Faster problem solving in Java with heuristic search

Learn about a Java implementation of a popular search algorithm for artificial intelligence

Matthew Hatem (mhatem@us.ibm.com)
Senior Software Engineer
IBM

Ethan Burns (burns.ethan@gmail.com)
Software Engineer
Google

Wheeler Ruml (ruml@cs.unh.edu)
Associate Professor
University of New Hampshire

16 July 2013

Learn about the field of heuristic search and its application to artificial intelligence. This article's authors show how they arrived at a successful Java™ implementation of the most widely used heuristic search algorithm. Their solution exploits an alternative framework to Java Collections and uses best practices for avoiding excessive garbage collection.

Solving problems by searching through a space of possible solutions is a fundamental technique in artificial intelligence called *state space search*. *Heuristic search* is a form of state space search that exploits knowledge about a problem to find solutions more efficiently. Heuristic search has enjoyed much success in a variety of domains. In this article, we introduce you to the field of heuristic search and present an implementation of A* — the most widely used heuristic search algorithm — in the Java programming language. Heuristic search algorithms pose high demands on computing resources and memory. We also show how we improved our Java implementation by avoiding expensive garbage collection and exploiting a high-performance alternative to the Java Collections Framework (JCF). All the code for the article is available for download.

Heuristic search

Many problems in computer science can be represented by a graph data structure, with paths in the graph representing potential solutions. Finding the optimal solution requires finding a shortest path. For example, imagine an autonomous video game character. Each move the character can make corresponds to an edge in the graph, and the character's objective is to find the shortest path to engage an opposing character.

Algorithms such as *depth-first* search and *breadth-first* search are popular graph-traversal algorithms. But they are considered *uninformed*, and they are often severely limited by the size of the problem they can solve. Moreover, depth-first search is not guaranteed to find an optimal solution (or any solution at all in some cases), and breadth-first search is guaranteed to find an optimal solution only in special cases. Heuristic search, in contrast, is an *informed* search that exploits knowledge about a problem, encoded in a *heuristic*, to solve the problem more efficiently. Heuristic search can solve many difficult problems that uninformed algorithms cannot.

Video game pathfinding is a popular domain for heuristic search, but it can also solve more-complex problems. The winner of the 2007 DARPA Urban Challenge self-driving car race used heuristic search to plan drivable routes that are both smooth and direct (see Resources). Heuristic search has also seen success in natural language processing, where it's used for syntactic parsing of text and stack decoding in speech recognition. It has also been applied in robotics and in the field of bioinformatics. Multiple Sequence Alignment (MSA), a well-studied informatics problem, can be solved faster using less memory with heuristic search, compared to the classic dynamic programming approach.

Heuristic search with Java

The Java programming language hasn't been a popular choice for implementing heuristic search because of its high demands for memory and computing resources. C/C++ is often preferred for performance reasons. We demonstrate that Java is a suitable programming language for implementing heuristic search. We start by showing that a textbook implementation of A* is indeed slow and exhausts available memory when solving a popular benchmark problem set. We address these performance issues by revisiting some critical implementation details and leveraging an alternative to the JCF.

Much of this work is an extension of work published in an academic paper co-written by this article's authors (see Resources). While the original work concentrates on C/C++ programming, here we show that many of the same ideas apply in Java.

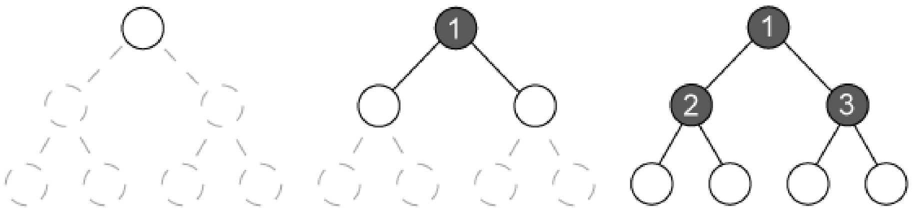
Breadth-first search

Familiarity with an implementation of breadth-first search — a simpler algorithm that shares many of the same concepts and terminology — will help you understand the details of implementing heuristic search. We'll take an *agent-centric* view of breadth-first search. In an agent-centric view, an agent is said to be in some *state* and has available to it a set of applicable *actions* that can be taken from that state. Applying an action transitions the agent from its current state to a new *successor* state. This view generalizes well to many types of problems.

The goal of a breadth-first search is to devise a series of actions that lead the agent from its initial state to a goal state. Starting from the initial state, breadth-first search proceeds by visiting the most recently generated states first. All of the applicable actions are applied at each visited state, generating new states that are then added to the list of unvisited states (also referred to as the search's *frontier*). The process of visiting a state and generating all of its successor states is commonly referred to as *expanding* the state.

You can think of this search procedure as generating a tree: The tree's root node represents the initial state, and child nodes are connected by edges that represent the actions that were used to generate them. Figure 1 shows an illustration of this search tree. The white circles represent nodes in the frontier of the search. The grey circles represent the nodes that have been expanded.

Figure 1. Breadth-first search order on a binary tree



Every node in the search tree represents some state, but two unique nodes can represent the same state. For example, a node at a different depth in the search tree can have the same state associated with it as another node higher in the tree. These *duplicate* nodes represent two different ways of achieving the same state in the search problem. Duplicates can be problematic, so all of the visited nodes must be remembered.

