forced to familiarize themselves with low-level details of the target systems and the sequential reality of execution pipelines.

In contrast, higher-level views to parallel programming have not really been explored much yet. Ensuring program correctness using abstractions carried over from the sequential era has kept programmers busy since the first parallel computers, with mixed success. Concurrency continues to be a challenge to programmers

accustomed to reasoning about sequential programs, and today is mostly associated with the undesirable properties of malfunctioning parallel programs.

This work begins with the assumption that concurrent interaction by itself can be a useful algorithmic component. Efficient, hardware exhausting parallel software is important and will continue to be so, but alternative approaches to organizing concurrent programs have not been explored that much. Instead of enforcing interaction rules to guarantee correctness, we keep interaction removed from correctness. Instead of replacing some program logic with concurrency, we augment a sequential algorithm with cooperation.

2.2 The parallel computing landscape

An overview of parallel computing serves as a backdrop for the cooperative algorithm that we develop in this thesis. Parallel computing capability is today available in many form factors from local high-performance computing to distributed clusters, grids and cloud computing setups. Even handheld mobile devices have hardware support for parallelism. These parallel computers, physical or logical, feature multiple processing units (cores) and complex high-performance functionality that cannot easily be abstracted away from the casual software developer.

There is a long tradition of algebraic transformations and other mathematical techniques that are commonly used to parallelize numerical computations, often for a particular application [2, 68, 110]. As a result, parallel hardware designs have over the years grown organically to meet these very specialized needs. Modern parallel computers feature complex processing pipelines and rich instruction sets that offer great parallel processing power to those who can harness them.

Core clock rates used to grow at a predictable rate, but since 2004, the trend has been towards *more* rather than *more capable* processing units [39]. This stems from the physical realities of energy and power that have begun to constrain maximum processor frequencies [9]. Beyond manufacturing limits, parallel hardware design issues today revolve around wire delays, power efficiency and the overall design complexity [37].

Esmaeilzadeh et al. [39] (see also Bose [14]) voice concerns about the longevity of the multi-core era, again due to seemingly inescapable physical limitations, but simply on lackluster hardware utilization as well. The argument goes that if the challenges of scalable parallel programming are not resolved, the currently dominant multi-core paradigm will not last. Nevertheless, the future of computing is parallel, and the immediate future of parallel computing lies with substantial local parallel processing power [9, 46, 103] complementing massive-scale distributed computing [8, 29].

Moving on to have a closer look at parallelism, Flynn's taxonomy [44, 110] offers

a classical demarcation of parallel computing approaches. While modern hardware does not fully align with this categorization, the classes define the main tradeoff that is still relevant today: generality vs. performance. The taxonomy reflects how instructions and data are organized in parallel architectures and comprises four categories:

- *SISD* (single instruction, single data) reflects the plain, sequential von Neumann architecture, where instruction and data are fetched from memory and executed on a single processor.
- *SIMD* (single instruction, multiple data) refers to data-parallelism, where the same operation is applied at some level of abstraction to several units of data of the same kind.[1]
- *MISD* (multiple instruction, single data) is the rarest of the four, referring to situations where the same data unit is used in separate computations and multiple different functional units.
- *MIMD* (multiple instruction, multiple data), the most general of the four, represents independent processing units free to process data asynchronously.[2]

Parallel hardware is MIMD when the cores are fully featured and general purpose, but caching effects, for example, make the processors much more coupled. SIMD offers a good approximation of the programming model associated with GPUs and other accelerators, but here, too, memory access policies and coherence issues dominate the parallel performance. Vector instructions on modern processors exhibit SIMD behavior at a smaller scale.

Modern parallel computers [37] form a logical computing unit by themselves or as part of a larger distributed computing setup. The variety and capacity of parallel processing power available to a software developer today is vast and will only increase in the future. The challenge is two-fold: first, applications must be written with a parallel mindset to expose parallelism, and second, the parallelism must be matched with the abstractions provided by the hardware. Despite decades of research and a broad effort into making this capability accessible, parallel programming remains a somewhat esoteric expert activity.

Parallelism comes in many forms, which are classified by granularity, the scale at which the effect is evident. Most low-level parallelism close to hardware is deterministic, but as larger and larger abstractions are parallelized, their execution

[1] The wonderful notion of *embarrassingly parallel* typically refers to (previously) missed opportunities for SIMD parallelism [8, 84].

[2] Even MIMD architectures represent the von Neumann style, as the same system bus serves all the processing units. One could even say that the "bottleneck" becomes even worse in multiprocessor systems, as with multiple users the bus gets contented even faster.

begins to take time and they have to be scheduled to be run. With dependencies, the scheduling order of simultaneous executions begins to matter, and if left uncontrolled, the results become nondeterministic.

To get a big picture view of parallelism and the origins of concurrency, we next go briefly through some of the levels of parallelism available to a programmer.[3] We start from the lowest levels of deterministic parallelism, but soon encounter nondeterminism as we conceptually move away from the hardware.

Bit-level parallelism follows from expanded word length in the processor, which translates into fewer total instructions as longer bit sequences can carry more information. *Instruction-level parallelism* is available through pipelining, where a sequence of instructions can pass through the functional units of the processor in synchronized lockstep instead of one at a time. If multiple independent functional units are available, the processor can employ multi-issue techniques and feature superscalarity or an extended instruction set. Multi-threading takes instruction-level parallelism further by minimizing processor idling time with thread scheduling.

Data-level parallelism (and *array parallelism*) is available in situations where the same operation is to be applied to a set of data units. Data-parallelism can be viewed as both a high-level programming concept as well as a collection of compiler optimization methods. The `map`-operation in functional languages and various work-distribution schemes in distributed computing is an example of the former view, while vector instructions and dependency-driven loop unrolling are examples of the latter. Data-parallelism has been an integral part of the parallel computing story for decades, but is gaining new speed as functional programming is now becoming mainstream [103].

Function or task-level parallelism offers the most general view to parallelism. A program with task-parallel parts allows program control to split into multiple parts in order to facilitate the simultaneous execution of instructions. Threads, processes and a multitude of other parallel programming abstractions belong in this category. Task-level parallelism can lead to concurrent phenomena and is therefore the form of parallelism that we concentrate on in this thesis.

Parallel computing efforts are not constrained by the resources of a single host machine. The notion of a logical computer can through point-to-point interconnects of various topologies and switching strategies be spread into clusters, cloud setups and all the way to data-center scale computing. These *distributed systems* [29] promise great computing resources in a scalable way that meets the variable needs of various parallel programs, the flip side being the additional woes resulting from network unreliability, nonuniformity and the complexity of spread out control.

[3]Note that the levels of parallelism are *not* mutually exclusive, but rather can and should be mixed and matched as necessary.

Local and distributed parallelism gives rise to a fundamental divide in parallel system design. Local parallelism is typically based on a *shared memory* system, while distributed configurations operate on a system based on *message passing*.

Shared memory systems feature threads in a shared address space, communicating and synchronizing through the primitives of the host machine. Message passing systems typically have a notion of remote processes managing an address space of their own, and a collection of dedicated message passing routines for explicit data sharing and synchronization.

Both shared memory and message passing systems can be realized in a variety of semantic flavors and using many technologies. OpenMP and MPI are currently the most prominent standards for shared memory and message passing computing, respectively, with users in both academia and the software industry. A wealth of parallel programming APIs and libraries from POSIX threads to GPGPU programming exist and continue to spring up.

2.3 Speedup and parallel performance

Some fundamental facts about parallel programs limit the performance gains that can be achieved through parallelism [37, 110]. Parallel programs do not necessarily scale in a linear fashion – many exhibit *diminishing returns* for each additional worker. Understanding the relation between sequential and parallel programs enables meaningful discussion on performance and scalability issues.

Denote execution time T, sequential software application A_{seq}, a parallelized version of the same application A_{par}, the fraction of total work (of A_{seq}) parallelized (in A_{par}) F, and available processing units P. *Amdahl's law* (or Amdahl's argument) [6, 112] notes that only the parallel fraction of a program can reap the benefits of parallel hardware, the sequential part being constant,

$$T[A_{par}] = T[A_{seq}] \times [(1-F) + F/P].$$

Defining ideal speedup from parallelization S_{par} as the ratio between the sequential and parallel versions gives

$$S_{par} = \frac{T[A_{seq}]}{T[A_{par}]} = \frac{1}{(1-F)+F/P} = \frac{P}{F+P(1-F)} < \frac{1}{1-F},$$

which means that "the maximum speedup (the maximum number of processors which can be used effectively) is the inverse of the fraction of time the task must proceed on a single thread" [112].

Amdahl's law is unforgiving. With a parallel fraction of 0.95 the speedup can be at most 20 and even with 0.99 the limit is reached at 100. Dubois [37] reminds

that in practice the scaling that is typically seen follows a "a mortar pattern": adding more workers after a certain point only makes things slower. The returns are first diminishing and then workers start getting in each other's way.

Speedup also raises a question about efficiency, the fraction of time a processor is usefully employed by computations that also have to be performed by a sequential program [112]. Let $C_{seq} = 1 \cdot T[A_{seq}]$ and $C_{par} = P \cdot T[A_{par}]$ be the cost of a sequential program running on a single processor and a parallel program running on P processors, respectively. Then we have notion of efficiency, E_{par} with,

$$E_{par} = \frac{C_{seq}}{C_{par}} = \frac{1 \cdot T[A_{seq}]}{P \cdot T[A_{par}]} = \frac{S_{par}}{P}.$$

Ideal speedup implies that best possible speedup, with perfectly parallelizing tasks, is linear, $S_{par} = P$, efficiency being $E_{par} = 1$. However, due to partially shared memory hierarchies and other processor phenomena, there are situations where *super-linear scaling* [22, 37], with $S_{par} > P$, $E_{par} < 1$, can occur. "Hot" data previously fetched for one processor may be used to serve the requests issued by other processors. Optimizing memory access patterns is standard practice in GPU programming and other high-performance computing.

Note that all speedup numbers must be taken with a grain of salt, as the comparison is fair only when the best possible algorithm is used in the sequential version. If the algorithms used in the sequential and parallel versions are intrinsically different, then all bets are off in comparing them. With the best possible algorithm chosen for both versions, the resulting speedup should always be less than the ideal. A sequential algorithm can always simulate the parallel algorithm.

Amdahl's law paints quite a bleak picture for parallel programming, but there is a bit more to the parallelism story. Amdahl's law assumes that the size of the problem to solve is fixed, or that parallelism is used to process the same data faster, but another assumption can also be made. Assuming that the execution time is fixed and workload size can increase with the number of processors leads to *Gustafson's law* (or Gustafson–Barsis' law) [50], with notation as before,

$$T[A_{par}] = (1 - F) + \frac{P \cdot F}{P} = 1, \qquad T[A_{seq}] = (1 - F) + P \cdot F,$$

where the sequential process takes P times as long to compute the *parallel* part. The sequential part stays the same, but an increase in P results in more work being done in the same time period, giving a total speedup in relation to completed work,

$$S_{par} = \frac{T[A_{seq}]}{T[A_{par}]} = \frac{(1 - F) + P \cdot F}{1} = 1 - F + P \cdot F,$$

meaning that for the parallel part of the program, linear scaling is possible. Where Amdahl's law condemns parallel architecture for failure, Gustafson's gives a glimmer of hope that parallel computing is indeed worth exploring [37].

While Amdahl's and Gustafson's laws are useful measures, the models are very simple and do not fully reflect the reality of parallel computing. Other, more detailed measures for parallel speedup have also been proposed over the years, including *isoefficiency* [49], and *simplified memory-bounded speedup* [124].

Parallel models maintain that executions are independent: concurrent interaction is not currently considered a factor in performance comparisons. While it is true that a single processor can always simulate many, dedicated concurrency-aware measures could prove useful from a practical performance point of view.

2.4 Parallel programming

In this work we present cooperation as a high-level programming concept based on concurrent interaction, but a good understanding of the various low-level issues of traditional parallel programming is still useful. While the agent abstraction we utilize does have well matching software abstractions, namely actors [1], most parallel programming today is still based on synchronization primitives. Low-level mechanisms have proven to be a poor fit for scalable concurrency, but fundamental parallelism issues cannot be ignored even with modern programming languages.

Herlihy and Shavit [56] give a thorough introduction to the art of parallel programming in the multi-core era, from fundamentals to advanced techniques: "The art of programming multi-cores, currently mastered by few, requires an understanding of new computational principles, algorithms and programming tools." Many pitfalls await the unprepared programmer, but even the sheer amount of complex parallel capability available today can be overwhelming.

Dubois [37] notes that regardless of the parallel programming model [68, 117], parallel software must be constructed in a specific way that exposes the parallelism. Parallelizable tasks must be identified, partitioned in a balanced way and coordinated so that the end result appears as if done in entirety by a single processor. Maintaining this deterministic behavior[4] requires careful control over the dependencies between the interacting executions.

Parallel programming – as we know it – is hard [9, 18, 46, 47, 90, 91, 110]. Parallel programmers face many design issues and outright problems in constructing well-performing parallel software. While new languages, techniques, tools and methodologies make it increasingly easy to harness parallel processing power, the "harsh realities of parallel programming" continue to haunt parallel programmers.

Dedicated and shared memory is organized in several levels of memory hierarchy from caches to slow storage, and application data must fit this structure well or some of the parallelism gains can be lost to various data-logistical overheads.

[4] Deterministic in the sense that for the same input, effectively the same operations take place every time.

Concurrent data access renders programs vulnerable to problems caused by *data races*, situations where the outcome of a computation depends on the order in which memory locations are read and written to. Maintaining memory consistency is expensive and overall correctness is even harder to enforce.

Synchronization builds on the notion of a *critical section*, a protected program segment with an upper bound on the number of simultaneously executing control threads. Synchronization primitives disallow simultaneity by enforcing *mutual exclusion*: locks, semaphores, monitors and other constructs project a sequential ordering on parallel execution. Some synchronization primitives, such as compare-and-swap, are hardware supported.

Synchronization primitives are used to build concurrent data structures, transactions and other higher-level concepts that simplify building parallel programs that function correctly, but as general purpose mechanisms they carry a performance overhead. Many distributed systems rely on a semi-centralized consensus mechanisms to ensure correctness. Parallel programmers are forced to address the tradeoff between correctness and performance in every application.

Synchronous execution and blocking communication mechanisms are slow, but asynchronous and nonblocking mechanisms require additional control. Failure to control interaction may lead to problems that are fatal, but difficult to encounter in testing, such as *deadlock*, where all processes or threads are waiting for each other, and *livelock*, where the system is not locked dead, but no progress is made either. Various priority schemes can lead to *starvation* and *priority inversion*, in which the resource distribution mechanism backfires and the system malfunctions.

Parallel programming is hard because programmers must manage various parallelism issues at several levels of abstraction simultaneously, all the while ensuring high performance. Progress in compiler technology, parallel hardware and programming languages continue to make aspects of parallel programming – especially "close to the metal" – more manageable, but task-level parallelism and the effects of concurrency remain an open challenge.

Hardware trends point to computing environments with dozens to hundreds of cores available; some with dedicated tasks, but most free for applications to make use of [110]. On the other hand, many platforms feature multiple chips that are already capable of general purpose computation. In the future the parallel computing environment will be highly heterogeneous, even within a single host.

Parallel programming is a critical problem that must be solved in order to really exploit concurrency, but the challenges parallelism presents are so complex, that it is unlikely that any single solution will prevail. Parallelizing efficiently and conveniently both locally and in a distributed manner will be key. Opportunities for parallelism take place at many levels, so approaches that can handle diversity in parallel execution will do well in the future.

The general approach to parallelism today is identifying and exploiting opportunities for parallelism as they arise in the development effort. High performance requires access to low level abstractions, but ideally hardware details would be hidden by default, only to be exposed when necessary.

If some of the performance can be sacrificed, then simplified, more generic parallel abstractions can be made available. This is the path taken in *parallel strategies* -based parallel programming, as featured in Haskell and other modern functional languages. Peyton Jones [103] argues that no single *cost model* will ever suit all programs and computers. Functional programming is a great match to parallel and concurrent programming at scale, and Haskell in particular, as it can host multiple paradigms simultaneously.

Rauber [110] notes that while there is a lot of research going on in the parallel programming community – a collaborative search for the right abstractions – there are already many effective techniques available. Parallel programming is still considered to be an expert activity, but the interest in parallel programming continues to be high. New computing ideas are here desperately needed.

One things is clear: allowing programs to compose several simultaneously executing parts – to compute in parallel and concurrently – gives rise to new ways of viewing computation and solving computational tasks.

2.5 Cooperation

Task-level parallelism features multiple threads of control that either operate independently or interact with each other in some way. As communication can be costly, and maintaining control over concurrency is even more so, most parallelism involves fairly independent subtasks and only limited interaction. While already this "easy" parallelism can give satisfactory speedup results, it misses the opportunity of using communication to advance the execution of individual subtasks.

Hogg and Hubermann [62] note that one needs to look no further than people and human societies to appreciate the power of *cooperation*. Groups of people can achieve together more than individuals in isolation and can solve problems of collective interest more efficiently than any single person acting alone. The challenge then is in implementing in software these mechanisms of coordination and communication that seem to work with humans.

Cooperation in a computing context involves agents that interact through information sharing. Crainic and Toulouse [31] distinguish between cooperation as an algorithm design strategy and cooperation as the decomposition of a distributed algorithm into concurrent processes. We focus on the first paradigm, where we augment stand-alone problem solvers with an explicit cooperation mechanism that combines these solvers into a single strategy.

More precisely, two features can be found in all cooperation mechanisms [31]:

1. a set of highly *autonomous programs* (APs), each implementing a particular method, and

2. a *cooperation scheme* combining these autonomous programs into a single problem-solving strategy.

Without predetermined order and control, cooperation among APs leads to unpredictable behavior. Deliberate and explicit cooperation realized in a physical system yields a sequence of interactions that are not independent, but rather correlated. This sometimes induces implicit structure and emergent phenomena, "ripple effects", commonly found in nature's dynamic processes [31].

In a computational setting small delays in instruction scheduling and execution are enough to generate unpredictable orderings, resulting in the notorious reproducibility issues associated with concurrency. Cooperation schemes aim to take advantage of this behavior and uncertainty – in much the same way as standard randomized algorithms do – by making the indirect interactions influence the participating APs. The challenge, of course, is in controlling the uncontrollable.

The study of indirect cooperation is in the beginning, and concrete results are scarce. Designing systems that consistently exhibit system-wide emergent cooperative behavior remains a challenge. Some tools from related fields, such as nonlinear dynamical systems and chaos theory, have been explored in the past, but no significant advances have been made in understanding cooperation. Crainic and Toulouse [32] believe that this trend will hold: "Research in [cooperative methods] will probably continue to be mostly empirical for some time in the future, while theoretical models are being built and put to the test."

The design of information exchange mechanisms is the key to good performance with cooperative methods. Too little or too much communication can lead to overheads that defeat the gains from parallelization. Crainic and Toulouse identify the main cooperation design issues [31]:

- content (what information to exchange),

- timing (when to exchange it),

- connectivity (the logical inter-processor structure),

- mode (synchronous or asynchronous communications),

- exploitation (what each autonomous program does with the received information), and

- scope (whether new information and knowledge is to be extracted from the exchanged data to guide the search).

Cooperation mechanisms can also be classified by the nature of the solutions being shared. *Adaptive memory* methods store partial solutions and combine them for new complete ones, which are then worked on by cooperating APs [111]. *Central memory* approaches pass complete solutions around in a neighborhood or population based configuration [83].

2.6 Cooperative search

Search is a fundamental task in computer science and natural target for cooperative approaches. The overall task and subtasks are very simple to define and there is a lot of work to distribute, but the problem is not immediately parallel, as the main abstraction is typically a rooted graph. In a heuristic setting the search algorithm has extra knowledge about the search space that is not encoded in the graph, but can be used to direct the search effort. We focus on heuristic search in our exploration of cooperation in this work.

Crainic and Toulouse [31, 32] together with Hail [30] establish a taxonomy of cooperative search under the header of *parallel meta-heuristics*, building on results from earlier work on parallel local search methods. They identify three dimensions: search control, search communication and search differentiation.

