Parallelizing Heuristic Computations for Planning Problems

by

Michael Nowicki

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

HONOURS IN COMPUTER SCIENCE

in

THE IRVING K. BARBER SCHOOL OF ARTS AND SCIENCES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

April 2015

Abstract

This work looks to improve upon the performance of state space search techniques such as iterative deepening by deferring the more computationally expensive part of the search among a number of threads. Previous work on parallel state space searches involves splitting the state space among threads or processors. In this work we investigate the potential of an alternative approach in which a primary thread performs a traditional A* search while the remaining threads compute the heuristic values of newly generated states. The other focus in this work is the Causal Graph heuristic. We reformulate the causal graph heuristic as a constrained shortest path problem with the labels of the transitions acting as the constraints. Our initial results demonstrate that our approach to the parallelization of state space search algorithms is able to provide sub-linear speed-ups over the traditional A* search algorithm, with the ability to achieve linear to super-linear speed-ups in certain domains. With these speed-ups the algorithms are able to solve more planning tasks than the sequential algorithm.

Preface

This Honours Thesis is an original, unpublished work done independently by the author, Michael Nowicki.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank Dr. Yong Gao for his input and guidance throughout this project. His efforts were invaluable in the formulation of the algorithms proposed. He also helped guide the writing of this work, making sure that the material is presented in a clear, formulaic manner.

Additionally I would like to thank the other students in our Honours Group who were willing to listen to ideas not related to their own topics, and provided input that helped in the development of this work.

Chapter 1

Introduction

State space search is a major area of work in artificial intelligence as it is fundamental to automate problem solving. There are many algorithms that have been designed to search state spaces efficiently either using a greedy method such as Best-First Search to select the most promising nodes on the frontier of the state space, or using a memory bound algorithm such as Iterative Deepening. As modern processors have developed a major trend has been the creation of microprocessors with multiple logical cores to provide additional computing power. As a result it has become a main point of research to examine ways in which algorithms can be modified into parallel equivalents to take advantage of the additional computational power available. This work looks to contribute a modification to the traditional A* and Iterative Deepening A* (IDA*) algorithms. These search algorithms use heuristics to guide the search and more accurate heuristics can be more computationally intensive to compute. For this reason we look to compute the heuristic values in parallel while performing a state space search.

Traditional planners explore the state space using greedy or best first strategies such as Greedy Best First Search (GBFS), A*, and Hill Climbing [RN10, Hel04, BG01]. We expand upon IDA*[Kor85] which performs a cost-bounded depth first search through the state space. The difficulty with finding a solution to a planning task is that it has been shown to be **PSPACE**-complete[Byl94]. In an effort to overcome this one major area of work has been to parallelize search algorithms to traverse the state space more efficiently. A wide variety of algorithmic techniques have been proposed along this line including mapping states to processors[KFB13], partitioning the state space[Mah11], and running multiple iterative deepening searches with different depth bounds for each processor[PFK93].

To guide a best first search heuristics are used to approximate the distance from the goal state. This is done by relaxing the problem and attempting to find the solution to a more simple problem. One well known technique is called *delete relaxation*. Heuristics formulated using this technique ignore the negative effects of the operators in a planning problem[HN01]. Computing the additive heuristic h^+ is an **NP**-hard problem and in practice approximations of the heuristic are used. However, these approximations provide inadmissible estimates making the heuristics inadmissible[HD09]. These heuristics are useful in practice and are implemented in a number of state-of-the-art planners, however ignoring the delete effects of operators and relaxing the task further to form an approximation can ignore too much information about the task[BG01]. Several other heuristics have been proposed to overcome these difficulties such as the use of critical paths [HG00], landmarks [KD09] and abstraction heuristics such as pattern databases. [Ede01]

An alternative heuristic that has been developed is known as the Causal Graph Heuristic [Hel04]. This heuristic is used to capture relationships between variables to model how they interact through different actions. The heuristic uses two directed graph structures known as the causal graph and domain transition graphs to model dependencies between variables, and how variables can change their values.

In this work we examine parallel heuristic evaluation of states while performing a state-space search for planning problems, as well as propose an extension to the causal graph heuristic. Experimental results demonstrate that if a computationally intensive heuristic is used then a significant speed-up can be achieved without incurring large space or communication overhead that typically arises in parallel heuristic search.

1.1 Problem Representation

Planning problems traditionally use propositional logic to represent the task and this language is known as Planning Domain Definition Language (PDDL)[Mcd00]. PDDL represents a task as a tuple $\langle P, \mathcal{O}, s_0, s_* \rangle$, where P is a finite set of propositional statements which can be either positive or negative, $\mathcal{O}$ is a finite set of operators with pre- and post-conditions, s_0 is the initial state of the task and s_* is the goal state. An operator can be applied to a state if the current state satisfies an operators preconditions. A plan is then defined to be the sequence of operations that can be applied to transform the initial state to a goal state[RN10].

Although planning tasks are traditionally represented using propositional logic for the purpose of the heuristic used the problem must be translated into an alternate form. This is known as the SAS^+ planning language [BN93], in which variables can be defined to have multiple values within a finite domain. Problems represented in this form are known as Multivalued Planning Tasks (MPT). Formally an MPT can be defined as follows:

Definition 1.1. Multi-valued Planning Task
A MPT is a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_*, s_g \rangle$ where:

- $\mathcal{V}$ is a finite set of variables, and each variable $v \in V$ has a finite domain $\mathcal{D}_v$.
- $\mathcal{O}$ is a finite set of operators, where each operators is defined as the tuple $\langle pre, eff \rangle$.
- s_0 is the initial state of the planning task.
- s_* is a subset of variables of the planning task representing the goal of the task.

In a planning task a state is a mapping for each $v \in \mathcal{V}$ to a particular value $s(v) \in \mathcal{D}_v$, and the domain size of each variable is a finite integer. A plan is then defined as a sequence of operators that can be applied beginning at s_0 that achieves s_*. An operator can be applied to a state s if $s(v) = \text{pre}(v)$ whenever $\text{pre}(v)$ is defined. Applying the operator leads to a successor state such that whenever $\text{eff}(v)$ is defined: $s(v) \leftarrow \text{eff}(v)$. For the purposes of this work we assume operators have uniform action costs and the task does not contain conditional effects or axioms.

To illustrate the concepts in this work we will use a simple instance from the gripper domain. In this problem there is a robot that is free to move between two rooms and has two arms. Initially one room is filled with a number of balls, and the task is for the robot to move all the balls from one room to another. To keep the example size small we will focus on the task involving four balls. When the problem is converted into a multi-valued planning task there are four variables representing the state of the balls, one representing the state of the robot, and one variable representing the state of each arms. Using the translator implemented for Fast-Downward[Hel06] the problem can be converted to the MPT formalism and the world can be represented using seven multi-valued variables. Operators are also converted so that the preconditions and effects modify the variables in their domain. An example of the problem representation in Figure 1.1.

The rest of this work is organized as follows. The next chapter provides a literature review of work done on parallel heuristic searches and the

Variables:
b1, b2, b3, b4 ϵ {*at-room-a, at-room-b, carried*}
r ϵ {*at-room-a, at-room-b*}
a1, a2 ϵ {*carry-ball-1, carry-ball-2, carry-ball-3, carry-ball-4, free*}
Operators:
Operator 1: *move rooma roomb*
  Preconditions: r = *at-room-a*
  Effects: r = *at-room-b*
Operator 2: *pick ball1 rooma left*
  Preconditions: r = *at-room-a*, b1 = *at-room-a*, a1 = *free*
  Effects: a1 = *carry-ball-1*, b1 = *carried*
  ...
Operator 34: *drop ball4 roomb right*
  Preconditions: r = *at-room-b*, a2 = *carry-ball-4*, b4 = *carried*
  Effects: a2 = *free*, b4 = *at-room-b*
Initial State:
b1 = *at-room-a*, b2 = *at-room-a*, b3 = *at-room-a*, b4 = *at-room-a*
r = *at-room-a*, a1 = *free*, a2 = *free*
Goal:
b1 = *at-room-b*, b2 = *at-room-b*, b3 = *at-room-b*, b4 = *at-room-b*

Figure 1.1: Representing the Gripper Problem as an MPT

causal graph heuristic. The third chapter discusses the parallel algorithms implemented for this paper, with the fourth chapter outlining our proposed modification of the causal graph heuristic. The fifth chapter provides the empirical results from testing the planner on common benchmark problems, and the final chapter provides a conclusion and thoughts for future work.

Chapter 2

Literature Review

In this chapter we will formally review some of the previous work that has been done with parallel planning problems and best-first searches, as well as discuss the causal graph heuristic and the graph structures that underline it.

2.1 Informed State Space Search

Before discussing the parallel programming techniques for state space searches we will review the graph search techniques fundamental to them. The A* algorithm is a heuristic search of a state space, combining the strengths of Dijkstra's Single-Source Shortest Path algorithm and Greedy Best First Search (GBFS). Like Dijkstra's algorithm a frontier is expanded from the initial node and a priority queue, called the open list, is used to maintain an ordering of the nodes in the frontier. Nodes are ordered not by their path cost g, as in Dijkstra's, or their heuristic estimates h, as in GBFS, but the linear combination of the two $f = g + h$. As nodes are expanded they are stored in a closed list to detect duplicates. If the node is already in the open list it retains only the one with the lower path cost (g-value).

The choice of heuristic implemented with the A* algorithm greatly effects the algorithms performance. If the heuristic used is admissible then A* is optimal if performing a tree search. If the state space is a graph the heuristic must satisfy the stronger condition of *consistency* for A* to be optimal for a graph search[RN10]. A heuristic is consistent if for every successor node n' of n the heuristic estimate of n is never more than the heuristic estimate of the cost of reaching the successor plus its heuristic estimate. This results in a general triangle inequality:

$$h(n) \leq cost(n, n') + h(n') \tag{2.1}$$

Another strength of A* is that it is optimally efficient compared to GBFS or uniform-cost searches. If A* is implemented with a consistent heuristic and the optimal solution has cost C' it is guaranteed to expand every node with $f(n) < C'$ and therefore will expand the the minimal set of nodes to reach the goal state[RN10]. Greedy searches such as GBFS risk heading towards dead ends or plateauing when not considering the path cost of reaching a state. Finally, A* is a complete search algorithm when the branching factor b is a finite number and each step costs more than some ϵ [RN10]. The A* search algorithm has many strengths for performing an informed search of the state space, however the algorithm suffers from the memory overhead incurred from maintaining the open and closed lists during search. For this reason work has been done to combine the strengths of A* search with more memory efficient searchs such as depth first search.

A commonly used memory-bounded extension of A* search is known as Iterative Deepening A*[Kor85]. The work done by Korf was focused on analyzing different tree search algorithms in terms of time and space complexity as well as the quality of the solutions they detect. The primary algorithm discussed is Depth-First Iterative Deepening (DFID) which performs a DFS to a specific depth, incrementing the depth until a solution is found. At the end of each iteration all the nodes that have been explored are discarded and the search restarts from the root node. Given the algorithm starts with depth 1 and continues until a solution is found the algorithm returns an optimal solution. If the solution is at depth d in the search tree the algorithm requires $\mathcal{O}(d)$ space.

Algorithm 1: IDA*

```
Input: Initial state, goal state, and applicable operators
Output: If a path to the goal is found or not
1  $h \leftarrow heuristic\_value(initial\_state)$;
2  $bound \leftarrow h$;
3  $min \leftarrow \infty$;
4  plan = $\langle\rangle$;
5  while $plan == \langle\rangle$ do
6      result ← DFS(initial_state, 0, bound);
7      if $result == \infty$ then
8          return NO_PLAN;
9      end
10     else
11         return PLAN;
12     end
13     bound ← result;
14 end
```

The branching factor of a problem, b, is defined as the number of new successors that can be generated by applying a single operator to a node, averaged among every node in the state space[Kor85]. Since this algorithm must check every path in the state space before going to the next iteration [Kor85] proved that DFID has an asymptotic time complexity $\mathcal{O}\left(b^d\right)$. He also demonstrates that the additional computations needed to regenerate nodes each iteration does not impact the overall complexity of the search as the majority of the work of the algorithm is performed during the final iteration of the search.

Algorithm 2: DFS

```
Input: Current state, current path cost $g$, bound
Output: Minimum distance to leaf node or goal
1  $f \leftarrow g + compute\_heuristic(state)$;
2  if $f > bound$ then
3      return f;
4  end
5  if is_goal(state) then
6      plan ← trace_path(state);
7      return f;
8  end
9  min ← $\infty$;
10 for each successor of state do
11     result ← DFS(successor, g+1, bound);
12     if $plan \neq \langle\rangle$ then
13         return result;
14     end
15     if $result < min$ then
16         min ← result;
17     end
18 end
19 return min;
```

The major result of this work is that if IDA* is combined with a consistent heuristic it is an optimal algorithm with respect to time, space, and solution quality compared to the set of optimal tree-search algorithms[Kor85]. Combined with the fact that the IDA* algorithm is much more simple to implement as there are no open and closed lists to maintain it is a natural starting point to try to extend the algorithm to exploit the additional computing power available with modern microprocessors.

2.2 Parallel Informed State Space Search

One of the major trends in microprocessor development has been decreasing clock speeds with an increase in the number of logical processing units. As a result there has been a large focus on developing parallel algorithms to take advantage of the additional computing power provided by having multiple processors. Work on parallel algorithms has been ongoing for several decades, focusing on improving the scalability of parallel algorithms as the number of cores increases. As hardware and algorithmic techniques have advanced new extensions to the original ideas have shown improvement in speed-up and efficiency.

When considering the task of parallelizng a best-first search algorithm an intuitive idea would be to use one open and closed list that all threads are able to access. To ensure that all data read and writes are valid these data structures must be locked before a thread can access them. Burns et. al. [EBZ10] implemented this technique on four-way grid path-finding problems and demonstrated how this strategy performs worse than sequential A*. Because of the locks on the open and closed lists processors remain idle trying to access the lists. This *synchronization overhead* leads to a higher run time and their results demonstrate that this technique does not scale well. When designing parallel algorithms it is necessary to be aware of any type of parallel overhead that can be created by the algorithm[Bar14].

In previous work done on parallel heuristic search there have been three types of overhead that effect parallel performance. The first is synchronization overhead mentioned above where processors must wait for all threads to reach a common synchronization point. The second is communication overhead which is caused by delays as processors send and receive information from each other. The final type is search overhead, which is the additional search time needed in parallel to complete the search. While the final type of overhead may seem counter intuitive, it contributes the most to the total overhead of parallel heuristic search algorithms[KFB13]. This is because when sequential A* reaches a goal state the algorithm guarantees that the state was reached optimally, however parallel searches may find alternate solutions and will not terminate until all processors can guarantee the optimality of a solution[KFB13].

There have been many different techniques used to design parallel heuristic search algorithms. We will look at some of the algorithmic techniques which were used as a base to develop our parallel algorithms and analyze the empirical results they obtained.

2.2.1 Parallelizing State Expansion

One of the first parallel implementations of the A* algorithm is known as Parallel Retraction A* (PRA*) [EMNH95]. This algorithm is a parallel implementation of a sequential algorithm called Retracting A* which functions similarily to A*. A retraction scheme is used to avoid exhausting memory, nodes in the frontier could be retracted into their parents by storing their f values in the parent nodes to free space[EMNH95]. That parent node is then reinserted into the open list to potentially be expanded again.

The PRA* algorithm splits the task of expanding states among different processors and was designed for a distributed memory system. Each processor maintains a bucket of states to expand and when a state is generated a hash function is used to determine which processor will work on that state. The choice of hash function is extremely important to make sure that states are distributed evenly among processors, a technique known as load balancing[EMNH95].

A message passing interface is used to broadcast the state to the destination processor. The receiving processor checks its bucket to see if the state is a duplicate of one it currently has. These two operations resulted in being the most time consuming portions of the search[EMNH95]. This led to significant communication overhead in their implementation as the processor that generates a state must wait until the receiving processor broadcasts that it has received the state[KFB13]. Their work demonstrated near linear speed-ups and scale up to thousands of processors with limited memory[EMNH95] .Their implementation was affected by large overhead caused by using synchronous communications between processors[VBH10].

One extension of this work was created by Kishimoto, Fukunaga, and Botea called Hash Distributed A* (HDA*) and was designed to run on both distributed memory systems and multi-core processors with shared memory[KFB13]. Their implementation was built using the MPI framework which allows for synchronous and asynchronous communication between processors. One of the aims of their work was to minimize the amount of overhead incurred by parallel search, and the MPI framework allows processors to send and receive states from other processors using optimized, non-blocking methods[KFB13].

Experimental analysis of HDA* was conducted on shared and distributed memory systems with up to 128 processors and 128 GB of RAM built on the

Fast Downward planner. On these distributed memory systems their results demonstrate speed-ups of 30-60 times that of the sequential algorithm of a state-of-the-art sequential planner. On shared memory systems with 8 cores speed-ups of 3.6-6.3 are reported. Additionally their implementation maintains all processors at nearly 100% load during search as all communication is asynchronous and no additional load balancing was needed to distribute states[KFB13].

Another way to partition the planning task for multi-agent searching is to give each processor disjoint sets of operators to search with, which is called Multi-Agent A* (MA-A*)[NB12]. With this technique no two agents will generate the same children from a state while performing an A* search. To ensure agents expand states that are relevant to their set of operators [NB12] developed a message passing scheme to broadcast states to processors where the state satisfies the preconditions of one or more operators. The receiving processor checks to see if its open list already contains the state, updating if needed to the state found with a lower path cost. They found that passing states to other processors once they were expanded resulted in the best performance as it kept the communication overhead to a minimum.

The first goal state reached might not be the optimal solution so a different termination condition needs to be used to control the search. Their algorithm terminates once it can be verified that no processor contains a state in its open list with a cost less than the goal state found, and implemented this in a way that does not need to pause the search to examine the open list of each processor. This ensures that the algorithm returns the optimal solution to reach the goal state.

In their evaluations they implemented two versions of MA-A*. The first allowed all agents the access to a global heuristic function that has completed knowledge of the task. Their second implementation had each processor operate with a private heuristic, allowing each agent to search the state space in its own way unique to the set of operators it owns. Their results demonstrated super-linear speed ups in some situations which they attribute to the additional pruning gained as each agent partially expands a state, as well as how their algorithm exploits some of the symmetry between operators.

These works represent some of the ways in which states can be expanded in parallel to improve the performance of state space searches. They demonstrate the wide variety of design strategies that can be used to distribute the work of state expansion, focusing on equal work distribution either by ways of mapping states to processors evenly or by partitioning the information about the task among different agents. Next we look at another common design technique that splits the state space up among the search agents.

2.2.2 Partitioning the Search Space

An alternate idea to parallelizing heuristic search is to partition the state space among the processors, called tree decomposition. This differs from parallel state expansion in a fundamental way. In the algorithms mentioned above the processor works on states according to their hashed-value which means that they can process nodes in the state space that are not associated in any way, hopping from one state to the next in the graph. The following works present different techniques to partition parts of the state space for each processor to search.

One of the first attempts to parallelize heuristic search in this fashion can be attributed to Powley et. al. who implemented the first version of a parallel IDA* search called SIDA* (Single-Instruction Multiple-Data IDA*). The main idea of SIDA* is to distribute an initial collection of nodes among all the processors from a main processor. This led to a high level of overhead at the beginning of the search as the majority of initial states generated must be distributed to other processors to attempt to achieve initial load balancing[PFK93].

Once the initial distribution of nodes has completed the main search algorithm begins. In SIDA* each processor performs its own IDA* search in its sub-tree, to potentially different search depths. These processors may finish at different times, determined by the structure of the state space. The iteration of IDA* is performed by all processors, with an iteration being considered complete when all processors have searched their sub-tree, all leaf nodes have an f value greater than the cost bound[PFK93]. The bound of the next iteration is adjusted to the minimum cost leaf node of all nodes that exceed the current bound. If a goal state was found the search continues on all processors with a bound less than or equal to the solution cost to ensure the returned solution is optimal. Because the state space was split among processors based on the structure of the space a complicated load balancing scheme was used to ensure all processors were being used during the search. Their implementation used dynamic triggers to determine when additional balancing was needed. Their empirical results demonstrated a speed up of 42% with over 16,000 processors being utilized, with over 25% of the search time being dedicated to load balancing[PFK93].

