5-2013

The Effects of Abstraction on Best NBlock First Search

Justin R. Redd
Utah State University

THE EFFECTS OF ABSTRACTION ON BEST *N*BLOCK FIRST SEARCH

by

Justin Redd

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

Dr. Daniel Bryce Major Professor	Dr. Dan Watson Committee Member
Dr. Scott Cannon Committee Member	Dr. Mark R. McLellan Vice President for Research and Dean of the School of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2012

ABSTRACT

The Effects of Abstraction on Best *N*Block First Search

by

Justin R. Redd, Master of Science

Utah State University, 2012

Major Professor: Dr. Daniel Bryce
Department: Computer Science

Search is an important aspect of Artificial Intelligence and many advances have been achieved in finding optimal solutions for a variety of search problems. Up until recently most search problems were solved using a serial-single threaded approach. Speed is extremely important and one way to decrease the amount of time needed to find a solution is to use better hardware. A single threaded approach is limited in this way because newer processors are not much faster than previous generations. Instead industry has added more cores to allow more threads to work at the same time. In order to solve this limitation and take advantage of newer multi-core processors, many parallel approaches have been developed. The best approach to parallel search is an algorithm named Parallel Best-*N* Block First Search (PBNF). PBNF relies on an abstraction function to divide up the work in a way that allows threads to work efficiently with little contention. This thesis studies the way this abstraction function chooses to build the abstraction and demonstrates that better abstractions can be built. This abstraction focuses on goal variables on ways to keep the number of abstract states as small as possible while adding as many variables as feasible.

(44 pages)

PUBLIC ABSTRACT

JUSTIN REDD

Search is an important aspect of Artificial Intelligence. Efficiently searching for solutions to large problems is important. One way to scale search large in problems quickly is to divide the work between multiple processors. There are many ways to divide this work using abstractions. This thesis examines the previous ways this has been done in the past and introduces other ways to more efficiently divide the work and search in parallel.

ACKNOWLEDGMENTS

I would like to thank my major professor, Dr. Daniel Bryce, for his patience, knowledge, and encouragement. I would also like to thank him, after the fact, for working me to the point at times of hating his guts. He stretched me in ways I didn't know were possible and I believe the things I learned are worth it. I would also like to thank my committee members for their willingness to be a part of this process.

I would like to give special thanks to my wonderful wife, Sarah, who took on a much larger burden these past few years and who at times was like a single parent to my awesome sons Nathan and Kalen. She didn't complain, but rather encouraged me and helped me succeed.

Justin Redd

CONTENTS

Page

ABSTRACT iii

PUBLIC ABSTRACT v

ACKNOWLEDGMENTS vi

LIST OF TABLES viii

LIST OF EQUATIONS ix

LIST OF FIGURES ix

INTRODUCTION 1

BACKGROUND 4

- Search 4
- Planning Problems 6
- Earliest approaches in parallel search 7
- Parallel Structured Duplicate Detection 8
- Domain Independent Structured Duplicate Detection 11
- PBNF 13
- Merge and Shrink Heuristic 15

EMPIRICAL RESULTS 17

- The Fast Downward Planner Integration 17
- PBNF Smallest 17
- Test Setup and Domains 17
- 3.4 Results 18

REFERENCES 28

APPENDIX 30

LIST OF TABLES

Table Page

1 A STRIPS example 6

2 Merge strategies that work well and a description how they work 16

LIST OF FIGURES

Figure Page

1 The state space for the vacuum world. 5

2 A* search Algorithm........ 5

3 Two disjoint duplicate detection scopes 6

4 Detecting distjoint duplicate detection scopes........ 10

5 Abstract state-space graph. 13

6 A sketch of basic PBNF search, showing locking [1]. 15

7 P-value to wall time for all problems all methods........ 19

8 P-value to wall time in seconds where P-value does better. 20

9 P-value to wall time in seconds where a better P-value does worse. 22

10 Best Abstractions. 23

11 Best Abstractions over 50%. 24

12 The average percentage speedup over PBNF default. 25

13 The average slowdown to PBNF default. 25

LIST OF EQUATIONS

Equation 1 [11] 11

Equation 2 [11] 11

CHAPTER 1

INTRODUCTION

An important element in Artificial Intelligence (AI) is using search to solve problems and it has many applications. Shipping companies use search to route packages from a business to their customers in the most efficient path possible. Manufacturers can use it to find the best way to build their product. It is used in vehicle navigation, video games, network routing, robotics, and function approximation, among others. Search problems are abundant and worthy of study.

One of the largest problems in AI is search. A simple example of this can be found when one uses a map application on the internet, phone, or car GPS. Planning an optimal route from source A to destination B can be a challenging problem. One could leave city A and take many different paths: using freeways, highways, or even dirt roads. Routes can be scenic, include detours, or multiple stops on the way. For example, source A has two possible roads out of town, and each city on the way to destination B has two possible routes out of the city. If one passes only ten junctions there are 1024 possible routes. Each route may be different, and its not always which is best. A naïve computer program might use brute force and find the answer to a simple version of the routing problem, but would quickly fail if, for example, planning a route between LA and Miami. Finding the optimal route via brute force would be infeasible even on the fastest computer.

There are many approaches that have been developed to search these large spaces to find solutions. Some approaches use the idea of heuristics (educated guesses as to which direction to take next), abstractions (generalization of the problem to a smaller version), or decomposition (where the problem is divided into smaller sub problems) [10] Most recent work focusses on serial approaches that use a single thread to search the search space. Clock speed increases on processors have decreased in the last few years and it is anticipated that more cores will continue to be added to gain more performance. For this reason some exciting work has been done on parallel methods in searching that provide leverage with these additional cores.

Parallelization can be achieved if the computational work can be divided. Therefore, an important aspect of parallelizing search is creating an abstraction of the state space that enables threads to efficiently work together. Abstraction organizes the search space so that states are not duplicated by other threads; this is important because the threads would otherwise constantly synchronize through mutual exclusive locks to avoid race conditions.

One of the most recent successful works on parallelizing search, created by Burns et al. [2], is called Parallel Best N-block First (PBNF). In this approach abstractions are created using a abstraction quality metric P*, which captures the degree of locality in the abstract state space graph. Locality is defined by Zhou and Hansen [13] as the maximum number of successors of any abstract node compared to the overall number of abstract nodes. PBNF and how the abstraction is created will be described later in this thesis.

There has also been much work on creating admissible heuristics using abstraction; the best of which, created by Helmert et al. [6] is called Merge and Shrink (M&S). The idea behind M&S is to create an abstract version of a problem that can be solved quickly to provide a heuristic for solving the real problem. The abstract problem is created by merging a subset of the state variables (i.e., taking a cross product of their values) and shrinking (removing irrelevant abstract states to keep the size of the abstraction manageable).

The aim of this thesis is to validate whether minimizing P* is correlated with parallel search performance. It will also explore if alternate abstraction methods can be found that better divide the search effort, resulting in faster search times. It will describe in greater detail what has been done in the past and test if new abstraction methods created for M&S are better.

