Dynamically Linking Stochastic Database That Links Entries Based on Similarity

Salvatore Skare

June 2016

0.1 Introduction

According to the third edition of Artificial Intelligence A Modern Approach, "the - 'knowledge bottleneck' in Al—the problem of how to express all the knowledge that a system needs—may be solved in many applications by learning methods rather than hand-coded knowledge engineering" (Russell, Norvig, 2010, p. 28). AI agents that are capable of learning about their environment instead of having a static set of data are vastly more flexible and adaptable. The challenge then, is to find a way to store learned information that is easy for intelligent agents to draw connections and make inferences from. Computer systems are very good at storing value pairs, that is a variable and a value, but this method creates disjointed data-points that the programmer then has to keep track of and figure out what to do with. With the advent of object oriented programming languages, some of these data can be sorted and grouped into objects, which can then preform operations on it. While better than value pairs, objects are still a long ways away from being automatic and must be tailored for specific data and operations, which does not scale upwards to an environment with many unforeseen scenarios that must be learned. A better system would be one that mimics how humans and other mammals store memories, as a collection of overlapping mental prototypes and concepts.

0.2 Specification

In psychology, concepts are characteristics that define a group of related items or ideas. Although the specifics of how memory works remain unknown, the basic high level mechanic is that the brain takes new information and separates it into chunks. These chunks are then added to concept groups by comparing them with mental prototypes of the concepts (Myers, 2013, ch. 8). This process can be replicated on a basic level in software. To achieve this, I started with the following two structures:

```c
typedef struct input input;
typedef struct input  //input struct, stores the individual inputs in a linked list.
{
    void* data;        //actual input data, can be anything
    size_t dataSize;   //size of the input data
    long int link;     //link to the most similar memory, entered by linkInput function
    float confidence; //confidence of the link, added by linkInput function
    input* next;       //pointer to next input in the linked list
};
typedef struct memory memory;
typedef struct memory           //memory struct, stores input lists
{
    long int uuid;              //UUID for the memory, just counts up from 0
    input* inputs[NUMINPUTS]; //pointers to input lists
    memory* next;               //pointer to next memory in liked list
} memory;
```

The *memory* struct acts as a wrapper for the inputs it contains, chained together in a linked list. The inputs are then referenced in each memory as an array of linked lists. Each input contains the UUID of another memory that is similar to it, determined by a function called *linkInput*:

```c
/* Parses over the database and links the input pattern. The database arg is the database to be searched, and the type is the index of the input pattern. */
void linkInput(input* pattern, int type, memory* database)
{
    float cost = 0, lastCost = 0, simConf = 0;
    memory* next = database;
    long int steps = 0, mostSim = -1;
    do
    {
        float matchprob = 0;
        input* currInput = next->inputs[type];
        input* tmpSim;
        while(currInput != NULL) //iterate over all inputs in current memory
        {
            float similarity = compareInputs(pattern, currInput, type); //compare them with the pattern input
            if(similarity > matchprob) //if the input is the most similar of all inputs in the memory so far, save it
            {
                matchprob = similarity;
                tmpSim = currInput;
            }
            input* tmp = currInput->next;
            currInput = tmp;
        }
        if(matchprob > simConf) //if the current similar is the most similar so far, save it
        {
            mostSim = next->uuid;
            simConf = matchprob;
        }
        lastCost = cost;
        steps++;
        cost = steps / (matchprob + 1); //calculate algorithm cost
        if(matchprob > BRANCH_LIMIT) //if the current memory is more similar than the BRANCH_LIMIT, go to the linked memory instead of iterating linearly
        {
            memory* loop = next;
            while(loop->uuid > tmpSim->link)
            {
                memory* tmp = loop->next;
                loop = tmp;
                if(loop == NULL)
                {
                    break;
                }
            }
            next = loop;
        }
        else //iteralte over memory list linearly
        {
            memory* tmp = next->next;
            next = tmp;
        }
    } while(cost * STOP_LIMIT > lastCost && next != NULL); //check stop conditions
    pattern->link = mostSim; //link input to the most similar one found
    pattern->confidence = simConf;
    return;
}
```

As you can see, the *linkInput* function calculates a cost of continuing, and compares it to *STOP_LIMIT*, so if the database grows very large in size, it isn't traversed needlessly to its end. Also, each similarity with the current memory is compared to *BRANCH_LIMIT*, and if it's greater than that constant, the next pointer is incremented until it reaches the UUID of that memory instead of traversing linearly, decreasing the run-time of the function. As demonstrated later in section 0.4, the run-time can be further reduced by taking advantage of parallel computing.