Listing 1 shows the pseudocode for breadth-first search:

Listing 1. Pseudocode for breadth-first search

```
function: BREADTH-FIRST-SEARCH(initial)
open ← {initial}
closed ← 0
loop do:
     if EMPTY(open) then return failure
     node ← SHALLOWEST(open)
     closed ← ADD(closed, node)
     for each action in ACTIONS(node)
          successor ← APPLY(action, node)
          if successor in closed then continue
          if GOAL(successor) then return SOLUTION(node)
          open ← INSERT(open, successor)
```

In Listing 1, we keep the frontier of the search in a list we call the *open list* (Line 2). The nodes that have been visited are kept in a list we call the *closed list* (Line 3). The closed list helps to ensure that we do not duplicate search efforts by revisiting any node more than once. A node is added to the frontier only if it is not already in the closed list. The search loop continues until the open list is empty or until a goal is found.

You might have noticed in Figure 1 that breadth-first search proceeds by visiting all nodes at each depth layer of the search tree before moving to the next layer. For problems in which all of the actions have the same cost, all edges in the search tree have the same weight, and in this case breadth-first search is guaranteed to find an optimal solution. That is, the first goal that is generated is along the shortest path from the initial state.

In certain domains, each action has a different cost. For these domains, the edges in the search tree have nonuniform weights. The cost of a solution in this case is the sum of all edge weights along the path from the root to the goal. For these domains, breadth-first search is not guaranteed to find an optimal solution. Moreover, breadth-first search must expand all nodes at every depth layer of the tree until the goal is generated. The memory required to store these depth layers can quickly exceed the available memory on most modern computers. This limits breadth-first search to a narrow set of small problems.

Dijkstra's algorithm is an extension of breadth-first search that orders the nodes on the search frontier (sorts the open list) by the cost to reach the node from the initial state. It is guaranteed to find the optimal solution in the case of uniform or nonuniform action costs (assuming costs are nonnegative). However, it must visit all nodes whose cost is less than the optimal solution, and thus it is limited to smaller problems. The next section describes an algorithm that is capable of solving large problems by significantly reducing the number of nodes that must be visited to find an optimal solution.

The A* search algorithm

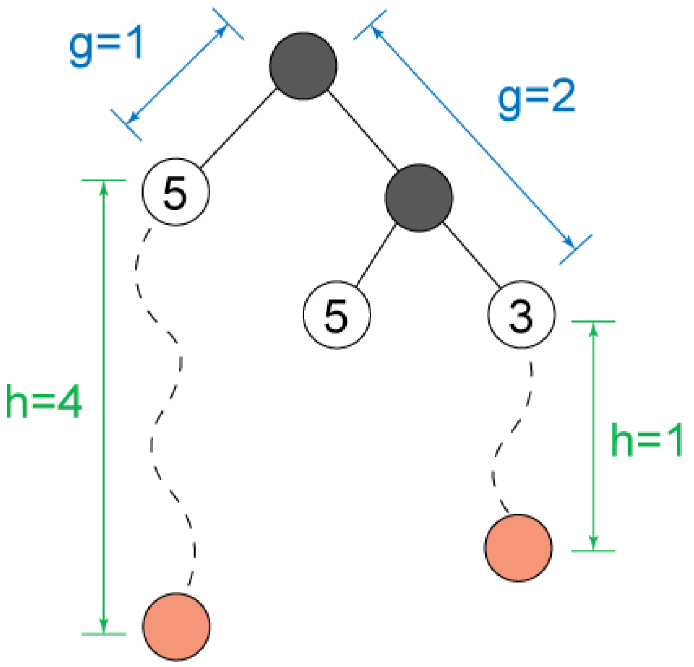
The A* algorithm, or some variation of it, is one of the most widely used heuristic search algorithms. A* can be viewed as an extension of Dijkstra's algorithm that exploits knowledge about a problem to reduce the number of computations necessary to find a solution while still guaranteeing optimal solutions. The A* and Dijkstra's algorithms are classic examples of *best-first* graph-traversal algorithms. They are best-first because they visit the best-looking nodes — the nodes that appear to be on a shortest path to a goal — first until a solution is found. For many problems, it is critical to find optimal solutions, and this is what makes algorithms like A* so important.

What separates A* from other graph-traversal algorithms is its use of heuristics. A heuristic is a bit of knowledge — a rule of thumb — about a problem that allows you to make better decisions. In the context of search algorithms, *heuristic* has a specific meaning: a function that estimates the cost remaining to reach a goal from a particular node. A* can take advantage of heuristics to avoid unnecessary computation by deciding which nodes appear to be the most promising to visit. A* tries to avoid visiting nodes in the graph that do not appear to lead to an optimal solution and can often find solutions quickly and with less memory than less-informed algorithms.

The way A* determines which nodes appear to be the most promising is by computing a value — we refer to it as the *f value*— for each node and sorting the open list by this value. The f value is computed using two other values, the *g value* and *h value* of a node. A node's g value is the total cost of all actions required to reach a node from the initial state. The estimated cost to reach the goal from a node is its h value. This estimate is the heuristic in heuristic search. The nodes with the lowest f values are the ones that appear to be the most promising nodes to visit.

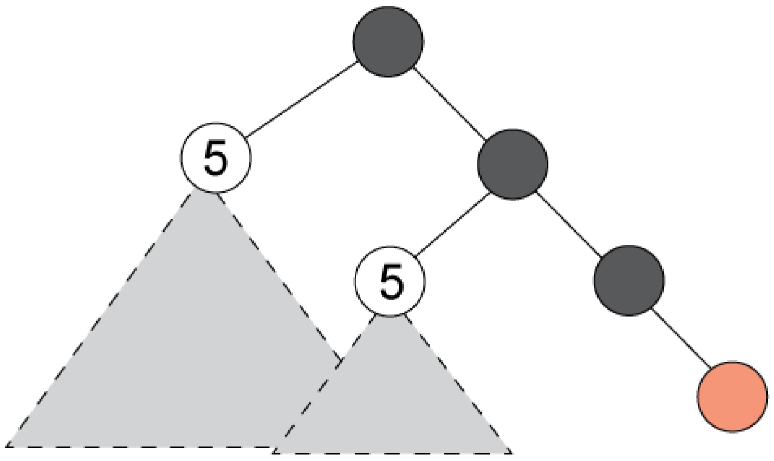
Figure 2 illustrates this search procedure:

Figure 2. A* search order on f values



In the example in Figure 2, the frontier has three nodes. Two have an f value of 5 and one has an f value of 3. The node with the smallest f value is expanded next and leads immediately to a goal. This spares A* from having to visit any of the subtrees under the other two nodes, as illustrated in Figure 3. This makes A* much more efficient than algorithms like breadth-first search.