Around the same time work was done on performing parallel IDA*[MD93]. Their work differed from Powley's as each sub-tree was searched to the same threshold depth and was heavily focused on maintaining load balance during search. They developed four different techniques for expanding and sharing nodes among all the processors. Their search algorithm, IDPS, was also run on a Connection-Machine 2 with over 16,000 processors and demonstrated significant efficiency gains and speed ups. Their work showed gains only believed to be possible on Multiple-Instruction Multiple-Data (MIMD) machines, an encouraging result that parallelization on SIMD machines can be a powerful technique if managed properly[MD93].

A more recent approach at partitioning the state space parallel search was done by [Mah11] called Parallel Multithreaded Iterative Deepening A* (PMIDA*). Similar to SIDA* the search algorithm begins with one processor expanding the root state. Instead of using f values to attempt load balancing before parallel search begins PMIDA* performs a depth-limited BFS, determined by the size of the problem and number of processors available. The state space is expanded sequentially until a collection of nodes are generated such that each processor receives one sub-root node per thread.

Once the parallel search begins each thread performs IDA* on its assigned sub-tree. Each thread performs an iteration using the same global threshold depth while maintaining their own open and closed lists. Having private open and closed lists does lead to the creation of duplicate states which can result in increasing memory overhead. Using private lists was chosen to reduce the overhead of concurrent access to shared global structures, minimizing synchronization overhead. When a thread completes an iteration of IDA* a global threshold checks if the local next threshold is better than the current value of the global next threshold, and all threads update to the minimum of all local thresholds before beginning the next iteration. Unlike the algorithms used in [MD93] and [PFK93] once a single goal state is reached all threads are signalled to stop, as [Mah11] proves that the PMIDA* algorithm returns an optimal solution. Their experimental analysis of PMIDA* showed an increase in efficiency from 31% to 74% on travelling salesman and vertex cover, increasing as the size of the problems increased; and up to 80% on hard 15-puzzle instances. Their work shows the potential gains that can be achieved by exploiting modern multithreading techniques and keeping communication between threads to a minimum.

Another modification of the sequential best-first search algorithm is called K-Best-First Search (KBFS)[FKK03]. The algorithm expands the top K nodes on the open list allowing the algorithm to consider several potential paths with the goal of minimizing the chances of being guided down an incorrect path for too long. This technique was extended into an algorithm called Adaptive K-Parallel best first search[VBH10]. Their implementation makes use of threads, which do not need to be the same as the number of processors available. Each thread acts as an independent agent expanding K states concurrently.

Each thread has access to global open and closed lists, with each thread performing KBFS in parallel. Every iteration a thread extracts the K best nodes from the open list to expand. Threads do not wait for each other to complete expanding all of their K nodes to minimize the time spent idle, the only synchronization points are when the threads access the open and closed lists. Since each thread is processing K nodes in parallel the main loop controlling the search must be modified so the algorithm terminates correctly. The parallel algorithm cannot terminate when the open list is empty as one or more threads may be expanding K nodes each and the search could continue from their successors. They used a global counter to indicate how many threads are currently expanding nodes. Once a goal state is found a global flag is used to indicate that all the threads should terminate the search. This termination condition does not guarantee an optimal solution is found and their implementation was designed for satisfying planning tasks, although they suggest methods to expand it to optimal planning. Their implementation was extended to include a restart strategy and the ability to dynamically change the number of threads during the search.

Their experimental evaluation demonstrated that implementing the algorithm with up to 64 threads lead to being able to solve more planning instances than sequential KBFS. Just as KBFS allows the search to consider many potential paths in one iteration their parallel implementation allows K threads to evaluate as many potential paths each, allowing their algorithm to find solutions on harder instances. They computed a *winning ratio* to evaluate how their algorithm scales and demonstrated that the advantages of multithreading come from harder problems. Combined with restart strategies and dynamic thread creation they were able to find solutions to even more problems and report super linear speedups of many tasks.

These works represent a wide variety of techniques for developing a parallel heuristic search algorithm. The diversity of all these algorithms demonstrate the challenges that are involved in parallel search and the unique ideas that have been presented to overcome them. Their works influenced the design of the parallel algorithms in this work, taking from their core design principles to help further improve parallel search.

2.3 Modelling Causal Dependencies

In order to use the causal graph heuristic there are two graph structures that must be constructed to be able to model dependencies. One captures the relationships between variables, indicating which variable depends on other variables. The other structure models how variables change values within their domains, and what conditions must be satisfied to make a transition. We briefly describe each of these structures and provide their definitions as described in the original causal graph work.

2.3.1 Causal Graph

Here we provide the definition used in [Hel04]. We assume that the problem is represented as a multivalued planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_*, s_g \rangle$.

Definition 2.1. Causal Graph
Given a variable set $\mathcal{V}$ of an MPT, the causal graph CG(Π) is a directed graph (V, A) with an arc between (v, v') if and only if $v \neq v'$ and one of the following is true:

- There exists an $o \,\epsilon\, \mathcal{O}$ with pre(v) and eff(v') defined.
- There exists an $o \,\epsilon\, \mathcal{O}$ with eff(v) and eff(v') defined.

The first case represents how the causal graph includes an arc between two state variables (v, v') if an operator changes the value of the target variable, v', and is dependent on the value of the source variable, v. The second case is necessary as it represents how there may be an operator that modifies the value of another variable as a co-effect[Hel06]. Without capturing this dependency the action sequence generated using a causal graph constructed using the first case only may generate action sequences that negatively effect the values of the co-dependent state variable, such as undoing a goal state already reached.

Returning to the example in this work we can see that the second operator will produce four arcs within the causal graph: $\{(r, a1)$, $(r, b1)$, $(b1, a1)$, $(a1, b1)\}$. A look over this arc set reveals that a cycle is now introduced in the causal graph between the arm of the robot and one of the balls. This intuitively makes sense: if we want to pick up the first ball from room A then the ball *must* be in room A, however this extra dependency can create some unwanted effects for the causal graph heuristic. See Figure 2.1(a) for the natural causal graph that is constructed for the gripper problem.

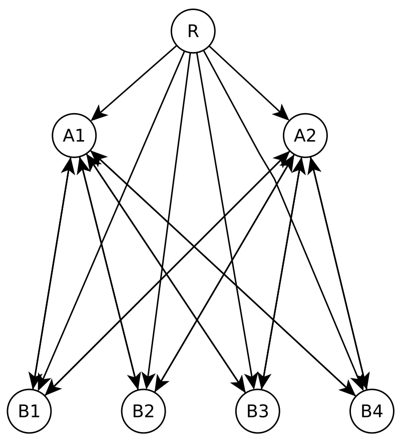
For the causal graph to be used for heuristic evaluation it is necessary for the causal graph to be acyclic so that an ordering can be imposed on the variables, creating a hierarchical structure[Hel06]. Ensuring that the causal graph is acyclic then allows us to decompose the problem into subtasks, focusing on a subset of variables in the causal graph to find independent plans before combining the results. As mentioned earlier, one of the shortcomings of other heuristics such as h^+ is that too much information may be lost with respect to the planning task if all preconditions (or delete effects) are ignored; therefore any cyclic causal graph should have arcs removed in a systematic way to preserve as much information as possible.

Helmert outlined an algorithm that greedily removes arcs within the causal graph based on how *important* the causal relationship is[Hel06]. The idea of the algorithm as as follows. Strongly connected components are identified within the causal graph. Within the connected component each arc is assigned a weight equal to the number of operators that induce the arc. For each variable within the component we take the cumulative weight of all the incoming arcs and remove the node v with the lowest weight. This variable is set to the lowest level, such that $v \prec v'$, and all arcs from v to v' are removed from the causal graph. This is repeated until one vertex remains in the strongly connected component. Using this strategy the causal graph of the gripper problem can be pruned to result in the graph in Figure 2.1(b).

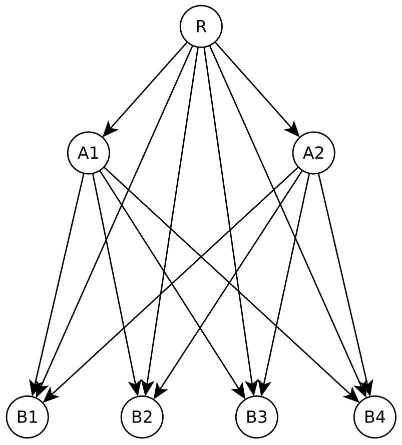
The hierarchical relationship between all the variables becomes much more apparent once the causal graph is acyclic. Within the causal graph we call a variable a low-level variable if the variable is able to change its value regardless of the values of the other variables; it has no inward arcs in the causal graph. Variables that are dependent on the values of their ancestors in the causal graph are called high-level variables[Hel04]. In the context of the gripper example the robot body is the lowest level variable, while the balls are the high level variables.

2.3.2 Domain Transition Graphs

The second causal structure that is needed for the causal graph heuristic are domain transition graphs (DTG). Unlike the causal graph which models relationships between variables, a domain transition graph is constructed for each variable in the planning task. This graph represents how each variable is able to change its values within its domain. We will again follow Helmert's



(a) The natural causal graph.



(b) The pruned causal graph.

Figure 2.1: Gripper Task Causal Graph

definition of the domain transition graph [Hel04].

Definition 2.2. Domain Transition Graph
Let $v \epsilon V$ of a MPT $\Pi = \langle \mathcal{V}, \mathcal{O}, s_*, s_g \rangle$, the domain transition graph G_v is a directed graph with the vertex set $\mathcal{D}_v$ an arc between nodes if one of the following is satisfied:

- If there is an $o \epsilon \mathcal{O}$ with $pre(v) = d$ and $eff(v) = d'$ then we add an arc (d, d')
- If there is an $o \epsilon \mathcal{O}$ with $pre(v)$ undefined, and $eff(v) = d'$, then we add arcs (d, d'), $\forall d \epsilon \mathcal{D}_v \backslash \{d'\}$

Arcs in the domain transition graph may have labels that indicate preconditions that must be satisfied for the transition to occur. More formally:

- For each arc there is a label L that contains pre$|(\mathcal{V} \backslash \{v\})$, and say that there is a transition for v from d to d' under the condition L.

It is possible for there to be multiple arcs from (d, d') as there may be different operators that cause the transition under different conditions. See Figure 2.2 for examples of the domain transition graphs for the three different variable types. The transition graph for each ball appears in the top left. We can see there are two arcs from each node to the others in the graph. This is because of the two arms that the robot can use; it can either cause the transitions using the left arm or the right arm, so the transition graph models the two potential ways that the state of the ball may change.

The labels on each arc are induced by the hierarchy of the variables, only ancestors of a variable v in the causal graph may appear in labels on the arcs of the domain transition graph for v. If there is a variable in the causal graph with no inward arcs then the arcs within its domain transition graph will have no label. These low level variables can change values in its domain without any constraints imposed by the current assignment of the other variables. In Figure 2.2 the bottom left graph represents the robot body, which in the causal graph has no inward arcs in the pruned causal graph, so there are no constraints on the transitions for the body. The balls, which are the highest level variables in the planning problem, are constrained by each of the variables in some way.

As states are defined by the current value each variable is assigned a plan and can be viewed as the traversal of each domain transition graph simultaneously[DD01]. Therefore, the assigned value of a variable corresponds to a particular node in that variables domain transition graph, which is called the *active vertex*[Hel04]. An operator can be used to change the value of a variable to a new active vertex. This is allowed if and only if the active vertex for each variable on the label of the transition is equal to the value in the condition. For example, the transition from *at-room-b* to *carried* for any of the balls requires that the robot body is *at-room-b* and either arm one or two is *free*.

2.4 Causal Graph Heuristic

Now that the structures have been created and cycles have been removed from the causal graph we can use these graphs to generate heuristic estimates for the cost of the optimal solution. When the causal graph heuristic, $h_{cg}(s)$, was first proposed there were two conditions on the structure of the causal graph; it needs to be acyclic and can be decomposed into more restrictive subtasks known as **SAS$^+$-1** tasks. A SAS$^+$-1 task has a particular variable v such that there is an arc from every other variable to v and no other arcs,

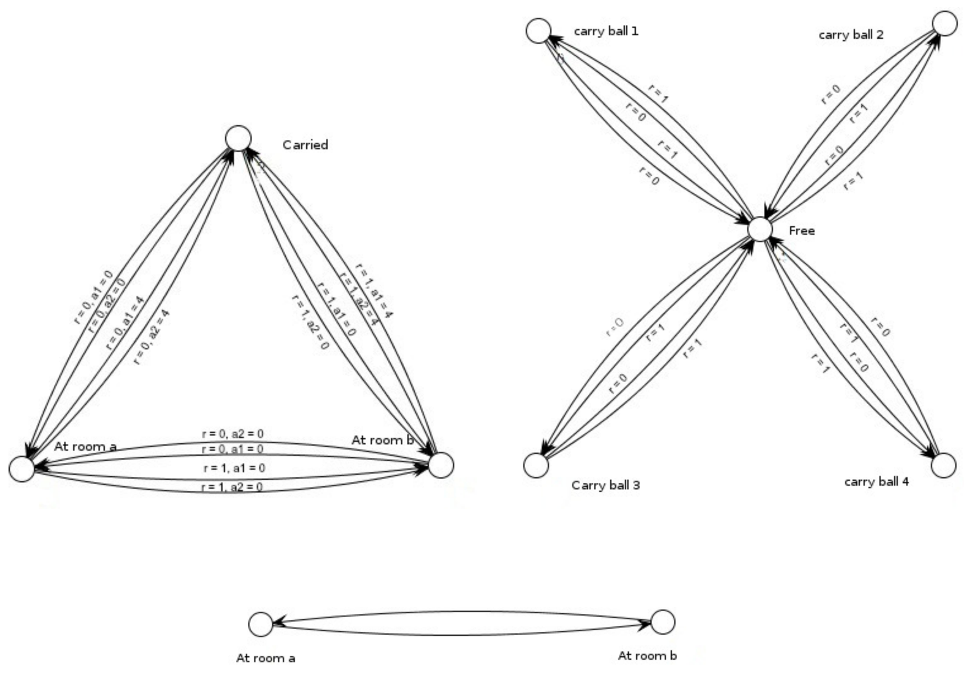


Figure 2.2: DTG's for a ball (left), an arm (right), and the robot (bottom).

and a goal must be defined for v[Hel04].

The reason for focusing on this restricted set of problems is that an algorithm can solve each SAS$^+$-1 task, with each solution being valid. Another benefit is that while general SAS$^+$ planning is **PSpace**-complete, determining plan existence of SAS$^+$-1 tasks is in **NP**-complete[Hel04].

The general algorithm first presented in [Hel04] for attempting to find a solution to SAS$^+$-1 tasks is a modified version of Dijkstra's search for single-source shortest path problems. For each variable v in which $s_*(v) = d'$ is defined we identify its predecessors in the causal graph. Let π denote the plan found so far with $\pi = \langle\rangle$ to begin. We then consider all the possible high level transitions for v given its current value d and the values each of its predecessors is currently assigned. For each outward edge from d in the DTG of v we check to see if there is a label and if the preconditions on the label can be solved for. If they cannot then we ignore that transition and move on, however if these preconditions can be met then we determine π_L, the minimal plan to satisfy the preconditions. We can then update the plan to be $\pi = \pi + \pi_L$. When calculating $h_{cg}(s)$ all high level transitions are assumed to have a cost of 1, so the heuristic estimate is the magnitude of the plan returned by the above algorithm, $|\pi|$. This process is continuously repeated until there is no transition within the DTG of v that yields a shorter plan length to d', or if no path can be found then the algorithm returns a failure[Hel04].

An important structural feature of the low level DTG is strong connectedness. If the DTG for every low level variable in the task is strongly connected then it is always possible to satisfy the preconditions of all transitions within the DTG of the high level variable[Hel04]. In practical planning tasks where the causal graph can be decomposed into multiple SAS$^+$-1 subtasks the heuristic estimate for $h_{cg}(s)$ is the sum of the minimum plan cost for each subtask, as formally defined in Equation (2.2) [Hel04] :

$$h_{cg}(s) = \sum_{v \in s_*} cost_v(s(v), s_*(v)) \tag{2.2}$$

While this algorithm provides a method for solving SAS$^+$-1 these structures are quite rare in practice. Consider the pruned causal graph in Figure 2.1(b), the induced subtasks for each goal variable (nodes 0-3) do not appear to exhibit the SAS$^+$-1 structure in their induced subgraph. Since the heuristic is based on this induced graph structure it does not consider the effects of variables that are more than one layer away in the causal graph, it only considers the immediate ancestors of the variable.

When we compute $h_{cg}(s)$ for each goal variable we can see the DTG for each transition is dependent on the arm variable and body variable, while the arm is also dependent on the state of the body and this structure does not conform to the SAS$^+$-1 definition. We also notice that the induced subgraph of the arm and body do form an SAS$^+$-1 task and can be solved for. We compute the minimum cost of solving the task to satisfy the label precondition for the arm, then separately compute the cost for satisfying the precondition of the body, and combine these results to generate the heuristic estimate for $s(v)$. The formal algorithm outlining the original causal graph heuristic can be found in [Hel06].

To illustrate this in practice consider the gripper task of moving one ball from room A to B. Initially the robot body is in room A, both arms are free, and all balls are in room A; these correspond to the active vertices for each DTG. Now, consider moving ball 1 from room A to room B. Beginning with the highest level variable in the causal graph we examine the node with the minimum cost, specifically the active vertex $d =$ *at-room-a*, with $cost(d, d) = 0$. All other costs $cost(d, d') = \infty$, $\forall d' \epsilon \mathcal{D}_v \backslash \{d\}$.

There are four arcs out of the source node, two to *carried* and two more to *at-room-b*, and each transition has two preconditions. Let us consider the transition from *at-room-a* to *at-room-b*. The algorithm is called recursively to then solve satisfying the preconditions. Without loss of generality we examine the arc with the precondition that $r =$ *at-room-b* and either arm $a1 =$ *carry-ball-1* (computing either arc is the same process). To satisfy the precondition on the arm a transition must occur in the DTG of that arm from *free* to *carry-ball-1*, and the arcs are labelled with a precondition on the robot body. There are two arcs from *free* to *carry-ball-1* with two different labels, one where the robot is at *at-room-a* and the other when is it *at-room-b*. Computing the transition cost under the first label results in a cost 0, as we are already in room A, while the other arc has a label cost of 1 as the body must move to room B. Therefore, the overall cost of this transition is 1, the sum of the label cost plus the transition cost. The next precondition to satisfy is that the body is in room B. When we look at the DTG for the body there are no labels and we can freely change from room A to room B with a cost of 1 for the transition. Overall, we see that satisfying each precondition on the label for this arc has a cost of 2 and the final step is to increase this cost by the transition cost of one. Therefore, $cost_v(s(v), s_*(v)) = 1 + 1 + 1 = 3$ which is the optimal number of steps to solve this specific subtask. The process is the same for each goal variable in this task, the heuristic evaluation result is:

$$\begin{aligned} h_{cg}(s) &= \sum_{v \in s_*} cost_v(s(v), s_*(v)) \\ &= 3 + 3 + 3 + 3 = 12 \end{aligned}$$

2.4.1 Causal Graph Heuristic Limitations

The causal graph heuristic provides a powerful technique for finding plans for tasks, however the additive nature of the heuristic renders the heuristic inadmissible in most planning domains. For any planning task, if there is a dependency between goal variables or variables in the label of a transition condition then the heuristic estimate is inadmissible as the positive effects between variables may lead to an overestimate[GLY$^+$08]. The heuristic only returns an optimal estimate to solve the task if there is only one variable in the goal and every operator has at most one condition [Hel04], which is why if a SAS$^+$-1 has a solution the heuristic returns an optimal estimate.

In the running example it is clear that the goal state is defined for more than one state variable and operators are not unary, and indeed the heuristic estimate of 12 is greater than the optimal solution of the task. When we look how the algorithm finds these relaxed plans it computes each subtask individually and sums the result together and each goal variable is unaware of any other goal variables. Human reasoning realizes that the robot has two arms and could move two balls simultaneously, leading to an optimal plan in 11 steps. The heuristic is not able to capture any positive interactions between goal variables as there is no causal dependency between them and because of this the heuristic assumes that goal variables must be achieved individually.

The issue with non-unary operators is that we are recursively solving each condition on the transition, and these are summed into the result of the high-level variables' cost. Computing the heuristic estimate for satisfying these preconditions may have the same problem as the goal variable so the heuristic is unable to capture positive interactions between state variables as each subtask is solved.

