November 13, 2016

Disjoint Sets Union/Find Algorithms

Here we discuss one of the most elegant algorithms in the CS toolkit. Some of the features adding to the elegance:

- The algorithms keep track of subtle features of partitions using a very simple and compact data structure design
- The algorithms served as the platform for the introduction of self-organizing data structures, a very influential area of design and research
- Applications of the algorithms are wide, deep, and important
- Analysis of the algorithms is quite subtle and slippery, just the opposite of the data structure and algorithm bodies

The data structure (in all of its variances) and algorithms are known by at least three names: Disjoint Sets, Union-Find, and Partition. We will first describe an initial version of the data structure and associated algorithms, and then discuss a few representative applications. Then we will improve the efficiency of the algorithms, and conclude with a proof of a very good bound on the amortized runtime of the algorithms.

Outside the Box

Best Lecture Slides	PU	
Best Video	UCB	
Analysis Papers	UMD	ACM
Demos	random maze generator	`LIB/area51/ranmaze.x`
	maze analyzer	`LIB/area51/mazemaster.x`

Textbook deconstruction. The chapter on disjoint sets is one of the more readable. After surveying ways to do it inefficiently, they settle on the implementation discussed in these notes. There is a good discussion of the Ackerman function and its inverse α. Their proof of the tight bound $\Theta(\alpha(n))$ (discovered by Robert Tarjan - see ref ACM above) on worst case amortized runtime of union/find uses potential functions.

1 Partition: The Disjoint Sets Union-Find ADT

So we have used all three names in the title of this section. Any of the three will mean the same thing in the remainder of this chapter. At the level of optimized code, our algorithm class will be called `Partition`.

Recall that a *partition* of a set U is defined to be a collection $\mathcal{P}$ of subsets of U such that:

1. Every element of U is in one of the sets in $\mathcal{P}$
2. No two sets in $\mathcal{P}$ have elements in common

In other words: U is the union of all sets in $\mathcal{P}$ and $\mathcal{P}$ is pairwise disjoint. U is called the *universe* of $\mathcal{P}$ and $\mathcal{P}$ is a *partition* of U.

Recall also that partitions and equivalence relations are reflections of one another. A partition $\mathcal{P}$ determines an equivalance relation R on U by defining

$$xRy \text{ iff } x \text{ and } y \text{ belong to the same set in } \mathcal{P}$$

Similarly, if R is an equivalance relation on U, then the set of *equivalence classes* of R is a partition:

$$R[x] = \{y \in U : xRy\}$$
$$\mathcal{P} = \{R[x] : x \in U\}$$

The union-find algorithms take as arguments elements of a universe and operate on partitions of that universe. The two most fundamental algorithms are, not surprisingly, Union and Find:

Union (x, y)	Modifies the partition by forming the union of the set containing x and the set containing y
Find (x, y)	Returns true iff x and y belong to the same set in the partition

2 Implementing Union-Find

A simple and clever way to implement these algorithms is using a tree model. Assume that the universe is an initial segment of non-negative integers $U = \{0, 1, \ldots, n-1\}$. Let v be a vector of integer elements with size n. We will hold the parent of element x in $v[x]$, and denote roots (elements without parents) with value -1:

$$v[x] = \text{parent of } x \text{ , if } x \text{ has a parent}$$
$$v[x] = -1, \text{ if } x \text{ has no parent}$$

The set of elements in a tree represents a set in the partition of U.

Thus v contains the information defining a *forest*, a collection of trees, and trees in the forest represent the sets in the partition. The structure is initialized to the

partition of singleton sets $\mathcal{P} = \{\{0\}, \{1\}, \ldots \{n-1\}\}$, which means every element is in a tree of one node by itself. This is repesented by $v[x] = -1$ for all x.

Using this representation, we define two subsidiary operations on the data structure:

Union (x, y)	= **Link**(**Root**(x),**Root**(y))
Find (x, y)	= (**Root**(x) == **Root**(y))
Link $(root1, root2)$	Modifies the partition by merging the two roots
Root (x)	Returns the root of the tree containing x

Union and Find are implemented in terms of Link and Root, which are representation-specific.

