

Hardware Limited Artificial Intelligence: Desirability Assessment for Board Games

By Andy Matange & Michael Daly

Contents:

1. Introduction
2. Rules of the Game
3. Development for the Game Boy Advance Console
 - 3.1 CPU and Memory Limitations
 - 3.2 Operating System Limitations & Development Environment
 - 3.3 Interface Limitations
4. Board Game Artificial Intelligence
 - 4.1 Overview
 - 4.2 Desirability Assessment
 - 4.3 General Procedure
 - 4.4 Probability
 - 4.5 Resource Variety
 - 4.6 Distance From Roads
 - 4.7 Resource Availability
 - 4.8 Remove Illegal Vertices
 - 4.9 Final Processing
 - 4.10 Additional Consideration Possibilities
5. AI Alternatives
 - 5.1 Heuristic Tree Implementation
6. Further Application
7. Reflection
 - 7.1 Challenges
 - 7.2 Successes
8. References
9. Developers' Comments
 - 9.1 Project Statistics
 - 9.2 Mike's Comments
 - 9.3 Andy's Comments

1. Introduction

Our research involved applying our knowledge of AI, graphics, and operating systems to develop a video game for the Game Boy Advance® (GBA). The research falls into two categories: developing software for the hardware-limited GBA, and developing an effective Artificial Intelligence (AI) to play a board game. This document covers the rules of the game, our research goals of hardware-limited development and AI, the alternative designs considered, and the applications of what we learned.

The board game we chose to implement was *Settlers of Catan*©. The game involves resource management, construction, foresight, and chance. Resource management and construction were aspects of the game that were easy to abstract such that the AI could understand them. Unique challenges to our AI were found in the fact that the game also uses random variables (dice rolls), which makes planning and prediction more complex.

Our first research goal, concerning hardware limitations, was purely to build our own experience with this type of development. We had to be concerned with three primary categories of limitations. The first is the simplicity of the operating system (OS) on board the GBA hardware. The second is the relatively small amount of memory available for use by our program. Finally, interface between the player and our program is limited to the buttons and screen of the GBA. All of these aspects put considerable constraints on the software design and graphical layout of the program.

Developing an AI that can effectively play the game against a human opponent was our second research goal. First we had to analyze the rules of the game and abstract them in a way that could be represented mathematically. Next, we developed an algorithm that could assess the desirability of positions in the game. Using this assessment, the AI could select the most advantageous goal to work towards.

During the planning phase, a few alternatives to our final design choices were considered. For the development hardware, we decided to use the GBA to expand the breadth of our knowledge, and to give our project tighter focus. The two AI implementations we considered were a logical search AI and a heuristic tree approach. We thought that the logical search AI would more closely reflect human choices in the game and better fit the hardware limitations.

Designing under such limitations is often necessary for specialized software development such as web development, mobile devices, and game development. The AI concepts that we worked with can be applied to many situations. There are a large number of board games which can be abstracted and played with a similar decision making algorithm as we used for *Settlers of Catan*©. Real world situations can sometimes mimic rules imposed by board games, and this approach can be useful in solving such problems.

2. Rules of the game:

Our game was based off of the board game *Settlers of Catan*©. Following is the description of the rules that we used in our implementation, which is a subset of the full rules of the original game.



Figure 1: Board Layout

- 1) Board setup: The playing board is composed of 19 hexagonal tiles (hexes) arranged in a honeycomb pattern (Figure 1).
- 2) Hexes: Each hex has a resource associated with it (discussed in item 3), which is visually represented by the graphic of the tile. Additionally, each hex has a probability associated with it (discussed in item 6).
- 3) Resources: There are five types of resources: wheat, wool, lumber, brick, and ore. 18 of the hexes on the board have one of these resources associated with it, and the 19th hex is a desert, and has no resources associated with it.
- 4) Property placement: Each player can control 3 types of property on the board: towns, roads, and cities. Towns must be built on the corners of the hexes, usually placed on the corners of three hexes joined together (vertices). Cities may only be placed in place of existing towns on vertices. Roads may be built on the edges between two hexes. Roads, towns, and cities can only be placed adjoining existing roads belonging to the same player.



