CS303 Theory of Computation: Implementation of the Machines

Wah Loon Keng[*]

April 21, 2015

Abstract

We study and implement all the machines leading up to the Turing machine: DFA, NFA, PDA. To illustrate their relationship and hierarchy of powers, we emphasize the machines as subsets of the more general ones. We start with the Non-deterministic Turing Machine, and increasingly restrict its functions as we descend the hierarchy to the less powerful machines. The polymorphism allows all machines to run on the same implemented code, and stresses that Turing Machine is the most powerful of all.

1 Introduction

This paper assumes the reader's familiarity with the theory of computation, and with core ideas such as computation history, configuration, languages and set theory. This entire project is written in `Javascript`, and is public on GitHub at: `https://github.com/kengz/Machines`.

The Turing Machine(TM) is the most general class of computing machine that we know of. Its power is measured by the problems it can solved, or equivalently, the languages it can decide. In the universe of languages, we can roughly classify from the most powerful to the ever strictly less powerful classes of languages as follow:

$$\textit{Turing-recognizable} \supset \textit{Turing-decidable} \supset \textit{Context Free} \supset \textit{Regular}$$

Equivalently when expressed with machines, we get the hierarchy:

$$\textit{TM} \supset \textit{TM (halting)} \supset \textit{PDA} \supset \textit{NFA/DFA}$$

[*]Lafayette College, Easton, PA 18042, USA. kengw@lafayette.edu.

Furthermore, a machine is also equivalently identified with the class of problems it can solve, thereby establishing the equivalence among languages, classes of problems, and classes of machines. This insight will be useful later.

With this idea in mind, our project implements all these machines by starting from the most generic class – the Turing machine, and progressively restricts its power as we descend the hierarchy. The aim is to illustrate the factors that affect the power of a machine and differentiate the classes.

Implementation by restriction also allows the use of polymorphism, which in our opinion expresses the idea cleaner and more naturally.

Most reasonable variants of the Turing machine are equivalent in power, thus this allows us to choose one that works best for our purpose. We choose the Non-deterministic Turing Machine(NTM) due to its tree structure that represents non-determinism more clearly than an equivalent Turing machine.

Draft:

1. Turing machine, use NTM for tree. Present parts of machines. Memory.

2. slowly descend the hierarchy, with more restriction on machines. changes in memory access

3. describe language class and restriction by algorithm

2 General Purpose Machine

To simulate all the other machines, our choice of this general purpose machine is the Non-deterministic Turing Machine(NTM), which is a computation history with instances of TM configurations branching out non-deterministically from its start.

To recap, a configuration is a triple: the machine's state, its head location, and its tape content. This can encode the entirety of not only the TM, but also the subordinate machines.

The NTM can be represented using a tree structure, whose node is a machine configuration, with root as the starting configuration. The tree depth of a node is the number of computation steps it is from the root: each step of computation non-deterministically expands the tree by one level down, and epsilon transitions expand the tree sideway on the same level. When expanding, the original node gets copied, and a computation step is applied to get the next configuration, which is then added to the tree.

For practicality, we want our machines to be halting, i.e. all problems must be decidable. The NTM decides at the first level of tree that contains a halting(accepting/rejecting) configuration. Each class of machines may have a slightly different halting configuration.

The NTM as a computation history represents non-determinism naturally; it can also capture epsilon transitions by side-way expansion. A non-deterministic computation history is simply a degenerate tree without branching. The full power of a TM is captured locally in the configuration node and manifested from its components: the head, the tape, and its transition functions. Since NTM and TM are equivalent, we provide the definition for only TM below.

3 Machine Powers and Restrictions

3.1 Turing Machine (TM)

Definition 1. *A **Turing Machine** is a 7-tuple:*

1. *Q: the non-empty set of machine states,*
2. *Σ: the non-empty set of tape symbols,*
3. *$\Gamma \subset \Sigma$: the set of input symbols,*
4. *b: the blank symbol,*
5. *$\delta : Q \times \Sigma \mapsto Q \times \Sigma \times \{L, R\}$: the transition table,*
6. *$q_0 \in Q$: the starting state,*
7. *$F \subset Q$: the set of accept states.*

As an equivalent, more physical description, the machine is a configuration which has:

1. *a state,*
2. *an countably infinite tape which serves as its memory,*
3. *a head that has random access to read and write on the tape.*

Any discrete information is quantizable, and can be encoded bijectively into a binary string − this is the hallmark of Shannon's information theory. Therefore, w.l.o.g., TM can take any encoded binary string as its tape input and manipulate the information in any way it wants using its random access.

