Discovering and Searching Loosely Coupled Subproblems in Dou Shou Qi

Joseph Burnett

May 27, 2010

Abstract

We present a technique for discovering, isolating and searching loosely coupled subproblems in the game of Dou Shou Qi. Subproblem discovery utilizes a minimum weight spanning tree to find clustering structure [10] in a complete graph of causality, which captures the distance between pieces in terms of a lower-bound on time-to-effect. A heuristic extends the duration of the split by isolating subproblems, effectively trimming branches of the game tree which do not conform to the clustering structure. Searching loosely coupled subproblems in Go has been explored by Berlekamp [2]. Basic Dou Shou Qi strategy is outlined and used to develop a position evaluation function for the minimax algorithm. Additionally an algorithm to update a minimum spanning tree in $O(n)$ time per off-line set of neighborhood updates is proposed, conditional upon its being rooted in some simple transform of a Euclidean or Manhattan metric, and upon a maximum edge weight change which is constantly upper-bound. The minimum spanning tree updating algorithm, which uses simple data structures, improves on the more general result of Frederickson's $O(\sqrt{|E|})$ time per update [5] by allowing $n-1$ neighborhood updates to occur in $O(n)$ time on a complete graph, effectively in constant time per update.

1 Introduction

In this thesis we are going to explore a method of breaking apart a board game into subproblems to allow the minimax algorithm to search a position more efficiently [7]. The space in which we will explore this technique is the Chinese game of Dou Shou Qi (pronounced *doe show chee*) which is popular among Chinese children and is similar to chess [1]. We will first introduce the history, rules and strategy of Dou Shou Qi, and then propose a heuristic position evaluation function. Then we will define our problem and establish the causality space in which we will use a minimum spanning tree to discover subproblems. We will also propose an algorithm to maintain a minimum spanning tree in linear time per ply, and a heuristic to artificially maintain a partition, allowing deeper searching of the game tree. After outlining the logistics of running a split search and the issues involved in implementation, we will present the results of our testing and our analysis of the data.

1.1 Definitions

Graph A *graph* is a set of *vertices* (points) and *edges* (lines between points). An edge is a relationship between two vertices and can have a numeric *weight* associated with it. Two vertices connected by an edge are said to be *adjacent* to one another. All our graphs will be *simple graphs* (zero or one edge per pair of vertices and exactly two distinct vertices per edge) and *undirected* (edges go both ways). A *complete graph* has an edge for every pair of vertices, where the edges number exactly $\frac{1}{2}(n^2 - n)$ and n is the number of vertices in the graph.

Minimum Spanning Tree A *spanning tree* is a graph in which there is a path between every pair of vertices and there are no *cycles* (more than one path between two vertices). A spanning tree of a graph, G, will include all of its vertices and $n-1$ of its edges. A *minimum spanning tree* is a spanning tree of a graph such that the sum of the minimum spanning tree edge weights is less-than or equal-to the sum of the edge weights of any other spanning tree of G. Minimum spanning trees are not always unique.

Game Tree A *game tree* is a map of every possible sequence of moves in a game, where vertices are positions and edges are moves. The *branching factor* of the tree is the maximum number of moves available at any position. Game trees are usually terminated at a predetermined depth because there are an exponential number of vertices, too many to extend the tree to the end of the game. In fact, because the rules of Dou Shou Qi allow the players to repeat their moves endlessly, the game tree is infinite. Positions at the leaves are called *terminal positions.*

Depth First Search A *depth first search* is a search algorithm for a game tree which explores the subtree below each move from a position until it reaches a terminal position. Then it backtracks to the last position with unexplored moves, explores the next move, and repeats until all positions have been searched.

Zero-Sum Game A *zero-sum game* is a closed system in which everything won by a player is lost by another player[11, page 11]. Dou Shou Qi and chess are both zero-sum games because only one player can win.

a9	b9	c9	d9	e9	f9	g9
a8	b8	c8	d8	e8	f8	g8
a7	b7	c7	d7	e7	f7	g7
a6	b6	c6	d6	e6	f6	g6
a5	b5	c5	d5	e5	f5	g5
a4	b4	c4	d4	e4	f4	g4
a3	b3	c3	d3	e3	f3	g3
a2	b2	c2	d2	e2	f2	g2
a1	b1	c1	d1	e1	f1	g1

Figure 1: Algebraic notation for the Dou Shou Qi board. Columns are identified by the letters `a` through `f` and rows identified by the numbers `1` through `9` with square `a1` in the lower-left corner of the board.

Minimax Algorithm *Minimax* is an exponential search algorithm based on the Minimax Principle of game theory. It considers every possible move from a given position to a predetermined depth by running a depth first search on the game tree and scoring the terminal nodes with an evaluation function. As Minimax searches, it alternately minimizes and maximizes the best possible score at each *ply* (level of the tree) to find the *continuation* (sequence of moves) which leads to the best possible terminal score.

Coupling *Coupling* is how two objects interact with each other. *Tight coupling* is when there are a lot of dependencies between objects and *loose coupling* is when dependencies are sparse. *Loosely coupled subproblems* are subsets of a game's pieces that have little or nothing to do with each other for some fixed period of time.

1.2 Conventions

Players A and B We will refer to the two players of Dou Shou Qi as players A and B. Player A's pieces are red and begin on the lower half of the board, while player B's pieces are black and begin on the upper half of the board. Player A makes the first move at the beginning of the game and all of our examples are written such that A is the next player to move.

Algebraic Notation We use the algebraic notation of chess to address the squares of the Dou Shou Qi board. Figure 1 gives the address of each square. Columns are identified by the letters `a` through `f` and rows identified by the numbers `1` through `9` with square `a1` in the lower-left corner of the board.

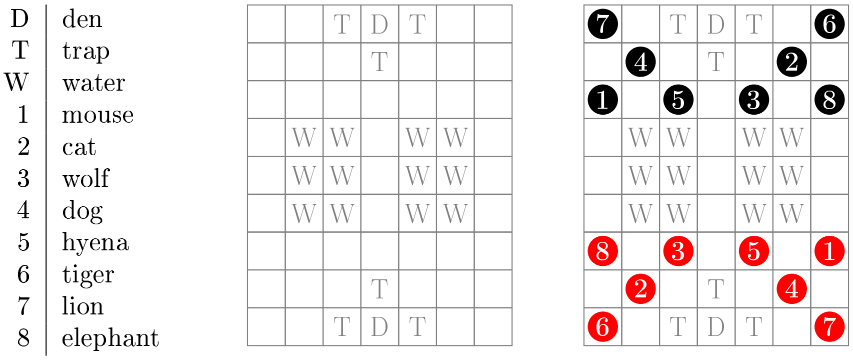
The Editorial 'We' This thesis uses the editorial 'we' because it conveys a sense that the reader is along for the ride and is sharing ownership of the material. Hopefully this document is accessible and entertaining enough to make this a reality.

2 Dou Shou Qi

The game of Dou Shou Qi is likely derived from Xiang Qi, the Chinese version of chess. It has been suggested that the modern game of Stratego took some of its form from Dou Shou Qi [1] and there are similarities between the two, such as the pivotal role of the spy and the mouse in Stratego and Dou Shou Qi, respectively. The signature feature of the Stratego terrain, two central bodies of water, is also present on the Dou Shou Qi board. However, in Dou Shou Qi all of the pieces are known, making it a game of complete knowledge, unlike Stratego in which there is a large element of deception [3]. From a game theoretical perspective, this makes the two games highly dissimilar.

For the purposes of this paper, the history of the game and its cultural content are of no consequence. Dou Shou Qi was selected primarily because it is sufficiently complex to make a polynomial-time algorithmic solution infeasible while remaining simple enough to provide a rich environment for the development of a new heuristic. Unlike the game of chess, Dou Shou Qi strategy and theory is relatively unexplored.

2.1 Rules of the Game



D	den
T	trap
W	water
1	mouse
2	cat
3	wolf
4	dog
5	hyena
6	tiger
7	lion
8	elephant

(a) The squares and pieces

		T	D	T		
			T			
	W	W		W	W	
	W	W		W	W	
	W	W		W	W	
			T			
		T	D	T		

(b) The board

❼		T	D	T		❻
	❹		T		❷	
❶		❺		❸		❽
	W	W		W	W	
	W	W		W	W	
	W	W		W	W	
⑧		③		⑤		①
	②		T		④	
⑥		T	D	T		⑦

(c) The initial setup

Figure 2: The game of Dou Shou Qi. Pieces are labelled with their strength from the mouse (1) to the elephant (8). The seven by nine Dou Shou Qi board has two bodies of water (W) in the middle and two dens (D) at the top and bottom, surrounded by traps (T). The initial setup is always the same with player A's red pieces in the bottom half and player B's black pieces in the top half.

Dou Shou Qi is a two-player, zero-sum strategy game that looks like a cross between chess, checkers and Stratego in which the players move vertically and horizontally. Pieces are labelled with their strength from the mouse (1) to the elephant (8). The seven by nine Dou Shou Qi board has two bodies of water (W) in the middle and two dens (D) at the top and bottom, surrounded by traps (T). The initial setup is always the same with player A's red pieces in the bottom half and player B's black pieces in the top half (see Figure 2). The objective of the game is to place a piece in the opponent's den which is surrounded by traps. Each of the pieces represents a jungle animal and is numbered in a hierarchy from 1 to 8 according to its relative strength. The pieces of the game are as follows: the mouse (1), cat (2), wolf (3), dog (4), hyena (5), tiger (6), lion (7) and elephant (8). (On all diagrams we will label the pieces by their strength for ease of reference.) Each animal can take those pieces of equal or lesser strength with two exceptions: the elephant can take any piece except the mouse, and the mouse can take the elephant.

While standing on an opponent's trap a piece has an effective strength of zero and is therefore vulnerable to capture by any opposing piece; however, a player's own traps have no adverse effects. Water squares can be occupied only by mice (1) which cannot attack across the land-water boundary. The lion (7) and tiger (6) can jump vertically or horizontally across the water (even capturing while doing so) but cannot jump over a square occupied by any mouse. The den can only be occupied by an opposing piece at which point the game is over.

2.2 Strategy and Position Evaluation

An analysis of the strategy of Dou Shou Qi is necessary to define an objective function that can input a position and assign it a numerical value based on some set of strategic considerations. Such a function is used to evaluate the terminal nodes of a minimax search.

The basics of Dou Shou Qi gameplay were learned by printing out the board on a 8.5 by 11 inch sheet of paper, labelling a set of eight pennies and eight nickels with the numbers 1 through 8, and hauling the whole arrangement to every coffee shop, dinner party and pub that the author was invited to. That is to say, the strategy of Dou Shou Qi presented in this thesis is just a starting point and does not benefit from a dedicated body of theoretical research or even an expert player, so we will start with the more familiar game of chess and assimilate the relevant elements of its strategy.

2.2.1 Chess Strategy

Czechoslovak Grandmaster, Ludek Pachman, defines six factors in his book Modern Chess Strategy that can be used to evaluate a Chess position [12, page 2]:

1. The material relationship; that is, material equality or the material superiority of one side.
2. The power of the individual pieces.
3. The quality of the individual pawns.
4. The position of the pawns; that is, the pawn structure.
5. The position of the kings.
6. Co-operation amongst the pieces and pawns.

Pachman's first factor is directly applicable to Dou Shou Qi in which material superiority is enough to tip the balance in favor of one player or another. This is partly due to the fact that any piece may enter the opponent's den and therefore even the smallest piece can be important in maintaining the balance of power. Pachman's second factor can be incorporated into the first by considering the relative worth of each piece. In chess a rough standard of material worth evaluates the pieces in terms of one pawn. A knight or bishop is worth 3, a rook is worth 5, a queen is worth 9 and a king has infinite worth because he is the objective of the game [12, page 11]. Likewise, we can place relative values on the pieces of Dou Shou Qi and use those values to calculate the material relationship of a position. The absolute value of such assignments does not matter; only the ratio between them is important.

Dou Shou Qi does not have pawns (a piece which can move forward only, attacks along a diagonal and requires the support of other pawns,) but it does have pieces whose quality increase and decrease with position and material. The mouse (1) is the weakest piece and cannot hold its own against the others. However, it is the only piece capable of swimming in the water, blocking the jump of a tiger (6) or lion (7) and, most importantly, taking an elephant (8). The importance of the mouse in counterbalancing the strength of the elephant is acknowledged in their initial positions, directly across from each other. The most important job of a mouse is to keep the elephant in check. For that reason, an elephant which does not have an opposing mouse can be considered a piece of "higher" quality. Likewise, a mouse without an opposing elephant is a piece of "lower" quality because it cannot fulfill its primary role. Additionally, a mouse that has been "passed" or is further from its den than an opposing elephant is severely degraded and therefore of lesser quality. In this small way, Pachman's third and fourth factors have a parallel consideration in Dou Shou Qi strategy.

The position of the kings (or dens,) is irrelevant because the game's objective is in a fixed position for both players. However, cooperation among the pieces is important. Consider the vulnerability of the mouse, who must be accompanied or protected by a stronger piece in order to pursue the elephant.

2.2.2 Dou Shou Qi Strategy

Through the adaptation of Pachman's chess factors, we propose four factors with which to evaluate a Dou Shou Qi position.

Material relationship is exactly as Pachman defined, the equality or superiority of one side's material or the other's. Evaluation of material relationship is a simple matter of assigning a value to each piece type and tallying the pieces of both sides.

Development of the pieces takes into account the time required to move a piece into its strongest position. For example, a lion and tiger are often most effective when placed at the edge of the water where they can leap across into enemy territory, and so a lion or tiger at the base of the water is "more developed". Likewise, the mouse is often most effective in the water where it can block jumping animals and is not vulnerable to attack by stronger pieces. Development is the general notion that pieces have had time to get where they need to go. It is also the sum of the development of each individual piece, therefore each piece is a subproblem that can be evaluated independently and rapidly. (There are at most 62 possible places for a given piece to be.)

Quality of the elephant and mouse consists of observing whether each elephant has an opposing mouse and which is closer to the mouse's den. Mouse and elephant quality involves tallying and measuring distance, also a trivial task.

Cooperation among the pieces is a measure of how the pieces support each other. It is the most difficult factor to represent and evaluate because cooperation could involve any combination of 8 pieces on any subset of the 62 squares. It requires a higher level of thinking than the other factors because it takes into account the power structure of the board as well as possible attacking and defending scenarios.

Limited aspects of cooperation can be encoded in the development factor by making squares "highly developed" for certain pieces which, when occupied by those pieces, also creates a supporting relationship. For example, from the starting position, moving the elephant (8) from `a3` to `b3` and the cat (2) from `b2` to `a2` creates a supporting relationship between the cat, which protects the elephant from the opposing mouse (1), and the elephant which protects the cat from the lion (7) who could leap over the water. Therefore the square `b3` can be considered developed for an elephant and the square `a2` developed for a cat.

