11/30/15

Andrew Shirtz

CS480 – Fall 2015

An Implementation of a Hierarchical Addressing of the $\mathbb{A}_n^*$ Lattice

The $\mathbb{A}_n^*$ lattice is the image of $\mathbb{Z}^{(n+1)}$ projected onto a hyperplane that is orthogonal to the vector $(1, 1, \dots, 1)$. As a result of this relationship, we reference elements in $\mathbb{A}_n^*$ by their pre-images from $\mathbb{Z}^{(n+1)}$. It is important to note that this projection creates an equivalence relation for the elements of $\mathbb{R}^{(n+1)}$.

$x, y \in \mathbb{R}^{(n+1)}$

$x \sim y \Rightarrow \Phi(x) = \Phi(y)$

Where Φ is the projection from $\mathbb{R}^{(n+1)}$ to a hyperplane orthogonal to (1, 1, …,1)

With this equivalence in mind, it is acceptable and prudent to use a standard form for elements of $\mathbb{Z}^{(n+1)}$ which we will call a *Lattice Address.*

An element of $\mathbb{Z}^{(n+1)}$ is a *Lattice Address* if:

- No dimension of the element has a negative value
- The value of at least one dimension is 0

Conversion from an arbitrary element of $\mathbb{Z}^{(n+1)}$ to a lattice address is performed by the following procedure:

$x \in \mathbb{Z}^{(n+1)}, x = (x_0, x_1, \dots, x_n)$

$m \coloneqq x_0$

$\forall\, x_i \in x$

    If ($m < x_i$)

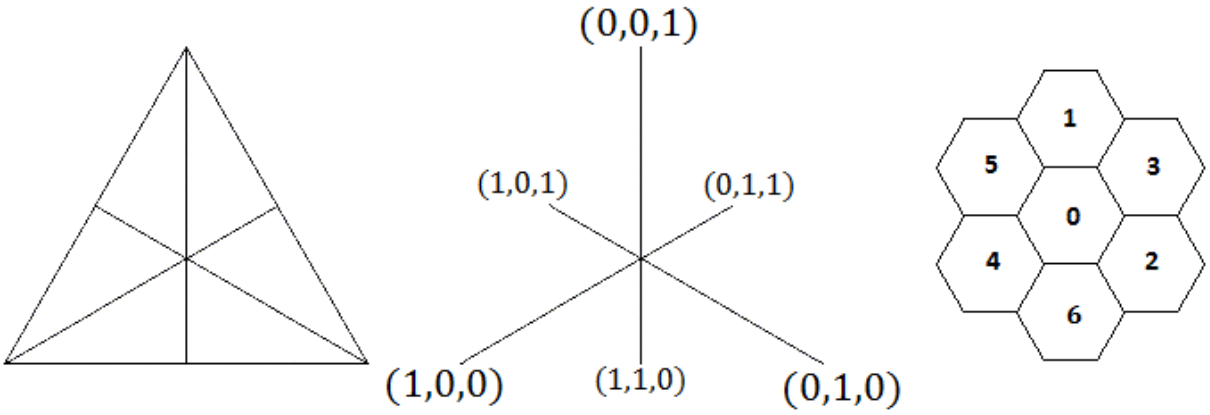
        $m \coloneqq x_i$

$\forall\, x_i \in x$

    $x_i \coloneqq x_i - m$

As the $\mathbb{A}_n^*$ lattice lies in $\mathbb{R}^n$, we must use a procedure to convert a vector in $\mathbb{R}^n$ to an element of $\mathbb{A}_n^*$. The paper *"An isomorphism between the p-adic integers and a ring associated with a tiling of N-space by permutohedra"* (Kitto, Vince, & Wilson; 1991) describes the $\mathbb{A}_n^*$ lattice with the following definition:

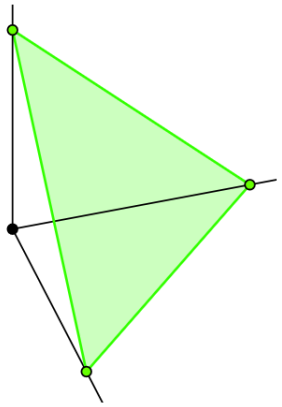
> "$\mathbb{A}_n^*$ is the lattice in $\mathbb{R}^n$ generated by the set of vertices $\{ v_0, \dots, v_n \}$ of a regular n-simplex with barycenter at the origin."