CHAPTER 2
BACKGROUND

This section surveys related prior work, including search for planning problems, early approaches at parallel search, parallel structured duplicate detection, domain independent structured duplicate detection to create abstractions on planning problems, PBNF search, and M&S abstraction-based heuristics.

Search

A central problem AI is solving problems with search. Honavar [8] explains that a problem is reduced to a series of states and a series of operators that transfer one state to another. The set of states is called the state space and the representation of states as nodes and operators with edges is called the state space graph. A search agent's task in AI is to find a path through the state space graph that transfers the original state to a goal state. Below is a figure representing a problem called vacuum world from [10]. The figure 1 displays the states and operators in a state space graph.

Honavar [8] points out that on most non trivial problems a blind exhaustive search is not feasible due to the sheer size of a search space that will grow exponentially. For this reason, much work has focused on algorithms that can search the state space quickly to find the solution. Many of these algorithms use heuristics to take educated guesses as to which direction the search should take to get closer to the solution. One of these algorithms is called A*.

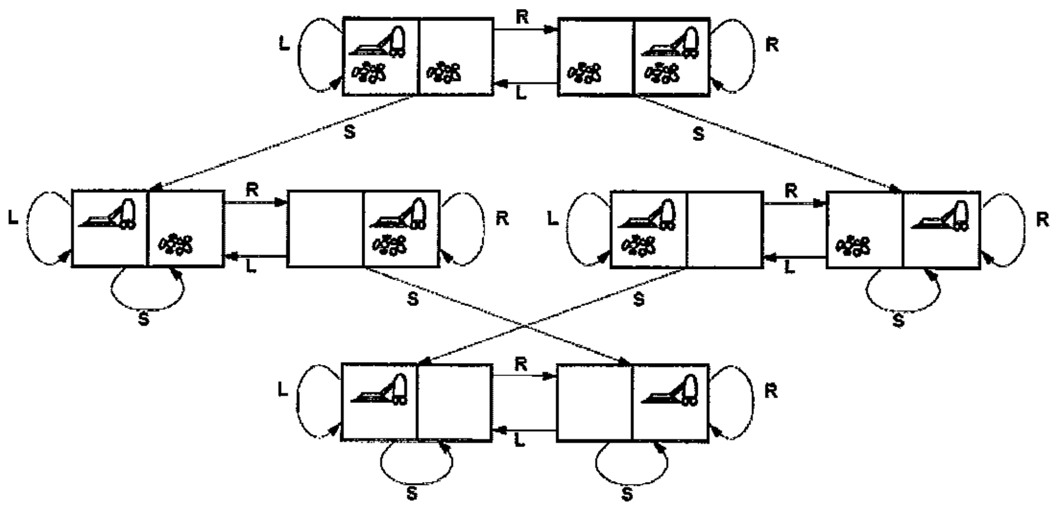


Figure 1: The state space for the vacuum world. Links denote actions: L=left, R=Right, S=Suck. (Reproduced with permission from [10])

Russell and Norvig [10] teach that A* is a best-first search that will find a solution from the initial state to the goal state in a least-cost way. Since the entire state space is not in memory, nodes must be generated as the search is performed. A* generates the nodes that are best first. To accomplish this nodes that have the lowest f value are generated first. The f value is calculated by adding the cost (g) and heuristic (h) values. A* maintains this list (ordered by f value) of nodes to visit in a priority queue that is called the open list. For efficiency a closed list is also maintained so already visited nodes are not expanded again. Figure 2 lists the pseduocode of the A* algorithm.

```
Initialize *queue* to an empty priority queue (min queue)
Initialize *closed* to be an empty set Insert the start state into the *queue*
While (*queue* is not empty)
    *node* ← Dequeue an element off *queue*
    If (*node* is a goal state) //Solution is found
    If (*node* ∉ *closed*)
        Add *node* to *closed*
        Add all successors of *node* to *queue*
```

Figure 2: A search Algorithm. (Reproduced with permission from [10])*

Russell and Norvig [10] also teach that search can find optimal or suboptimal solutions. An optimal or best solution has the lowest cost. A suboptimal approach searches for any solution without guaranteeing optimality. It is typically more difficult to find an optimal solution. A* with admissible heuristics (one that does not over estimate distance to goal) guarantees optimal solutions. The experiments ran for this paper was run to find optimal solutions.

Planning Problems

A large set of problems that need to be solved in AI are called planning problems. One automated planner developed by Richard Fikes and Nils Nilsson in 1971 is called STRIPS (Stanford Research Institute Problem Solver). A STRIPS instance consists of the following, an initial state, a goal state, a set of atoms (variables) and set of operators or actions. Each operator includes preconditions (what must be present before the operator can be performed) and post conditions (what will be present after the operator is applied) [10]. Below is an example STRIPS problem where a kid in location A wants the candy on the counter in location B, but must move a chair in order to reach them. All the problems that were run for this paper are STRIPS problems.

Table 1: A STRIPS example

```
Initial state: At(A), Level(low), ChairAt(C), CandyAt(B)
Goal state:    Have(Candy)
--------------------------------------------------------
Actions:
          // move from X to Y
          _Move(X, Y)_
          Preconditions:  At(X), Level(low)
          Postconditions: not At(X), At(Y)
```

```
// climb up on the chair
_ClimbUp(Location)_
Preconditions:  At(Location), ChairAt(Location), Level(low)
Postconditions: Level(high), not Level(low)

// climb down from the chair
_ClimbDown(Location)_
Preconditions:  At(Location), ChairAt(Location), Level(high)
Postconditions: Level(low), not Level(high)

// move kid and chair from X to Y
_MoveChair(X, Y)_
Preconditions:  At(X), ChairAt(X), Level(low)
Postconditions: ChairAt(Y), not ChairAt(X), At(Y), not At(X)

// take the candy
_TakeCandy(Location)_
Preconditions:  At(Location), CandyAt(Location), Level(high)
Postconditions: Have(candy)
```

Earliest approaches in parallel search

Above was discussed search in general, planning problems, and a serial approach called A*. This section will discuss some early parallel approaches.

There have been many different approaches to parallelizing search. The very first methods used a depth first searching method to parallelize the search. They are called distributed tree search by Ferguson and Korf [5], and parallel window search by Prowley and Korf [9]. They will not be described in this paper.

Burns et al. [2] explain that the simplest approach to parallelizing a best-first search is called Parallel A* (PA*). They explain that PA* has one master open and closed list that are both protected by mutexes. A thread must gain access to the open list through the mutex before it can either insert or check nodes on the open or closed lists. This approach requires excessive data synchronization and because of this it performs worse than serial A*.

Burns et al. [2] teach that another approach is called parallel retracting A* (PRA*). In this approach each thread contains its own open and closed lists that are protected by mutexes. A hash function is used to divide up the search space to each thread. When a node is expanded the successors are added to the appropriate thread's open list by running through the hash function. A thread then communicates with the appropriate destination thread's open list by obtaining a lock and inserting into it. For this method to be successful the hash function has to divide up the search space well. While better then PA* excessive synchronization is still required and therefore it still performs worse than serial A*.

