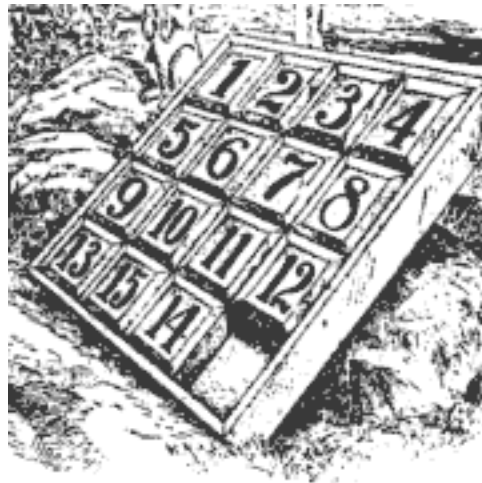


# Project 1: Slide

*Due: 10:00 PM, Feb 12, 2009*

## Contents

1	Sliding Tiles	2
2	A* Search	2
3	Unsolvability	3
4	What to Do	3
5	Extra Credit	4
6	What to Handin	4
7	Support Code	4
8	Resources	7



The sliding tiles puzzle was invented in America in the 1870s. The game is played on an  $n \times m$  grid with  $nm - 1$  tiles numbered 1 through  $nm - 1$ , and one vacant space left open for “sliding.” In 1878, Sam Loyd, a well-known puzzlemaker (inventor of Parcheesi), published a challenge in a Sunday New York newspaper. He presented a  $4 \times 4$  grid of sliding tiles with all the tiles in order, except the 14 and the 15, which were swapped. Loyd offered \$1000 (equivalent to about \$40,000 in today’s dollars) to the first individual who could solve the puzzle by sliding the numbers into order. A puzzle craze swept America, so much so that employers were forced to put up notices prohibiting employees to play the puzzle during office hours. Lucky for Sam, he never awarded anyone the \$1000. You are about to learn why no one was able to claim it.

# 1 Sliding Tiles

For this project, you will program two versions of sliding tiles, both of which are played on an  $n \times n$  grid, with  $n^2 - 1$  numbered tiles and one open space. The goal (in both versions) is to move all of the tiles into row-major order (i.e., 1 to  $n$  in the first row,  $n + 1$  to  $2n$  in the second row,  $\dots$ , and finally,  $n^2 - n + 1$  to  $n^2 - 1$  in the last row), with the blank square in the bottom-right-most corner (see Figure 1). Any tile that is *adjacent* to the open space may be moved into the open space.

The two versions of sliding tiles differ in their definitions of adjacency. In the first, which is played on a square board, tile  $x$  is adjacent to tile  $y$  iff  $x$  is immediately to the to left, right, above or below  $y$  (See Figure 2). In the second, which is played on a torus, tile  $x$  is adjacent to tile  $y$  if  $x$  is immediately to the to left, right, above or below  $y$ ; or, if  $x$  lies along an edge of a row, or column, then  $x$  is adjacent to the square that lies at the opposite end of the row, or column (See Figure 3).

A board with  $n^2 - 1$  tiles is referred to as the  $n^2 - 1$  puzzle: e.g., the 3-puzzle (for  $n = 2$ ), 8-puzzle (for  $n = 3$ ), the 15-puzzle (for  $n = 4$ ), the 24-puzzle (for  $n = 5$ ), etc. No computer program has ever solved random instances of sliding tiles larger than the 24-puzzle. Yours should be able to solve a few instances of the 15-puzzle, most instances of the 8-puzzle, and all instances of the 3-puzzle.

Figure 1: Goal State.

1	2	3
4	5	6
7	8	

Figure 2: Legal moves on a square board: An arrow indicates the direction in which a tile can be moved. An 'X' means that a tile cannot be moved.

X	↓	X
→		←
X	↑	X

→		←
X	↑	X
X	X	X

	←	X
↑	X	X
X	X	X

Figure 3: Legal moves on a torus: An arrow indicates the direction in which a tile can be moved. An 'X' means that a tile cannot be moved.

X	↓	X
→		←
X	↑	X

→		←
X	↑	X
X	↓	X

	←	→
↑	X	X
↓	X	X

# 2 A\* Search

Your goal in this assignment is to write a program that can solve random instances of sliding tiles puzzles (both versions), and identify instances that are not solvable. To accomplish this, you

will implement A\* search using the support code provided. The details of the A\* algorithm were presented in class, but we reiterate a few of the more salient points here.

A\* search on trees is optimal when implemented with an admissible heuristic function. This means that the algorithm is guaranteed to find an optimal path through the search tree, as long as the estimates of the distance from a node to the nearest goal are optimistic. That is, the heuristic function can never report that a node is farther away from the nearest goal than it actually is. One admissible heuristic for sliding tiles played on a square board is Manhattan distance.