An additional issue is the overall complexity to compute the heuristic. The original causal graph heuristic was proposed for planning tasks with acyclic causal graphs which were believed to provide simpler domains to work on. In the original work however, Helmert demonstrates that plan existence and generation for a simple SAS$^+$-1 task is **NP**-complete and

[JJL13] demonstrated that extending the technique to planning tasks with acyclic causal graphs determining plan existence is still **PSPACE**-complete. Further, since determining the optimal plan for a task is at least as difficult as determining plan existence then optimal planning for tasks with acyclic causal graphs is also **PSPACE**-complete[JJL13].

In the work done by Geffner and Helmert in [Gef07, HFG08] they were able to demonstrate that the causal graph heuristic can be reduced to the additive heuristic when the causal graph of a planning task is acyclic. The major difference between the causal graph heuristic and the additive heuristic is that the additive heuristic does not keep track of the side effects of operators; instead it evaluates the cost of achieving each of the conditions on the operator with the same state[Gef07]. They developed a heuristic that combines the additive and causal graph heuristic which they call the *Context Enhanced Additive Heuristic* and is denoted $h^{cea}(s)$[HFG08].

The underlying idea behind this heuristic is to choose different states s_i, called contexts, with which to evaluate the heuristic function for each goal variable. For an operator of the form $\langle pre(v) = d,\ \mathit{eff}(v) = d' \rangle$ the state is updated to reflect the variable achieving $v = d'$. All other conditions are evaluated from the resulting state[HFG08]. This heuristic has several qualities that tie it closely to the causal graph heuristic and the additive heuristic. In the case of a planning task where the causal graph is acyclic they demonstrate $h^{cg}(s) = h^{cea}(s)$ for all states. They further demonstrate that if the planning task is composed of only boolean variables then $h^{cea}(s) = h^{add}(s)$ for all states[Gef07]. An additional benifit that comes from the context enhanced heuristic is that it does not require that the causal graph is pruned into an acyclic graph, and ties together two different areas of heuristic work for planning tasks.

A different attempt to overcome the additive nature of the causal graph heuristic led to the creation of the *causal graph mixed heuristic*[GLY$^+$08]. This heuristic is a combination of the causal graph heuristic and the causal graph max heuristic, where the max causal graph heuristic is the maximum cost of the causal graph heuristic for each variable. They created a linear combination of these two heuristics resulting in the heuristic shown in Equation 2.3, where $0 \leq \alpha \leq 1$ and $0 \leq \beta \leq 1$[GLY$^+$08] :

$$h^{am}_{cg}(s) = \alpha \sum_{v \,\epsilon\, s_*} cost(s(v), s_*(v)) + \beta[max_{v \,\epsilon s_*} cost(s(v), s_*(v))] \tag{2.3}$$

This heuristic is equivalent to the causal graph heuristic if $\alpha = 1$ and $\beta = 0$, or is the max causal graph heuristic if the coefficients are reversed.

These coefficients allow for the heuristic to be fine-tuned to generate as close to an optimal estimate as possible, however selecting these values can be difficult and typically requires some experimentation to find the best combination[GLY$^+$08]. Overall their work demonstrates that using this mixed heuristic results in shorter search times and can find optimal solutions in situations where neither the causal graph heuristic or maximum causal graph heuristic can.

Chapter 3

Parallel Implementations

The algorithms implemented for this work perform a cost bound depth first search combining a heuristic estimate and path cost to determine if a branch should be pruned. The underlying principle is to allow one processor to proceed through the state space much like A* or IDA*, however newly generated states are handed off to secondary threads to be evaluated. We believe such a strategy will provide a more efficient way to search the state space when the heuristic involved is computationally intensive. It has been reported that parallel heuristic evaluation was implemented in chess systems such as **Deep-Blue** [FH90] and **HITECH** [Ebe87][PFK93]. These systems are highly domain specific, and literature could not be found applying this technique with domain independent heuristics for planning problems.

The motivation for applying these techniques to planning problems is that computing the heuristic value for states is a non-trivial task, however most of the states evaluated do not belong to the solution path. Because of this in a sequential, or parallel search, the algorithm requires additional time to evaluate these states. A common strategy used by planners is *deferred heuristic evaluation* [Hel06], where states are stored in the frontier with its' parents heuristic estimate. A state does not have its' h value computed until it has been selected for expansion. We believe our extensions take the concept of deferred evaluation a step further by having the main search algorithm ignore the need to compute heuristic values. States are selected for expansion only when it has been evaluated by one of the supporting threads.

A common theme within the literature of parallel heuristic search is the desire to maintain a level of simplicity in the parallel algorithms. The reasoning for this is primarily simplicity as debugging parallel algorithms with non-deterministic behaviour is extremely difficult[KFB13], and analyzing the algorithms behaviour is much more simple when the parallel algorithm is only a minor extension to the sequential algorithm[VBH10].

With these desired qualities in mind the implementation of these algorithms use the OpenMP 3.0 (OpenMP 2008) API for C++. Using this collection of interfaces allows for converting from sequential to parallel code relatively easily, and if the program is compiled in an environment without OpenMP the compiler can skip the OpenMP directives and attempts to run the code sequentially.

3.1 A* With Parallel Heuristic Evaluation

The main entry point for this algorithm is the *Thread Launcher* function (Algorithm 3). Before the parallel search begins two shared booleans are created which are used as flags to indicate when threads should terminate. Next the global structures are created: a priority queue for states which the main thread will use when selecting the next node to expand, and a stack for the successor states where the main thread places all the children after they have been generated. In the OpenMP framework these data structures will be shared globally between all threads. The final step is to compute the heuristic value of the initial state and to place the state in the open list.

After these structures have been initialized the first OpenMP directive is encountered; the *pragma parallel* directive defines a parallel block where the code inside is run by the number of processors that have been specified. Each thread can be uniquely identified by an integer $[0, K-1]$ where K is the number of processors being used. In this block a particular thread is identified (the one selected does not matter, we selected thread 0 for simplicity) and this thread will spend the remainder of it's lifetime within the *Main Thread DFS* function (Algorithm 4). All other threads create a private instance of the heuristic function and enter their own instance of the *Compute Heuristic* function (Algorithm 5), and the search algorithm begins.

When the main thread enters the search algorithm it sets the first bound to the initial heuristic estimate and then enters two loops for the duration of the search. The outer loop is used to coordinate the search among all the threads; there is no way to break out unless the search finds a solution or one of the threads has indicated to terminate the search. The inner loop functions similarly to A* with a few exceptions.

The first significant difference comes with determining when an iteration terminates. As we are exchanging states between the main *search thread* and *computation threads* we cannot guarantee an iteration is complete when the open list is empty. To ensure an iteration completes correctly we must ensure three things: the open list is empty, the processing list is empty, and none of the computation threads are working on a state. Only once all of these conditions are true can we proceed to the next depth. The use of a bound for this extension of A* is to minimize the plan length that is found. Without

Algorithm 3: Thread Launcher for ID with a Queue/Stack

```
    Input: An initial state, goal state, and applicable operators
    Output: A valid plan or a failure
 1  $solved \leftarrow false$; $killed \leftarrow false$;
 2  priority_queue⟨$State$⟩ $open\_list$;
 3  stack⟨$State$⟩ $process\_list$;
 4  $h \leftarrow compute\_heuristic(initial\_state)$;
 5  $open\_list.push(initial\_state)$;
 6  #Pragma Parallel:
 7  if ($thread\_num == 0$) then
 8      plan ← Main Thread DFS($solved, killed$);
 9  else
10      Compute Heuristics($solved, killed, heuristic$);
11  end
12  #End Parallel
13  return plan;
```

the use of a bound to iteratively expand the state space the algorithm is able to find plans of lengths 10-100 times longer than an optimal solution. While this is beneficial to determine plan existence efficiently, and satisfies the goal of satisfying planning tasks, we wanted to ensure that our algorithm returns solutions as close to optimality as possible, especially if it is to be combined with an admissible heuristic.

As the search algorithm proceeds the main thread removes the current best state from the top of the open list. As all the threads are able to access the open list a lock must be used before a thread attempts to insert or remove a state from the priority queue. If the thread is unable to remove a state from the open list because the list is empty or locked by another thread the algorithm will wait until it is able to remove a node from the open list or the other threads signal to stop the search. If a state is safely removed from the queue we check if the total cost of the state $f \leq bound$. If the state is within the allowed search depth we check if it is a goal state, in which case we trace the path back through the state space to capture the sequence of operators that lead to the goal. If the state is within the search depth but is not the goal state we generate the set of successors of the current state. Once we have a collection of these children we place a lock on the *process list* and insert the children on the top of the stack to be handled by the computation threads. Using a stack for this process helps maintain

Algorithm 4: MAIN THREAD DFS

```
Input: An initial state and booleans to act as flags for other threads
Output: A valid plan or a failure
1  $min \leftarrow \infty$;
2  $bound \leftarrow h$;
3  while $\neg solved$ and $\neg killed$ do
4      while $\neg open\_list.empty()$ or $(ThreadsBusy)$ do
5          State $current$;
6          Boolean $safe \leftarrow false$;
7          #Lock open_list
8          if $(\neg open\_list.empty())$ then
9              $current \leftarrow open\_list.top()$;
10             $safe \leftarrow true$;
11         end
12         if $(\neg safe)$ then
13             $continue$;
14         end
           // Goal and Bound checks before expansion
           // Expand state:
15         for each Operator $op \in \mathcal{O}$ do
16             State $child \leftarrow$generate_successor$(op, current)$;
17             #Lock process_list:
18             $process\_list.push(child)$;
19         end
20     end
21     $bound \leftarrow min$;
22 end
23 return FAILURE;
```

an ordering of nodes expanded similar to A*. By pushing the successors of the current best state on the top of the stack the computation threads will compute their heuristic values first and place them in the open list.

Algorithm 5: COMPUTE HEURISTIC

```
Input: Boolean flags indicating if the main thread has terminated.
Output: Null
while $\neg solved$ and $\neg killed$ do
    State $current$;
    Boolean $safe \leftarrow false$;
    #Lock process_list
    if ($\neg process\_list.empty()$) then
        $current \leftarrow process\_list.top()$;
        $safe \leftarrow true$;
    end
    if ($\neg safe$) then
        $continue$;
    end
    $h \leftarrow$ compute_heuristic($current$);
    $current.set\_h(h)$;
    #Lock open_list
    $open\_list.push(current)$;
end
```

The more computationally intensive part of the search is handled by the remaining threads running the *Compute Heuristic* algorithm. Similar to the main search method each thread is contained within a while-loop that will not end until the task has been solved or a thread indicates that the search should be terminated. As all threads are able to access the stack for processing nodes the stack must be locked before a thread tries to remove a state. If the stack is empty or locked the thread will wait until it is able to remove a state or is told to terminate. Once a state has been removed from the stack a thread computes the total cost for the state. Once a thread has computed the heuristic estimate of a state it locks the open list and inserts the state into the open list to be expanded by the primary thread.

The way that the algorithm is structured the primary thread is able to proceed through the state space in a best-first manner. As the secondary threads process children in a last in, first out, manner the children of the previous best state are evaluated first. If combined with a consistent heuristic all children would have an f value less than or equal to its' parent, and some would be the next states selected to expand once they are placed into the open list.

3.1.1 Algorithmic Correctness

The correctness of the algorithm follows from the sequential algorithm which it is based on. Since the search is only performed by one processor if an admissible heuristic is used the algorithm is guaranteed to return an optimal solution as A*. If the heuristic does not overestimate the cost of the optimal solution the bound on any iteration is no more than the cost of the optimal solution so the first goal state achieved will be optimal.

While there is a non-deterministic aspect to the algorithm in terms of how states are inserted and removed from the shared structures this only effects the order in which nodes are evaluated. With an admissible heuristic the search depth will not exceed the length of the optimal solution and for this reason the main thread will correctly terminate when a path to the goal is found.

The space complexity is equivalent to that of A*. The two shared structures store the equivalent number of nodes as the open list in the traditional A* search, and a closed list is also used to minimize the number of re-expanded states. Therefore the combination of these lists will contain at most $\mathcal{O}(bd)$ different states. The overall time complexity is also equivalent to A* in the worst case. When the optimal solution is a depth d away and an admissible heuristic is used in the worst case the goal node is the last state explored during an iteration with d as the bound so overall the amount of time taken to explore the state space is $\mathcal{O}\left(b^d\right)$ in the worst case.

3.2 IDA* With Parallel Heuristic Evaluation

An alternate approach to parallel heuristic evaluation we implemented is based on the IDA* algorithm. Our implementation starts in a similar manner as the algorithm described in the last section with the difference of using a stack for the open list instead of a priority queue. Without the OpenMP directives this algorithm could run as a sequential stack based iterative deepening algorithm using the same technique mentioned earlier. The main thread still remains as the sole thread performing a search through the state space using Algorithm 4. The remaining threads are sent to the Compute Heuristic function, Algorithm 5. Once a state has been expanded and the successors have been generated each child is then pushed onto the stack of states to process.

Algorithm 6: Thread Launcher for ID with Stacks

```
   Input: An initial state, goal state, and applicable operators
   Output: A valid plan or a failure
 1 $solved \leftarrow false$; $killed \leftarrow false$;
 2 stack$\langle State \rangle$ $open\_list$;
 3 stack$\langle State \rangle$ $process\_list$;
 4 $h \leftarrow calculate\_h\_val(initial\_state)$;
 5 $open\_list.push(initial\_state)$;
 6 #Pragma Parallel:
 7 if ($thread\_num == 0$) then
 8     plan $\leftarrow$ Main Thread DFS($solved, killed$);
 9 end
10 else
11     Compute Heuristics($solved, killed, heuristic$);
12 end
13 #End Parallel
14 return plan;
```

With this technique the search algorithm is able to proceed in a style similar to a traditional depth-first search through a state space. Starting from the initial state the secondary threads begin to process the initial set of successors and places them on the stack to search next. When the successors of this node are generated they are placed on the stack, possibly on top of states generated as the successor of the initial state. The main processor and spawned threads alternate placing and removing nodes on their respective stacks. As planning tasks typically have large branching factors there will be more successors generated than spawned processors available. This quality allows the search to proceed in a pseudo-depth first manner.

To illustrate this consider how the algorithm will behave when beginning on a typical planning task and see Figure 3.1 for reference. The initial state is expanded and the collection of children are placed onto the processing stack. At the same time, when not locked, threads remove a state from the top of the stack to evaluate and insert on the open stack in a possibly different order, the red nodes. The main thread then removes one of these states to expand placing its successors, the green nodes, on top of the processing stack. The spawned threads could potentially still be computing the

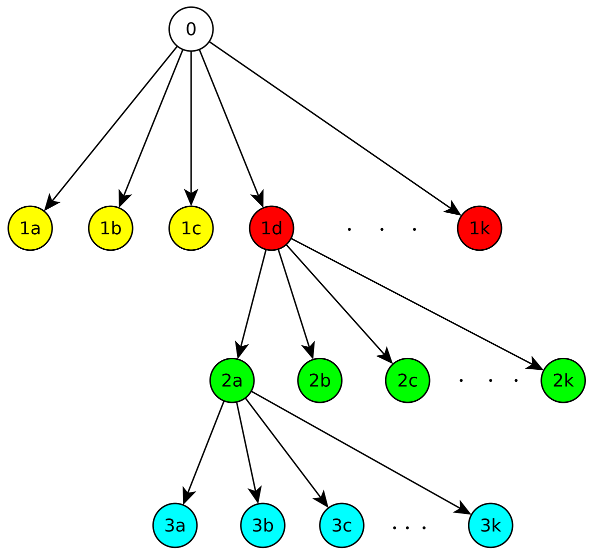


Figure 3.1: Example of Stack-based Parallel Evaluation.

heuristic values of the initial successors, yellow nodes. These new successors (green) then get placed on top of the process stack, while the main thread removes the top node from the open list which is typically an initial successor, yellow or red node, early into the search. While this expansion occurs the other threads place the green set of successors on the open list, while the main thread places the blue successors on the processing list. This trend continues through out the search, the algorithm searches the state space in a manner that combines breadth and depth first search, similar to a beam search [RN10] of game-trees without being as well guided.

To see how this algorithm is equivalent to the sequential IDA* algorithm only a minor modification needs to be made. An additional step could be added to the search function; if the *open list* is empty check the *process list*, if it is not empty determine the states heuristic value and expand the state. These details are omitted for simplicity in the pseudo-code and were not implemented for our empirical analysis.

3.2.1 Algorithmic Correctness

The correctness of the algorithm follows from the sequential IDA* algorithm which it is based on. Since the search is only performed by one processor if an admissible heuristic is used the algorithm is guaranteed to return an optimal solution[Kor85].

There is a non-deterministic aspect to the algorithm with how states are inserted and removed from the shared stacks. However, with an admissible heuristic the search depth will not exceed the depth of the optimal solution so the main thread will correctly terminate when a path to the goal is found.

The space complexity is equivalent to that of A*. The two shared structures store the equivalent number of nodes as the open list in the traditional A* search, and a closed list is also used to minimize the number of re-expanded states. Therefore the combination of these lists will contain at most $\mathcal{O}(bd)$ different states. The overall time complexity is also equivalent to A* in the worst case. When the optimal solution is a depth d away and an admissible heuristic is used in the worst case the goal node is the last state explored during an iteration with d as the bound so overall the amount of time taken to explore the state space is $\mathcal{O}\left(b^d\right)$ in the worst case.

Chapter 4

Constrained Causal Graph Heuristic

The heuristic implemented for this planner extends the causal graph heuristic. We reformulate the causal graph heuristic as a constrained shortest path problem. This problem is typically solved using dynamic programming where a table is maintained for all vertices and constraints. In the context of the causal graph heuristic we create a table *distances* where the rows correspond to values $d \,\epsilon\, \mathcal{D}_v$ in a variables domain, and columns represent the transitions t. When we update the entry $distances[d][t]$ we have found the current shortest path to the value d under the constraint that the preconditions on the label of t out of d have been satisfied.

4.1 Constrained CGH with Actions

Our proposed extension to the causal graph heuristic takes the additive form, summing the cost of satisfying each goal variable as shown in Equation 4.1. An important difference between the causal graph heuristic and our formulation is the state used to compute the plan, $s_h(v)$. This state, called the heuristic state, changes as we solve each sub-task of the overall planning task. When we begin the heuristic evaluation of a state the heuristic state is the current world state: $s_h(v) \leftarrow s(v),\ \forall\, v \,\epsilon \mathcal{V}$. As we compute the plans for each goal variable we maintain a record of the modifications made to the state. Once the plan has been found for v we update the state $s_h(v) = s_*(v)$. We also update $s_h(v') = d^*_{v'}$, where v' is an ancestor of v in the causal graph, and $d^*_{v'}$ is the assignment of v' after taking the transition that got the variable v to take the value $s_*(v)$.

$$h_{ccga}(s) = \sum_{v \,\epsilon\, s_*} cost(s_h(v), s_*(v)) \tag{4.1}$$

To the world state as we traverse the domain transition graph of a variable a local state is used s_l to capture the transition effects. When computing the plan for a variable v we maintain a copy of the local state. Initially $s_l \leftarrow s_h$ to store any of the side effects from solving the previous subtasks. The motivation for this is to capture as much information about the planning task as possible while computing the heuristic value. Computing the causal graph heuristic requires solving subtasks in the planning problem, and in many domains the changes to variables from solving a subtask can have a positive effect on the subsequent subtask.

The other significant difference between the original causal graph heuristic and our proposed extension is the use of the full operator. When domain transition graphs are constructed some of the effects of an operator may be ignored if the effected variable is not a causal ancestor. Our heuristic ignores these effects when constructing domain transition graphs, however when we follow a transition in the DTG we apply the full effects of the operator to the local state. Using this strategy our relaxed plans are able to more closely represent the overall planning task by allowing transitions in one subtask to effect the variables in another subtask.

4.1.1 Constrained Shortest Path Formulation

The technique of solving the constrained shortest path problem traditionally uses dynamic programming techniques. In the context of traversing a domain transition graph the function $cost(s_h(v), s_*(v))$ can be defined using a recurrence relation. Let $s_h(v) = d$, $s_*(v) = d^*$, $\mathcal{C}$ represent the cost of a transition t, and $N(d)$ represent the set of vertices that have transitions that lead to d in the domain transition graph. The optimal cost to achieve the goal value d^* is computed by determining the domain transition value that leads to d^* that has the lowest cost of being reached under the constraint of having the preconditions on the transition already being satisfied. We use that value and add the action cost as the heuristic estimate for $s_*(v)$. The recurrence relation can be defined as follows.

$$OPT(d^*) = min_{d' \epsilon N(d^*)} \left\{ distance[d'][t] + \mathcal{C},\ \infty \right\} \tag{4.2}$$

In the case where $d = d^*$ the result is 0, as no transitions must be taken because the planning task has already satisfied the goal value. If no path to the goal value can be found then the algorithm returns infinity to represent this. To capture the state of the world as we traverse the domain transition graph a local copy of the state, $s_l(v)$, is used.