Figure 2: Initial Player Setup

- 5) Player setup: At the start of the game, each player has 2 towns and 2 roads placed on the map (Figure 2).
- 6) Probability: Each hex has a number associated with it that corresponds to a roll of two six-sided dice. The number 7 is never assigned to any hexes. When rolling two six sided dice, the five probability categories are 'Best', which refers to any hex that has a 6 or 8 assigned to it (the most frequent rolls after 7), 'Good' referring to rolls of 5 and 9, 'Medium' for 4 and 10, 'Poor' for 3 and 11, and 'Worst' for 2 and 12.
- 7) At the start of each turn: At the beginning of each player's turn, a dice roll is generated and if a player has a town built on one of the corners of the hexes with that probability value, then that player receives one of the resource type of the rolled hex. If a city is on the corner of a rolled hex, the owning player receives two of the hex's resource.
- 8) During a player's turn: During their turn, each player can build a road, build a new town, upgrade a town to a city, trade 4 types of one resource for 1 of any other type, or end their turn.
- 9) Roads: A road costs one brick resource and one lumber resource and must be placed on an edge where the same player owns a road on an adjacent edge.
- 10) Towns: A town may be placed on a vertex only when the same player owns a road on one of the three edges meeting at that vertex. Additionally, a town cannot be built on a vertex adjacent to another town. A town costs one brick, lumber, wheat, and wool.
- 11) Cities: A city may be placed on a vertex only when the same player already owns a town on that vertex. A city replaces a town, and costs three ore and two wheat.
- 12) Points: Players get one point for each town they control, two points for each city, and two points for having the longest road of all players that is a path of at least five unique edges.
- 13) Victory: A player wins the game when they have 8 points.

3. Developing for the Game Boy Advance console

3.1 CPU and Memory Limitations

The Game Boy Advance®, created by Nintendo®, is a hand-held gaming device with the following specifications listed on its official webpage (www.nintendo.com/techspecgba):

- CPU : 32-Bit ARM with embedded memory
- Memory : 32 Kbyte + 96 Kbyte VRAM (in CPU), 256 Kbyte WRAM (external of CPU)
- Screen : (diagonal) reflective TFT color LCD
- Resolution : 240 x 160 pixels
- Color : Can display 511 simultaneous colors in character mode and 32,768 simultaneous colors in bitmap mode
- Software : Fully compatible with Game Boy and Game Boy Color Game Paks

The important things to note about the hardware are that it has a 16 MHz CPU clock speed, and 256K of memory for the program's general use (non-graphical). These limitations are important to keep in mind because many AI algorithms rely heavily both on intense calculation and large amounts of data storage.

In addition to the hardware limitations, we also had to consider the fact that the GBA runs a custom operating system that does not provide many of the conveniences of a PC OS. Finally, we had to make design decisions on the implementation of our interface based around the GBA hardware.

The first design decision that was made as a result of the limited memory was to implement the program using entirely procedural code. Using object-oriented programming would have made representation of the board much easier, as we could have built an easily searchable node and edge network with each object storing all that needs to be known about structures and ownership. It was believed, however, that the amount of overhead due to pointers would be too great.

In one early implementation we considered making a tree of possible game states, which would require making multiple copies of our game state data structure. This would have taken up memory so rapidly that our tree would not have enough depth to predict later rounds and make a good decision.

We wanted to pack the data needed to represent the state of the game in as small of a space as possible. To do this, we represented the game state as an array of bytes. Each byte in the array would represent an edge, a vertex, or the center of a hex. Each of these types of array elements would have a unique bit code that stored the data relevant to its purpose.

For the edges, we needed to store only the player that owns the road. Since there are a maximum of four players, we only needed two bits to store this, then another bit to represent if the edge was free and could be built on. For Nodes, we needed to store if the node was free to be built on, what player controlled the node if it had a building, and what type of building was there (town or city). For the hex elements, we needed to store what type of resource was associated with that hex; there are five resources, so we needed 3 bits, and what die roll would trigger that hex, which indicates probability. There were 12 possible values for die rolls, so we needed 4 bits. Since all of the data for

each element type fit into one byte, we could represent a full game board by a global array of 145 bytes. There was exactly one byte for each edge, vertex, and hex.

To ease calculation, we created a function that referenced a lookup table that you could pass in an x and y value in map coordinates, and it would return the corresponding byte in the game state array. This implementation made searching and traversing the vertex network a little more difficult, but took up a fraction of the memory that an object-oriented approach would.

Another problem the CPU presented was that it was incapable of floating point arithmetic. We designed our AI to analyze the game state by weighing certain factors. This would have been easy if we could multiply the preference value of a given node by some decimal value between 0 and 1. Instead, we had to translate our calculations to integers, making the scaling system less easy to understand.

3.2 Operating System Limitations & Development Environment

The second major difficulty that the hardware presented was the limited operating system of the GBA. The first problem we faced was that we could not include any external libraries into our program. Because the GBA does not have any standard reference libraries, we had to code everything from scratch. What this meant was that we could not use dynamic memory allocation, did not have access to any string manipulation functions, and did not have any math functions for random number generation.

By careful planning, we found that dynamic memory allocation was not necessary. If we considered the execution of the code carefully, we could predict exactly how much storage was needed for a given calculation, so we could allocate that amount as a global before execution.