The *search control dimension* examines how the global search is managed. There is either a single process overseeing the proceedings, as in master-slave configurations, or the control has been spread out to several autonomous programs that manage the search collegially in collaboration or not. These are identified as *1-control* ($1C$) and *p-control* (pC), respectively.

The *search communication dimension* reflects information exchange policies. The main divide is between synchronous and asynchronous modes of communication. The former is a fixed-schedule policy, where APs stop and share information in a predefined pattern or as instructed by external authority, while the latter one emphasizes the APs' autonomy. Asynchronous communication relies on connections that are established and brought down dynamically during execution. To further reflect the varying nature of the information exchanged, we have four communication classes: *Rigid* (RS) and *Knowledge Synchronization* (KS) and, symmetrically, *Collegial* (C) and *Knowledge Collegial* (KC).

Approaches following a Rigid Synchronization policy make no use of the simultaneously executing searches beyond the best overall solution being established once all independent APs stop. This the classic independent parallel multi-start approach, which is easy to implement and may give good results, but does not

hold up against cooperative methods.

Knowledge Synchronization cooperation strategies take the same general approach as the Rigid ones, but attempt to take advantage of parallel exploration by synchronizing at predetermined intervals. In master-slave configurations the master collects results and usually restarts the search from the best location. Giving search APs the power to initiate synchronization with some or all other searches gives more flexibility. This is the case, for example, in the migration mechanism in parallel genetic algorithms.

Asynchronous strategies divide into Collegial or Knowledge Collegial based on the quantity and quality of the information being shared and the "new" knowledge that is inferred from this exchange. Collegial methods focus solely on the "good" solutions, or if a memory mechanism is present, extract solutions from there.

More advanced designs are Knowledge Collegial and feature additional functionality for the creation of new information based on exchanged solutions. Asynchronous strategies often use a "memory pool" (or "blackboard", "data warehouse", etc.), a shared solution repository. Using the pool simply as an intermediary storage is perhaps only Collegial, but adding filters, aggregation, learning or other processing makes the approach more and more Knowledge Collegial.

Finally, the third dimension of cooperative search is *search differentiation*, which also a divides in four based on search origin and the search portfolio. Cooperative search can begin from a single shared point or population (SP) or from multiple initial points (MP) at the same time. The search strategy portfolio can consist of multiple different strategies (DS) or just a collection of the same one (SS). In total, we have four alternatives – $SPSS$, $SPDS$, $MPSS$, and $MPDS$.

Beyond basic cooperation, some more advanced approaches to cooperative search have been proposed and explored. *Multi-level cooperative search* [127] takes a more controlled view to cooperation by relying on layered information diffusion, much like in hierarchical neural networks. Here, a single AP works on the original problem, with others positioned on different levels of abstraction and communicating strictly with their adjacent levels. *Hyper-heuristic* methods have also been successfully combined with cooperative search, with applications in domain-independent planning [98].

2.7 Agents and multi-agent systems

In the AI tradition the principal objects under study are *agents* [115]. An agent is an entity that acts – and does so in a way that makes the concept of a distinct subject worthwhile. Encapsulating functionality into objects with identity can make reasoning about systems more intuitive and approachable. Agents are a natural notion for research that seeks to produce results to be eventually put to

use in robots and other embodied systems, but applies to general software as well.

Agents can be defined in many ways and different characterizations make sense in different applications. Perhaps the one fundamental aspect of all agents lies in their relation to the environment in which they operate. Agents interact with the environment by receiving percepts, perceptual input, and acting or reacting accordingly. Agent behavior can also be viewed as a function, with each possible percept sequence mapped to an action, giving rise to the notion of an algorithmic agent (or software agent) and agent-oriented programming [93, 120].

Agents model behavior and their properties are dependent on the system being modeled. Fundamentally, agents can be seen as delegates to whom system designers give a portion of the overall control. If only simple behavior is desired, less cognitive capacity is needed in the agents, but as task complexity grows, more elaborate engineering is called for. Some of the more relevant agent properties from an algorithmic cooperation point of view are presented below, compiled from [99, 115, 132]:

- **Intelligence**: Capacity for meaningful interaction. Percepts are received from the environment and actions are performed based on that.
- **Rationality**: Capacity to pursue goals. All activity is directed towards achieving goals, "doing the right thing".
- **Autonomy**: Authority over own behavior up to self-awareness. No separate user. The ability to self-modify the way in which objectives are achieved.
- **Cognitivity**: Capacity to learn and improve through experience.
- **Contextuality**: Capacity for situational assessment. Ability to not only react, but to reason about events.
- **Complexity**: Degree of behavioral sophistication ranging from idle repetition to deep cognitive processes and nuanced actions.
- **Locality**: Sense of location in the environment. Agents interact only with their immediate environment. No designated global controller or omniscience.

After agents themselves, the main component in any agent-based system is *the environment*. Agents can be understood to be coupled with their environment: actions make no sense without the environment in which they take place. This separates agent systems from expert systems, where a computing process simply reasons about data [132]. In computing, the objective is to find a solution to a given problem, which involves framing the problem in computational terms. With agents this configuration is known as the *task environment*.

While environment has a very physical connotation, the idea is present in even the most abstract settings. For example, all communication is defined with respect to the environment, even if the environment is simply a message passing topology. Software agents enjoy environments that on one hand are direct, rich and unlimited, but on the other hand are disconnected from the physical world.

Main dichotomies of environment attributes include [115, 116, 129]:

- **Real vs. virtual**: Real environments are those that humans interact with directly, while virtual environments are the private domain of artificial things.
- **Discrete vs. continuous**: A discrete environment has a natural notion of isolated units, such as squares in a grid, while a continuous environment operates on a uniform scale. This applies to time and state and the way percepts and actions take place.
- **Accessibility vs. inaccessibility**: Agents can perceive the whole environment and can collect a complete picture of the state of the environment, or the environment is observable only partially or not at all.
- **Dimensional vs. ephemeral**: Agents have a location and take space in the environment, or their presence is more abstract and transient. For example, dimensional agents might be able to collide with one another, while ephemeral agents must rely on abstract interaction.
- **Deterministic vs. stochastic/nondeterministic**: Deterministic environments have definite causality: the next state is fully defined by current state and pending agent action. Stochastic environments have no such guarantee, as outcomes are quantified in terms of probabilities. Nondeterministic environments have a set of possible outcomes, but no associated probabilities.
- **Episodic vs. sequential**: Agents experience the environment in episodes that are not dependent on each other, or decisions made by agents have an impact on how the environment presents itself later on.
- **Dynamic vs. static**: Environment changes during agent deliberation, or not.
- **Known vs. unknown**: The agent is or is not aware of the nature of the environment. For example, an agent that knows the rules of a game is in a different position compared to another that does not – regardless of the information they receive about the current match.

The real power of the agent abstraction comes to fore only when the environment is not dedicated to a single agent, but open to a population of agents. These

multi-agent systems (MAS) [121, 132] feature agents that are capable of independent action, but can also interact and work together as a group to do more than what comes naturally to one. Multi-agent systems have varied applications in the real world ranging from games and graphics to logistics and networking, and can manifest highly desirable properties such as dynamic load balancing, fault-tolerance, self-organization and scalability.

The general discipline of MAS, and specifically *cooperating multi-agent systems* (CoMAS) [99], is a good match to cooperation-based parallel search. The broad field of *computational intelligence* [38] overlaps with multi-agent systems, featuring nature-inspired soft computational methodologies such as artificial neural networks, evolutionary computation, and fuzzy logic systems. Bio-mimicry, or bionics, the application of mechanisms found in biological systems to engineering and technology, has been a treasure trove for various algorithmic ideas, often featuring agents.

Much of the recent work on agent systems is application driven. Robocup[5] is a notable initiative that aims to advance the state of the art in intelligent mobile robotics through competitions in competitive soccer and emergency rescue scenarios. Many conferences in the AI, machine learning and computational methods disciplines welcome multi-agent topics; Panait and Luke [99] give good pointers.

[5] `http://www.robocup.org/`

Chapter 3

A! – Cooperative heuristic search

Imagine a large derelict mansion and a small object hidden within it. If you were to look for it alone, you would probably first search through one room, then another, and so on. Things would be a bit different if you had friends to help you. You could split up and search different rooms, or focus on one of the rooms and split the area within it. You cooperate, but only in deciding how to divide the common task.

Now imagine further that the hidden object had a special property that made it roughly detectable when it was not fully found yet. It could be anything from a smell to a deep, low-pitch sound, but it would intensify when you got closer and quickly fade out over a distance. Suppose that the hidden object had a "smell" like this – how would you search for the object now?

Alone you would again go room by room, but this time you would only explore the entire room if your nose gave you a good reason. You would simply pop in each one of the rooms enough to get a good sniff and then decide to either move on or have a closer look.

With friends you can again split the common task. If rooms are divided evenly, the one with a smell can be found faster, but the cooperation need not end there. The one who caught the smell can call the others and you can all focus your efforts on that one room. You cooperate not only in dividing up the work, but also by passing amongst you information about your progress.

Determining the minimum cost path through a graph is a classic search problem that can readily be found in various application domains from navigation to circuit board layout. The task of finding an optimal path by searching through a complete search space of all possible paths grows exponentially in size and quickly becomes infeasible. However, if the target problem admits a heuristic, a suitable search focusing function, the search effort can be greatly sped up. Heuristic search employs the heuristic function to direct the search towards regions of the search space that are likely to lead to good solutions.

This chapter presents a cooperation variant of the standard heuristic search algorithm A*. We first review the operation of vanilla A* and then continue to develop a concurrent, agent-based cooperation mechanism on top of it. The third section presents the complete algorithm, dubbed A! search. The final section offers a discussion on implementation issues and the tradeoffs present in this approach.

3.1 Vanilla A*

3.1.1 Algorithm details

Formalized in the late 1960s [53] and in diverse use today, A* is the most widely known form of heuristic search. A* is an informed best-first search algorithm, where additional knowledge of the target problem is encoded into the search as a heuristic function. Chapter 3 of the standard AI textbook by Russell and Norvig [115] presents an overview of A* and its many variants, summarized below.

A* can be viewed as an extension of Dijkstra's graph algorithm [35] for single-source shortest path search. An evaluation function $f(u)$ is used to decide which node u of the search space is explored next. The evaluation function reflects a cost estimate in the search domain, so the node with the lowest estimated value is selected first. The candidate node list is maintained as a priority queue prioritized on the estimated value and updated as the search proceeds through the graph.

While Dijkstra's algorithm follows a plain selection policy based on the effective cost to reach a given node, $g(u)$,

$$f(u) = g(u),$$

A* includes a complementing heuristic component, $h(u)$,

$$f(u) = g(u) + h(u),$$

that reflects the estimated cost from the state in node u to a goal state. We can view $f(u)$ in A* as the estimated cost of a solution passing through node u: the first component is the length of the best known path to u and the second reflects

our understanding of the remaining distance. Note that a null heuristic, $h(u) = 0$, yields a search that is equivalent to Dijkstra's.

Algorithm 1 gives the pseudocode for A*. The three data structures `openHeap`, `closedSet` and `pathMap` manage the estimated cost priority queue, visited nodes and node–node path successor relation, respectively. Nodes are explored one at a time in min-heap order until a solution is found or the search space is exhausted. As in Dijkstra's algorithm, the cost estimates in the priority queue get improved all through the search as states get explored.

If a goal node is found, the solution path is constructed in reverse from the `pathMap`, abstracted here into `derivePath`. Otherwise, the search advances by adding unseen neighbors of the `current` node to the heap and by updating cost estimates for the nodes already in the heap, if shorter paths have been discovered.

A* is optimal in the sense that it always finds the shortest path, if given a heuristic function $h(u)$ that satisfies certain properties. First, $h(u)$ must be *admissible* in that it never overestimates the cost to reach the goal state. As $g(u)$, the cost to reach u, is known once u is visited, this ensures that the heuristic never overestimates the true cost of a solution, which would defeat optimality. Admissible heuristics are in a sense optimistic, as the cost of solving a problem is estimated to be no greater than it actually is.

Admissible heuristics are sufficient to guarantee optimality for tree-structured search spaces, but a general graph search requires a slightly stronger property, *consistency* (or *monotonicity*). A heuristic $h(u)$ is consistent if for every node u and every successor v of u reachable by any action a, the estimated cost of reaching the goal from u is no greater than the combined cost of getting to v and reaching the goal from there, a kind of a triangle inequality:

$$h(u) \leq c(u, a, v) + h(v).$$

If $h(u)$ is consistent, the values of $f(u)$ along any path are nondecreasing: if v is a successor of u, then $g(v) = g(u) + c(u, a, v)$, and

$$f(v) = g(v) + h(v) = g(u) + c(u, a, v) + h(v) \geq g(u) + h(u) = f(u).$$

When A* selects a node for expansion, the optimal path to that node has already been found. By monotonicity, the conflicting node would already have been selected first. Given these properties for $h(u)$, the sequence of nodes expanded in A* is in nondecreasing order of $f(u)$ and the first goal node selected for expansion must yield an optimal solution – all later (goal) nodes will be at least as expensive.

The optimality of the solution discovered by A* does not mean that A* is in a runtime sense an optimal algorithm for finding it. Still, as a best-first search, A* is also *optimally efficient* up to tiebreaks among equally valued nodes: given the same heuristic, no algorithm can expand fewer nodes. We return to matter of tiebreaking in our cooperative algorithm.

Algorithm 1 : **Vanilla A* Search**

Require: NODE $start$, PREDICATE $isGoal$, HEURISTIC h
Ensure: $path$ from $start$ to nearest node satisfying $isGoal$ is shortest possible

```
openHeap ← FibonacciHeap⟨INTEGER, NODE⟩
closedSet ← Set⟨NODE⟩
pathMap ← Map⟨NODE, NODE⟩
openHeap.insert({0, start})
repeat
  current ← openHeap.pop()
  if isGoal(current) then
    return path ← derivePath(current, start, pathMap)
  end if

  for each u in current.getNeighbors() do
    if closedSet.contains(u) then
      continue
    end if
    g ← current.g + dist(current, u)
    f ← g + h(u)
    improved ← openHeap.update(u, f)
    if improved then
      pathMap.update(u, current)
    end if
  end for

  closedSet.add(current)
until openHeap.isEmpty()
raise failure
```

3.1.2 Variants and related work

A number of improvements to the vanilla A* algorithm have been developed over the years. Due to the substantial memory footprint of A*, memory-bounded variants, such as *iterative-deepening A** (*IDA**) [78] and *simplified memory-bounded A** (SMA) [114], have found use in many practical applications.

Pathfinding in a partially visible graph – a common task in mobile robotics – requires an incremental, version of A*, such as *Lifelong Planning A** (LPA*) [76] or *dynamic A** (*D**) [123]. Heuristic search ideas are applicable to a bidirectional search setting as well [115].

A huge body of work exists on search heuristics, Pearl's book [101] being the

definitive treatise. The admissibility constraint of the heuristic function can be relaxed at the expense of optimality, but with substantial gains in search speed. Several A* derived algorithms have been shown to be ϵ-*admissible* in that the solution path is no worse than $(1+\epsilon)$ times the optimal solution. These include various static [101] and dynamic [105] weighting approaches, and focused selection methods, such as the two-heuristic A^*_ϵ [102].

Domain-independent heuristics feature prominently in the related field of classical planning [115], the problems in which are often used for heuristic search benchmarking purposes [11, 72]. The broad field of *metaheuristics* [12] explores methods such as heuristic selection, generation and learning. Problem-specific *pattern databases* [33] offer a way to speed up search by often several orders of magnitude, making the solution of whole new instance categories feasible [115].

Finally, on the theoretical side A* and its descendants pose considerable analytical challenges as the time complexity of the algorithm depends on the heuristic being applied. Pearl [101] is the definitive textbook for heuristic search; Russell and Norvig [115] present a more recent view. Farreny [42] offers some theoretical foundation for heuristic search by proving general properties of completeness and (sub-)admissibility for a range of A*-inspired algorithms.

Parallel variants of A* are discussed in Chapter 5.

3.2 Constructing cooperative search

In this section we develop an agent-based cooperation mechanism for use in heuristic search. We begin with agents and a multi-agent system formulation of search and continue with the communication dimension of cooperation. We introduce actors as the natural agent abstraction and consider some of the challenges that face agents searching together. The ideas presented here are used in the next section to establish a cooperative heuristic search algorithm.

3.2.1 Cooperating agents in multi-agent search

Cooperation begins with those who cooperate. We wish to have a mechanism that enables information sharing among concurrently operating search workers. This is a natural multi-agent system and we will proceed by defining one using the nomenclature of multi-agent systems from Section 2.7.

Heuristic search can become computationally intensive in many ways. For example, calculating the heuristic values can be costly, the memory footprint can grow exponentially with the search space, and the main data structure – the priority queue – can quickly become a bottleneck. The idea in cooperation is to direct and focus the search by sharing information, which in a sense aims to reduce the

work done by a single worker. We wish to have a cooperation mechanism that is straightforward and lightweight enough for this extra effort to be worth it.

An A* worker is a simple agent. It receives information about the search graph one node at a time as the nodes get visited, and proceeds to visit more as soon as the search progress is reflected in the open and closed lists. The main percept in A* is the set of new nodes in the frontier, including their heuristic values, found by extending the search to an unopened node. The main action is the maintenance of the internal data structures – search bookkeeping.

The task in A* is to find a node that satisfies a goal defining predicate and to remember the path taken to reach it. The heuristic function directs the search towards the goal and we aim to use cooperation to further accelerate this process. A* workers are intelligent agents in that they turn their internally managed state into actions, and rational in the sense that all these actions are goal-oriented.

A* agents cannot be considered highly autonomous or cognitive. While there is no central controller and the agents are free to keep running their search without interrupt, the search procedure itself is fixed and typically has no control parameters. No behavior changing learning takes place and all the percepts are treated equally, regardless of their order. A* agents make little use of their operational context, though the mutable priority queue reflects search progress, and the predecessor structure both in a way record history.

On the complexity scale A* agents are more in idle repetition. In vanilla A*, no use of locality is made beyond the heuristic flavor in search. For example, to appreciate the difference, incorporating some stochastic elements into the search in a simulated annealing [115] fashion would increase locality, contextuality and complexity – and perhaps search performance. The new search would be more intelligent as it would explore the search space in a more adventurous way, throttling this process as the search progresses.

We can describe the environment in a vanilla A* system again using terminology from Section 2.7. The A* environment is virtual, discrete, accessible, ephemeral, deterministic, sequential, static and known. The graph underlying the search, while possibly implicit, is fixed and fully available if connected. A* agents have a position in their search – current node in the graph – and a rudimentary sense of direction. With only a single agent the system is fully deterministic.

On the other hand with multiple copies of the agents and meaningful interaction between them, we have a multi-agent system. Adding cooperation into the search adds a new percept, the shared information, and a corresponding action, the application of this information into the search state.

Compared to a single-agent system, the environment in a multi-agent search system sees a fundamental change. As the agents search and share information, the collective search effort exhibits concurrency if messaging is asynchronous and

no control is imposed. The environment becomes nondeterministic, as executions interleave in unpredictable ways, and dynamic, as the order in which information arrives and gets processed has an effect on the search agents.

3.2.2 Cooperation is communication

The hypothesis underlying this thesis is that cooperating agents outperform agents in isolation. The difference lies in the relationship the agents have with the environment: the percepts that agents receive and the actions they then execute. In a search context with software agents, this notion is made concrete in the communication policy: cooperation is communication, and isolation the complete lack thereof. Cooperating agents produce and consume shared information, isolated agents keep to themselves.

We can identify a suitable cooperation plan by following terminology from Crainic et al. [30, 31], outlined in Section 2.5. We seek to explore the effect of concurrency and unconstrained interaction, so instead of a rigid master-slave approach, we want the overall global search to be managed collegially by several processes (p-control, pC).