To find the optimal path cost to d^* the path to each value d' along the path to d^* must be computed. The subtask of determining the optimal cost of reaching the value d', for every transition t' out of d', and satisfying the preconditions on the label L on t' is defined by the following recurrence:

$$OPT(\langle d', t' \rangle) = min_{d \epsilon N(d')} \left\{ distance[d][t] + \mathcal{C} + \sum_{v' \, \epsilon \, L} cost(s_l(v'), pre(v')), \infty \right\} \tag{4.3}$$

When the algorithm begins we populate the table row of the value currently assigned for the current goal value being solved for, $s_h(v) = d$. We compute all entries in *distances*$[d]$ by solving the cost of achieving the preconditions for each variable on the label L of every transition t out of d using the following recursive sum:

$$distances[d][t] = \sum_{v' \, \epsilon \, L} cost(s_l(v'), pre(v')) \tag{4.4}$$

4.1.2 Algorithmic Description

When the heuristic function is called the algorithm attempts to find a plan of reaching the goal value for each goal variable. The algorithm begins by populating a priority queue using Algorithm 7. We examine the current value $s_h(v) = d$ of the variable v being solved for in the subtask. For every transition out of d in the domain transition graph of v we recursively solve the labels on the transition and insert a node into the priority queue that stores the source value d, the transition t, and the cost of being at d with the labels of t being satisfied. Additionally the local state is stored to represent the world state after the preconditions have been satisfied.

Once the priority queue has been populated with all pairings $\langle d, t \rangle$ we perform a modified version of Dijkstra's Single-Source Shortest-Path algorithm, Algorithm 8. In each iteration we remove the best vertex from the priority queue. This vertex represents the current shortest path to a value within the domain transition graph with all the preconditions on the label of the transition being satisfied. We further extend the shortest path search by including an additional step where we apply the complete effects of the operator the transition represents. Performing the evaluation in this manner allows us to estimate the cost by attempting to capture positive effects between variables, similar to the context enhanced causal graph heuristic proposed in [HFG08].

When we remove a vertex $\langle d, t \rangle$ from the priority queue during the search we look up the current best cost in the *distances* table. If the vertices cost is greater than the current best value then we ignore the vertex, otherwise

Algorithm 7: PREPROCESSING

```
Input: Current Variable $v$, Current Value $d$, Current State
       $assignment$
Output: A populated priority queue to begin the search
1  $label\_cost \leftarrow 0$;
2  for each $transition\ t\ out\ of\ d$ do
3      State local_assignment ← assignment;
4      for each precondition $pre(v') = u\ on\ label\ of\ t$ do
5          $label\_cost \leftarrow label\_cost +$ CCGHA$(v', u, assignment)$;
6          local_assignment[v'] ← u;
7      end
8      $distance$[d][t] = $label\_cost$;
9      Vertex v$(d, label\_cost, transition\_id, local\_assignment)$;
10     open_list.push($v$);
11 end
12 return $open\_list$;
```

we apply the full effects of the operator with $s_l(v') = \text{eff}(v')$ for all variables defined in the effects of the operator. If the destination value d' of the transition is not the goal value then we examine every transition t' out of d' and recursively solve the conditions on the label of t' and insert the new vertex $\langle d', t' \rangle$ into the priority queue with the cost using the equation:

$$\text{distances}[d'][t'] = \text{distances}[d][t] + \mathcal{C} + \sum_{v' \,\epsilon\, L'} cost(s_l(v'), pre(v')) \tag{4.5}$$

If the destination of the transition is the goal value d^* being searched for, or if the destination has no outward transitions, we can ignore the sum in the above equation. This situation is indicated by setting the transition ID to -1. We push the pair $\langle d', -1 \rangle$ into the queue with the path cost as computed using the simplified equation 4.6.

$$pathcost(d', -1) = pathcost(d, t) + \mathcal{C} \tag{4.6}$$

When we remove a node from the queue that is not the goal value, and the transition ID is -1 we record the cost and continue the search. Otherwise, we've reached the goal value for v and can update the heuristic state $s_h \leftarrow s_l$ before returning the path cost to reach d^*. When the next sub-task is solved the heuristic state used is modified to reflect the changes made when solving the earlier sub-tasks. If there is no such path to d^* then we return ∞ to indicate that a solution could not be found from that state.

To illustrate how positive interactions are captured under this formulation consider an operator that effects multiple variables, the high level variable v and one or more of its ancestors in the causal graph v'. Without loss of generality, before the transition could be used in the domain transition graph a search through the domain transition graph of v' is needed to get $s_l(v) = \text{pre}(v')$. As a result of following the transition in the domain transition graph of v the value of v changes from $s_l(v) = d$ to $s_l(v) = d'$. Because of the pruning strategy used to create the causal structures other effects of the operator, $s_l(v') = u = \text{eff}(v')$, are not captured in the updated local state. If we have not reached the goal value for v then we continue the search out of d' in the graph where at least one of the transitions includes v' in its' label. If the label has the value $\text{pre}(v') = u$, the value achieved by the previous action, the positive interaction between variables is ignored in the original causal graph formulation and we must compute cost(s(v'), pre(v')) every time a precondition on v' appears. Similarly, if the effect of the operator transforms v' to a value that is not one of the preconditions of a transition out of d' the modified algorithm captures the negative effect of the transition. Using the proposed algorithm above we maintain the acyclic causal graph to exploit the hierarchical relationship between variables while using the maximum amount information about the operators of the planning task.

When a transition is followed in the DTG of a variable v we first consider the updated value d'. If we have reached the goal value we can again ignore any transitions out of that node and we place a new vertex in the open list with the path cost equal to the cost of reaching that value. We follow the same method for any vertices with no outward transitions, and ignore any of those values if they are removed from the queue and is not the goal value. In this manner we search for a path in the domain transition graph from the starting value to the goal value under the constraint that it minimizes the total cost of satisfying the labels of the transitions.

Returning to the gripper example, consider a world state where one of the balls is being held while we are computing the heuristic estimate of moving a different ball from room A to room B. When we solve the subtask of moving the other ball to room B the heuristic state is updated to reflect the robot being in room B. When the next ball is evaluated, the one being held, the robot is already in room B and can drop the ball instead of starting back in room A and having to take the extra transition.

In planning tasks where there are no labels on the transitions or there

Algorithm 8: CONSTRAINED HEURISTIC WITH FULL ACTIONS (CCGHA)

```
Input: The subtask variable $v$, goal value, and current state $state$
Output: An estimated distance
open_list ← preprocessing();
while (¬open_list.empty()) do
    Vertex $v$ ← open_list.top();
    if (v.source = goal_value) then
        state ← v.assignment;
        return v.cost;
    end
    if distance[v.d][v.t] < v.pathcost then
        continue;
    end
    distance[v.d][v.t] = v.pathcost;
    State next_assignment ← apply_operator(v.assignment, v.op);
    next_distance ← v.pathcost + $\mathcal{C}$;
    $d'$ ← transitions[v.t].cost;
    if (d' has no outward transitions OR d' = goal_value) then
        open_list ← (d', next_distance, -1, next_assignment);
        continue;
    end
    for each transition $t'$ out of $d'$ do
        label_cost ← 0;
        for each precondition pre(v') = u on $t'$ do
            // Recursively solve label conditions
            label_cost ← label_cost+ CCGHA($v', u, state$);
            next_assignment[v'] ← u;
        end
        next_distance ← label_cost + next_distance;
        open_list ← ($d'$, next_distance, transition_id, next_assignment);
    end
end
return $\infty$;
```

are no positive interactions between high level variables their ancestors this heuristic reduces to the original causal graph heuristic. As a result of this, and because of its' additive form, this proposed extension is also inadmissible. Further, states may be returned with infinite heuristic values indicating that a solution cannot be reached, even if a path to the goal exists rendering the heuristic incomplete. Our empirical analysis indicates that this heuristic is able to improve search performance in certain domains by reducing the number of nodes that must be expanded to reach the goal state. Further, using this formulation a best-first search can find solutions with path lengths less than or equal to that of the causal graph heuristic on most domains tested.

Chapter 5

Experimental Evaluation

There are two components to the planner implemented for this work that we analyse. The first is comparing how the two parallel algorithms perform when compared against sequential A* and IDA*, and how the parallel algorithms scale. Secondly we look at the performance of the planner using an implementation of the original causal graph heuristic and the constrained causal graph heuristic with full actions. All experiments were run with the same planner as we are interested in how each algorithm is able to improve a simple planner. The planner does not feature any advanced pruning techniques, preferred operator selection or goal management. The heuristics used in the experiments are inadmissible as the planner has been designed for satisfying planning tasks. The planner was run on an Intel Xeon X5650 with up to 8 processors running at 2.67 GHz and 2 GB of RAM, search time was restricted to 30 minutes.

The problem set being used to benchmark the performance of this planner comes from the *International Planning Competition* that are included in the Fast-Downward repository. Our experiments tested the performance of the parallel algorithms by using threads equal to powers of 2, from 2 to 64 threads, to examine the performance of the algorithms when the number of threads exceeded the number of processors available. Some differences can be observed in the run-time and number of state expansions performed by the parallel algorithms because of non-deterministic qualities of the algorithms. These benchmarks were tested several times to ensure that the variations we observed were not too significant and the run-times used in this analysis are the average of all runs.

5.1 Parallel Performance

When analyzing the performance of parallel algorithms there are two common benchmarks used. The first measure used is *speedup* which is the comparison of the runtime between the sequential algorithm and parallel algorithm[RR10]. Speedup is important as it demonstrates the saving in run time that can be achieved by executing the program on p processors.

Speedup is defined by Equation 5.1, where $T_1(n)$ is the time it takes to execute the algorithm sequentially, p is the number of processors used for parallel execution, and $T_p(n)$ is the parallel execution time.

$$S_p(n) = \frac{T_1(n)}{T_p(n)} \tag{5.1}$$

When designing parallel algorithms the goal is to achieve a speedup $S_p(n) \geq p$, a linear to super-linear speedup. Achieving super-linear speedups is rare in practice as the overheads induced by parallel algorithms introduce additional CPU time needed to complete the task. As mentioned in chapter 2 search overheads, data dependencies and communication overhead can all negatively impact the performance of the parallel algorithm[RR10]. If processors wait to access a global data structure, or temporarily stop their search to broadcast messages to each other, then additional runtime is needed during the search. The overhead effects the efficiency of the algorithm which can result in sub-linear speedups with $S_p(n) < p$.

Our results demonstrate that on average sub-linear to linear speedups can be achieved using our parallel techniques, with the ability to achieve super-linear speedups on certain instances. Detailed plots of the speedups achieved can be found in Appendix A for the different benchmarks for 2 to 32 threads. In the *logistics* plot, Figure A.1, the general trend illustrates that speedups on average are sub-linear, with some approaching linear speedups. Several trends can be observed from the following plot. First, we can see that when the problem instance is relatively easy the greater speedups were achieved using four threads. When more threads are used on these instances the overhead incurred from locking the shared lists negatively impacts the parallel performance when the search is relatively simple. Further, because additional states are expanded that are not along the solution path the extra state expansions negatively effect the efficiency of the algorithm. The closer the speedup is to 1 then the closer its run time is to that of the sequential algorithm, the efficiency is nearly $\frac{1}{p}$ where p is the number of processors. Efficiency is the ratio of the sequential search time to the number of processors multiplied by the parallel search time, or speedup divided by the number of processors. This is illustrated in Equation 5.2[Fos95]:

$$E(n) = \frac{T_1(n)}{p \cdot T_p(n)} = \frac{1}{p} \cdot S_p(n) \tag{5.2}$$

As problem instances begin to get more difficult we begin to observe closer to linear speedups on 8 threads, while some instances can be solved with super linear speedups with 4 threads. By deferring the heuristic evaluation to the remaining threads the main thread can proceed through the state space more efficiently. Additionally because a stack structure is used to store successor states the secondary threads are continuously evaluating the most recently generated successors. These problem instances could all be solved sequentially in a few seconds using A*, when fewer threads were used in this domain there is not only a reduction in the search time, but also in the number of states that the main thread must expand compared to running the same instance with 8 threads. As a result super linear speedups could be reached on some of these instances. On average the four threaded runs experienced speedups of 3.76 times, an efficiency of 0.942, while the 8 threaded version experienced 4.26 times speedup, an efficiency of 0.53. This domain demonstrates that when the problem instance is relatively simple the synchronization overhead and search overhead can dramatically effect the performance when increasing the number of threads.

Figure A.2 is the same plot as described above, however this plot is comparing the speedups on the *logistics* domain using the proposed CCGHA heuristic. Running the benchmark tests with the proposed heuristic it can be observed that the 4 and 8 thread runs experienced the largest improved efficiencies at 0.48, corresponding to a 1.9 times speedup with 4 threads and 3.84 times speedup with 8 threads on average. The sequential A* search on this domain using the CCGHA heuristic was typically higher compared to using the causal graph heuristic due to the longer time needed to evaluate the heuristic the search could be performed more efficiently by deferring the evaluations to the secondary threads. As the number of threads increased to be larger than the number of physical processors available we observed a decrease in performance, with the average speedup becoming 1.4 times with 16 threads and 0.61 with 32 threads, demonstrating that the overhead incurred from the need to switch between active and inactive threads decreases the efficiency of the search.

Table 5.1 and Table 5.2 illustrates some of the speedups achieved on various benchmark problems, comparing the performance of the parallel algorithms using the original causal graph heuristic and the constrained causal graph heuristic. This table demonstrates one of the unexpected results; the speedups using the original causal graph formulation were typically larger than those achieved using the constrained formulation. As our proposed formulation typically requires longer to compute it was thought that it would experience greater speedups over the causal graph heuristic, however our results illustrate that this is not the case. This lead us to believe that both the heuristic used and the structure of the state space has an effect on the performance of the parallel implementations.

Table 5.1: A* Speedups with 8 Cores.

Problem	CGH Speedup	CCGHA Speedup
Block 11	3.53	3.40
Block 15	2.92	2.76
Logistics 19	7.16	8.62
Logistics 25	7.45	4.04
Logistics 28	7.99	7.17
Satellite 10	7.46	-
Scheduling 10	2.92	2.37
Transportation 5	3.71	3.36

Figure A.5 illustrates the speedups achieved on the first 10 *satellite* benchmarks using 2 - 16 threads using the causal graph heuristic. This plot indicates that the parallel A* algorithm scales well on the more difficult problem instances, and achieves super linear speedups on some domains. An important trend that can be observed from this plot is the speedups with 16 threads. The speedups achieved are rarely better than the speedups with 2 threads. When experimenting with the benchmarks with more threads than phyical processors available we observed the greatest differences in terms of run-time and node expansion. The processors automatically schedule the threads and the main search thread may be forced to sleep for portions of the search leading to a higher run-time. A consequence of this is that the secondary threads spend more time evaluating states that do not lead to goal states increasing the number of expansions the primary thread must perform. For this reason our analysis is more focused on the speedups achieved when the number of threads is less than or equal to the number of processors available.

The additional overhead required for the parallel search negatively impacts the efficiency of the parallel search algorithms. The main source of overhead is synchronization overhead, where threads must wait to access the global open and process lists. Additionally this leads to an increase in search overhead as well as the number of nodes generated and expanded is consistently greater than the amount generated by the sequential search algorithm. As the secondary threads wait, potentially with a state along a solution path, the main thread may end up searching several dead end paths before getting the opportunity to expand the state that gets us closer to a goal. Using a priority queue for the open list allows these nodes to be sorted for the main thread to search the state space in a greedy manner. Using a stack for the open list allows the search to proceed in a depth first manner.

One domain that experienced unusual behaviour with our parallel algorithms is the *gripper* domain. As the problem instances became more difficult we observed a decrease in speedups when using the causal graph heuristic. When the same set of benchmarks were used with the constrained causal graph heuristic we observed extreme super-linear speedups. We believe that the speedups in this domain is a result of the parallel algorithms combined with a more informed heuristic. In this domain we are able to guide the search more efficiently towards a solution. The additional overhead of searching paths that do not lead to a solution negatively impacts the efficiency of the algorithm. The manner in which nodes are expanded and successors are placed on a stack allows the search algorithm to find solutions much more efficiently by proceeding deeper into the state space. The *gripper* domain has a high number of paths that lead to goal states, and combined with our extension to the causal graph heuristic our parallel algorithms are able to exploit this to experience a massive speedup. Additionally, on average the extended A* algorithm was able to detect a goal state with 2-4 times fewer node expansions than sequential A* on the same instance. While the domain is considered to be trivial[HFG08], this demonstrates that our implementations can take advantage of the structures of the state space to experience large speedups.

Table 5.2: IDA* Speedups with 8 Cores.

Problem	CGH Speedup	CCGHA Speedup
Block 11	7.13	-
Block 15	9.26	9.99
Logistics 19	2.15	7.92
Logistics 25	2.65	7.93
Logistics 28	5.94	15.02
Satellite 10	-	-
Scheduling 10	0.82	9.38
Transportation 5	5.20	1.74

The modified IDA* algorithm with parallel heuristic evaluation experience the largest speedups on average. The reason for linear, and occasionally super-linear, speedups is most likely due to the fact that an inadmissible heuristic was used for the evaluations. As a result the search algorithm could get to the bounding depths more efficiently. Since the bound is larger than the optimal solution depth, and solvable planning tasks have many goal states, we are able to reach a non-optimal solution more quickly.

Another situation where overhead reduced the efficiency of our implementations was when 2 threads were used to perform the search. In domains where the heuristic evaluation is non-trivial the secondary thread is not able to populate the open list efficiently enough to keep the main thread searching. This results in additional CPU time being wasted as the primary thread waits until it is able to expand another state.

There were some domains that appear to be an exception to this as illustrated by the two *scheduling* speedup plots (Figure A.9 and Figure A.10) where extremely large speedups were achieved when fewer processors were available. The reason for these speedups is most likely due to the fact that the majority of the states have equivalent heuristic values, requiring the sequential search algorithm to try many different paths because the number of states with equal f values is much larger. When the search is done with our parallel algorithms the main search thread does not have to worry about these ties and is able to go deeper into the state space. When 8 processors are in use more states are in the open list for the main thread to expand, however since there are still many states with equivalent f values the thread expands many more states. On the 9th *scheduling* benchmark where the speedup is over 1000 we observe a drastic difference in the number of states expanded; with only 2 threads the search is able to find a solution with less than 20 node expansions compared to over 5500 expansions when the search is performed sequentially or with 8 or more threads.

5.2 Comparing Heuristic Performance

To compare the performance of the extension to the causal graph heuristic we analyzed the number of states expanded to reach a solution as well as the path length returned. Our observations revealed that our extension to the causal graph heuristic generates solution lengths that are less than or equal to that of the original causal graph heuristic for a large set of domains tested. In this section we will focus on domains that were solvable by both heuristics, analyzing their performance by comparing solution lengths and search times. Table 5.3 shows some of the solution lengths found for different problem instances.

The constrained causal graph heuristic is an incomplete algorithm, and this was illustrated in the *airport*, *sokoban*, and *elevators* domains where the heuristic returned infinite values for solvable tasks. Of the benchmarks

Table 5.3: Comparison of Solution Lengths.

Problem	CGH Plan Length	CCGHA Length
Block 10	22	20
Block 14	20	24
Block 15	18	16
Elevators 12	16	17
Elevators 13	15	15
Gripper 5	45	37
Gripper 7	61	55
Logistics 25	65	65
Logistics 28	79	71
Rover 4	8	8
Satellite 4	18	18
Schedule 9	6	5
Transportation 6	29	25

tested the CCGHA performed most poorly on the *satellite* and *rovers* domains. In these domains the constrained heuristic values were much lower and overly optimistic resulting in a higher number of node expansions, increasing the search time. In both domains the planner used was unable to solve 75% (15/20) of the problem instances as either memory was fully exhausted or the timer had expired. On the *elevators* domain 70% (6/20) of the problems returned infinite heuristic values for the initial state indicating that the heuristic cannot find a path to the goal.

The *blocksworld* domain provides good examples to compare the two heuristics. The domain is of particular interest because there are lots of interactions between state variables[GLY$^+$08] and is a domain that the causal graph heuristic does not fair as well one[Hel06]. Figure A.13 illustrates the solution quality found using the original causal graph formulation and constrained formulation in the *blocksworld* domain. We can see that on average our proposed extension to the causal graph heuristic is able to guide the A* search towards a solution at a shallower depth in the state space.

