Implementing Fast Heuristic Search Code

Ethan Burns and **Matthew Hatem** and **Michael J. Leighton** and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
eaburns, mhatem, mjr58, ruml at *cs.unh.edu*

Abstract

Published papers rarely disclose implementation details. In this paper we show how such details can account for speedups of up to a factor of 28 for different implementations of the same algorithm. We perform an in-depth analysis of the most popular benchmark in heuristic search: the 15-puzzle. We study implementation choices in C++ for both IDA* and A* using the Manhattan distance heuristic. Results suggest that several optimizations deemed critical in folklore provide only small improvements while seemingly innocuous choices can play a large role. These results are important for ensuring that the correct conclusions are drawn from empirical comparisons.

Introduction

Implementing efficient heuristic search code can be quite difficult, especially for a new practitioner who has merely read Russell and Norvig (2010) and is not familiar with all of the tricks of the trade. This paper focuses on these undocumented tricks. Our purpose is not to promote code optimization over algorithmic improvements—we agree with the commonly accepted truism that a better algorithm will outperform a better implementation (McGeoch 2012)—rather, the goals of this paper are to indicate the performance of an efficient tiles solver as a function of CPU performance, demonstrate tricks that are used in state-of-the-art implementations such as Korf's IDA* solver (Korf 1985), to show which of these improvements have the greatest effect, and to increase the overall quality of empirical work in heuristic search by reducing the effect that programmer skill has on the outcome of empirical comparisons.

Many of the results in this paper focus on solving time. We argue that this is a necessary evil for heuristic search, because there is currently no good theory to generate fine-grained predictions of algorithmic performance that are sufficient for comparing different techniques. Traditional complexity analysis provides imprecise bounds on scaling behavior. Performance predictors such as those used in portfolios (Xu et al. 2008) only predict the performance of an implementation and not that of algorithmic techniques. Other methods such as CDP (Zahavi et al. 2010) and BLFS (Rose, Burns, and Ruml 2011) predict the number of nodes expanded, however metrics such as node expansions aren't indicative of search performance since they do not account for computational overhead. For example, all implementations shown in Table 1 expand the same number of nodes, yet, there is over a $14\times$ difference in solving times. Because practitioners usually care about how quickly a problem is solved, we are left comparing algorithms directly on solving time.

The difference between a fast implementation and a slow implementation of the same algorithm can be significant enough to lead to incorrect conclusions from empirical studies. It can even be unfair when two algorithms have equally 'slow' implementations, as speed-ups can affect different algorithms differently. Furthermore, reviewers often demand state-of-the-art results. During a recent reviewer discussion, one of us saw a review that said, "I have no confidence that the authors' algorithm is any faster than Korf's original IDA* code on these problems, and may in fact be a good bit slower. Thus, there are really no relevant positive experimental results in this paper, in my opinion." There was no claim that the results in question were incorrect or that the paper's algorithmic ideas were without merit, just that the authors of the rejected paper were not familiar with all of the tricks that are required to achieve state-of-the-art performance. Currently, it can be difficult to find these tricks without looking at Korf's source code directly, which itself does not indicate which optimizations are important. In this paper, we remedy this problem.

We will focus on the two most popular heuristic search algorithms, IDA* and A*, and the most popular benchmark domain, the 15-puzzle. We aggregate and document the performance impact of many common and uncommon implementation choices in a single paper that can be drawn upon by heuristic search researchers, especially practitioners who are new to the field. Even senior researchers may find some of our results surprising. Full source code is available at www.search-handbook.org/code/socs12.

IDA*

The 15-puzzle (Slocum and Sonneveld 2006) consists of a 4x4 board of 15 numbered tiles and a blank space into which adjacent tiles can be slid. The goal is to take a given configuration and to slide the tiles around until they are in the

goal configuration with the blank in the upper-left corner and the tiles numbered 1–15 in left-to-right, top-to-bottom order. The puzzle has $16!/2 = 10,461,394,944,000$ reachable states, and so it is often infeasible to solve it using a memory-bound search algorithm such as A* (Hart, Nilsson, and Raphael 1968) with the classic Manhattan distance heuristic that sums the horizontal and vertical distance from each tile to its goal location. Iterative-deepening A* (IDA*, Korf 1985) uses an amount of memory that is only linear in the maximum search depth, and was the first algorithm to reliably solve random 15-puzzle instances. IDA* performs a series of f-value-bounded depth-first searches. When one bounded search fails to find a solution, the bound is increased to the minimum f value that was out-of-bounds on the previous iteration and the search begins again.

The Base Implementation

We begin with a base implementation indicative of what may be created by a new heuristic search researcher. Since researchers often want to run experiments on multiple domains using multiple algorithms, our base implementation requires any search domain to implement a simple interface with five virtual methods (which, in C++, are equivalent to singly-indirect function calls): `initial` returns the initial state; `h` returns the heuristic value; `isgoal` returns true if the given state is a goal; `expand` returns a vector (in C++, this is a dynamically sized array) containing information about the successors of the given state; and `release` frees a search state. From the search algorithm's perspective, each state is an opaque pointer that refers to an object for which the internal representation is unknown, allowing the same IDA* code to be used across many different domains.