Figure 3. Unnecessary for A* to visit the subtrees under nodes with higher f values



If the heuristic that A* uses is *admissible*, then A* visits only the nodes necessary to find an optimal solution. A* is popular for this reason. No other algorithm guarantees an optimal solution by visiting fewer nodes than A* with an admissible heuristic. For the heuristic estimate to be admissible, it must be a lower bound: a value less than or equal to the cost of reaching the goal. If the heuristic satisfies an additional property, *consistency*, then every state is generated for the first time through an optimal path, and the algorithm can be more efficient in dealing with duplicates.

As in the breadth-first search in the preceding section, A* maintains two data structures. The nodes that have been generated but not yet visited are stored in an *open list*, and all canonical nodes that have been visited are stored in a *closed list*. The implementations of these data structures, and how they are used, have a large impact on performance. We examine this in more detail in a later section. Listing 2 shows the complete pseudocode for a textbook A* search:

Listing 2. Pseudocode for A* search

```
function: A*-SEARCH(initial)
open ← {initial}
closed ← 0
loop do:
     if EMPTY(open) then return failure
     node ← BEST(open)
     if GOAL(node) then return SOLUTION(node)
     closed ← ADD(closed, node)
     for each action in ACTIONS(node)
          successor ← APPLY(action, node)
          if  successor in open or successor in closed
              IMPROVE(successor)
          else
              open ← INSERT(open, successor)
```

In Listing 2, A* begins with just the initial node in the open list. At each iteration of the loop, the best node on the open list is removed. Next, all applicable actions for the best node on `open` are applied, generating all possible successor nodes. For each successor node, we check to see if the state that it represents has been visited already. If it hasn't, we add it to the open list. If it has been visited, we need to determine if we have arrived at this state through a better path. If so, we need to place this node on the open list and remove the suboptimal node.

We can simplify this pseudocode with two assumptions about the problems we will be solving: We assume that all actions have the same cost, and that we have an admissible and consistent heuristic. Because the heuristic is consistent and all actions in the domain have the same cost, we can never revisit a state through a better path. It has also been shown that for some domains, it is more efficient to have duplicate nodes in the open list than it is to check for duplicates each time we generate a new node. Therefore, we can simplify the implementation by appending all new successor nodes to the open list regardless of whether they have been visited already. We simplify the pseudocode by combining the last four lines of Listing 2 into a single line. We still need to avoid cycles, so we must check for duplicates before expanding a node. We can omit the details of the `IMPROVE` function because it's no longer needed in the simplified version. Listing 3 shows the simplified pseudocode:

Listing 3. Simplified pseudocode for A* search

```
function: A*-SEARCH(initial)
open ← {initial}
closed ← 0
loop do:
     if EMPTY(open) then return failure
     node ← BEST(open)
     if node in closed continue
     if GOAL(node) then return SOLUTION(node)
     closed ← ADD(closed, node)
     for each action in ACTIONS(node)
          successor ← APPLY(action, node)
          open ← INSERT(open, successor)
```

A textbook implementation of A* in Java

In this section, we walk through a textbook implementation of A* in Java based on our simplified pseudocode in Listing 3. As you'll see, this implementation is incapable of solving a standard heuristic search benchmark within a memory constraint of 30GB.

We want our implementation to be as general as possible, so we start by defining some interfaces to abstract the problem that A* will be solving. Any problem we want to solve with A* must implement the `Domain` interface. The `Domain` interface provides methods for:

- Querying the initial state
- Querying the applicable actions of a state
- Computing the heuristic value of a state
- Generating successor states

Listing 4 shows the complete code for the `Domain` interface:

Listing 4. Java source for the `Domain` interface

```
public interface Domain<T> {
  public T initial();
  public int h(T state);
  public boolean isGoal(T state);
  public int numActions(T state);
  public int nthAction(T state, int nth);
  public Edge<T> apply (T state, int op);
  public T copy(T state);
}
```

The A* search generates edge and node objects for the search tree, so we need `Edge` and `Node` classes. Each node contains four fields: the state that the node represents, a reference to the parent node, and the node's `g` and `h` values. Listing 5 shows the complete code for the `Node` class:

Listing 5. Java source for the `Node` class

```
class Node<T> {
  final int f, g, pop;
  final Node parent;
  final T state;
  private Node (T state, Node parent, int cost, int pop) {
    this.g = (parent != null) ? parent.g+cost : cost;
    this.f = g + domain.h(state);
    this.pop = pop;
    this.parent = parent;
    this.state = state;
  }
}
```

Each edge has three fields: the cost or weight of the edge, the action used to generate the successor node for the edge, and the action used to generate the parent node for the edge. Listing 6 shows the complete code for the `Edge` class:

Listing 6. Java source for the `Edge` class

```
public class Edge<T> {
  public int cost;
  public int action;
  public int parentAction;
  public Edge(int cost, int action, int parentAction) {
    this.cost = cost;
    this.action = action;
    this.parentAction = parentAction;
  }
}
```