In our position evaluation function, we will use the first two factors because they are the most versatile and the easiest to represent:

1. Material relationship.
2. Development of the pieces.

2.3 Representing Dou Shou Qi Strategy

Representation of Dou Shou Qi material factors is easily accomplished by assigning a positive, integer value to each type of piece. Development factors differ from piece-to-piece, so we can assign an integer value to each square for each type that can occupy it. A piece-by-piece analysis of basic Dou Shou Qi strategy suggests reasonable values for each of the tables, given in Figure 3.

2.3.1 Assigning Material and Development Values

Each piece has its own unique characteristics and roles to play in the game as a function of its strength and initial position. Our classification of these pieces is based on observation and experience playing the game and is therefore intuitive and heuristic in nature. We assign pieces a material value on a scale from 200 to 1000 and squares a development value on a scale from 0 to 50 (with the exception of the den which is valued at $+\infty$.) The material scale is higher because, in general, having pieces is more important than occupying squares.

By convention we assign the starting position of a piece the value of 10. Values generally increase toward the opposing den because, in the absence of other considerations, moving forward is a good thing. A game cannot be won without advancing some of a player's pieces across the board to the opposing den.

Piece	Value
Mouse–1	500
Cat–2	200
Wolf–3	300
Dog–4	400
Hyena–5	500
Tiger–6	800
Lion–7	900
Elephant–8	1000

(a) Material Values – *Material*⟨*t*⟩

11	13	50	∞	50	13	13
11	12	13	50	13	13	13
10	11	11	13	13	13	13
8	9	9	11	12	12	13
8	9	9	11	12	12	12
8	9	9	10	12	12	11
8	8	8	9	10	10	10
8	8	8	9	9	9	9
8	8	8	0	8	8	8

(b) Mouse Values – *Development*⟨1, *s*⟩

11	15	50	∞	50	15	11
11	11	15	50	15	11	11
10	11	11	15	11	11	10
10	0	0	10	0	0	8
10	0	0	8	0	0	8
10	0	0	8	0	0	8
10	10	10	8	8	8	8
13	10	8	8	8	8	8
8	8	8	0	8	8	8

(c) Cat Values – *Development*⟨2, *s*⟩

11	15	50	∞	50	15	11
10	11	15	50	15	11	10
9	10	11	15	11	10	9
9	0	0	10	0	0	9
8	0	0	8	0	0	8
8	0	0	8	0	0	8
8	8	10	8	8	8	8
8	12	13	8	8	8	8
8	12	12	0	8	8	8

(d) Wolf Values – *Development*⟨3, *s*⟩

11	15	50	∞	50	15	11
10	11	15	50	15	11	10
9	10	11	15	11	10	9
9	0	0	10	0	0	9
8	0	0	8	0	0	8
8	0	0	8	0	0	8
8	8	8	8	8	8	8
8	8	8	8	13	10	8
8	8	8	0	12	12	8

(e) Dog Values – *Development*⟨4, *s*⟩

14	15	50	∞	50	15	14
13	14	15	50	15	14	13
13	13	14	15	14	13	13
12	0	0	15	0	0	12
11	0	0	14	0	0	11
10	0	0	13	0	0	10
9	9	9	10	10	9	9
9	9	9	9	9	9	9
9	9	9	0	9	9	9

(f) Hyena Values – *Development*⟨5, *s*⟩

25	30	50	∞	50	30	25
25	25	30	50	30	25	25
18	20	20	30	20	20	18
15	0	0	15	0	0	15
15	0	0	15	0	0	15
15	0	0	15	0	0	15
14	16	16	14	16	16	14
12	14	12	12	12	12	12
10	12	12	0	12	12	10

(g) Tiger Values – *Development*⟨6, *s*⟩

25	30	50	∞	50	30	25
25	25	30	50	30	25	25
18	20	20	30	20	20	18
15	0	0	15	0	0	15
15	0	0	15	0	0	15
15	0	0	15	0	0	15
14	16	16	14	16	16	14
12	12	12	12	12	14	12
10	12	12	0	12	12	10

(h) Lion Values – *Development*⟨7, *s*⟩

25	30	50	∞	50	30	25
25	25	30	50	30	25	25
18	20	20	30	20	20	18
16	0	0	16	0	0	16
14	0	0	14	0	0	14
12	0	0	12	0	0	12
10	15	14	14	14	14	12
11	11	11	11	11	11	11
11	11	11	0	11	11	11

(i) Elephant Values – *Development*⟨8, *s*⟩

Figure 3: Dou Shou Qi positional and development values. Each piece has a material value from 200 to 1000 and each square for each piece has a development value from 0 to 50 (and infinity for opponent's den.)

The Mouse (1) The mouse's primary job is to oppose the elephant. If player B's elephant were to try and circumvent player A's mouse by using the center or opposite aisles[1], A's elephant would have plenty of time to intercept. Therefore the mouse should remain on its initial side. Moving into the water is a good development because it protects the mouse from the attack of stronger pieces and allows it to block the jumping lion and tiger. We assign the mouse a material value of 500 and the squares g3, f3 and e3 a development value of 10 because movement along the bottom of the water is strategically neutral. The squares on the right side of the board are more highly valued to encourage the mouse to stay on that side. All of the water squares on the right side are slightly elevated at a value of 12 to encourage the mouse to seek safety in the water.

The Cat (2) The cat is a relatively weak piece but its placement opposite the mouse provides it with a valuable supporting role. We assign it a material value of 200 and the square a2 a value of 13 to encourage the cat to move into a supporting position for the elephant (which will highly value the square b3.) Most of the other squares around the cat are valued at 10 because they are strategically neutral.

The Wolf (3) Because of the defensive nature of the cat, the wolf is unlikely to be a pivotal defender because the only pieces it can master, the mouse and the cat, will remain on their own side. The wolf is also not a useful aggressor because of its relative weakness. However, because an opposing piece must pass through a trap to enter the den, the wolf can play a guarding role by occupying square c2 which is adjacent to two traps. The wolf is assigned a material value of 300 and the square c2 a development value of 13. The four squares of b2, c2, b1 and c1 are elevated to encourage the wolf to move away from the water, allowing the elephant and tiger to maneuver.

The Dog (4) The dog, like the wolf, is not a strong enough piece to play a central role as an aggressor. It is best utilized as a guard for the two traps adjacent to square e2. If a piece is to be moved up the center aisle, the hyena is the logical choice because it is as close as the dog and is the stronger piece. The dog is valued at 400 and it is kept out the way by elevated values at squares e2, f2, e1 and f1.

The Hyena (5) The hyena is the best piece to use in the center aisle because it can stop the advance of any piece except the lion and tiger, who are more likely to be jumping over the water, or the elephant, who is more likely to be lumbering up the side. A developed hyena is usually centered at square d3 or moving forward along the center aisle. The hyena is assigned a material value of 500 and the center aisle squares are increasing rapidly in value as they move toward the opposing den.

The Tiger (6) Because of its jumping capabilities, the tiger is an excellent offensive piece. Positioned at the base of the water, it can be on the opposing side in one move. In the center aisle it can block the advance of any lesser piece through all three aisles. When on the enemy's side, a tiger can leap back across the water to safety, provided there is not a mouse blocking its path, and can be used to exert pressure on the enemy position without committing to the slow and long-lasting movements of the elephant. For these reasons, positions along side the water are considered developed and positions on the enemy's side

[1] the narrow passages between the water (squares a4, a5, a6, d4, d5, d6, g4, g5 and g6)

are highly valued. The tiger is assigned a material value of 800 and the base of the water (squares b2, c3, e3 and f3) is valued at 16. All squares across the water are at least 20.

The Lion (7) The lion has all the benefits of a tiger with the added capability of forcing a tiger from the middle aisle or its own territory. The lion is an excellent piece to have across the water and is most valuable in assaulting the opposing den, so its development is similar to the tiger's. It is assigned a material value of 900 and, like the tiger, the squares at the base of the water are valued at 16 and the far side of the water at least 20.

The Elephant (8) Significant development of the elephant can only occur when its opposing mouse is either taken or by-passed. For that reason, the elephant's initial development is to assume a defensive posture against an aggressing lion at square b3. However, once its opposing mouse has been neutralized, the elephant can move forward into enemy territory to cause massive disruption that can only be stopped by the opposing elephant. An elephant is valued at 1000 and the square b3 assigned a development value of 15. The aisles are rapidly increasing in value and the squares past the water on the opponent's side are valued at least 18.

3 Minimax

Minimax is a versatile algorithm for computing a sequence of moves, called a principal continuation, which will lead to the least amount of regret[7]. It is based on the minimax principle which is the first and most important proof of Game Theory, first given in John von Neumann and Oskar Morgenstern's seminal work, *Theory of Games and Economic Behavior*[16]. An understanding of basic game theory is necessary to appreciate the theoretical underpinnings of the minimax algorithm.

3.1 The Minimax Principle of Game Theory

Game theory is not a field that concerns itself with the play of a particular game but rather with all games[13]. Consider the game of "Matching Pennies" in which two players secretly place a penny on the table, either head's up or tail's up, and then simultaneously reveal their choice to their opponent. Player A will win the game if the pennies match sides, while player B will win if they do not. Since the game is two-player and zero-sum, the pay-off structure can be stated in terms of benefit to player A.

Figure 4 (a) shows the pay-off structure for player A and B as a two-by-two matrix in which each strategy provides the opportunity to either win or lose 1, depending on what strategy their opponent chooses. Each player will rationally know that their opponent will select the strategy with the best pay-off and will choose the best strategy to counter. There can be pairs of strategies that will minimize the damage for both parties which are called saddle points. At such points both players can minimize their maximum loss, thus the name of the minimax principle. Figure 4 (b) shows a slightly modified pay-off structure which creates a saddle point when player A chooses the tails strategy and player B chooses the heads strategy.

Anatol Rapoport explains the meaning of the minimax principle in terms of saddle points[13, page 60]:

> If a two-person zero-sum game has a saddle point, the best each player can do (assuming both to be rational) is to choose the strategy which contains a saddle point.

	A_h	A_t
B_h	1	-1
B_t	-1	1

(a) Penny game pay-off structure

	A_h	A_t
B_h	-2	**-1**
B_t	-1	4

(b) Pay-off structure with saddle point at A_t, B_h

Figure 4: The game of matching pennies. Figure (b) has a saddle point when player A chooses the tails strategy and player B chooses the heads strategy.

Matching pennies is an example of game with imperfect information because each player is unaware of the move their opponent has selected until both have committed to their respective moves. A game of perfect information is one in which both players know the exact state of the game at any point. Chess and Dou Shou Qi are both games of perfect information because the players take turns and can see all the pieces. Games of perfect information also have saddle points which remain the best pair of strategies for both players [13, page 62]. Each strategy contains all of the moves that will be made in the face of every move of the opponent [13, page 44] so for a game such as chess or Dou Shou Qi, the number of strategies is exponential in the number of plies until the game is ended. As long as there are a finite number of plies in the game, the number of strategies is also finite, but they are just too many to compute. If there are only two moves available at each ply and a game lasts 50 plies, the number of strategies for each player is more than the number of atoms in the earth, an unmanageable amount of information.

3.2 The Minimax Algorithm

The minimax algorithm stops the play at a certain depth (number of plies) and creates an artificial pay-off structure through a position evaluation function. Winning the game is still paramount so we assign a win for A the value of $+\infty$ and a win for B the value of $-\infty$ so that our evaluation function can spit out whatever numbers it wants and winning will still be the best pay-off.

The following is the pseudocode for the minimax algorithm [17][7]:

```
MINIMAX(position, depth)
if game is over or $depth = 0$
        do return EVALUATE(position)
$a \leftarrow -\infty$
for each $move \in position$
        do $p' \leftarrow$ MAKE-MOVE(position, move)
            $a \leftarrow max(a, -\text{MINIMAX}(p', depth - 1))$
return $a$
```

Minimax is a depth first search of the game tree which alternately selects the maximum and minimum scores attainable at each ply. The result is a principal continuation that minimizes the maximum loss (or regret) over the depth searched. The time complexity is $O(b^n)$ where n is the search depth and b is the maximum branching factor of the game tree, but the space complexity is $O(n)$ because it doesn't need to retain all the information about each branch. Figure 5 shows an example of a minimax search.

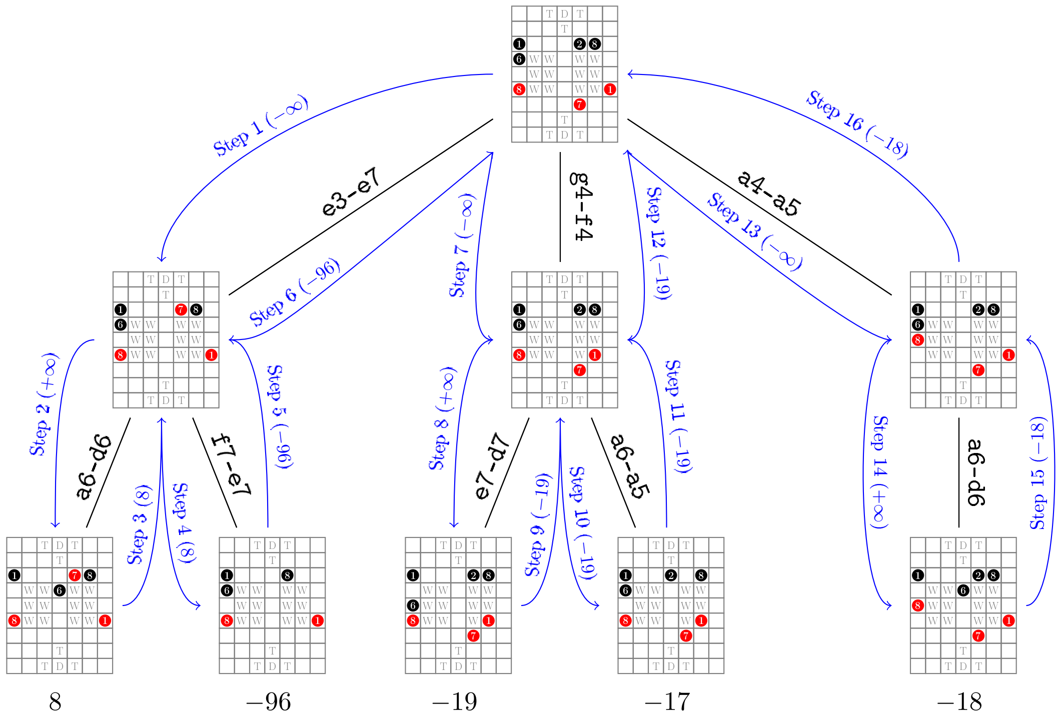


Figure 5: An example of a minimax search. Depth-first searching begins at the root and moves down along the left-most edges, representing individual moves from each position. Scores are backed up from the terminal nodes, alternately maximizing and minimizing at each level.