In the sliding tiles implementation, each state is a record containing a field called *blank* to track the current blank position, a field h with the cost-to-go estimate computed at generation, and an array *tiles* of 16 integers (we tried bytes, however, they decreased performance of IDA* by a factor of 0.9), storing the tile at each board position. Given this representation, the `h`, `isgoal`, and `release` functions are all simple one-liners: `h` returns the h field; `isgoal` return true if $h = 0$; and `release` uses C++'s `delete` keyword. The `initial` function allocates and returns a new state with tile positions specified by the program's input.

Most of the domain's implementation is in the expansion function. Figure 1 shows high-level pseudo-code for `expand`. This pseudo-code almost exactly matches the C++ code; in the real implementation each vector entry contains not only a successor state but also the generating operator represented as the destination blank position and the operator that would re-generate the parent state. This operator information is used to remove simple 2-cycles during search. The constant $Ntiles = 16$ represents the number of entries in the *.tiles* array of each state, and $Width = 4$ is the number of columns in the 4x4 tiles board.

The EXPAND function tests each operator (up, left, right, and down, the same order used by Korf's solver) for applicability (lines 2, 4, 6, and 8), and for each applicable operator a new child is added to the *kids* vector (lines 3, 5, 7, and 9). Each child is generated by creating a new state (line 11), copying the parent's tiles array (line 12), updating the position of the tile that was moved (line 13, note that we don't bother to update $kid.tiles\,[newblank] = 0$), tracking the new blank position (line 14), and computing the Manhattan distance (line 15). Each node generation allocates a new state record because, as mentioned before, the search relies on opaque pointers to allow the same code to work for multiple domains.

```
EXPAND(State s)
 1. vector kids // creates an empty vector
 2. if s.blank ≥ Width then
 3.     kids.push(KID(s, s.blank − Width))
 4. if s.blank mod Width > 0 then
 5.     kids.push(KID(s, s.blank − 1))
 6. if s.blank mod Width < Width − 1 then
 7.     kids.push(KID(s, s.blank + 1))
 8. if s.blank < Ntiles − Width then
 9.     kids.push(KID(s, s.blank + Width))
10. return kids
KID(State s, int newblank)
11. State kid = new State
12. copy s.tiles to kid.tiles
13. kid.tiles [s.blank] = s.tiles [newblank]
14. kid.blank = newblank
15. kid.h = MANHATTAN-DIST(kid.tiles, kid.blank)
16. return kid
```

Figure 1: The base implementation's expand function.

We ran our base implementation of IDA* on all 100 15-puzzle instances from Korf (1985). We used a Dell PowerEdge T710 with dual quad-core Intel Xeon X5550 2.66Ghz processors and 48GB RAM running Ubuntu Linux. All binaries were compiled with GCC version 4.4.3-4ubuntu5.1 using the `-O3` flag. The total search time required to solve all instances was 9,298 seconds.

Incremental Manhattan Distance

One popular optimization for the sliding tiles puzzle is to compute the Manhattan distance value of each state incrementally. Since only one tile moves at a time, the Manhattan distance of a new state can be computed from its parent's by subtracting the Manhattan distance for the moving tile and adding its new Manhattan distance. Our implementation of this optimization is based on the one from Korf's solver: we pre-compute the change in Manhattan distance for every possible single move of each tile and store them in a table before searching. The heuristic computation then becomes a table lookup and an addition (see Figure 3, line 27). With this slight modification (46 additional lines of code, including blank lines and comments), the new time to solve all 100 instances drops down to 5,476 seconds, almost a factor of 2 improvement.

Operator Pre-computation

We reduce expansion overhead even further by pre-computing applicable operators—another optimization used in Korf's solver. Instead of using four if-statements to determine whether the four positions adjacent to the blank are

```
EXPAND(State s)
  17. vector kids // creates an empty vector
  18. for i = 0 to optab [s.blank] .n − 1 do
  19.     kids.push(KID(s, optab [s.blank] .ops [i]))
  20. return kids
```

Figure 2: The new expand function with an operator table.

```
NOPS(State s)
  21. return optab [s.blank] .n
NTHOP(State s, int n)
  22. return optab [s.blank] .ops [n]
APPLY(State s, int newblank)
  23. Undo u = new Undo()
  24. u.h = s.h; u.blank = s.blank
  25. tile = s.tiles [newblank]
  26. s.tiles [s.blank] = tile
  27. s.h + = mdincr [tile] [newblank] [s.blank]
  28. s.blank = newblank
UNDO(State s, Undo u)
  29. s.tiles [s.blank] = s.tiles [u.blank]
  30. s.h = u.h; s.blank = u.blank
  31. delete u
```

Figure 3: Changes for in-place modification.

legal, we can pre-compute the legal successor positions for each of the 16 possible blank locations before searching and store them in an 'operator table.' Figure 2 shows the new expand function. While this optimization only adds about 17 lines of code, it only improves the performance a little. The new time for all 100 instances is reduced to 5,394 seconds, only a 1.015 factor improvement.