String manipulation would have been helpful, but we planned our GUI such that it was not necessary. Since there was no variation of the text displayed at each dialog, those could all be constants. Where we needed numbers to display resources and status, we only had to come up with a conversion function from integer to a short character array. A well-planned GUI meant that it was not necessary to use string manipulation functions.

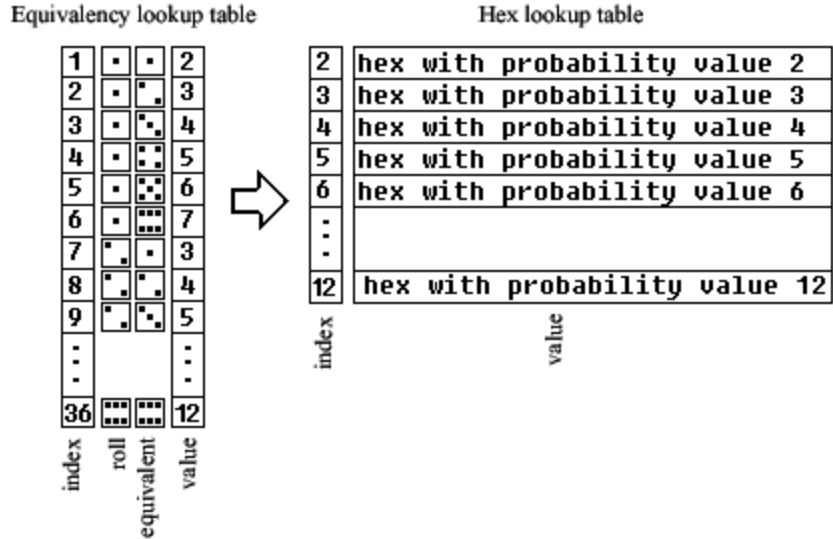


Figure 3: Dice Rolling Process

Random number generation was important to us because each turn two dice had to be rolled to determine what hexes distributed resources. We did not have access to a random number function, so instead we created a global variable that incremented each screen refresh. The value of this number counted up to 36, then started over at 1. The importance of this is that there are 36 possible combinations of numbers between two six-sided die. Once a roll needed to be made, this number was used to reference the equivalency lookup table (Figure 3) for the dice roll; with the lookup value ranging from 2-12. This dice roll referenced the hex lookup table that told the coordinates of the rolled hexes. The importance of having 36 values in the first lookup table is so that higher probability rolls (such as 7) can occur as frequently as they do naturally, instead of having a uniform distribution (as if we had counted from 0 to 10 and ignored the first lookup table).

To develop this game, we used a library called HAM©. HAM takes C or C++ code and compiles it to a ROM file that can be loaded by an emulator. In addition to compiling code, HAM is the only library that can be referenced from our code, providing an interface with the GBA hardware. One obstacle we faced with the language was that it does not perform things like array index out-of-bounds checks. Instead, it would just overwrite the memory outside of the arrays definition. This made it more difficult to find the source of bugs because we had no indication of where and when global data was being corrupted.

3.3 Interface Limitations



Figure 4: The Game Boy Advance
(Copyright Nintendo®, <http://www.gameboyadvance.com>)

The GBA has a 240 x 160 pixel display for all graphics, a 4-direction control pad, and 6 buttons for input (Figure 4). Because of this, we had to carefully consider the graphical presentation of our game such that we could fit all of the necessary information on screen at once. Additionally, we had to consider how the user would control the game.

The hardware of the GBA has a dedicated infrastructure for handling backgrounds that are broken up into repeated 8 pixels by 8 pixels tiles. To conserve video memory and make icon placing easier, we wanted to come up with a map that had relevant locations aligned to an 8x8 pixel grid. We decided that the most efficient way of doing this was to fit a hex in a 24x16 area, such that each hex would take up six 8x8 tiles, and that each hex's region would overlap the other hexes on each of the corner pieces. To make a regulation board using these dimensions, we used up exactly the 160 pixels of height, but only needed 176 pixels of width. This means that we had exactly 64 pixels to the side of the map to display all GUI information.



Figure 5: Game Screen Layout

The GUI information we wanted to display fell into two categories: the status of the player, including resources and points, and context sensitive information about whatever the cursor is pointing at. The 64x160 area on the side was divided in half, with the bottom half being dedicated to player status. We used icons for resources to conserve screen space and increase the aesthetic appeal of the game. Fortunately, the 6 16x16 resource icons (one for points), as well as the player name, filled that half of the GUI perfectly (Figure 5).

When the cursor was over edges and vertices, the top half of the GUI could easily fit in the resource icons representing the cost of building a road, town, or city at that location. When the cursor was over a hex, the GUI displays the resource icon that hex distributes as well as the chance category for that hex (Figure 5). No extra information could be given because each text character takes an 8x8 tile, and the borders of the GUI window take up 16 pixels. This leaves only enough space for 6 characters to be displayed on each line. This kept the interface very simple.