The plots A.16 to A.19 illustrate the search times when using the CGH and CCGHA with a sequential A* implementation. We consistently observe that the causal graph heuristic is more efficient to compute than the extended CCGHA formulation, with the exception of the *gripper* domain. The improvement to plan quality comes at a trade-off of an increase in search time.

In the *logistics* domain the CCGHA generates larger heuristic estimates and longer run-times than using the original causal graph, see Figure A.16. While we observe a large increase in the runtime towards the end of the benchmark set we continue to see solutions of a better quality, illustrated in Figure A.2. The reason for the overestimate is because multiple packages may be at the same city to begin with, and need to travel along the same route. As the heuristic state is updated once a solution to the subtask of one package has been found the trucks/planes may be in different cities than they were in originally and need to move back to their original location to get the next package, leading to an overestimate.

Chapter 6

Conclusion

In this work we proposed an extension to the traditional A* and IDA* search algorithms in the context of satisfying planning tasks. These extensions defer the heuristic evaluation of a state to secondary threads, allowing a primary search thread to proceed through the state space more efficiently. Additionally, we proposed a modification to the causal graph heuristic, combining its strengths with the strengths of the context enhanced additive heuristic. Our formulation frames the heuristic as a constrained shortest path problem and extends the search to use the full effects of the operators that transitions in domain transition graphs represent. This new formulation also maintains a heuristic state that is updated to reflect the changes to the state as domain transition graphs are traversed while solving a planning problems subtasks in an effort to capture as much information about the planning task when computing the heuristic value.

The results of this work demonstrate how modifying a state space search by deferring heuristic evaluation to parallel threads is able to provide speedups near linear when the problem instance and heuristic evaluation is non-trivial. The largest gains were to be had when running a modified IDA* with parallel heuristic evaluation.

While the planner that was implementing the parallel search algorithms is not as efficient as modern state-of-the-art planners, the results obtained from these experiments demonstrate the potential gains that can be achieved by implementing these techniques on a simple planner. Our analysis indicates that this technique scales well with the number of processors available and is even able to experience super-linear speedups on occasion. It also demonstrates how a simple planner can be extended to incorporate a simple parallel evaluation strategy to improve the performance of the planner in terms of reducing search time and increasing the number of problem instances that could be satisfied.

The most unexpected observation from this work was how the heuristic impacted the speedups of the parallel search algorithms. It was anticipated that our implementation would experience larger speedups with a computationally expensive heuristic, however our results demonstrate that this is not always the case. It was observed that the original causal graph heuristic experienced greater speedups, on average, overall. This work also demonstrates that the structure of the state space greatly impacts the performance of our algorithms. While they scale well with the number of processors, domains such as Scheduling experienced greater speedups with 2 - 4 processors as opposed to 8 or more. In domains where there are large numbers of states that generate the same estimated cost value much of the search time is spent expanding nodes at the same level without proceeding deeper into the state space. By deferring heuristic evaluation to fewer secondary threads, and by using a stack for nodes to be processed, the open list gets populated more quickly with nodes deeper in the state space allowing the algorithm to detect goal states in far fewer expansions than when the search is done sequentially or with a large number of processors.

The heuristic used was found to greatly impact the performance of the planner. In situations where the heuristic underestimates the distance to the goal both parallel algorithms described in this work must explore every state within the bound before proceeding to the next depth, like IDA*. This is why domains such as *gripper* experienced decreasing efficiencies with the original causal graph heuristic. As the problem instances got more difficult the bounds needed to increase multiple times and for this reason every state in the early bounds needed to be checked resulting in an increased overhead from regenerating the nodes, combined with the overhead for parallel search. When the same domain was searched with the CCGHA heuristic it was able to experience such large speedups. The heuristic estimates for the domain quickly decrease returning values closer to optimal, and because many paths to a goal state exist within the original bound our parallel algorithm is able to reach one of the goal states after increasing the bound only once.

When these heuristics were used for the sequential A* searches quite often a solution was found at a shallower depth than the heuristic estimate from the initial state. When the extension to A* with parallel heuristic evaluation was run for the same domains a trade-off in solution length was observed. Most solutions were returned at the initial overestimate depth, even though shorter solutions exist.

Additionally, our proposed extension to the causal graph heuristic demonstrates the ability to guide a best first search to goal states in fewer steps. While the extended heuristic proposed in this work is not admissible or consistent, it does demonstrate strengths over the causal graph in certain planning domains, those with additive effects between variables. With our extension a simple planner is able to find solutions of improved quality at the cost of longer search times.

6.1 Future Work

There are several ways in which the work done in this paper can be extended. The first is to implement this version of the causal graph heuristic on a state of the art planner to further test the strength of the heuristic in other planning domains. This would allow for a more thorough comparison between the original causal graph heuristic and our extension on a planner that incorporates preferred actions and goal management. Further, the heuristic could be extended to allow for planning problems that have conditional effect and axioms, two features not handled by this planner.

Modern planners also make use of various goal management techniques to further improve search efficiencies. This is useful in ordering operators to speed up the search, however it may be beneficial for the heuristic proposed in this work. Since the state of the world is updated after each subtask is solved the heuristic could further be strengthened by adding some form of ordering over the goal variables that would lead to the most positive effects being captured and resulting in a more informed heuristic estimate.

Additionally, it would be beneficial to examine the performance of parallel heuristic evaluation on a state of the art planner. A state of the art planner like Fast Downward includes the ability to evaluate a state using multiple heuristics and it would be worth analyzing if further performance gains can be achieved when multiple heuristics are computed in parallel. Conducting further analysis with the search algorithms using parallel heuristic evaluation with a computing cluster or some other massively parallel machine would allow for further analysis into the scalability of the algorithms when the number of physical processors available is in the hundreds or even thousands.

Another potential extension to this work would be to combine parallel heuristic evaluation with a parallel algorithm that partitions the state space, as discussed in Chapter 2. It would be worth analyzing if techniques such as parallel window search could experience further speedups by having a team of processors that compute the heuristic values of the states within its window. Further analysis on the choice of heuristic used, combined with the parallel algorithm, would also be beneficial to determine the strength of the parallel algorithms when admissible heuristics are used and the initial bound is much closer to that of the optimal plan length.

We also believe that our approach to performing parallel heuristic evaluations could be extended to other problems that involve graph or tree searches, such as Constraint Satisfaction Problems. It could be beneficial to modify our extension to IDA* to work with constraint propagation, allowing a primary thread to quickly attempt different potential assignments while secondary threads enforce a level of *k-consistency* to accelerate the search.

Bibliography

[Bar14] Blaise Barney. *Introduction to Parallel Programming*. Lawrence Livermore National Laboratory, 2014. → pages 9

[BG01] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001. → pages 1, 2

[BN93] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *COMPUTATIONAL INTELLIGENCE*, 11:625–655, 1993. → pages 2

[Boa08] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0, 2008. http://www.openmp.org. → pages

[Byl94] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994. → pages 1

[DD01] Carmel Domshlak and Yefim Dinitz. Multi-agent off-line coordination:structure and complexity. In *PROCEEDINGS OF THE 6TH EUROPEAN CONFERENCE ON PLANNING (ECP'01)*, pages 277–288, 2001. → pages 18

[Ebe87] Carl Ebeling. *All the Right Moves.* MIT Press, Cambridge, MA, USA, 1987. → pages 25

[EBZ10] Wheeler Ruml, Ethan Burns, Sofia Lemons and Rong Zhou. Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, 39:689–743, 2010. → pages 9

[Ede01] Stefan Edelkamp. Planning with pattern databases. In *PROCEEDINGS OF THE 6TH EUROPEAN CONFERENCE ON PLANNING (ECP-01)*, pages 13–24, 2001. → pages 2

[EMNH95] Matthew Evett, Ambuj Mahanti, Dana Nau, and James Hendler. PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25:133–143, 1995. → pages 10

[FH90] M. Campbell A. Nowatzyk F.H. Hsu, T. Anantharman. A grandmaster chess machine, scientific american. *Scientific American*, pages 44–50, October 1990. → pages 25

[FKK03] Ariel Felner, Sarit Kraus, and Richard E. Korf. KBFS: K-best-first search. *Annals of Mathematics and Artificial Intelligence*, 39:2003, 2003. → pages 13

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. → pages 42

[Gef07] Héctor Geffner. The causal graph heuristic is the additive heuristic plus context. 2007. → pages 23

[GLY$^+$08] Wen-Xiang Gu, Jin-Li Li, Ming-Hao Yin, Jun-Shu Wang, and Jin-Yan Wang. A novel causal graph based heuristic for solving planning problem. In *2008 International Conference on Machine Learning and Cybernetics*, volume 4, pages 2223–2228, July 2008. → pages 22, 23, 24, 47

[HD09] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: Whats the difference anyway?, 2009. → pages 2

[Hel04] Malte Helmert. A planning heuristic based on causal graph analysis. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004*, pages 303–312. AAAI Press, 2004. → pages 1, 2, 15, 16, 17, 18, 20, 22

[Hel06] Malte Helmert. The fast downward planning system. *Journal of Artifical Intelligence Research*, pages 191–246, 2006. → pages 3, 15, 16, 21, 25, 47

[HG00] Patrik Haslum and Héctor Geffner. Admissible heuristics for optimal planning. In *In Proceedings of AIPS-00*, pages 140–149, 2000. → pages 2

[HFG08] Malte Helmert, Albert ludwigs-universität Freiburg, and Héctor Geffner. Unifying the causal graph and additive heuristics. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS*. AAAI Press, 2008. → pages 23, 36, 45

[HN01] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:2001, 2001. → pages 1

[JJL13] Anders Jonsson, Peter Jonsson, and Tomas Lw. When acyclicity is not enough: Limitations of the causal graph. 2013. → pages 23

[KD09] Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. 2009. → pages 2

[KFB13] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195(0):222 – 248, 2013. → pages 1, 9, 10, 11, 25

[Kor85] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97 – 109, 1985. → pages 1, 7, 8, 33

[Mah11] Basel A. Mahafzah. Parallel multithreaded IDA* heuristic search: algorithm design and performance evaluation. *International Journal of Parallel, Emergent and Distributed Systems*, 26(1):61–82, 2011. → pages 1, 13

[Mcd00] Drew Mcdermott. The 1998 AI planning systems competition. *AI Magazine*, 21:35–55, 2000. → pages 2

[MD93] Ambuj Mahanti and Charles J. Daniels. A SIMD approach to parallel heuristic search. *Artificial Intelligence*, 60:243–282, 1993. → pages 12, 13

[NB12] Raz Nissim and Ronen Brafman. Multi-agent A* for parallel and distributed systems. In *IN 11th ICAPS WORKSHOP ON HEURISTICS AND SEARCH FOR DOMAIN-INDEPENDENT PLANNING*, pages 43–51, 2012. → pages 11

[PFK93] Curt Powley, Chris Ferguson, and Richard E. Korf. Depth-first heuristic search on a *SIMD* machine. *Artificial Intelligence*, 60(2):199 – 242, 1993. → pages 1, 12, 13, 25

[RN10] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*. Prentice Hall, 3 edition, 2010. → pages 1, 2, 6, 7, 32

[RR10] Thomas Rauber and Gudula Rünger. *Parallel programming : for multicore and cluster systems*. Springer-Verlag, Berlin, 2010. → pages 41, 42

[VBH10] Vincent Vidal, Lucas Bordeaux, and Youssef Hamadi. Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS*. AAAI Press, 2010. → pages 10, 13, 25

Appendix

Appendix A

Tables and Plots

A.1 Plots of A* Speedups

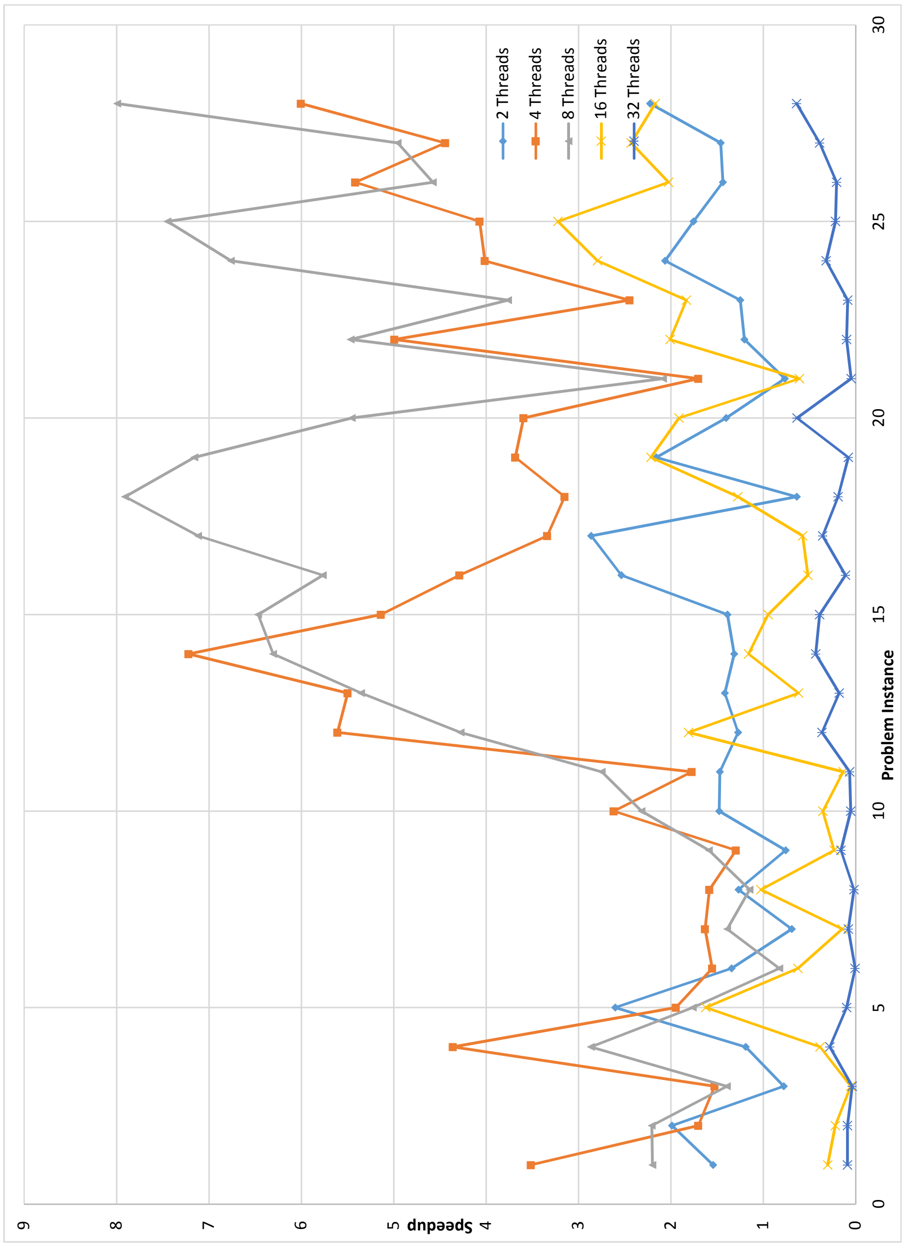


Figure A.1: Plot of Speedups on the Logistics Domain with CGH.

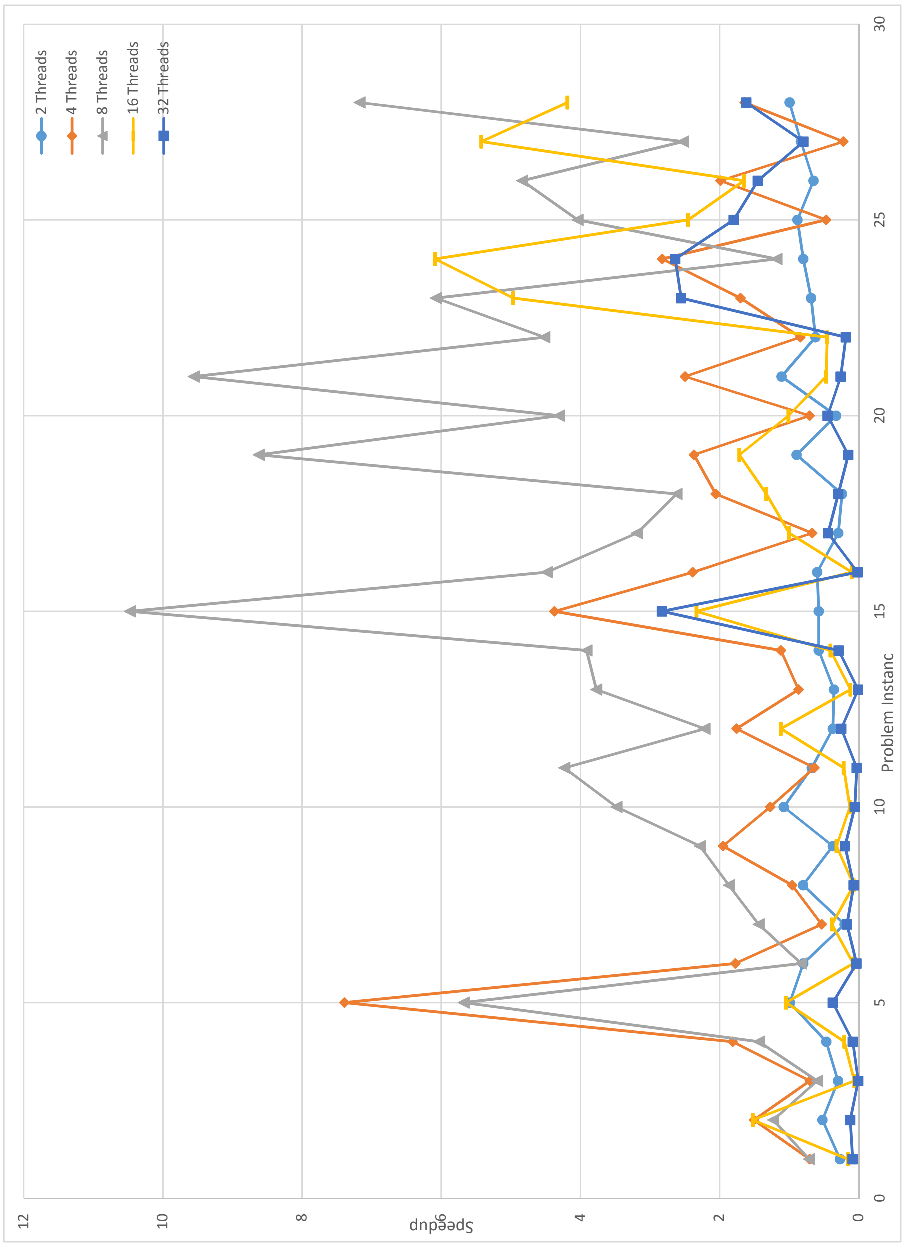


Figure A.2: Plot of Speedups on the Logistics Domain with CCGHA.