In-place Modification

The main source of remaining overhead in the expansion function is the copy performed when generating each new successor (line 12 in Figure 1). Since IDA* only considers a single node for expansion at a time, and because our recursive implementation of depth-first search always backtracks to the parent before considering the next successor, it is possible to avoid copying entirely. We will use an optimization called in-place modification where the search only uses a single state that is modified to generate each successor and reverted upon backtracking. The benefits are threefold. First, there is no need to copy the state. In any domain where the cost of applying an operator plus the subsequent undo to revert back to the parent is cheaper than the cost of copying the state, avoiding the copy will be beneficial. The second benefit is that there is no need to allocate successor states. While memory is not a concern with depth-first search, the cost of allocation can be quite expensive. Finally, because only a single state is used, it is quite likely that this state will remain in the cache for very fast access.

To handle in-place modification we add four methods and one additional record type to our search domain interface. The functions are: `nops` to get the number of applicable operators in a state, `nthop` to get the nth operator, used to eliminate simple 2-cycles, `apply` to apply an operator to a state, and `undo` to revert the application of an operator. The result of applying an operator is an 'undo record' that holds information used by `undo` to revert change. For tiles, this information is the heuristic value and blank position of the parent. Figure 3 shows the modifications required to handle in-place modification. Undo records are treated as opaque pointers by the search algorithm, much as states were previously, and thus they are allocated on the heap. (Allocations could be avoided by keeping a stack of undo records, however, the next optimization will remove the need for any opaque pointers and all allocations). Using in-place modification, the search time required is 2,179 seconds, approximately a 2.5 factor improvement over the previous version.

C++ Templates

The final optimization that we present for IDA* is to replace virtual method calls with C++ templates. C++ templates are a form of meta-programming that allow functions to be parameterized on a given type, or a class (many popular languages provide similar functionality). In C++, template instantiation is similar to a find-and-replace, or a macro expansion that replaces references to our search domain with the methods from the domain class itself. Just as before, we parameterize the search algorithm on the search domain, except instead of using virtual calls, the template instantiates our search algorithm at compile-time with the given parameter class. This means that all virtual method calls are replaced with normal function calls that the compiler will then inline. An additional benefit of using C++ templates for our search algorithm is that the algorithm can refer directly to the search state record defined by the domain, eliminating any need for opaque pointers and all memory allocation. The resulting machine code should look as though we wrote the search algorithm specifically for our sliding tiles solver when in fact it is generalized to any search domain that implements our interface.

Using our template implementation along with the three previous optimizations, all of the 100 tiles instances are solved in 634 seconds. This is a 3.4 factor improvement over the previous implementation and a 14.7 factor improvement over our initial base implementation. In fact, our base implementation required more time to solve each of instances 60, 82, and 88 than our final implementation needed to solve the entire set!

Operator Ordering

The choice of operator ordering can have a significant effect on the performance of IDA*, as a good ordering can effectively eliminate the final (largest) iteration of search and a bad ordering will enumerate the entire iteration. There are only $4! = 24$ possible fixed operator orderings for the sliding tile puzzle, so we solved Korf's puzzle instances with each one using our most optimized solver. Surprisingly, it seems that not all instances are affected by operator ordering to the same degree. For example, operator order has a significant effect on the solving time for instance 82 but has seemingly no effect on instance 88. Also, half of the orderings seem to take about twice as long as the other half. The best ordering on Korf's 100 instances (up, right, left, down)

	secs	imp.	nodes/sec
Base implementation	9,298	–	1,982,479
Incremental heuristic	5,476	1.7×	3,365,735
Oper. pre-computation	5,394	1.7×	3,417,218
In-place modification	2,179	2.3×	8,457,031
C++ templates	**634**	14.7×	**29,074,838**
Korf's solver	666	14.0×	27,637,121

Table 1: IDA* performance on Korf's 100 15 puzzles.

required 628 seconds to solve the entire set. The median ordering (up, down, left, right) needed 728 seconds and the absolute worst ordering (left, down, up, right) needed 942 seconds. Korf's operator ordering was the 2nd best ordering for these instances.