Parallel Structured Duplicate Detection

A better alternative to what has been discussed earlier is parallel structured duplicate detection (PSDD). PSDD's advantage is it avoids the need to lock on every node generation and the need to pass individual nodes between threads. PSDD was originally developed by Zhou and Hansen [12]. It was based on their previous work to limit slow disk I/O operations in search by localizing memory references to abstract states where expansion could happen without having to constantly swap another part of the search graph into memory. This approach was called structured duplicate detection (SDD). PSDD and SDD use an abstraction function that assigns many states in the search space to one state in an abstract space. In their implementation stored nodes of an abstract state is called an *n*block. An nblock *n*' is a successor to another nblock *n* iff (1) *n'* is a successor of *n* and (2) the states *s* and *s'* from the original state space map to *n'* and *n* respectively.

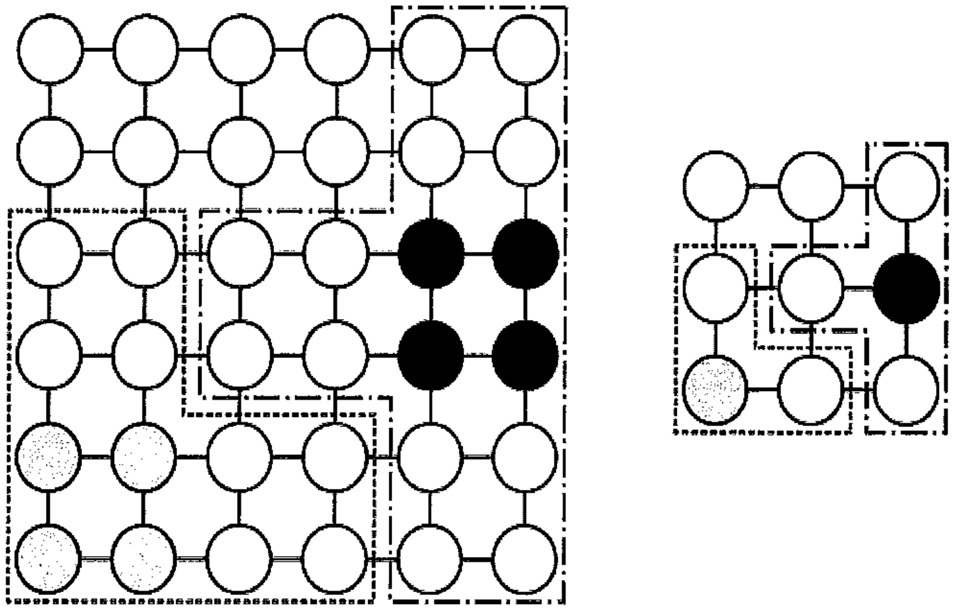


Figure 3: Two disjoint duplicate detection scopes. (Reproduced with permission Burns et al. 2010)

The search is more efficient if duplicate states are not generated and considerable work has been done to find duplicates in search. Zhou and Hansen [12] explain that for efficient duplicate detection in PSDD, each n-block is equipped with its own open and closed lists. Duplicates can easily be found using a concept called "duplicate–detection scope." An *n*block's duplicate detection scope can be defined as any successor to any stored node in the *n*block. Figure 3 shows the duplicate detection scopes in both the state space and the abstract state space. Because the duplicate detection scopes are disjoint (not successors to *n*blocks that are being expanded) two threads can work at the same time and need only check for duplicates in their respective scopes.

Finding *n*blocks that are disjoint can be difficult. Zhou and Hansen [11] explain this is done by giving each *n*block a variable named σ. When an *n*block is taken off the list of available *n*blocks all the σ-values for all the successor *n*blocks are incremented by one. When an *n*block is finished expanding nodes and replaced the successor σ-values are decreased by one. Only *n*blocks with an σ-value of zero can be expanded. This enables expansion to occur without the need to lock the graph and synchronize between threads. Below in figure 4 this process is demonstrated.

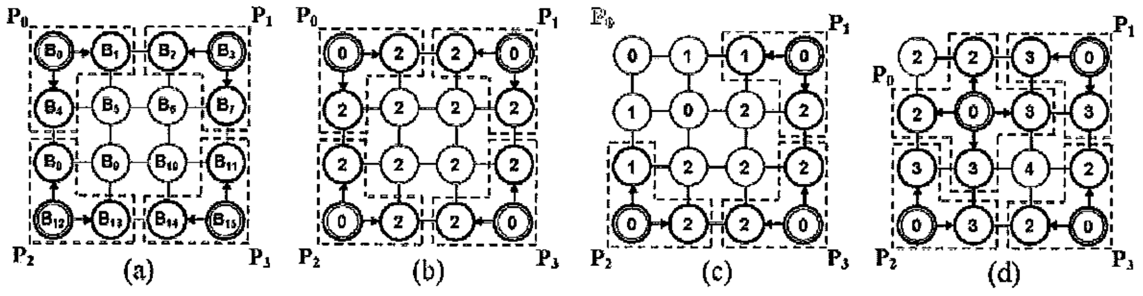


Figure 4: Panel (a) shows that nodes that map to abstract nodes B0, B, B12, and B15 have disjoint duplicate detection scopes, each of which can be assigned to one of the four processors P0 to P3 for parallel node expansions. Panel (b) shows the σ-value of each abstract node for the parallel configuration in (a). Panel (c) shows the σ-value of each abstract node after the release of the duplicate-detection scope occupied by processor P0 in (b). Panel (d) shows a new duplicate-detection scope occupied by P0 and the new σ-values. Abstract nodes filled with gray are those that have already been expanded (Reproduced with permission from [11])

Zhou and Hansen [12] used a breadth-first heuristic search to parallelize SDD. Their solution while it cut down on the synchronization needed it still did not perform better then serial A*.

Domain Independent Structured Duplicate Detection

It is important to be able to create an abstraction on any domain on the fly. In 2006 Zhou and Hansen [13] developed a way to build an abstraction automatically on any STRIPS planning problem that captures the local structure needed for SDD. Locality of an abstract graph is defined as the maximum number of successors of any abstract node compared to the overall number of abstract nodes (see equation 1 below). Zhou and Hansen noticed the smaller this ratio the more effective SDD and therefore PSDD would be. They also noticed this ratio can be reduced by increasing the "resolution" (i.e. adding state variable to the abstraction). But the resolution shouldn't be reduced too much for two reasons. One, the abstract graph must fit in memory, and second, if the resolution is too large then each abstract state will only have a few original state space states assigned to it. If this is the case then *n*blocks will have to be constantly swapped reducing performance. For this reason they placed a bound on the size of the abstract state space (see equation 2 below).