Based off this definition, elements of $\mathbb{A}_n^*$ can be thought of as any linear combination with integer coefficients of the vectors $\{ v_0, \dots, v_n \}$ described above. At first these two definitions may appear different, but if the "regular n-simplex" is taken to be the *standard n-simplex,* that is the simplex in $\mathbb{R}^{(n+1)}$ with the $(n+1)$ vertices listed below, and $\mathbb{R}^n$ is taken as the n -dimensional hyperplane orthogonal to (1, 1, …, 1), the two definitions are equivalent. Similarly, the equivalence relation stated above appears as linear dependence within this set of basis vectors.

$$\vec{e}_0 = (1, 0, \dots, 0)$$
$$\vec{e}_1 = (0, 1, \dots, 0)$$
$$\vdots$$
$$\vec{e}_n = (0, 0, \dots, 1)$$

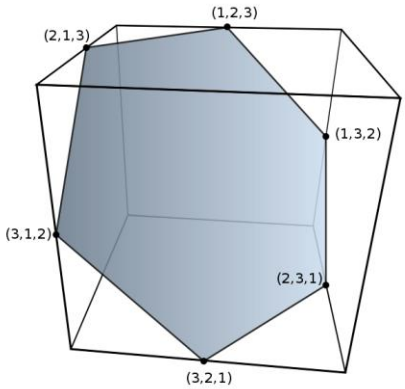
The *standard n-simplex*

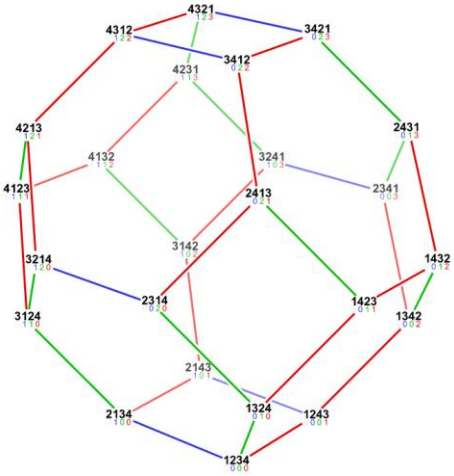


With this second definition as a starting point, converting from a vector in $\mathbb{R}^n$ to a lattice address in $\mathbb{A}_n^*$ is an application of Gaussian elimination to find the coefficients of the linear combination for the given vector. As there will be $(n+1)$ basis vectors, each corresponding to a vertex of the simplex, with which to describe the n elements of the given vector, one of the basis vectors can be excluded from the elimination arbitrarily. The n-tuple that results from the Gaussian elimination is converted to a vector in $\mathbb{R}^{(n+1)}$ by adding a dimension, corresponding to the excluded basis vector, to the n-tuple with the value of zero. The arbitrary exclusion of a basis vector is acceptable as the

resulting $(n+1)$-tuples are equivalent under the given relation. The $(n+1)$-tuple that results from the elimination does not resolve to an element of $\mathbb{A}_n^*$ as the entries of this tuple are most likely not integers.

The mapping from $\mathbb{R}^n$ to $\mathbb{A}_n^*$ amounts to a partitioning of space. To this end we use the *Voronoi Tessellation* of the $\mathbb{A}_n^*$ lattice with the effect of assigning representative elements from $\mathbb{A}_n^*$ to the resulting subsets of $\mathbb{R}^n$. In using this tessellation we also ensure that the points within the subset are closer, by the Euclidean metric, to their representative than to any other element of $\mathbb{A}_n^*$. As the spacing of the $\mathbb{A}_n^*$ lattice is regular, assignment of a vector to the appropriate Voronoi cell representative can be accomplished by rounding the elements of the $(n+1)$-tuple that resulted from the Gaussian elimination. The Voronoi cells generated by the elements of the lattice $\mathbb{A}_n^*$ are *Permutohedra* of order $(n+1)$. For example; $\mathbb{A}_2^*$ produces hexagons, $\mathbb{A}_3^*$ produces truncated octohedra.





At this moment we must take a moment to talk about *Permutohedra* before proceeding to the next section. A *permutohedra* is a $(n+1)$-dimensional polytope whose vertices are created by permuting the elements of the $(n+1)$-tuple $(1, 2, \ldots, n)$. The permutohedron of order $(n+1)$ lies entirely within the n-dimensional hyperplane defined by

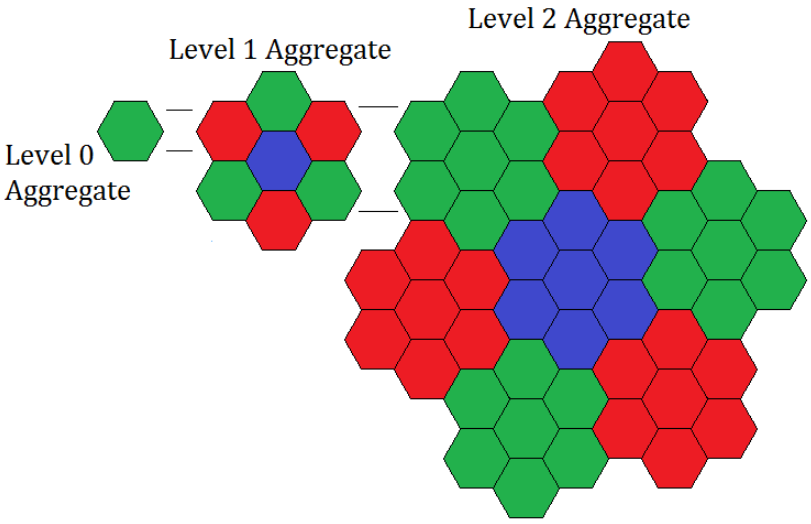
$$\{\vec{v}\,|v_0 + v_1 + v_{(n-1)} = \tfrac{(n+1)*(n+2)}{2}\}$$