Since the sets Q, Σ are finite, each transition table δ is a finite subset of the power set of all countably finite permutation of $Q \times \Sigma \mapsto Q \times \Sigma \times \{L, R\}$, taken over all possible sets of Q, Σ. The collection of all such transition tables is simply all finite subsets in this power set, thus its cardinality is countably infinite.

This puts the size of the Turing Machine class at countably infinite, that is, it can solve that many classes of problems/languages. However, this is insufficient to solve *all* classes of problems as there exists much more of them. This statement is reflected in the Godel's Incompleteness Theorems, or equivalently expressed with machines in Turing's Undecidability Theorem.

As an interesting sidenote, the relationship above generalizes properly to quantum information theory and quantum machines. The only major addition is that the information bit used in quantum theory is a *qubit*, which is an orthogonal Hilbert space basis that can be superpositioned. However, this does not give the quantum machines too much more power albeit its extra features, and we shall not dwell into the details.

3.1.1 Power of the Class of TMs - a Quick Discourse

"How powerful is the class of Turing machines," one may ask. We argue that it is the absolute upper bound that is physically realizable. The theorems of Godel and Turing partition the set of all problems into decidable(solvable) and undecidable. The former can be solved by a Turing machine; the latter cannot be solved in any way.

Moreover, we posit that there cannot exist a physically realizable machine that is more powerful than a Turing machine. Here is why: in physics, especially in quantum theory, all observables are quantized, i.e. all measurements in the universe are discrete. Regardless of the continuum we can use to formalize physical theories, the physical reality that we can access is discrete, so we can theorize: *the continuum of the real number is not physically realizable.*

For a hypothetical machine to be more powerful that a Turing machine, not only it has to do the impossible by solving beyond the class of decidable problems, it will also have to transcend the countable infinity, i.e. the hypothetical machine has to be on the level of the continuum. But this is not physically realizable as we said earlier; even if it were, by quantum theory, all the observables we extract from it will still not be on the continuum.

Finally, we theorize that the human brain is not beyond, but is in the class of Turing machines. Regardless of how massive the human memory(tape) is, there is still countably finite units of it in the brain, which makes it discrete.

We are still trying to understand and reconstruct the human brain with a computer, or to say, we are looking for a Turing machine that mimics our brain. This task is as daunting as searching for one rational number within the space of rational numbers, because we are looking for *one* Turing machine among all countably infinite Turing machine. A search by brute force is simply impossible. Nevertheless, the human brain is a Turing machine.

To reiterate, the class of Turing machines is the most powerful class of machines that is physically realizable, and there is nothing in our physical theories that can trancend it.

3.1.2 Implementing TM using NTM

This is equivalent to the NTM, so it is implemented without any restriction.

3.2 Pushdown Automata (PDA)

Definition 2. *A* ***Pushdown Automata*** *is a 6-tuple:*

1. *Q: the non-empty set of machine states,*
2. *Σ: the non-empty set of stack symbols,*
3. *Γ: the set of input symbols,*
4. *$\delta : Q \times \Gamma \times \Sigma \mapsto Q \times \Sigma$: the transition table,*
5. *$q_0 \in Q$: the starting state,*
6. *$F \subset Q$: the set of accept states.*

Note that this is obtained by restricting the definition of a Turing machine. PDA has no blank symbols, and we can extend Σ to include Γ so that Σ can be the set of all tape symbols, just as in a TM. This suggests that we can fit the input string and the stack together onto one single tape, and obviously we can simulate the actions of a PDA by a TM.