The minimax search shown in Figure 5 demonstrates how the algorithm performs a depth-first search, beginning at the root and moving down along the left-most edges. Terminal nodes are scored using the function from Section 3.3 and backed up the tree, alternately maximizing and minimizing the gains for player A at each level. At step 11 which moves upward from a terminal node, the minimal value of −19 was retained over −17, an event which did not occur at step 5 which replaced the previous value of 8 with the smaller value of −96. After the completion of the search, the move `a4-a5` will be selected because it is the continuation that leads to the maximum score, −18. A search for player B's best move would alternately minimize and maximize from the root instead of maximizing and minimizing.

If the search depth were increased beyond two plies, the continuation through `a4-a5` would likely return a much lower score because B's mouse at `a7` can advance downward, forcing A's elephant at `a5` to retreat. A search depth of two was not sufficient to foresee this situation and demonstrates the advantage provided by deeper searching of the game tree.

3.3 An Evaluation Function for Dou Shou Qi Positions

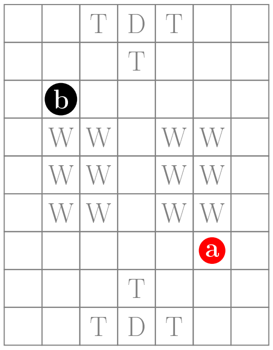


Figure 6: The OppositeSquare function which rotates the board. Piece b is at at $OppositeSquare[a]$.

At the heart of the minimax algorithm is the position evaluation function. Because Dou Shou Qi is zero-sum, we can represent the position in terms of benefit to player A. The same set of values can be used for both players by rotating and negating the development tables when we consider player B's pieces.

In practice, the use of both positional and development values is redundant because the development tables are specific for each piece. The material values could be added to all the entries in their respective development tables, and the material table done away with. We retain the material table for clarity only. The composite set of factors is $C[t, s]$ for a piece of type t at square s.

$$C[t, s] \;=\; Material[t] + Development[t, s]$$

Evaluation of a position sums the composite factors for all pieces belonging to player A, $\{a_1...a_l\}$ where l is the number of pieces that player A has on the board, and subtracts the sum of the composite factors for all pieces belonging to B, $\{b_1...b_m\}$ where m is the number of pieces that player B has on the board. Player B's summation uses the function $OppositeSquare$ to "rotate" the board so we can use one set of development values. $OppositeSquare$ returns the square on which the piece would be if it belonged to A instead of B (as seen in Figure 6.)

$$Evaluate[position] \;=\; \sum_{a_1}^{a_l} C[Type[a], Square[a]] - \sum_{b_1}^{b_m} C[Type[b], OppositeSquare[b]]$$

4 Split Searching

Our technique to discover, isolate and search subproblems takes advantage of Dou Shou Qi's naturally occurring subproblems which are loosely coupled and derive from the relatively slow piece movement of the game. Unlike chess, in which a bishop or a rook can traverse the length of the board in one move, Dou Shou Qi animals must move only one square at a time. Even the lion and tiger move slowly when they are not leaping over the water.

Using a function which establishes a lower-bound on time-to-effect between two given pieces, we will do subproblem discovery in a complete graph with a clustering algorithm. A minimum spanning tree can perform clustering and also provide a gauge to determine how far apart two clusters are. The largest edge in the minimum spanning tree can be used as a threshold to detect when natural subproblems are likely to be present. Clusters can then be isolated and searched independently.

4.1 Faster Searching

When a minimax search is conducted using limited time resources, split searching can see further ahead, an advantage which could lead to stronger game play. Because splitting the board into two separate minimax searches reduces the base of the exponent, it reduces the work required to search to a given level in the game tree.

The number of nodes in a game tree with branching factor, b, and depth, d, is expressed as follows:

$$\frac{b^{d+1} - 1}{b - 1}$$

Splitting a position produces two game trees to be searched to a new depth of d' with branching factors of b_1 and b_2. If the split places an equal number of pieces in each half, the game trees for both halves will be the same size and $b_1 = b_2 = b/2$. The total search space can be expressed as:

$$2\left[\frac{(b/2)^{d'+1} - 1}{(b/2) - 1}\right]$$

If both split and non-split searches are given the same amount of time, the size of the trees they can search will be the same. (We can drop the -1 in the numerator because b^d is assumed to be large enough that it doesn't matter.)

$$2\left[\frac{(b/2)^{d'+1}}{(b/2)-1}\right] = \frac{b^{d+1}}{b-1}$$

In the game tree b is approximately 32, the number of pieces (8) times the number of moves for each piece (4). When searching a game tree, deeper is always better, and the depth of a split search should increase over a non-split search by approximately 25%.

$$\begin{aligned}
2\left[\frac{(32/2)^{d'+1}}{(32/2)-1}\right] &= \frac{(32)^{d+1}}{32-1}\\
log_{16}\left(\tfrac{2}{15}(16)^{d'+1}\right) &= log_{16}\left(\tfrac{1}{31}(32)^{d+1}\right)\\
log_{16}\left(\tfrac{2}{15}\right)+d'+1 &= log_{16}\left(\tfrac{1}{31}\right)+d+1+log_{16}\left(2^{d+1}\right)\\
d' &= d+\frac{d+1}{4}+C
\end{aligned}$$

(C is a small .198 ply penalty for splitting, but it is a constant factor.)

Alternately, we can consider the time required to search to a given depth. Setting $d' = d$, we evaluate the ratio of split search time to global search time as a function of depth. (We begin with the expressions from step 2 of the previous calculation.)

$$\begin{aligned}
\frac{t'}{t} &= \frac{\frac{2}{15}(16)^{d+1}}{\frac{1}{31}(32)^{d+1}}\\
&= \frac{62}{15}\cdot\frac{(16)^{d+1}}{(16)^{d+1}(2)^{d+1}}\\
&= \frac{62}{15}\cdot\frac{1}{(2)^{d+1}}
\end{aligned}$$

4.2 Limitations

If no subproblems naturally exist, then the board is not split and searching remains global. If a split is forced on a position along lines which do not represent natural subproblems, the resulting split search could be suboptimal because the pieces across subproblems cannot interact. The splitting threshold is a key factor for the success of this technique.

5 Subproblems

In this section we define the space in which we conduct subproblem discovery. Our function is based on the physics concept of causality and captures time-to-effect between every pair of pieces in a position. To establish a causality function, we define a Dou Shou Qi effect cone and several operations for them.

Definition 5.1 *A Dou Shou Qi subproblem is one of two disjoint subsets, S_1 or S_2, which result when the pieces of a Dou Shou Qi board position are split such that S_1 will not affect S_2 for a minimum time, the splitting threshold, and vice versa.*

We have left the length of time during which S_1 and S_2 should remain separate an undefined function because it is a property of the game itself and will be discovered through gameplay. We could declare the splitting threshold to be only one move and create half a dozen subproblems, none of which are likely to correspond with a naturally occurring subproblem. Such a partition would be unlikely to lead to success because the pieces of the subproblems could not work together. Alternately we could define the splitting threshold to be the entire course of the game which will surely yield zero subproblems because, given enough time, every piece can affect every other piece (which is what makes the game interesting.)

5.1 Natural Subproblems

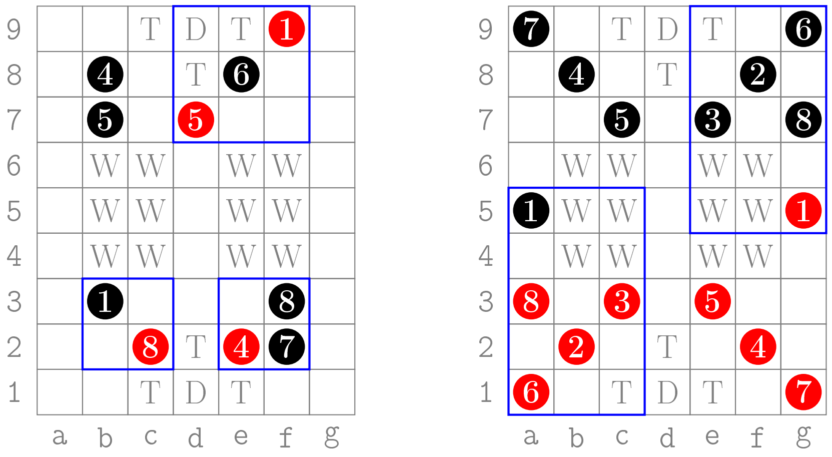
There are several factors which contribute to the natural development of subproblems. Pieces move one square at a time with the exception of the tiger and lion who are limited to a fixed distance. Interaction only occurs at close proximity so, as the pieces spread out, they tend to affect each other less. The two dens are the twin objectives toward which the pieces gravitate, and the nuclei for clusters which can become subproblems. In addition to the divided objectives, the water provides three separate avenues of approach (or aisles) down which the non-jumping and non-swimming pieces can move. This allows assaulting forces to circumvent each other rather than meeting in the middle, further contributing to the clustering effect of the dens.

5.2 Examples

Figure 7 shows examples of several subproblems a human might identify. The first example position contains three subproblems, two of which will result in the end of the game. One is in the upper-right corner of the board with player A's mouse (1) and hyena (5) against player B's tiger (6), next to player B's den. The second subproblem is in the lower-right corner and contains player B's elephant (8) and lion (7) against player A's dog (4), next to player A's den. The third subproblem is in the lower-left corner with player A's mouse (1) against player B's elephant (8), next to player A's den. The second example position is near the beginning of a game when there are subproblems to be resolved between the mice and the elephants. The mice have advanced to within two squares of the elephants.

6 Causality

By Definition 5.1 we are concerned about the time during which two subsets will remain separate rather than their physical distance. For that reason, we use the concept of causality as defined in physics, which states that, because nothing can travel faster than the speed of light, bodies that are separated by a distance cannot affect each other in less than the time it would take their light to travel between them. Effects of one body on another are



(a) A position with three subproblems {`d7 e8 f9`}, {`b3 c2`}, {`e2 f2 f3`}, and two extraneous pieces at `b7` and `b8`

(b) A position with two subproblems {`a1 a3 a5 b2 c3`}, {`e7 f8 g5 g7 g9`}, and extraneous pieces at `a9 b8 c7 e3 f2` and `g1`

Figure 7: Some example positions with subproblems. Blue squares outline the subproblems that a human might identify.

all the physical forces, electromagnetic, strong nuclear, weak nuclear and gravitational. The concept of causality can be visualized as a "light cone" in which space occupies a hyperplane along two axes and time a third. A light cone is projected to show the space-time in which a body can exert its effects, as seen in Figure 8 (a). A body at the origin can only affect bodies within its effect cone, which expands as it is projected upward.

6.1 Dou Shou Qi Effects

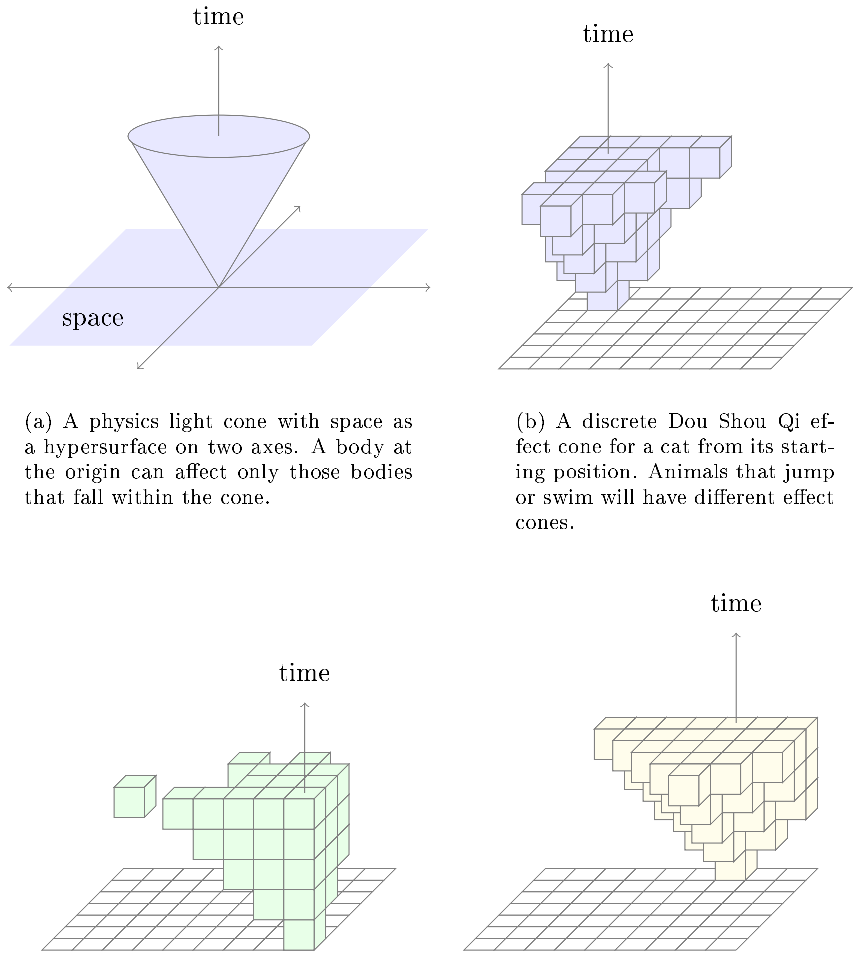
To adapt the principle of causality to the game of Dou Shou Qi, we define the effects that one piece may have on another. An effect is defined as follows:

Definition 6.1 *One piece, u, affects another, v, when it prevents v from making a move it would have been able to in the absence of u. Generally these include up, down, left, right and a non-move (or pass.)*

Definition 6.2 *u can affect v by:*

1. *taking v*
2. *blocking v by preventing a move without taking*
3. *forcing v to move or not to move*

Skipping a turn is not allowed in the rules of Dou Shou Qi, but we are thinking about individual pieces, some of which will not be selected to move. To allow some pieces to stay in the same place, we allow individual pieces the use of a non-move.



(a) A physics light cone with space as a hypersurface on two axes. A body at the origin can affect only those bodies that fall within the cone.

(b) A discrete Dou Shou Qi effect cone for a cat from its starting position. Animals that jump or swim will have different effect cones.

(c) A discrete Dou Shou Qi effect cone for a tiger from its starting position.

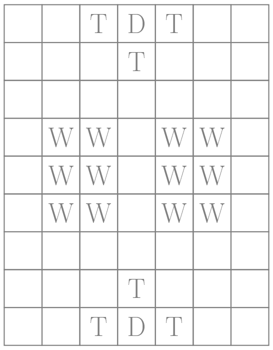
(d) A discrete Dou Shou Qi effect cone for a mouse from its starting position.

Figure 8: The concept of a light cone extended to Dou Shou Qi. A body at the origin can only affect bodies within its effect cone, which expands as it is projected upward.