Summary

Table 1 shows a summary of the performance of all of the optimizations that we implemented for IDA*. As a reference, the performance of Korf's implementation is shown as the last row. Korf's solver is written in C and it includes all of the optimizations that were listed here, except for the use of C++ templates which are unnecessary as the solver is domain-specific. Korf's solver is a bit slower than the fastest implementation in this table, this is probably a product of minor differences between the code generated by the C and C++ compilers. The column labelled 'imp.' shows the improvement over the base implementation, e.g., the time required by the base implementation was 2.3 times more than that required by the variant that uses all optimizations up to and including in-place modification. Finally, the column labeled 'nodes/sec' shows the expansion rate for each variant. All implementations in this table expanded a total of 18,433,671,328 nodes summed over all instances, so all improvements can be credited to the implementation, and not a reduction in search effort. With the exception of operator pre-computation, each optimization approximately halves the time required to solve the instance set, with the final implementation being greater than a factor of 14.7 improvement over the base implementation.

Based on Table 1, we would recommend using all of these optimizations in an efficient implementation of IDA* with the exception of operator pre-computation which seems unnecessary. In some cases it may be impossible to implement the template optimization. In these cases, the performance benefits that we gained with templates can instead be realized by sacrificing generality and implementing a domain-specific solver.

Scaling Study

To allow comparison to our timing results, we performed a study of our optimized solver's performance across a variety of different hardware. We compare the results of the *SPECint_base* rating of various machines (from the SPEC CPU 2006 benchmark set, which evaluates the integer processing capabilities of hardware) to the time required for our IDA* implementation to solve all 100 of Korf's 15-puzzles.

Figure 4a shows the scaling behavior across six different machines with SPECint_base numbers of: 15.8, 16.5, 22.2, 29.3, 35.6, and 40.0; the greater the number, the faster the hardware performed on the CPU 2006 benchmark. The circles represent the actual data points and the dashed line shows the line of best fit ($y = -10.8x + 973$). As expected, machines with greater SPECint_base numbers tended to solve our instance set faster; the slowest hardware required 50% more time than the fastest. Using these results, we hope that future researchers can find their machines on the SPEC website (www.spec.org) to estimate how fast an efficient implementation should perform on Korf's instance set.

Case Studies

To validate the results from the previous section, we performed three case studies where independently created search implementations were subjected to the optimizations described above. These case studies further emphasize the importance of using the correct optimizations, as in two cases an incorrect conclusion was drawn from an empirical study using an unoptimized IDA* implementation.

Objective Caml

First, we compare three sliding tile solvers written in the Objective Caml (OCaml) programming language. The first two were existing solvers that were written to operate on many different search domains. The first solver was the least optimized; it only included the incremental Manhattan distance optimization and required 24,186 seconds to solve all of Korf's instances. The second solver included all optimizations listed above except for compile-time templates as they are not available in the language; it required 3,261 seconds to solve all instances, a 7.4 factor improvement on its predecessor. Finally, we wrote a new implementation specifically for tiles solving, i.e., it did not operate on an abstract representation of the search domain. The new solver was able to solve all instances in 852 seconds, a 3.8x improvement over the second implementation, which approximately matches the improvement we saw when switching the C++ implementation to use templates, and a 28× improvement over the first, unoptimized version!

Short Circuit

The next implementation we consider was created to evaluate the state-of-the-art hierarchical search algorithm called Short Circuit (Leighton, Ruml, and Holte 2011). IDA* was already implemented in Short Circuit's C++ code base; it used the same state and goal representation, successor function, and goal test as the Short Circuit algorithm, but none of the optimizations described above were present in either IDA* or Short Circuit. The initial IDA* implementation took 6,162 seconds to solve all of Korf's 100 15-puzzles and Short Circuit was nearly 3 times faster, requiring 2,066 seconds. We then modified both Short Circuit and IDA* taking into account the optimizations described above where applicable. The speedups achieved by the new IDA* over the base implementation were: incremental Manhattan distance computation 2.0×, operator pre-computation only 2.1×, in-place modification 3.6×, and templates only gave a 4.8× speedup. With the exception of the template optimization,

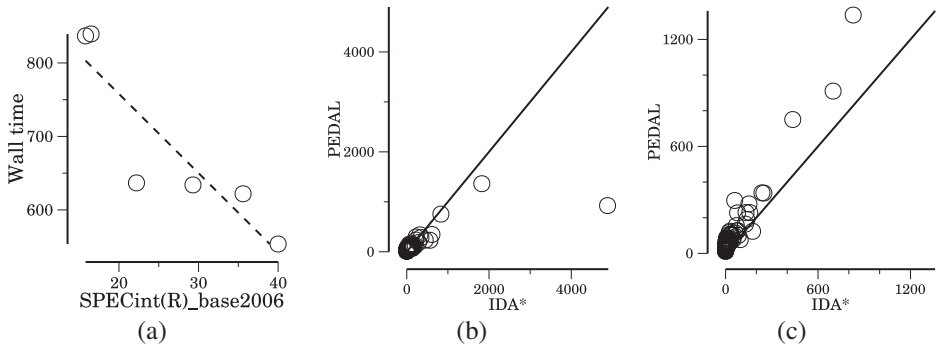

Figure 4: Solving time versus SPECint_base (a), and PEDAL compared to the original (b) and improved (c) IDA*.