Figure 6: Build Dialog

When the player wishes to make a choice such as to build a town, road, or city, we had to draw a GUI window over top of the map (Figure 6). These windows used icons to represent costs and gave the player a chance to confirm construction.

For player input, we used the directional pad to control the movement of a cursor. The behavior of the cursor was determined by a finite state machine. For example, the direction pad usually moves the cursor around the playing board. When the cursor is over edges or vertices that can be built upon, the cursor icon changes to represent what can be built there. At this point, the player can hit the 'A' button to bring up the build dialog, and the cursor changes state and behaves differently. Since we had very few buttons to work with, the simplest interface would be to have 'A' as the universal accept button and 'B' as the universal cancel button. All decisions would be made by pressing 'A', which usually triggers a GUI menu giving more options or describing what just happened. At any point when a menu is on screen, the player can press 'B' to get rid of it and return the cursor to its map state.

4. Board Game Artificial Intelligence

4.1 Overview

The AI implemented for this game loosely follows a combination of the logical and search AI models. A logical AI “decides what to do by inferring that certain actions are appropriate for achieving its goals,” whereas a search AI “[examines] large numbers of possibilities” to determine the best situation (McCarthy). The AI we built is a combination of the two, which we call a logical search AI, since it both infers what action is best for reaching a goal, as well as frequently searching through the graph that represents the game.

To build an AI that could follow the rules and strategy of the game and apply them to find advantageous positions, the rules themselves had to be converted into a mathematical form. Those aspects of the game which were directly related to player interaction with the game board: placement of roads, towns, and cities, were already implemented before AI development began. The remaining areas that required such abstraction were the more strategic elements of game play: resource quality (probability and type) and distances between vertices.

4.2 Desirability Assessment

The desirability assessment measures how advantageous each potential game state is in advancing the AI player towards winning the game. Using the mathematical representations that were derived, we could create an AI that determines its next move by going through a series of decision stages. The AI first analyzes every vertex on the graph in stages, each stage representing one strategic aspect, to determine the desirability of that vertex, and selects the highest weighted one as its goal. This goal is the vertex that would be most advantageous resource-wise to capture or convert to a city.

Once the goal is chosen, the AI determines the best course of action to reach and capture the vertex: either build roads towards it or build a town or city on it, whichever is allowed at the time. The following sections describe how the strategic elements and game rules considered by each stage of the AI’s decision making process are represented mathematically and how each stage affects the desirability of the vertices.