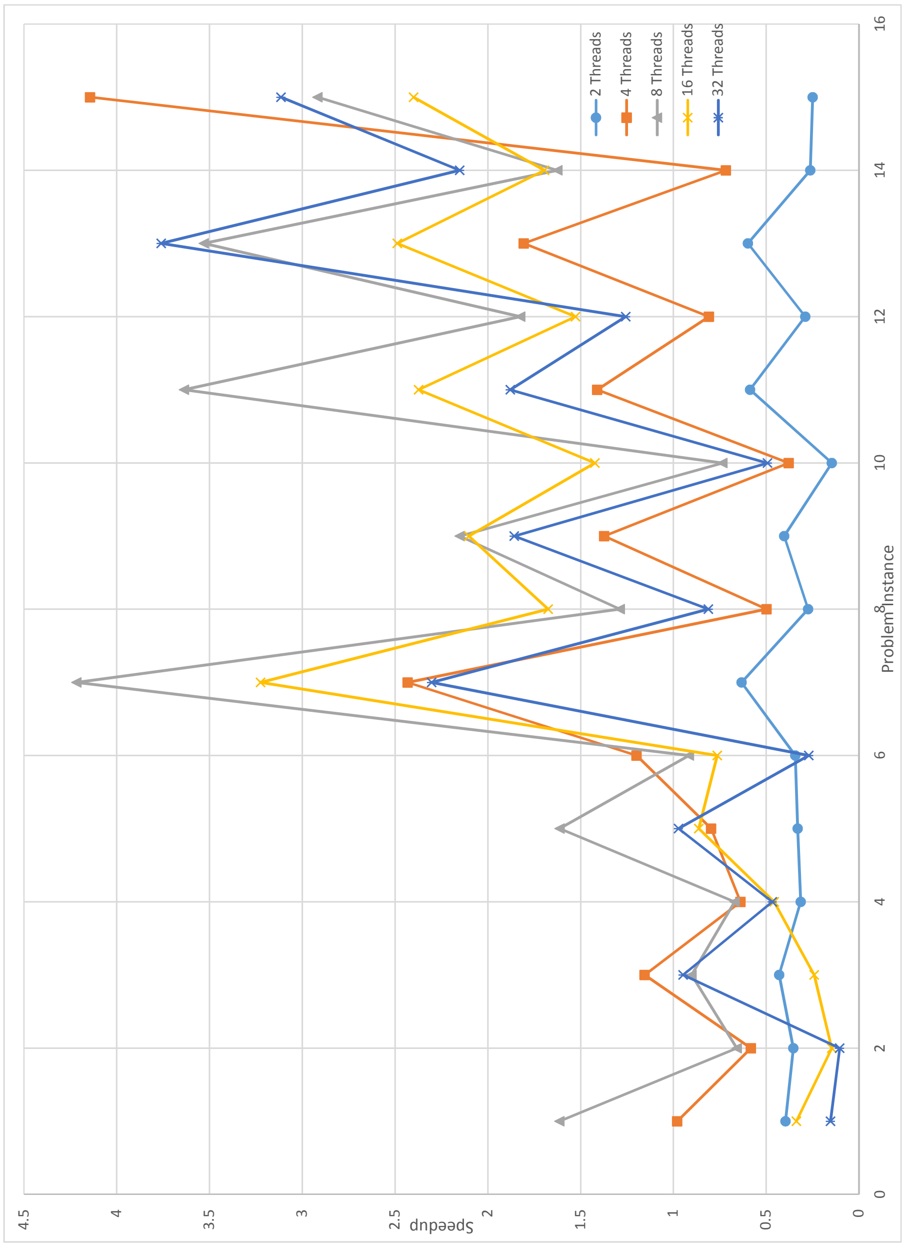


Figure A.3: Plot of Speedups on the Blockworld Domain with CGH.

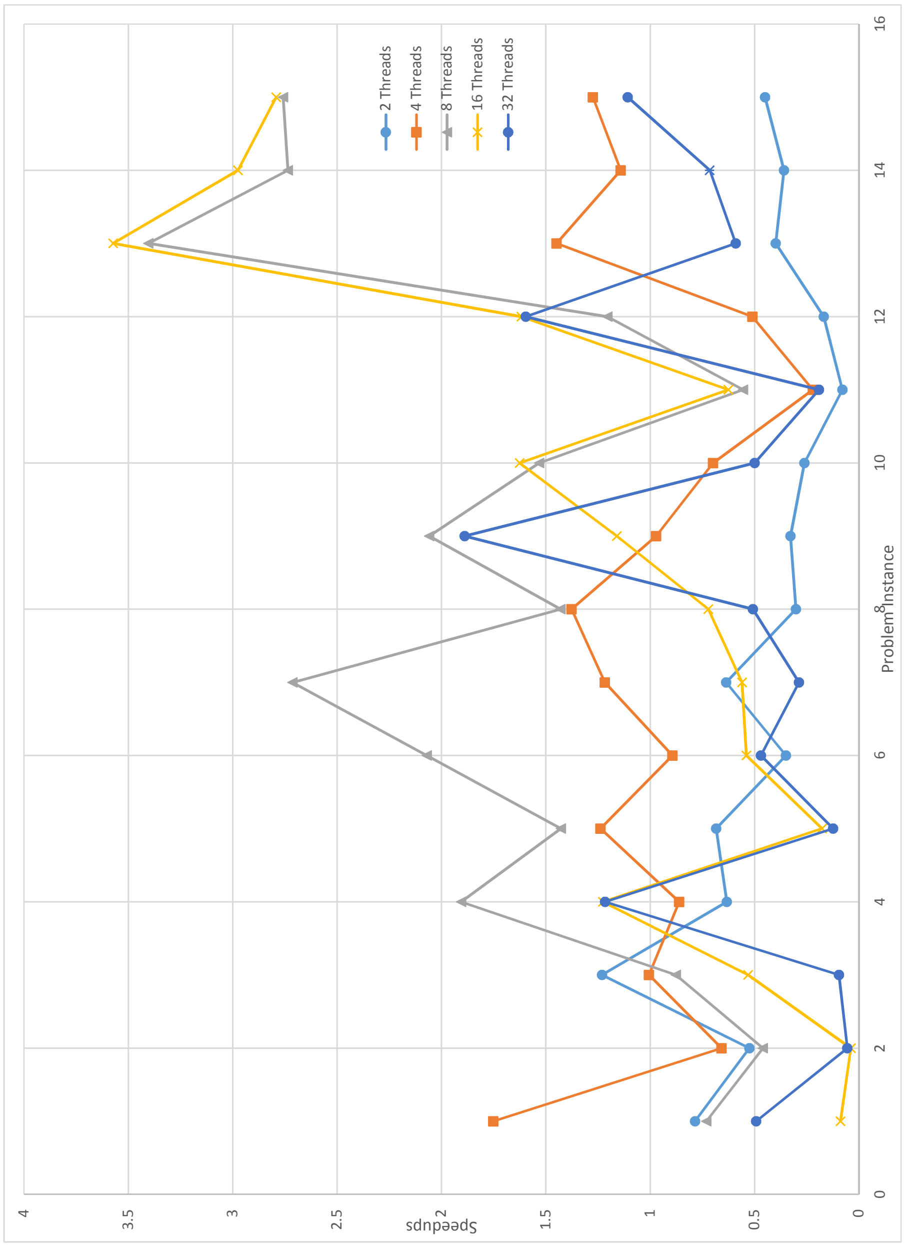


Figure A.4: Plot of Speedups on the Blockworld Domain with CCGHA.

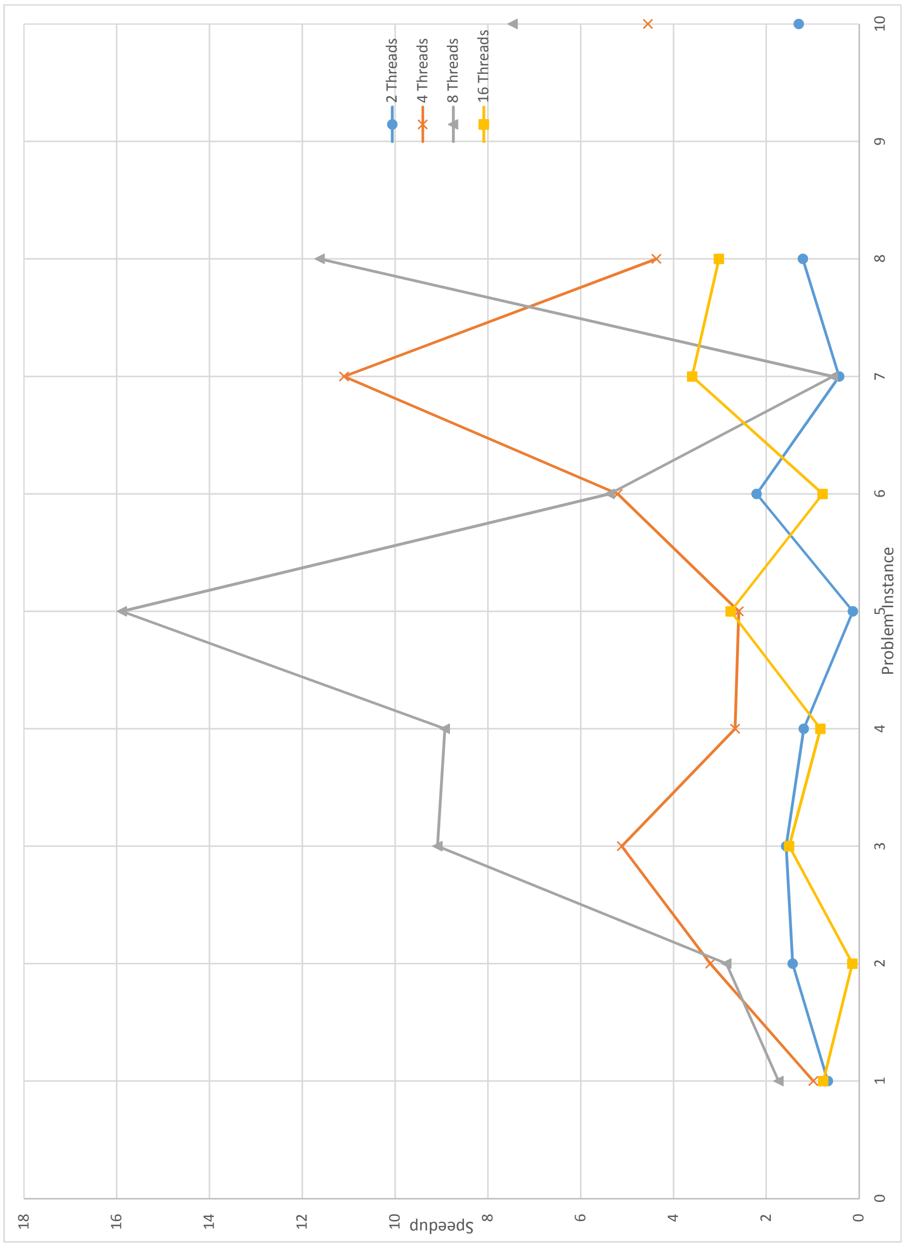


Figure A.5: Plot of Speedups on the Satellites Domain with CGH.

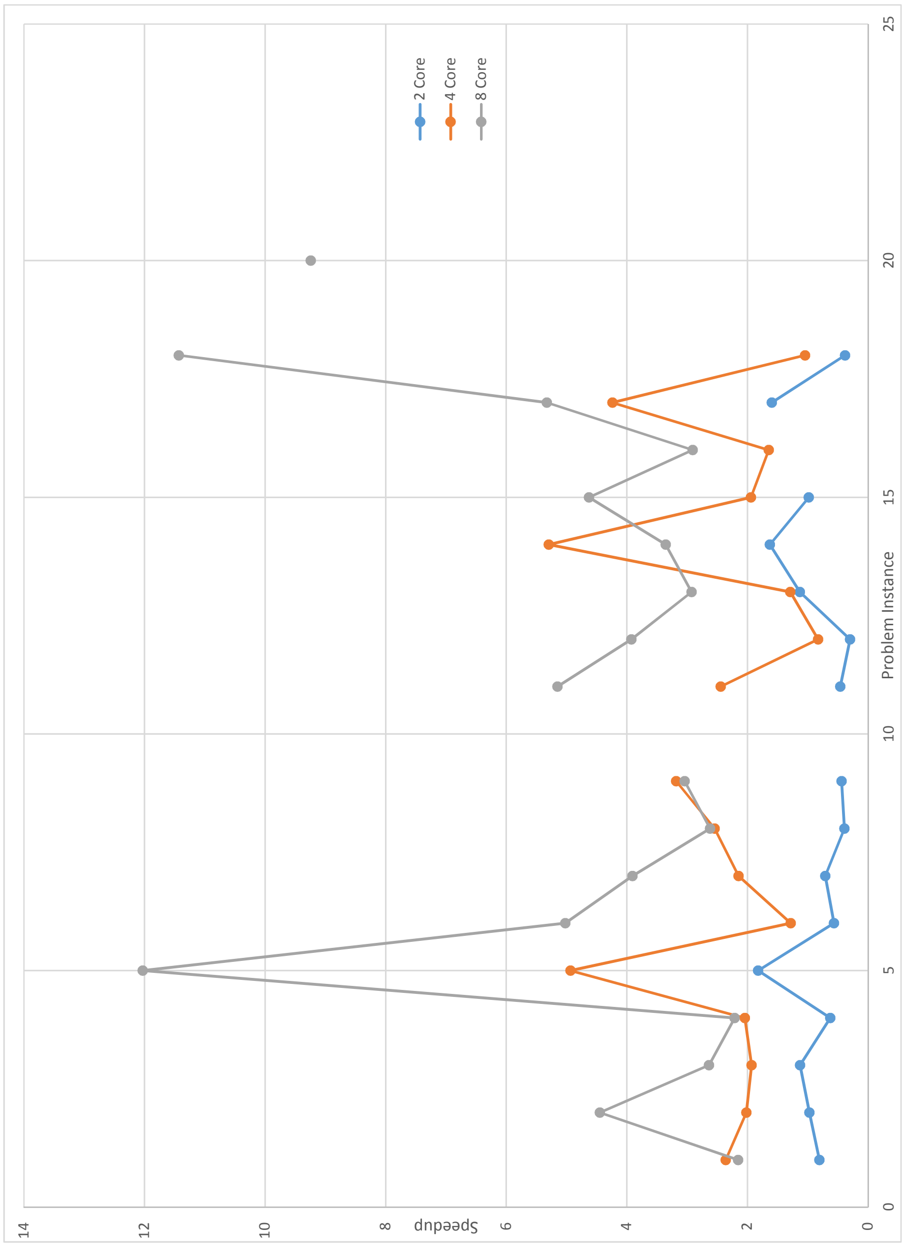


Figure A.6: Plot of Speedup on the Elevators Domain.

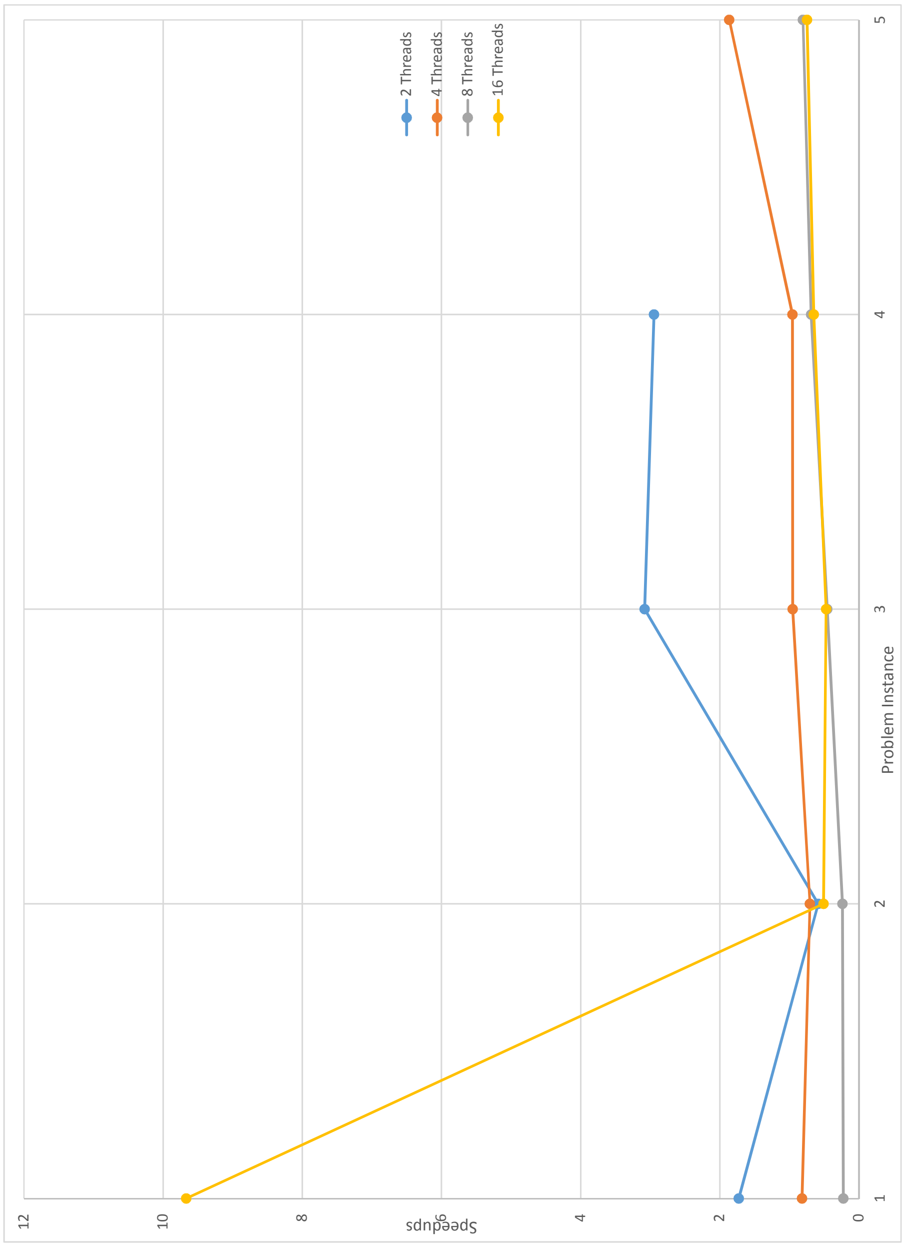


Figure A.7: Plot of Speedup on the Gripper Domain with CGH.

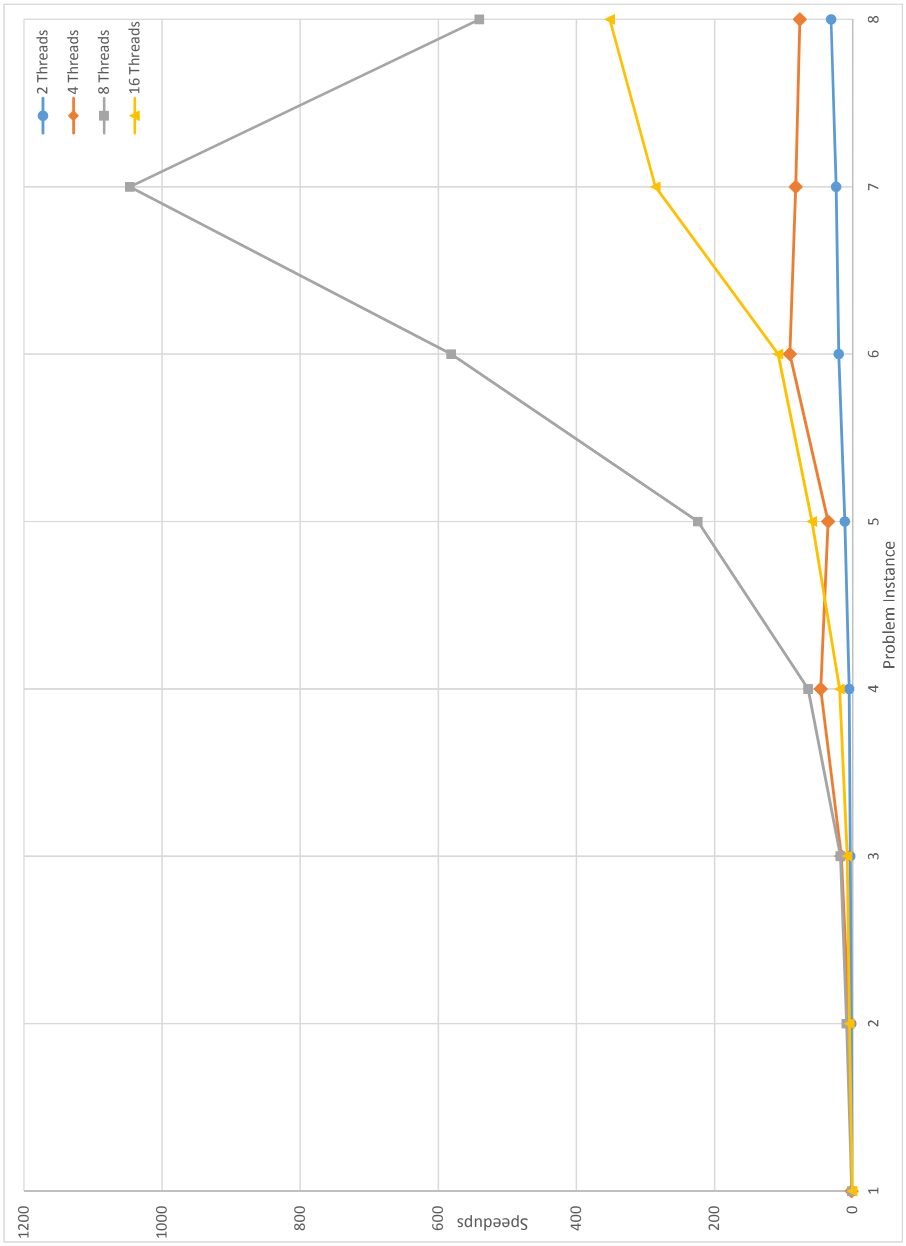


Figure A.8: Plot of Speedup on the Gripper Domain with CCGHA.