The Root operation is easily implemented as "follow the parent pointer chain up to the root":

```
int Root(x) // pure search - no change of state
{
  if (v[x] < 0) // x is a root
    return x;
  else
    return Root(v[x]);  // recursive: root = root of parent
}
```

NOTE. The run time of Root(x) is the length of the path from x to the root above x

The Link operation is also straightforward to implement. Because it is clear that the trees in the forest would be best kept short and bushy, we take a bit of care to ensure the tree heights grow slowly. For this we need a way to keep track of tree height.

Because we have stored -1 in $v[x]$ to denote that x is a root in the forest, and all other v elements are non-negative, we can encode (or compress) more information into v[root]. The clever idea is to re-define v as follows:

$$v[x] = \text{parent of } x \text{ , if } x \text{ has a parent}$$
$$v[x] = -1 - \text{ height, if } x \text{ has no parent}$$

The test $v[x] < 0$ remains a valid indicator that x is a root, but the value $v[x]$ now contains the height information: height of (x = root) is $-1 - v[x]$ (a non-negative value). Note that the Find and Root operation implementations require no changes.

Now we can define Link in such a way as to keep tree height from growing unless the two trees have the same height to begin with, and then the new tree has height one more than these:

```
void Link(x,y)
{
  if (v[x] > v[y])      // height at x < height at y
    v[x] = y;           // make y parent of x
  else if (v[x] < v[y]) // height at y < height at x
    v[y] = x;           // make x parent of y
  else                  // heights are the same
  {
    v[y] = x;           // make x parent of y
    --v[x];             // increase rank of x
  }
}
```

Attach the smaller of the two trees under the root of the larger, which does not change the height. Otherwise attach the second root under the first and increase height by one.

LEMMA 1. The height of a tree t in the Partition forest is $\leq \log_2 t$.size.

COROLLARY. The height of any tree in the forest $\leq \log_2 n$.

COROLLARY. The worst case runtime for Union, Find, and Root is $\leq O(\log_2 n)$.

NOTE. The runtime of Link is $\Theta(1)$.

LEMMA 2. Once a node in our forest ceases to be a root, it never returns to root status.

3 Union by Rank and Path Compression

The two public operations for the ADT are defined in terms of the structure-specific operations Link and Root, and Link has constant runtime. Therefore it makes sense to concentrate on improving Root. We will not be able to improve the worst case runtime, but we can improve the amortized worst case runtime to what is, in practical terms, a constant. The idea, like the others in this chapter, is remarkably simple: When following the path from x to the root, modify all parents in this path to be the root.

```
int Root(x) // path compression - modifies tree structure
{
  if (v[x] < 0) // x is a root
    return x;
  else
    return v[x] = Root(v[x]);  // recursively sets parent
}
```

It is unusual to have a Find [aka Search, Includes, ...] operation that cannot be programmed as a const method. (We use the previous non-path-compression version as a const method in the Partition code.) In fact, this morphing version of Root creates a potential problem: How do we maintain the correct height stored with the root of the tree as we make the tree shallower? The answer turns out to be (yes - simple): don't bother. We just change what we call it!

We define this value to be the *rank* of the root, and note several facts about rank:

LEMMA 3. For any root x, x.height $\leq x$.rank.

LEMMA 4. For any root x, x.rank $\leq \log_2(x.\text{size})$.

(See Exercises at end of chapter.)

Putting the lemmas together with discussion of the mechanisims, we now have proved the following best-possible analysis of the Partition algorithms:

THEOREM 1. For a Partition data structure with union-by-rank, with or without path compression, the worst case runtime of Union, Find, and Root is $O(\log n)$ and of Link is $O(1)$.

4 Amortized Runtime of Partition

Theorem 1 above gives a very fast, and tight, upper bound on the runtime of the Union/Find algorithms. But there is a much better result for amortized runtime. The latter is especially important due to the use cases for Partition data structures, which tend to make many calls, often proceeding from the initial partition of singletons to the terminal partition consisting of one set. Note that there are n calls to Union required to get from initial to terminal state of a partition, and the distinct ways to get there is much larger. (See Exercises.)

DEFINITION. $\log^* n$ is defined as the smallest integer k such that $\log^{(k)} n <= 1$, where $\log^{(k)}$ is the k-fold functional composition of log:

$$\log^* n = \min\{k | \log^{(k)} n <= 1\} \text{ , where}$$
$$\log^{(1)} n = \log n$$
$$\log^{(k)} n = \log \log^{(k-1)} n$$

In other words, $\log^* n$ is the number of times log must be applied to n to get to a value that is no greater than one.