In another header file, two functions dealing with inputs, *compareInputs* and *getInput*, are defined. Each of these take a variable *type* as an argument and use it to determine which input to act upon. Because the inputs will differ from agent to agent depending on the environment, these two functions need to be re-written for each application, although the declarations remain the same. The *getInput* function returns a pointer to a newly malloc'd input, with its *data* variable pointing to the raw data that is *dataSize* large. It also calls *linkInput* on the new input, so it is fully linked before it is returned as well. The *compareInputs* function takes two inputs and compares them, returning a float between 0 and 1 that represents numerically how similar they are, with 1 being 100% similar and 0 being 0% similar.

The linking of inputs to other memory groupings is what sets this structured data representation apart from a simple linked list. These links build up relationships between disparate connected data, allowing the agent to easily retrieve concepts from the database. This is done through the *compileMem* function, which creates a compilation of connected memories based on an input:

```
/* Compiles a new memory list, int levels deep, based on what
    pattern is linked to. Dataset should point to the database, and
     list should point to a NULL memory pointer, that the composite
     will eventually be stored in. */
void compileMem(input* pattern, memory* dataset, memory** list, int
     levels)
{
    if(levels == 0)
    {
        return;
    }
    if(list == NULL)
    {
        return;
    }
    memory* dataList = newMemory(*list);
    *list = dataList;
    memory* loop = dataset;
    while(loop->uuid > pattern->link) //jump to the linked memory
    in pattern
    {
        memory* tmp = loop->next;
        loop = tmp;
        if(loop == NULL)
        {
            return NULL;
        }
    }
    for(int i = 0; i < NUMINPUTS; i++) //copy the input data into a
     new list
    {
        input* parser = loop->inputs[i];
        while(parser != NULL)
        {
            input* newInput = malloc(sizeof(input));
            void* newData = malloc(parser->dataSize);
            memmove(newData, parser->data, parser->dataSize);
            memmove(newInput, parser, sizeof(input));
            newInput->data = newData;
            addInput(dataList, newInput, i);
            input* tmp = parser->next;
            parser = tmp;
        }
    }
    for(int i = 0; i < NUMINPUTS; i++) //iterate over all inputs in
     the memory
    {
        input* parser = loop->inputs[i];
        while(parser != NULL)
        {
            compileMem(parser, loop, list, levels-1); //recurse
    until levels reaches 0
            input* tmp = parser->next;
            parser = tmp;
        }
    }
    return;
}
```

This is a recursive function, that takes an argument, *levels*, to limit its recursion depth. Starting with the *input *pattern*, the function compiles a list of all the linked memories, and the memories linked by the inputs in those memories, etc. up until *levels* deep. The returned composite memory will have the most relevant, similar memories at the beginning and, as it gets bigger, the more tangentially related memories farther down the list. The next example demonstrates how this can be used to store data and generate concepts.

0.3 Example 1

The following example has two inputs, an image of a simple shape of a basic color, and text describing it, such as "blue oval". For image processing, the OpenCV library is used. Images are compared by taking an average of FLANN feature matching and a simple histogram. Text is just compared based on word frequency. One or both inputs can be given to the database on each iteration of the loop. If only one input is specified, it is added to the database, and used to create a composite memory one level deep. The first input of the other type from this composite is then displayed on the screen.

```
while(true)
 {
     //get inputs
     input* text = getInput(0, database);
     printf("\n");
     input* image = getInput(1, database);
     printf("\n");
     if(text == NULL && image == NULL)
     {
         //exit the program
         break;
     }
     else if(text == NULL && image != NULL)
     {
         //outputting text based on image
         memory* composite = NULL;
         compileMem(image, database, &composite, 1);
         database = AddtoMem(image, 1, database, NULL);
         input* outLoop = composite->inputs[0];
         while(outLoop != NULL)
         {
             printf("%s\n", (char *)outLoop->data);
             input* tmp = outLoop->next;
             outLoop = tmp;
         }
         disassemble(composite);
     }
     else if(text != NULL && image == NULL)
     {
         //outputting image based on text
         memory* composite = NULL;
         compileMem(text, database, &composite, 1);
         database = AddtoMem(text, 0, database, NULL);
         input* outLoop = composite->inputs[1];
         while(outLoop != NULL)
         {
             int rows, cols, type;
```

```
                size_t step;
                void* data = malloc(outLoop->dataSize - sizeof(int)*3+sizeof(size_t));
                memmove(&rows, outLoop->data, sizeof(int));
                memmove(&cols, outLoop->data+sizeof(int), sizeof(int));
                memmove(&type, outLoop->data+sizeof(int)*2, sizeof(int));
                memmove(&step, outLoop->data+sizeof(int)*3, sizeof(size_t));
                memmove(data, outLoop->data+sizeof(int)*3+sizeof(size_t), outLoop->dataSize - sizeof(int)*3+sizeof(size_t));
                Mat img(rows, cols, type, data, step);
                namedWindow( "Image", WINDOW_AUTOSIZE );
                imshow("Image", img);
                waitKey(0);
                input* tmp = outLoop->next;
                outLoop = tmp;
            }
            disassemble(composite);
    }
    else if(text != NULL && image != NULL)
    {
        //save both inputs and move on
        database = AddtoMem(text, 0, database, NULL);
        database = AddtoMem(image, 1, database, NULL);
    }
}
```