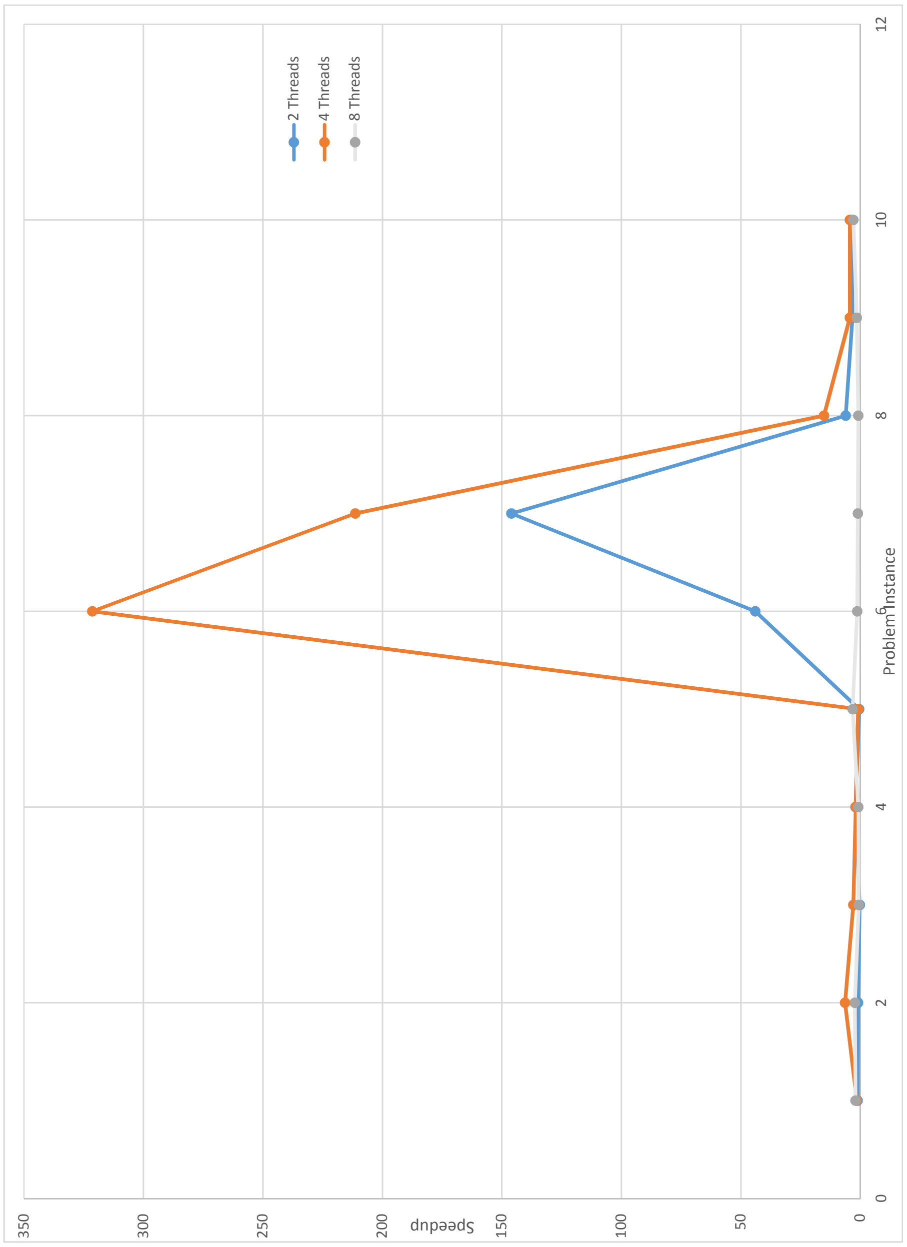


Figure A.9: Plot of Speedup on the Scheduling Domain with CGH.

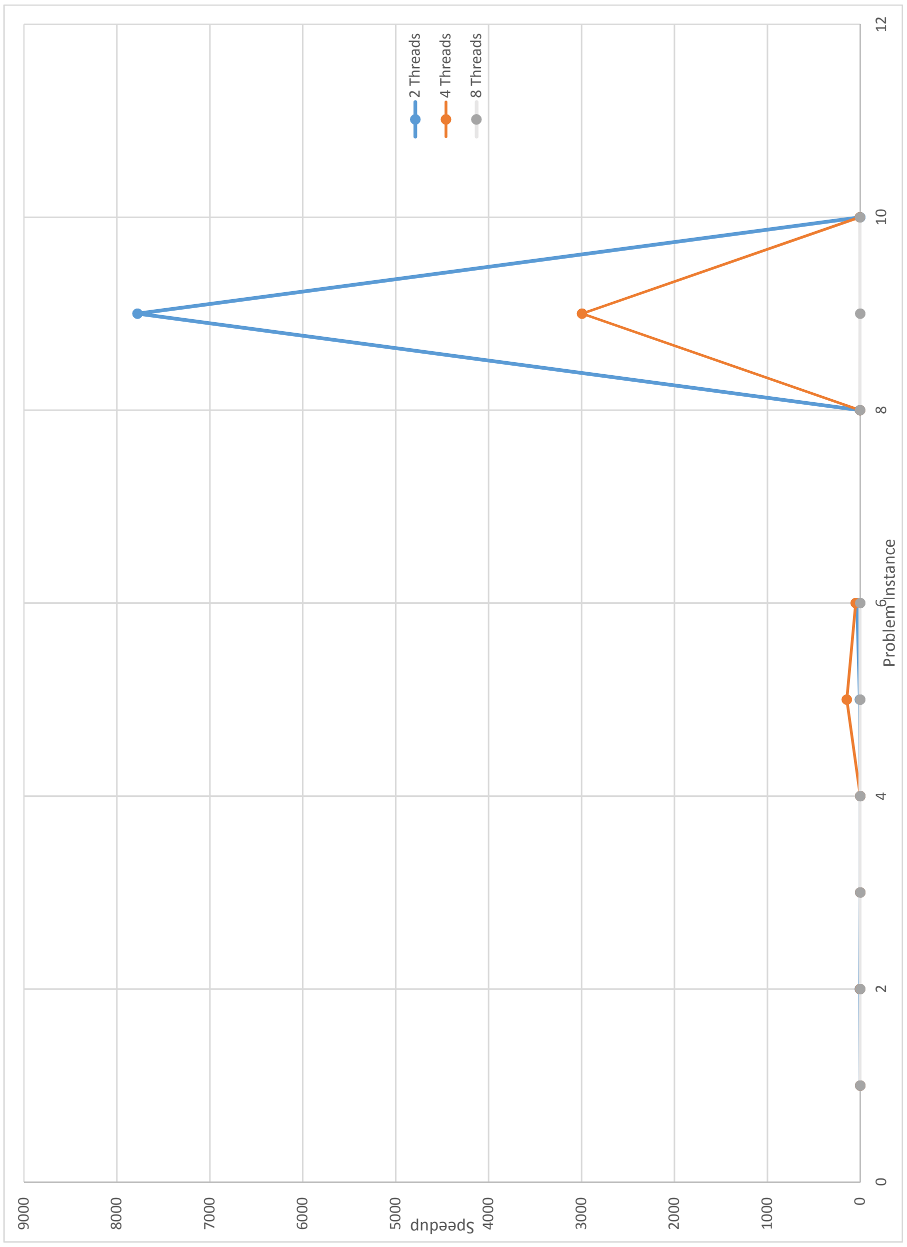


Figure A.10: Plot of Speedup on the Scheduling Domain with CCGHA.

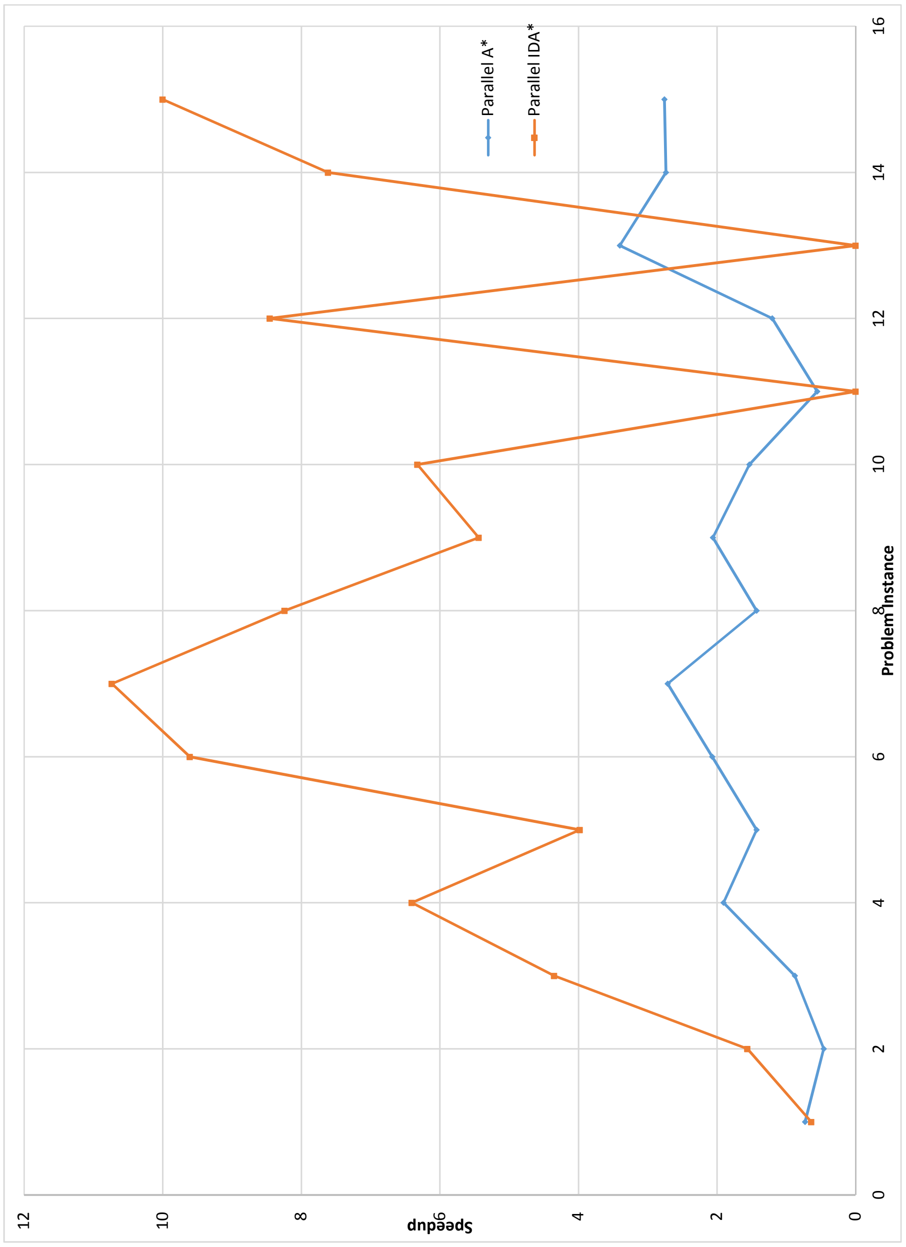


Figure A.11: Comparing Parallel A* and IDA* Speedups on Blocksworld Domain with CCGHA.

A.2 Comparing Solution Lengths and Runtimes

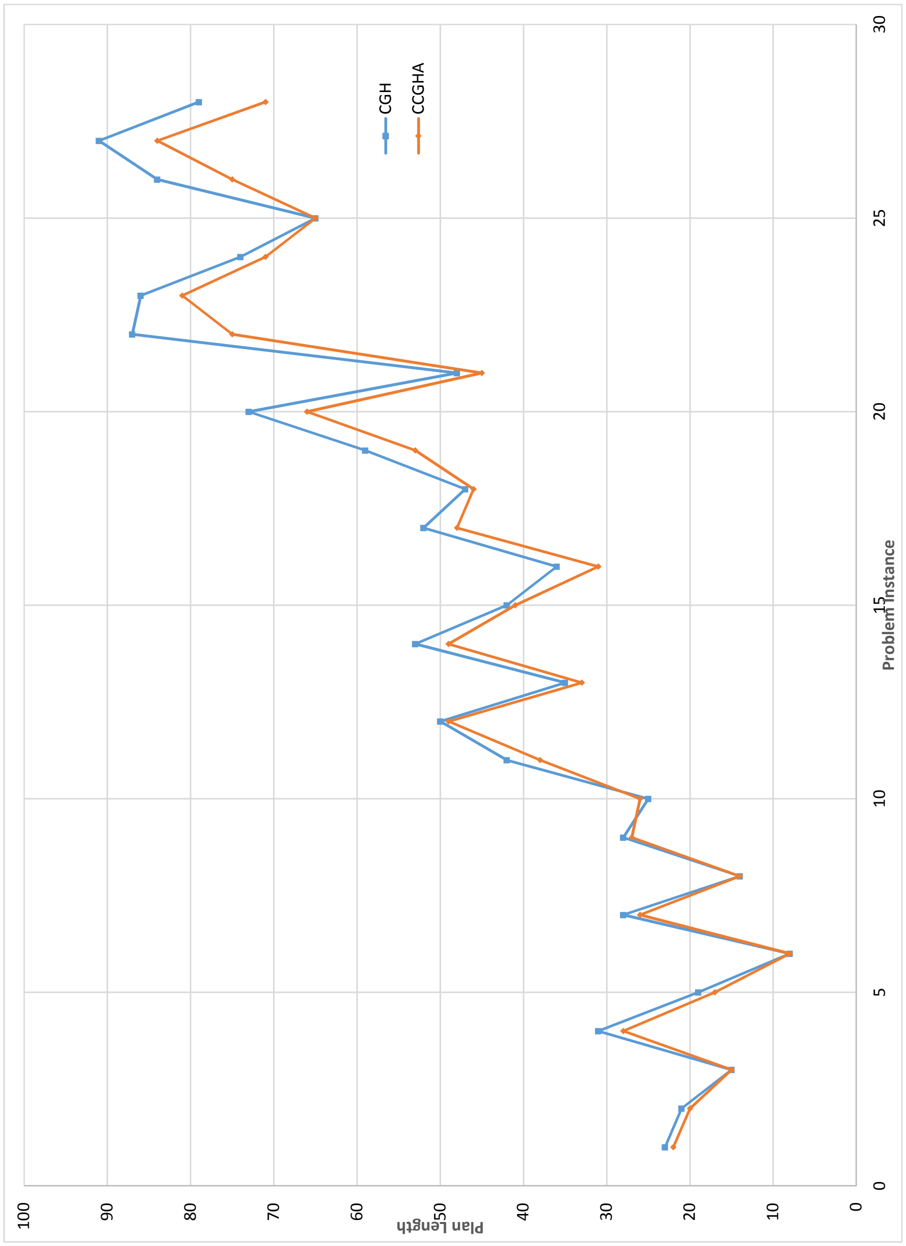


Figure A.12: Logistics Runtimes (top) and Solution Lengths (bottom)

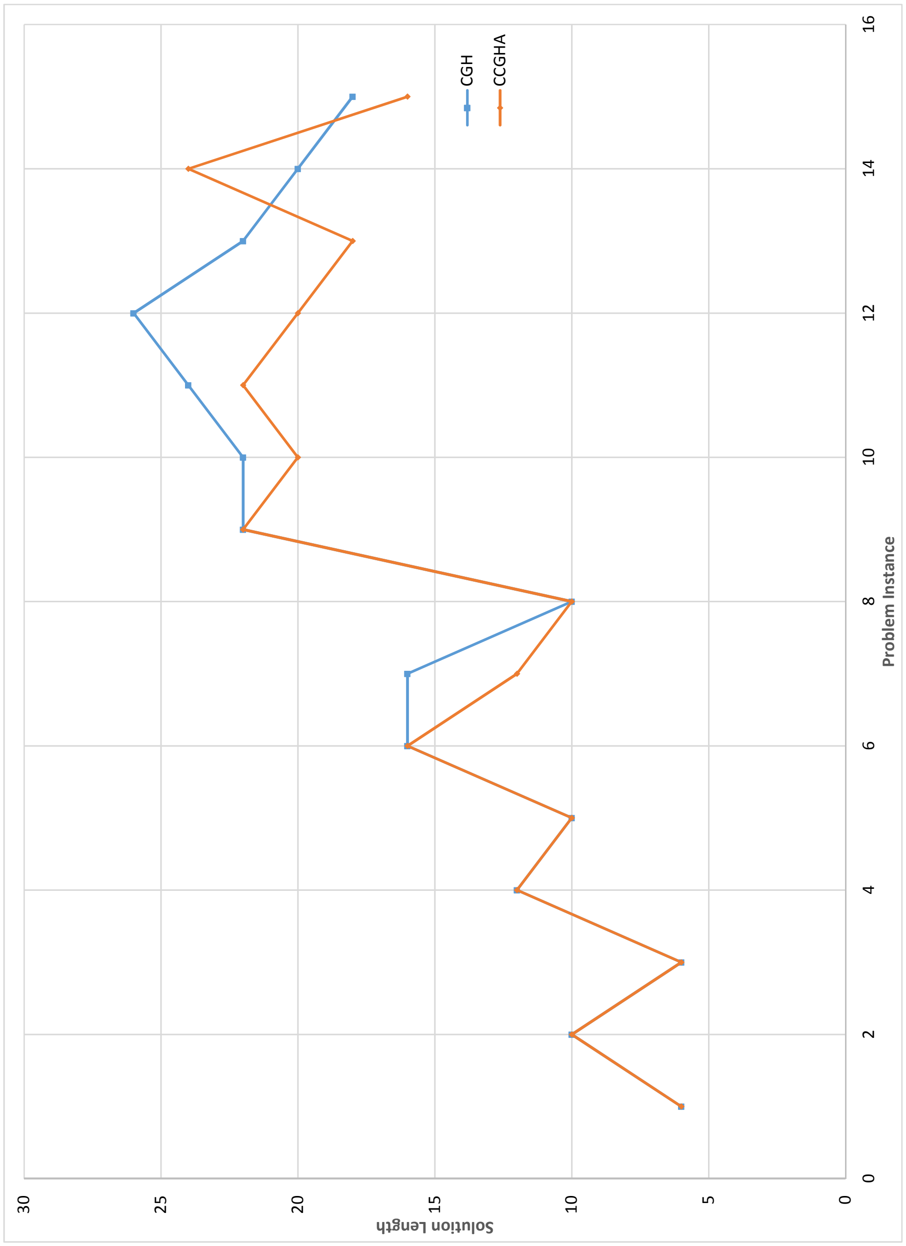


Figure A.13: Plot of Solution Lengths for Blocksworld Domain.