As an equivalent, more physical description, the machine is a configuration which has:

1. *a state,*
2. *an countably infinite tape which serves as its memory, with the stack concatenated to the end of the input,*
3. *a head that can only read the input portion, and read and write on the stack portion.*

The PDA is a restricted Turing machine: it does not have random access to its tape, and thus cannot manipulate the information freely as it wants. The input part of the tape can only be read in a strict sequence from start to end. The stack of the machine is an extra memory reserve for the machine with countably infinite capacity, however it cannot be randomly accessed (otherwise we could use the stack to emulate a complete Turing machine by first copying the input tape over); only the top of the stack is accessible.

3.2.1 Power of the PDAs

The power of a PDA is derived from its stack − a memory reserve with countably infinite capacity and very restrictive access. The PDA is also non-deterministic, thus like an NTM it computes in parallel − by branching out to explore different computation history until it finds a configuration that is halting.

The class of PDAs is equivalent to the class of Context Free Languages (CFL), which is strictly smaller than the class of Turing-decidable/recognizable languages. The restriction on CFLs can be seen structurally on its parse tree, or abstractly on the pumping lemma for CFLs. We know there are Turing-decidable languages that are outside of the CFLs, therefore PDAs can solve only smaller classes of problems.

3.2.2 Implementing PDA using NTM

PDA has the non-determinism that is captured naturally by the NTM. At each node of a tree branch is an instance of the PDA, and we use a restricted TM to simulate it. This is done by converting the definition of a PDA into a restricted definition of a TM. The TM appends the stack to the end of the input on its tape, and moves its head back and forth to read the next input and read/write on the top of the stack:

Algorithm **Conversion of a PDA to a restricted TM**

Given a PDA, all its definitions correspond directly to the definition of a TM, except for the transittion table, which we illustrate here. For every PDA rule in the form of instantaneous description $\delta(q_1, a, \alpha) = (q_2, \beta)$, add to TM the rules:

1. $\delta(q_1, a) = (q_{1a}, \dot{a}, R)$ to mark a as read, then move right,
2. $\delta(q_{1a}, t) = (q_{1a}, t, R)$ for all $t \in \Sigma$ to keep moving right until,
3. $\delta(q_{1a}, b) = (q_{1ab}, b, L)$ at the end of stack, where b is the blank symbol, move left,
4. the top of stack:

   4.1. If $\beta = \epsilon$ (nothing is pushed), then $\delta(q_{1ab}, \alpha) = (q_{2a}, b, L)$ to replace α with b, 4.2. else if $\beta = \alpha$ (push back the popped α), then $\delta(q_{1ab}, \alpha) = (q_{2a}, \alpha, L)$

4.3. else β contains a new symbol to be pushed to stack, $\delta(q_{1ab}, \alpha) = (q_{1ab\alpha}, \alpha, r)$ to push back the popped α, move right to b, and $\delta(q_{1ab\alpha}, b) = (q_{2a}, \gamma, L)$ to push the new symbol $\gamma \in \beta$ to the top of stack, then move left,

5. $\delta(q_{2a}, t) = (q_{2a}, t, L)$ for all $t \in \Sigma$ to keep moving left until,
6. $\delta(q_{2a}, \dot{a}) = (q_2, \mathtt{x}, R)$ reaching at the input symbol $\dot{a}$ marked earlier; cross it out as processed with x, and proceed to the right to carry out the next computation for PDA.

3.2.3 DPDA

There is a deterministic variant of the PDAs called DPDA that is less powerful. Although we omit this class called the DCFL from our major discussions, it is worth noting that DCFL sits in between CFL and the regular languages. The algorithm above will accept and convert its definition into a TM, which will then have a degenerate, deterministic tree structure when computing.

The DPDA is less powerful due to it lacking the non-deterministic structure. It cannot explore different possibilities of computation history in parallel, and this imposes an even stronger restriction on the use of its stack memory − it can only be popped and pushed in a single way for a given instance of DPDA.

A more elegant interpretation is this: we can view a DPDA as a single thread in the whole tree of PDA which starts from the same root but ends in a different leaf of the tree. Therefore, a PDA is just a collection of multiple parallel instances of DPDA; a DPDA is an instance in the whole tree of PDA. Whereas a PDA accepts whenever one of its leaves accepts, a DPDA as a single thread will only accept if its single leaf accepts.