The A* algorithm itself will implement the `SearchAlgorithm` interface and requires only the `Domain` and `Edge` interfaces. The `SearchAlgorithm` interface provides just one method to perform a search with the specified initial state. The `search()` method returns an instance of `SearchResult`. The `SearchResult` class provides statistics about the search. The definition of the `SearchAlgorithm interface` is in Listing 7:

Listing 7. Java source for `SearchAlgorithm` interface

```
 public interface SearchAlgorithm<T> {
  public SearchResult<T> search(T state);
}
```

The choice of data structures to use for the open and closed lists is an important implementation detail. We'll use Java's `PriorityQueue` to implement the open list. The `PriorityQueue` is an implementation of a balanced binary heap with O(log *n*) time for enqueuing and dequeuing elements, linear time to test whether an element is in the queue, and constant time to access the head of the queue. The binary heap is a popular data structure for implementing the open list. You'll see later that for some domains, the open list can be implemented with a more efficient data structure called a *bucket priority queue*.

We must implement the `Comparator` interface to allow the `PriorityQueue` to sort the nodes properly. For the A* algorithm, we need to sort each node by its f value. In domains that have many nodes with the same f value, one simple optimization is to break ties by choosing the node with the higher g value. Take a minute to convince yourself why tie breaking this way can improve the performance of A*. (Hint: `h` is an estimate; `g` is not.) Listing 8 contains the complete code for our `Comparator` implementation:

Listing 8. Java source for the `NodeComparator` class

```
class NodeComparator implements Comparator<Node> {
  public int compare(Node a, Node b) {
    if (a.f == b.f) {
      return b.g - a.g;
    }
    else {
      return a.f - b.f;
    }
  }
}
```

The other data structure we need to implement is the closed list. An obvious choice for this is Java's `HashMap` class. The `HashMap` class is an implementation of a hashtable with expected constant time for retrieving and adding elements, provided we use a good hash function. We must override the `hashcode()` and `equals()` methods of the class responsible for implementing the state for a domain. We'll take a look at this implementation in the next section.

Finally, we need to implement the `SearchAlgorithm` interface. To do this, we implement the `search()` method using the pseudocode from Listing 3. Listing 9 shows the full code for the A* `search()` method:

Listing 9. Java source for A* `search()` method

```
 public SearchResult<T> search(T init) {
  Node initNode = new Node(init, null, 0, 0 -1);
  open.add(initNode);
  while (!open.isEmpty() && path.isEmpty()) {
    Node n = open.poll();
    if (closed.containsKey(n.state)) continue;
    if (domain.isGoal(n.state)) {
      for (Node p = n; p != null; p = p.parent)
        path.add(p.state);
      break;
    }
    closed.put(n.state, n);
    for (int i = 0; i < domain.numActions(n.state); i++) {
      int op = domain.nthAction(n.state, i);
      if (op == n.pop) continue;
      T successor = domain.copy(n.state);
      Edge<T> edge = domain.apply(successor, op);
      Node node = new Node(successor, n, edge.cost, edge.pop);
      open.add(node);
    }
  }
  return new SearchResult<T>(path, expanded, generated);
}
```

To evaluate our A* implementation, we need a problem to run it on. In the next section, we describe a popular domain for evaluating heuristic search algorithms. All actions in this domain have the same cost, and the heuristic we use is admissible, so our simplified implementation is sufficient.

The 15 puzzle benchmark

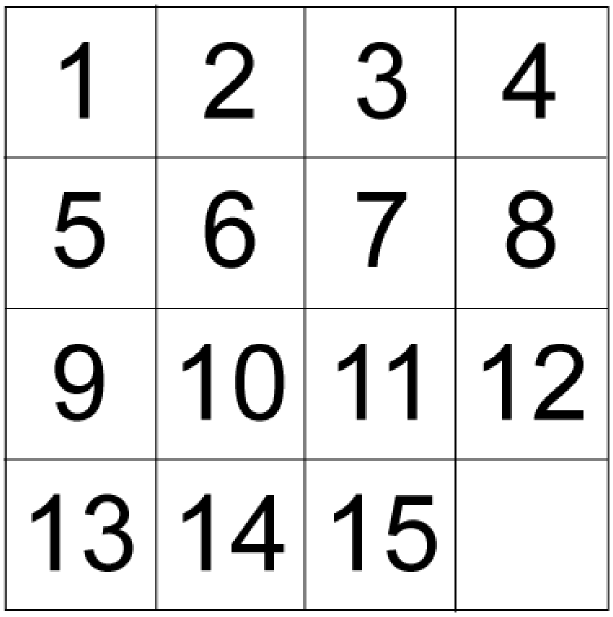
For the purposes of this article, we focus on a *toy* domain we call the 15 puzzle. This simple domain has well-understood properties and is a standard benchmark for evaluating heuristic search algorithms. (Some have referred to these puzzles as the "fruit flies" of AI research.) The 15 puzzle is a type of sliding-tiles puzzle that has 15 tiles arranged on a 4x4 grid. A tile has 16 possible locations, with one location always being empty. Tiles adjacent to the empty location can *slide* from one location to the other. The objective is to slide tiles until a goal configuration of the puzzle is reached. Figure 4 shows the puzzle with the tiles in a random configuration:

Figure 4. Random configuration of the 15 puzzle

3	2	1	4
5		11	8
9	7	10	12
13	14	6	15

Figure 5 shows the tiles in the goal configuration:

Figure 5. Goal configuration of the 15 puzzle

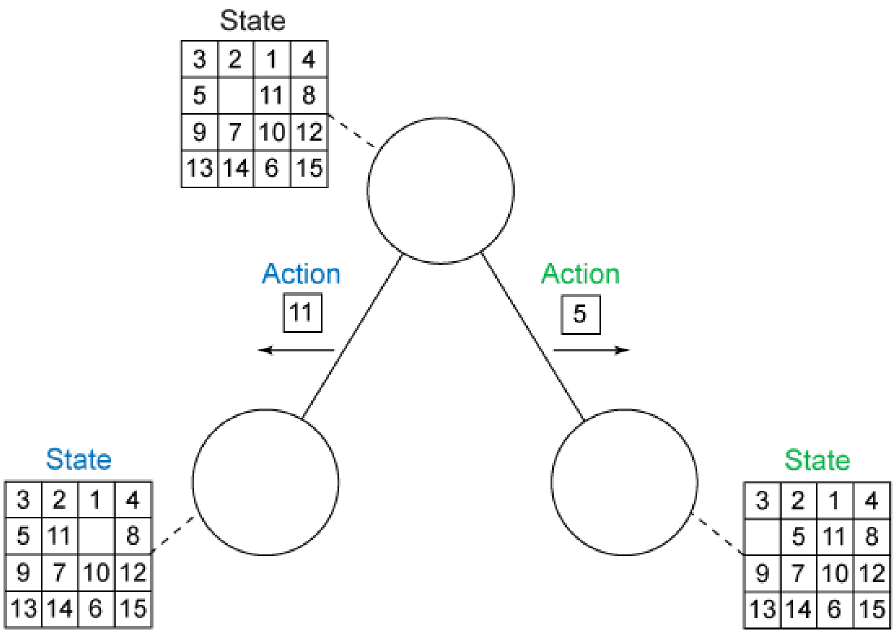


As a heuristic search benchmark, we want to find the goal configuration for this puzzle, starting from some initial configuration, using the fewest possible moves.

The heuristic we will use for this domain is called the *Manhattan distance* heuristic. A tile's Manhattan distance is the number of vertical and horizontal moves the tile must make to reach its goal position. To compute the heuristic of a state we take the sum of the Manhattan distance of all tiles in the puzzle, ignoring the blank. The sum of all of these distances for any state is guaranteed to be a lower bound on the cost to reach the goal state of the puzzle, because you can never move the tiles to each of their goal configurations by making fewer moves.

It might not seem intuitive at first, but we can model the 15 puzzle with a graph by representing each possible configuration of the tiles as nodes in the graph. An edge connects two nodes if there is a single action that transforms one configuration into the other. An action in this domain is sliding a tile into the blank space. Figure 4 illustrates this search graph:

Figure 6. State-space search graph for the 15 puzzle



There are 16! possible ways to arrange 15 tiles on the grid, but there are actually "only" 16!/2 = 10,461,394,944,000 reachable configurations or states of the 15 puzzle. This is because the physical constraints of the puzzle allow us to reach exactly half of all possible configurations. To get an idea of the size of this state space, suppose that we could represent a state with just a single byte (which is impossible). To store the entire state space we would need more than 10TB of memory. This would far exceed the memory limit of most modern computers. We'll show how heuristic search can solve the puzzle optimally while visiting only a small fraction of the state space.

Running the experiments

Our experiments use a famous set of starting configurations of the 15 puzzle called the Korf 100 set. This set is named after Richard E. Korf, who published the first results showing that random 15 puzzle configurations could be solved using an *iterative-deepening* variant of A* called IDA*. Since these results were published, the 100 random instances Korf used in his experiments have been reused in countless subsequent experiments in heuristic search. We also optimize our implementation so that the iterative-deepening technique is no longer necessary.

We solve one starting configuration at a time. Each starting configuration is stored in a separate plain-text file. The file's location is specified in the command-line arguments for starting the experiments. We need an entry point to our Java program that will process the command-line argument, generate a problem instance, and run the A* search. We call this entry-point class `TileSolver`.

Statistics regarding the performance of the search are printed to standard output at the end of each run. The statistic we're most interested in is the wall-clock time. We sum the wall clock times of all runs to get a total running time for this benchmark.

The source code for this article contains Ant tasks for automating the experiments. You can run the full experiment with:

```
ant TileSolver.korf100 -Dalgorithm="efficient"
```

You can specify either `efficient` or `textbook` for `algorithm`.

We also provide an Ant target for running a single instance. For example:

```
ant TileSolver -Dinstance="12" -Dalgorithm="textbook"
```

And we provide an Ant target for running the subset of the benchmark that excludes the three most difficult instances:

```
ant TileSolver.korf97 -Dalgorithm="textbook"
```

Chances are that your computer doesn't have enough memory to complete the full experiment using the textbook implementation. To avoid swapping, you should be careful to limit the amount of memory available to the Java process. If you are running this experiment on Linux, you can use a shell command like `ulimit` to set the memory limits for the active shell.

At first we don't succeed

Table 1 shows the results for all of the techniques that we used. The results for the textbook A* implementation are in the first row. (We describe packed states and HPPC, and their associated results, in subsequent sections.)

Table 1. Results of three variants of A* solving 97 instances of the Korf 100 15 puzzle benchmark

Algorithm	Max. memory usage	Total running time
Textbook	25GB	1,846 sec
Packed states	11GB	1,628 sec
HPPC	7GB	1,084 sec

The textbook implementation was unable to solve all of our test instances. We failed to solve three of the most difficult instances, and for the instances we could solve, it took more than 1,800 seconds. These are not great results, considering that the best implementation in C/C++ can solve all 100 of these instances in less than 600 seconds.

We failed to solve the hardest three instances because of memory constraints. At every iteration of the search, removing a node from the open list and expanding it usually leads to the generation of several more nodes. As the number of generated nodes grows, so does the amount of memory required to store them in the open list. However, this demand for memory is not specific to our Java implementation; an equivalent implementation in C/C++ would also fail.