which surprisingly didn't boost performance as much as it did for our optimized solver, these numbers tend to agree with the results in Table 1.The new IDA* implementation was able to solve all 100 instances in 1,316 seconds. Only two of the optimizations were applicable to Short Circuit, operator pre-computation and removal of virtual methods, allowing Short Circuit to solve all instances in 1,886 seconds. As we can see, the $3\times$ performance advantage that Short Circuit had over IDA* can be attributed to inefficiencies in the implementation. An IDA* implementation with state-of-the-art optimization techniques is faster than Short Circuit in this domain. Note that the results presented by Leighton, Ruml, and Holte (2011) are still valid, as IDA* on the 15-puzzle with Manhattan distance is expected to outperform Short Circuit. Short Circuit was also shown to work well on other domains better suited for hierarchical search.

PEDAL

Hatem, Burns, and Ruml (2011) introduced a new external-memory search algorithm called Parallel External search with Dynamic A* Layering (PEDAL) While the unit-cost 15 puzzle is not a domain for which PEDAL was specifically designed, PEDAL was shown to compare favorably to IDA* on Korf's 100 instances. During the presentation of our paper at AAAI 2011, an audience member pointed out that Korf's solver was significantly faster than the solver used in the PEDAL comparison. [1]

We have updated both the IDA* and PEDAL implementations using all of the optimizations described above where applicable with the exception of templates, as this would have required a major change to the PEDAL code base. Figure 4b shows the results of the original comparison using the original IDA* operator ordering from the paper, and Figure 4c shows the results with improved implementations of PEDAL and IDA*, using Korf's operator ordering. The axes of these plots show the time required for PEDAL (y axis) and IDA* (x axis) to solve each instance, represented by a circle in the plot. The diagonal line shows $y = x$, and circles above the line are instances where IDA* performed better, whereas circles below represent instances where PEDAL performed better. With these implementations, PEDAL no longer outperforms IDA*, however further optimization of PEDAL's I/O may change the picture again yet again.

[1] This remark motivated the research leading to this paper.

A*

We now turn to the most popular heuristic search algorithm of all, A* (Hart, Nilsson, and Raphael 1968).

The Base Implementation

In our base implementation of A*, the open list is implemented as a binary min heap ordered on f and breaking ties by preferring high g, the closed list is implemented as a hash table using chaining to resolve collisions, and the fields of our state record were changed to bytes to conserve memory. The A* algorithm is modeled after the one described in the 2nd (older) edition of Russell and Norvig (2003) it allows duplicate search states to each have a node on the open list. This implementation required 1,695 seconds to solve all of the 100 instances with the exception of three instances that exhausted a 46 gigabyte memory limit (instances 60, 82, and 88). For reference, our heavily optimized IDA* solver required only 370 seconds to solve these 97 instances. When we include both the incremental Manhattan distance and the operator table optimizations as described for IDA*, solving time only decreases to 1,513 seconds. This is because A* spends much of its time operating on its open list and performing hash table lookups, so these optimizations have little effect.

Detecting Duplicates on Open

One common optimization is to maintain only a single node for each state and to update the nodes location on the open list if it is re-encountered via a cheaper path. This optimization has become so common that it is now used in the A* pseudo-code in the 3rd (the latest as of this writing) edition of Russell and Norvig (2010), and in the Heuristic Search book by Edelkamp and Schrödl (2012).

With this optimization, each search state is represented by a single canonical node that is added to a hash table the first time the state is generated. Each time a new state is generated, the hash table is checked for the canonical node, and if the canonical node is found and the newly generated state was reached via a cheaper path then the canonical node is updated. To update a node, its g value is set to the new lower g value, its f value is adjusted appropriately, its parent node is changed to be the parent that generated the node via the cheaper path, and finally its open list position is updated based on its new f value (note that we don't bother

to check that the duplicate resides on the open list: because the Manhattan distance heuristic is consistent, the duplicate must be on open if a cheaper path to it was found). When generating a state, if the canonical node is not found in the hash table then it is the first time that the state was generated, so it is added to both the open list and hash table, becoming the canonical representation.

Using this optimization, the solving time for the 97 instances that were solvable within the memory limit was 1,968 seconds, which is slower than the base implementation. The reason for this slowdown is likely because this optimization requires a hash table lookup for each node generation. The hash table lookup is relatively expensive compared to the cost of a node expansion for the sliding tiles domain. Additionally, there are very few duplicates in the sliding tiles puzzle and so the extra testing for duplicates on the open list is not very helpful. This is a rather surprising result as this optimization seems to be quite ubiquitous. We suspect that in domains where hash table operations are inexpensive compared to the cost of expanding a node (as is the case in domain-independent planning) and domains where there are many duplicates (as is the case in grid pathfinding) this optimization is much more beneficial. Due to the degradation in performance, this optimization is not used in the following subsections, however, we will revisit it briefly after we speed up our closed list operations.