Using Manhattan distance, A\* search can be used to solve most instances of the 8-puzzle and a few instances of the 15-puzzle, but it cannot solve the 24-puzzle. To solve the 24-puzzle requires better (i.e., tighter) heuristics. The better the heuristic, the faster A\* converges on the goal; and, perhaps more importantly, the more nodes that can be pruned along the way. Thus, the quality of a heuristic often determines whether a difficult problem is tractable or not.

You should think hard about the heuristics you design for this assignment. Try to make sure they are admissible so that your implementation is optimal, but not too optimistic so as to lead A\* search to the goal too slowly using too much memory. For example,  $h(n) = 0$ , while an admissible heuristic, is too optimistic to be useful. You are welcome to consult the AI literature in your quest to develop improved heuristics, but you must cite all references.

If you want to solve bigger and harder versions of sliding tiles, you can pay attention to the graph-theoretic nature of the puzzles. In particular, you can keep track of the board configurations that you have already encountered, and either promote or re-open them if you reach them again via a shorter path, and otherwise discard them. This additional bookkeeping is not necessary for optimality (A\* search on graphs is optimal when implemented with a consistent heuristic function, and most admissible heuristics are also consistent.), but it does improve performance.

### 3 Unsolvability

Although A\* does always terminate, it will only do so after exploring the whole state space if there is no solution. In some search problems, such as sliding tiles, it is possible to detect unsolvability more easily. Loyd's puzzle, with the 14 and 15 swapped, is known to be unsolvable. The solvability of a board is related to the number of swaps between it and the goal state. You can read about a quick check for solvability on square boards here: <http://mathworld.wolfram.com/15Puzzle.html>.

### 4 What to Do

There are four parts to this assignment: a straightforward warm-up exercise, and three more challenging follow-up components designed to make you think.

#### 1. Warm Up:

- (a) Implement the `SlideBoard` class; that is, implement the search nodes, which consist primarily of sliding tiles boards.

**Tip:** Try to keep the size of each node to a minimum. By cutting down on the amount of space required at each node (even if only by a little), your program will be able to

examine more nodes, making it more likely to find a solution to hard problems before the system's memory is exhausted.

## 2. Part I:

- (a) Implement A\* search on trees using the Manhattan distance heuristic that is invoked from the `SlideSolver` class when the `solve()` method is called on a square board. Make sure you use the `SlideSearchNode` class to make your algorithm abstract.
- (b) Make your search algorithm complete for square boards: If an instance of the puzzle is solvable, then the `solve()` method should return an optimal path to the solution as a `LinkedList` of `SlideBoard` instances. If an instance of the puzzle is unsolvable, then the `solve()` method should return `null`.

## 3. Part II:

- (a) Extend your program so that it can handle torus-shaped boards.
- (b) The Manhattan distance as defined on square boards is not an admissible heuristic for sliding tiles played on a torus. (Why not?) Implement an admissible heuristic for this version of the puzzle.

## 5 Extra Credit

For extra credit, you can implement alternative admissible heuristics, other than ones based on Manhattan distance. Feel free to use external sources to help you come up with ideas, but do tell us: Is your heuristic original, or a variant of a known heuristic? Further, argue that your heuristics are admissible. Beyond coming up with clever admissible heuristics, you can also implement A\* search on graphs. You should notice a marked improvement in run time after doing this.

## 6 What to Handin

Please hand in the following:

1. All the code necessary to solve Parts I and II of the assignment, and any extra credit.
2. Written description of your heuristic for torus-shaped boards, including an argument that it is admissible. **Note:** All written work must be submitted *electronically*, as a pdf document.

For a quick refresher on how to hand in assignments, please refer back to the course missive.

## 7 Support Code

To copy the support code into your `/u/yourlogin/course/cs141/projects/slide` directory, run the script `/course/cs141/bin/cs141install slide`. This will install all of the support code in

your `/course/cs141/projects/slide` directory, including some test puzzles. Note that the files must be in a subdirectory named `slide` or else Java will get confused, as all of the classes belong to the package `slide`. Included with the support code is a file called `build.xml`, which allows you to easily compile and run your code using the program `ant`. If you haven't used `ant` before, you can read all about it later in the handout.

Once you have copied the source files you can see a demo by typing `ant rundemo` in the shell. You'll notice that the GUI includes a "Load" button that allows you to load puzzle descriptions from a file. We have provided a few sample puzzles for you. Puzzle files have the extension `.sld` and have the following format:

```
<# rows> <# cols>
<1,1> <1,2> <1,3> ... <1, n> <2, 1> <2, 2> ... <2, n> ... <n, n>
```

`<1,2>` means the value of the square at row 1 and column 2, the value of the empty square is 0. Feel free to manually create your own puzzle files. You will also be coding a constructor for your `SlideBoard` class that will create a random test instance when the "Random" button is pushed.