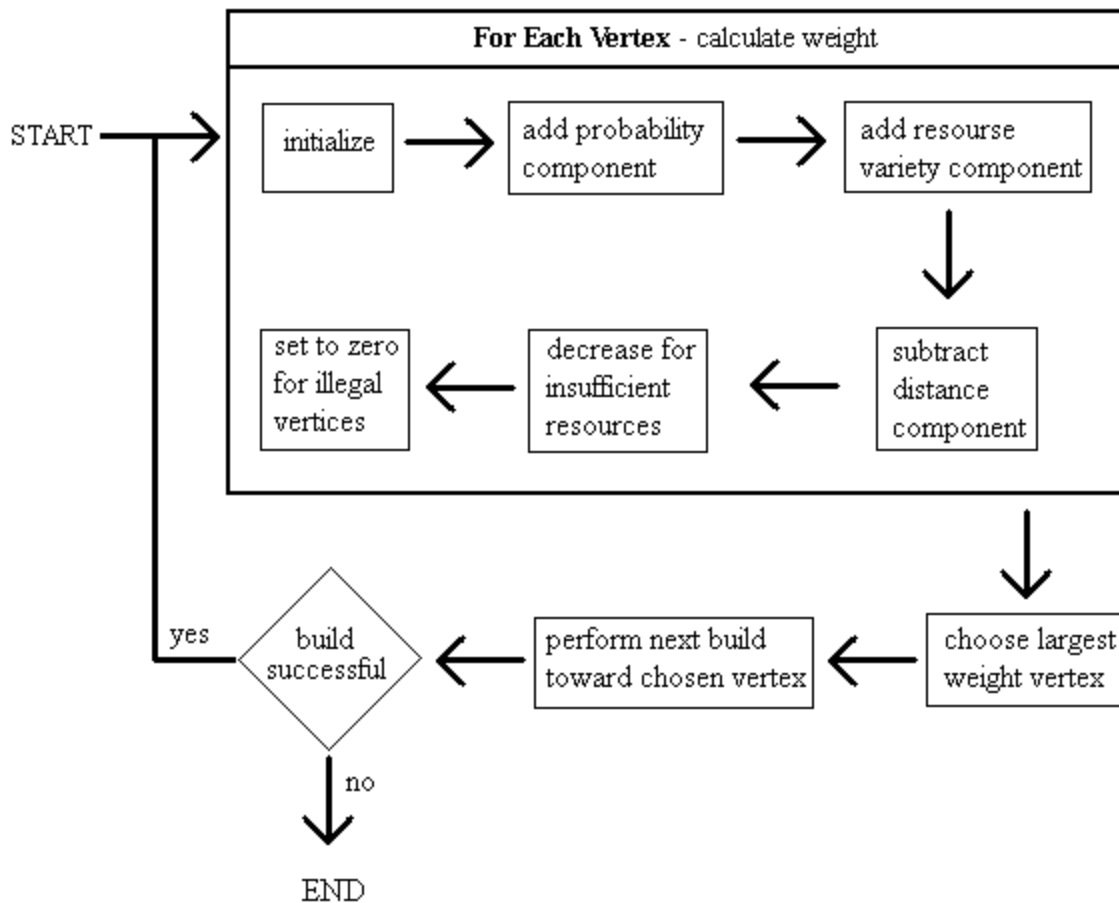


Figure 7: AI Decision Process

4.3 General Procedure

At the beginning of AI execution, the weights for each vertex are all initialized to a default value to prevent any from becoming less than zero during the process. In each step of the decision making process, the AI calculates a weight for each vertex and adds the value to that vertex's total accumulated weight. After the final weight for each vertex is calculated, the one with the highest weight is selected as the goal vertex (Figure 7). This is the vertex determined to be most desirable and is the next vertex the AI player will attempt to capture.

Once the goal is selected, the AI player: builds roads to reach the vertex if it does not already have the player's road adjacent to it, builds a town if it is reach but not occupied, or builds a city if it is already controlled by the AI player. Each of these builds can only occur if the AI player has enough resources. If the AI does not have sufficient resources for the next selected build, it ends its turn.

The following sections detail each stage in the decision making process. Each vertex's weight is initialized to one hundred, and during the individual stages, the weight

is either increased or decreased by the calculated factor. Zero is the minimum weight given to a vertex, and the weight is capped at three hundred.

4.4 Probability

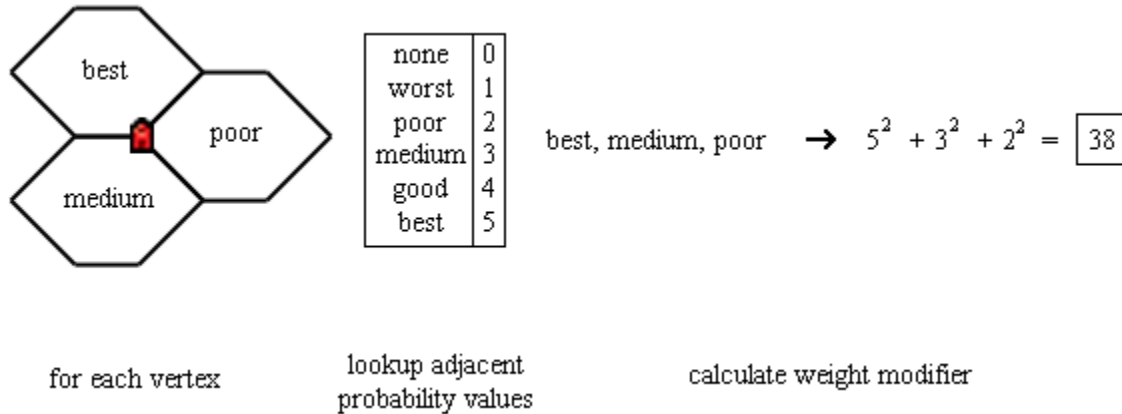


Figure 8: Probability Weight Modifier Calculation

When selecting a vertex to control, players tend to look for those vertices which are surrounded by resource tiles with higher probabilities of yielding resources, regardless of the resource type. Likewise, the first step of the AI decision involves taking the probability of getting resources from each of the three tiles surrounding each vertex, and using these probabilities, modifying the weight of each vertex. Using each surrounding hex's probability, a value is generated that represents the number of times that hex would be chosen out of a set of thirty-six rolls assuming uniform roll distribution. For each resource tile adjacent to each vertex, we took the square of this calculated value, and added it to the vertex's weight. There are five different categories of probability, so the weight modifier for probability ranged from zero to twenty-five. This way, the vertices surrounded by higher probability tiles would have considerably higher weight than those with low probability tiles (Figure 8).