A simple database, with the inputs "red circle", "green square", "red triangle", "blue triangle", and "yellow circle" with the respective images would look like this:

```
|-mem:4----------|
|inputs:         |
|   input0 [link:0 confidence:0.500000],
|   input1 [link:2 confidence:0.269749],
|----------------|
     |
     |
     |
|-mem:3----------|
|inputs:         |
|   input0 [link:2 confidence:0.500000],
|   input1 [link:2 confidence:0.819149],
|----------------|
     |
     |
     |
|-mem:2----------|
|inputs:         |
|   input0 [link:0 confidence:0.500000],
|   input1 [link:1 confidence:0.576634],
|----------------|
     |
     |
     |
|-mem:1----------|
|inputs:         |
|   input0 [link:-1 confidence:0.000000],
|   input1 [link:0 confidence:0.336166],
|----------------|
     |
     |
     |
|-mem:0----------|
|inputs:         |
|   input0 [link:-1 confidence:0.000000],
|   input1 [link:-1 confidence:0.000000],
|----------------|
     |
     |
     |
  |----|
  |NULL|
  |----|
```

Note the >80% image link confidence between the second triangle and the first, the highest in the database. Because each entry only uses two words, the only possible values for the text confidence are 0%, 50%, and 100%. Now, to see if the database has learned any concepts from these inputs, we can give it an input of a green oval, a shape it has never seen before. When I do this, the output is, ”green square” as that is the only green shape in the database. The database is successfully able to create a prototype for the color green, and categorize never before seen shapes into it based on color. Likewise, if I only enter ”triangle”, the program displays a blue triangle. This example is not intelligent of course, it merely recalls information based on similarity, but this same concept can be extended to give an intelligent agent a way to store and recall data in a meaningful way.

0.4 Example 2

In this example application, I integrated the database with a neural net to control an autonomous robot. With a recurrent neural network, a short-term memory can be simulated (Russell, Norvig, 2010, p. 729), but by placing data retrieved from *compileMem* into the input neurons of a neural network we can simulate more permanent memory storage. For the robot program, the inputs are an image from a front facing camera, a distance obtained from an ultrasonic sensor on the front, light levels read from three photoresistors, and a score indicating how well the current configuration preformed in the environment. The goal of the robot was to avoid obstacles and seek out sources of light, simulating an application of solar recharging.

The robot itself ran very little of the actual code, instead it would send inputs wirelessly to a server, which would then process them. All inputs except for the image were normalized to a value between zero and 1, and would be given to the first input neurons. The remaining input neurons were populated with data from a compiled memory based on the image input. This way, the neural net would not only have current data to base decisions on, but also remembered data and the resulting scores. The output of the net was two values that corresponded to two sets of three actions, the first being reverse, stay still, or go forwards, and the second being turn left, don’t turn, or turn right. The resulting set of actions was sent to the robot which would preform them, calculate a score, and send it back to the server. The server would then add all current inputs to the database and go back to waiting for the robot to send more inputs.

The neural net was created using the FANN library. The net was first trained with basic light seeking and obstacle avoiding behaviors using the RPROP training algorithm. Once the program had begun running, the weights in the net were adjusted by a simulated annealing algorithm, to further optimize towards better performance as measured by the score returned from the robot.