THEOREM 2. For a Partition data structure with union-by-rank and path compression, the worst-case runtime of a sequence of m Partition operations is $O(m \log^* n)$.

COROLLARY. The amortized runtime of Partition operations is $O(\log^* n)$.

There are two proofs recommended. One using cost accounting is given below, and one using potential functions is given in the textbook. Quite a few variations of these may be found online.

4.1 Growth of $\log^* n$

A way to get intuition on $\log^* n$ is to realize that (1) it is monotonically increasing, and (2) the value changes with stacked exponents of 2. Define the stacked powers of 2 to be $P_0 = 1, P_1 = 2, \ldots, P_k = 2^{P_{k-1}}, \ldots$.

$$P_k = 2^{(2^{(2^{\cdots})})} \quad k \text{ times}$$

Then show

$$\log^{(k)}(P_k) = 1$$

by induction: $\log^{(k)}(P_k) = \log^{(k)} 2^{P_{k-1}} = \log^{(k-1)}(\log 2^{P_{k-1}}) = \log^{(k-1)}(P_{k-1}) = \ldots = \log P_1 = P_0 = 1$. That is, applying $\log$ k times to P_k yields 1. Therefore, by definition,

$$\log^*(P_k) = k$$

Note that

$$P_1 = 2^{P_0} = 2^1 = 2$$
$$P_2 = 2^{P_1} = 2^2 = 4$$
$$P_3 = 2^{P_2} = 2^4 = 16$$
$$P_4 = 2^{P_3} = 2^{16} = 65536$$
$$P_5 = 2^{P_4} = 2^{65536} = \text{a number too big to write in decimal notation}$$

and therefore the transitions in value occur at $1 = \log^* 2$, $2 = \log^* 4$, $3 = \log^* 16$, $4 = \log^* 2^{16}$, $5 = \log^* 2^{65536}$, ... These values represent the last point before the change occurs. In summary:

$$\log^*(x) = \begin{cases} 1 & \text{for } 1 < x \leq 2 \\ 2 & \text{for } 2 < x \leq 4 \\ 3 & \text{for } 4 < x \leq 16 \\ 4 & \text{for } 16 < x \leq 65536 \\ 5 & \text{for } 65536 < x \leq 2^{65536} \\ \geq 6 & \text{for } 2^{65536} \leq x \end{cases}$$

Note that 2^{65536} is approximately 10^{19660}: one followed by 19,660 zeros.[1] The number of atoms in the reachable universe is estimated to be 10^{80}, so we are unlikely to compute with such large numbers any time soon. Thus $\log^* n \leq 5$ for all useable values of n, and for all practical purposes we may consider it constant when it appears in an estimate of asymptotic runtime or runspace.

4.2 Tight Bound

It is known that the bound of Theorem 2 can be made tight at $\Theta(m\alpha(n))$, where α is the inverse of the Ackerman function. α grows even more slowly than $\log^* n$, but still is not mathematically constant. So while in practical terms we consider the union/find algorithms to have constant amortized runtime, as a matter of principle they do not. But it's close.

4.3 Proof of Theorem 2

This is left open for now. There are several readable versions available, including the textbook [uses Potential Functions], Mark Weiss's text[2] [uses cost accounting], and the papers cited outside the box.

---

[1] Using the approximation $2^n = (2^{10})^{n/10} \approx (10^3)^{n/10} = 10^{3n/10}$, derived from the observation that $2^{10} = 1024 \approx 1000 = 10^3$.

[2] Mark Allen Weiss, *Data Structures and Algorithm Analysis in C++* 3e, Addison Wesley, 2006.

5 Applications of Union-Find

5.1 Random Maze Generation

A nifty application of Partition is in generating a random maze. Students from Data Structures [fsu::COP4530] may recall a maze solving project (*The Rat* or *Rat Pack*) in which mazes of square cells are encoded and solved, with either depth-first or breadth-first search. One issue in those projects is the tedium of creating large maze files that are self-consistent. A maze file is *self-consistent* iff all adjacent pairs of cells agree on whether their common wall is up or down. More details of the maze coding system are given in an appendix.