Equation 1 (Zhou and Hansen [11])

$$\delta(\mathrm{P}) = \max_{sp \in Sp} \frac{|Successors(sp)|}{|Sp|}$$

Equation 2 (Zhou and Hansen [11])

$$P^{*} = \arg\min_{P}\{\beta(P) | M \geq |Sp|\}$$

Zhou and Hansen's [11] algorithm works as follows: first state space constraints are exploited. This is done by exploiting XOR constraints. XOR constraints are constraints that specify that only one atom can be true at a time. An example of this from kid problem above is (XOR (ChairAt(Y) ChairAt(X)). This states that the chair is at X or it is at Y but it cannot be at both. They discovered these constraints using an algorithm described by Edelkamp and Helmert [3].

Zhou and Hansen's algorithm uses a greedy approach to minimize Equation (1). Instead of minimizing δ(P) for all possible combinations of XOR groups, the greedy algorithm adds one XOR group a time that minimizes δ(P). They explain that the algorithm first finds XOR group that creates the lowest δ(P). The algorithm then loops through all the other XOR groups choosing the XOR group that when merged with the first minimizes δ(P) . This process is repeated until either the upper bound (M) is met on the size of the abstraction or until all the XOR groups have been used up. It is not common to run out of XOR groups before the upper bound is met [11].

Following is a simple example they used to describe the above algorithm in the logistics domain. The logistics domain's goal is to move packages from one location to another using trucks, and airplanes. Say one has two packages {pkg1, pkg2}, two locations {loc1, loc2}, two airports {airport1, airport2}, two trucks {truck1, truck2}, and one airplane {plane1}. Two of the XOR groups are shown in figure 5 below. An oval is an abstract state and inside the oval is the atom included in that abstract state. An arrow represents an operator that transforms one abstract state to another. Figure 5(a) shows an abstract state graph that is abstracted on the location of pkg1. Figure 5(b) is abstracted on the location of truck1. Figure 5(a) has a localized ratio of 3/7 and figure 5(b) has one of 2/2 which it means it has no locality at all [11].

Zhou and Hansen [11] go on to explain that the resolution can be increased by adding atoms as stated above. One could add the location of pkg2 and since there are also 7 possible positions the number of combinations for the locations of these two packages is 7 x 7 = 49, the size of the new abstract graph. The number of successors of the abstract node increases from 3 to 5. Thus the new localization ratio is 5/49. This process is continued until one runs out of atoms or the bound M is reached. If two different abstractions create the same P* than the one that creates the smallest overall abstraction is chosen [9].

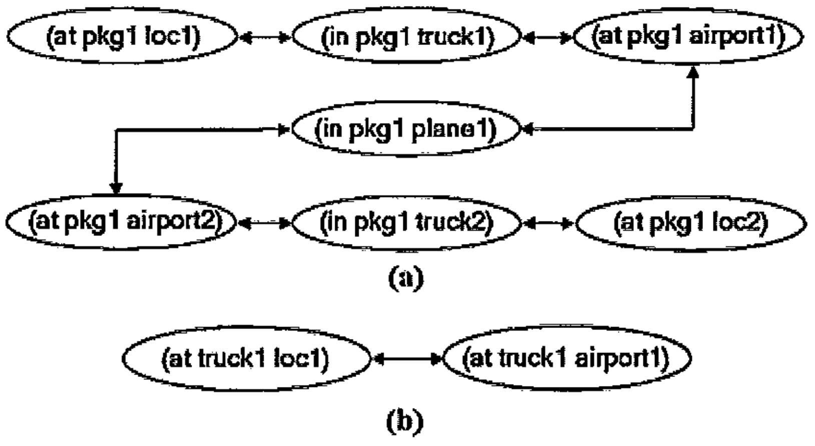


Figure 5: Abstract state-space graph (with self-loops omitted) for logistics. Panel (a) shows an abstract state-space graph based on the location of pkg1. Panel (b) shows another abstract state-space graph based on the location of truck1. Reproduced with permission [9].

PBNF

As stated above PSDD does not perform better then serial A* this is due to threads not being kept busy expanding nodes with the lowest f-value. For this reason

Burns, Lemons, Ruml, and Zhou [2] created Parallel Best-NBlock-First (PBNF). PBNF combines the duplicate detection scope of PSDD and SDD and joins it with an idea from Localized A* (LA*) of Edelkamp and Schrodl [4]. Like LA* PBNF maintains a heap of *n*Blocks ordered by their best f-value. This allows PBNF to approximate an ideal parallel search.

Burns et al. [1] explain that in PBNF threads use the heap of free *n*Blocks to find the best one to expand nodes on. The thread continues to expand nodes as long as they have a better f-value than the next one on the heap. If an acquired *n*Block becomes worse than the next one on the heap it attempts to release the current one and acquire a better one. Since there is no layer synchronization, the first solution found may not be the optimal solution. Search must therefore continue until all open nodes have a worse f-value then the "incumbent" solution.

Since PBNF is not strictly best-first search, for the reasons stated above, Burns et al. [1] implemented some optimizations to decrease overhead. PBNF requires a minimum number of expansions before a new nBlock can be acquired. Also instead of sleeping when a lock is attempted and rejected the thread continues expanding until the lock can be obtained. This cuts down the overhead of having to swap *n*blocks too often and wasting time waiting for *n*blocks to free up. Figure 6 is some pseudo code for PBNF.

PBNF on many domains performs better than A* and sometimes even better when only one thread is used [1].

```
1. while there is an nblock with open nodes
2.     lock; b ← best free nblock; unlock
3.     while b is no worse than the best free nblock or
4.             we've done fewer than m expansions
5.         n ← best open node in b
6.         if f (n) > f (incumbent), prune all open nodes in b
7.         else if n is a goal
8.             if f (n) < f (incumbent)
9.                 lock; incumbent ← n; unlock
10.        else for each child c of n
11.            insert c in the open list of the appropriate nblock
```

Figure 6: A sketch of basic PBNF search, showing locking (Reproduced with permission from [1]).

Merge and Shrink Heuristic

Merge and Shrink (M&S) was developed by Helmert et al. [6] be used as a heuristic in optimal planning in 2007. M&S works by creating an abstraction of the state space and then finding the optimal solution through that abstract state space. Distances in the abstract space are preprocessed and a lookup table is created and stored in memory that can be used during search.

Helmert et al. [6] explain that M&S first creates an atomic abstraction on each variable in the domain. It then merges two of the atomic abstractions into one abstraction. Because the abstract space can become very large as abstractions are merged a shrinking step is also included where the abstract state space is shrunk down to a predetermined size. There are many different strategies to merging abstractions and different strategies on how to shrink an abstract space. Below is a table of different merging strategies that were used in the study.

Table 2: Merge strategies that work well and a description how they work (Helmert et al. [6])

Type	Description
Merge Causal Graph Goal	1. Order atoms to merge from highest variable number to lowest variable number(closet to causal root go first) 2. First, loop through the atoms trying to merge on a causally connected variable 3. Second, if none is found, loop through the atoms and merge on first goal variable (opposite of below)
Merge Goal Causal Graph	1. Order the atoms to merge from the highest variable number to lowest variable number (closet to causal root go first) 2. First, loop through atoms and try to merge on a goal variable 3. Second, if none is found merge on first causally connected variable (opposite of above)

CHAPTER 3
EMPIRICAL RESULTS

This section will describe how the tests were implemented, what was tested and the results. This section will show that abstractions based on P* are not well correlated to faster search times. Some alternative abstractions that perform better will also be presented.