Image processing was once again accomplished with the OpenCV library. In order to speed up image comparison, the task was distributed among a Beowulf cluster using OpenMPI. A modified version of the *linkInput* was created to take advantage of parallel computing:

```
void p_linkInput(input* pattern, int type, memory* database, int
    world_size)
{
    if(world_size < 2) //need more than 1 host
    {
        linkInput(pattern, type, database);
        return;
    }
    MPI_Datatype inputBuffer;
    MPI_Type_contiguous((int)sizeof(input), MPI_BYTE, &inputBuffer)
    ;
    MPI_Type_commit(&inputBuffer);
    float cost = 0, lastCost = 0, simConf = 0;
    memory* next = database;
    int index = 1;
    long int steps = 0, mostSim = -1, tmpuuid;
    input* inputList[world_size];
    long int uuidList[world_size];
    //zero input list
    for(int i = 0; i < world_size; i++)
    {
        inputList[i] = NULL;
    }
    do
    {
        float matchprob = 0;
        //select a batch of world_size inputs
        input* currInput = next->inputs[type];

        input* tmpSim;
        while(currInput != NULL)
        {
            inputList[index] = currInput;
            uuidList[index] = next->uuid;
            if(index == world_size)
            {
                float similarity[world_size];
                MPI_Request handles[world_size];
                //now that we have gathered enough values, send them to nodes
                for(int i = 1; i < world_size; i++)
                {
                    MPI_Send(pattern, 1, inputBuffer, i, 0, MPI_COMM_WORLD);
                    MPI_Send(pattern->data, pattern->dataSize, MPI_BYTE, i, 0, MPI_COMM_WORLD);
                    MPI_Send(inputList[i], 1, inputBuffer, i, 0, MPI_COMM_WORLD);
                    MPI_Send(inputList[i]->data, inputList[i]->dataSize, MPI_BYTE, i, 0, MPI_COMM_WORLD);
                    MPI_Send(&type, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                    MPI_Irecv(&similarity[i], 1, MPI_FLOAT, i, 0, MPI_COMM_WORLD, &handles[i]);
                }
                for(int i = 1; i < world_size; i++)
                {
                    //now wait for all those things to return
                    MPI_Wait(&handles[i], MPI_STATUS_IGNORE);
                    if(similarity[i] > matchprob) //if the input is the most similar of all inputs in the memory so far, save it
                    {
                        matchprob = similarity[i];
                        tmpSim = inputList[i];
                        tmpuuid = uuidList[i];
                    }
                }
                index = 0;
                //zero input list
                for(int i = 0; i < world_size; i++)
                {
                    inputList[i] = NULL;
                }
            }
            index++;
            input* tmp = currInput->next;
            currInput = tmp;
        }
        if(matchprob > simConf) //if the current similar is the most similar so far, save it
        {
            mostSim = tmpuuid;
            simConf = matchprob;
        }
        lastCost = cost;
        steps++;

        cost = steps / (matchprob + 1); //calculate algorithm cost
        if(matchprob > BRANCH_LIMIT && index == 1) //if the current
```

```
     memory is more similar than the BRANCH_LIMIT, go to the linked
     memory instead of iterating linearly
        {
            memory* loop = next;
            while(loop->uuid > tmpSim->link)
            {
                memory* tmp = loop->next;
                loop = tmp;
                if(loop == NULL)
                {
                    break;
                }
            }
            next = loop;
        }
        else //iterate over memory list linearly
        {
            memory* tmp = next->next;
            next = tmp;
        }
    } while(cost * STOP_LIMIT > lastCost && next != NULL); //check
    stop conditions
    //check to see if there are any leftovers that haven't been
    processed yet
    if(inputList[1] != NULL)
    {
        int remainderSize;
        for(int i = 1; i < world_size; i++)
        {
            if(inputList[i] == NULL)
            {
                remainderSize = i;
                break;
            }
        }
        float similarity[remainderSize];
        MPI_Request handles[remainderSize];
        //now that we have gathered enough values, send them to
    nodes
        for(int i = 1; i < remainderSize; i++)
        {
            MPI_Send(pattern, 1, inputBuffer, i, 0, MPI_COMM_WORLD)
    ;
            MPI_Send(pattern->data, pattern->dataSize, MPI_BYTE, i,
     0, MPI_COMM_WORLD);
            MPI_Send(inputList[i], 1, inputBuffer, i, 0,
    MPI_COMM_WORLD);
            MPI_Send(inputList[i]->data, inputList[i]->dataSize,
    MPI_BYTE, i, 0, MPI_COMM_WORLD);
            MPI_Send(&type, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            MPI_Irecv(&similarity[i], 1, MPI_FLOAT, i, 0,
    MPI_COMM_WORLD, &handles[i]);
        }
        for(int i = 1; i < remainderSize; i++)
        {
            //now wait for all those things to return
            MPI_Wait(&handles[i], MPI_STATUS_IGNORE);
            if(similarity[i] > simConf) //if the input is the most similar of all inputs in the memory so far, save it
            {
                mostSim = uuidList[i];
                simConf = similarity[i];
            }
        }
    }
    pattern->link = mostSim; //link input to the most similar one found
    pattern->confidence = simConf;
    return;
}
```