However, bear in mind that the presence of memory stack is crucial to setting DPDA and PDA apart, because when we remove the stack, we will restore equivalence between the deterministic and non-deterministic variants of a machine, as we shall see below.

3.3 Non-deterministic and Deterministic Finite Automata (NFA and DFA)

NFA and DFA have a similar tree-and-thread relationship: NFA is the whole tree, DFA is a single thread. Despite that, in contrast to PDA and NPDA, DFA and NFA are equivalent. The explanation is as follow:

DFA can simulate an NFA by gradually tracing out all of its possible thread, because a DFA a reversible. It consists only of an immutable input tape, and its head moves in a single direction, while its state transition is completely determined by its transition table. Say a DFA traverses down a path in the tree, it can go back up deterministically by taking the inverse of its δ transition function, which is invertible. This allows a DFA to traverse and thus simulate an NFA tree reliably without loss of information.

However, this equivalence does not apply for DPDA and PDA, because the computation for DPDA is not invertible. A DPDA cannot go back up its thread to traverse the other part of the PDA tree because there will be a loss of information when it cannot know whether it has previously pushed a stack symbol, or what it has popped. This makes a computation for PDA irreversible, and causes the DPDA to be less powerful.

The equivalence is restored for TM and NTM because the machines have full access to its memory. With some care they can prevent information loss and make computation reversible. This is reflected in the proof of equivalence between TM and NTM.

We now define the DFA, which is equivalent to the NFA analogous to how TM is equivalent to NTM via the tree structure.

Definition 3. *A **Turing Machine** is a 7-tuple:*

1. *Q: the non-empty set of machine states,*
2. *Σ: the non-empty set of tape symbols,*
3. *$\Gamma \subset \Sigma$: the set of input symbols,*
4. *b: the blank symbol,*
5. *$\delta : Q \times \Sigma \mapsto Q \times \Sigma \times \{L, R\}$: the transition table,*
6. *$q_0 \in Q$: the starting state,*
7. *$F \subset Q$: the set of accept states.*

As an equivalent, more physical description, the machine is a configuration which has:

1. *a state,*
2. *an countably infinite tape which serves as its memory,*
3. *a head that has random access to read and write on the tape.*

Definition 4. *A **map** is a connected, undirected planar graph, with 42 nodes, each representing a country. The nodes are named with indices $0 - 41$, and are connected the same way as are countries on the game board by undirected edge of weight 1.*

We assign data fields to each node, namely its country name, the continent it is in, its player owner, the number of armies of the owner in it, its worth and pressure as determined by some metric described below.

Definition 5. *A **region** is a connected subgraph consisting of nodes all owned by the same player. Each player can own many regions, which together partition the map.*

Definition 6. *A **border** node of an AI is its node that is adjacent to at least an enemy node.*

Definition 7. *A **attackable** node for an AI an enemy node adjacent to its border.*

Definition 8. *The **shape** of a region is the measure of its shape/roundess. To compute the shape, find the maximum and minimum distances between the border nodes in the region, and shape=(max-min)/max. If a region is round, shape = 0; if it is a line, shape = 1.*

Definition 9. ***Radius** is the measure of shortest distance from an origin node. We identify the neighbors of node $\mathcal{O}$ at radius k to be the nodes whose shortest distance from $\mathcal{O}$ is k.*

Furthermore, we define the fields that will be useful in our algorithms:

Definition 10. *The **worth** of a node is the measure of its importance to an AI, as calculated by its internal metric algorithm, and is used by the AI to prioritize its decisions: which node should it defend/attack first.*

Definition 11. *The **pressure** of a node as perceived by an AI is the measure of the average army distribution around the node, up to 5 unit radii away. It is calculated by the AI's internal metric and used to prioritize decisions.*

Note that the worth and pressure of a node are not the same when calculated by opposing AIs due to different perceptions, metric and AI personalities. Each AI will be calculating these values for all 42 nodes at each turn.