We do away with superfluous synchronization, and aim to simply augment the vanilla A* search in an unobtrusive way (Collegial communication, C). In this work we focus on revealing concurrent phenomena, so we initially employ the same search strategy uniformly, but multiple strategies could also be experimented with. A* is point-initialized by nature, so our search differentiation policy is 'Same point, same strategy' ($SPSS$). The full cooperation policy – emphasizing autonomy, concurrency and simplicity – then becomes $pC/C/SPSS$.

Details matter for performance, and all the more so when it comes to concurrent cooperation. A truism in the parallel computing community goes "There is no parallelism without tears" [103], referring to the extreme difficulty of having high-level programming constructs work well with a variety of low-level hardware details. Squeezing the last bit of parallel performance requires a deep understanding of the target problem and the whole computing system, and efficient cooperation is no exception.

Effective communication designs for cooperative search depend on the underlying hardware, for example, some platforms are good at synchronizing executions, while others excel in all-to-all message broadcasts. Therefore finding optimal general purpose communication strategies for cooperative search is beyond the scope of this work. We simply wish to augment A* with basic cooperation mechanisms and set the stage for concurrent phenomena.

We will make use of Crainic and Toulouse's six cooperation design issues [31] from Section 2.5, but focus on aspects that are relevant to cooperation in A^*. We

can extract three dimensions of cooperative communication: content ("What?"), method ("How? Where? When?"), and pattern ("By whom? To whom?").

The *content* of messages passed around in cooperation is naturally problem dependent and can take many shapes. A* is a graph search looking for the shortest path, so a partial solution would be a string of nodes (or a node and transformations) and a current best solution a complete path from a start node to a goal node. Another approach would be to share history, recently visited states, or some other set of interesting nodes.

As nodes per unit of time is an important metric in search computation, heavy communication costs, including message processing, are an issue. Vanilla A* is known to suffer from a substantial memory footprint [115], so small payload messages are in order. For example, node identity is typically concisely represented and easily serialized for distribution. On the other hand moving whole open/closed lists is unmanageable. Functional data structures with persistence and memory mapping [97] offer some interesting possibilities for cooperation mechanisms.

With the focus being on concurrency rather than optimal cooperation, we opt for simplicity. A* is fundamentally about heuristic-driven exploration, so we want search workers to help one another – and themselves – in finding the most auspicious nodes and paths quickly. One viable option is for workers to simply share the best nodes and heuristic values as they encounter them, and use this information to direct the search effort as it emerges. This is our approach in this work: we encapsulate shared best node information into a secondary heuristic. We elaborate on this in Section 3.3.

The *method* of communication matters as well. A number of reports argue in favor of asynchronous messaging over synchronization [11, 23, 31, 66, 72, 98]. Crainic and Toulouse [31] state on cooperation messaging modes: "Compared to independent and most asynchronous strategies, synchronous cooperative methods display larger computational overheads, appear less reactive to the evolution of the global parallel search, and conduct to the premature convergence of the associated dynamic process."

With asynchronous messaging, timing is less important from a blocking perspective, but is still essential in establishing "the ripple effects" of concurrency. As progress information trickles down to individual workers, the A* paths begin to meander within their local environment, still staying cohesive globally. Cooperation emerges from this concurrent interaction and path diversity.

Successful cooperative strategies are built on controlled, parsimonious, and timely exchanges of meaningful information. Communication can be divided into direct and indirect modes, where the former consists of explicit messaging between agents and the latter emphasizes the role of the environment. Direct messaging ranges from one-to-one to broadcasting, with various information diffusion topolo-

gies – such as meshes and tori – in between. These local-sharing topologies are designed to reflect the communication channels in the underlying hardware.

Indirect messaging is based on some form of memory external to the agents and may also include computational elements. If the agent system operates in a nontrivial environment, some metadata can be attached to messages to further improve their usefulness. Abstract and simple "data pools" feature in some implementations, serving as a central messaging source and target.

For cooperative heuristic search, one could either A) carefully maintain disjoint search space division and have the searching agents know their neighborhood, or B) let the agents operate more freely and share information gregariously. The former has been explored extensively in the past [72], but the second – much more in tune with concurrent phenomena – has received less attention. We explore option B in this thesis and review some A approaches briefly in Chapter 5.

Finally, the *pattern* dimension involves the overall messaging protocol. All workers can consume and contribute progress information and a straightforward way to do this is to have each broadcast what they know. An even simpler and more efficient way to diffuse information is to follow a *publish-subscribe model* [29, 40], where a unique address is used as a messaging target and all subscribers receive messages from a broker entity, much like in a web chat.

A publish-subscribe model is a natural choice in distributed systems, but works well also locally. While the outlined cooperation scheme can be implemented as a distributed multi-agent system, local configurations enable fast, asynchronous interaction through shared memory and multi-threading. While issues of connectivity and other hardware specific details are important, they are mostly out of scope in this study. The actor abstraction that makes reasonable use of local resources is discusses briefly in the next section.

3.2.3 Actors as minimal cooperating agents

The multi-agent system and its cooperation policy outlined in this chapter point towards lightweight software components and simple communication mechanisms. The crux of cooperative search lies with efficient asynchrony, which, together with multi-threading and interleaved execution, gives rise to concurrency.

Martin [89] argues that there are essentially two views to concurrency: *concurrent programming*, whose proponents believe that *actions* are inherently concurrent, and *concurrency control*, where concurrent access to *data* is key. The concurrent programming tradition begins with Hoare's communicating sequential processes [59] and actors [1] and currently enjoys a revival in modern programming languages, such as Scala, Rust, Go and Haskell. Concurrency control also has a long history from early database systems to atomic transactions, transactional memory and other constructs available to programmers today.

Concurrent programming leaves interaction and synchronization management to programmers, offering mostly just primitives, while concurrency control tries to establish some order by abstracting away most of the issues in execution interleaving. We focus on the former, as the aim of this thesis is to explore the possibilities of free execution entanglement rather than neat coexistence.

We require three kinds of functionality from the agents participating in cooperative search:

- the search itself, including the heuristic, data structures and goal predicate;
- asynchronous messaging functionality, non-blocking send and receive; and
- a shared information utilization mechanism, "the secondary heuristic".

Cooperative search is straightforward to implement on top of existing A* programs by first extracting the search routine into a self-contained worker construct. Actors [1, 93] are an excellent abstraction for this and readily available as part of many languages or as a dedicated library. Various message queue primitives get the asynchronous messaging job done as well, and some even find them preferable to actor abstractions [57]. Both actors and plain message queues are suitable for broadcasting-based communication and publish-subscribe information diffusion and even more complex topologies than is needed here.

Additional benefits of the actor abstraction include strong encapsulation and simple interaction between actors of all kinds. While identical actors are suitable for examining the issues of interest in this thesis, some performance gains can be obtained from portfolio-based algorithms, where a population of different actors cooperate [7]. Even actors of the same kind can exhibit divergent behavior and take on roles dynamically [96], if equipped with enough cognitive capability.

3.2.4 Challenges in searching together

As a graph-based algorithm, A* is hard to parallelize. Kishimoto et al. [72] note that three different kinds of sources of overhead make efficient implementations difficult to realize. *Search overhead* occurs when the parallel version expands more nodes than the sequential one, arising from non-disjoint search space division. Division schemes also suffer from short-sightedness, as a found solution cannot be guaranteed to be optimal. *Synchronization overhead* refers to the idle time wasted at synchronization points, such as locks on shared data. *Communication overhead* occurs when information is exchanged in a distributed setting.

Reducing these overheads is challenging because of interdependencies. For example, reducing search overhead can increase communication costs. With an agent-based concurrency-oriented approach as previously described, we choose to

suffer some search and communication overhead, and gain in minimal synchronization needs. The rationale behind this setup lies with the resources locally available in modern computers: many cores, fast interconnects and plenty of memory.

Careful partitioning of the search space in a shared memory setting typically leads to *work stealing* designs, where processors maintain local work queues and even out misbalanced division of the total effort by acquiring more work from their neighbors. In A*, this further leads to either centralized open-list strategies, where concurrent access is a major issue, or decentralized ones with load balancing, revisiting and duplicate detection issues. Kishimoto [72] et al. give an overview of the various methods that have been tried to address these issues.

Not all parallelization mechanisms are based on search-space partitioning. One can parallelize the computation performed on each state, but this is less useful if the problem has naturally very simple nodes. Running a different search algorithm on each processor gives rise to portfolio algorithms [65], which have been successful in, e.g., SAT competitions [7, 128].

Another approach is to follow a parallel window search policy [106], a variant of *IDA** where each processor is searching from the same root, but with a different bound, i.e., different iterations of *IDA** are run in parallel. Dynamic approaches are also possible: the EUREKA system [27] uses machine learning techniques to automatically configure parallel *IDA** for various problems.

The next section introduces our cooperative heuristic search algorithm that offers a concurrent take on searching in parallel.

3.3 The A! algorithm

We are now ready to discuss our cooperative search algorithm, A!. We begin with an overview of the algorithm and then show that A! maintains the optimality of A*. We then proceed with a description of the cooperation architecture and give a complete pseudocode representation of the algorithm. Variants of A! are discussed briefly at the end of the section.

3.3.1 Overview

The A! (*a-bang*)[1] 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 primary and secondary

[1]The exclamation mark is not only present in the logotype of the author's university, but also denotes message transmission in CSP [59] and its descendants.

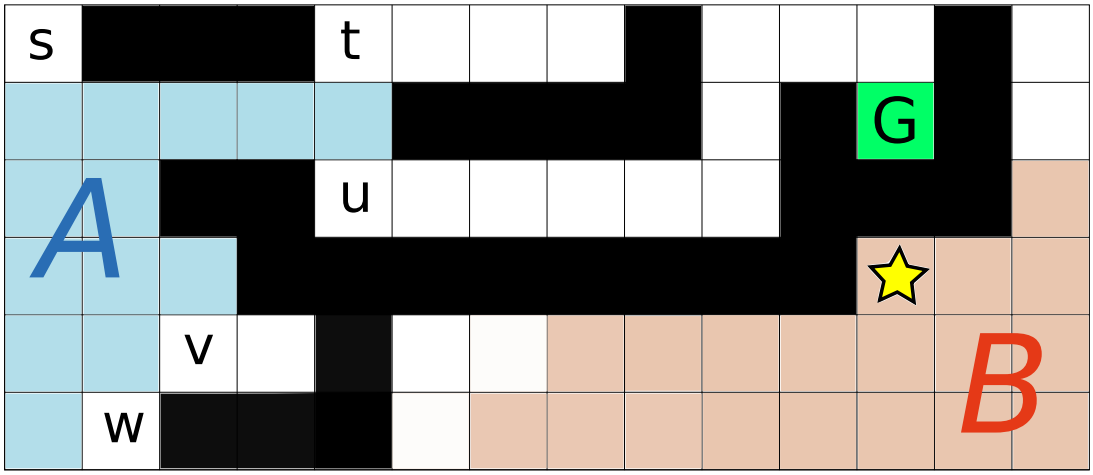


Figure 3.1: A! search on a grid graph with two agents, A and B. A chooses between graph nodes t and u, both with equal estimates for the shortest distance (*1st heuristic*) to goal G. In vanilla A* the decision is arbitrary. In A!, A will choose u next, because it has a lower distance (*2nd heuristic*) to the best node that it has heard of, the starry one discovered by B.

heuristic functions – 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 the message traffic flow.

The primary heuristic is the one used in 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! chooses among up to k best nodes of equal value based on information acquired during the search.

Figure 3.1 illustrates the overall A! concept, showing A! with two agents sharing information about best encountered nodes. Agent A is able to use information provided by B to prefer one of the equally valued alternatives based on the secondary heuristic. The primary heuristic guarantees optimality.

The primary heuristic can reflect the secondary one, $h(u) = \hat{h}(u,g)$, for all nodes u and goal node g, but this is not necessary. For example in Figure 3.1,

we have A! executing over locations in a grid graph with Manhattan distance, the length of the direct path along the grid without obstacles, as both the primary and secondary heuristic.

As discussed in Section 3.2.2, one simple, effective choice for information to share is the best encountered node. This directly implies a distance-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 if agents track not just information shared by others, but their own progress as well, the degenerate case of A! with a single agent is *not* necessarily vanilla A*. With even one agent, passing progress information as a parameter to the secondary heuristic function results in 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.

3.3.2 Optimality

Some heuristics have degenerate corner cases where they perform poorly by giving overly optimistic, low estimates. This can be remedied to some extent by using multiple competitive heuristics and selecting, e.g., the maximum or average value. If two heuristics are independent from one another, two heuristic estimates can also be combined by adding them together, but this is generally not the case: the sum of two cost estimates can very well be greater than the real remaining distance. This breaks heuristic monotonicity and the optimality guarantee.

A! uses two heuristics in concert, but does not add them together. A! maintains optimality – finds the shortest path – if the primary heuristic is monotonous. We give a straightforward proof of the optimality of A! following the argument for the optimality of A* itself given by Russell and Norvig [115].

Theorem (***Optimality of A!***)**.** *We claim that A! is optimal. This follows from two facts: a) the values of $f(u) = g(u) + h(u)$ along any path are nondecreasing, and b) whenever A! selects a node for expansion, the optimal path to that node has been found. Let S be a subset of all equally good next-to-open candidate nodes.*

First, for all u and their included successors $v \in S$, $g(v) = g(u) + c(u, a, v)$ for some action a and cost function c. By definition of S, $f(v)$ is the same for all $v \in S$, while $g(v)$ and $h(v)$ can vary. As in single-source A, we have*

$$f(v) = g(v) + h(v) = g(u) + c(u, a, v) + h(v) \geq +h(u) = f(u),$$

for all v and u, so no matter what node is chosen from the set, the values of $f(u)$ along all resulting paths is nondecreasing.

Second, if A! would select for expansion a node to which an optimal path has not been found, there would have to be another frontier node w on the optimal path from start to node v. Because f is nondecreasing along any path, by the systematic frontier expansion property of A, w would have a lower f-cost than v and would have been selected first. This holds for all $v \in S$ independently. But then we have both $f(v) < f(w)$ and $f(w) < f(v)$ – contradiction.*

Therefore, any node $v \in S$ selected as the highest ranking node according to $\hat{h}(v, b)$ maintains the nondecreasing order of f, and so on the discovery of the first goal node g, the optimal path has been found.

Corollary (***Generalized A!***)**.** *By the previous argument, any ranking function can be used in selecting among the nodes in S.*

3.3.3 Cooperation architecture

From Section 3.2 we know that asynchronous communication between agents is at the heart of cooperative search. The workers each run their own upgraded A* search, but share their progress as they explore the search space. The agents send and receive information that directs and diversifies the paths they explore in a nondeterministic way. Simply by heeding these hints at their own pace, the agents as a collective demonstrate algorithmic randomness that is not explicitly encoded into the algorithm, but rather emerges implicitly as the search progresses.

The core of the cooperation mechanism in A! is a shared, best-effort notion of a globally superior node seen in the graph by any of the agents. If an agent encounters a node that has a better cost value, $g(u) + h(u)$, than what the agent considers to be the current best, the others agents are informed about the node. The message broker entity can lie here in-between the agents to facilitate simple communication interfaces through centralized channels while offering other benefits of indirect communication, such as filtering.

Note that instantaneous information diffusion and integration is *not* the primary goal: delay and disagreement leads to diversity, as agents can – even from the same state – proceed to explore different parts of the search space. This is vital especially in the beginning of the search. On the contrary, to further accentuate this effect, acceptable best node improvement can be set to zero, or even time-dependently at-random negative as in simulated annealing. Short-circuiting the best-update mechanism by directly updating the best with the current adds a *momentum effect* that manifests itself already with a single agent: the ongoing search prefers nodes near the previously explored ones.

Because we want each agent to be able to pass information to every other agent, we have a kind of a chat layout for messaging, which is well served with a

publish-subscribe design pattern [40]. This is a very scalable design for distributed systems, but works equally well in a local setting.

Finally, program termination on path discovery can take place through a special message over the standard messaging channel, but as a one-off event happening in a local setting, a shared termination variable can also be used as an expedient way to conclude all search effort.

The actor programming abstraction encapsulates the upgraded A* search of A! into a main worker routine and enables participation in the publish-subscribe scheme through asynchronous messaging primitives. The agents explore the search space, share information about their progress and make use of the progress of others. We continue with a more concise description of the algorithm.

3.3.4 Algorithm details

We present the entire A! algorithm as a collection of pseudocode snippets, beginning with Algorithm 2, which serves as the main body of the algorithm. The algorithm essentially just launches search workers and waits for one of them to finish. A message broker entity takes care of the publish-subscribe communication scheme. The message broker allows for the workers to interact with one another through simple port interfaces.[2]

Algorithm 2 : **A!Search**

Require: $N > 0$, NODE *start*, PREDICATE *isGoal*, HEURISTIC h, HEURISTIC $\hat{h}$
Ensure: *path* from *start* to nearest node satisfying *isGoal* is shortest possible

```
mb ← MsgBroker()
for i = 0 to N do
  workers[i] ← A!Solver(mb.portOut, mb.portIn, start, isGoal, h, ĥ)
end for
for each worker in workers in parallel do
  worker.launch()
end for

wait for termination
return path ← getPath(workers)
```

Given the problem instance, including the *two* heuristic functions discussed in Section 3.3.1, the algorithm returns a path from the start node to the found goal

[2]The port abstraction is in line with how, e.g., the ZeroMQ library [58] presents publish-subscribe interfaces.

Algorithm 3 : **A!Solver**

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

```
openHeap ← FibonacciHeap⟨INTEGER, NODE⟩
closedSet ← Set⟨NODE⟩
pathMap ← Map⟨NODE, NODE⟩

current ← start
repeat
  if isGoal(current) then
    terminate(current, start, pathMap)
  end if

  closedSet.add(current)

  for each u in current.getNeighbors() do
    if closedSet.contains(u) then
      continue
    end if
    g ← current.g + dist(current, u)
    f ← g + h(u)
    improved ← openHeap.update(u, f)
    if improved then
      pathMap.update(u, current)
    end if
  end for

  peekList ← openHeap.getPeekList()
  if isEmpty(peekList) then
    terminate()
  end if

  current ← A!Select(peekList, portIn, portOut, h, ĥ)
  openHeap.remove(current)

until termination
```

Algorithm 4 : **A!Select**

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

```
update, updateH ← asyncRecv(portIn)
if updateH < bestH then
    best, bestH ← update, updateH
end if
select ← peekList.pop()
selectD ← ĥ(select, best)
for each u in peekList do
    d ← ĥ(u, best)
    if d < selectD then
        select, selectD ← u, d
    end if
end for
if h(select) < bestH then
    best, bestH ← select, selectH
    asyncSend(portOut, {best, bestH})
end if
return select
```

node, if one is reached. The path is derived from a path map of successor nodes by the worker that finds a goal first, and then forwarded on termination to the main function. This mechanism is abstracted here into `getPath(..)`.

The workers run `A!Solver`, the procedure outlined in Algorithm 3, which has four parts inside a loop that is repeated until program termination. The first part 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, pass it forward and terminate. This process is abstracted into `terminate(..)`.

The second part is the A* expansion. We use the 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 order by estimated total cost. The `pathMap` maps nodes to one another, establishing the successor relation used in solution path derivation.

The third part is a cursory peek into the up-to-date `openHeap`. The idea in A! is to obtain a list of interesting nodes and be smart about selecting among them, whereas in vanilla A* the routine simply draws one from the top. The peek is a bounded traversal of the Fibonacci heap, where we build a list of nodes that share the cost of the top node. The `peekList` therefore contains nodes that are equally good in the sense that they share the same estimated total cost.

Algorithm 5 : **MsgBroker**

```
best, bestH ← NONE, MAXINT
repeat
  update, updateH ← blockRecv(portIn)
  if updateH < bestH then
    best, bestH ← update, updateH
    publish(portOut, best, besth)
  end if
until termination
```

The final part contains the selection routine, which for A! is given as Algorithm 4. After one of the nodes has been selected – in one way or another – it is removed from `openHeap` and turned into `current`. Node removal from a Fibonacci heap is a relatively fast operation, thanks to the `decreaseKey` functionality.