This hyperplane is orthogonal to the vector (1, 1, …, 1). This hyperplane can be tiled by translated copies of the permutohedron. The barycenters of the permutohedra in this tiling are the elements of $\mathbb{A}_n^*$. This tiling can be described through *Aggregates.*

"*Aggregates* form a nested sequence of tessellations of n-space." - Kitto et al. (1991)

These structures are constructed inductively with the following process:

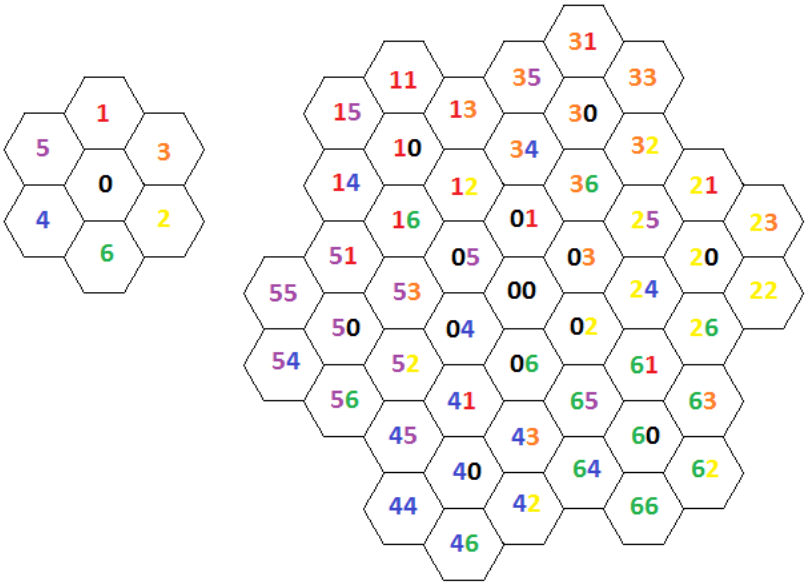
- The 0 index Aggregate is a single permutohedron, this can be taken as centered about the origin
- The $(k+1)^{st}$ index Aggregate is produced by surrounding the k^{th} index Aggregate with $2^{(n+1)} - 2$ translated copies of the k^{th} index Aggregate



We will hereby refer to the index of an aggregate as *Aggregate Level* for the remainder of this paper. Furthermore, from Kitto et al. (1991), aggregates have the following properties:

- For each $k = 0, 1, \ldots$
  - $\mathbb{R}^n$ is tiled by translated copies of the k^{th} aggregate
  - The $(k+1)^{st}$ aggregate is tiled by $2^{(n+1)} - 1$ translated copies of the k^{th} aggregate
  - Every element of $\mathbb{A}_n^*$ lies in some aggregate

Kitto et al. (1991) introduce the *Canonical Address* to reference an element of the $\mathbb{A}_n^*$ lattice. A *Canonical Address* is a sequence of elements from $\mathbb{Z}_{2^{(n+1)}-1}$. Each digit of the address denotes which of the $\left(2^{(n+1)} - 1\right)$ level $(k-1)$ aggregates, that make up the level k aggregate, the element of $\mathbb{A}_n^*$ lies in. The level $(k-1)$ aggregates are numbered as follows.



It should be noted that for level **1** Aggregates, the tile numbering lines up with the basis vectors of the $\mathbb{A}_n^*$ lattice; such that tiles numbered 2^0, 2^1, ..., 2^n correspond to lattice addresses (1, 0, …, 0), (0, 1, …, 0), …, (0, 0, …, 1) respectively.

Kitto et al. provide conversions between lattice addresses and canonical addresses; these procedures depend on a $(n+1) \times (n+1)$ circulant matrix $\boldsymbol{B}$.

$$\boldsymbol{B} = \begin{bmatrix} 2 & 0 & \cdots & 0 & -1 \\ -1 & 2 & \cdots & 0 & 0 \\ 0 & -1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & -1 & 2 \end{bmatrix}$$