The Fast Downward Planner Integration

The source code of PBNF on planning problems is not available to the public. Therefore, In order to test the different forms of abstraction with PBNF considerable time was spent in implementing PBNF into Helmert's fast downward planner [7]. Now not only is PBNF available to all for planning but in the fast downward planner PBNF has access to a full suite of abstractions, and heuristics.

3.2 PBNF Smallest

Another abstraction method was created for this study name PBNF Smallest. PBNF Smallest attempts to keep the abstraction size as small as possible. This does a couple things: (1) it allows more variables to be merged and (2) it keeps the overall abstraction smaller. Both help to improve search performance, as shown below.

Test Setup and Domains

The different abstraction methods were applied and tested on 82 different search domains. Within each domain eight different problems were selected. The wall clock time and P^* were then recorded. Each test was run five times due to the fact that PBNF is not deterministic and an average was acquired. A time limit of 30 minutes and a memory limit of 31 GBs were also set.

The experiment was performed on many problems in all available domains in Fast Downward planner. This was done to be able to get a clear picture of how the original abstraction for PBNF would work across the board and to find an abstraction method that works better across all domains and problems.

The test was run on a Linux machine with 12 AMD Opteron(tm) 4164 EE processors running at 1800 MHz with 33 GB of ram, four of the processors were used in the experiment.

Each abstraction was kept under 6000 nodes. It was noticed that if one went over 6000 nodes performance degraded. It was also discovered when one went lower performance also suffered. The M&S heuristic was used for the heuristic value used by PBNF.

Results

The first thing learned after many tests is that abstractions created with M&S that had both the merging and shrinking steps were slower than those that were only merged. This is due to the fact that in the shrinking step the abstraction is shrunk by f-value. This is problematic for PBNF if every n-block has only states with the same f-value. This does not allow the work to be efficiently broken up. Nevertheless, it was learned that using the different merging strategies described above minus the shrinking did speed up search time.

Another item that was learned from the many tests that were run is that *P** may not be the best way to build an abstraction. Below in figure 7 is chart containing *P** to processor time over all domains from all the best abstraction methods that were implemented (PBNF Default, PBNF Smallest, Merge CG Goal, and Merge Goal CG).

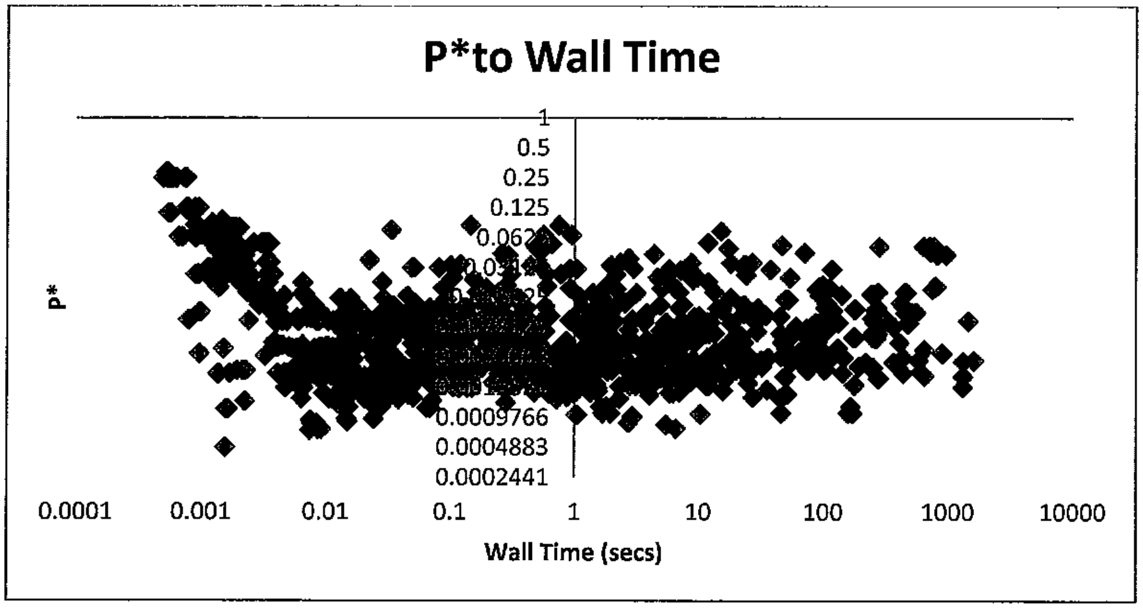


Figure 7: P to processor wall time for all problems all methods the correlation coefficient is -.028.*

As can be seen above other than the very first, there is very little correlation between *P** and the processor time overall. The computed correlation coefficient is -.028 which again indicates very little correlation, and that it is slightly negative.

Although it would be erroneous to conclude that all abstractions created trying to minimize P are less effective. Below in figure 8 are some sample cases where a minimized P* equated to faster search times. The Y axis is P* and the X axis is wall time in seconds on the processors. P* and wall time is shown for each of the four abstraction methods that were used PBNF_DEFAULT (minimizing P*), Goal CG (causally connected variables merged first), Goal CG (goal variables merged first), and Smallest (merge the variable that keeps the overall abstraction the smallest first).