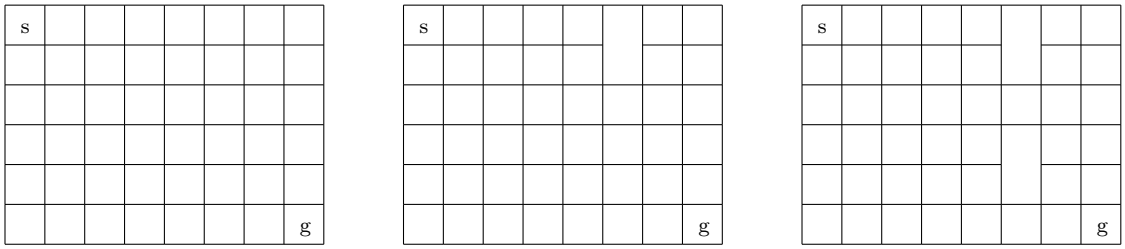
The Partition class facilitates a very efficient implementation of the random maze generating algorithm:

```
void RanMaze
{
  begin with a rectangular maze of all closed cells
  numrows = number of rows of cells;
  numcols = number of columns of cells;
  start = cell at (0,0);
  goal  = cell at (numrows-1,numcols-1);
  numcells = numrows * numcols;
  Partition p(numcells); // p represents the maze components
  while (!p.Find(start, goal)) // goal not reachable
  {
    select a wall at random;
    x, y = the two cells having this wall in common
    if (!p.Find(x,y))     // y not reachable from x
    {
      remove the wall from the maze;
      p.Union(x,y);      // x and y now in same component
    }
  }
}
```

Two cells in a maze are *mutually reachable* iff there is a path from one to the other. (And, of course, there is one in the other direction, since our mazes are not "directed".) Thus reachability can be answered using a maze solving routine, depth- or breadth-first search, to look for a path from one to the other. This would make the reachability question a $\Omega(n^2)$ algorithm in worst case.

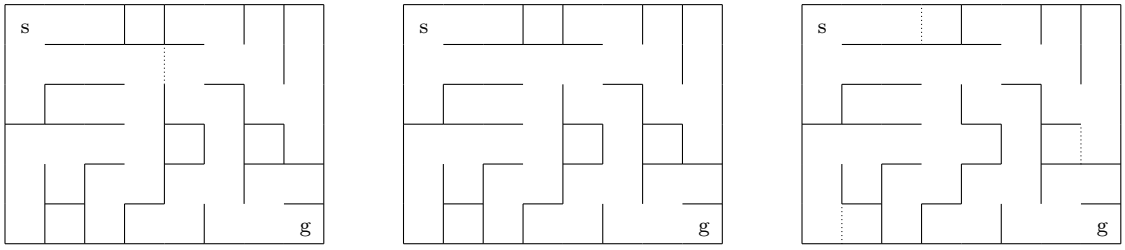
Partition reduces the reachability question to (almost) constant time on average. We maintain a partition on the universe of cells. Whenever a wall seperating two cells x and y is removed, we call $Union(x, y)$ to record that x and y are now in the same component of the partition. The reachability question for two cells x and y is $Find(x, y)$. Note that by replacing "find a path from x to y" by "is there a path from x to y", we reduce the worst case runtime of RanMaze from $\Theta(mn^2)$ to $\Theta(m\alpha(n))$, where m is the number of random walls selected. Again, FAPP, the runtime is $\Theta(m)$.

Here we show the first three states of an evolving 6×8 maze being generated by RanMaze, starting with the "all walls up" initial state followed by twice making a random selection of a wall that is removed:



Of course at this point we still have a ways to go before the goal g is reachable from the start s.

The next three mazes show the state after 40 or so steps along the process.



The first on the left is the last state in which s and g are not mutually reachable. Note that there is still a visually apparent separation between the two. The next wall randomly chosen is indicated as a dotted line. The call $Find(x, y)$, where x and y are the two cells with that wall in common, returns false, telling us that x and y are not mutually reachable, so the wall is removed, producing the maze in the center. That solution is refined by continuing to randomly select, test, and conditionally remove walls until all pairs are mutually reachable, yielding the maze on the right. The last three walls removed are shown as dotted lines.

Figure 1 shows a maze produced by RanMaze, stopped at the stage that cell 9,999 on the lower right corner is reachable from cell 0 on the upper left corner. The path depicted was found using depth-first search.

FIGURE 1. A 100×100 maze generated by RanMaze, terminated when the lower right cell 9999 is reachable from the upper left cell 0. At this stage there are still several inaccessible cells. The solution path shown was obtained using depth-first search.