This matrix takes a lattice address from the center of a level k aggregate to the center of a level $(k+1)$ aggregate. Similarly, $\boldsymbol{B}^{-1}$ takes a lattice address from the center of a level k aggregate to the center of a level $(k-1)$ aggregate. Converting from a canonical address to a lattice address uses the following procedure. Note that the examples given are for $\mathbb{A}_3^*$.

```
void convertTupleToLatAddr (Tuple *tup, LatAddr *outAddr)
{
    outAddr->a = (tup->val >> 0) & 1;
    outAddr->b = (tup->val >> 1) & 1;
    outAddr->c = (tup->val >> 2) & 1;
    outAddr->d = (tup->val >> 3) & 1;
    cleanupLatAddr(outAddr);
}

void convertCanAddrToLatAddr (CanAddr *cAddr, LatAddr *lAddr)
{
    lAddr->a = 0;
    lAddr->b = 0;
    lAddr->c = 0;
    lAddr->d = 0;

    LatAddr tmp, cpy;
    int i, j;

    for (i = MAX_CAN_ADDR_LEN-1; i >= 0; i--)
    {
        // Convert the given digit to a level 1 aggregate lattice address
        convertTupleToLatAddr (&(cAddr->addr[i]), &tmp);

        // Apply the 'B' matrix to the resulting lattice address
        // The transform is applied the same number of times as the significance of the digit
        //  Least significant digit: 0 Times
        //  Second least significant digit: 1 Time, etc.
        for (j = 0; j < (MAX_CAN_ADDR_LEN - (i+1)); j++)
        {
            cpy = tmp;
            applyBMatrixToLatAddr(&cpy, &tmp);
        }

        addLatAddr(lAddr, &tmp, lAddr);
    }
}
```

The conversion from lattice address to canonical address is more complicated; the procedure is shown below.

```
unsigned int produceCanAddr (LatAddr *inAddr, CanAddr *outAddr)
{
    Tuple cur;
    unsigned int i;

    for (i = 0; i < MAX_CAN_ADDR_LEN; i++)
    {
        outAddr->addr[i].val = 0;
    }

    i = 0;

    // While the lattice address is not equal to (0, 0, ...,0)
    while ((inAddr->a != inAddr->b || inAddr->a != inAddr->c || inAddr->a != inAddr->d) && i <= MAX_CAN_ADDR_LEN)
    {
        i++;

        // Set the value of the ith most significant digit to be:
        cur.val =  (inAddr->a + (2 * inAddr->b) + (4 * inAddr->c) + (8 * inAddr->d)) % 15;
        outAddr->addr[MAX_CAN_ADDR_LEN - (i)] = cur;

        // Decrement the appropriate dimensions of the lattice address
        inAddr->a = inAddr->a - (cur.val & 1);
        inAddr->b = inAddr->b - ((cur.val >> 1) & 1);
        inAddr->c = inAddr->c - ((cur.val >> 2) & 1);
        inAddr->d = inAddr->d - ((cur.val >> 3) & 1);

        int a = inAddr->a;
        int b = inAddr->b;
        int c = inAddr->c;
        int d = inAddr->d;

        // Apply the inverse B matrix
        inAddr->a = ((8*a) + (1*b) + (2*c) + (4*d))/15;
        inAddr->b = ((4*a) + (8*b) + (1*c) + (2*d))/15;
        inAddr->c = ((2*a) + (4*b) + (8*c) + (1*d))/15;
        inAddr->d = ((1*a) + (2*b) + (4*c) + (8*d))/15;
    }
    return i;
}
```

The possible values of a digit in a canonical address range from 0 to $2^{(n+1)} - 1$, which can be represented with $(n + 1)$ bits. In the second example included at the end of this paper, canonical addresses are implemented as bit-fields; access and mutation of the data is handled with bit shifting and masking.

With these definitions and behaviors in place, we can discuss the main topic of this paper; a space partitioning tree with which to store the elements of $\mathbb{A}_n^*$. With the relationship between canonical addresses and aggregates, a hierarchical structure is apparent. For the next section we will use the term *Tile* to refer to a leaf node of this tree which corresponds to a single element of $\mathbb{A}_n^*$; equivalently, a tile refers to a level 0 aggregate. It is important to note that tile and leaf node are not equivalent definitions, as nodes of the tree which correspond to aggregates whose level is greater than zero may have no children, and therefore meet the definition of leaf node. All tiles are leaf nodes, but not all leaf nodes are tiles. A major point to consider is that the hierarchical structure is built by aggregation instead of subdivision; this implies that branches of this tree cannot grow past a tile, increasing the depth of the tree requires that another level of the tree be added at the root.

While much of the behavior of the tree is context dependent, several pieces of information have appeared to be universally useful. Each node of the tree should store the following members; aggregate level and canonical address. Nodes that are not tiles will also store an array of $2^{(n+1)} - 1$ references to child nodes. For these non-tile nodes, the canonical address is truncated by filling in all digits less significant than the node's aggregate level with zeros. This is important as the canonical address of a non-tile node references the center of the corresponding aggregate. For an example, see the nearest neighbor search in the following section.