C++ Templates

One of the most beneficial optimizations for IDA* was the switch from virtual method calls and requiring allocation for opaque pointers to the use of C++ templates. A* can be modified similarly. With IDA*, one of the benefits of this optimization was obviating the need for any memory allocation. In A*, the amount of allocation can be reduced with this optimization as the domain interface doesn't need to return any heap allocated structures, however, A* still must allocate search nodes. Using this optimization, the time to solve the 97 instances drops to 1,273 seconds. In addition, with this optimization, the solver is now able to solve both instance 60 and 82 within the memory available on our machine. The time required to solve all instances except for 88 was 1,960 seconds.

Pool Allocation

A* is a memory-bound algorithm. This means that it will continue to consume more and more memory as it continues to run. Memory allocation can be rather expensive and to alleviate the cost it is often beneficial to allocate memory in a small number of large chunks instead of large number of small chunks. We modified our template-based A* solver to use an allocation pool to allocate nodes. The pool implementation is rather simple: it allocates blocks of 1,024 nodes at a time (other values were tried with little effect). A node pool reduces the number of expensive heap allocations, and subsequently the memory overhead required to track many small heap-allocated nodes, and it has potential to increase cache locality by ensuring that nodes reside in contiguous blocks of memory. Using pool-based allocation, our solver was able to solve *all* 100 instances in 2,375 seconds and it required only 1,184 seconds for the 97 instances solved by the base. So far we have only achieved a 1.4 factor improvement from where we started.

Packed State Representations

The reason that we have yet to see a big improvement is that A* spends much of its time operating on the open and closed lists. Each closed list operation requires hashing a 16-entry array and possibly performing equality tests on this array to resolve hash collisions. Our open list is implemented as a binary heap which has $\mathcal{O}(\log_2 n)$ performance for insert, remove, and update operations. This optimization and the following ones attempt to optimize the access to data structures. The first such optimization is targeted at both reducing memory usage and the cost of hash table operations.

For the 15-puzzle, the tiles are numbered 1–15, and thus each tile can be represented in 4 bits. Our current implementation uses a byte for each tile. We can half the memory requirement for each tile state by packing the tiles into 8 bytes. On the machine used in our experiments this is the size of a single word. As a consequence, our search states will take up half of the amount of memory, our hash function can simply return the packed state representation, and equality tests performed by our hash table can be simplified to equality test between two numbers containing our packed state representation. The resulting solver was able to solve all 100 of Korf's 15-puzzle instances in 1,896 seconds, and it only needed 1,051 seconds to solve the 97 instances solved by the base.

Intrusive Data Structures

Another way to optimize the hash table is to make it an 'intrusive' hash table. Our hash table resolves collisions via chaining, and entries added to the hash table are wrapped in a record with two pointers: one to the entry and one to the next element in the chain, forming a linked list. Instead of allocating a record for each hash table entry, we can 'intrude' on the entries being stored in the table by requiring them to provide a next pointer for the hash table's use. This reduces the memory requirement of A* by at least 16 bytes per node on a 64 bit machine, and it also reduces the number of allocations. By using an intrusive hash table, our solving time for all 100 instances was 1,269 seconds and only 709 seconds for the 97 instances solved by the base implementation. Using all of these optimizations, we finally have achieved a $2.4\times$ performance improvement over the base.

Since the last two optimizations targeted the performance of the hash table, it raises the question as to whether or not removing duplicate nodes from the open list would now be beneficial. Previously, this optimization was disregarded because the closed list operations were too costly to realize a benefit. Unfortunately, removing duplicate nodes from the open list is still detrimental to performance even with the reduction in the cost of accessing the hash table. Using all of these optimizations, the solving time when duplicate nodes are removed from the open list increases to 1,944 seconds for the set of 100 instances and 1,049 seconds for the 97 instances solved by the base solver. We conclude that it is not

worth avoiding duplicate nodes on the open list in a sliding tile puzzle solver.

Array-based Priority Queues

Now, we will take advantage of the fact that the sliding tiles domain has only unit-cost edges. This allows us to replace the binary heap used for our open list with a more efficient data structure: a 1-level bucket priority queue (Dial 1969). A 1-level bucket priority queue is simply an array indexed by f values, and each entry in the array is a list of the nodes on the priority queue with the corresponding f value. In the 15-puzzle, there are only a small number of possible f values, and each of these possible values is a small integer. An 1-level bucket priority queue allows constant-time insert, remove, and update operations in our open list.

Insertion into the priority queue simply adds the inserted node to the front of the list at the array index corresponding to the node's f value. Since there is a fixed number of possible f values, lookup can be performed by quickly finding the smallest array index with a non-empty list. This operation can be sped up slightly by keeping track of a conservative estimate of the minimum array index on each insertion operation (see Edelkamp and Schrödl (2012) for details). This is a big improvement over the $\mathcal{O}(\log_2 n)$ operations a binary heap, especially considering the fact that the open list can contain hundreds of millions of nodes and it is accessed in a very tight loop. Using a 1-level bucked priority queue, our A* solver was able to solve all 100 instances in 727 seconds and the 97 instances solved by the base implementation in only 450 seconds. At this point, our A* solver is nearly on par with our IDA* solver in terms of time.