During subproblem discovery, we are concerned with only effects 1 and 2 which involve two pieces interacting directly. Effect 3 is more subtle and occurs across subproblems, which we will handle by relaxing the rules of the game during searching and allowing a non-move (or a pass) from any position. This accounts for the fact that not every subproblem will receive a move during actual game play (in fact, only one can) and allows us to search each subproblem for a move and a non-move score.

6.2 Dou Shou Qi Space and Movement

The space in which we discover subproblems is Dou Shou Qi space and is the board on which the game is played. Dou Shou Qi space consists of 63 discrete squares as shown in Figure 9.



		T	D	T		
			T			
	W	W		W	W	
	W	W		W	W	
	W	W		W	W	
			T			
		T	D	T		

Figure 9: Dou Shou Qi Space which is the Dou Shou Qi gameboard.

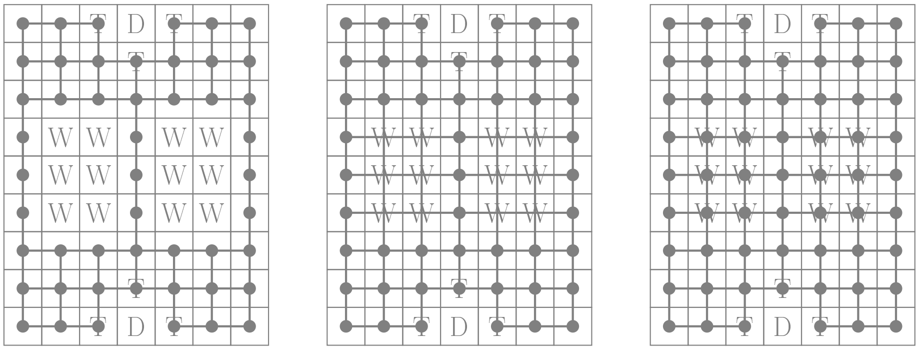
Movement in Dou Shou Qi space is dependent on piece type. The mouse is able to swim, the lion and tiger can jump, and the rest of the pieces move around the water. We represent the three movement types by three undirected graphs, G_s, G_j and G_n for swimming, jumping and normal moving respectively, in which vertices represent all reachable squares and edges connect vertices that are one move apart, shown in Figure 10. The length of the shortest path between pieces, $d[u, v]$, can vary as seen in Figure 11 in which piece u is a different distance from piece v depending on piece u's movement type.

6.3 Dou Shou Qi Effect Cones

Definition 6.3 *A Dou Shou Qi effect cone is a discrete, three-dimensional projection of every location a piece can occupy in Dou Shou Qi space-time.*

Effects 1 and 2 in Dou Shou Qi space require adjacency, so effect cones will follow piece movement. We begin our analysis of Dou Shou Qi causality with the free-body assumption, which allows a piece to move at every turn and behave as though it were the only piece on the board.

Definition 6.4 *A Dou Shou Qi free-body piece can move at every ply according to the movement graph for its piece type and behaves as though it were the only piece on the board.*

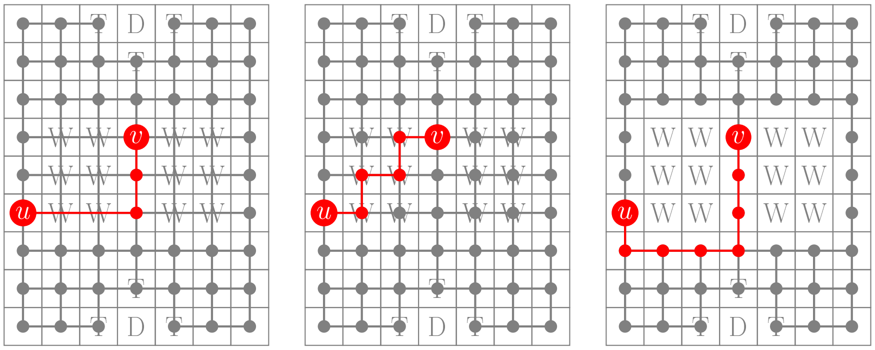


(a) G_n

(b) G_j

(c) G_s

Figure 10: Movement graphs in Dou Shou Qi space. G_n defines the movement for normal pieces which must move around the water. G_j defines the movement for jumping pieces which leap over the water. G_s defines the movement for swimming pieces which move through the water.



(a) $d\langle u, v\rangle = 3$ *for* G_j

(b) $d\langle u, v\rangle = 5$ *for* G_s

(c) $d\langle u, v\rangle = 7$ *for* G_n

Figure 11: Different types of pieces yield different distances. Piece u is a different distance from piece v depending on piece u's movement type.

A free-body effect cone is the projection of every square that free-body can occupy and is the starting point for our analysis of the effects one piece can have on another over time. We will define several functions to transform free-body effect cones into other effect cones that follow the rules of Dou Shou Qi.

An effect cone starts at the vertex representing the current location of the free-body. With each discrete time unit (one ply) all vertices adjacent to the cone are added to the cone. The projection of an effect cone can generate a counter-intuitive shape because of the underlying movement graph. For example, the cat effect cone in Figure 8 (b) wraps around the water and the den, which it cannot enter. The tiger effect cone in Figure 8 (c) jumps across the water and encompasses the square `b7` at $t = 4$, seemingly in the middle of the air. Nevertheless, free-body effect cones accurately capture the set of Dou Shou Qi squares which can be occupied over time.

Effect cones can be represented by a 7-by-9 matrix (representing Dou Shou Qi space) in which each entry is the height of the effect cone's surface over that square. Once a square has been added to the cone, it never leaves, so the point in space-time where that square is added is the only information that needs to be recorded.

6.4 Effect Cone Functions

Our objective is to build a function to capture minimum time-to-effect between pieces in a given Dou Shou Qi position. We will use several sub-functions that operate on effect cones.

6.4.1 Function $A(\ )$

A free-body effect cone assumes that a Dou Shou Qi body is free to move at every turn, but the rules of Dou Shou Qi allow a player to move only every other ply. We define a function, $A(U)$, which takes a free-body effect cone, U, for a piece belonging to player A and returns an effect cone that moves only at odd plies.

Definition 6.5

$$(A(U))_{ij} \;=\; max(0, 2 \cdot u_{ij} - 1)$$

Each matrix element is multiplied by two and reduced by one because all odd ply moves belong to Player A. We are projecting effect cones only into the future, so we force every element to be non-negative. $A(\ )$ of a cat's effect cone at square `f8` is as follows:

$$A\left(\begin{bmatrix} 6 & 5 & 4 & \infty & 2 & 1 & 2 \\ 5 & 4 & 3 & 2 & 1 & 0 & 1 \\ 6 & 5 & 4 & 3 & 2 & 1 & 2 \\ 7 & \infty & \infty & 4 & \infty & \infty & 3 \\ 8 & \infty & \infty & 5 & \infty & \infty & 4 \\ 9 & \infty & \infty & 6 & \infty & \infty & 5 \\ 10 & 9 & 8 & 7 & 8 & 7 & 6 \\ 11 & 10 & 9 & 8 & 9 & 8 & 7 \\ 12 & 11 & 10 & 9 & 10 & 9 & 8 \end{bmatrix}\right) = \begin{bmatrix} 11 & 9 & 7 & \infty & 3 & 1 & 3 \\ 9 & 7 & 5 & 3 & 1 & 0 & 1 \\ 11 & 9 & 7 & 5 & 3 & 1 & 3 \\ 13 & \infty & \infty & 7 & \infty & \infty & 5 \\ 15 & \infty & \infty & 9 & \infty & \infty & 7 \\ 17 & \infty & \infty & 11 & \infty & \infty & 9 \\ 19 & 17 & 15 & 13 & 15 & 13 & 11 \\ 21 & 19 & 17 & 15 & 17 & 15 & 13 \\ 23 & 21 & 19 & 17 & 19 & 17 & 15 \end{bmatrix}$$

6.4.2 Function $B(\ )$

We define $B(\ )$ which translates a free-body effect cone for a player B piece into an effect cone which also follows the rules of Dou Shou Qi.

Definition 6.6

$$(B(U))_{ij} \;=\; 2 \cdot u_{ij}$$

Each element is multiplied by two because player B can move only at even plies. $B(\ )$ of a tiger's effect cone from square `a1` is as follows:

$$B\left(\begin{bmatrix} 7 & 6 & 7 & 8 & 9 & 10 & 10 \\ 6 & 5 & 6 & 7 & 8 & 9 & 9 \\ 5 & 4 & 5 & 6 & 7 & 8 & 8 \\ 5 & \infty & \infty & 6 & \infty & \infty & 7 \\ 4 & \infty & \infty & 5 & \infty & \infty & 6 \\ 3 & \infty & \infty & 4 & \infty & \infty & 5 \\ 2 & 3 & 4 & 5 & 6 & 7 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & \infty & 6 & 7 & 8 \end{bmatrix}\right) = \begin{bmatrix} 14 & 12 & 14 & 16 & 18 & 20 & 20 \\ 12 & 10 & 12 & 14 & 16 & 18 & 18 \\ 10 & 8 & 10 & 12 & 14 & 16 & 16 \\ 10 & \infty & \infty & 12 & \infty & \infty & 14 \\ 8 & \infty & \infty & 10 & \infty & \infty & 12 \\ 6 & \infty & \infty & 8 & \infty & \infty & 10 \\ 4 & 6 & 8 & 10 & 12 & 14 & 12 \\ 2 & 4 & 6 & 8 & 10 & 12 & 14 \\ 0 & 2 & 4 & \infty & 12 & 14 & 16 \end{bmatrix}$$

6.4.3 Maxima of Effect Cones

Given two pieces, u and v, which belong to players A and B respectively, we can find the minimum time required for them both to reach a given square. First, we project their free-body effect cones, U and V, and calculate $A(U)$ and $B(V)$. By comparing the same elements in their respective effect cones, we can discover who will arrive last, which must be the minimum time required for both pieces to occupy the same square. The function $Max(U, V)$ applies $max()$ between each element of two effect cones.

Definition 6.7

$$(Max(U, V))_{ij} \; = \; max(u_{ij}, v_{ij})$$

The minimum time required for a player A's cat at `f8` and player B's tiger at `a1` to reach any given square is as follows:

$$Max(A(U), B(V)) \; = \; \begin{bmatrix} 14 & 12 & 14 & \infty & 18 & 20 & 20 \\ 12 & 10 & 12 & 14 & 16 & 18 & 18 \\ 11 & 9 & 10 & 12 & 14 & 16 & 16 \\ 13 & \infty & \infty & 12 & \infty & \infty & 14 \\ 15 & \infty & \infty & 10 & \infty & \infty & 12 \\ 17 & \infty & \infty & 11 & \infty & \infty & 10 \\ 19 & 17 & 15 & 13 & 15 & 14 & 12 \\ 21 & 19 & 17 & 15 & 17 & 15 & 14 \\ 23 & 21 & 19 & \infty & 19 & 17 & 16 \end{bmatrix}$$

Notice that the element representing square `b7` is the smallest element in the matrix. This is the first square that both pieces could occupy.

6.4.4 Minimum Height of an Effect Cone

The function $MinHeight(\ )$ reports the minimum element of the matrix representing an effect cone. $MinHeight$ of the previous result would return a value of 9.

7 A Causality Function

Given two free-body effect cones, we can apply a series of functions to determine the minimum time before two pieces can reach the same square. Because effects occur with adjacency, we are looking for what happens just before that point, so we subtract one ply from the result. When two pieces are on the same side, it takes twice as long for them to reach a given square because they must wait for the other player to move.

We define a piecewise causality function that takes two pieces and determines their minimum time-to-effect.

Definition 7.1

$$T[u,v] = \begin{cases} MinHeight(\,Max(\,A(U),\,B(V)\,)\,) - 1 & if\ u \in A\ and\ v \in B \\ MinHeight(\,Max(\,B(U),\,A(V)\,)\,) - 1 & if\ u \in B\ and\ v \in A \\ 2 \cdot MinHeight(\,Max(\,A(U),\,A(V)\,)\,) - 1 & if\ u \in A\ and\ v \in A \\ 2 \cdot MinHeight(\,Max(\,B(U),\,B(V)\,)\,) - 1 & if\ u \in B\ and\ v \in B \end{cases}$$

7.1 A Complete Casuality Graph

With our causality function for Dou Shou Qi space, $T[u,v]$, we can analyze a complete Dou Shou Qi position and have some sense of the distance between each pair of pieces in terms of minimum time-to-effect. The next part of this thesis will be the proposal of an algorithm to discover loosely coupled subproblems in Dou Shou Qi causality space. The problem is best phrased as a graph problem so we translate a position and its occupying pieces into a complete graph.

Definition 7.2 *Let G_T be a complete, undirected graph with vertices V for each piece in a Dou Shou Qi position and edges E with weight $T[u,v]$ for each $u,v \in V, u \neq v$.*

8 Finding Subproblems

We defined a Dou Shou Qi subproblem in terms of disjoint subsets pieces which would remain separate for a minimum time, the splitting threshold. We adapt that definition to be a graph problem whose solution we will either accept or reject based on whether is crosses the threshold. Given the undirected, complete graph G_T with vertices V and edges E, we want to find a partition of V into disjoint subsets S_1 and S_2 such that the minimum edge between S_1 and S_2 is maximized. The minimum edge between S_1 and S_2 is the greatest length of time we are guaranteed that the two subsets will not affect each other and is the variable we are trying to maximize, called T_{max}. The magnitude of T_{max} must cross the splitting threshold at which point we expect to find subproblems.

8.1 Using Minimum Spanning Trees

Our clustering problem can be solved by computing a minimum spanning tree of G_T and retrieving S_1 and S_2 by traversing the two subtrees attached to the largest edge. The use of minimum spanning trees to identify clustering structure in Euclidean space has been previously proposed by Florina *et al*[10]. Our algorithm will partition the graph into only two subsets, but additional partitioning could be done recursively to find more subproblems.

8.1.1 Using Kruskal's minimum spanning tree Algorithm

To prove that finding an minimum spanning tree tree of G_T provides the correct solution to our partitioning problem, we will define a slightly modified version of Kruskal's minimum spanning tree algorithm which runs until there are two sets remaining, S_1 and S_2.

Kruskal's minimum spanning tree algorithm is a greedy algorithm that makes every vertex into a set and then works its way through the edges in order from smallest to largest. Each time it encounters an edge that connects two vertices not in the same set, it unions the sets and adds the edge to the minimum spanning tree. This continues until only one set remains and the minimum spanning tree is completely known [15, page 569].

```
MST-PARTITION($G, w$)
for each vertex $v \in V[G]$
    do MAKE-SET($v$)
$setcount \leftarrow |V|$
$tmax \leftarrow 0$
sort the edges of $E$ into nondecreasing order by weight $w$
for each edge $(u, v) \in E$, taken in nondecreasing order by weight
    do if FIND-SET($u$) $\neq$ FIND-SET($v$)
        then $tmax \leftarrow w$
            if $setcount > 2$
              then UNION($u, v$)
                  $setcount \leftarrow setcount - 1$
              else return $S_1$, $S_2$, $tmax$
```

8.1.2 Proof of MST-Partition

The proof of the correctness of MST-Partition is done by contradiction. T_{best} is defined as the maximum attainable value of T_{max} for a given position.