Two distinct implementations of this system have been produced, both of which will be discussed in detail. The first implementation was produced as an optimization to the *k Nearest Neighbors* algorithm, hereby referred to as *kNN*, for a research project conducted by Children's Hospital Los Angeles. The input data were sets of cell nuclei locations in three dimensions, having as many as 300,000 points each.

The basic design of the optimization is a limitation of the search space for candidate neighbors. The important design considerations for this implementation consists of two main parts, the need for an efficient nearest neighbor search and pre-existing sparse data. Creating an efficient nearest neighbor search was accomplished by selecting a representative element of $\mathbb{A}_3^*$ for each data point so as to store the data points in a 15-ary tree by canonical address. This allowed for entire branches to be discarded when searching for neighbors. Insertion of data points into the tree structure was lazily evaluated, which avoided the creation of unnecessary nodes. The implementations of both the search and insertion have been included at the end of this section.

Designing with these two considerations in mind, the runtime for a data set of 65,000+ points was 2 minutes as opposed to the 16+ hours of the naïve algorithm. Similarly, a dataset of 247,000+ points ran in 15 minutes; the naïve implementation ran for two weeks before the process was aborted.

```
void selectiveFlood (Node *curNd, unsigned int depth, LatAddr *center, unsigned int searchRadius, pFuncCB *callBack, void *params)
{
    // Found a leaf node with parked DPE, call naive kNN
    if (curNd->parkedDPE != NULL)
    {
        callBack (curNd, depth, params);
    }
    else
    {
        // Find Center of the aggregate of curNd
        LatAddr tileCenter;
        convertCanAddrToLatAddr (&(curNd->cAddr), &tileCenter);

        // Find distance between the data point and aggregate center
        scaleLatAddr (&tileCenter, -1);
        addLatAddr (&tileCenter, center, &tileCenter);

        double dist;
        dist = getLatAddrMagnitude (&tileCenter);

        // The radius of the aggregate is approximated as 3^(Aggregate Level)
        double tileRadius = pow (3.0, (double) (MAX_CAN_ADDR_LEN - depth));

        if ((dist + tileRadius) < searchRadius)
        {
            // This entire branch should be searched
            nodeTraverse (curNd, depth, params, callBack);
        }
        else if ((dist - tileRadius) > searchRadius)
        {
            // Do not search this branch at all
            return;
        }
        else
        {
            // Recursively call this function on all children
            int i;
            for (i = 0; i < 15; i++)
            {
                if (curNd->children[i] != NULL)
                {
                    selectiveFlood (curNd->children[i], depth+1, center, searchRadius, callBack, params);
                }
            }
        }
    }
}
```

This code above is the implementation of the nearest neighbor search.

DataPointEntry is a struct that holds; a Euclidean Vector from $\mathbb{R}^3$, a Lattice Address for $\mathbb{A}_3^*$, and a matching Canonical Address.

The function pointed to by *callback* calculates the distance between operand data point and the 'parked' DataPointEntry, and updates the list of closest neighbors if necessary.

```c
void nodeInsert (Node *curNd, DataPointEntry *dpe, unsigned int depth)
{
    if (depth > MAX_CAN_ADDR_LEN)
    {
        fprintf(stderr, "DataPoint Collision\n");
    }

    curNd->population += 1;

    // The new DPE is in a branch by itself, park the DPE and return
    if  (curNd->population == 1)    { curNd->parkedDPE = dpe; }

    // There is a parked DPE
    else if (curNd->population == 2)
    {
        // Continue insertion of the parked DPE
        allocateChild       (curNd, curNd->parkedDPE->cAddr.addr[depth], depth);
        nodeInsert          (curNd->children[curNd->parkedDPE->cAddr.addr[depth].val], curNd->parkedDPE, depth + 1);
        curNd->parkedDPE = NULL;

        // Continue insertion of new DPE
        allocateChild       (curNd, dpe->cAddr.addr[depth], depth);
        nodeInsert          (curNd->children[dpe->cAddr.addr[depth].val], dpe, depth + 1);
    }

    // Branch with other data points but no parked DPE, continue recursion on child denoted by the new DPE's CanAddr
    else
    {
        allocateChild   (curNd, dpe->cAddr.addr[depth], depth);
        nodeInsert      (curNd->children[dpe->cAddr.addr[depth].val], dpe, depth + 1);
    }
}
```

Above is the implementation of the lazily evaluated tree insertion, this function is called on every DataPointEntry that was created from Preprocessing.

```
anshirtz@euclid:~/final$ time ./target/release/Preprocess input/test_input.csv processed_output
Longest CanAddr: 11

real    0m0.126s
user    0m0.112s
sys     0m0.012s
anshirtz@euclid:~/final$ time ./target/release/kNearestNeighbors processed_output kNN_output

real    2m31.667s
user    2m31.500s
sys     0m0.064s
anshirtz@euclid:~/final$
```

```
anshirtz@euclid:~/final$ time ./target/release/Preprocess input/test_full_set.csv processed_output
Longest CanAddr: 11

real    0m0.369s
user    0m0.344s
sys     0m0.024s
anshirtz@euclid:~/final$ time ./target/release/kNearestNeighbors processed_output kNN_output

real    15m33.486s
user    15m32.584s
sys     0m0.228s
anshirtz@euclid:~/final$
```