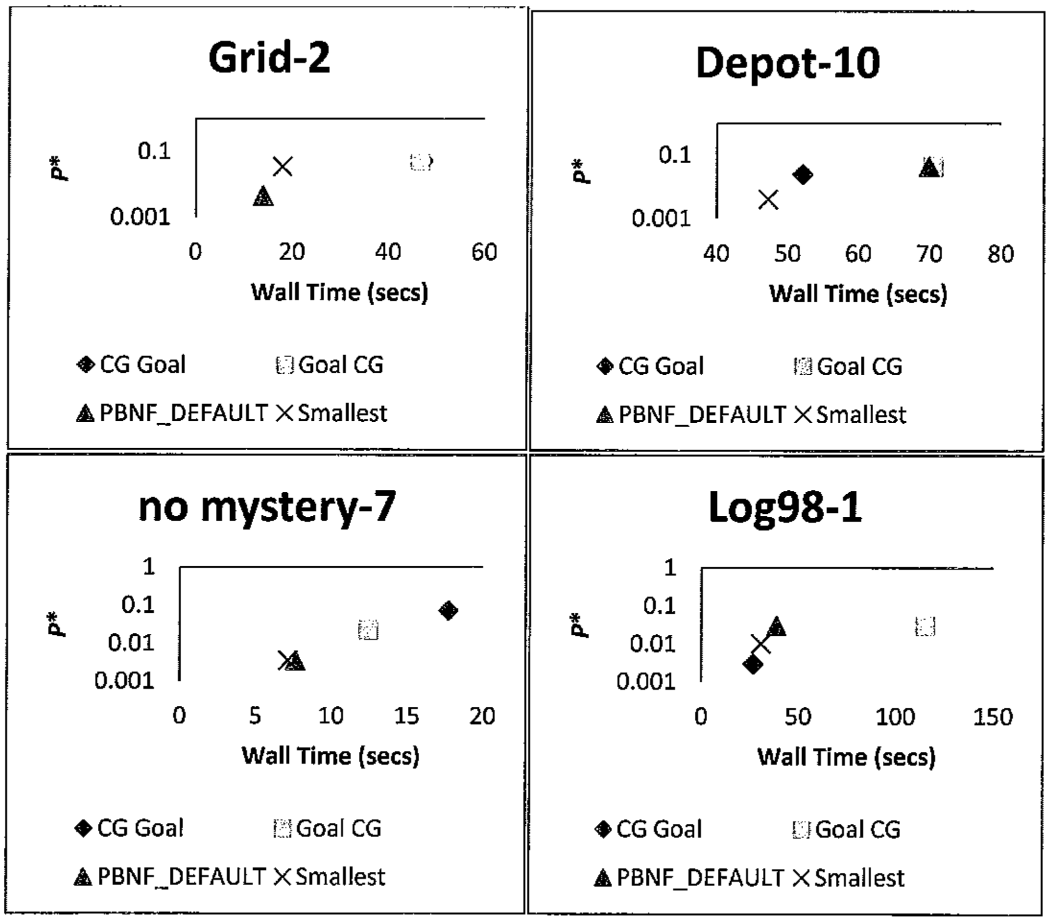


Figure 8: P to wall time in seconds where P* does better in search time. CG Goal, Goal CG, PBNF_DEFAULT and Smallest are the different merging strategies.*

As can be seen from above those with lower P* values perform better. What also can be noticed is that while PBNF Default tries to find abstractions that have the lowest

P* it does not always succeed. This is probably due to the fact that PBNF default uses a greedy approach to save time and therefore does not get the optimal P* value. Another item of note is that how an abstraction is built can have a significant impact on search time. It can also be seen that other abstraction methods can be built that perform better than PBNF Default.

As shown above having a lower P* does not mean a better abstraction, even though in some cases this may be the case. Below in figure 9 are some cases on a problem basis where an abstraction has a lower P* but does worse than other abstractions that have a higher P* values. The Y axis is P* and the X axis is wall time in seconds on the processors. P* and wall time is shown for each of the four abstraction methods that were used PBNF_DEFAULT (minimizing P*), Goal CG (causally connected variables merged first), Goal CG (goal variables merged first), and Smallest (merge the variable that keeps the overall abstraction the smallest first).

In figure 9 the trend is almost the opposite of figure 8. It is nearly the case that a lower P* hurts performance. Again it can be noticed that how an abstraction is put together can significantly affect performance. Finally, in three of the four problems PBNF Default is beat by other methods.

From figures 8 and 9 one can begin to see that while PBNF Default can find good abstractions other methods might perform better on average.

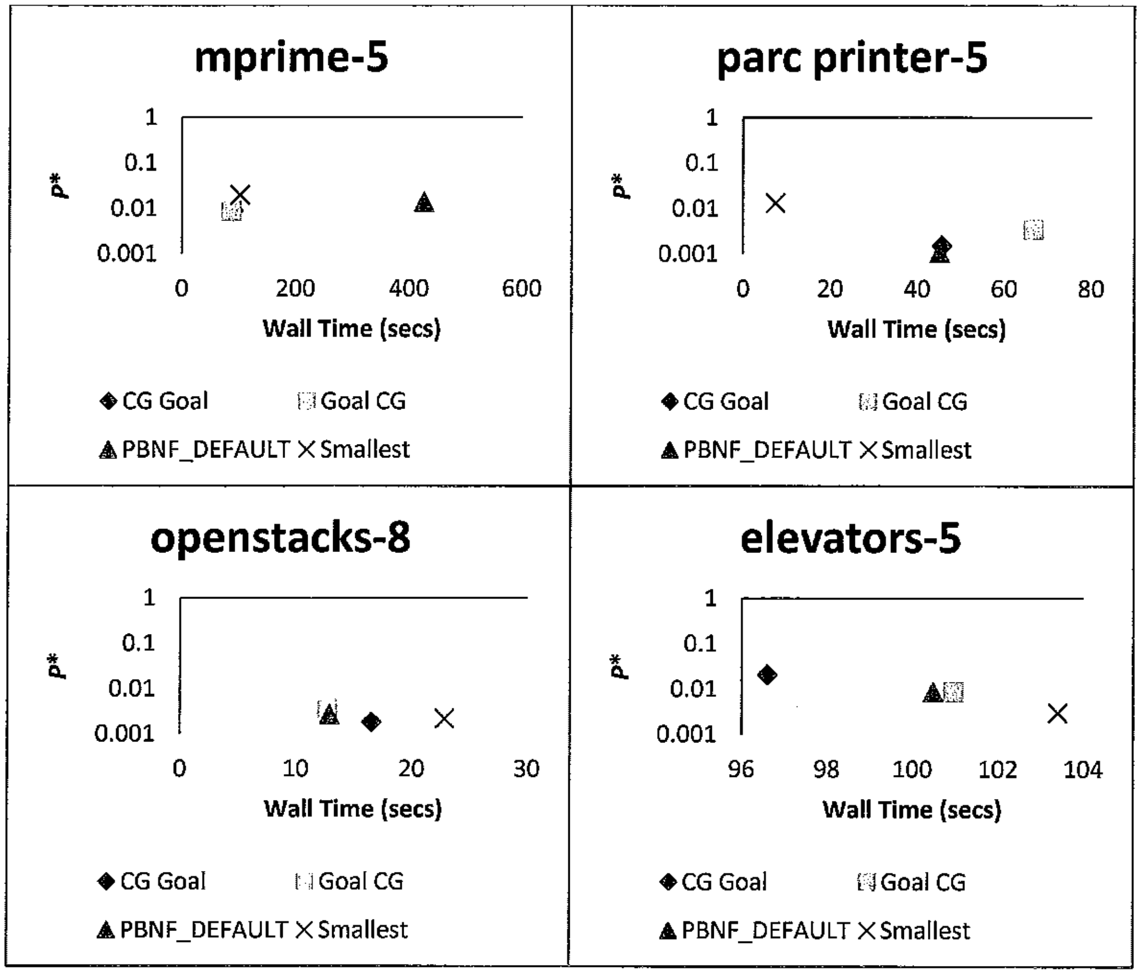


Figure 9: P-value to wall time in seconds where a better P-value does worse in processor time. CG Goal, Goal CG, PBNF_DEFAULT and Smallest are the different merging strategies.

Below in figure 10 is a graph that shows how often each approach won on 533 total tests. A tie is when all approaches are within 10% of each other in search time. The other bars are for each type of abstraction method pbnf (minimizing P*), cg(causally connected variables merged first), goal(goal variables merged first), and smallest (merge the variable that keeps the overall abstraction the smallest first).

