AI for Risk Game

Wah Loon Keng* Benjamin H. Draves†

April 19, 2015

Abstract

We implement an AI to play the board game *Risk*. The game is formalized as an optimization problem in graph theory, where countries are represented as nodes in an undirected graph, and decisions are considered based on the graph properties. We introduce solution algorithms, with variable parametrizations, which are then implement as AIs with different personalities in `JavaScript`. We set up games among AIs to measure their performances, and discover unusual game strategies.

1 Introduction

The board game *World Domination RISK* ® is a game of military strategy, where players of different factions try to conquer all 42 countries on the map, by deploying armies to attack and defend. We use the classic *Hasbro* version of the game; the rules are well known and can easily be found online, but they will be included as we walk through our algorithms.

In this paper, we first set up our formalization of the game as a graph optimization problem. Next we introduce graph algorithms to carry out each of the game moves, and explain our reasoning behind them. Then, we combine these to implement an AI, with variable personalities based on the parametrization of its internal algorithms. The different AIs then play multiple games against one another, with their results recorded. Finally, we analyze their performances, and find several unusual, interesting strategies discovered by the AIs that are never observed from human players.

This entire project is public on GitHub: `https://github.com/kengz/Risk-game`.

*Lafayette College, Easton, PA 18042, USA. kengw@lafayette.edu.
†Lafayette College, Easton, PA 18042, USA. dravesb@lafayette.edu.

2 Formalization

The game is inherently dependent on the board, which is a world map of 42 countries, interconnected in specific ways. Decisions to attack or defend are based on the distribution of the armies, the surroundings of a location, and connectivity of countries. These motivate our formalizing the game board and algorithms based on an undirected graph. From now we shall refer to the graph representation as *map*:

Definition 1. *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 2. *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 3. *A **border** node of an AI is its node that is adjacent to at least an enemy node.*

Definition 4. *A **attackable** node for an AI an enemy node adjacent to its border.*

Definition 5. *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 6. ***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 7. *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 8. *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 9. *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.*

3 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.

3.1 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 8. 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.

3.2 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 10. *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.

3.3 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 $= \frac{\text{(number of nodes with the same owner in the same continent } \mathcal{O})}{\text{(total number of nodes in the continent)}}$
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 5.
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.

3.4 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.

3.5 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.

3.6 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.

3.7 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.

4 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.

5 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.

Draft:

1. Formalization of problem

2. Algorithms and decisions

3. AI Implementation and variations

4. results, performance, analysis

5. AI behaviors, surprises

6 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.

7 Citations

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