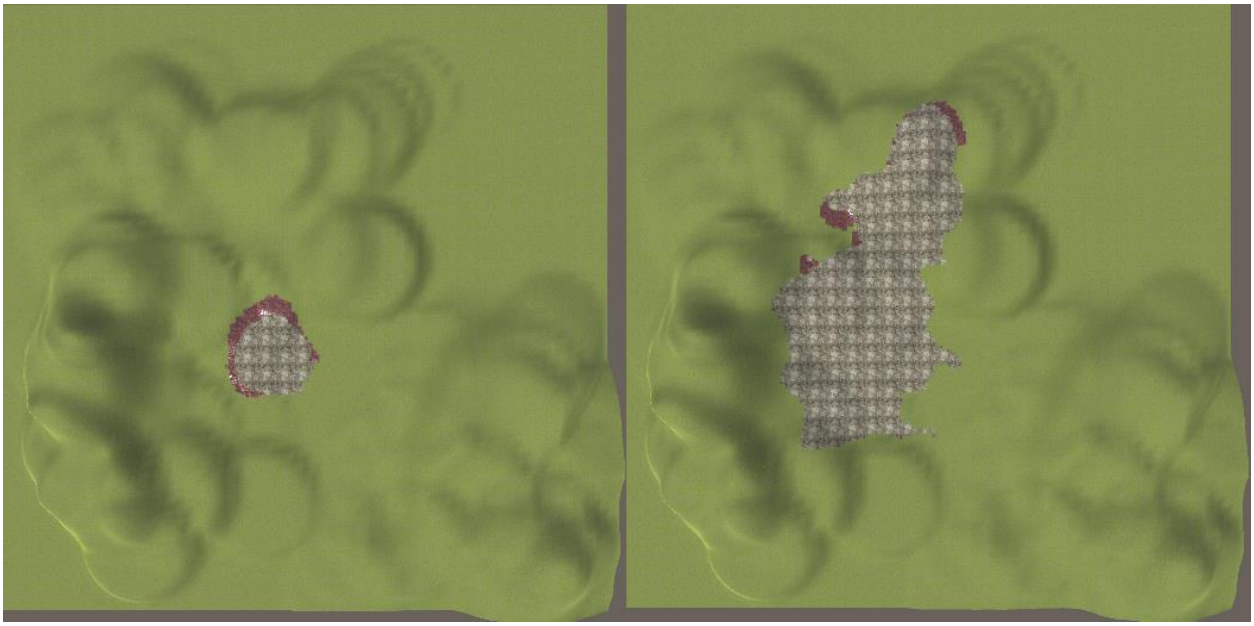
Runtimes for a set of 65,000 points and a set of 247,000 points. Preprocessing creates a DataPointEntry for each data point, in doing so it calculates the canonical addresses and prints the length of the longest canonical address, which is necessary as we must know the tree depth before insertion.

The second implementation of this system was for a forest fire propagation simulation that was created for the Microsoft Alpha Game Jam 2015. This simulation used hexagonal cellular automata to store the conditions in small areas of the forest. As each cell of the simulation is a hexagon, we use the $\mathbb{A}_2^*$ lattice to store and access these hexagonal areas. The design considerations for this system included; lazy branch creation, saving of tile state, and branch destruction. These considerations are all related in that we want to avoid creating or keeping unnecessary nodes or tiles, but also be able to save the state of a tile. This is important so that we do not have tiles that exhaust their fuel and burn out, only to be reignited later. Simply put, tiles that have burned should not burn again. As a side note; when ignited, a tile retrieves references to its neighbors; when extinguished, these references are removed.