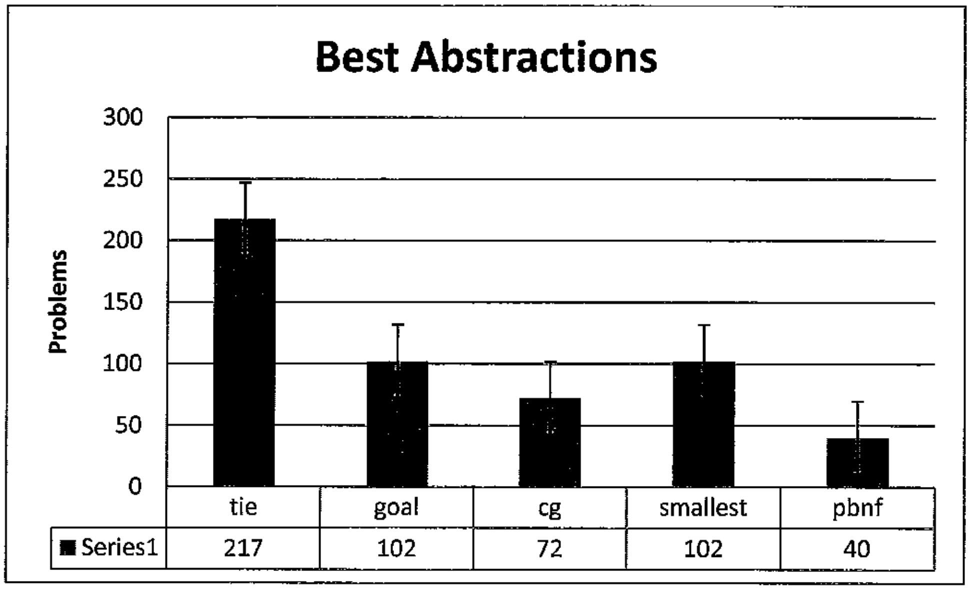


Figure 10: The amount of the time each abstraction approach had the lowest run time in 533 tests. A tie is when all approaches are within 10% of each other.

As can be seen from above in figure 10, most often the above approaches are within 10% of each other in their results. It can also be seen that when there is a difference in results PBNF Default performs the worst with only 40 times being the best choice. The best is Smallest and Goal with 102 time each having the fastest search times.

Below in figure 11 is an attempt to show which approaches have a drastic improvement when they are the fastest. For example Goal CG may win most often but it may be by only a small margin, say 11%. While that approach is better it is not significant. Below in figure 11 one can see approaches that perform significantly better. Only problems are shown that have an improvement of over 50% from the slowest abstraction to the fastest abstraction. The different abstraction methods are as follows: pbnf (minimizing P*), cg(causally connected variables merged first), goal(goal variables merged first), and smallest (merge the variable that keeps the overall abstraction the smallest first).

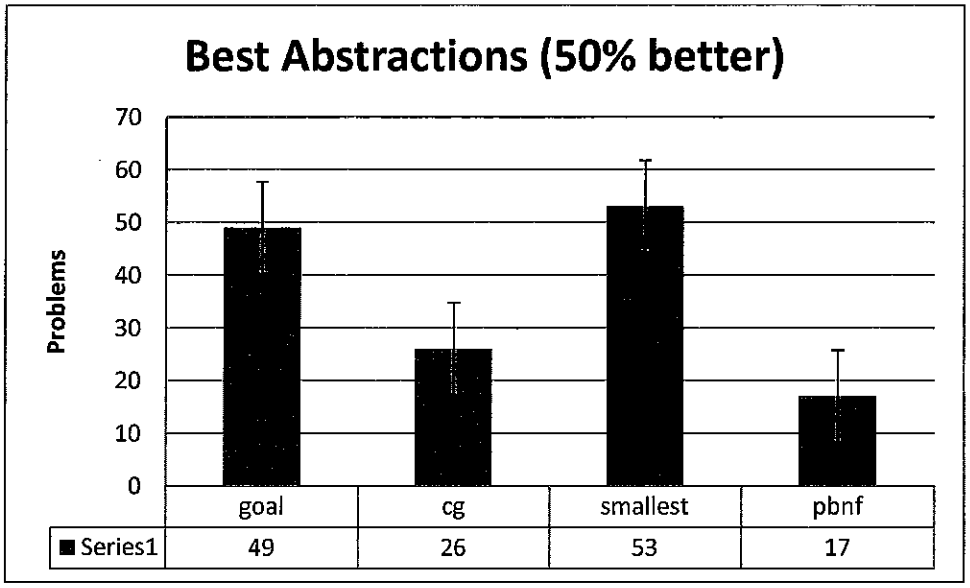
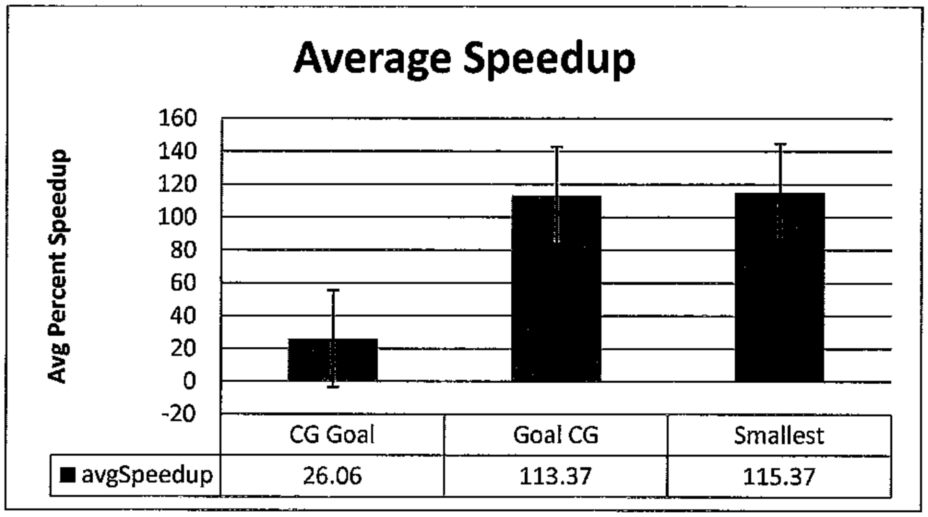


Figure 11: A count of how many times each abstraction was the best choice when the best choice was at least 50% better than the worst choice.