4.5 Resource Variety

The second factor in determining the desirability of a vertex is the number of different types of resources that can be gained by controlling a particular tile. Controlling a vertex which is adjacent to only one type of resource severely limits its usefulness even if the probability is high, simply because building requires multiple resource types. For this reason, we incorporate resource variety into the calculation. For each different resource type adjacent to each vertex, the weight of the vertex is increased by five to account for resource variety.

4.6 Distance from Player's Roads

The next factor to incorporate is the distance each vertex is from the closest road that a player controls. Those vertices that are closer to the player's road network will be

much quicker to reach and control, and will require less resource expenditure by the player to acquire. Also, closer vertices are less likely to be intercepted by another player than a vertex that is farther away from the player's road network. To take into account the fact that farther vertices are considerably less desirable than closer vertices, we run an all-pairs implementation of Dijkstra's shortest path algorithm. This results in a matrix giving the shortest path lengths between any two vertices on the graph, in numbers of roads, as well as the vertices comprising each shortest path. When examining each vertex u during this section of the AI decision process, the closest vertex v adjacent to a road controlled by the AI player is found, and this path length is squared and subtracted from the weight of u . After this calculation, the vertices far from the AI player's road network lose much of their weight, while those closer will reduce in value much less.

4.7 Resource Availability

When a human plays the game, he often has a particular action he wishes to perform, but not enough resources to do so. However, he may have enough resources to perform a different action, and so must decide whether it would be more effective to build what he can, or to save the resources until he can build what he could not before. For example, a person may wish to build a town at a particular vertex, but does not have enough wool resource. However he does have a brick and lumber to build a road elsewhere. In this case, the player must decide whether to build the road, or to wait a turn or more until he acquires a wool resource either from a tile or by trading, and then build the town on the next turn.

Similarly, the AI player should also be able to decide whether it is more effective to save resources and build something later, or to use all the available resources to build everything it can. In many cases, the vertex where the town will be placed is much more important than continuing to build roads. In order to account for such situations, after the previous weighting of the vertices is complete, each vertex is reduced by a fixed value if there are insufficient resources after all possible trading to perform the action required to advance toward or capture the vertex, whether that be building a road, town, or city. If, after this decrease in weight, the most desirable vertex still cannot be built towards due to insufficient resources, then the AI ends its turn to save up for building later.

4.8 Remove Illegal Vertices

After all weighting is complete, the AI searches through all vertices and removes those that are illegal to build on. This could be due to there being no possible path from the player's roads to the vertex, the vertex being owned by an opponent, or the vertex having the current player's city already on it. To remove vertices, we simply set their weight to zero, the minimum weight value.

4.9 Final Processing

All weight and removal assessments have been completed at this point. The AI procedure finds the highest weighted vertex in the set, and selects that as its goal for the next move. If the vertex has the player's town on it, then a city will be built if possible. If the vertex is adjacent to one of the player's roads, then the AI player will build a town if resources are available. Otherwise, the player will build a road to the next vertex in the shortest path from the closest reached vertex to the goal. If there are not enough resources to complete the chosen action, the AI ends its turn, otherwise it begins the entire decision making process anew based on the new game state.

4.10 Additional Consideration Possibilities

The stages of the AI decision making process explained above involve components that a human would take into consideration when selecting an ideal vertex to control. These components were represented mathematically in order to give the AI a way to assess their effect on the desirability of each vertex. Naturally, the factors considered by the AI are not the only ones that affect desirability of vertices. One additional factor would be resource need. Essentially, this stage, if added to the AI, would involve comparing the types of resources acquired at the target vertex with those already being produced by controlled vertices. A vertex would be weighted more heavily if it produced a resource which the AI player had a scarcity of. For instance, if a vertex was adjacent to a lumber producing tile, and the player had no towns or cities on vertices adjacent to these types of tiles, then the vertex being analyzed would increase weight considerably.

Since the AI was developed as a step by step analysis of the vertices, adding new weight affecting factors is relatively simple. Each time a new factor is discovered and needs to be included, we can simply make an additional stage in the process which affects the same set of overall weights for the vertices.