Finally, we introduce a data structure as the raw representation of overall army distribution on the map for various calculations:

Definition 12. *The **Radius Matrix (RM)** from an origin node $\mathcal{O}$ is the matrix that better represents the connectivity of neighbor nodes of the origin within some radius. It is enumerated by the Radius Matrix Algorithm below, and each entry is the name of some node.*

*Its corresponding **Army Matrix (AM)** is a different representation of the RM, with each entry now being $z \in \mathbb{Z}$, where $|z|$ is the number of armies at the node, and z is positive if the node is owned by the calculating AI, and negative otherwise.*

4 Algorithms

We now enumerate the algorithms for each step of the game, which will collectively form the final algorithm used by the AI to play the game.

4.1 The Matrix Algorithms

Algorithm **Radius Matrix (RM) for an origin node $\mathcal{O}$**

Starting from an origin node $\mathcal{O}$, initialize an empty matrix for its RM,

1. Add the index of each adjacent node (at radius 1) of $\mathcal{O}$ to a new row in RM.
2. Repeat for $i \in \{2, 3, ..., n\}$, where n is the maximum radius covered:

   For each entry p at column i, get all n_p of its adjacent nodes at radius $i + 1$ from $\mathcal{O}$.
3. Duplicate the row of entry p while appending to it each of the n_p adjacent nodes at column $i + 1$. If $n_p = 0$, append "*empty*" instead. The process is akin to a Cartesian product.
4. Return the RM for $\mathcal{O}$.

Note that the column number will coincide with the radius from $\mathcal{O}$. The RM with n columns is a representation of the connectivity from the origin up to radius n, where each row is the shortest path from the origin to a point at radius n, and there may exist many such paths.

Algorithm **Army Matrix (AM) for an origin node $\mathcal{O}$**

We can convert an RM into AM, a representation using the number of armies,

1. Find the RM for node $\mathcal{O}$ using the RM algorithm.
2. For each entry p in RM, if node p has the same owner as $\mathcal{O}$, replace the entry with the number of army at p; else, replace with the negative of the number of army at p. If an entry p is "*empty*", append 0 instead.
3. return the AM for $\mathcal{O}$.

This transforms an RM into its alternate form AM, which gives a representation of the army distribution and connectivity around the origin node $\mathcal{O}$. This matrix can be used for calculating the **pressure** from definition 11. For our project we calculate the matrices up to radius 5, which we think is sufficient given that per game turn a player can only move adjacently among nodes.

4.2 The Pressure Algorithm

The pressure of each node from an AI's point of view is the average number of army surrounding the node. More positive pressure indicates the node is a better stronghold of the AI; more negative pressure indicates is surrounded by more enemies.

The calculation of pressure depends on the AI's perception of threat, which can be represented using a metric that varies based on its personality.

Definition 13. *The **threat perception** of an AI is the way it sees the threat of army distribution up to some radius away poses on an origin node. E.g. 10 enemy armies further away poses less threat than 5 enemy armies nearby. The **threat perception** is quantified by defining a metric: a normalized vector or length = max radius of AM, where the individual value of the vector is the weight multiplied to the army number at that radius. The procedure is describe below.*

Algorithm **The Metric Algorithm**

To enumerate the metric for an AI's threat perception, with scope radius = 5,

1. Choose a weight function, for example, constant, Gaussian,
2. Evaluate function values for with the input distance vector $\{1, 2, 3, 4, 5\}$
3. Renormalize the output vector and return it as the metric vector.

This metric vector $\mathbf{w}$ is then dotted with a row $\mathbf{r}$ in the AM, which is a list of number of armies at incremental distance away from an origin node $\mathcal{O}$, and the partial pressure PP for it is:

$$PP(\mathbf{r}) = \mathbf{w} \cdot \mathbf{r}$$

Algorithm **The Pressure Algorithm**

The AI calculates the pressure for each node using its personality trait **threat-perception**, or the metric vector $\mathbf{w}$:

1. Update the data fields of the map and call the AM algorithm to compute the AMs for all 42 nodes.
2. For each node $\mathcal{O}$, compute the dot product between $\mathbf{w}$ and each row of the node's AM; the result is a column vector $\mathbf{c}$.
3. The first column of the original RM is a repeated list of m adjacent nodes of $\mathcal{O}$, suppose each node i repeats q_i times in the column, so in total the column has length

$q_1 + q_2 + \cdots + q_m$. Renormalize this sequence into $nq_1 + nq_2 + \cdots + nq_m$ For each batch q_i of the column vector $\mathbf{c}$ from above, take its mean, then multiply by the renormalized weight nq_i. Then sum all m of the results, call this scalar $s(\mathcal{O})$.