Take a few minutes to understand the `SlideBoard`, `SlideSearchNode`, `TorusSlideSearchNode`, and `SlideSolver` classes since you will be working in these files. You will be required to implement several methods in `SlideBoard`, `SlideSearchNode`, `TorusSlideSearchNode`, and the `solve()` method in `SlideSolver`. The `SlideBoard` class maintains the state of a board. You can store this state information in a data structure of your choosing. The `SlideSearchNode` and `TorusSlideSearchNode` act as instances of search nodes. `SlideSearchNode` extends the `UHSearchNode` class (a more generic search node from the cs141 `search` library described below), and the `TorusSlideSearchNode` extends `SlideSearchNode`. Both of these classes implement the `Comparable` interface so that they can be properly ordered in a priority queue.

## Priority Queues

The A\* search algorithm that you will implement requires returning the node of lowest value in a set according to some heuristic function. Finding the lowest value node in a set requires either searching through the set, sorting the set each time it is queried, or sorting the set and maintaining the sorted order as new elements are added. As you have probably learned, Computer Scientists spend lots of time and energy developing clever ways of implementing speedy versions of such sets. One such set is the priority queue. A priority queue is an abstract data type that keeps the node with the smallest value at the front of the list. Java's implementation, `java.util.PriorityQueue`, is ordered, meaning that you can easily grab the lowest valued element in constant time, and it guarantees  $\log(n)$  running time for the `add()`, `remove()`, and `contains()` methods. We recommend you use Java's priority queue; however, there are a few things that you will need to know in order to maintain your sanity while working with these classes.

- The `Comparable` interface : This interface is the key to keeping the elements of your `PriorityQueue` ordered. Any objects that will be put into the set must implement this interface, otherwise the behavior of the set is "unspecified".<sup>1</sup> Implementing this interface requires only that your object have the `int compareTo(Object other)` method declared.

---

<sup>1</sup>There is an additional way of maintaining order in a `PriorityQueue` using an object called a `Comparator` that works in a very similar way. You can read more about `Comparators` in the Java API if you're interested.

- `int compareTo(Object other)` : The behavior of this method seems simple enough, if this object is less than `other` as specified by some arbitrary function, then `compareTo()` should return `-1`. If this object is equal to `other` it should return `0`, and if this object is greater than `other` it should return `1`. The important thing to note is that `compareTo()` should *only* return `0` if the two objects being compared are equal. If only their function values are equal, it *should not* return `0`. We cannot stress these points enough, as it will cause you much pain and hours of debugging if you ignore these rules.
- `poll()` and `peek()` : After your elements have been inserted and ordered by the `PriorityQueue`, you'll want to grab the one with the smallest value. The method `poll()` will remove and return the smallest (first) element of the queue, and `peek()` will return the smallest element without removing it. Make sure you're using the right one!

**Note:** As always you can get additional information from the Java API which is linked off of the course website, or from the TAs during hours.

## CS141 Libraries

The CS141 TAs have put together a library of Java classes that you can access while working on your assignments. The classes include abstract nodes and interfaces that can be used to make your code more general. For example, the A\* search algorithm that you implement for this assignment should apply to any search problem. We ask you to use this library not only because it will make your life easier but because it will leave you with that warm fuzzy feeling of knowing you've made something you can use over and over again as the need arises.

For this project, you will be using the `search.*` library. This search library includes a bunch of generic search nodes that are suitable for any search problem. The classes `HeuristicSearchNode`, `SearchNode`, `UHSearchNode` and `Unsolvable` in the `search.*` library are installed in your `slide` package. Extensive documentation for the support code can be found by typing the path

```
file:///course/cs141/lib/search/docs/index.html
```

into your favorite browser.

## The slideGUI Package

This package includes a nice little user interface for your code. You don't ever have to worry about the classes in this package, but if you're interested, they too can be found in the `lib/` directory.

## Ant

Apache Ant is a Java-based build tool—like `make`, but not as broken. When you type `ant` in your shell, it looks at the xml file called `build.xml` in the current directory. This file consists mainly of a series of targets that `ant` can build. We've created the build file for you, but you can check it out to see what targets are available. For example, typing `ant` compiles your code, `ant run` runs your program, `ant rundemo` starts the TA demo, and `ant clean` removes class files. For as much information as you could possibly want on ant, see <http://ant.apache.org/manual/index.html>.

To compile, `ant` will need the `JAVA_HOME` environment variable set. It's easiest to put this in your `.environment` file like this:

```
setenvvar JAVA_HOME /pro/java/Linux/jdk1.5.0
```

And a final note on Eclipse: this IDE can certainly make your life easier while you're coding in Java. However, just because your project runs in the Eclipse environment doesn't mean we can run it once you hand it in. Make sure your project compiles and runs with the build file we give you, so that we can run your code too, and give you a grade.

## 8 Resources

1. *Sliding Piece Puzzles*, by Edward Hordern. Oxford University Press, 1986.
2. 15 Puzzle, by Jerry Slocum and Eric W. Weisstein. In *MathWorld—A Wolfram Web Resource*. <http://mathworld.wolfram.com/15Puzzle.html>