One thing that is lost with the 1-level bucket priority queue is good tie breaking. Recall that our binary heap ordered nodes by increasing f values, breaking ties by preferring nodes with higher g. The reasoning is that the goal can be found more quickly in the final f layer of search. The 1-level bucket priority queue achieves constant time insertions and removals by appending to and removing from a list, leaving the list itself unsorted; ties are broken in last-in-first-out order. Our final optimization uses a nested variant of the 1-level bucket priority queue. In essence, this new structure is a 1-level bucket priority queue ordered on f where each entry is in fact another 1-level bucket priority queue ordered on g (this is slightly different from the 2-level bucket priority queue described by Edelkamp and Schrödl 2012, as our goal is to introduce tie-breaking and the 2-level queue is to reduce the size of the structure in the face of many distinct f values). Since insertion and removal are constant time for both the outer and inner queues, they are also constant time for the 2-dimensional priority queue. Using this final optimization, our solver was able to solve all 100 instances in 516 seconds, and the 97 instances solved by the base only needed 290 seconds, giving a $5.8\times$ improvement over the base implementation.

Summary

Table 2 shows a summary of the results for all optimizations on the 97 instances solved by the base implementation. We have included a column showing the maximum amount of memory that A* required to solve these instances, given in gigabytes. Overall, the final implementation was $5.8\times$ faster than the base implementation and required 40% of the memory to solve all 97 instances. All variants expanded 922,237,451 nodes on this set with the exception of the two variants that disallowed duplicates on the open list which both expanded 917,203,704 nodes total and the 1-level bucket priority and its nested variant which expanded 1,572,478,563 and 922,124,752 nodes respectively. Except for the 1-level bucket open list, all of these variants expanded a similar number of nodes and thus the performance improvement is not from reducing the search effort, but rather from reducing the amount of per-node computation.

	secs	imp.	GB	imp.
Base implementation	1,695	–	28	–
Incr. MD and oper. table	1,513	$1.1\times$	28	$1.0\times$
Avoiding dups on open	1,968	$0.9\times$	28	$1.0\times$
C++ templates	1,273	$1.3\times$	23	$1.2\times$
Pool allocation	1,184	$1.4\times$	20	$1.4\times$
Packed states	1,051	$1.6\times$	18	$1.6\times$
Intrusive data structures	709	$2.4\times$	15	$1.9\times$
Avoiding dups on open (2)	1,049	$1.6\times$	14	$2.0\times$
1-level bucket open list	450	$3.8\times$	21	$1.3\times$
Nested bucket open list	**290**	$5.8\times$	**11**	$2.5\times$

Table 2: A* performance on Korf's 100 15 puzzles, excluding instances 60, 82, and 88.

	secs	GB	nodes/sec
Pool allocation	2,375	45	656,954
Packed states	1,896	38	822,907
Intrusive data structures	1,269	29	1,229,574
Avoiding dups on open (2)	1,944	28	798,349
1-level bucket open list	727	36	**3,293,048**
Nested bucket open list	**516**	**27**	3,016,135

Table 3: A* performance on Korf's 100 15 puzzles.

Given these results, the nested bucket-based priority queue appears to be the most important optimization for A* on this domain. The bucket priority queues performed so much better than a binary heap that it seems unfair to handicap A* with a binary heap on domains that have unit-cost actions. In addition, a reasonable implementation should also use incremental Manhattan distance, pooled node allocation, and packed states as they should be applicable in any implementation, they all increase performance and decrease memory usage. C++-style templates are recommended if they are available. Intrusive data structures are mildly beneficial, however, they can be difficult to reason about and increase the complexity of the implementation, thus we would deem them unnecessary in the general case.

Finally, since six different settings were able to solve all 100 instances, we have included results for these implementations on all of the instances in Table 3. This table also includes a nodes per second column for comparison with the IDA* results in Table 1. A* with the nested bucket-based priority was faster than IDA*, however it required about 27 gigabytes compared to IDA*'s 12 megabytes. Additionally, A*'s expansion rate was significantly less than that of IDA*; the fastest A* implementation expanded nodes ap-

proximately $9\times$ slower than the fastest IDA* implementation. The reason that A* was faster overall was because IDA* expands roughly $12\times$ more nodes in total.

Discussion

As discussed above, we have verified some of our results using an IDA* implementation written in Objective Caml. OCaml is a high-level, garbage collected, programming language and thus we would expect that these results would also carry over to other languages such as Java. (Optimized Java code is available at the URL given earlier.) Future work in this area may find it interesting to look at the impact of garbage collection on search performance. We speculate that algorithms like A*, which allocate a lot of nodes that remain in memory for the life of the search, perform poorly with garbage collectors that are tuned for allocating many short-lived objects (Chailloux, Manoury, and Pagano 2000).