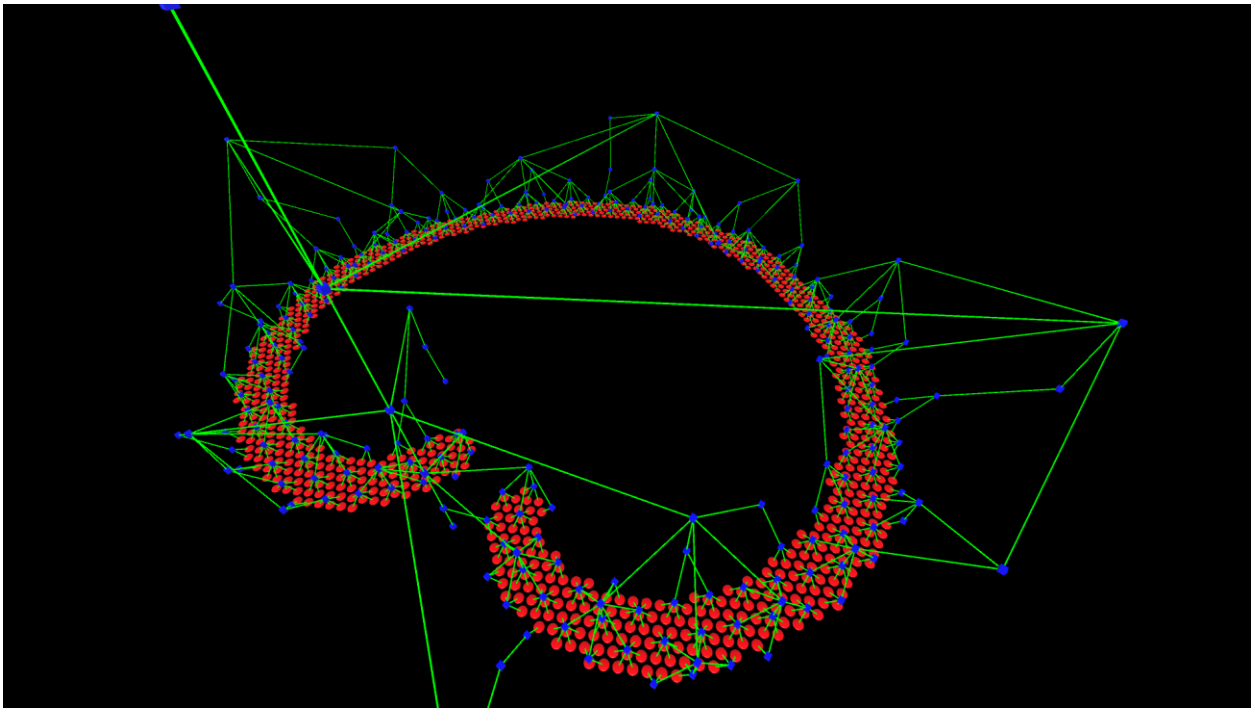
The implementation of lazy branch creation here is different from the lazy data insertion of the previous example; in this system a branch is created with full length, that is to say that it terminates with a tile. Retrieving references to tiles is handled through a method to be called on the root node, the important parameter to which is the canonical address of the tile to be returned. As the tree is traversed to access the tile requested, nodes along the path are created if they do not yet already exist.

Tiles can have the following states; untouched, on fire, burned, wetted, or incombustible. These states comprise a bit-field, stored in a single byte. Non-tile nodes have an array of these bit-fields, one for each child. When a tile changes state; either from warming by adjacent fires, exhausting their fuel supply, or by external events; the tile notifies its parent who then records the tile's state. As a secondary action, the parent node compares the state of all its children. If the state of each child is identical, the node then notifies its parent, continuing up the branch as necessary. The utility of this process will become apparent in the next section.

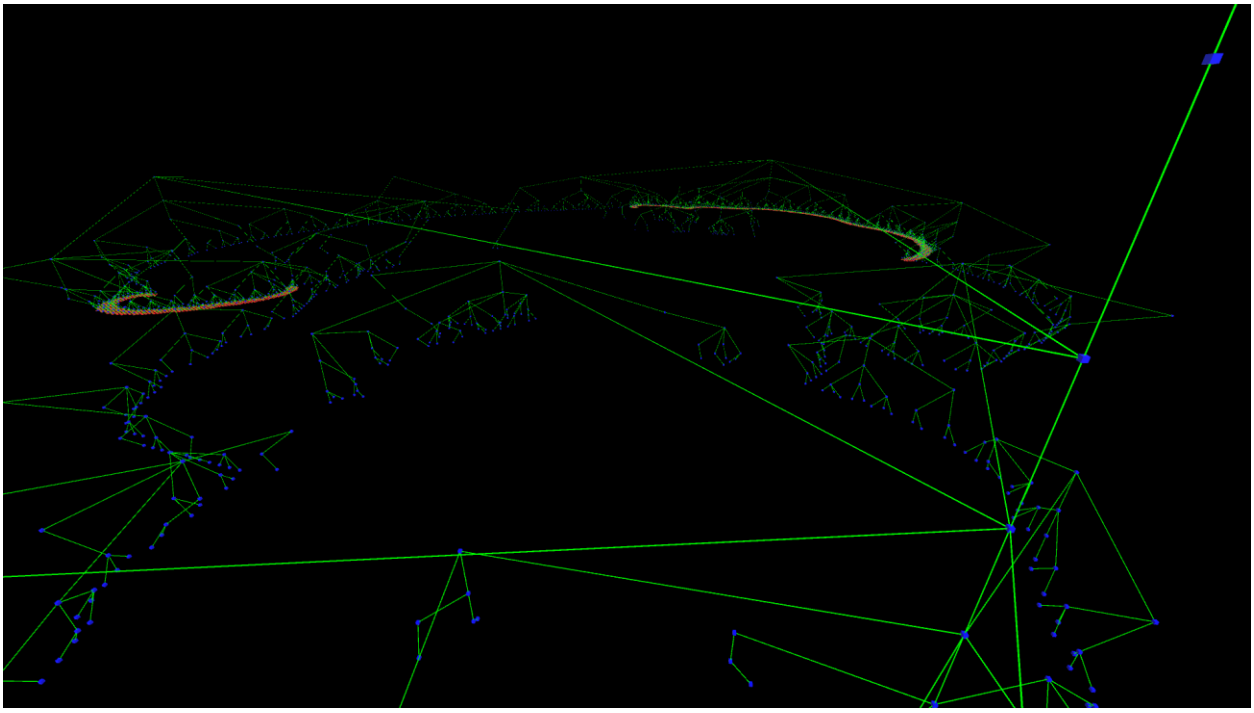
A tile need no longer exist if it is not actively on fire and it is not being referenced by a tile that is actively on fire. If neither of these conditions are met, the tile is available to be freed and it does so by notifying its parent node. The parent node then removes the reference for the tile from its collection of children, prompting the child to be garbage collected. The parent node then checks if it has a reference to any living child, and checks if its children have heterogeneous state. If neither of these conditions are met, the node itself is unnecessary and notifies its parent that it should be removed. With this process, entire branches of the tree can be removed if the area represented by the branch is not actively on fire and has homogenous state.