Proof 8.1 *Suppose there exists an ideal partitioning of V into two non-empty, disjoint sets, S_1 and S_2, that yields T_{best}. It is true that every set is a subset of either S_1 or S_2 which is the invariant of the algorithm. Initially the invariant is trivially true because every set consists of one vertex and every vertex must be in either S_1 or S_2. Between every pair of sets there exists a limiting edge such that every other edge between the two sets is greater-than or equal. As long as there are more than two sets remaining, choose the smallest limiting edge, e_{min}, and union the two sets it connects, U and V. $U \cup V$ must be a subset of either S_1 or S_2. If it was part of both, the weight of e_{min} would be T_{best} since every other limiting edge is greater-than-or-equal to e_{min}. However, we could have attained a higher value of T_{best} by unioning U and V as S'_1 and unioning the remaining sets as S'_2. The limiting edge between S'_1 and S'_2 would then be greater-than-or-equal to e_{min}, so we could have had a larger value of T_{best}, which is impossible by definition. (At the very least, we could have had an equivalent partitioning.) So by contradiction, $U \cup V$ must be either a subset of either S_1 or S_2, not both.*

8.1.3 Any Minimum Spanning Tree Algorithm Will Do

Our algorithm optimally partitions the graph by finding an minimum spanning tree, which we could have computed using any minimum spanning tree algorithm. The configuration of a Dou Shou Qi board changes very little from ply to ply so we don't need to recompute an minimum spanning tree each time we move a piece. Section 8.2 proposes an algorithm to update an minimum spanning tree in $O(n)$ time which is a significant time savings over the $O(n^2)$ time required to compute.

8.2 Maintaining an Minimum Spanning Tree in O(n) Time

Updating a minimum spanning tree is a difficult problem that is sensitive to the density of the graph, and in our complete causality graph, $|E| = \Theta(n^2)$. Previously researched algorithms allow for adding edges, removing edges and making arbitrary edge weight changes in $O(|E|)$ per update using a simple DRD[2] tree[14] or $O(\sqrt{|E|})$ per update using more complex dynamic data structures[5]. Our algorithm operates under a special set of constraints that allow it to make $n-1$ off-line updates in $O(n)$ time, an improvement over the more general case.

8.2.1 Special Constraints

Our problem places five special constraints on the location, number and magnitude of edge update operations.

1. We are performing exactly $n-1$ off-line[3] updates.
2. All updates are neighborhood[4] updates.
3. Weights change by exactly ± 1 or 0.
4. The maximum degree of an minimum spanning tree vertex is upper bounded by a constant factor.
5. Equal edge weights are upper bounded by a linear factor.

1. $n-1$ offline updates We have defined a complete graph and every vertex is attached to exactly $n-1$ other vertices. When a piece is moved, the distance from that piece to every other piece changes and requires the respective edges to be updated. The rest of the edges remain unchanged.

2. Neighborhood updates Because we are moving one piece, only the edges attached to the vertex representing that piece will change. Since they all share a common vertex, we can perform our updates on just the neighborhood of that vertex.

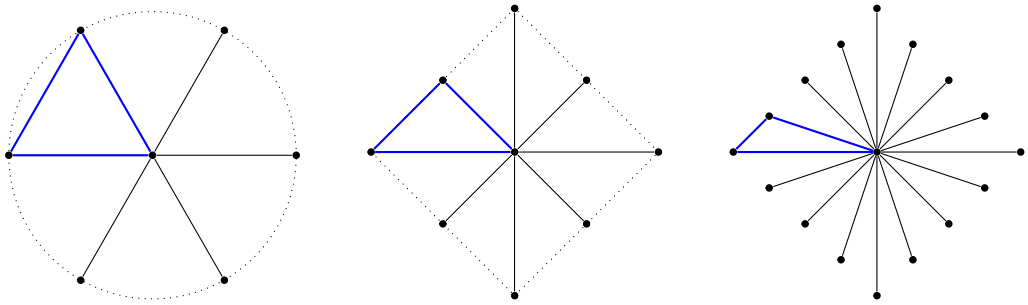
3. Weights change by exactly 1 or 0 Our causality function calculates the distance in time between two pieces in Dou Shou Qi space. When moving a piece, one unit of time expires and therefore the distance could not have changed by more than 1. All edge weights are integer values, therefore all edge updates must be plus/minus 1 or 0. Some pieces can jump the water and cover four squares, but that is taken into account in the movement graph such that a single jumping move is given a weight of 1.

[2] doubly-linked reversed dynamic tree

[3] the minimum spanning tree is returned only after all updates have been completed

[4] edges are attached to a common vertex

4. Maximum degree upper bounded To show that the maximum degree of an minimum spanning tree vertex in Dou Shou Qi causality space is upper bounded we will begin with Euclidean metric-space in which the maximum degree of an minimum spanning tree vertex is 6.



(a) A Euclidean minimum spanning tree vertex of degree 6. The blue triangle is equidistant and equiangular.

(b) A Manhattan minimum spanning tree vertex of degree 8. The blue triangle is equidistant and the dotted line represents all points equidistant from the center vertex (a circle.)

(c) A causality minimum spanning tree vertex of degree 16. The blue triangle could be equidistant. Note: causality space cannot be drawn to scale on a plane.

Figure 12: Vertices of maximal degree. The maximum degree of an minimum spanning tree vertex depends on the metric which is used to calculate its edge weights.

Proof 8.2 *Given a complete graph G in Euclidean plane with vertices V and edges E, consider any three vertices $\in V$ and the three edges that connect them. The minimum spanning tree of G cannot contain the largest of the three edges. If it did, one of the other two smaller edges could have been used to obtain a lighter minimum spanning tree which is impossible by definition. If the two smaller edges are part of the minimum spanning tree then the angle between them cannot be less than 60 degrees or they would not be the smaller edges. If a vertex in the minimum spanning tree is of maximal degree then it cannot have more than 6 adjacent edges or some of the angles between them will be less than 60 degrees. Therefore the maximum degree of an minimum spanning tree in Euclidean space is 6.*

To extend this proof to our causality function, we need another metric as a stepping stone. In accordance with the rules of the game, Dou Shou Qi pieces can move only vertically and horizontally, making the Manhattan metric, in which movement is limited to one axis at a time, the most natural metric to describe the distance between two pieces. We define the Manhattan metric $M[u, v]$ where u and v are two pieces with coordinates (x_1, y_1) and (x_2, y_2) respectively as follows:

$$M[(x_1, y_1), (x_2, y_2)] = |x_2 - x_1| + |y_2 - y_1|$$

The limiting factor of the maximum minimum spanning tree vertex degree in Proof 7.3 is the angle at which the opposite side can no longer be the largest edge. Extending the proof to a Manhattan metric, we find that the maximum degree for a vertex in a Manhattan minimum spanning tree is 8. In a Manhattan metric-space an equidistant triangle is not necessarily equiangular[8] and the angle across from the largest side of a triangle can be as small as 45 degrees. See Figure 12 (b).

Lemma 8.3 *The angle opposite the largest side of a triangle in Manhattan metric-space cannot be smaller than 45 degrees. Therefore by Proof 8.2 the maximum degree of a vertex in a Manhattan minimum spanning tree is 8.*

Our causality function is based on the minimum time that it would take two pieces to reach the same square. When the pieces are on opposite sides, they can close the distance between them at least as fast as one square per ply. Jumping and swimming pieces are never further from each other than the Manhattan distance because the mouse uses Manhattan movement, and the lion and tiger can jump the water to travel faster than the mouse. Only normal pieces have to travel around the water which is a diversion of at most four plies. Therefore, when pieces are on opposite sides, their causality distance is at most the Manhattan distance plus four. When pieces are on the same side, they can close the distance between them at a rate no less than one square every two plies, so their causality distance could be double.

$$T[u,v] \leq 2 \cdot (M[u,v] + 4)$$

The fastest movement on the board is jumping horizontally over both bodies of water to get to a square across the board, which shortens the distance by four squares. Therefore the shortest causality distance is no less than the Manhattan distance minus four.

$$T[u,v] \geq M[u,v] - 4$$

Within a constant, a causality distance will be a least the Manhattan distance and no more than twice the Manhattan distance. In the worst case, the two tree edges adjacent to v could be minimum (the Manhattan distance) and the farthest side of the triangle could be maximum (twice the Manhattan distance), thus creating an equidistant triangle. The adjacent edges would then create an angle of 22.5 degrees, half the 45 degrees of the Manhattan metric, establishing an upper-bound on the degree of a causality minimum spanning tree of 16. See Figure 12 (c).

Lemma 8.4 *The angle opposite the largest side of a triangle in Dou Shou Qi causality space cannot be smaller than 22.5 degrees. Therefore by Proof 8.2 the maximum degree of a vertex in a Dou Shou Qi causality minimum spanning tree is 16.*

A maximum degree of 16 may sound like a lot but we are only concerned with how that number relates to the number of vertices in the graph. By Lemma 8.4 the degree of the vertices of an minimum spanning tree in Dou Shou Qi causality space is upper bounded by $O(1)$.

5. Equal edge weights upper bounded In a complete graph in Euclidean space the number of equal edge weights is upper bounded by $O(n)$. We prove this by induction.

Proof 8.5 *Consider a complete graph of points in Euclidean space with three vertices whose three edges have a equal weight w_i. This case is trivially true because there are n vertices and $O(n)$ edges of weight w_i. Now consider a complete graph with n vertices and $O(n)$ edges of a equal weight, w_i. When adding one vertex v to the graph, it can form a cluster of vertices of at most 7 that are equidistant by edges of weight w_i. This follows from Proof 8.2. v adds at most $O(1)$ edges to the set of w_i edges but adds $O(n)$ edges to the complete graph, so a complete graph of $n+1$ vertices will have $O(n+1) = O(n)$ edges of weight w_i. Repeat this argument for all edge weights in the limited or unlimited universe of possible edge weights $w_0 ... w_j$.*

Extending Proof 8.5 to the Manhattan metric and causality function is a simple matter of adjusting the largest possible cluster of equidistant vertices.

Lemma 8.6 *The maximum number of equidistant points in a cluster in Manhattan metric-space is 9, which follows from Lemma 8.3. Therefore according to Proof 8.5 the number of equal edge weights in a complete Manhattan graph is upper bounded by $O(n)$.*

Lemma 8.7 *The maximum number of equidistant point in a cluster in Dou Shou Qi causality space is 17, which follows from Lemma 8.4. Therefore according to Proof 8.5 the number of equal edge weights in a complete Dou Shou Qi causality graph is upper bounded by $O(n)$.*

8.2.2 Minimum Spanning Tree Update Operations

To show that our minimum spanning tree update algorithm runs in $O(n)$ time we must show that its operations run in $O(n)$ time. These operations include changing edge weights, adding vertices and removing vertices. We will address edge weight changes in four cases, increasing and decreasing both tree and non-tree edges.

To facilitate the management of our off-line updates, we create a reduced set of edges E' which initially contains all $n-1$ tree edges. We add $O(n)$ edges to E' and then recompute our minimum spanning tree from the reduced edge set, a technique used by David Eppstein in his off-line minimum spanning tree updating algorithm[4]. The best minimum spanning tree algorithms can run in $O(|E|)$[14] time so there is no penalty for our final recomputation since $|E'| = O(n)$.

We also root the minimum spanning tree at vertex v which represents the piece being moved, a technique used on the DRD trees of Ribeiro and Toso[14]. All of the updates are then between the root and its children which allows us to consider the subtrees of which there are $O(1)$ by Lemma 8.4. Rooting the minimum spanning tree at v takes an $O(n)$ traversal and therefore causes no penalty.

When representing a graph with an adjacency list, we can hash the edges of each vertex by weight, allowing us to access edges of a given weight in constant time.

Increase Non-Tree Edge Increasing a non-tree edge e is a trivial case. e runs from v to one of the subtrees which is already attached to v by a smaller edge. Increasing a non-tree edge guarantees that it will remain a non-tree edge.

Decrease Tree Edge Decreasing a tree edge is also a trivial case. It was the smallest edge connecting the subtree to the rest of the minimum spanning tree. Decreasing the edge guarantees that it will remain the smallest edge to connect the subtree and therefore it remains part of the minimum spanning tree.

Decrease Non-Tree Edge Decreasing a non-tree edge could displace another tree edge somewhere in the subtree that it connects to. The correct way to fix the minimum spanning tree is to add e, creating a cycle, and then remove the largest edge in the cycle[14, page 4],[4, page 3]. We will do this indirectly by simply adding e to E' and allowing the recomputed minimum spanning tree to leave out the largest edge of the cycle we create.

Increase Tree Edge Increasing a tree edge e is the most difficult case because, if we need to remove e from the minimum spanning tree, there are $\Theta(n^2)$ candidate edges to consider when replacing it. In the worst case all of the tree-edge updates will be increases and we will have to remove all but one of them from the minimum spanning tree. In this case we must search all of the $\Theta(n^2)$ edges of the subtrees to reconnect the graph. This is made easier by Lemmas 8.4 and 8.7 which place upper bounds on the number of subtrees and equal edge weights respectively.

Any edge that will reconnect a disconnected subtree will be one less than the new weight of the increasing tree edge.

Proof 8.8 *Consider a subtree whose connecting edge e with weight w must be increased by 1. If it is necessary to remove e, there will be a minimum edge e' from the subtree to the rest of the graph which will take its place. The weight of e' cannot be less than w or it would have been part of the minimum spanning tree instead of e. The weight of e' also cannot be greater than w or e could remain part of the minimum spanning tree after being increased. Therefore any edge e' that replaces e must be the same weight as e.*

In accordance with Proof 8.8 we only have to look at edges of weight w in each subtree where w is the weight of the edge adjacent to v. There are $O(1)$ subtrees that can be disconnected so we must traverse each subtree and add the $O(n)$ edges of weight w to E'. (Remember that there are at most $O(n)$ edges of a given weight in the graph.)

Adding a Piece Adding a piece is against the rules of the game but most implementations would want to have this feature so the user can undo a move. This operation is a trivial case. We simply add the new vertex and add its edges to the reduced edge set of E'. There will be $O(n)$ new edges and so recomputation of the minimum spanning tree will run in $O(n)$ time.