5.2 Graph Algorithms

Some important graph algorithms need to keep up with connectivity questions similar to those for mazes, and Partition facilitates these. A famous example is Kruskal's algorithm for finding a minimum spanning tree in a weighted graph. The following is a more general pre-cursor to Kruskal:

Given an undirected graph $G = (V, E)$, a *spanning tree* is a subgraph $T = (V_T, E_T)$ such that

(1) The vertices of T are the same as those of G: $V_T = V$
(2) T is connected
(3) T contains no cycles

A spanning tree is a subgraph of G that is a connected acyclic and uses all the vertices of G. Note that if G has a spanning tree then G is connected - the spanning tree itself contains a (unique) path between any two vertices.

We can use the Partition data structure to assist in constructing a spanning tree for any connected graph: Begin by initializing T to the forest consisting of only the vertices: $F = (V, E_F = \emptyset)$. For each edge in $e \in E$: If e does not form a cycle with other edges in E_F, add the edge to E_F, thus joining two trees in the forest. We use a Partition to keep track of the connectivity of E_F. The process terminates when every edge has been considered.

```
Spanning Tree Algorithm [STA]
given: Graph     G = (V,E)
use:   Queue     Q of edges in E // in any order
       Partition p               // connectivity bookkeeping

initialize p(|V|)      // partition initialized to singletons
initialize F = empty   // forest initialized to vertices
for each edge e=[x,y] in Q
{
  // loop invariant: F contains no cycles
  if (!p.Find(x,y)) // x and y are in different trees in F
  {
    add e to F      // joins the two trees in F
    p.Union(x,y)    // records the join in the partition
  }
}
output (V,F)
```

THEOREM 3. IF G is a connected graph, the output of the Spanning Tree Algorithm is a spanning tree.

Proof. Clearly (1) holds, since we started with all vertices of `G`. We need to show that (2) `F` is connected and (3) `F` contains no cycles.

To see that `F` is connected, first consider one edge `e = [x,y]` in `G`. At the time `e` was investigated, `p.Union(x,y)` returned either true or false. If false, then `e` would have been added to `F` so that `x` and `y` are connected by `e`. If true, then there was already a path

connecting `x` and `y` in `F`. Either way, `x` and `y` are connected in `F`. If we apply this reasoning to every edge in a path, we conclude that every pair of vertices of the path is connected in `F`. Therefore `F` is connected, proving (2).

We next show the loop invariant is true. The base case is obvious, since `F` contains no edges. At the beginning of some execution of the loop body, assume the loop invariant is true. During the execution of the body, either `p.find(x,y)` returns true, in which case `F` is not modified, or `p.find(x,y)` returns false, in which case `e=[x,y]` is added to `F`. If the inclusion of `e` creates a cycle `C` in `F`$\cup\{$`e`$\}$, then `C - {e}` is a path in `F` connecting `x` and `y`, which would mean that `p.find(x,y)` returned true, a contradiction. Therefore the loop invariant is true at the end of execution of the loop body. By induction, the loop invariant is true for all executions of the body, proving (3). □

The spanning tree algorithm STA is an example of *greedy* algorithm due to the way the new edge is selected: take the first possible choice (i.e., the next edge in the ordering) and use it unless doing so would violate the desired outcome (i.e., create a cycle in `F`).

The runtime of the STA is estimated as follows (where n is the number of vertices and ARQ is the amortized runtime cost of queue operations)

Initialize `Q`	ARQ
Initialize `p`	$\Theta(\|V\|)$
Initialize `F`	$\Theta(1)$
process one edge	$\Theta(\alpha(\|V\|))$
get next edge	ARQ

Since the loop runs $|E|$ times, these components yield the overall estimate of $\Theta(|V| + |E|\alpha(|V|) + |E|ARQ)$. Finally, recall that in a connected graph $|V| \leq 1 + |E|$, so our final estimate is:

$$\text{STA Runtime} \quad = \Theta(|E|\alpha(|V|) + |E| \times ARQ)$$

which reduces to

$$\text{STA Runtime} \quad = \Theta(|E|\alpha(|V|))$$

if we make no costly assumptions about the queue operations. Note that, for all practical purposes, this means the STA runtime is $\Theta(|E| \times ARQ)$. We will encounter a better way to find spanning trees using breadth-first search, but the STA provides the best known way to find minimal spanning trees in a sparse network (Kruskal's algorithm).