The non-root nodes all run the following function on startup:

```
void p_compare()
{
    int flag = 0;
    while(true)
    {
        while(!flag)
        {
            MPI_Iprobe(0, 0, MPI_COMM_WORLD, &flag, MPI_STATUS_IGNORE);
        }
        int number_amount;
        input* input1 = (input*)malloc(sizeof(input));
        input* input2 = (input*)malloc(sizeof(input));
        MPI_Datatype inputBuffer;
        MPI_Type_contiguous((int)sizeof(input), MPI_BYTE, &inputBuffer);
        MPI_Type_commit(&inputBuffer);

        MPI_Recv(input1, 1, inputBuffer, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        input1->data = malloc(input1->dataSize);
        MPI_Recv(input1->data, (int)input1->dataSize, MPI_BYTE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        MPI_Recv(input2, 1, inputBuffer, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        input2->data = malloc(input2->dataSize);
        MPI_Recv(input2->data, (int)input2->dataSize, MPI_BYTE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        int type;
        MPI_Recv(&type, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        float similarity = compareInputs(input1, input2, type);
        MPI_Send(&similarity, 1, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
        free(input1->data);
        free(input1);
        free(input2->data);
        free(input2);
    }
}
```

Instead of processing everything serially, this method gathers up a list of inputs, then distributes them among the cluster to compare, selecting the most similar out of the list, and going forward from there.

0.5 Future Improvements

There are two improvements that I would have liked to add to the database and that I believe would greatly improve its performance in the future. The first of which would be to add the ability to cache sections of the database to disk, instead of keeping it all in memory. In only the first several days in which I ran the robot program, its database ballooned to over 2 gigabytes in size, and would take around 20 seconds to save. This was not a concern at the time because the server had more than enough memory, but implementing caching and incremental saves would allow the database to work much better in low RAM configurations.

The second improvement is also to save on system resources, specifically disk space. Similarly to how humans forget information that our brains classify as non-important, inputs that are either duplicates of other inputs or fall over some similarity threshold could be discarded and replaced with meta-data about them instead. This would improve storage size as well as speed up compare operations.

0.6 Possible Future Applications

In computer science, clustering is a process of taking a collection of unclassified objects and a means for measuring their similarity, and find classes of objects such that some standard of quality is met. (Onder, 2014, p. 4) Clustering algorithms are used by search engines to find clusters of related web pages based on keywords (p. 12), differentiate between clusters of stars based on their radiation levels (p. 9), and many other applications that require an algorithm to make summaries out of large amounts of unclassified data. A dynamically linking database could find clusters in a novel, more in depth way than a standard k-means or CLUSTER/2 clustering algorithms.

This type of database excels at classifying objects, and could be integrated into a system that needs to be able to tell things apart, classify them, or seek out specific objects. In section 0.4, there was an example of integrating a dynamically linking database with a neural net, but it would be even easier to integrate one with a rational agent that needs to classify objects in its environment, such as object recognition systems. Any system that needs to remember outcomes of actions and use them to predict the outcome of future actions could also benefit from this kind of database.

References

Russel, S., & Norvig, P. (2010). *Artificial Intelligence A Modern Approach Third Edition.* Upper Saddle River, New Jersey: Pearson Education, Inc.

Myers, D. G. (2013). *Psychology Tenth Edition.* New York, New York: Worth Publishers.

Onder, N. (2014). *Clustering.* [Powerpoint slides]. Retrieved from http://www.cs.mtu.edu/ñilufer/classes/cs4811/2014-spring/lecture-slides/cs4811-ch18-clustering.pdf