Removing a Piece Removing a piece after a capture is a more difficult case. Consider each subtree of the removed vertex v, being captured by vertex u. If the root of the subtree, s, belonged to the same player as u, then we are guaranteed that it belongs to a different player than v (because capturing always happens by the other player's pieces.) In that case, v is closer to s than u was to s, so the edge from s to v can safely be added to the minimum spanning tree. If s belonged to a different player than u, then we are guaranteed that v is further from s than u (probably twice as far.) In this case, we must search the subtree under s to find a minimum edge to reconnect that subtree.

Searching a subtree of $O(n)$ vertices for a minimum edge to another subtree can take $O(n^2)$ time in the worst case. We can mitigate the worst case by keeping track of the size of the subtrees and traversing the smallest disconnected subtree first to increase the odds of finding an edge outward, but it is still $O(n^2)$. We cannot currently prove that removing a vertex can be done in $O(n)$ time, so Update-MST can only be used for piece movement, not capture.

8.2.3 Update-MST

Given a complete graph G, a minimum weight spanning tree T and a vertex v which has been moved, the following algorithm will update the edges adjacent to v and update T to be the new minimum weight spanning tree.

```
Update-MST($G, T, v$)
 1  $E' \leftarrow Edges[T]$
 2  $A \leftarrow \emptyset$
 3  root $T$ at vertex $v$
 4  for each subtree $s$ of $T$
 5       do $Weight[s] \leftarrow$ weight of edge adjacent to $v$
 6  for each edge $e$ adjacent to $v$
 7       do
 8          if $e \notin T$ and is decreasing
 9               do $E' \leftarrow E' \cup e$
                    ▷ Add increasing non-tree edges to reduced edge set
10          if $e \in T$ and is increasing
11               do $A \leftarrow A \cup Subtree[e]$
                    ▷ Remember subtrees to reconnect
12          Update-Edge($e$)
13  for each subtree $s \in A$
14       do for each vertex $u$ in $s$
15               do add edges adjacent to $u$ of $Weight[s]$ to $E'$
16  $T' \leftarrow$ Compute-MST($V, E'$)
17  return $T'$
```

8.2.4 Optimization of MST-Update

The algorithm given in Subsection 8.2.3 establishes an $O(n)$ running time but has at least one inefficiency that can be easily corrected. The asymptotic running time will not change but we can significantly reduce the number of edges added to E' at Line 15.

Line 15 adds all of the edges of weight w to E', but we only require enough edges to guarantee the connectivity of each subtree. When rooting the minimum spanning tree at v

we perform a traversal during which we can label each vertex according to its subtree. Any edge added to E' must satisfy both subtrees it connects, s_1 and s_2. Proof 8.8 establishes that an edge reconnecting a subtree must be of weight w, therefore $Weight[s_1] = Weight[s_2]$ and we need only reconnect subtrees of equal $Weight$ values.

We can assume that subtrees with different $Weight$ values are connected via v and focus exclusively on subtrees of the same $Weight$ value. Given a set of k subtrees to reconnect, $\{s_1, ..., s_k\}$, we can make each subtree its own set, union sets that are connected and stop when there is one set remaining (just as in Kruskal's minimum spanning tree algorithm.) Therefore the number of edges we need to reconnect $\{s_1, ..., s_k\}$ is $k-1$. Because k is upper-bounded by $O(1)$ by Lemma 8.4, we can add a constant number of edges to E' rather than $O(n)$ to replace increasing tree edges.

9 Searching Beyond T_{max}

In this section we propose a way to isolate subproblems discovered by our previous algorithm. Forcing a subset of pieces to remain separate from another subset of pieces is effectively trimming branches of the game tree which involve interaction between subsets, so subproblem isolation is heuristic in nature. Previously we established T_{max} as the maximum number of plies during which our subproblems remain strictly unaffected by each other, placing a relatively shallow upper-bound on the maximum depth of a split minimax search. We want a means of isolating subproblems so we can extend the split search.

Our graph partitioning algorithm uses a minimum spanning tree to find subproblems and, based on Definition 5.1, requires that the pieces of one partition be unable to affect the pieces of another. After the time represented by T_{max}, this guarantee can no longer be upheld because if the pieces of one subproblem were allowed to stray close enough to the pieces of another, our minimax search may count on using the same square twice.

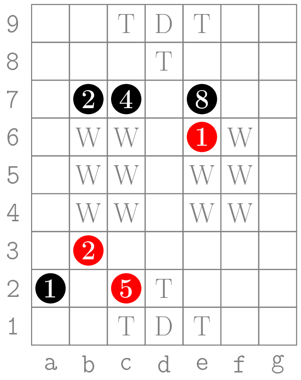


Figure 13: Two subproblems that should not affect each other.

The position shown in Figure 13 has two distinct subproblems, one in the lower-left corner with B's mouse (1) and A's cat (2) and hyena (5), and another in the upper half with B's cat (2), dog (4) and elephant (8) and A's mouse (1). Given a T_{max} threshold of 3 or 4, the pieces will be partitioned into subproblems {a2,b3,c2} and {b7,c7,e7,e6}, each of which will be unaware of the other. If the search is not stopped at a particular depth or the movements of the pieces restricted, player A's cat at b3 might be tempted to turn away from the mouse, who is no real threat with the hyena at c2, and run toward to enemy's den for an easy win. In fact, the situation isn't that simple because there are a lot of pieces to contend with on player B's side. (The best strategy for player A is to kill the mouse at a2 using the cat and the hyena, thereby freeing the hyena to aggress.)

The imposition of a sustained split of the board is a reasonable extension of our algorithm. Assuming the correctness of our subproblem detection algorithm, each subset of pieces should accurately encapsulate a subproblem and the pieces therein should not be running willy-nilly toward other partitions. A subproblem is a tightly knit set of pieces that must maintain their relative proximity to support a resolution of the problem. If two pieces are moving rapidly toward one another, it is unlikely they are in separate, naturally occurring subproblems.

To maintain the integrity of the subsets, we can either contain them dynamically or statically. A dynamic approach would involve repartitioning G_T at intervals during the minimax search. A static solution could forcibly maintain the partition as they existed when the search began, only updating the partitioning when a piece is moved during game-play.

9.1 Dynamic Repartitioning

Dynamic partitioning would involve keeping track of where the pieces are "on-the-fly". If a pair of pieces, u and v, in separate subproblems come too close to one another, we might want to remove the partition that separates them, combining u and v into one subproblem at a particular depth of the tree. Repartitioning at depth requires the program to know the state of the board at that depth, which is not feasible. Minimax is exploring each subproblem for a continuation that leads to the minimum maximum loss and a split search allows both partitions to unfold simultaneously. The appearance of u and v in close proximity at a particular depth is deceiving because their respective subproblems cannot develop simultaneously in actual game play and there is no guarantee an equal amount of time will be spent on each subproblem.

Additionally, the principal continuation is not known for each partition until the minimax algorithm has completed. Therefore only positions along the principal continuation are worth considering in a repartitioning of the board because the rest of the intermediate positions were discarded by the algorithm. Repartitioning the board according to a position in the principal continuation may lead to a different set of terminal positions and therefore a different principal continuation, an apparent paradox. Such circular reasoning makes the problem of dynamic repartitioning beyond the scope of this thesis.

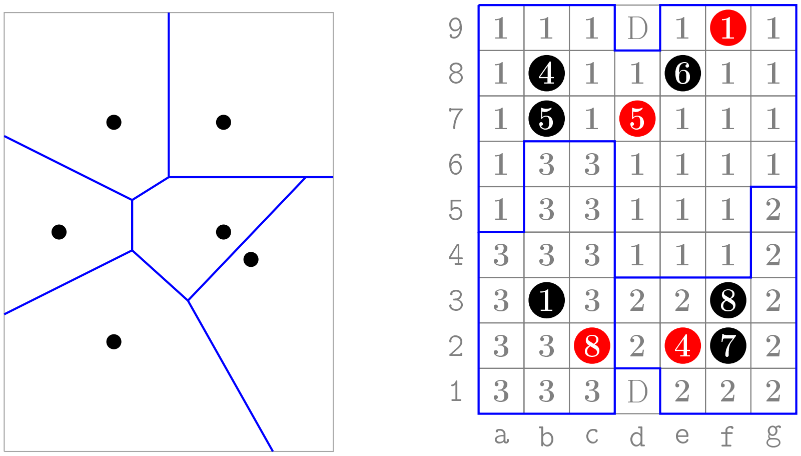
9.2 Static Partitioning

A simpler approach, and the one taken here, is to forcibly maintain the original partitioning of G_T throughout the minimax search. This places certain areas of the board "off limits" to subproblems and thereby excludes large portions of the game tree from the minimax search (which are those branches whose roots are off-limit moves.) The risk is that the globally best continuation lies within one of those discarded branches, making the result of the minimax search suboptimal.

Static partitioning is much easier to implement than dynamic repartitioning because we don't have to worry about the subproblems on the fly or the correctness of a dynamic algorithm. We just have to guarantee that each partition doesn't interfere with the others and vice versa.

9.3 Voronoi Diagrams

In order to keep the partitions separate we will fence them in with a variation of a Voronoi diagram[5] in Dou Shou Qi causality space. Instead of generating points we will use subproblems, partitioning the squares of Dou Shou Qi space such that each square in a given polygon is closer to its generating subproblem than any other (see Figure 14.)



9	1	1	1	D	1	①	1
8	1	❹	1	1	❻	1	1
7	1	❺	1	⑤	1	1	1
6	1	3	3	1	1	1	1
5	1	3	3	1	1	1	2
4	3	3	3	1	1	1	2
3	3	❶	3	2	2	❽	2
2	3	3	⑧	2	④	❼	2
1	3	3	3	D	2	2	2
	a	b	c	d	e	f	g

(a) A Voronoi Diagram. Every point in a given polygon is closer to its generating point than any other generating point.

(b) A partitioning of the example given in Figure 7 (a). Every square in a given polygon is closer (or as close) to a piece in its subproblem than any piece in any other subproblem. Note: `b7` and `b8` are part of subproblem 1 because T_{thresh} is set at two plys.

Figure 14: Voronoi diagrams extended to board partitioning.

[5] "The partitioning of a plane with n points into convex polygons such that each polygon contains exactly one generating point and every point in a given polygon is closer to its generating point than to any other." [9]

The dens are a special case because their occupation signals the end of the game, therefore we leave them accessible to any subproblem that contains an adjacent trap. Allowing only one subproblem to access a den would preclude the other subproblems from finding a winning (or losing) continuation.

9.4 Partition-Board

To partition a Dou Shou Qi board like a Voronoi diagram, we would like know which subproblem is closest to any given square. For that purpose we define a new operation for effect cones.

9.4.1 Minima of Effect Cones

We would like to project an effect cone from all of the pieces of a subproblem simultaneously and record only the smallest values for each square. To accomplish this, we project effect cones from each piece independently and then use the function $Min(U, V)$ to take the minima of them all.

Definition 9.1

$$(Min[U, V])_{ij} \;=\; min(u_{ij}, v_{ij})$$

9.4.2 Cumulative-Effectcone

```
CUMULATIVE-EFFECTCONE($S$)
$L \leftarrow \emptyset$
▷ list of effect cones
$C \leftarrow \infty$
▷ infinite effect cone
for each piece $p \in S$
      do if $p \in A$
            $L \leftarrow L \cup A$[PROJECT-EFFECTCONE($p$)]
    else
            $L \leftarrow L \cup B$[PROJECT-EFFECTCONE($p$)]
FOLD(MIN(),C,L)
▷ take the min of all cones
```

Cumulative-Effectcone returns a matrix which identifies the minimum time needed for a piece from that subproblem to arrive at a given square. By projecting cumulative effect cones from each subproblem, we can determine which subproblem is closest to each square by comparing the same element in each effect cone.

We can isolate pieces for a while to search deeper than T_{max}, but subproblems are not strictly decoupled and they will eventually interact. The benefits of split searching must be balanced against the detriment of searching a split position too deeply. This must be taken into account when setting the splitting threshold, which must take into account the available resources.

Our partitioning algorithm adds a term, linear in the size of the board, to the overall time complexity of a split search. We can run a partitioned search of a position to any depth in $O(n+b+2^d)$ time, where n is the number of pieces in the position, b is the size of the board and d is the search depth. The $O(2^d)$ term will dominate the other two, so taking the time to partition the board is not detrimental to the time complexity of the overall algorithm.

10 Putting it all together

Our program's entry point is the Search-Position function which inputs a position and returns a suggested move. It uses several modules to 1) generate a complete graph, 2) compute the minimum spanning tree, 3) divide the given position into subproblems, 4) search each subproblem and evaluate the results to return a single move.

10.1 Search-Position

Search-Position can be divided into two phases, the decomposition of a position into subproblems and the searching of the respective subproblems. The first phase can run in $O(n+b)$ time because we can maintain a minimum spanning tree in $O(n)$ time and partition the subproblems in $O(b)$ time where n is the number of pieces in the position and b is the size of the board. The second phase runs in $O(2^n)$ time because the minimax algorithm is exponential. We must provide the splitting threshold T_{thresh} to indicate the granularity at which Find-Subproblems should partition the board.

```
SEARCH-POSITION($P, T_{thresh}$)
  $G_T \leftarrow$ COMPLETE-GRAPH($P$)                 ▷ Compute complete graph with $T[u, v]$
  $M \leftarrow$ COMPUTE-MST($G_T$)                    ▷ Compute or update minimum spanning tree
  $S \leftarrow$ BUILD-SUBPROBLEMS($T_{thresh}, P, M$)  ▷ Use the minimum spanning tree to find subproblems
  $move \leftarrow$ SEARCH-SUBPROBLEMS($S$)            ▷ Search the subproblems (Phase 2)
  return $move$
```

10.2 Complete-Graph

The Complete-Graph module creates a vertex for each piece in the position and an undirected edge for every $u, v \in P$ with weight $T[u, v]$. We established the function $T[u, v]$ in 7 where T represents the distance in time that one piece is from another. Complete-Graph is just a preprocessing step to translate the position into G_T.

10.3 Compute-MST

It does not matter what algorithm we use to compute the minimum spanning tree of G_T because the $O(2^n)$ term of minimax dominates any minimum spanning tree finding algorithm's run time. We can even recompute the minimum spanning tree from scratch rather than updating it, which makes the implementation more simple. If we elect to implement the Update-MST algorithm from Section 8.2, Compute-MST could determine which vertex has moved, update the previously computed minimum spanning tree and return the result in $O(n)$ time.