The selection routine `A!Select` features the real core of the A! algorithm: the cooperation functionality and the application of the secondary heuristic. The first part begins the cooperation routine `asyncRecv`, that brings new information into the agent. The read is asynchronous in that if there is no message, the algorithm proceeds without any delay. The routine shown in Algorithm 4 gives a version with best-information being shared and aggregated, but other cooperation schemes can naturally also be constructed here.

The second part features the inclusion of the new information, as encapsulated into the secondary heuristic function. The `peekList` is essentially sorted on $\hat{h}$, the second heuristic, and the highest ranking node is then selected. The routine concludes with the mirror procedure of the first one: 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.

To complete the tour of A! , Algorithm 5 gives the routine used by the message broker entity to manage traffic. The MsgBroker actor waits for updates and then distributes them through the publication mechanism, if progress has been made. More elaborate cooperation schemes could be implemented in much the same way.

Alternative selection policies to `A!Select` include the random selection routine (Algorithm 6), and vanilla selection (Algorithm 7), which turns the algorithm into a convoluted version of the vanilla A*. Hybrid policies, such as Algorithm 8, are of course possible. The selection policies are explored and compared in the experiments presented in Chapter 4.

Algorithm 6 : **A?Select**

Require: LIST *peekList*
Ensure: *select* is a random node drawn from *peekList*
  return *select* ← *Random.choice*(*peekList*)

Algorithm 7 : **A*Select**

Require: LIST *peekList*
Ensure: *select* is the node in *peekList* that would be chosen in vanilla A*
  return *select* ← *peekList.head*()

Algorithm 8 : **A"Select**

Require: LIST *peekList*, PORT *portIn*, *portOut*, HEURISTIC h, $\hat{h}$, PROBABILITY p
Ensure: we use **A!Select** with a probability of p, **A?Select** with $1-p$
  if $Random.nextFloat(0,1) < p$ **then**
    return $A!Select(peekList, portIn, portOut, h, \hat{h})$
  else
    return $A?Select(peekList)$
  end if

3.4 Tradeoffs and implementation challenges

Implementing A! presents challenges beyond the complexities of the algorithm itself. In this section we consider some of the practical issues that arise in implementing A!. These include memory locality, search encapsulation, worker interaction, memory usage, communication patterns, data structures, knowledge representation and concurrency support in programming languages.

Locality is a major design decision in parallel computing. Both local shared memory approaches and distributed message passing schemes have their benefits. One of the early inspirations for this work is in the ongoing multi-core revolution [9], but the A! algorithm itself can function as a distributed one as well. Indeed actors, for example, are a great fit for a distributed system [1].

However, the distributed direction presents additional challenges that are not that relevant to demonstrate cooperation, which is the main focus of this thesis. The ideas discussed here are well worth exploring in a distributed setting, in concert with work division schemes, for example, but these topics are outside the scope of this work and therefore left for future research.

Another decision closely related to the local/distributed issue is that of execution encapsulation. We use actors, which typically are implemented with threads, while a process-based abstraction is more suitable for a distributed system. We wish to emphasize the autonomy of the search workers – exploration of search space at their own terms – so open and closed lists are not shared, but rather each worker manages their own. This follows from reports indicating that lock contention on shared data structures is highly undesirable [20, 72].

Note that in contrast to most parallelization ideas described in the existing literature – such as partitioning and hashing – in this work we take a nonchalant view of the whole issue, suffering work duplication while emphasizing cooperation.

The A! algorithm features explicitly shared progress information as the only piece of shared data. A more indirect interaction mechanism would emphasize the role of the environment abstraction, which has certain appeal in the emergent cooperation sense [15, 31], but this chosen approach gives a nice balance of order and "ripple effects". The essence of A* search is there, but wrapped in a thin cloak of nondeterminism.

Another general issue is the choice in using A* and not the memory-efficient variant *IDA** [78]. *IDA** makes it possible to do iterative improvement and use many workers, the flip side being repeated path traversal. These are essentially the same issues as the repeated path troubles faced in undivided search space. *IDA** is based on A*, but the approach to search space exploration is differently. *IDA**, and other iterative approaches, are exhaustive on each iteration, so there is less to be gained from cooperation: exploration order does not matter. On the other hand, in A!, the cooperation can quickly lead to good progress exactly because the order *does* matter.

The real tradeoff is in the memory blowup multiplied by agent numbers: there is redundancy in the agents in an autonomous model. The *IDA** search variant, with path-only memory, is a good fit to shared lists and cleanly split search space, and the parallelization [72], even with its issues, is straightforward – do more of the same. A! is a bit more messy.

In A! all of the agents can proceed in any direction and can explore any set of nodes in the graph. More structure could well be imposed, and indeed many powerful algorithmic ideas are based on making good use of the structure of the problem instance, but the core of agent-based cooperation is autonomy. Various local, shared and distributed memory mechanisms developed to deal with memory blowup in A* should work without issues in A!.

Emergent cooperation, where the entanglement of the participating agents is even more prominent, is a feature in the work of Nitschke [96], Crainic [31] and others. Much of this work lies at the very fringe of computer science and suffers from a lack of sound results, but some ideas, such as bio-inspired algorithms [38, 96]

and natural computing [12, 24] are gaining some traction.

Effective cooperation means that the agents are able to communicate in a smooth, unobtrusive way that does not slow down their main functions too much. What is sent matters, when it is sent matters, all-to-all or something else, underlying hardware – all of this matters. Best-sharing, as previously described, is one way to do it, a simple one, but sending more state data and being clever in reasoning about it will likely give even better results. The flip side is the communication cost. With simple messages, the agents spend a minimal time processing the inbound and outgoing data.

Hickey [57] argues that actors are an unnecessary complication on top of message queue semantics. Scholliers et al. [119] also find actors to not be the ideal abstraction for concurrent programming. The main critique is focused on scalability, which is perhaps a lesser concern here in a local setting, as core count is currently and in the near future only in the low tens rather than thousands [14, 39]. The criticism is valid, though, and actors are introduced here only as one way to enable cooperation. Actors are not the center piece here.

One practical issue in implementing A! turned out to be the difficulty of terminating quickly and in a clean fashion. Concurrent programming is notoriously difficult, yes, but simply killing the process leaves something to be desired, too. There is the issue of multiple agents finishing at the same time, and returning but a singe value. In the reference implementation, rudimentary synchronization was used to manage this, but more scalable solutions are needed especially for a potential distributed version of the algorithm.

The main computational bottleneck of the A! algorithm is the priority queue used to sort nodes by their estimated cost. The Fibonacci heap [28, 45] is a good match to the modifiable priority queue needs of the A*. There are also d-ary heaps [69], which are fast for the same problem, and the highly complex Brodal queues [16] of which there is a persistent functional version [17]. For A!, the peek functionality outlined in A!Solver (Algorithm 3) requires an at most k-deep traversal of the tree to gather a peekList, so a data structure optimized for k-deep peeks would be ideal, but currently one is not available.

While a long time coming, the functional view to parallelism is poised to really break through in the near future: uniprocessors are not getting any faster, but compilers and functional languages targeting multi-core processors are getting better [104]. The reactive style of programming [10], with its absence of explicit control flow, is likely to play well with parallel programming in general [23] and cooperation-based algorithms, such as A!, in particular.

A! is straightforward to implement using programming language constructs like the goroutines in go, actors in Erlang, and the `core.async` library in Clojure. State and its management are important in all A! implementations, as opting in

for unshared lists makes efficient memory usage a priority. Nodes must be *very* efficiently represented, and caching and other tricks should certainly be made use of, but on the other hand this would go against the idea of programming at a high level of abstraction. Ultimately one hopes to decouple correctness from the practical complexity without losing efficiency [23].

A* search is a straightforward sequential algorithm, but the inherently parallel A! must give up some of that simplicity in favor of enabling cooperation. What is lost in turning a deterministic algorithm into a concurrent nondeterministic one is gained in performance. The next chapter presents a suite of experiments that show how A! performs in the n-puzzle problem context.

Chapter 4

Solving n-puzzles with A!

This chapter presents a series of A! experiments conducted on a collection of instances from the n-puzzle family of sliding block puzzles. We first give a simple problem formulation for the puzzles and then establish a suitable version of the A! algorithm for solving them. The rest of the chapter is dedicated to reporting results obtained from computational experiments.

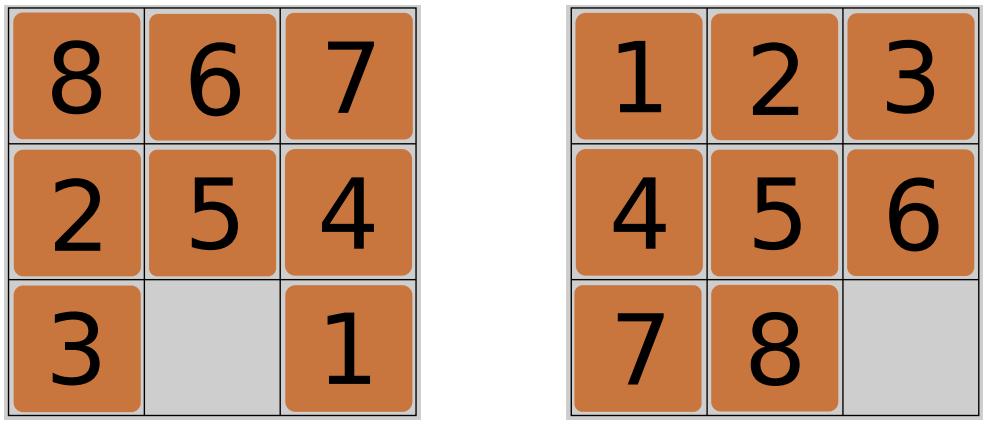
We focus on two dimensions that are especially interesting from a cooperative search point of view: the overall benefit from cooperation and the extent to which the method is scalable. Additional experiments in explored path diversity and heuristic function sensitivity offer more insight into the workings of the new algorithm. Finally, a hybrid algorithm combining A! and random exploration is briefly tested.

The results show that A! consistently outperforms both vanilla A* and a randomized parallel version of A*. Adding more agents to A! clearly improves the performance, but the returns are diminishing. A! appears to utilize given heuristics better than the reference algorithms and explores the search space in a slightly more efficient way. Hybridizing A! with randomization does not result in a more capable search algorithm, indicating that A! evades the superfluous paths that a randomized search spends time on.

4.1 The n-puzzle

The *n-puzzle*[1] is a classic sliding block (or tile) problem family with a long history in AI research [36, 115] – and an even longer history outside it [70, 122]. The 15-puzzle attracted attention from the general public and mathematicians alike from as early on as the 1870s [122].

[1] We adopt this slightly awkward name to refer to the family of (m^2-1)-puzzles of this kind.



(a) A start state. (b) The standard goal state.

Figure 4.1: 8-puzzle states. On the left an initial state; one of the two hardest instances in its class with an optimal, shortest solution sequence of length 31. On the right one of the possible goal states. The other common goal state has the empty slot in the top left corner, but any other configuration can also be used.

The *8-puzzle* features a 3×3 board with eight numbered blocks and a blank, the *15-puzzle* having 15 blocks on a 4×4 board, and so on. The objective of the puzzle is to arrange the blocks to match a goal configuration by sliding the blocks horizontally and vertically onto the blank square, constantly making way for new moves. Figure 4.1 shows an initial state and a goal state for the 8-puzzle.

While literally a toy problem, the n-puzzle is less trivial than may seem at first glance. Finding a solution to instances of even a large version of the puzzle is not very hard, but the task of finding the shortest sequence of moves to reach the goal configuration is computationally interesting. The task of finding a k-bound sequence of moves for the general $m \times m$ (or $m^2 - 1$) version of the puzzle was proven NP-complete by Ratner and Warmuth [108].

Russell and Norvig [115] give a search problem formulation:

- **States:** A state describes the location of each of the tiles and the blank.
- **Initial state:** Any state can be designated as the initial state. Any goal state can be reached from exactly half of all the possible initial states. This follows from a parity argument on the cycles of a permutation representation of the state [109].[2]

[2]This fact was used to great effect by game designer Sam Loyd in the 1870s: an odd-parity 15-puzzle state cannot be turned into an even-parity goal state by any sequence, so he was able to issue cash prize challenges for solving his unsolvable puzzles. [122]

- **Actions:** The actions can be described relative to the blank, with *Left*, *Right*, *Up*, and *Down*. The set of enabled actions is dependent on where the blank is on the grid, with moves resulting in blocks being off-board being illegal.
- **Transitions:** Given a state and a legal action, the resulting state is straightforward to compute as a move of the blank to an adjacent location by moving the targeted block in the the currently blank location.
- **Goal test:** Check whether the state matches the goal configuration.
- **Path cost:** Steps cost 1 each, so path cost is the number of steps in the solution.

Sliding block puzzles from the n-puzzle family are often used as test problems for new search algorithms, because the task is easy to describe and the search space size grows quickly with larger boards. Russell and Norvig [115] justify the use of heuristic search methods by the numbers as follows.

The average solution cost for a randomly generated 8-puzzle instance is about 22 steps and the the 8-puzzle has an *average branching factor* – average out-degree of the nodes in the search graph – of about three: four moves are possible when the blank is in the middle, three on the edges and two in the corners. An exhaustive tree search to depth 22 would look at about $3^{22} \approx 3.1 \times 10^{11}$ states but a graph search cuts this down to $9!/2 = 181{,}400$ distinct states – a cut by a factor of $170{,}000$. This is completely manageable, but doesn't scale: the 15-puzzle has around 1.3 trillion states, and the 24-puzzle around 10^{25}.

Heuristic algorithms and efficient data structures and representations are needed to bring random instance solution times to under few seconds. Some common heuristics for the n-puzzles include, in increasing order of effectiveness [115]:

- **Misplaced tile count:** The number of tiles in positions that do not match their goal positions. For example, with block 5 in the right place, the state in Figure 4.1a has a heuristic value of 7 with respect to the goal state in Figure 4.1b.
- **Manhattan distance:** (taxicab/L_1-/rectilinear distance) The sum of distances the tiles are from their target positions, counted as moves along the grid. For the state in Figure 4.1a we have, from block one, $4+2+4+2+0+2+4+3 = 21$.
- **Manhattan distance with linear collisions [52]:** The linear collisions extension to Manhattan distance makes use of the fact that two blocks on the right row, but in the wrong order must pass each other to reach their targets. Figure 4.1a has collisions on each row, e.g., $8-7$, $5-4$, and $3-1$. Linear collisions can be calculated for both rows and columns and can be combined for a single "criticism" on top of regular Manhattan distance. Hansson et al. [52] give a complete algorithm.

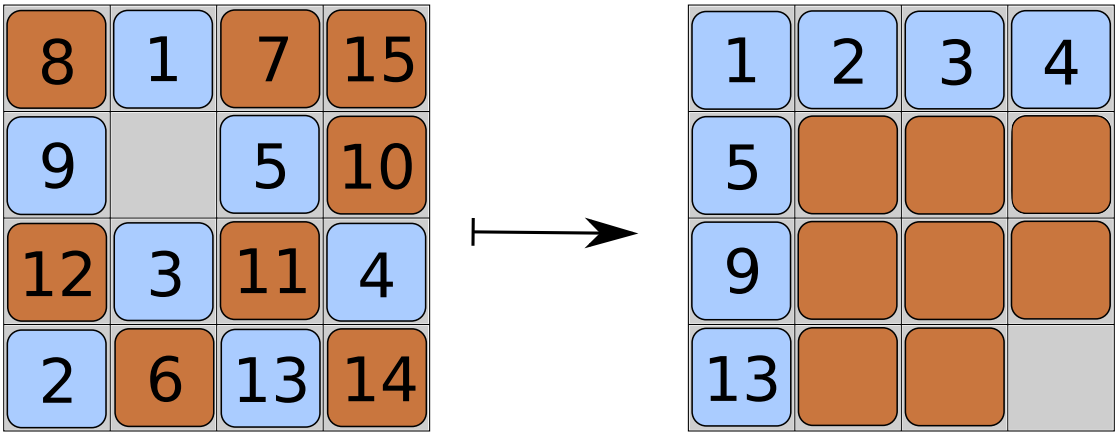


Figure 4.2: Pattern databases map configurations to partial solutions. The blue tiles in the state on the left can be moved to their target positions on the right in some number of steps, which is then stored in a massive lookup table. Solving the entire puzzle takes at least as many moves as the length of the shortest sequence that is needed to solve the sub-task formed by the blue tiles.

- **Walking distance:** Similar in spirit to linear collisions, the walking distance heuristic is based on walk patterns that approximate the horizontal and vertical moves necessary to take a block to the target. The walking distance heuristic was described and implemented by Takahashi Ken'ichiro in 2002, but it remains unpublished in English. Some details are available on his website.[3]

- **Pattern databases [34]:** The very general idea of a pattern database is to store a collection of precomputed solutions to sub-problems, effectively trading execution time for memory. A number of different subtasks can be used, and their selection is mostly dictated by the available resources. Figure 4.2 shows the kind of entries one could use in a 15-puzzle pattern database. Multiple databases – or indeed any heuristics – can be combined by taking, e.g., the maximum or the average of the values they provide. Addition is typically not possible, as it generally results in overestimation.

- **Additive/disjoint pattern databases [43, 79, 80]:** The logical continuation to multiple competing databases is to have them collaborate. If pattern databases are built of disjoint sets of patterns, counting only moves by nodes in the their respective sets, the sum of non-overlapping heuristics stays admissible. Korf and Felner [80] present a top/bottom half divided pattern heuristic for the 15-puzzle that takes 550 megabytes of memory and reduces the number of

[3] `http://www.ic-net.or.jp/home/takaken/e/15pz/index.html`

generated states by four orders of magnitude compared to Manhattan distance heuristic. Note that Manhattan distance can be considered an extreme example of a disjoint set pattern heuristic: each set is a single block.

- **Learned heuristics [67]:** Most recently, the tools of machine learning have been applied to discovering heuristics for the 24-puzzle. The idea in learned heuristics is that solving some instances gives an idea about the nature of the problem: encountered subtasks can be used to solve similar subtasks in new instances. The major challenge is – not unlike in machine learning in general – in finding good sets of features to learn on. Taking the learning concept a step further leads to hyper- and metaheuristic methods [19].

The heuristics outlined above have been designed for the sequential search case, but are equally applicable in parallel search as well.

4.2 Experimental setting

The experiments described in this chapter are based on repeated executions of four versions of heuristic search, all based on a single implementation: vanilla A*, a randomized parallel variant A?, the cooperative A!, and the hybrid A″.

In this section we explain some details about the implementation and outline the experiments themselves. Related material is available online[4] and at request.

4.2.1 A! implementation

The implementation used in the experiments is based on an A* for n-puzzle implementation by Brian Borowski[5], denote 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. Using *IDA**, BBI is capable of solving random 15-puzzle instances in a few seconds and the hardest 80-move 15-puzzles in under a minute on a 3.30GHz Xeon E3-1230 V2 machine. The vanilla BBI-A* uses roughly a gigabyte of memory per minute, exhausting typical desktop computing resources on medium difficulty 15-puzzles (∼50-60 move optimal paths).

BBI features three heuristics: Manhattan distance (MD), MD with linear collisions (LC) [52], and a 6-6-3-partitioned disjoint pattern database (PDB) for the 15-puzzle [43]. The solver can use state space dividing work pool multi-threading, and uses efficient integer state representation with shift operations as well as a custom Fibonacci heap backed priority queue [28]. BBI can also solve 8-puzzles.

[4] `https://github.com/ajhalme/coop-agents`
[5] `http://www.brian-borowski.com/Software/Puzzle/`

Algorithm 9 : **Recursive Fibonacci heap peek** (`getPeekListR`)

Require: LIST *peekList*, NODE *top*, INTEGER *minCost*, INTEGER *depth*, INTEGER k
Ensure: The *peekList* contains nodes with a cost of *minCost*
  if $top.cost > minCost$ **or** $peekList.size() \geq k$ **or** $depth \geq k$ **then**
    return
  end if
  $peekList.add(top)$
  for each *child* **in** $top.GetChildren()$ **do**
    $getPeekListR(peekList, child, minCost, depth + 1)$
  end for
  if $top.right \neq top$ **and** $top.right \neq min$ **then**
    $getPeekListR(peekList, top.rigth, minCost, depth)$
  end if
  return

The following modifications were done in order to build A! on top of BBI-A*:

- A* functionality was encapsulated into a worker agent,
- a MessageBroker component was added and connected to the workers in a publish-subscribe pattern [29, 40] using ZeroMQ [58],
- the Fibonacci heap was extended with a recursive `peekList`-routine, Algorithm 9, that allows simple k-deep equal-cost node peeking into the heap,[6]
- (node, cost)-tuple sending and receiving functionality was added to workers, and
- the selection policies were implemented, notably *a!Select* with cooperation and the secondary heuristic function, $\hat{h}$.