In their paper, Burns et al. (see Resources) show that an efficient C/C++ implementation of A* search could solve this benchmark with less than 30GB of memory — so we weren't ready to give up on a Java A* implementation just yet. There are additional techniques, discussed in the following sections, that we can apply to be more efficient about how memory is used. The result is an efficient Java implementation of A* that is capable of quickly solving the entire benchmark.

Packing states

When we examine the memory usage of our A* search using a profiler such as VisualVM, we see that all the memory is being taken up by the `Node` class and more directly by the `TileState` class. To reduce memory usage, we need to revisit the implementation of these classes.

Every tile state must store the positions of all 15 tiles. We do this by storing the location of each tile in an array of 15 integers. We can represent these locations more concisely by packing them into a 64-bit integer (a `long` in Java). When we need to store a node in the open list, we can store just the packed representation of the state. Doing so will save 52 bytes per node. Solving the hardest instance in the benchmark requires storing approximately 533 million nodes. By packing the state representation, we save more than 25GB of memory!

To maintain the general nature of our implementation, we need to extend the `SearchDomain` interface to have methods for packing and unpacking states. Before storing a node on the open list, we'll now produce a packed representation of the state and store this packed representation in the `Node` class instead of a pointer to a state. When we need to generate a node's successors, we simply unpack the state. Listing 10 shows the implementation for the `pack()` method:

Listing 10. Java source for the `pack()` method

```
public long pack(TileState s) {
  long word = 0;
  s.tiles[s.blank] = 0;
  for (int i = 0; i < Ntiles; i++)
    word = (word << 4) | s.tiles[i];
  return word;
}
```

Listing 11 shows the implementation for the `unpack()` method:

Listing 11. Java source for the `unpack()` method

```
public void unpack(long packed, TileState state) {
  state.h = 0;
  state.blank = -1;
  for (int i = numTiles - 1; i >= 0; i--) {
    int t = (int) packed & 0xF;
    packed >>= 4;
    state.tiles[i] = t;
    if (t == 0)
      state.blank = i;
    else
      state.h += md[t][i];
  }
}
```

Because the packed representation is a canonical form of the state, we can store the packed representation in the closed list. We can't just store primitives in the `HashMap` class. They need to be wrapped in an instance of the `Long` class.

The second row in Table 1 shows the result of running our experiment with a packed state representation. By using a packed state representation, we have reduced the amount of memory used by 55 percent and improved the running time a bit, but we still cannot solve the entire benchmark.

The problem with the Java Collections Framework

If you think that wrapping every packed state representation in an instance of `Long` seems like a lot of overhead, you are correct. It is a waste of memory and it can lead to excessive garbage collection. JDK 1.5 added support for *autoboxing*, which automatically converts primitive values to their object representation (`long` to `Long`) and vice versa. With large collections, these conversions can degrade memory and CPU performance.

JDK 1.5 also introduced Java generics: a feature often compared to C++ templates. Burns et al. show that C++ templates provide a huge performance benefit for implementing heuristic search. Generics provide no such benefit. Generics are implemented using *type-erasure*, which removes (erases) all type information at compile time. As a result, type information must be checked at run time, which can lead to performance issues for large collections.

More about memory

For more information on memory usage in Java, with an emphasis on many of the JCF classes, we recommend that you read Chris Bailey's excellent article on developerWorks, "From Java code to Java heap."

The implementation of the `HashMap` class reveals some additional memory overhead. `HashMap` stores an array of instances of an internal `HashMap$Entry` class. Each time we add an element to `HashMap`, a new entry is created and added to the array. Implementations of this entry class typically contain three object references and one 32-bit integer reference, for a total of 32 bytes of per entry. With 533 million nodes in our closed list, we will have more than 15GB of overhead.

Next, we introduce an alternative to the `HashMap` class that enables us to reduce memory even further by storing primitives directly.

High Performance Primitive Collections

Because we're storing primitives only in the closed list now, we can take advantage of the High Performance Primitive Collections (HPPC). HPPC is an alternative collections framework that lets you store primitive values directly without all of the overhead of JCF (see Resources). In contrast to Java generics, HPPC uses a technique similar to C++ templates whereby separate implementations of each collection class and Java primitive type are generated at compile time. This eliminates the need to wrap primitive values with classes like `Long` and `Integer` when you store them in a collection. As a side effect, it enables you to avoid a lot of the casting that is otherwise necessary with JCF.

Other alternatives to JCF for storing primitive values are also available. The Apache Commons Primitive Collections and fastutils are two excellent examples. However, we think that the design of HPPC has a one significant advantage for implementing high-performance algorithms: It exposes the internal data storage for each of the collection classes. Having direct access to this storage can enable many optimizations. For example, if we wanted to store the open or closed list on disk, we could do this more efficiently by having direct access to the underlying data arrays rather than having indirect access to the data through an iterator.

We can modify our A* implementation to use an instance of the `LongOpenHashSet` class for the closed list. The changes we need to make are quite simple. We no longer need to override the `hashcode` and `equals` methods of the state `class`, because we're storing just primitive values. The closed list is a set (it doesn't contain duplicate elements), so we only need to store values rather than key/value pairs.

The third row in Table 1 shows the results of running our experiments with HPPC in place of JCF. With HPPC we've reduced the amount of memory used by 27 percent and the run time by 33 percent.