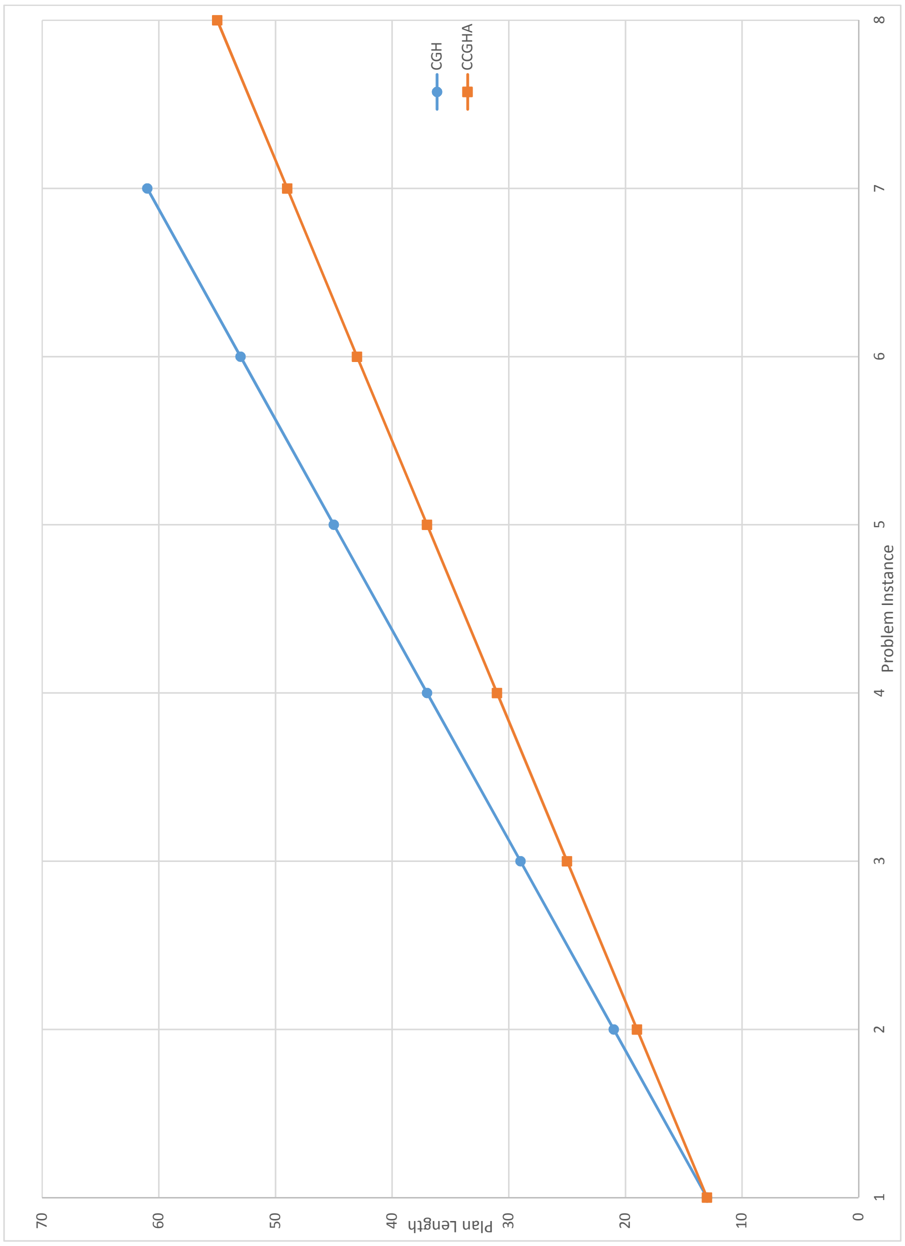


Figure A.14: Plot Comparing Solution Lengths for the Gripper Domain.

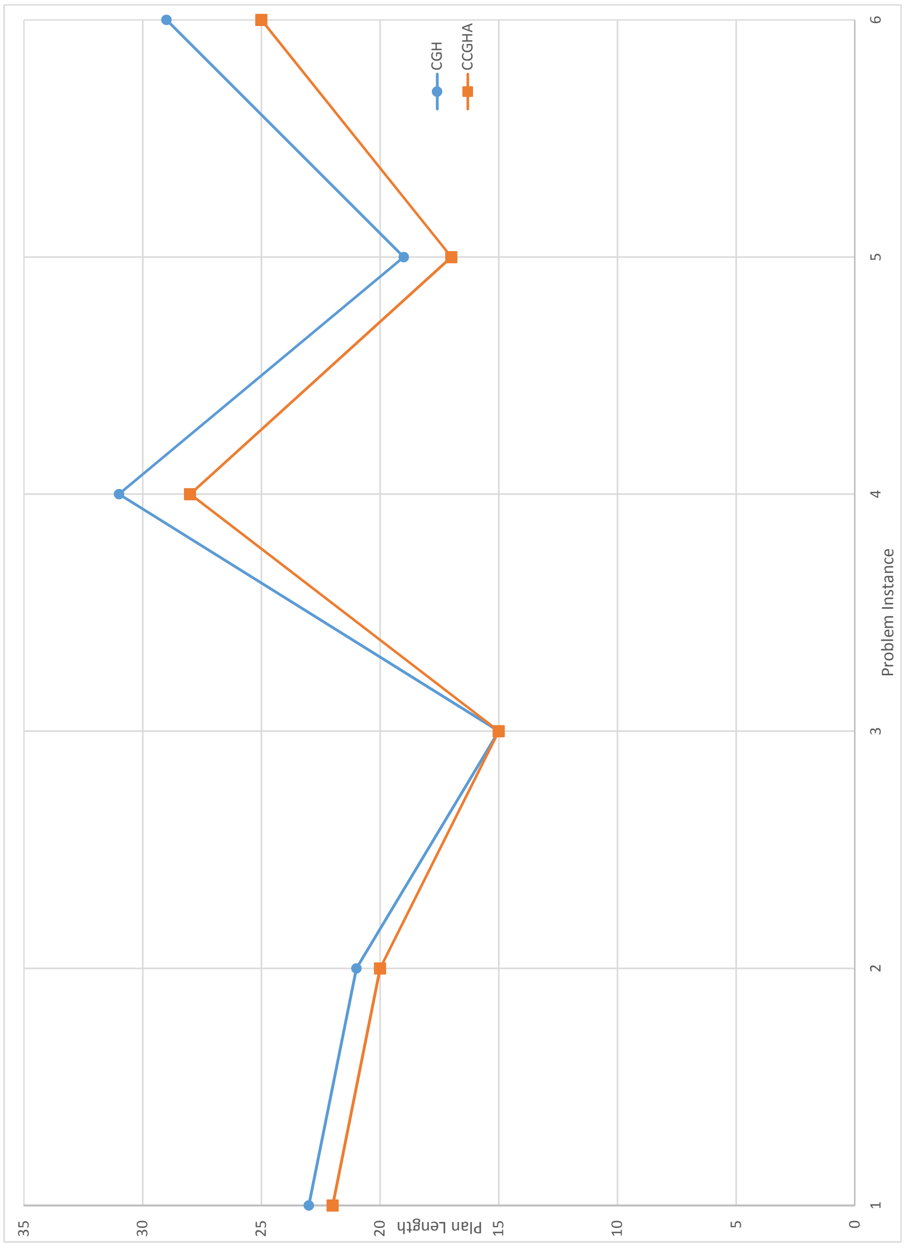


Figure A.15: Plot Comparing Solution Lengths for the Transportation Domain.

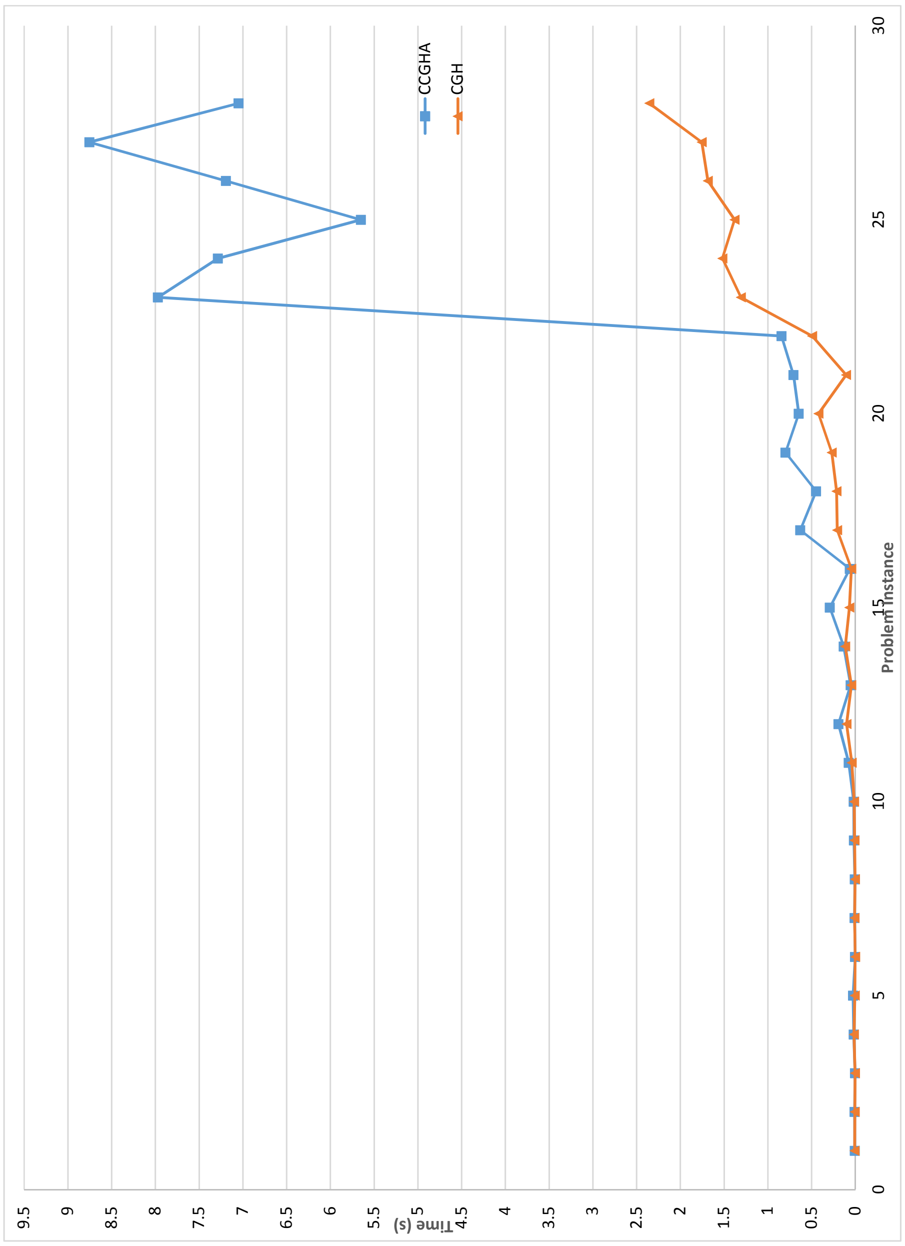


Figure A.16: Plot of Logistics Search Times with CGH and CCGHA.

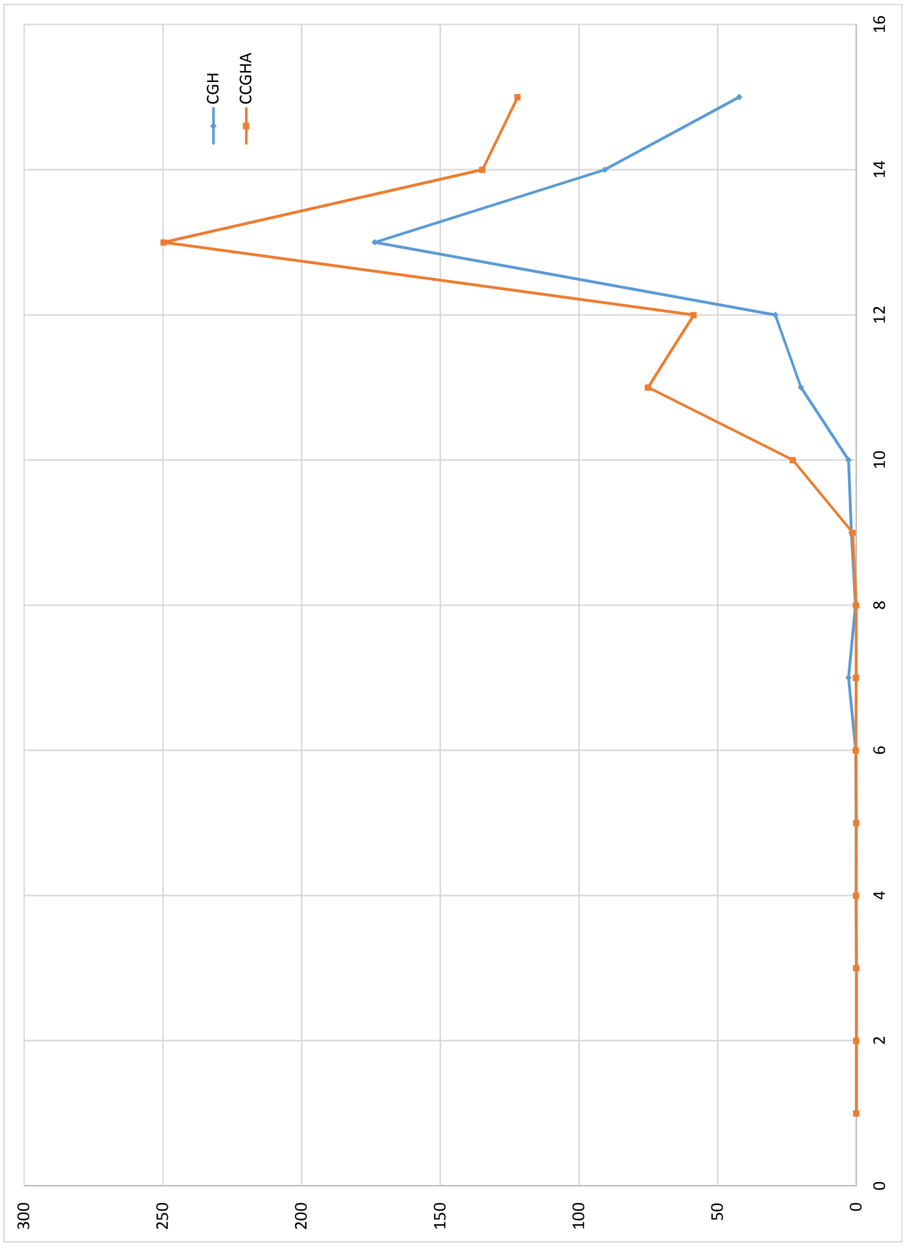


Figure A.17: Plot of Blocksworld Search Times with CGH and CCGHA.

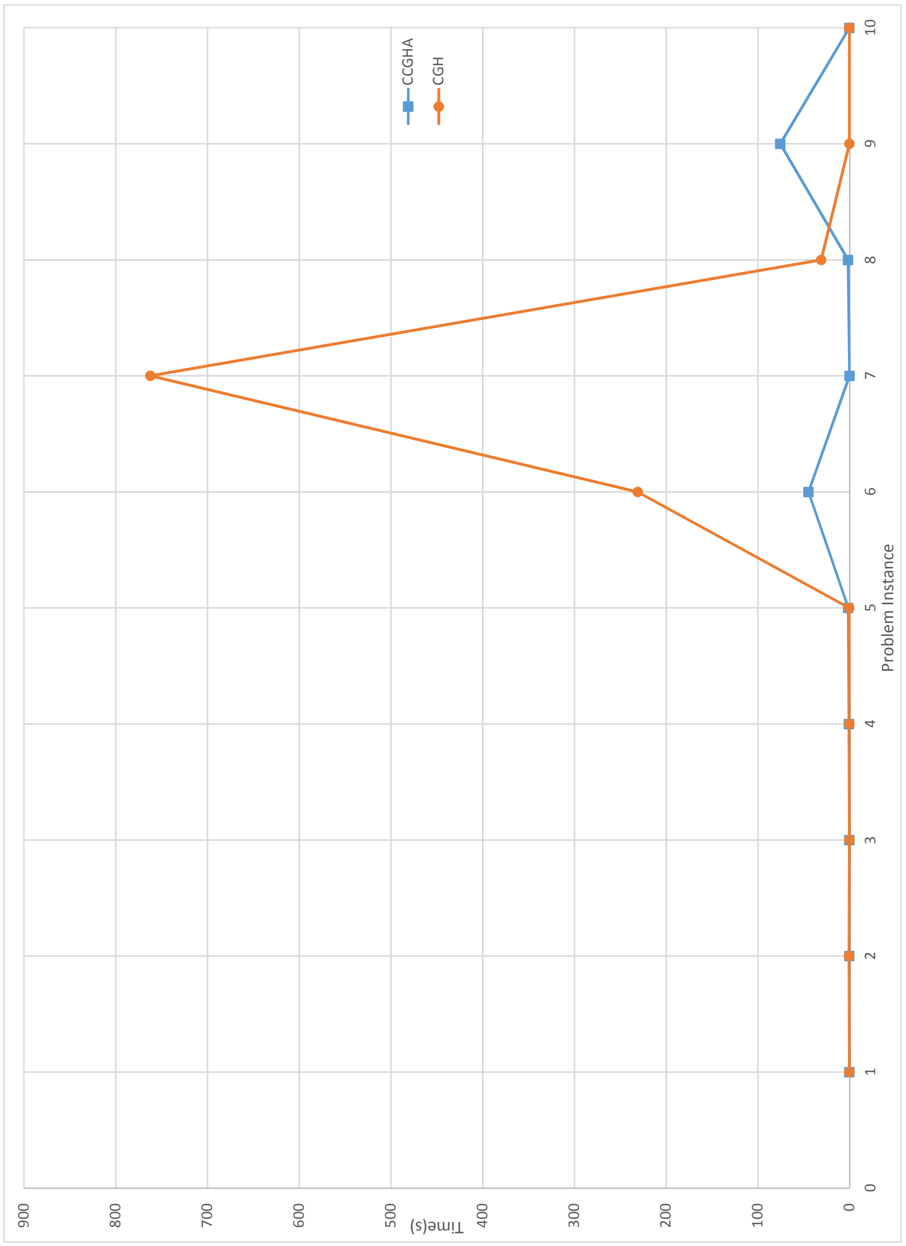


Figure A.18: Plot of Schedule Search Times with CGH and CCGHA.

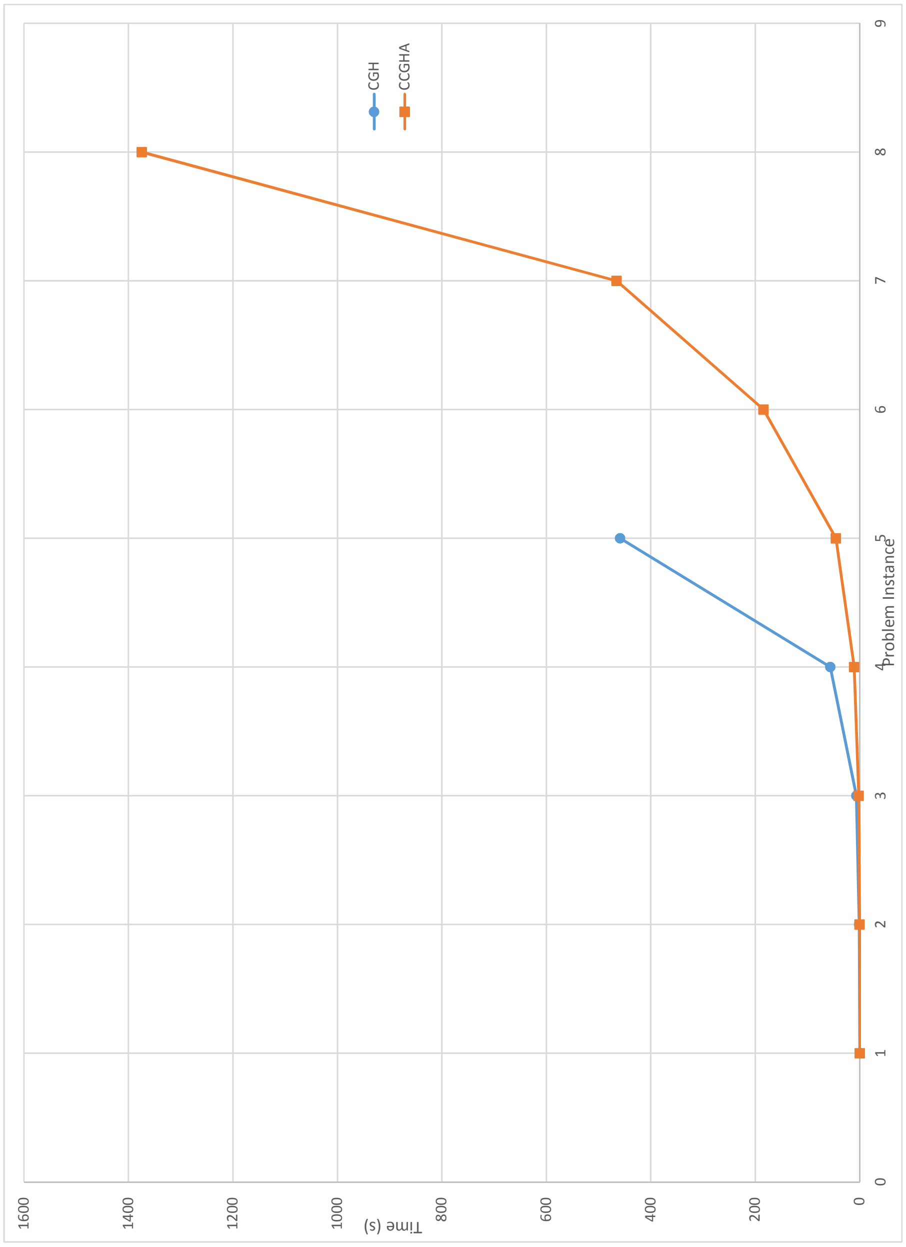


Figure A.19: Plot of Gripper Search Times with CGH and CCGHA.