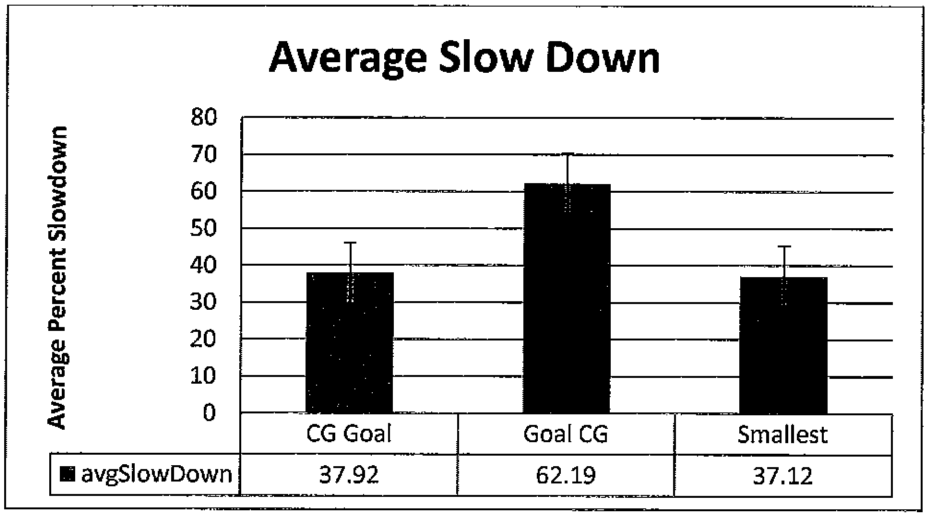
Again one can observe that using the abstraction methods of Goal CG and Smallest produce search times that are significantly faster then PBNF Default.

There is a danger in just using the above merging strategies because one method may be the best option in many cases but it could be be inefficient in other cases. In the below figures (12 & 13) an attempt is made to see on average which result performs the best on all cases. Figure 12 shows the average speedup of one method over PBNF Default and figure 13 shows the average slowdown compared to PBNF Default. Cg Goal merges on causually connected first, Goal CG merges on goal variables first and the Smallest abstraction technique tries to keep the overal abstraction small.



	CG Goal	Goal CG	Smallest
■ avgSpeedup	26.06	113.37	115.37

Figure 12: The average percentage speedup over PBNF default.



	CG Goal	Goal CG	Smallest
■ avgSlowDown	37.92	62.19	37.12

Figure 13: The average slowdown to PBNF default.

As can be seen from above CG Goal and PBNF Smallest are the best methods. Not only do they have on average over a 100% speedup but when they rarely lose it is only around 37% slowdown on average.

In summary creating abstractions according to P* is not correlated to better search time on average. There are some domains and problems where minimizing the P-value produces faster search times. The merge strategies Goal CG and PBNF Smallest on average are the best merge strategies. Goal CG and PBNF Smallest on average are 113% and 115% faster than PBNF Default.

CONCLUSION

Many different approaches can be used to create an abstraction. In many cases the type of abstraction does not have a huge effect on search time. Although on larger search problems the way an abstraction is built can have dramatic effects on the efficiency of the search. Building an abstraction by minimizing the P value can produce good results, although in most cases this is not the best way to build an abstraction. While the overall abstraction built by M&S is a worse abstraction for PBNF, a couple of the merging strategies perform better namely Goal CG and CG Goal. The best method to build an abstraction is to merge variables that are goal variables or to keep the overall abstraction small and the variable count high as displayed by Goal CG and PBNF Smallest, respectively.

REFERENCES

[1] Burns, E., Lemons, S., Ruml, W., and Zhou, R. Parallel Best-First Search: The Role of Abstraction. In *Proceedings of the AAAI-10 Workshop on Abstraction, Reformulation, and Approximation,* Atlanta, GA, July 2010

[2] Burns, E., Lemons, S., Zhou, R., and Ruml, W. Best-first heuristic search for multi-core machines. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 2009.

[3] Edelkamp, S. and Helmert, M. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proceedings of the 5th European Conference on Planning (1999)*, 135–147.

[4] Edelkamp, S. and Schr¨odl, S. Localizing a*. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (2000)*, 885–890.

[5] Ferguson, C. & Korf, R. E. Distributed tree search and its applications to alpha beta pruning. In *Proceedings of the Seventh National Conference on Artificial Intelligence (1988).*

[6] Helmert, M., Haslum, P., and Hoffmann, J. Flexible abstraction heuristics for optimal sequential planning. In Proceedings ICAPS (2007), 176–183.

[7] Helmert M. The fast downward planning system. *J. Artif. Int. Res.* 26, 1 (July 2006), 191-246.

[8] Honavar, V. Artificial Intelligence: An Overview AI Research Laboratory-Department of Computer Science- Iowa State University- Ames, 2006.

[9] Powley, C. & Korf, R. E. 1991. Single-agent parallel window search. *IEEE Transactions Pattern Analysis Machine Intelligence*, *13*(5)(1991), 466–477.

[10] Russell S. and Norvig P. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 1995.

[11] Zhou, R. and Hansen, E. A. 2004. Structured duplicate detection in external-memory graph search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04).*

[12] Zhou, R. and Hansen, E. A. 2007. Parallel structured duplicate detection. In *Proceedings of the Twenty Second Conference on Artificial Intelligence (AAAI-07).*

[13] Zhou, R. and Hansen, E. Domain-independent structured duplicate detection. In *Proceedings. AAAI-06*, 683–688.

APPENDIX

Permission to Use Figures

Ethan,

I am currently writing a thesis that deals with different types of abstractions with PBNF, we have spoke before. Anyway I would like get permission to use some of your figures you used in your papers. The figures are listed below.

Figure 2 from Best-First Heuristic Search for Multicore Machines (Two disjoint duplicate detection scopes)
Figure 1 from Parallel Best-First Search: The Role of Abstraction (A sketch of basic PBNF Search)

If you need more information let me know.

Thanks so much,

Justin Redd

Hi Justin,

I have talked with my coauthors, and everyone agrees that it is fine for you to use these figures as long as you identify the original sources, and as long as you are willing send us a copy of your final thesis.

Best,
Ethan

Dr. Norvig and Dr. Russell,

I am a master's student at Utah State University. I would like to use figure 3.3 in *Artificial Intelligence: A Modern Approach *first edition (The

state space for the vacuum world) in my thesis I am currently writing. I would also like to use figure 3.19 in the 2nd edition (A* search algorithm).

I am assuming email is the best way to get permission, but as I have never done this before (write a thesis), please let me know if there is another route I should take or something I need to do.

Justin Redd

permission granted

Dr. Zhou,

I have spoke to you before about PBNF. I am currently writing my thesis and I would like to have permission to include a few of your figures in my thesis from various of your papers.

- Figure 2 from Parallel Structured Duplicate Detection 2007 (finding open duplicate detection scopes with sigma)
- Figure 2 from Best-First Heuristic Search for Multicore Machines (Two disjoint duplicate detection scopes)
- Equation 1 from Domain-Independent Structed Duplicate detection (best abstraction for SDD)
- Figure 1 from Domain-Independent Structed Duplicate detection (Abstract state-space graphs for logistics)
- Figure 1 from Parallel Best-First Search: The Role of Abstraction (A sketch of basic PBNF Search)

I am assuming email is the best way to get permission to use the figures and I also assume that you can give permission to the above. If there is some other way to gainpermission please let me know, I have never done this before.

Thanks so much,

Justin Redd

Hi Justin,

You can certainly use any figures from papers of which I'm the first author. I also would like to give you permission for the other papers. However, I'd recommend you check with the first author of those papers, just to be sure. In your inquiry, feel free to mention that I have no problem granting you the permission.

Good luck with your thesis,

Rong