5. AI Alternatives

5.1 Heuristic Tree Implementation

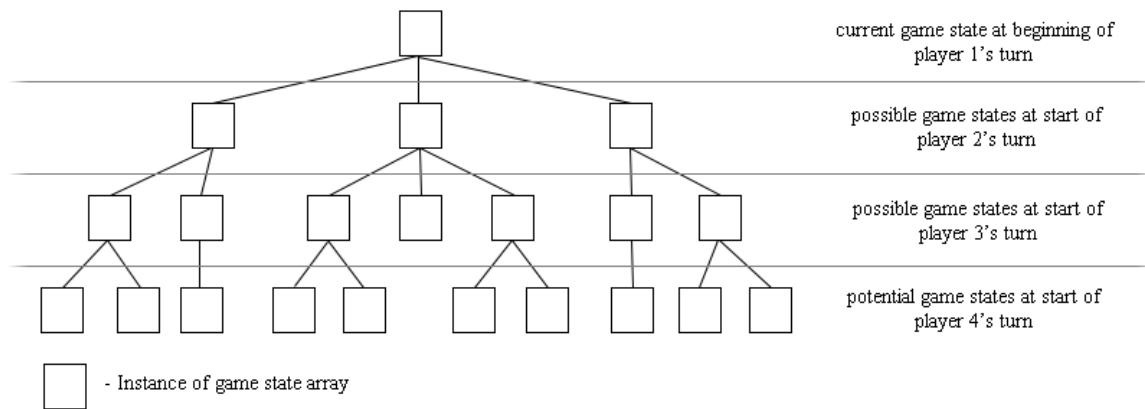


Figure 9: Heuristic Tree

For this project, we implemented a logical search AI, which combined aspects of logical AI and search AI to give intelligent play without a large amount of overhead. Another option we considered was a heuristic tree approach. With this AI, the current game state would act as the root of a tree, with the children of each node comprising all of the possible game states that could arise from a move being made on that node's game state. Each level of the tree would then represent a player's possible combinations of moves for each possible game state on the previous level. Thus, the root is the current game state before the AI makes its moves, the next level presents each game state that could result from each possible series of moves of the AI player, and subsequent levels

contain the game states resulting from the possible move series of each player in turn for each of the game states of previous levels (Figure 8).

A function would be created to assess each game state and give it a quality value, where a game state in which the AI player is in a better overall position would have a higher value. This tree would be as many levels deep as possible to allow maximum look-ahead, and the AI would make moves corresponding to the sub-tree of the root node that presented the most favorable game states. It is quite apparent that this AI model would require much larger amounts of system resources than the AI currently implemented.

6. Further Application

The logical search approach to giving an AI a goal could be applied to a multitude of situations. The ideas can be useful for a variety of games as well as some real-world situations.

With minor modification, the logical search AI could be useful in situations where an AI needs to traverse a node network to reach an advantageous goal. For example, such a situation would be Pac-man. In Pac-man, each ghost would independently choose a target node to move to based off of some weighted factors such as the nodes distance from Pac-man, and Pac-man's direction.

With significant modification, the AI could consider a continuous space, rather than a discrete node network. This would be useful for a real time strategy game where the AI must search the map for advantageous positions to settle or attack.

In addition to games, the AI could be useful in some real world situations that require network traversal with desirability assessment. For example, an automobile navigation system could use this to choose a destination. The advantage of the logical search algorithm in this situation is that it can consider many factors into its route decision. Specifically, if the driver wished to find a restaurant, the algorithm could find all of the nearby locations, and weight them based off of distance, traffic conditions, and driver preferences. Additional factors could be considered, such as average pricing and customer reviews.

7. Reflection

7.1 Challenges

We were not able to reference any external libraries in our program. As a result, the program had a lot of lines of code because we had to write everything ourselves. One example of this is with window handling. Instead of having a library to display a window, we had to manually set the graphics where the window should be and use a finite state machine to manage the cursor control while the window was open. The sheer volume of lines made finding pieces of code cumbersome.

Another challenge was formatting graphics such that the emulator could load them. Because the GBA OS does not load data from a file structure, all of the graphics data had to be included in the program code. To do this, we used a conversion utility called `gfx2gba`, which was included with HAM. One complication is that all of the sprites and all of the backgrounds had to share a common limited palette. This meant that when we added graphics after our early builds, we had to make a new palette which could change the colors that were indexed by the previously compiled graphics. To solve this problem, we initially looked at the color index arrays of each graphic in hex, and manually set them to index the appropriate color. Later, we learned more about creating shared palettes in Photoshop©, and were able to make all of our graphics compatible.

Memory management was another challenge that we had to overcome. HAM did not have support for dynamic memory allocation, which would have made the code simpler to implement. We also encountered a memory leak problem causing our lookup tables to be overwritten in some situations. This problem was hard to isolate because of the lack of debugging capability. We resolved the problem by reloading the tables.

Creating an efficient data structure for the game state also posed a challenge. This structure was critical to the rest of the implementation, and so had to be efficient in both memory usage and seek time. Creating an array of bytes to hold edges, vertices, and tiles was the most compact, and lookup tables were used to convert from the data structure to screen coordinates and to differentiate between each feature type. The data for each feature also had to be packed to fit within one byte of memory.

Since we could not reference external libraries and we had to write our own string manipulation functions, we were not able to output debug information until a significant portion of the graphics code was written. The quantity of debug data that could be displayed was also limited by the screen size and resolution. Thus, debugging our code was a tedious process.