5.3 Percolation Modeling

This and the next application I first learned about from notes of Robert Sedgewick and Kevin Wayne. The subject is *percolation* which is a general process in which one entity travels through a substrate.[3] Examples include the classic of water percolating through soil and the more modern electrons percolating through a metal lattice. S&W give this interesting table:

model	*system*	*vacant site*	*occupied site*	*percolates*
electricity	material	conducts	insulates	conducts
fluid flow	material	empty	blocked	porous
social interaction	population	person	empty	communicates

Percolation theory uses lattices to model the substrate and vertices or edges as sites which may be in one of at least two states (e.g., empty/full, open/blocked). The *percolation threshold* is "the critical fraction of lattice points that must be filled to create a continuous path of nearest neighbors from one side to another" - how many randomly selected sites must be unblocked for there to be a path through the lattice.[4] For many kinds of lattices this threshold has been calculated exactly. Surprisingly, finding a closed form for the percolation threshold of a square lattice like the one we use for mazes is still an open problem. The threshold is estimated using our maze model and simulation.

Start a simulation trial with a square maze initialized to the state in which all internal walls on the left-most and right-most columns of the maze are removed. The trial stops when there is a path from the left "foyer" to the right "back porch". The number of walls removed is the outcome. Repeating through a number of trials gives an estimate of the percolation threshold as the expected number of walls that must be removed to created a path.

5.4 The Game of Hex

Hex was invented independently by Piet Hein (1942), a student at Niels Bohr's Institute of Theoretical Physics and John Nash (1948), a graduate student in mathematics at Princeton[5], and subsequently produced commercially by Parker Brothers (1952). Read about this simple game with deep subtext in several web resources including Wolfram Math World and Wikipedia. Partition can be used to decide when a player has won a match.

[3] Weisstein, Eric W. "Percolation Theory." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PercolationTheory.html

[4] Weisstein, Eric W. "Percolation Threshold." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PercolationThreshold.html

[5] Nash is the mathematician depicted in the biographical movie *A Beautiful Mind*. He died in a limosine accident in 2015, just a few miles from his home in Princeton on the return trip from accepting the Abel prize from the Norwegian Academy of Sciences.

Exercises

1. Prove Lemma 1.
2. Prove Lemma 2.
3. Prove Lemma 3.
4. Prove Lemma 4.
5. Find an iterative re-factoring of Root without path compression.
6. Find an iterative re-factoring of Root with path compression.
7. Consider a universe of n objects $U = \{0, 1, 2, \ldots, n-1\}$, and define the initial state $\mathcal{A}$ to be the partition of singletons and the terminal state $\mathcal{Z}$ to be the partition consisting of one set, the universe: $\mathcal{A} = \{\{0\}, \{1\}, \{2\} \ldots \{n-1\}\}$ and $\mathcal{Z} = \{U\}$. (1) Show that exactly n Union operations are required to get from $\mathcal{A}$ to $\mathcal{Z}$. (Hint: show that a Union operation reduces the number of components by 1.) (2) Find the number of distinct sequences of Union operations that transform $\mathcal{A}$ to $\mathcal{Z}$.

Software Engineering Projects

8. Create `partition.h` and `ranmaze.cpp` in C++. `partition.h` contains the definition and implementation of the class template `Partition`. `ranmaze.cpp` contains the implementation of the RanMaze algorithm that produces random maze files using the file specification in the Appendix: Maze Technology.
9. Devise a method for constructing a graph model of a maze so that all questions of searching the maze translate to graph search questions. Then use standard graph algorithms to produce a maze solver.
10. Design a precise modeling environment including algorithms to estimate the percolation threshold for square lattices.

Appendix: Maze Technology

A *maze* is a rectangular array of square cells such that:

1. The exterior perimeter of the rectangle is solid (impenetrable).
2. Walls between cells may be designated as extant, meaning a barrier is in place, or non-extant, meaning passage between cells at that location is not restricted.
3. Cells are numbered consecutively from left to right and top to bottom, beginning with cell number 0 in the upper left corner.

A *maze problem* consists of:

1. A maze
2. A designated "start" cell
3. A designated "goal" cell

A *path* in a maze is a sequence of cell numbers such that adjacent cells in the sequence share a common face with no wall. A *solution* to a maze problem is a path in the maze from the start cell to the goal cell.