Now that memory has been reduced by a total of 82 percent, we can solve the entire benchmark within our memory constraint. The results are in the first row of Table 2:

Table 2. Results of three variants of A* solving all 100 instances of the Korf 100 benchmark

Algorithm	Max. memory usage	Total running time
HPPC	30GB	1,892 sec
Nested bucket queue	30GB	1,090 sec
Avoid garbage collection	30GB	925 sec

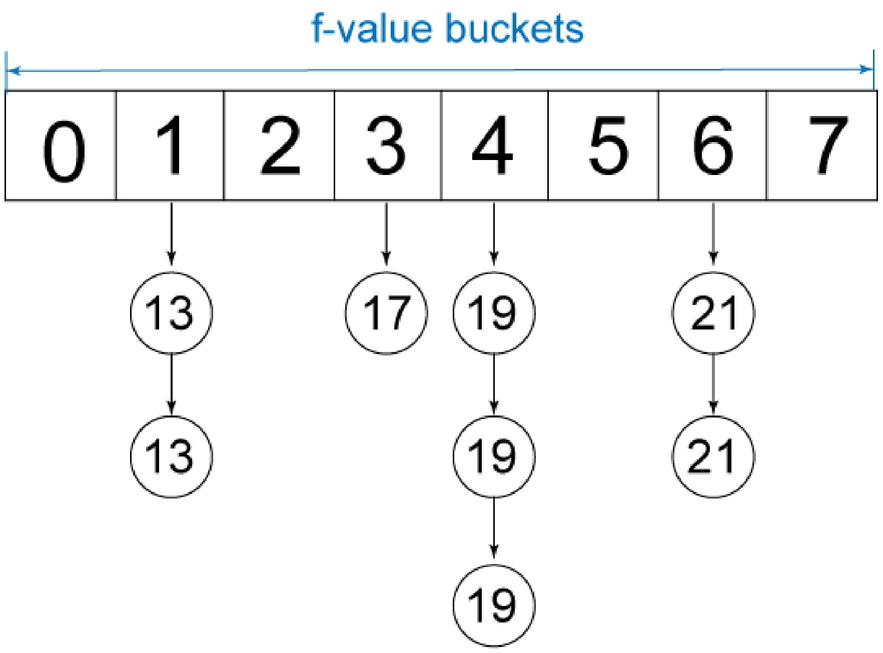
With HPPC, we can solve all 100 instances with 30GB of memory, but it takes more than 1,800 seconds. The other results in Table 2 reflect ways that we made our implementation faster by improving on the other important data structure: the open list.

The problem with `PriorityQueue`

Each time we add an element to the open list, the queue needs to be sorted again. `PriorityQueue` has O(log(*n*)) time for enqueue and dequeue operations. This is efficient when it comes to sorting, but it's certainly not free, especially for large values of *n*. Recall that for the most difficult problem instance we are adding more than 500 million nodes to the open list. Moreover, because all of the actions in our benchmark problem have the same cost, the range of possible f values is small. So the benefit of using a `PriorityQueue` might not be worth the overhead.

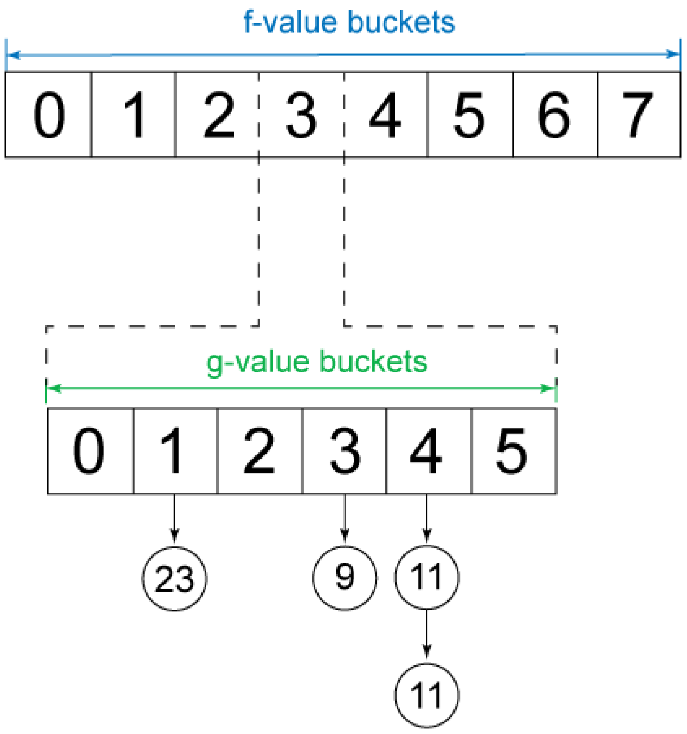
One alternative is to use a bucket-based priority queue. Assuming the action costs in our domain fall within a narrow range of values, we can define a fixed range of buckets: one bucket per f value. When we generate a node, we simply put it in the bucket with the corresponding f value. When we need to access the head of the queue, we look in the buckets with the smallest f value first until we find a node. This type of data structure, called a *1-level bucket priority queue*, enables constant-time enqueue and dequeue operations. Figure 7 illustrates this data structure:

Figure 7. 1-level bucket priority queue



Astute readers will notice that if we implement a 1-level bucket priority queue as described here, we lose the ability to break ties between nodes using their g values. You should have convinced yourself earlier that breaking ties this way is a worthwhile optimization. To maintain this optimization, we can implement a *nested* bucket priority queue. One level of buckets is used to represent the range of f values, and the nested level is used to represent the range of g values. Figure 8 illustrates this data structure:

Figure 8. Nested bucket priority queue



We can now update our A* implementation to use a nested bucket priority queue for the open list. The complete implementation of the nested bucket priority queue can be found in the BucketHeap.java file included with the source code for this article (see Download).

The second row of Table 2 shows the results of running our experiment using the nested bucket priority queue. By using a nested bucket priority queue instead of `PriorityQueue`, we have improved our running time by nearly 58 percent, but it still takes over 1,000 seconds. We can do one more simple thing to improve the running time.