The secondary heuristic was set to be MD-to-best for the MD primary heuristic, and LC-to-best for LC and PDB primary heuristics, i.e., for MD: $h = \hat{h}$ =MD, for LC: $h = \hat{h}$ =LC, and for PDB: h =PDB, $\hat{h}$ =LC.

Four selection routines were implemented for the experiments. The cooperative search routine A! (Alg. 4, Section 3.3.4), based on the secondary heuristic $\hat{h}$, was the main experimental target. It was pitted against the random (A?, Alg. 6) and vanilla selection (A*, Alg. 7) policies. Additionally, a hybrid selection routine based on a simple combination of cooperative and random selection (A″, Alg. 8) was used at various split thresholds p: $Pr(A!) = p$, and $Pr(A?) = 1 - p$.

[6] This is a very rough data structure hack in lieu of a modified heap that keeps track of not only the top node, but up to k nodes that share the minimum cost of the top node.

4.2.2 Experiment design

The test suite is a randomly generated collection of 8- and 15-puzzle instances. The instances were solved with BBI and grouped by the length of the optimal solution, the shortest sequence of moves from the given start position to the goal position. Randomly generated impossible instances – off-parity with respect to the goal state – were discarded.

Grouped this way, the instances within a given group proved to differ greatly in their difficulty. Regardless of method and heuristic, the average number of opened nodes during the search for instances in a optimal length 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.

For example, after visiting on average some 2500 states per agent, A! with eight agents and the PDB heuristic finds the optimal 52 step path for the 15-puzzle

$$[0\ 9\ 8\ 10\ 14\ 13\ 12\ 3\ 6\ 7\ 4\ 15\ 11\ 5\ 2\ 1],$$

shown here as permutation vector over the $(n+1)$ tile locations. This is in contrast with another 15-puzzle, also optimal in 52 steps,

$$[3\ 6\ 9\ 13\ 7\ 0\ 4\ 11\ 5\ 1\ 14\ 12\ 10\ 15\ 8\ 2],$$

which requires about $200,000$ node visits per agent. The relative standard deviation for A! runs is around 20% for both. This difficulty variability was observed with all evaluated methods, as well as Borowski's *IDA** implementation, and is in line with reports in the literature [78, 80]. The grouping is still justified, as with enough instances in each group, some general trends become apparent.

To experimentally show that one method is conclusively superior to another with all of this variability requires a vast suite of instances and many runs on them, but this is beyond the scope of this work. With many interesting variables to choose from in this setup, a full study with the hardest instances would take an inordinate amount of resources, so we instead present the results from a reasonable test suite and hope to glean some insights and general trends from the data.

Runtime is not considered at length here nor later in the work. The goal in this work was not to build a competitive n-puzzle solver, but to study cooperation effects in A!. While the implemented solver can solve medium difficulty 15-puzzles in a few minutes, Borowski's program, for example, solves random 15-puzzles much faster and even the hardest instances in under a minute. Optimized state-of-the-art solvers defeat the evaluated implementations in every aspect and operate at a completely different scale, working on instances from the next level, the 24-puzzles, and beyond. We return to the relationship between this work and the state-of-the-art in Chapter 5.

The majority of the computational experiments presented in this work were performed using the computer resources of the Aalto University School of Science "Science-IT" project. The cluster that ran the test suites comprised a mixture of blade servers, featuring 2.6GHz Opteron 2435, 2.67GHz Xeon X5650, and 2.8GHz Xeon E5 2680 v2 processors.

To give a sense of the experimental scale, the final results shown in Figures 4.3 through 4.6 alone required about a full year of single core processing time. There are 20 groups of 100 instances, each instance is iterated five times and lasts upwards of 60 seconds on average. We evaluated two methods with 1, 2, 4, 6, and 8 agents at one core per agent, and did one extra run for the vanilla A* case.

4.3 Computational results

This section presents an overview A! performance 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 cooperation benefits slightly more, making a preliminary case in favor of cooperation.

Third, we focus on a few instances that represent the problem well and look at how the explored path diversity differs between the methods under study. We try to understand the qualitative nature of the methods and give a simple visualization of the search space explored by each method. Fourth, we consider the relative performance gains from an improvement in the heuristic, and argue that the cooperative method benefits more from the improvement.

Finally, we consider a hybrid approach, A″, where we combine A! and A?, as featured in Algorithm 8, and show that the resulting algorithm does not represent an improvement: A″ appears to explore states that A! correctly chooses to ignore.

4.3.1 Cooperation benefit and scalability

To see how A! performs against the competition, we set the algorithms to solve a suite of instances and observe how many vertices are opened during the searches. Figures 4.3 and 4.4 show 100 15-puzzles from each of the 40–59 optimal length groups, run on A*, A? and A! in five agent configurations: 1, 2, 4, 6, and 8 agents. The primary heuristic used here is the 6-6-3 pattern database. We use the median of five runs: this was found to be a reasonable compromise between result quality and available computing resources.

The figures show how the three algorithms compare on instances of the randomly generated instance set, pruned to show only instances that pass in 10 min-

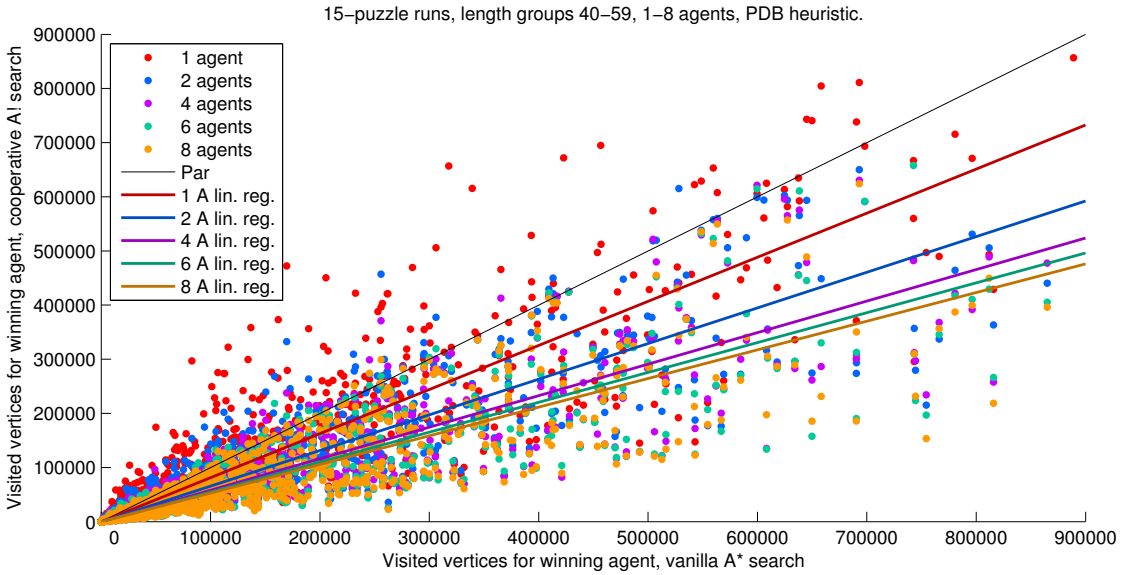


Figure 4.3: Relative performance of A* (x-axis) and A! (y-axis). The black line is par, so data points below it represent instances for which A! performs better than A*. Agent count is evaluated in five batches – 1, 2, 4, 6, and 8 agents – with the respective trend lines showing how the methods compare.

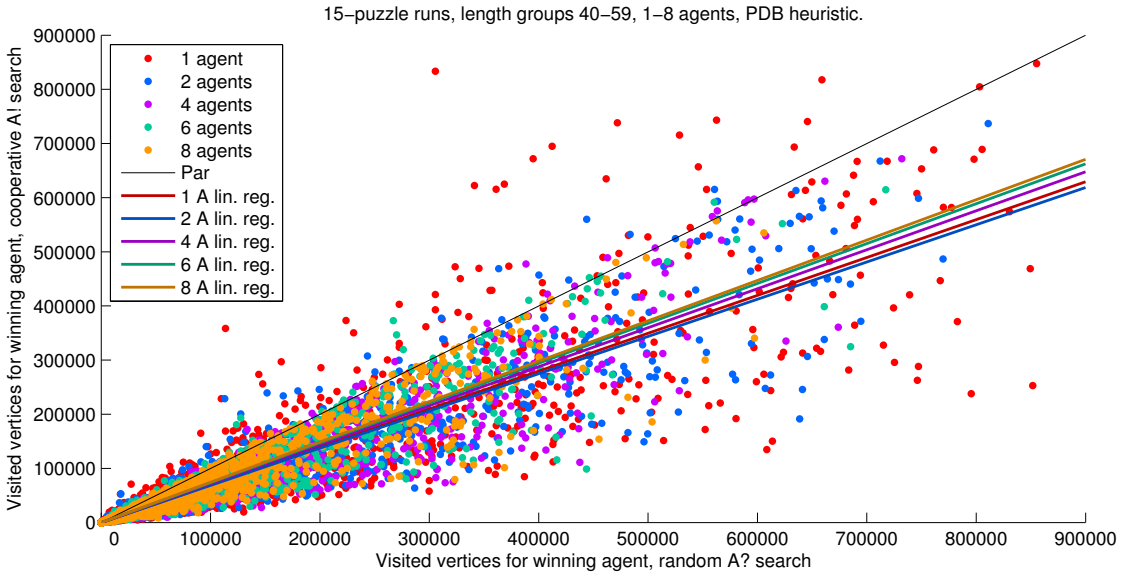


Figure 4.4: Relative performance of A? and A!. As before, the data points and trend lines below par-line reflect the benefit from cooperation. The slopes vary from around $\frac{7}{10}$ to $\frac{8}{10}$, reflecting a $25-40\%$ performance difference in favor of A!.

utes using the methods. We focus on the opened states per core -metric, specifically the number of states the winning agent opens as it searches. This correlates with the total number of opened nodes over all agents, including repetition, as agents explore states at roughly the same rate. It also correlates with total execution time with harder instances, but below five seconds, this measure is less informative.

In Figure 4.3 we see the majority of the points falling under the black par-line, indicating that search using A* is more sluggish than with A!, with more states being 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*.

In Figure 4.4 we compare A! with A?. The trend lines are more uniform this time, suggesting that cooperation is beneficial in all the tested agent configurations and to a somewhat similar degree. A clear majority of the instance data points lie under the par line, in favor of A!. Overall, the data supports the cooperation benefit hypothesis. For the one agent case we also see the momentum effect from the secondary heuristic: the single agent favors the nodes opened recently.

The trend lines are roughly in reverse order compared to Figure 4.3, with A? suffering more from fewer agents. While the effect is not substantial, this suggests – at least for these implementations – a difference in diminishing returns in favor of A?: adding an agent improves A? performance more than that of A!. Cooperation is still valuable, A! outperforms A? for all agent configurations, but A! could benefit from some kind of diversity boosting mechanisms that reduces search overhead and makes additional agents less redundant.

Figure 4.5 offers a different view to the instances. The figure shows the runs grouped by optimal solution path lengths, with each method and agent configuration as a separate data point. The value on the y-axis is the group mean of instance medians for the number of states visited by the winning agent – 100 instances at five runs per instance per point. We see a rather consistent pattern with the vanilla A* setting a baseline, A? runs improving on that a bit, and finally multi-agent A! getting the lowest values.

Searching efficiently in parallel requires a scalable algorithm. Adding new agents clearly increases A! performance, but less so than for A?, as is visible in Figure 4.6. 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 are averaged over for a pair of thick mean-of-means curves that summarize over the thousands of data points drawn evenly from the

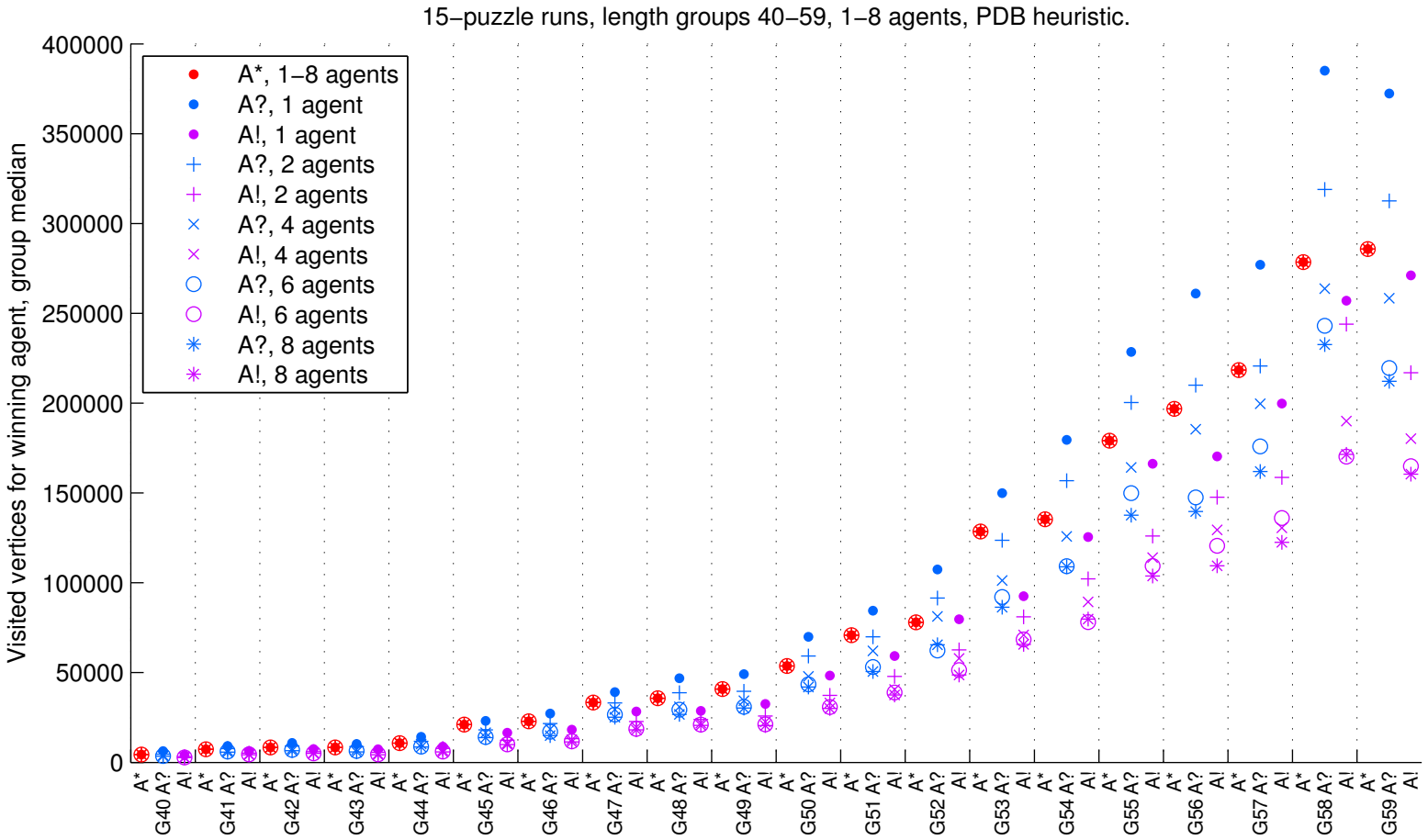


Figure 4.5: Vanilla A*, random A? and cooperative A! performance on 15-puzzles grouped by the length of their optimal solution paths with each group having 100 instances. The figure shows some general trends for search configurations of 1, 2, 4, 6, and 8 agents. The medians get more erratic towards the harder instances to the right hand side of the figure, but the general trends are already visible at this experimental scale. Cooperation appears to be more beneficial as instance hardness increases.

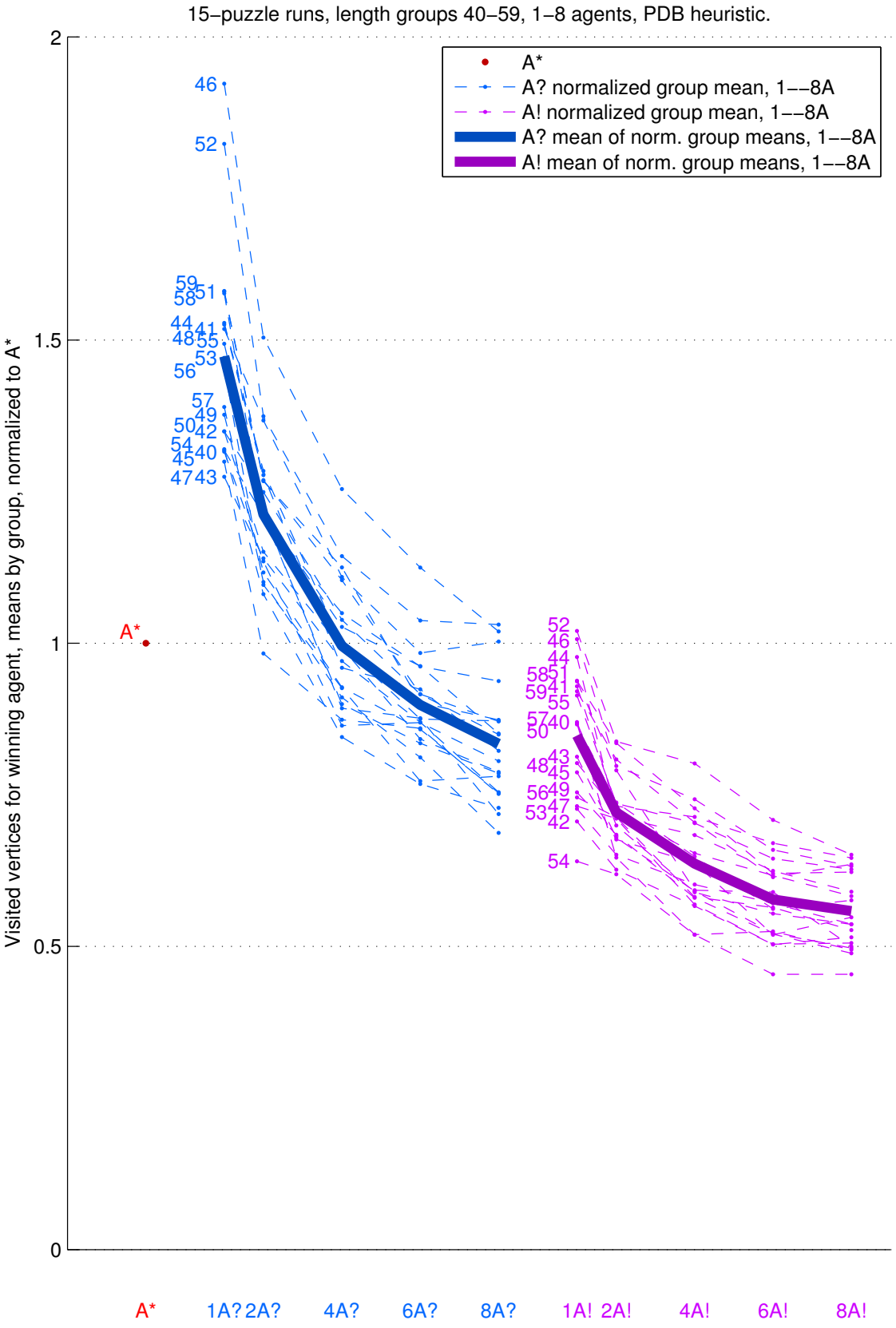


Figure 4.6: A*-normalized length group means – 100 instances per group, five iterations per instance – demonstrating the scaling benefit from adding more agents. The trend lines overlap to a moderate extent, suggesting rudimentary asymptotic bounds for the algorithms.