4. Now that the column $\mathbf{c}$ has been reduced to a scalar representing the average army distribution around the origin $\mathcal{O}$, account for the number of armies (sign-sensitive, negative for enemy) here $a(\mathcal{O})$ by adding the scalar, and return the pressure of node $\mathcal{O}$, $P(\mathcal{O}) = s(\mathcal{O}) + a(\mathcal{O})$.

Thus at each turn, the AI updates the data fields and calculates the pressure, i.e. the average number of surrounding armies, for each node, using its threat perception metric.

4.3 The Worth Algorithm

Algorithm **The Worth Algorithm**

At each turn, the AI evaluates the worth of each node to prioritize its attacks and defenses. Suppose it considers m factors, each of which assumes a real positive value, with more positive being more worthy. To compute the final worth scalar, simply order the m factors from the most vital, and dot it with a factor vector $\{10^{m-1}, ..., 100, 10, 1\}$.

For our AI, we consider the following factors (ordered from the most important). For each node $\mathcal{O}$ calculate and append to the list of factors:

1. continent-fraction $= \frac{\text{(number of nodes with the same owner in the same continent } \mathcal{O}\text{)}}{\text{(total number of nodes in the continent)}}$
2. If $\mathcal{O}$ is own node, the region-index: Enumerate for each player its nodes, and group them by regions, then order them from the biggest to the smallest regions. The region-index of $\mathcal{O}$ is its index in this list. Or if $\mathcal{O}$ is enemy, the attackable index: of the region list enumerated above, extract the sublist with nodes that are attackable, i.e. is an enemy adjacent to one of your nodes. The attackable index of $\mathcal{O}$ is its index in this sublist; -1 otherwise.
3. shape: find the region $\mathcal{O}$ is in and compute the shape as in definition 8.
4. degree: the degree of $\mathcal{O}$, i.e. the number of adjacent nodes it has.
5. pressure: as calculated from the pressure algorithm.
6. Finally, return the dot product between this factor list and $\{10^4, 1000, 100, 10, 1\}$.

After obtaining an ordered list of worth nodes, we can partition it while preserving the order into lists of border nodes and attackable nodes, and reorder them based on strategies.

Furthermore, the AI makes it context-sensitive by remembering the pressures from the past turn, and reorder the list based on pressure-drop between turns.

To justify our factors above, conquering a whole continent gives a player extra armies per turn while strengthening the region. Furthermore, a larger region is harder to attack than a smaller region, thus we place more importance on the nodes there.

The shape of a region is vital for defense and army mobility during fortification. This is the classic problem of minimizing the surface/volume ratio, or in this lower dimensional case, the perimeter/surface ratio. Intuitively, a thin region is vulnerable, and has bad army mobility. A thick, concentric shape is stronger. The average distance between nodes is also shorter and thus aids mobility. Moreover, a node with higher degree improves mobility since it can reach many other nodes.

The pressure measures the ease of attacking or defending a node; the less negative a node is, the less enemy presence it has. It is wise to not attack the enemy's stronghold, but to seek its weak point of entry, which can be detected from a less-negative pressure.

4.4 The Priority Algorithm

At each game turn, the AI updates the priority nodes to attack/defend. The list of priority nodes depends on the AI's personality trait **priority**, whether it is agressive (attack-then-defend) or defensive (defend-then-attack).

Algorithm **The Priority Algorithm**

1. Update the data fields for the AI.
2. Call the pressure algorithm on the map.
3. Call the worth algorithm on the map.
4. Repartition the worth nodes and reorder by attackables/borders first based on the AI's personality, whether agressive or defensive. Furthermore, for the attackable, choose the best origin of attack by the highest pressure.