Avoiding garbage collection

Garbage collection is often considered a bottleneck in Java. Many excellent articles on the subject of tuning garbage collection in the JVM are available that apply here (see Resources), so we won't go into any details on that subject.

A* typically generates many short-lived state and edge objects and incurs a great deal of costly garbage collection. By reusing objects, we can reduce the amount of garbage collection that is required. We can make some simple changes to do this. In each iteration of the A* search loop, we allocate a new edge and a new state (533 million for the hardest problem). Instead of allocating new objects each time, we can reuse the same state and edge objects across all iterations of the loop.

To have reusable edge and state objects, we need to modify the `Domain` interface. Instead of the `apply()` method returning an instance of `Edge`, we need to provide our own instance that is modified by the call to `apply()`. The changes to `edge` are not incremental, so we don't need to worry about which values are stored in `edge` before we pass it to `apply()`. However, the changes that `apply()` makes to the `state` object *are* incremental. To properly generate all possible successor states, without needing to copy the state, we need a way to undo the changes that are made. For this, we must extend the `Domain` interface to have a `undo()` method. Listing 12 shows the changes to the `Domain` interface:

The updated `Domain` interface

```
public interface Domain<T> {
  ...
   public void apply(T state, Edge<T> edge, int op);
   public void undo(T state, Edge<T> edge);
  ...
}
```

The third row in Table 2 shows the results of our final experiment. By recycling our state and edge objects, we avoid costly garbage collection and reduce our running time by more than 15 percent. With our highly efficient Java implementation of A*, we can now solve the entire benchmark set in 925 seconds using just 30GB of memory. This is an excellent result, considering that the best C/ C++ implementation takes 540 seconds and requires 27GB. Our Java implementation is just 1.7 times slower and requires approximately the same amount of memory.

Conclusion

In this article, we introduced you to heuristic search. We presented the A* algorithm and illustrated a textbook implementation in Java. We demonstrated that this implementation suffers from performance issues and is unable to solve a standard benchmark problem with reasonable time or memory constraints. We addressed these problems by taking advantage of HPPC and some techniques for reducing memory use and avoiding costly garbage collection. Our improved implementation was able to solve the benchmark with modest time and memory constraints, demonstrating that Java is a fine choice for implementing heuristic search algorithms. Furthermore, the techniques we present in this article can also be applied to many real-world Java applications. For example, in some cases using HPPC can immediately improve the performance of any Java application that stores a large number of primitive values.

Acknowledgments

We gratefully acknowledge support from the NSF (grants 0812141 and 1150068), DARPA (grant N10AP20029), and a University of New Hampshire Dissertation Year Fellowship.

Downloads

Description	Name	Size
Sample code	j-ai-code.zip	58KB

Resources

Learn

- "Implementing Fast Heuristic Search Code" (Ethan Burns et al, *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, 2012): You can learn more about implementing fast heuristic search code in this academic article.
- "Planning Long Dynamically-Feasible Maneuvers for Autonomous Vehicles" (Maxim Likhachev and Dave Ferguson, *International Journal of Robotics Research*, August 2009): This paper presents an algorithm for generating complex dynamically feasible maneuvers for autonomous vehicles traveling at high speeds over large distances.
- *Artificial Intelligence: A Modern Approach* (Stuart Russel and Peter Norvig, Prentice Hall, 2010): This book is one of the best resources for learning more about AI.
- *The Fifteen Puzzle* (Jerry Slocum and Dic Sonneveld, Slocum Puzzle Foundation, 2006): This book is devoted entirely to the history and mathematics behind the 15 puzzle.
- High Performance Primitive Collections: HPPC is a high-performance collections alternative to JCF for storing primitive values.
- "Dynamic programming and sequence alignment" (Paul D. Reiners, developerWorks, March 2008): Read about MSA and how computer science aids molecular biology.
- "From Java code to Java heap" (Chris Bailey, developerWorks, February 2012): Find out more about how memory is used by Java and the JCF.
- "*Java theory and practice*: Garbage collection and performance" (Brian Goetz, developerWorks, January 2004): Learn more about the performance of the garbage collectors in the JVM.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- HPPC: Download HPPC.

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the authors

Matthew Hatem

Matthew Hatem is a Ph.D. candidate at the University of New Hampshire. His research includes parallel and external memory heuristic search algorithms and applications. His work has been published in the proceedings of the Association for the Advancement of Artificial Intelligence and of the Symposium on Combinatorial Search. He is also a senior software engineer at IBM, currently working for the Watson Solutions team. Previously, he worked on various Lotus products and was a committer at eclipse.org.

Ethan Burns

Ethan Burns is a software engineer at Google and has a Ph.D. from the University of New Hampshire. He has worked on a variety of projects including an iSCSI Linux kernel module, parallelizing the SPIN model checker, parallel heuristic search and automated planning, planning techniques for robotics, and algorithms for heuristic search under time pressure. He was awarded the Richard Lyczak memorial teaching award in 2007, the best program committee award for the Symposium on Combinatorial Search in 2012, and a UNH Dissertation Year Fellowship for 2012 - 2013.

Wheeler Ruml

Wheeler Ruml is an associate professor of computer science at the University of New Hampshire, where he leads the UNH Artificial Intelligence Group. He is co-founder of the International Symposium on Combinatorial Search and co-chair of the 2014 ICAPS Conference. He was selected for the DARPA Computer Science Study Panel, an NSF CAREER award, and the UNH Outstanding Assistant Professor Award. Before joining UNH, he led a team at Xerox PARC that used AI techniques to build the world's fastest printer. He received his Ph.D. from Harvard University in 2002.