7.2 Successes

The workload of our code was split into the categories of AI code and game programming. Each of us worked on one portion of this workload separately then integrated at our regular meetings. This aspect of the development went especially smooth; we barely had to modify our interfaces to be compatible, so we had the AI playing the game shortly after its code was written (without the game interface). This was due to thorough planning and communication at the beginning of development.

Another aspect of the development that went well was creating an AI that behaved in an intelligent manner. When playing against computer opponents, the choices they make seem reasonable. We could not consider our AI a success if its decisions did not appear to be reasonable to a human player.

While the lack of libraries added to the difficulty of programming, it also made development simpler. Because we never had to call an external function, the naming conventions across variables, constants, and function names were consistent and readable. Additionally, the code was all at a relatively easy to follow low level without any black-box function calls. Using a finite state machine for input and graphics simplified following the flow of code execution as well.

The layout we designed for menus as well as the cursor's context-sensitivity made interacting with the game very intuitive. Designing a logical user interface was more difficult than it seemed like it would be. But with a game this complicated, it was necessary to layout graphics and menus such that it makes sense to a new user. In this respect, our UI was a success.

Overall, the program was relatively painless. Because of the limitations of the platform, we were forced to adopt a design simplicity that made the program easy to understand. Careful planning, division of labor, and design simplicity made the development go very quickly, which was vital considering the limited timeframe of our project.

8. References

Nintendo Game Boy Advance © : <http://www.gameboyadvance.com>

Settlers of Catan©: <http://www.die-siedler.com> (German)

HAM© by Emanuel Schleussinger <http://www.ngine.de>

McCarthy, John. *What is Artificial Intelligence*, March 2003.
<http://www-formal.stanford.edu/jmc/whatisai/whatisai.html>

9. Developers' Comments

9.1 Project Statistics

Some project statistics	
Development Timeframe	6 weeks
Number of Developers	2
Code Size	3,246 lines of C
Function Count	78
Size of compiled ROM file	175 KB
Pixels of art content	206336

9.2 Mike's Comments

I was surprised at the ease of development for the GBA. HAM does a good job of abstracting the interface with the GBA hardware, so with only a few lines of code, we already had a running ROM. After that, it was just building our game on top of it.

Andy and I split the work needed for this project into 1) game development and graphics, and 2) artificial intelligence. For the first week or two of development, neither of these sides had any code written. Once we decided on the underlying data structure of the game state, we both started working completely independently; Andy on AI, and myself on the rest. When we met to integrate the first iteration of the AI, Andy had not even looked at any of the GBA code I'd written. Miraculously, within a few minutes of making sure the function call parameters were the same across our code, we compiled and ran without an error. The AI started building things on our first try, which was shocking to me considering we'd independently wrote hundreds of lines of code without much consultation between us.

Initially, I thought that making an UI for the game would be very difficult considering I didn't have any helper functions to work with. However, since I wrote all the code myself, it was easy to understand. I never got to the point where there was an error I didn't understand and could fix easily. In contrast, I used GLUT (GL utility toolkit) for some OpenGL programs I've written, and they added a great deal to the difficulty of development because I didn't understand what was going on when I called them, which made it harder to track bugs. In the end, writing our own UI worked to our advantage.

This was the most productive and least painful programming project I've worked on during college. It was the largest project I've worked on, being at least 50% more work than any other. Despite the work, it was always fun, and rarely frustrating, which set it apart from most other programming I've done.

9.3 Andy's Comments

My work on this project focused primarily on the AI and underlying data structure and data packing, while Mike chose to work on art, graphics and game play elements. Since I had no familiarity with HAM at all, I decided to not spend any time initially working with that library, and instead wrote all the AI code purely with C. This made the

debugging for my code really easy, as I could just send a bunch of debug text to the console and sift through it. Once I had the code working as expected on the PC, we integrated it with the game play code that Mike had been writing. Within a few minutes, the AI was placing roads and towns on the map.

Everything seemed to be going perfectly, until the next turn, when the AI seemed to build at completely random, illegal locations. Baffled by this turn of events, I returned home to try to sort out this problem, comparing the GBA results with those from the original AI code on PC. The results were completely different between the platforms, and after a few days of experimentation with the GBA, I found that some of the lookup tables were being overwritten. Strangely, this phenomenon was not occurring on my PC version, and the only explanation was a possible memory leak occurring on the GBA. I created a workaround for the problem, and after that, the AI worked correctly again.

Aside from a few minor setbacks, this project went very smoothly overall. I was particularly surprised by the ease of integration with the game play code, which was operational within a few minutes of copying my code into Mike's framework. The moral of the story: taking a few hours out of development to thoroughly plan your project will save you from many hours spent pulling out your hair over bugs.