4.5 The Placement Algorithm

This describes how the AI places the armies into the its priority nodes based on its personality trait **placement**.

Algorithm **The Placement Algorithm**

1. If the trait is **cautious**, place armies along its priority nodes (if is enemy, use the best origin of attack) until all pressures are > 0, then with the extra armies, place 4 each down the same list; repeat until none left.
2. If the trait is **tactical**, place armies down the list until node pressure is > 4, then with extra, place 4 each down the list; repeat until none left.

4.6 The Attack Algorithm

The AI decides to launch attacks from the best attack origin (calculate with in priority list) based on its personality trait **attack**.

Algorithm **The Attack Algorithm**

For all attackables down the priority list,

1. If the trait is **rusher**, the AI harassess constantly, i.e. while the best attack origin has 2 more armies than the enemy target, keep attacking before moving to next target.
2. If the trait is **carry**, same as above, but the difference threshold is 4 (higher). Furthermore, the AI will accumulate the cards to reserve more armies for late game.

4.7 The Fortifying Algorithm

All AIs use the same fortifying algorithm.

Algorithm **The Fortifying Algorithm**

1. Find the border node $\mathcal{O}$ with the lowest pressure, and find a non-border ally node with higher pressure, transfer all but 1 troop to the border node if possible. This is to always push the unused central forces out to the borders where armies are mostly needed.
2. If no fortification done above, find a border node with the highest pressure, and transfer any neighboring armies (all but 1) to it. This is for the accumulation of armies during late game by making strong node even stronger.

This algorithm aims to create a center-weak border-strong army distribution within a region with, that is to utilize the maximum number of armies by putting them to the border nodes. Such distribution is efficient for both offense and defense. The second part of the algorithm kicks in when there is less border nodes during the late game, when one player controls a larger region. This will accumulate all armies toward a border node to overwhelm the enemy, hopefully ending the game quicker.

5 AI Algorithms and Personalities

We now put everything together to form the AI, which has four personality traits parametrized in its algorithms:

1. The **threat perception** trait /metric in the Pressure Algorithm; function variations: {Constant, Survival}.
2. The **priority** trait in the Priority Algorithm; variations: {agressive, defensive}.
3. The **placement** trait in the Placement Algorithm; variations: {cautious, tactical}.
4. The **attack** trait in the Attack Algorithm; variations: {rusher, carry}.

Thus there are $2^4 = 16$ AI personalities, and more if we allow richer variations.

Corresponding to the game moves per turn:

1. Getting and placing new armies;
2. Attacking, if you choose to, by rolling the dice;
3. Fortifying your position (moving troops between an adjacent pair of your nodes),

the AI has these primary methods:

1. Update: Call the Priority and Worth algorithms.
2. Get-and-Place-Armies: Call the Placement Algorithm.
3. Attack: Call the Attack Algorithm as many times as wanted.
4. Fortify: Call the Fortify Algorithm.

A game can have as many participating AIs as permitted by the rules. During the initial game setup, countries are randomly assigned to the AIs. Then, the AIs take turn to call their Update and Get-and-Place-Armies methods to complete the setup. Then the game begins and the AIs take turn to call all their primary methods in sequence, until the game terminates, or ties at the maximum number of rounds.

As opposed to physical game, the virtual game has no limit on the number of army pieces − it just keeps creating more as needed; nor it has the limit on the cards − it keeps reshuffling a new deck once an old one runs out.

6 Our Experiments

We begin our study from the simplest case: games with 2 players. Note that according to the rules, there is an inactive neutral player, and we mimic that by adding a third AI that is muted.

7 Conclusion

This paper sets out to solve the problem of the identification of the class of a block code. We do so by introducing a new *Canonical Bundled Form* as a unique class representation of the block code.

The Bundled Form and its algorithm too solves the special problem of determining the equivalence between matrices under column/row swapping, and the general problem which allows column-wise letter-permutation to the sub-problem. Row-permutation can be done by transposing the matrices.

8 Citations

H. Fripertinger. Enumeration, construction and random generation of block codes. *Designs, Codes and Cryptography,* Volume 14 Issue 3: 213-219, 1998.