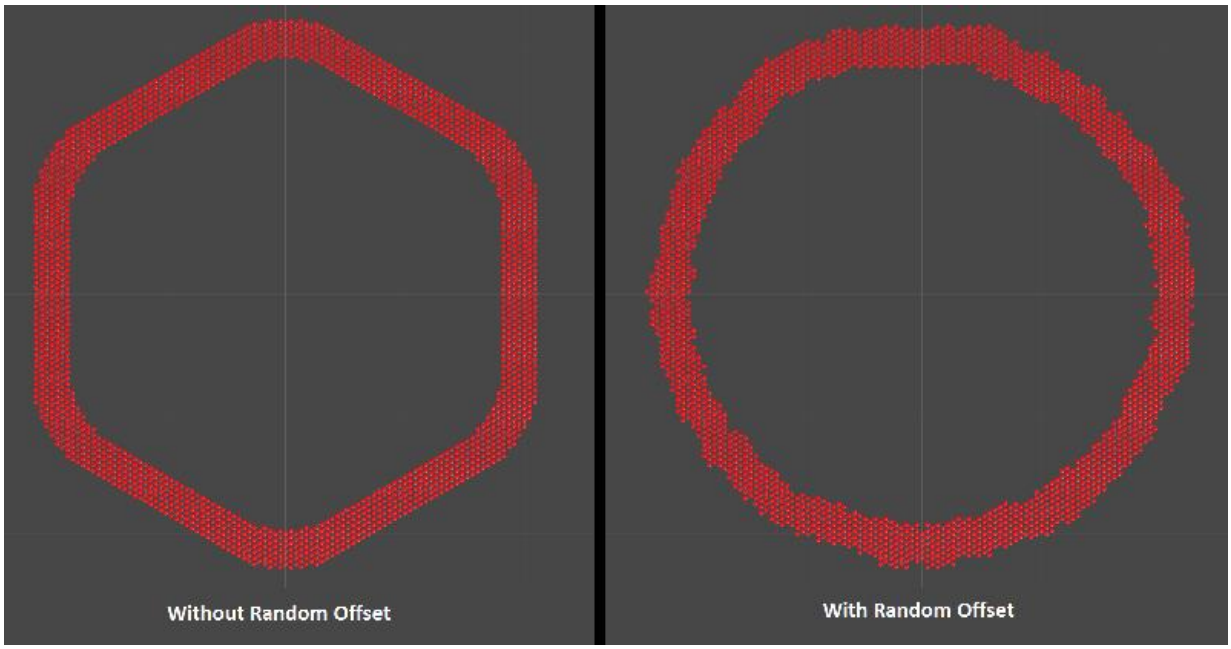
Two images of the original fire simulation. Red areas are on fire, gray areas have been burned. An area of the terrain's texture is changed when the tile for that area changes state.



This is a visualization of the tree structure. Blue cubes are interior nodes. Red circles are tiles. Each tile has fuel, elevation, and temperature. If on fire, a tile will heat its neighbors based on the following criteria; the tiles temperature, distance to neighbor, elevation difference, and wind direction.



Another image of the visualizer. Note the lack of low level branches in the center of the image. The nodes and tiles for this area have been freed due to the area's homogeneous state.



While not directly related to the addressing scheme, the above image shows an improvement to the accuracy of the model. In the image on the left, the distances between tiles are always calculated from the tile center. In the image on the right, each tile is assigned a random vector from within the unit circle which is included in the distance calculation.