$40-59$ optimal path length range.

For some of the groups in Figure 4.6, A! gets close to the $\frac{1}{2}$ threshold in A*-relative performance, 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! not only outperforms A* and A!, but scales to at least a few agents, but with quickly diminishing returns.

Overall, the cooperative A! algorithm appears to consistently overcome both vanilla A*and the random A? for the 15-puzzle instances considered here. While the n-puzzle is a standard benchmark for heuristic search, it is only a single problem – and a simple one at that. It is unclear whether the depth-first orientation, a property of the secondary heuristic in A!, is beneficial only in problems such as the n-puzzle, or generalizes to many search contexts, but the presented results at least encourage further study.

4.3.2 Heuristic impact

Three of the heuristics introduced in Section 4.1 were explored in this computational experiment for the 8- and 15-puzzles: Manhattan distance (MD), Manhattan distance with linear collisions (LD), and a 6-6-3 disjoint pattern database (PDB). Linear collisions improve MD significantly, and the PDB heuristic is greatly more accurate than LC. All the methods evaluated here explore several orders of magnitude more states with MD than with PDB.

Figures 4.7 and 4.8 show how the number of participating agents and the used search method influence the runs, respectively. On the x-axis we have the linear collision heuristic and on the y-axis the PDB heuristic, with instances solved with both heuristics as the data points. The black bar represents heuristic parity, where both heuristics direct the search equally well, while data points below it are in favor of the PDB heuristic. The PDB heuristic dominates LC completely.

From Figure 4.7, with data points colored by agent configurations, we see a somewhat uniform distribution of instances over the projection space. This leads to trend lines maintaining essentially the same direction: in comparing heuristic impact, the number of agents appears to be next to irrelevant.

We see a bit more impact when we color the data by method. In Figure 4.8, we have the exact same figure, but this time trend lines are drawn by method. While the effect is not substantial, there is a difference now between the trend lines. The interpretation is that A! benefits more from an improvement in the heuristic function. One reason for this could be that a good heuristic is a better match for deep rather than broad exploration.

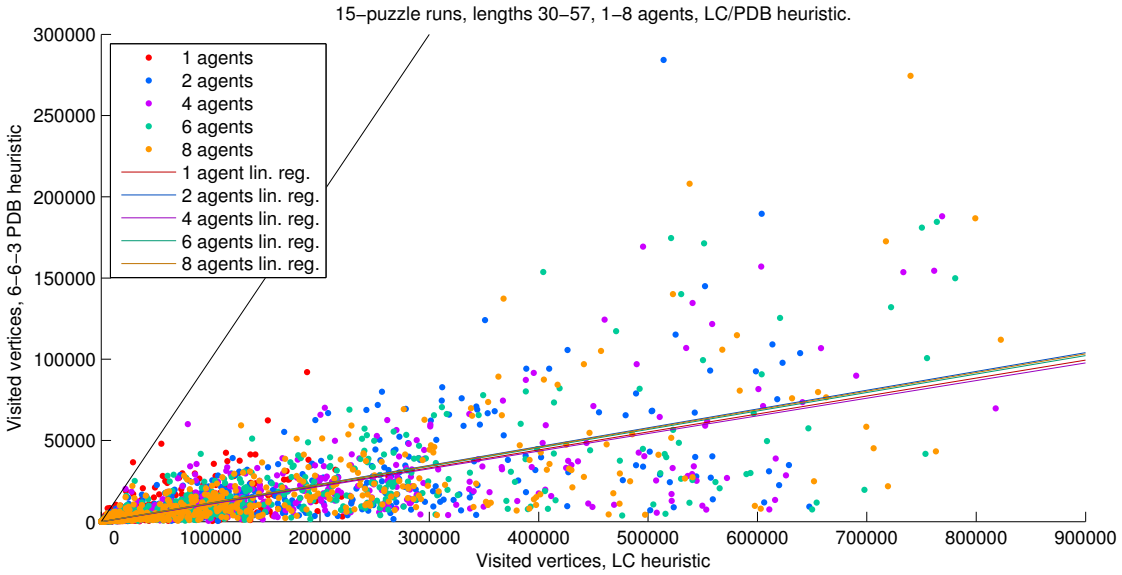


Figure 4.7: Heuristic comparison with data grouped by agent configuration. The trend lines being essentially the same indicates that the number of agents is not strongly correlated with heuristic impact: regardless of method, two agents benefit from a better heuristic as much (or little) as eight agents.

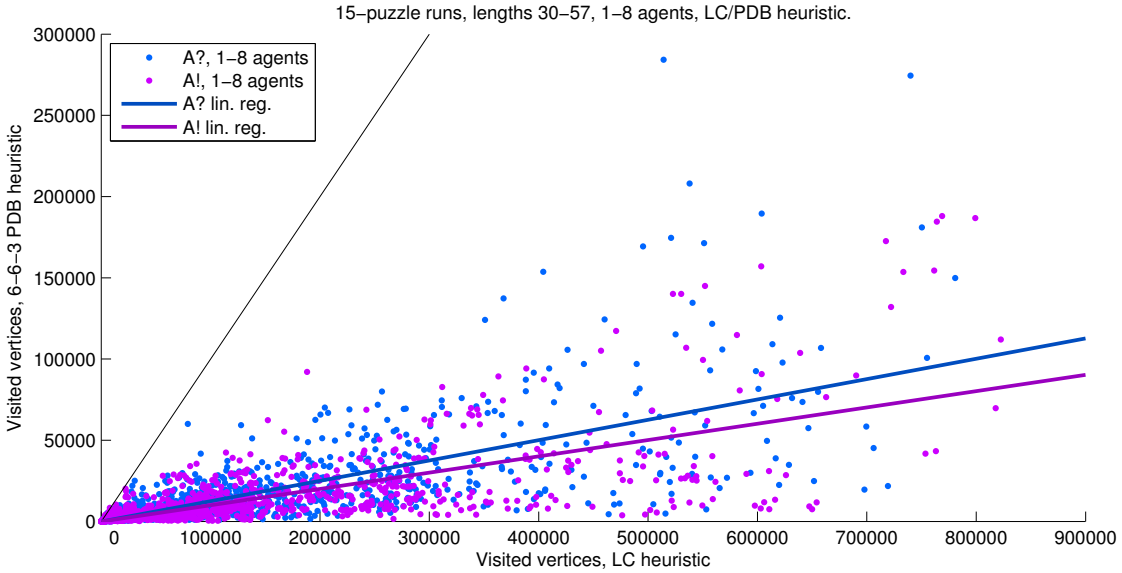


Figure 4.8: Heuristic comparison with data grouped by method. The now more visible difference in trend lines suggests that A! benefits more from the improved heuristic than A?. The slope is about $\frac{1}{8}$ for A?, and around $\frac{1}{10}$ for A!.

4.3.3 Path diversity

As discussed in section 3.2.4, one major challenge in parallel search algorithms lies with work distribution and search overhead due to replicated efforts. Vanilla A* is deterministic, so having multiple agents run that algorithm offers no algorithmic benefit, but presents a useful base case for experimental study. In particular, running multi-agent vanilla A* gives a rough idea about the kinds of overheads involved in even non-communicating parallel execution.

For this work, we wish to see how A! fares against A* and A? as measured in explored states. We can view parallel search overhead as *path diversity* – or rather lack thereof – referring to the order in which search agents explore the search space. If different agents mostly explore the same sates, path diversity is low, while perfectly non-overlapping agents follow highly diverse paths. Intuitively, we expect A* to have extremely low path diversity, but the impact that cooperation and the secondary heuristic have on path diversity in A! is far less clear.

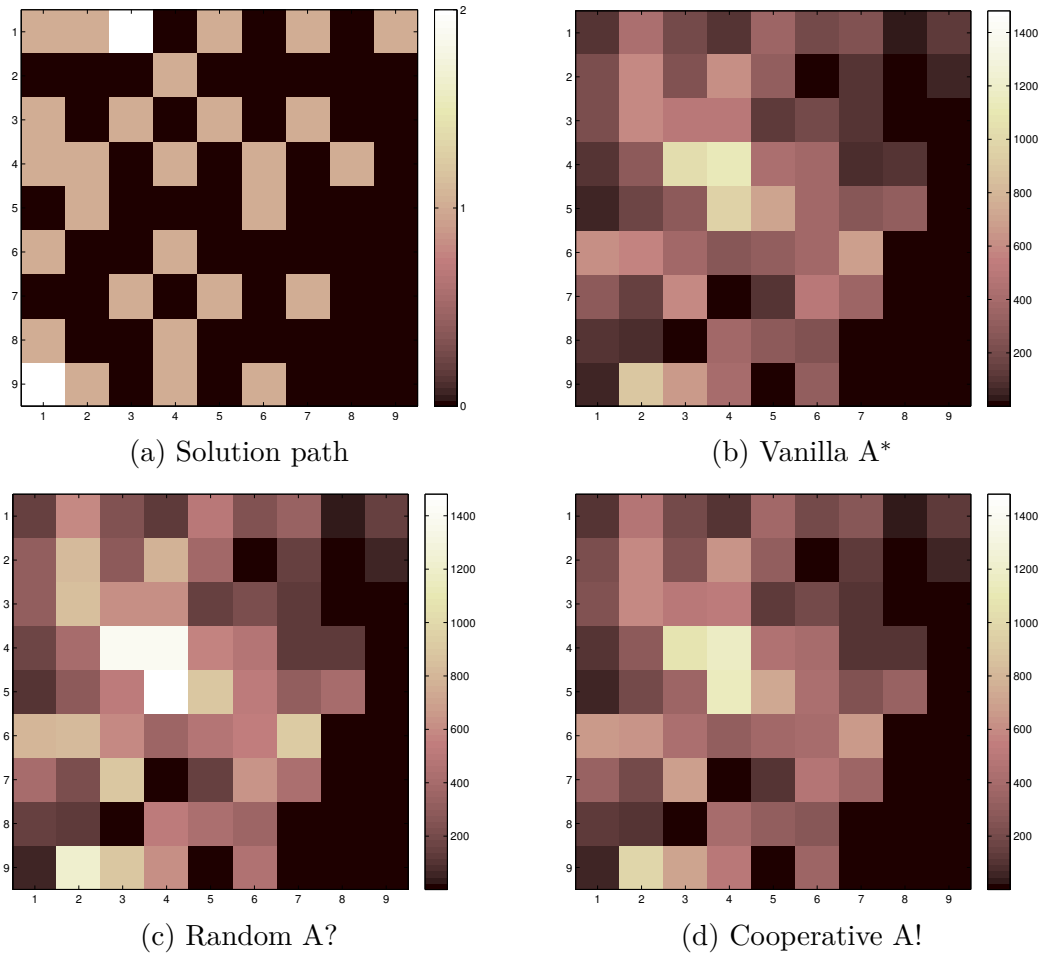
Beyond plain numeric evaluation, we turn to a visualization scheme to give us an informative view of the search space in which the algorithms operate. From the wealth of methods to consider for this task, we opt for topology preservation over accurate distances and select the nonlinear dimensionality reduction method of *self-organizing maps* (SOM) [77, 85].

A SOM is an artificial neural network (ANN) trained unsupervised, i.e., without validation labels and external error signals, to represent the input space in fewer dimensions. Given a collection of input samples, a SOM discovers a neighborhood maintaining map, with which subsequent data units can be projected into an output space that is more approachable for interpretation. SOM functionality is readily available in the MATLAB Neural Network Toolbox[7] and other computing environments. Here, we treat SOM as a black box method and employ it using reasonable defaults: batch weight/bias training rules, mean squared error guide measure and two hundred learning iterations.

The states of the n-puzzle are conveniently represented as permutation vectors. For example, we have the permutation vector s_1, [8 6 7 2 5 4 3 0 1], for the initial state we saw in Figure 4.1a. Viewing the tile locations as dimensions yields a natural distance function over the permutations. A more involved permutation metric [118] – mindful of the nontrivial transformations reflected in the legal puzzle moves[8] – could well be employed, but simple Euclidean distance in $(n+1)$ dimensions is sufficient for our SOM visualization purposes.

[7] http://www.mathworks.se/products/neural-network/

[8] E.g., [8 6 7 2 5 4 3 0 0] is an illegal state in the 8-puzzle, as it does not feature all of the tiles. While [8 6 7 2 5 3 4 0 1] is legal in the previous sense and only one "change" away from s_1, the state shouldn't be that close to it in the puzzle: the tile locations involved are on different sides of the board, and the new state is not even of the same parity as s_1.



(a) Solution path (b) Vanilla A*

(c) Random A? (d) Cooperative A!

Figure 4.9: Visualization of A*, A? and A! on 8-puzzle [8 6 7 2 5 4 3 0 1]. The heatmaps are derived from a 9×9 self-organizing map trained on an optimal solution path of 31 steps, shown top left mapped to the codomain.

METHOD	A*				A?				A!			
FREQ.	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
MEAN	648	414	269	4780	3290	1147	903	5245	562	600	1006	4544
RATIO	0.11	0.07	0.04	0.78	0.31	0.11	0.09	0.50	0.08	0.09	0.15	0.68
STD	476	363	235	525	1345	559	327	716	233	254	372	656

Table 4.1: State visits for A*, A? and A! on 8-puzzle [8 6 7 2 5 4 3 0 1]. The table gives the mean, ratio and standard deviation of state visit frequencies from ten iterations of A* on four agents with Manhattan heuristic.

Figures 4.9 and 4.10 together with the respective tables 4.1 and 4.2 represent two n-puzzle SOM systems – one featuring the running example starting from state s_1, the other one being an average difficulty 15-puzzle[9]. The selected 8-puzzle is one of the two hardest 8-puzzles, with an optimal solution of 31 moves, while the 15-puzzle has an optimal solution of 54 moves.

In both cases, a SOM is trained on the solution path and the states – including repeated visits by multiple agents – are mapped to a two dimensional grid. Figures 4.9a and 4.10a show how the solution path itself maps into the output space. The SOM learns an ANN that gives a somewhat homogeneous cover of the grid in Figure 4.9a, but much less so in Figure 4.10a. This might reflect the relative difficulty of the instances, with harder instances covering more area: a hard 8-puzzle leads to a near-complete 8×8 SOM cover, while an average 15-puzzle gives only a half-cover over a 16×16 SOM. While any insight on search space coverage might be very useful in improving parallel search diversity, we do not pursue this idea further here.

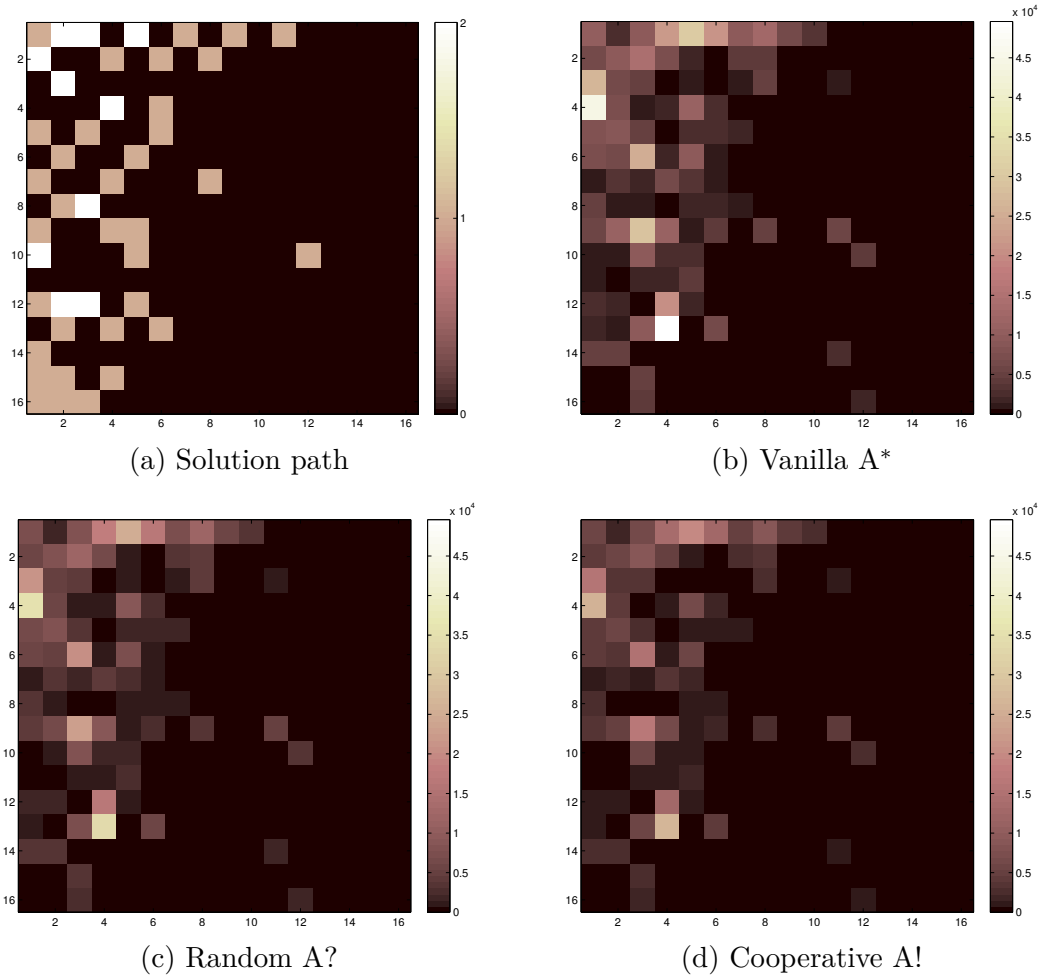
The algorithm heatmaps represent the average of ten runs with explored state space logged in each separately. All runs were done with four agents and using the Manhattan heuristic. Each cell represents a SOM "bucket" to which nodes from the puzzle domain get mapped. The color scheme scale is shared between the three algorithms, revealing very similar patterns and mostly intensity differences. The algorithms go through states in the same neighborhood, with some algorithms opening more nodes, others less in a given neighborhood.

Notably, for the instance in Figure 4.9 we find A* to perform much like A! , with A? lagging behind – a fact also reflected in Table 4.1. The table gives means and standard deviations for each state visit frequency group, e.g., for A*, on average about 4780 ($\sim 80\%$) nodes were visited by each one of the four agents, the standard deviation over ten iterations being 525. In the figure we see A? lighting up more than the competition, i.e., more nodes are getting opened, which is also apparent from the *mean* table row.

The instance in Figure 4.9 is not very hard, even with the lackluster Manhattan heuristic, and serves mostly as an introduction to the visualization: the power of A! – and A? – does not manifest yet. However, Table 4.1 suggests some additional considerations. First, running four A* agents does not yield four times the same logs, but one complete and three a bit short. This is due to the fast termination policy in which the first one to reach a goal ends the overall search. Also, scheduling and the runtime environment make some of the agents proceed more slowly than others. This results in a kind of a coexistence overhead, with *only* once, twice or thrice opened nodes being visible also in the A* *ratio* row.

Second, even adjusting for the coexistence overhead, A? clearly opens more

[9]Hardest 15-puzzles have an optimal solution of 80 moves.



(a) Solution path (b) Vanilla A*

(c) Random A? (d) Cooperative A!

Figure 4.10: 16×16 SOM visualization of A*, A? and A! on 54-optimal 15-puzzle [12 8 6 3 13 4 2 7 0 9 15 5 14 10 11 1].

METHOD	A*				A?				A!			
FREQ.	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
MEAN	661	862	346	158124	91459	32347	13256	76659	40168	8406	4253	75356
RATIO	0.00	0.01	0.00	0.99	0.43	0.15	0.06	0.36	0.31	0.07	0.03	0.59
STD	352	1573	379	1420	27768	27092	14508	4702	29408	8212	1800	1040

Table 4.2: State visits for A*, A? and A! on 15-puzzle [12 8 6 3 13 4 2 7 0 9 15 5 14 10 11 1]. The table gives the mean, ratio and standard deviation of state visit frequencies from ten iterations of A* on four agents with a 6-6-3 disjoint pattern database heuristic.

unique nodes – at least in this instance – than the other two methods. A? opens more nodes overall as well, but the ratios for singleton node visits and all four agents visiting differ from the other methods. Finally, the standard deviation row reflects the volatile nature of A? and also the stability of A!. A! performs in a much more consistent manner than A?.