We will use Prim's minimum spanning tree algorithm which starts at any given node and continuously adds the lightest edge in the minimum spanning tree's periphery until every vertex has been reached[15, page 572]. The running time of an adjacency list implementation of Prim's is $O(n^2)$ and will suffice for our purposes. (Remember the best minimum spanning tree algorithms runs in $O(|E|)$ which is $\Theta(n^2)$ in our complete graph.)

```
MST-PRIM($G, w, r$)
for each $u \in V[G]$
    do $key[u] \leftarrow \infty$
       $\pi[u] \leftarrow$ NIL
$key[r] \leftarrow 0$
$Q \leftarrow V[G]$
while $Q \neq \emptyset$
    do $u \leftarrow$ EXTRACT-MIN($Q$)
       for each $v \in Adj[u]$
           do if $v \in Q$ and $w(u, v) < key[u]$
                 then $\pi[v] \leftarrow u$
                      $key[v] \leftarrow w(u, v)$
```

10.4 Find-Subproblems

Given T_{thresh} and the minimum spanning tree of a position, we can find subproblems by removing the largest minimum spanning tree edge crossing the threshold, creating two disconnected components, m_1 and m_2. New positions can be built by traversing m_1 and m_2, adding the pieces in each component and setting off-limits squares using the algorithm Partition-Board algorithm given in Section 9.4. Each subproblem will consist of a disconnected component (now the minimum spanning tree of the subproblem,) a set of pieces in a position and a partition of the board which marks the in and out-of-bound squares. If a split occurred, each of the new subproblems can be fed back into Find-Subproblems recursively to make additional splits as necessary. Figure 15 demonstrates the recursive decomposition of a position in which a position is broken into two subproblems, one of which is recursively broken into two more subproblems.

After recursive binary splitting, a position will be decomposed into subproblems with minimum spanning trees containing no edges of weight greater than T_{thresh}. Rather than splitting recursively, we can remove all edges greater than T_{thresh} and make subproblems from the disconnected components.

```
BUILD-SUBPROBLEMS($T_{thresh}, P, M$)
$S \leftarrow \emptyset$
while $LargestEdge[M] > T_{thresh}$
    do Remove $LargestEdge[M]$
for each component $c' \in M$
    do
       Create new position $p'$ with pieces in $c$
       $S \leftarrow S \cup p'$
return $S$
```

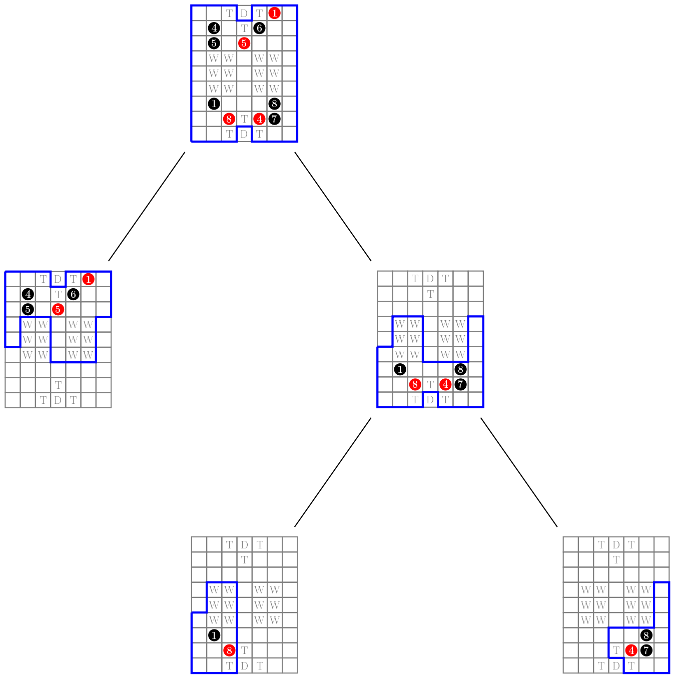


Figure 15: Finding the subproblems of a position. A position is broken into two subproblems, one of which is recursively broken into two more subproblems.

10.5 Search-Subproblems

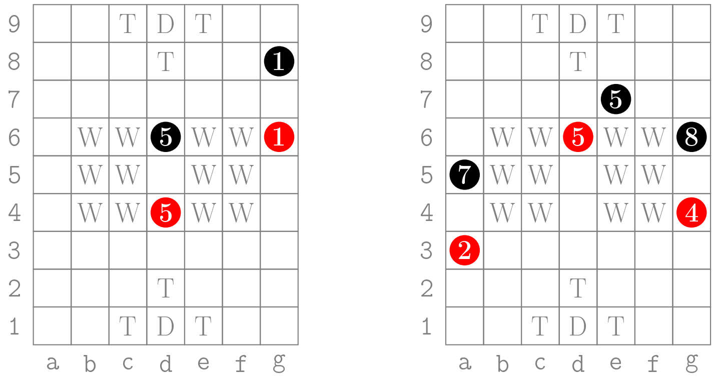
Running minimax against each subproblem is done with iterative deepening[6] to guarantee that a principal continuation will be available when the allotted time expires. Iterative deepening also allows the search to give higher priority to wins at shallower depths by stopping the search if a principal continuation leading to a win is found. Prioritizing shallow wins over deep wins is an essential part of split searching because the difference between winning and losing is often the depth at which a win and a loss occurs in two subproblems. The losing subproblem can be abandoned if the winning one will be resolved in less time.

10.5.1 Non-Moves

We have discussed the two effects that pieces can have in close proximity, but when a piece is forced to move or not move, the effect can span any distance. Only one piece can move at a time across all subproblems, so every subproblem must consider the possibility that a turn will not be taken. When searching a subproblem, a non-move is a legitimate option at each ply. So we relax the rules of the game to artificially insert a non-move to the generated list of moves at each ply.

To justify the use of non-moves, we must consider the motivation for their use in a continuation. The minimax algorithm will be selecting the move at each level that leads to

[6] searching repeatedly at increasing depths of $\{2, 4, 6, ...\}$



(a) A parity situation

(b) A hopeless situation

Figure 16: Non-move biased positions. In Figure (a), player A must move but cannot advance on player B. In Figure (b), player A cannot advance or get around B by using another aisle.

the best continuation and will include non-moves only when they are beneficial. According to our evaluation function, as defined in Section 3.3, there is always a marginal benefit to advancing one's pieces toward the opponent's den. Even if the search does not run deep enough into the game tree to see a move across the board, four to six plies should be enough to see some advantage to movement. There exists an inherent bias toward moving one's pieces because a non-move will not result in a better score.

We will address only the exceptions in which a non-move is preferable to a move. These exceptions are when 1) the game has been reduced to parity and 2) all paths leading to higher scores are blocked. Consider the parity situation presented in Figure 16 (a) in which the first player to move will be forced to retreat and eventually lose the game. It is advantageous for Player A to make a non-move and defer to player B. The second situation presented in Figure 16 (b), in which all paths to higher positions are blocked, is similar to the parity situation. In the short-run, retreating with `a3` and `g4` will avoid higher losses to player A, but in the long-run the game belongs to player B. Player A may opt for a non-move to wait and see what will happen because there is no advantage to getting a head start on a retreat.

The parity situation will likely result in a principal continuation of entirely non-moves, but according to the rules of the the game, Player A must move. To provide for this eventuality, we search each subproblem twice, first with a move and then with a non-move. The result is a move/non-move pair of scores which can be used to select the subproblem into which the player's turn should be invested. Electing to move in one subproblem is implying non-moves in the others. Because each piece is an atomic part of the positional evaluation function, scores from separate subproblems can be summed and compared.

For a set of subproblems $\{s_1, s_2, ..., s_m\}$ the implied score of each subproblem is the sum of its move score and every other non-move score.

$$ImpliedScore[s_i] = MoveScore[s_i] + \left(\sum_{j=1}^{m} NonMoveScore[s_j]\right) - NonMoveScore[s_i]$$

10.5.2 Search-Subproblems Algorithm

```
Search-Subproblems($S$)
$PC \leftarrow \emptyset$
while time remains                                ▷ Search a long as time permits
    do for each $d \in \{2, 4, ...\}$             ▷ Iterative deepening
        $max \leftarrow -\infty$                  ▷ Maximum implied score
        for each subproblem $s \in S$ with position $p$
            do
                $MoveScore[s] \leftarrow -\infty$
                for each move $m \in p$           ▷ Search all moves and a non-move
                    do
                        $p' \leftarrow$ Make-Move($p, m$)
                        $a \leftarrow -$Minimax($p', d-1$)
                        if $a > MoveScore[s]$
                           then $MoveScore[s] \leftarrow a$
                                $Move[s] \leftarrow m$
                $NonMoveScore[s] \leftarrow -$Minimax($p, d-1$)
        for each subproblem $s \in S$
            do if $ImpliedScore[s] > max$
                $max \leftarrow ImpliedScore[s]$
                $PC \leftarrow Move[s]$
        if $max = \infty$ return $PC$             ▷ Prioritize shallow wins
return $PC$
```

11 Results

To test our split-searching idea, we compare it to an alternative approach, the standard heuristic enhancement of minimax, alpha-beta pruning with move-ordering.

Alpha-beta pruning is a simple modification of the minimax algorithm that does not search branches of the game tree that cannot lead to a better score. It reduces the average number of moves explored at each position which increases the depth that can be searched. Figure 17 shows a tree with a branching factor of two and a depth of two. The left side terminates with nodes valued at 1 and 2 and the right side terminates with nodes valued at 0 and the variable x. When the tree is searched to maximize the root score, alpha-beta pruning will trim the branch under x because its value cannot affect the outcome of the search. The left side will return a value of 1 and the right side will return a value less-than or equal-to 0 and, because the algorithm is maximizing the root, it will never choose the right side.[6] Alpha-beta pruning trims all branches that follow this pattern by keeping track of the best score found at each level throughout the search.

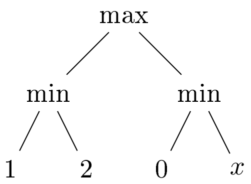


Figure 17: Alpha-beta pruning trims branches that do not affect the outcome of the minimax search. The left side returns a value of 1 and the right side will return a value less-than or equal-to 0. The branch under x is trimmed because minimax will not select the right half.

Move-ordering is an enhancement of alpha-beta pruning which searches the moves at each position in a specific order based on how likely they are to cause a refutation. We search capturing moves first, then lion, tiger, elephant and mouse moves, and then all remaining moves. On average, searching with move-ordering increases the depth of our searches by 16%. Move-ordering is a useful heuristic to compare to split-searching because it is relatively easy to implement and, like split-searching, its benefits are from increased search depth.

We ran tests on four configurations of split-searching and move-ordering:

1. Move-ordering vs. no heuristic
2. Split-searching vs. no heuristic
3. Split-searching vs. move-ordering
4. Split-searching and move-ordering vs. move-ordering

Our tests consisted of two computer simulated players, each with different searching heuristics, starting from a variety of interesting positions. Twenty positions were selected for testing, eight of which were modified starting positions in which one of player B's pieces was removed to put it at a disadvantage. One position was the unmodified starting position and eleven were scenarios from games in which one player had made a mistake or the board had become imbalanced for some reason. An example of an interesting position is given in Figure 18 where Player A has advanced too quickly with its mouse (1) up the right aisle and its elephant (8) up the left aisle. Player B's elephant is capable of running down the center aisle, bypassing player A's mouse and elephant. (Recall that only a mouse or an elephant can kill an elephant.) All of the 20 positions are given in Appendix A along with explanations of their significance.

The two players of a configuration played every position twice, once with the first player on side A and the second player on side B, and then again with the first player on side B and the second player on side A. Even though some of the positions were more advantageous for one side or the other, two equal players should have an equal number of wins and losses because they played both sides of every game.

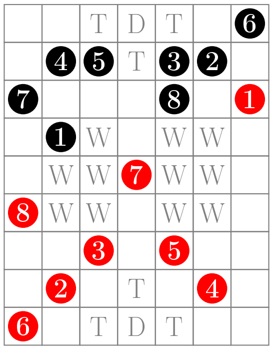


Figure 18: Player A has advanced too quickly with its mouse (1) and elephant (8). B's elephant can by-pass A's mouse and drive down the center aisle or B's tiger (7) could leap across the water before A's elephant can move down to defend.

Information was recorded in a text file for every game played, including every position of the game, every split-search and the subproblems it used, and the search depth of each player. Queries were written in Perl to parse the files for statistics about wins and losses, average search depth, game duration, split frequency, split timing and subproblem size. The algorithms for subproblem discovery and searching were implemented as presented in this thesis with addition of the alpha-beta pruning heuristic, which was written into the minimax module.

We played over 250 games using approximately 8.7 days of CPU time. Players were given 30 seconds per move and the game was called a draw if it lasted more than 200 moves, which was a reasonable cut-off point because almost all completed games had 98 moves or less. All games were played on a Linux cluster of Intel Xeon nodes running at 2.83 Ghz and each game was given access to only one CPU.

11.1 Win/Loss Record

A summary of the win/loss/draw record of all four configurations is given in Figure 19. Because every player had a chance to play both sides of every position, a configuration with equal players should show an equal number of wins and losses. Configurations 1 and 2 did well against no heuristic, but split-searching in configurations 3 and 4 was detrimental.

Conf.	Player	Wins	Draws	Losses	Opponent
1.	move-ordering	15	18	7	no heuristic
2.	split-searching	10	24	6	no heuristic
3.	split-searching	3	20	17	move-ordering
4.	both heuristics	6	20	14	move-ordering

Figure 19: Win/loss/draw record for all configurations. Equal players should show equal number of wins and losses. Configurations 1 and 2 did well against no heuristic, but split-searching in configurations 3 and 4 was detrimental.

Figure 20 shows the average search depth over all games for each player. Each additional heuristic increased search depth.

Player	Avg. Depth (plies)
no heuristic	7.76
move-ordering	8.97
split-searching	8.06
both heuristics	9.28

Figure 20: Average search depth for the four players. Each additional heuristic increased search depth.