Maze File Syntax

A maze file is designed to describe a maze problem as defined above. The file consists of unsigned integers that may be in any format in the file but are typically arranged to mimic the actual arrangement of cells in the maze problem. Maze files have following file format:

```
numrows numcols
(first row of codes)
(second row of codes)
...
(last row of codes)
start goal
[optional documentation or other data may be placed here]
```

We refer to such a file as a maze file. (Note that the actual format is optional, for human readability. The file is syntactically correct as long as there are the correct number of (uncorrupted) unsigned integer entries.)

Maze File Semantics

A maze file is interpreted as follows:

1. `numrows` and `numcols` are the number of rows and columns, respectively, in the maze.
2. Each entry in a row of codes (after `numrows` and `numcols` but before `start` and `goal`) is a decimal integer code in the range [0..15]. The code is interpreted as giving the "walls" in the cell. We will refer to this as a *walls code*. A specific walls code is interpreted by looking at the binary representation of the code, with 1's bit = North face, 2's bit = East face, 4's bit = South face, and 8's bit = West face.[6] (Interpret "up" as North when drawing pictures.) A bit value of 1 means the wall exists at that face. A bit value of 0 means the wall does not exist at that face. For example walls code 13 means a cell with North, South, and West walls but no East wall, because $(13)_{10} = 8 + 4 + 1 = (1101)_2$ (the 2's bit is 0).
3. Cells are numbered consecutively from left to right and top to bottom, beginning with cell 0 at the upper left. (Thus, a 4x6 maze has cells 0,1,2,...,23.) We refer to these as *cell numbers*. Note that `start` and `goal` are cell numbers (NOT walls codes). They inform where the maze starting location is and where the goal lies.
4. The file must be self-consistent: pairs of adjacent cells must agree on whether their common face has wall up or down.
5. The optional data following goal is treated as file documentation (i.e., ignored).

Walls Code Details

There is a straighforward and self-documenting way of keeping up with walls codes in a program. Define the codes for the four directions, and then use these to test, build, and manipulate any walls code:

```
const unsigned char NORTH = 0x01;  // byte (00000001) cell with north wall only
const unsigned char EAST  = 0x02;  // byte (00000010) cell with east wall only
const unsigned char SOUTH = 0x04;  // byte (00000100) cell with south wall only
const unsigned char WEST  = 0x08;  // byte (00001000) cell with west wall only

15 = NORTH|EAST|SOUTH|WEST;        // closed box
13 = NORTH|SOUTH|WEST;             // cell open on east side
 6 = EAST|SOUTH;                   // cell open on north and west sides

cell &= ~EAST;                     // remove east wall
cell |= SOUTH;                     // add south wall
(cell & WEST == 0)                 // cell has no west wall
```

[6] The encoding of maze cells described here originated with Tim Budd [Timothy Budd, Classic Data Structures in C++, Addison Wesley, 1994].

Appendix: Code

```
class Partition
{
public:
  explicit Partition ( long size ); // create singletons

  void Union ( long x , long y );      { Link(Root(x),Root(y)); }
  bool Find  ( long x , long y )       { return Root(x) == Root(y); }
  bool Find  ( long x , long y ) const { return Root(x) == Root(y); }

  void Reset         ( );              // reverts to singletons
  void Reset         ( long newsize ); // starts over with new size
  void PushSingleton ()                { v_.PushBack(-1); }

  long Size       () const;            { return v_.Size(); }
  long Components () const;         // returns number of components
  void Display    ( std::ostream& os ) const; // display subsets
  void Dump       ( std::ostream& os ) const; // internal structure

private:
  long Root  ( long x );                  // path compression
  long Root  ( long x ) const;            // no path compression
  void Link  ( long root1 , long root2 ); // merge/union two trees
  fsu::Vector <long> v_;
};

Partition::Partition ( long size ) : v_((size_t)size,-1)
{}
```

Coding notes

1. We recommend the Partition class be a class template, just for the practical effect that unused methods will not be compiled, keeping the executable code leaner. The integer type used as vector elements can be the template parameter, with default value `long`.
2. The model discussed here takes some liberties with types, mixing signed and unsigned integers. In C++ this is tolerated for the most part. But the programmer should take care to explicitly cast signed ints to `size_t` before using them as vector indices (as in the constructor above).
3. Mixing signed and unsigned ints may produce enough difficulties in Java that the implementation needs to change slightly, making use of two arrays of unsigned int types. Use $v[x] == x$ to mean x is a root, and maintain the height/rank information in a second array.