Figure 4.10 presents a larger 15×15 board for the larger 15-puzzle and the benefits from cooperation make an appearance. The heuristic is a pattern database this time. The instance has an optimal path of 54 steps, spread out in Figure 4.10a to the left and upper left side of the board by the SOM. This time the uninformed A* shows its true color, as the four agents travel much of the same search space, yielding $\sim 99\%$ all-repeat ratio in Table 4.2 with proportionally low deviation.

A? overcomes A* this time, A! doing better still. The *mean* row in the table shows how both A? and A! have plenty of unique nodes, with A! showing more variance in the low frequencies on the *std* row. The overall opened node median for these runs was around 126000 for A? and about 92000 for A!, but means don't fully reflect this because of the great variance between runs. That the explicitly randomized A? behaves erratically is not that surprising, but that the implicit randomness inherent in A! produces such volatility is unexpected.

This is where the visualization gives an idea of what is going on: A! seems to tread similar paths as A* and A?, but as the peaks appear noticeably dimmer, it spends less resources in exploring a given neighborhood. Intuitively, this could mean that the secondary heuristic is working as intended, directing the search beyond random browsing, but more data is needed to really back this claim.

While these two discussed instances lead to no conclusive results, the path diversity concept seems to work well together with fundamental data exploration measures mean, ratio and standard deviation. Visualizing the search space with a self-organizing map gives a useful new perspective on the nature of the algorithms, and helps highlight the similarities and differences between the three methods.

4.3.4 Hybrid performance

The diminishing returns from adding agents to any of the methods explored here is due to search overhead: new search workers mostly tread on paths that have already been explored by others. Path diversity is essential for A! in the sense that the secondary heuristic does not reach its full potential without sufficiently spread out search agents. There are many ways to tackle the issue of search overhead, such a various partitioning schemes, but in this section we consider a simple modification of the A! algorithm, that has the potential to increase exploration diversity.

From Table 4.2 we can see that, beyond A* taking search overhead to an extreme, A? tends to explore a more diverse set of states than A! – for better or worse. For the instance in question only a third of the states visited by a four

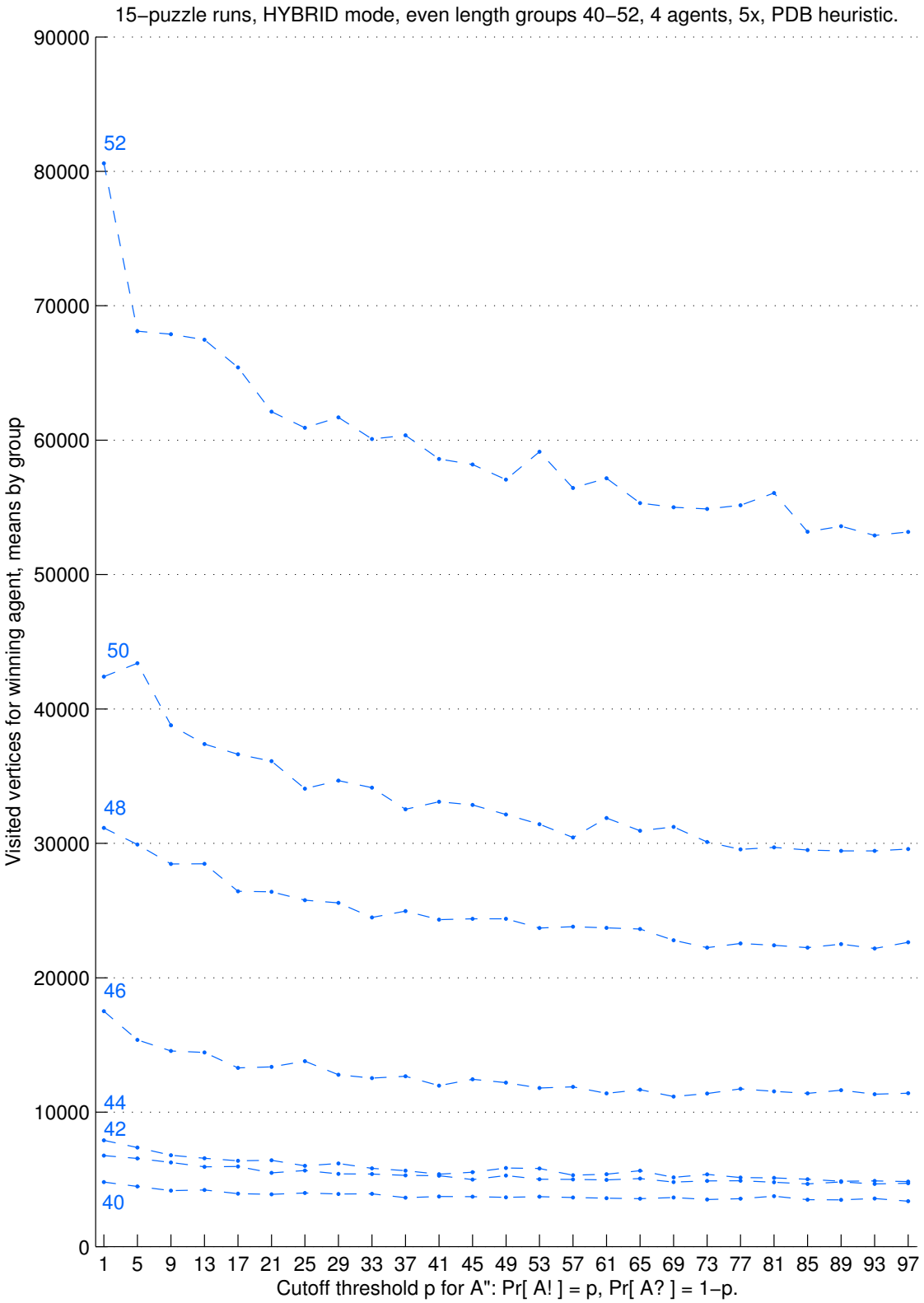


Figure 4.11: Hybrid A″ performance over multiple p-thresholds. Three iterations of 15-puzzle instances in the 45–54 range grouped by optimal path length. A! likelihood over A? grows with p to the right. The downward trending slopes suggest that adding some A? elements into A! does not improve the overall performance.

agent A! are unique, but for A? closer to a half of all states visited are unique, and there are about twice as many overall.

Ideally, we would like to have the focused search performance of A!, but also more diversity in the spirit of A? in one algorithm. As the two methods differ only in their selection policy, combining the two into a single approach is straightforward: we select between the two at random. This notion is captured in Algorithm 8 in Section 3.3.4, denoted A″. Randomly drawn values falling under threshold p give a round of A!, with A? being used otherwise.

Figure 4.11 shows how A″ performs on some medium sized 15-puzzle instances. Grouped again by optimal lengths, we see downward trends in the visited vertex count group averages for the winning agent of four as the cutoff threshold p is raised evenly from 1% to 97%. From left to right, the proportion of A! to A? increases, and no clear, consistent peaks emerge. The hybrid A″ does not represent an improvement: overall best results are obtained through unassisted A!.

While A! could likely benefit from more search diversification, augmenting the search with A? does not seem to be the right way to go about this. The secondary heuristic in A! appears to function in way that allows the algorithm to skip past the very states that A? is subject to waste time on.

4.4 Discussion

The n-puzzle experiments show that the cooperative A! algorithm outperforms both vanilla A* and the non-cooperative random parallel search A? in this problem context. A* is driven by a heuristic and expands the search in an orderly fashion and broadly in all directions, 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 randomization and parallelism, the search is less orderly and the agents can stumble on the right path faster than in systematic browsing, given enough agents. A! embraces concurrency and the parallel agents cooperate in focusing the search effort in areas that others have found promising. The secondary heuristic in A! serves as a global compass that augments the search effort, when the primary heuristic fails to disambiguate between candidates.

The experiments indicate that the nondeterministic cooperation emerging from asynchronous message exchange is beneficial at least for enabling the use of the secondary heuristic. The performance gains in A! might be explained by the active pursuit of search depth, information exchange having only a marginal impact, but on the other hand the cooperation mechanism does clearly influence the search. Without a shared, global sense of direction, the impact of any candidate ranking scheme is bound to be worthless: it is not enough to just go deep anywhere.

With more agents, the method was shown to work better, but with this simple

implementation, search overhead appeared to be an issue. The solver is not competitive, but then again that was never really a goal. Adding some randomization to A! in A″ did not help, and far more robust diversity increasing mechanisms are necessary for further scalability.

For the n-puzzle problems, A! demonstrated some extra sensitivity to heuristic improvement A*. The reason for this is not clear, but it is likely a fundamental factor in the method's overall performance – be it related to depth-first orientation or path diversity. Finally, the SOM visualization proved thought-provoking, but somewhat low-fidelity, and so a more appropriate visualization could help examine how search space exploration and cooperative mechanisms really go together.

Chapter 5

Related work

In this chapter we provide some additional background and references to research in the areas related to cooperative heuristic search, parallel A*, general concurrency and multi-agent cooperation.

5.1 Cooperative search

The idea of cooperative search is not a new one, but like parallel computing in general, it has somewhat fallen out of favor. Cooperative search enters the computing research lexicon in the late 1980s and early 1990s, a period of reduced funding and interest in AI technology known as the "AI winter" [115], when some researchers in the AI community started to think about parallel search in a new way.[1]

Parallel computing has a long history, and the benefit from running several independent search methods in parallel was well established early on, but first results in the constraint satisfaction context [26] suggested that keeping the methods isolated might not always be the best approach [63]. Instead of using portfolios of independent methods and relying on method variance alone, parallel search could be built around methods that actively pursue cooperation. More beneficial variance could be gained through the exchange and reuse of information gathered during the execution [63].

It had been previously observed that for many NP-hard problems, there is a distinct transition point at which problem instances begin to exhibit the kind of complexity and challenge that overcomes all algorithmic approaches [25, 71, 107, 130, 131]. These *phase transition* regions [60, 64, 107] – in analogy with various "energy barriers" found in physical processes – were characterized by high solution

[1] Curiously, the earliest references to *cooperative search* refer to collaborative communities of people in pursuit of a common goal, and, later, to cooperation between man and machine.

cost variance, and featured structures with large partial solutions that prevented early pruning by many kinds of heuristics [63].[2]

Clearwater, Hubermann and Hogg [26] presented results from constraint satisfaction search, where cooperative methods dominated non-cooperative ones for precisely these hard instances with many large partial solutions. Cooperation was found to be beneficial even in the case that most of the exchanged information turned out to be misleading [63].

Hogg and Williams [63] consider a cooperative mixture of two different search methods and apply this combination to graph coloring. The first method is a complete, depth-first backtracking search, while the other one begins with randomized initial configurations and uses heuristic repair to produce solutions. The methods exchange hints, partial solutions, with the help of a blackboard and variability is achieved through explicit, uniform randomness. Huberman [62] and Clearwater [26] helped develop these ideas. Kornfeld [81] was an early inspiration, already talking about the potential of concurrency in heuristic search.

The view to cooperation that Hogg and Williams present is of a master-slave kind, with a dominant uninformed search being augmented with a local optimization routine. Many of their ideas are applicable to the approach presented in this work, but we focus on cooperation that is more egalitarian in nature and specifically to cooperation that is realized through informed search and heuristic interaction. Instead of hints that are used to construct explicit candidates and establish areas of interest in the search space, the messaging in this work – while also very primitive – is more involved with building a sense of global direction on top of locally myopic heuristic search.

From the late 1990s onward, Michel Toulouse together with Theodore Crainic and collaborators began work on developing a theory of algorithmic cooperation and search in particular, beginning with parallel metaheuristics and culminating in the the taxonomy work presented in Sections 2.5 and 2.6 [30, 32, 125–127].

The cooperation results they present are inspired by *tabu search*, local search with a memory; *scatter search*, featuring search diversification (extrapolation) and intensification (interpolation) phases; *multi-level search*, where agents use different data sets and operate at a different level of abstraction; and genetic algorithms, colony methods and other bio-inspired approaches in *evolutionary computation*. Engelbrecht draws much of this work together in his book on *computational intelligence* [38]. Alba et al. [3, 4] offer extensive surveys on parallel metaheuristics.

Toulouse and Crainic posit that cooperation in parallel search stems from sharing gathered information among several sequential search programs, but emphasize that this is not mere hardware acceleration: the search patterns change completely

[2]Combinatorial search is a major driver behind interest in *quantum computing* and phase transitions have been theorized to exist also in that computational setting [61].

in cooperation [125]. There is a deeper system-wide process taking place during cooperative search, manifesting as ripple effects, self-organization, and stabilization towards attractors, but being poorly understood, it is hard to make use of.

The work in this thesis is heavily inspired by all of these ideas. The taxonomy of Toulouse and Crainic was used to derive a simple form of cooperation between agents and the emergent cooperation ideas are certainly at the heart of concurrent interaction. The techniques explored in computational intelligence, such as tabu memory and more extensively shared global state, could very well be applied to A!, but much of the previously described work is based on local search rather than pathfinding. It seems that these ideas have been explored less in informed parallel search of this kind.

The search for a theory of the emergent properties of cooperation has driven research on cooperative search to the fringes of computer science, where inspiration is drawn from nature, biology and sociology [31, 96, 126] as well as risk economics and game theory [88, 100].

More formal treatments of cooperation have also been proposed. Aldous and Vazirani [5] establish the family of *"Go with the winners"* cooperation algorithms, based on randomized particle set optimization, and offer an analysis of the key properties. Toulouse, Crainic and Sansó [126] focus on systemic control dependencies underlying cooperation. More recently Chazelle [24] has provided a solid footing for what he calls *natural algorithms*, a domain-independent language for describing complex interaction systems found in life sciences.

The work of Ouelhadj and Petrovic [98] and Barbucha [11] is probably closest to the work presented in this thesis. Ouelhadj and Petrovic build an agent-based hyper-heuristic framework in which agents can share best solutions of low-level heuristics and cooperate asynchronously, but while the approach is general they, too, are focused on local search. Barbucha also presents cooperating agents searching together for a solution to the vehicle routing problem, the focus being on multi-agent learning and evaluating aspects of communication. The same ideas are pursued in this work, but using different means and in a different context.

One reason for which cooperation is perhaps explored less in optimal algorithms of the kind that extend search paths from the root using the same heuristic information, is that A* is optimally efficient for any given consistent heuristic [115]. "No other optimal algorithm is guaranteed to expand fewer nodes than A* ", Russell and Norvig note, "(except possibly through tie-breaking among nodes with [a cost equal to the optimal path])." This, of course, is exactly what A! does.

Next, we'll review some parallel A* ideas and see to what extent cooperation has made an appearance there.

5.2 Parallel A*

To continue the brief review of A* topics started in Section 3.1.2, we now turn to parallel A*. Russell and Norvig [115] give a good overview of vanilla A*, but choose to not cover parallel search algorithms at length, claiming that it requires a lengthy discussion of parallel architectures. They note, however, that parallel search became a popular topic in the 1990s in both AI and theoretical computer science, and that the new multi-core and cluster computing era has brought it to the fore again.

First reports on the parallelization efforts of heuristic search are from the 1980s. Kumar, Ramesh and Rao [82] present a summary of the early results, featuring both centralized and distributed parallelism and various flavors of memory efficiency mechanisms. They make a note that parallel formulations primarily differ in the data structures they employ.

Powley and Korf [106] give a single-agent parallel window search formulation of *IDA** augmented with a global node ordering scheme. This is close to the ranking secondary heuristic idea employed in A!, but the approach simply tracks frontier nodes ands sorts them by $g(u)$-value adding momentum, but in a blind fashion.

Knight [73] presents a reactive agent oriented version of real-time-A*, with cooperative and noncooperative parallelism. Mahanti and Daniels [86] give a straightforward SIMD-parallel version of *IDA**. Grama and Kumar [48] give a comprehensive survey of basic parallel search algorithms, including studies on backtracking, A*, *IDA**, branch-and-bound, and dynamic programming methods. They also address issues such as load balancing, work-stealing, communication cost, scalability and speedup anomalies.

All of this research from close to two decades ago is worth exploring, when developing parallel search for specific applications, A! being no exception. Remarkably, many of the questions asked back then are still relevant today, with new applications and hardware only accentuating issues like load balancing.

Load balancing through hashing was an important development in parallel A*, and could well be a good candidate to improve diversity in A!. Evett et al. present hashing-based Parallel Retraction A* (PRA*) [41], an influential massively SIMD-parallel A* variant that makes good use of parallel computing resources and remains optimal for admissible heuristics.

Mahapatra and Dutt [87] use hashing extensively in load balancing and search-space partitioning. Later, Kishimoto et al. [72] picked up hashing for use in HDA*, hash distributed parallel A*, that scales to thousands of cores and terabytes of RAM. They emphasize simplicity in their approach and draw attention again to the notorious difficulty of parallel programming.

Another successful parallelization of A*, TDS, is focused on efficient state shar-

ing with distributed memory, and is based on transposition tables [113]. Further studies in search space partitioning have lead to to duplicate detection mechanisms [20] and external memory optimized distributed algorithms [133].

Without search space partitioning, parallelism can be achieved through parallelizing computation on the processing of individual nodes in a heavy graph, as was the case with the chess machine *Deep Blue* [21]. Algorithm portfolios and hybrids are another easy way to exploit parallelism, an example being the ManySAT solver [51]. Machine learning methods have been successfully applied to the discovery of well performing parallel configurations [27].

Classical planning tasks make good benchmarks and are often featured in parallel A* papers to give the methods credibility beyond puzzles [20, 55, 72]. Multi-agent formulations are sometimes given [75, 95], but cooperation effects do not appear to be studied directly. Information exchange, mostly considered from a search space partitioning and distributed load balancing angle, is not really considered from the point of view taken in this thesis. Most of A* parallelizations are deterministic and the rest probabilistic.

On the heuristics side, much of the efforts in parallel A* research relies on domain-independent methods rather than on improving the heuristic functions through parallel execution. It seems that the problem-specific heuristics research and A* parallelization efforts do not currently overlap much. The cooperative secondary heuristic idea of A! is related to the wealth of approaches explored in the parallel A* literature, but seems to not have been explicitly studied before.

Chapter 6

Conclusion

The goal of this thesis was to explore algorithmic cooperation with software agents in the context of heuristic search. The motivation behind this work is in making good use of multi-core computing hardware, and stems from the long-standing pursuit of parallel processing power. Specifically, this research focused on the effects of nondeterministic concurrency and the cooperation that emerges from asynchronous knowledge exchange between search agents.

The thesis has three main offerings. First – and foremost – we seek to establish that unconstrained cooperation, as a general purpose algorithmic idea, is worth pursuing in parallel computing. Second, we present a prototype cooperative heuristic search algorithm, A!, as a straightforward parallelization of the heuristic single-source shortest path algorithm A*. Third, we present an experimental study of A! in the standard benchmark n-puzzle toy problem context.

The hardware-oriented parallel computing tradition continues to deliver processing performance, but the software side is lagging behind. While peak performance is hard to achieve without low-level optimization, a higher level view is necessary for the design of everyday parallel software. The critical mainstream adoption of parallel programming requires abstractions that enable intuitive reasoning about systems with concurrently interacting parts.

Cooperation is the decisive difference between systems with many parts and systems that are more than the sum of their parts. Yet, cooperation is barely present in software today. Parallel programs often feature some internal communication, but this has traditionally been heavily confined and controlled: predictability and independence are favored over free interaction. Correctness is muddled up with performance. Some of this is dictated by the hardware reality, but this thesis argues that there is room to explore new parallel programming ideas. Cooperation is here promoted as one such idea.