The expansion rates of our IDA* solver show that node expansion in the sliding tile puzzle is very cheap, especially when compared to other uses of heuristic search, such as domain-independent planning where expansion rates on the order of tens of thousands of nodes per second are common (Zhou and Hansen 2011). This raises the question of the usefulness of the sliding tile puzzle as a popular proving ground for new search technology. Its expansion rate renders useless many techniques for reducing the number of node expansions at the expense of even the slightest overhead. Techniques such as ordering successors and eliminating cycles by keeping the current search trajectory in a hash table are costly enough that we didn't even consider them for our experiments. In other domains with a greater cost for node expansion or heuristic computation, such as the Sokoban (Junghanns and Schaeffer 2001), these techniques may be crucial for achieving high performance.

Given that optimizations can account for over an order of magnitude performance difference between implementations of the same algorithm, it is important to consider whether the type of competitive testing used in most papers on heuristic search is the right approach. Hooker (1995) argues against the use of such testing, claiming that it is nearly unscientific. Instead, Hooker says we should conduct experiments that tell why algorithms behave the way that they do, instead of merely determining which is faster. Ruml (2010) also argues against the idea of state-of-the-art performance, in favor of a deeper understanding of algorithms. The common theme is to relieve scientists from the burden of tedious code optimization, freeing time for the actual study of algorithms. We view this paper as aligned with these ideas: by accumulating a variety of well-known and documented optimizations, it will be easier to produce tuned implementations and researchers can focus on algorithm design instead. In addition, this will lower the barrier of entry to the the heuristic search community for new researchers.

Although we see work on the 15-puzzle as a necessary first step, the next work in this direction should explore performance on the 24-puzzle with the latest PDB heuristics.

Conclusions

We argue that a state-of-the-art implementation is necessary to ensure the accuracy of empirical results in algorithmic comparisons. To this end, we have presented a set of optimizations for the two most popular heuristic search algorithms, IDA* and A*, on the most popular heuristic search benchmark, the 15-puzzle. Given our results, we have recommended optimizations that we deem crucial for any serious implementation. To aid other researchers in determining whether or not their implementation is on par with the performance that reviewers expect, we have presented a study showing how we expect performance to scale with different hardware. We hope that the results presented in this paper will help to increase the quality of empirical work in heuristic search by helping researchers draw the correct conclusions when comparing search algorithms.

Acknowledgements

We acknowledge support from NSF (grant IIS-0812141) and the DARPA CSSG program (grant HR0011-09-1-0021).

References

Chailloux, E.; Manoury, P.; and Pagano, B. 2000. *Développement d'applications avec Objective CAML*. O'Reily. An English translation is available at http://caml.inria.fr/pub/docs/oreilly-book/.

Dial, R. 1969. Shortest-path forest with topological ordering. *CACM* 12(11):632–633.

Edelkamp, S., and Schrödl, S. 2012. *Heuristic Search: Theory and Applications*. Elsevier.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Hatem, M.; Burns, E.; and Ruml, W. 2011. Heuristic search for large problems with real costs. In *Proceedings of AAAI-2011*.

Hooker, J. N. 1995. Testing heuristics: We have it all wrong. *Journal of Heuristics* 1:33–42.

Junghanns, A., and Schaeffer, J. 2001. Sokoban: Enhancing general single-agent search methods using domain knowledge. *AIJ* 129:219–251.

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings IJCAI-85*, 1034–1036.

Leighton, M.; Ruml, W.; and Holte, R. 2011. Faster optimal and suboptimal hierarchical search. In *Proceedings SoCS-11*.

McGeoch, C. C. 2012. *A Guide to Experimental Algorithmics*. Cambridge University Press.

Rose, K.; Burns, E.; and Ruml, W. 2011. Best-first search for bounded-depth trees. In *Proceedings of SoCS-11*.

Ruml, W. 2010. The logic of benchmarking: A case against state-of-the-art performance. In *Proceedings of SoCS-10*.

Russell, S., and Norvig, P. 2nd edition 2003, 3rd edition 2010. *Artificial Intelligence: A Modern Approach*.

Slocum, J., and Sonneveld, D. 2006. *The 15 puzzle book: how it drove the world crazy*. Slocum Puzzle Foundation.

Xu, L.; Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2008. Satzilla: portfolio-based algorithm selection for sat. *JAIR* 32(1):565–606.

Zahavi, U.; Felner, A.; Burch, N.; and Holte, R. C. 2010. Predicting the performance of IDA* using conditional distributions. *JAIR* 37:41–83.

Zhou, R., and Hansen, E. 2011. Dynamic state-space partitioning in external-memory graph search. In *Proceedings of ICAPS-11*.