The A! algorithm described in Chapter 3 is a cooperative search algorithm that uses two heuristic functions, one individual and one collective, to explore search

space in a collaborative way. It is based on nondeterministic concurrency and *implicit randomness* rather than explicit stochastic processes: instead of probabilities, the method relies on unpredictable event interleaving. Concurrency has a notorious reputation in shared-state parallel computing, so making use of it in this is a somewhat radical idea and worth exploring some more.

A! is a parallelization of A* inspired by multi-agent systems. The search workers in A! run distinct A* -like searches and share information about their progress. The secondary heuristic ranks nodes in the search frontier based on their distance to the globally best node. This is simple cooperation that is constructed on top of A*, meaning that most of the A* improvements should be compatible without issue. The use of the secondary heuristic is independent of the primary one and only used to differentiate: A! retains the optimality of A*.

The experimental results outlined in Chapter 4 indicate that A! performs well in finding optimal solutions to the n-puzzle. Specifically, A! explores less of the legal move search space than both vanilla A* and a random differentiation variant of A*. The performance of A! clearly improves with more search agents, but the returns are diminishing. A! is sensitive to heuristic improvement, but constrained by search overhead from limited path diversity. The secondary heuristic in A! appears to work locally as intended, with no gains being observed from hybridization with random selection.

While experimental success in a single problem does not merit much claim for achievement, this study at least suggests that the ideas underlying A! – cooperation and implicit randomness – are worth pursuing further. Compared to isolated parallelism, nondeterministic concurrency is applied to great effect in A!. Asynchronous exchange of information, the ripple effects of interaction, can be used to enhance and augment parallel execution.

The extent to which A! performance can be attributed to cooperation is unclear, but the experiments indicate that there is more than just a momentum effect at play: a depth-first emphasis on equal-value nodes is less useful without the direction of the secondary heuristic. A more detailed study of these factors could also help understand the nature of implicit randomness as it appears here.

If the concurrent interaction in A! would be linearized in a deterministic or an explicitly randomized way, one would likely see performance comparable to the results presented in this work. On the other hand the secondary heuristic is inherently dynamic, so imposing some specific event ordering might result in even less diversity. There currently is no good theory about algorithmic cooperation.

The work presented in this thesis could be continued in a number of ways. First of all, the A! algorithm should be validated in other search contexts, preferably using standard tools and benchmarks, such as the Fast Downward planner [54], to enable comparisons with recent work in the community [72, 94]. In current shape,

the n-puzzle A! is not competitive, but embedding these concurrent interaction ideas into a modern A* parallelization might give interesting results.

A! would likely benefit a great deal from a mechanism that diversifies the search effort more. Hybridization with a simple random selection did not appear to help, but a more elaborate diversification scheme might make search space coverage less redundant, bringing total explored node count and execution time down. In a similar vein exploring problem-specific path diversity visualizations further could help discover some structural properties in the target problems, which then might be useful in developing diversification methods.

In the implementation details, a more efficient heap data structure, optimized for top-k peek access, could have a visible impact on the individual node processing rate. The shared memory structure could also be optimized beyond the current all-independent configuration. Finally, the current communication constructs could be improved to minimize redundant traffic and increase shared information quality.

Overall, nondeterministic concurrency has here been shown to work well in the cooperative heuristic search setting and exploring these ideas in other parallel computing contexts is a natural next step. There is real value in cooperation and algorithms are no exception.

Bibliography

[1] Gul Agha. Actors: A Model of Concurrent Computation in Distributed Systems. Technical report, MIT AI Laboratory, June 1985.

[2] Selim G. Akl. *Parallel Computation: Models and Methods*. Prentice Hall, 1996.

[3] Enrique Alba, editor. *Parallel Metaheuristics*. John Wiley & Sons, 2005.

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

[5] David Aldous and Umesh Vazirani. 'Go with the winners' algorithms. In *Proc. of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 492–501, November 1994.

[6] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proc. of the 1967 AFIPS Spring Joint Computer Conference*, pages 483–485, April 1967.

[7] Alejandro Arbelaez and Youssef Hamadi. Improving Parallel Local Search for SAT. In *Proc. of the 5th Intl. Conf. on Learning and Intelligent Optimization (LION '05)*, pages 46–60, January 2011.

[8] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A View of Cloud Computing. *Communications of the ACM*, 53(4):50, April 2010.

[9] Krste Asanovic, John Wawrzynek, David Wessel, Katherine Yelick, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, and Koushik Sen. A View of the Parallel Computing Landscape. *Communications of the ACM*, 52(10):56, October 2009.

[10] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Computing Surveys*, 45(4):1–34, August 2013.

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

[12] Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J. Gutjahr. A Survey on Metaheuristics for Stochastic Combinatorial Optimization. *Natural Computing*, 8(2):239–287, 2009.

[13] Guy E. Blelloch and Bruce M. Maggs. Parallel Algorithms. In *Algorithms and Theory of Computation Handbook*, chapter 25. Chapman & Hall/CRC, 2010.

[14] Pradip Bose. Is Dark Silicon Real? *Communications of the ACM*, 56(2):92, February 2013.

[15] Sandy Brand and Rafael Bidarra. Parallel Ripple Search - Scalable and Efficient Pathfinding for Multi-core Architectures. In *Proc. of the 4th Intl. Conf. on Motion in Games (MIG '11)*, pages 290–303, 2011.

[16] Gerth S. Brodal. Worst-Case Efficient Priority Queues. In *Proc. of the 7th ACM-SIAM Symposium on Discrete Algorithms (SODA '96)*, pages 52–58, January 1996.

[17] Gerth S. Brodal and Chris Okasaki. Optimal Purely Functional Priority Queues. *Journal of Functional Programming*, 6(6):839–857, 1996.

[18] Peter A. Buhr and Ashif S. Harji. Concurrent Urban Legends. *Concurrency and Computation: Practice and Experience*, 17(9):1133–1172, August 2005.

[19] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A Survey of the State of the Art. *Journal of the Operational Research Society*, 64(12):1695–1724, December 2013.

[20] 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.

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

[22] Anthony M. Castaldo and R. Clint Whaley. Scaling LAPACK Panel Operations Using Parallel Cache Assignment. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*, pages 223–232, 2010.

[23] K. Mani Chandy and Jayadev Misra. *Parallel Program Design - A Foundation*. Addison-Wesley, 1989.

[24] Bernard Chazelle. Natural Algorithms and Influence Systems. *Communications of the ACM*, 55(12):101–110, December 2012.

[25] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proc. of the 12th Intl. Joint Conference on Artificial Intelligence (IJCAI '91)*, volume 1, pages 331–337, August 1991.

[26] Scott H. Clearwater, Bernardo A. Huberman, and Tad Hogg. Cooperative Solution of Constraint Satisfaction Problems. *Science*, 254(5035):1181–3, November 1991.

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

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd ed.* The MIT Press, 2009.

[29] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design, 5th ed.* Addison-Wesley, 2012.

[30] Teodor G. Crainic and Nourredine Hail. Parallel Meta-Heuristics Applications. In *Parallel Metaheuristics*, chapter 19. John Wiley and Sons, 2005.

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

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

[33] Joseph Culberson and Jonathan Schaeffer. Searching with Pattern Databases. In *Proc. of the 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence (Canadian AI '96)*, pages 402–416, 1996.

[34] Joseph Culberson and Jonathan Schaeffer. Pattern Databases. *Computational Intelligence*, 14(3):318–334, 1998.

[35] Edsger W. Dijkstra. A Note on Two Problems In Connexion With Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[36] Jim. E. Doran and Donald Michie. Experiments with the Graph Traverser. *Proceedings of the Royal Society*, 294:235–259, 1966.

[37] Michel Dubois, Murali Annavaram, and Per Stenström. *Parallel Computer Organization and Design*. Cambridge University Press, 2012.

[38] Andries P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley Publishing, December 2007.

[39] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power Challenges May End the Multicore Era. *Communications of the ACM*, 56(2):93, February 2013.

[40] Patrick T. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[41] 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.

[42] Henri Farreny. Completeness and Admissibility for General Heuristic Search Algorithms - A Theoretical Study: Basic Concepts and Proofs. *Journal of Heuristics*, 5(3):353–376, 1999.

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

[44] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[45] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[46] Anwar Ghuloum. Viewpoint: Face the Inevitable, Embrace Parallelism. *Communications of the ACM*, 52(9):36, September 2009.

[47] Gregory Goth. Entering a Parallel Universe. *Communications of the ACM*, 52(9):15, September 2009.

[48] Ananth Grama and Vipin Kumar. Parallel Search Algorithms for Discrete Optimization Problems. *ORSA Journal on Computing*, 7(4):365–385, November 1995.

[49] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(3):12–21, August 1993.

[50] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

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

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

[53] 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.

[54] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26(1):191–246, July 2006.

[55] Malte Helmert, Patrik Haslum, and Jörg Hoffmann. Flexible Abstraction Heuristics for Optimal Sequential Planning. In *Proc. of the 17th Intl. Conf. on Automated Planning and Scheduling (ICAPS '07)*, pages 176–183, 2007.

[56] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[57] Rich Hickey. Values and Change - Clojure's Approach to Identity and State. Technical report, Clojure.org, 2012.

[58] Pieter Hintjens, editor. *ØMQ - The Guide*. iMatix Corporation and collaborators, 2014.

[59] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[60] Tad Hogg. Phase Transitions and the Search Problem. *Artificial Intelligence*, 81(1-2):1–15, March 1996.

[61] Tad Hogg. Quantum Computing and Phase Transitions in Combinatorial Search. *Journal of Artificial Intelligence Research*, 4(1):91–128, January 1996.

[62] 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.

[63] Tad Hogg and Colin P. Williams. Solving the Really Hard Problems with Cooperative Search. In *Proc. of the 11th National Conference on Artificial Intelligence (AAAI '93)*, pages 231–236, July 1993.

[64] Bernardo A. Huberman and Tad Hogg. Phase Transitions in Artificial Intelligence Systems. *Artificial Intelligence*, 33(2):155–171, October 1987.

[65] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An Economics Approach to Hard Computational Problems. *Science*, 275(5296):51–54, January 1997.

[66] Shams M. Imam and Vivek Sarkar. Integrating Task Parallelism with Actors. *ACM SIGPLAN Notices*, 47(10):753, November 2012.

[67] Shahab Jabbari Arfaee, Sandra Zilles, and Robert C. Holte. Learning Heuristic Functions for Large State Spaces. *Artificial Intelligence*, 175(16-17):2075–2098, October 2011.

[68] Joseph JaJa. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[69] Donald B. Johnson. Priority Queues with Update and Finding Minimum Spanning Trees. *Information Processing Letters*, 4(3):53–57, December 1975.

[70] William W. Johnson and William E. Story. Notes on the "15" Puzzle. *American Journal of Mathematics*, 2(4):397–404, 1879.

[71] Richard M. Karp and Judea Pearl. Searching For an Optimal Path in a Tree With Random Costs. *Artificial Intelligence*, 21(1-2):99–116, 1983.

[72] 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.

[73] Kevin Knight. Are Many Reactive Agents Better Than A Few Deliberative Ones? In *Proc. of the 13th Intl. Joint Conference on Artifical Intelligence (IJCAI '93)*, volume 1, pages 432–437, 1993.

[74] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd ed.* Addison-Wesley, 1973.

[75] Sven Koenig. Agent-Centered Search. *AI Magazine*, 22(4):109–131, October 2001.

[76] Sven Koenig, Maxim Likhachev, and David Furcy. Lifelong Planning A*. *Artificial Intelligence*, 155(1-2):93–146, May 2004.

[77] Teuvo Kohonen. *Self-Organizing Maps*. Springer, 2001.

[78] Richard E. Korf. Depth-First Iterative-Deepening: An optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, September 1985.

[79] Richard E. Korf. Recent Progress in the Design and Analysis of Admissible Heuristic Functions. In *Proc. of the 4th International Symposium on Abstraction, Reformulation, and Approximation (SARA '00)*, pages 45–55, 2000.

[80] Richard E. Korf and Ariel Felner. Disjoint Pattern Database Heuristics. *Artificial Intelligence*, 134(1-2):9–22, January 2002.

[81] 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.

[82] Vipin Kumar, K. Ramesh, and V. Nageshwara Rao. Parallel Best-First Search of State-Space Graphs: A Summary of Results. In *Proc. of the 7th National Conference on Artificial Intelligence (AAAI '88)*, pages 122–127, 1988.

[83] Alexandre Le Bouthillier, Teodor G. Crainic, and Peter Kropf. A Guided Cooperative Search for the Vehicle Routing Problem with Time Windows. *IEEE Intelligent Systems*, 20(4):36–42, August 2005.

[84] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[85] John A. Lee and Michel Verleysen. *Nonlinear Dimensionality Reduction*. Springer, 2007.

[86] Ambuj Mahanti and Charles J. Daniels. A SIMD Approach to Parallel Heuristic Search. *Artificial Intelligence*, 60(2):243–282, April 1993.

[87] Nihar R. Mahapatra and Shantanu Dutt. Scalable Global and Local Hashing Strategies for Duplicate Pruning in Parallel A* Graph Search. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):738–756, July 1997.

[88] Efrat Manisterski, David Sarne, and Sarit Kraus. Cooperative Search with Concurrent Interactions. *Journal of Artificial Intelligence Research*, 32:1–36, 2008.

[89] Bruce Martin. Concurrent Programming vs. Concurrency Control: Shared Events or Shared Data. *ACM SIGPLAN Notices*, 24(4):142–144, April 1989.

[90] Paul E. McKenney. Beyond Expert-Only Parallel Programming? In *Proc. of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*, pages 25–31, October 2012.

[91] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* The Linux Kernel Organization, 2014.

[92] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, December 1995.

[93] Stephen A. Neuendorffer. *Agent-Oriented Metaprogramming*. PhD thesis, University of California, Berkeley, 2004.

[94] Raz Nissim and Ronen Brafman. Multi-Agent A* For Parallel and Distributed Systems. In *Proc. of the 11th Intl. Conf. on Autonomous Agents and Multiagent Systems (AAMAS '12)*, pages 1265–1266, June 2012.

[95] Raz Nissim, Ronen Brafman, and Carmel Domshlak. A General, Fully Distributed Multi-Agent Planning Algorithm. In *Proc. of the 9th Intl. Conf. on Autonomous Agents and Multiagent Systems (AAMAS '10)*, pages 1323–1330, 2010.

[96] Geoff Nitschke. Emergence of Cooperation: State of the Art. *Artificial Life*, 11(3):367–96, January 2005.

[97] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

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

[99] Liviu Panait and Sean Luke. Cooperative Multi-Agent Learning: The State of the Art. *Autonomous Agents and Multi-Agent Systems*, 11(3):387–434, 2005.

[100] David C. Parkes and Bernardo A. Huberman. Multiagent Cooperative Search for Portfolio Selection. *Games and Economic Behavior*, 35(1-2):124–165, April 2001.

[101] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[102] Judea Pearl and Jin H. Kim. Studies in Semi-Admissible Heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(4):392–399, 1982.

[103] Simon Peyton Jones. The Future is Parallel, and the Future of Parallel is Declarative. *YOW! Australian Developer Conference*, 2011.

[104] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Proc. of the 28th IARCS Ann. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '08)*, volume 2, pages 383–414, 2008.

[105] Ira Pohl. The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting and Computational Issues in Heuristic Problem Solving. In *Proc. of the 3rd Intl. Joint Conference on Artificial Intelligence (IJCAI '73)*, pages 12–17, 1973.

[106] Curt Powley and Richard E. Korf. Single-Agent Parallel Window Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477, May 1991.

[107] Paul Walton Purdom Jr. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence*, 21(1-2):117–133, March 1983.

[108] Daniel Ratner and Manfred Warmuth. Finding a Shortest Solution for the NxN Extension of the 15-PUZZLE Is Intractable. In *Proc. of the 4th National Conference on Artificial Intelligence (AAAI '86)*, pages 168–172, 1986.

[109] Daniel Ratner and Manfred Warmuth. The (N^2-1)-puzzle and Related Relocation Problems. *Journal of Symbolic Computation*, 10(2):111–137, July 1990.

[110] Thomas Rauber and Gudula Rünger. *Parallel Programming for Multicore and Cluster Systems, 2nd ed.* Springer, 2010.

[111] Yves Rochat and Éric D. Taillard. Probabilistic Diversification and Intensification in Local Search for Vehicle Routing. *Journal of Heuristics*, 1(1):147–167, 1995.

[112] David P. Rodgers. Improvements in Multiprocessor System Design. *ACM SIGARCH Computer Architecture News*, 13(3):225–231, June 1985.

[113] John W. Romein, Aske Plaat, Henri E. Bal, and Jonathan Schaeffer. Transposition Table Driven Work Scheduling in Distributed Search. In *Proc. of the 15th National Conference on Artificial Intelligence (AAAI '99)*, pages 725–731, 1999.

[114] Stuart Russell. Efficient Memory-Bounded Search Methods. In *Proc. of the 10th European Conference on Artificial Intelligence (ECAI '92)*, pages 1–5, August 1992.

[115] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, 3rd ed.* Prentice Hall Press, 2009.

[116] Tomas Salamon. *Design of Agent-Based Models: Developing Computer Simulations for a Better Understanding of Social Processes.* Bruckner, 2011.

[117] John E. Savage. *Models of Computation - Exploring the Power of Computing.* Addison-Wesley, 1998.

[118] Tommaso Schiavinotto and Thomas Stützle. A Review of Metrics on Permutations for Search Landscape Analysis. *Computers & Operations Research*, 34(10):3143–3153, October 2007.

[119] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel Actor Monitors: Disentangling Task-Level Parallelism From Data Partitioning in the Actor Model. *Science of Computer Programming*, 80:52–64, 2014.

[120] Yoav Shoham. Agent-Oriented Programming. *Artificial Intelligence*, 60(1):51–92, March 1993.

[121] Yoav Shoham and Kevin Leyton-Brown. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations.* Cambridge University Press, 2009.

[122] Jerry Slocum and Dic Sonneveld. *The 15 Puzzle Book.* The Slocum Puzzle Foundation, 2006.

[123] Anthony Stentz. The Focussed D* Algorithm For Real-Time Replanning. In *Proc. of the 14th International Joint Conference on Artificial Intelligence (IJCAI '95)*, volume 2, pages 1652–1659, August 1995.

[124] Xian-He Sun and Lionel M. Ni. Another View on Parallel Speedup. In *Proc. of the 1990 ACM/IEEE Conference on Supercomputing (Supercomputing '90)*, pages 324–333, October 1990.

[125] Michel Toulouse, Teodor G. Crainic, and Brunilde Sansó. An Experimental Study of Systemic Behavior of Cooperative Search Algorithms. In *Meta-Heuristics*, chapter 26, pages 373–392. Springer, 1999.

[126] Michel Toulouse, Teodor G. Crainic, and Brunilde Sansó. Systemic Behavior of Cooperative Search Algorithms. *Parallel Computing*, 30(1):57–79, January 2004.

[127] Michel Toulouse, Krishnaiyan Thulasiraman, and Fred Glover. Multi-level Cooperative Search: A New Paradigm for Combinatorial Optimization and an Application to Graph Partitioning. In *Proc. of the 5th Intl. Euro-Par Conference on Parallel Processing (Euro-Par '99)*, pages 533–542, 1999.

[128] Richard A. Valenzano, Jonathan Schaeffer, and Nathan R. Sturtevant. Finding Better Candidate Algorithms for Portfolio-Based Planners. *Proc. of the 23rd Intl. Conf. on Automated Planning and Scheduling (ICAPS '13)*, 2013.

[129] Danny Weyns, Andrea Omicini, and James Odell. Environment As a First Class Abstraction in Multiagent Systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):5–30, 2007.

[130] Colin P. Williams and Tad Hogg. Using Deep Structure to Locate Hard Problems. In *Proc. of the 10th National Conference on Artificial Intelligence (AAAI '92)*, pages 472–477, July 1992.

[131] Colin P. Williams and Tad Hogg. Exploiting the Deep Structure of Constraint Problems. *Artificial Intelligence*, 70(1-2):73–117, October 1994.

[132] Michael J. Wooldridge. *An Introduction to MultiAgent Systems, 2nd ed.* Wiley, 2009.

[133] 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, July 2004.