11.2 Analysis

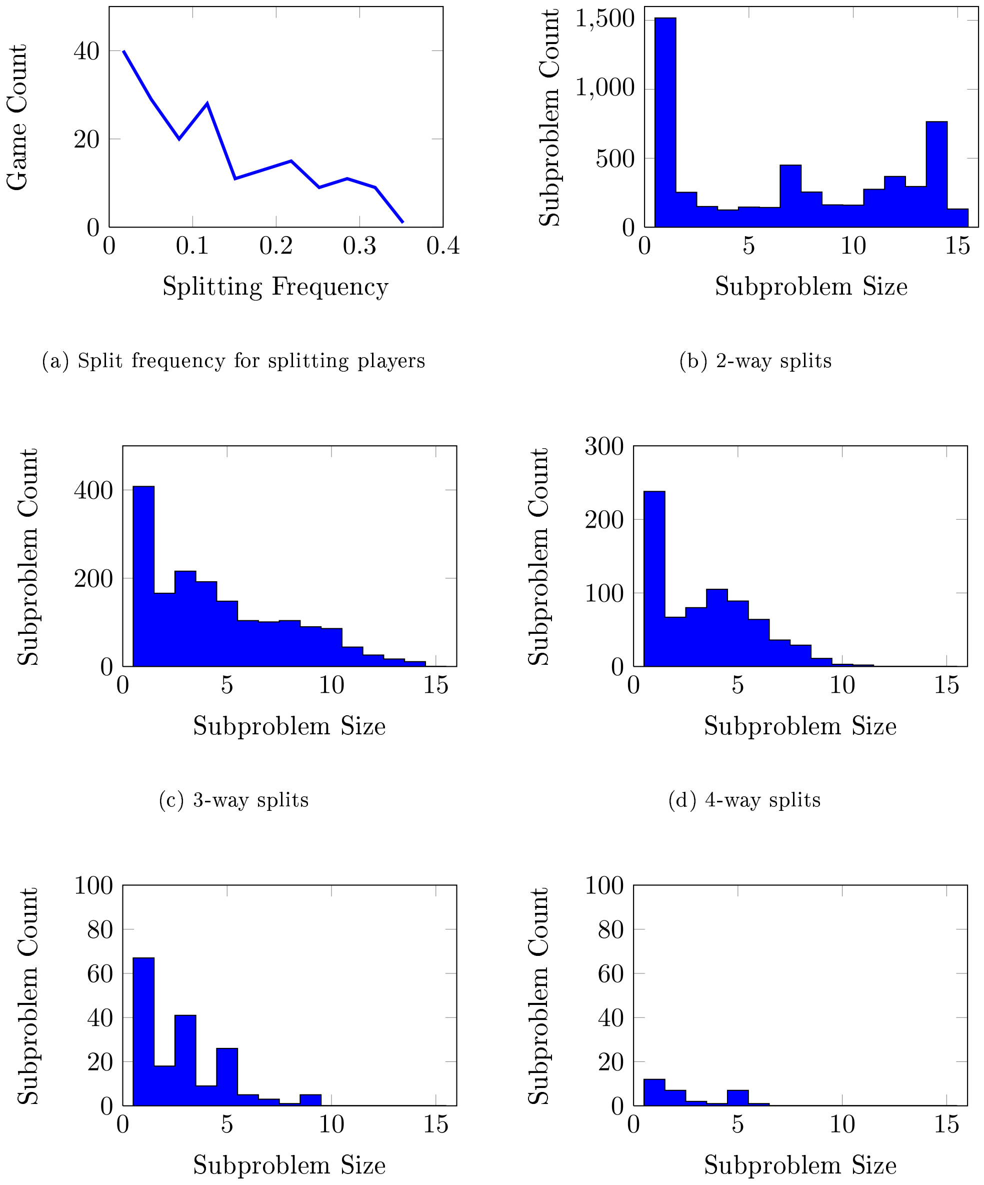
Move-ordering is better than nothing because it won eight more games than it lost. The move-ordering player was able to search 16% deeper which was enough to give it an edge over its non-move-ordering opponent. Increased search depth was expected and this configuration serves as a baseline for comparing move-ordering to split-searching.

Split-searching was also an improvement over nothing, winning four more games than it lost. However, its performance was inferior to move-ordering, a fact which follows logically from the lesser increase in depth, which was only 4%. Split searching was most frequently conducted on a 2-way split of the board, but splits of up to 6-ways were recorded. The performance of split-searching fell far short of the theoretically possible 25%, established in Section 4.1. This was due to the infrequent occurrence of splitting and the imbalanced nature of the subproblems.

Figure 21 (a) charts split frequency, which is the ratio of split-searched positions to all positions, per game, per splitting-player. No player was able to split more than one-third of the positions in a game and most of the games saw very infrequency split-searching. The lack of consistent splitting meant that there was not a consistent benefit from increased search depth. Additionally, the distribution of pieces among the subproblems adversely affected the efficiency of the split-search which benefits most from an even split. A subproblem of 15 pieces is not much of an improvement over a subproblem of 16 pieces. Figure 21 (b)-(f) show the histograms of n-way split subproblem sizes. Most of the splits were 2-way and a large portion of all splits created subproblems with only one piece.

When split-searching was played against move-ordering, the latter was shown to be more effective, winning by a large margin. This was expected after analysis of the first two configurations, which showed move-ordering to be a greater improvement than split-searching over nothing.

Both move-ordering and split-searching were independently successful, but split-searching on top of move-ordering was detrimental to performance. This was surprising because both heuristics benefit from increased search depth and together they were able to search more deeply on average than any other configuration. The underlying theory of split-searching involves finding and searching loosely coupled subproblems, which cannot affect each other



(a) Split frequency for splitting players

(b) 2-way splits

(c) 3-way splits

(d) 4-way splits

(e) 5-way splits

(f) 6-way splits

Figure 21: Splitting frequency and subproblem size by n-way split. Most splitting players found subproblems less than one-third of the time. Most of the splits were 2-way and a large number of the subproblems consisted of one piece, a split which does not provide much depth increase. However there were a significant number of somewhat evenly split 2-way subproblems between 5 and 10 pieces each.

for as long as the splitting threshold. In all of our tests, we set the threshold at four, so searching beyond four plies started trimming large parts of the game tree. The average search depth when using both heuristics was 9.28 plies, well beyond the splitting threshold.

Sec/Move	Wins	Draws	Losses
30	6	20	14
5	10	21	9
2	8	19	13

Figure 22: Win/loss/draw record for configuration four tests. Five-second searches out-performed both two and thirty-second searches.

Sec/Move	Avg. Depth (plies)
30	9.28
5	8.19
2	7.85

Figure 23: Average search depth for configuration four tests. Decreasing search times led to decreasing search depths.

It is our conjecture that the effectiveness of a split-search is related to our search depth and separation of the subproblems. When setting the splitting threshold, we should take the intended search depth as an input to determine how decoupled subproblems must be before they can be split-searched. If enough resources are available to search far beyond a threshold value, split-searching should not be used.

To test the relationship between depth and split-search performance, configuration four was tested two more times, allowing five and two seconds per move instead of 30, to decrease the depth searched by the combined heuristic player. The results are given in Figures 22 and 23 in which five-second searches out-performed both two and thirty-second searches, even through decreasing search times always led to decreasing search depths.

While it appears that search depth has some correlation to the performance of move-ordering with split-searching, it is not clear what the relationship is. A more linear increase in performance was expected from 30-second searching to 2-second searching as search depth decreased, but 5-seconds had the best performance.

11.3 Conclusion and Future Work

In this thesis we were able to use a physics-inspired model of causality, the intersection of effect cones, to capture minimum time-to-effect between every pair of pieces in a Dou Shou Qi position. Using a minimum spanning tree of this complete graph, we found clustering structure that corresponded to loosely coupled subproblems and searched them independently and more deeply, leading an improved performance of the minimax algorithm.

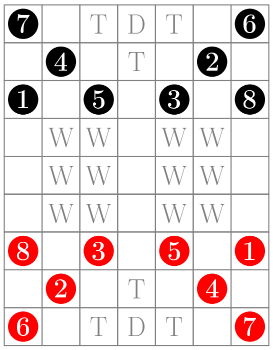
The depth to which subproblems can be searched independently appears to be limited by how far apart they are. Increased depth is a benefit of split-searching, but we cannot search subproblems too deeply. When combined with a depth-increasing heuristic like move-ordering, the benefits of split-searching are lost.

The splitting threshold controls the granularity of subproblem discovery such that split-searching is not over or under utilized and we believe that it should take depth into account. Throughout the thesis and our testing, when a splitting threshold was needed, it was set to 3 or 4. It remains undiscovered what the ideal function to determine a splitting threshold is and what exactly its inputs are.

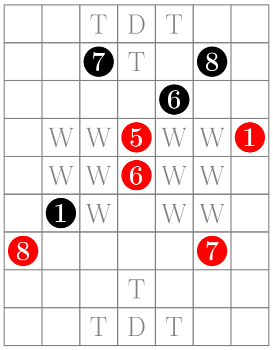
The game theoretical consequences of relaxing the rules of Dou Shou Qi to allow non-moves are not fully known. It would be useful to conduct a more formal analysis of the role that non-moves play in searching loosely coupled subproblems.

Appendix A - Interesting Positions

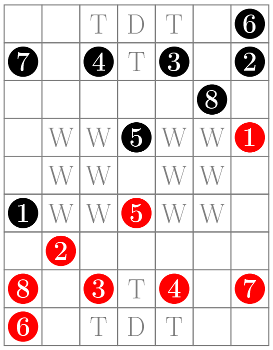
The following are the 20 positions used for testing:



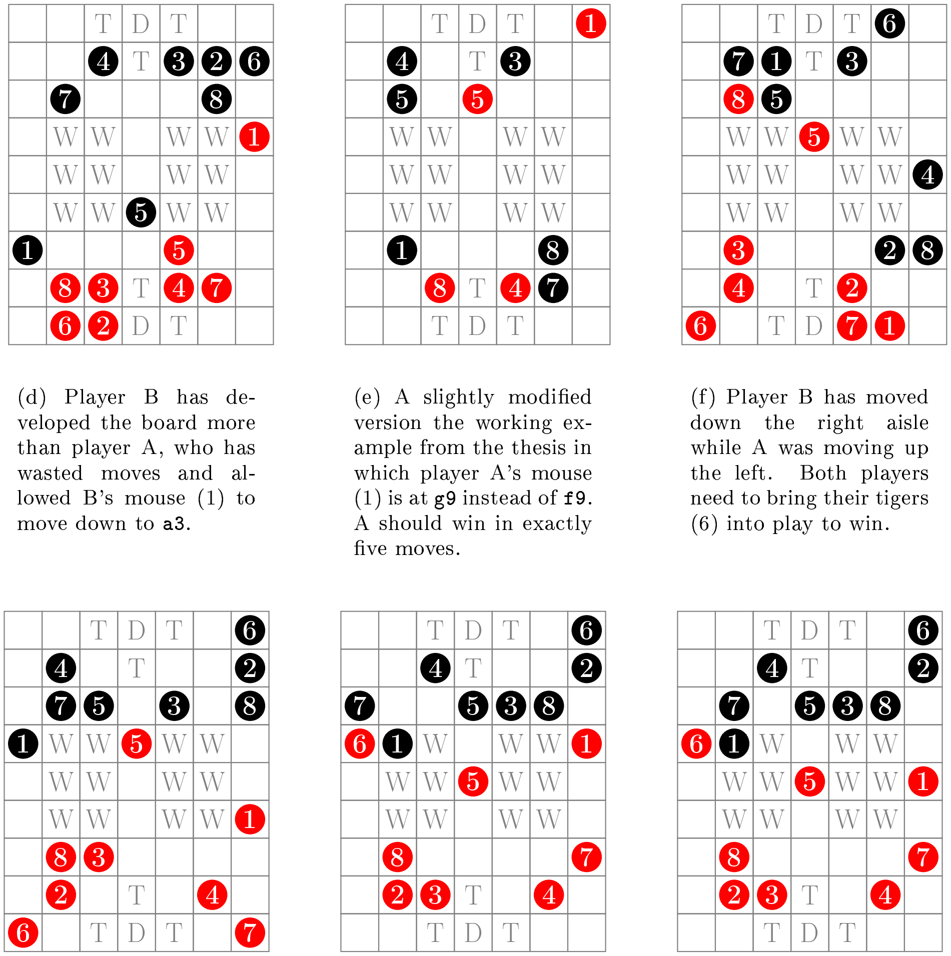
(a) The Dou Shou Qi starting position.



(b) Player A is putting pressure on the upper-right quadrant of the board with its mouse (1) and lion (7), but A needs to get its elephant (8) at `a3` past B's mouse at `b4` to win the game.



(c) Player A has moved its elephant (8) away from the water and player B has not. Because an elephant along the water is essential for preventing the opposing lion (7) from jumping over the water, this position should be disadvantageous for player A.



(d) Player B has developed the board more than player A, who has wasted moves and allowed B's mouse (1) to move down to `a3`.

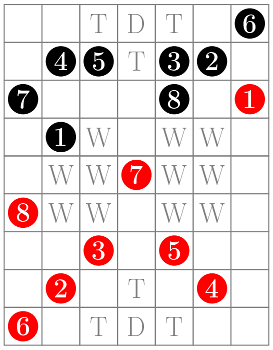
(e) A slightly modified version the working example from the thesis in which player A's mouse (1) is at `g9` instead of `f9`. A should win in exactly five moves.

(f) Player B has moved down the right aisle while A was moving up the left. Both players need to bring their tigers (6) into play to win.

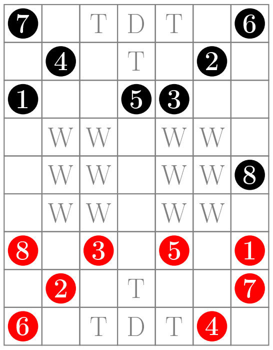
(g) Player B has allowed A's hyena (5) to advance too far and now B's lion (7) is stuck. B's hyena should have moved over sooner to give the lion room to operate.

(h) Player A has let B's lion (7) get too close to its tiger (6) and he is going to lose its hyena (5) in the retreat. A should have seen this coming when B moved b4-b5.

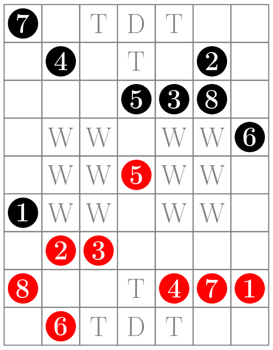
(i) This position immediately precedes position (h). If player A moves g5-g6, B will move its lion over and chase A's tiger.



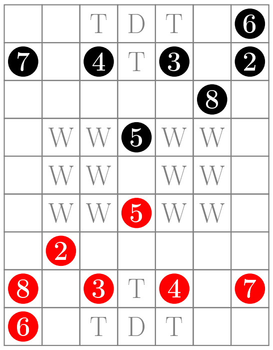
(j) Player A has been too aggressive. B's elephant (8) can by-pass A's mouse (1) and drive down the center aisle or B's tiger (7) could leap across the water before A's elephant can move down to defend.



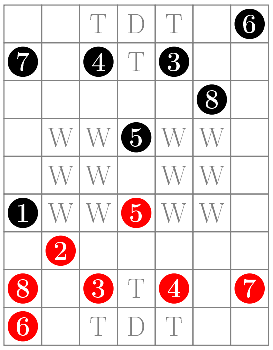
(k) Player B seems to have forgotten that a mouse (1) can capture an elephant (8) and is going to be chased all the way back to its side. Player A should sacrifice its mouse to kill the elephant so its lion (7) can jump over the water.



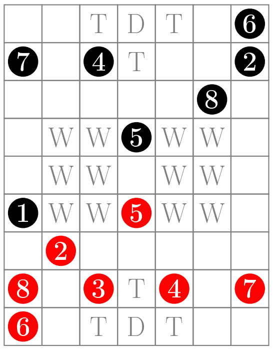
(l) Player A has left its hyena (5) at `d5` as B brought down its tiger (6) to `g6`. B's tiger will be able to come down the center aisle and take a few pieces.



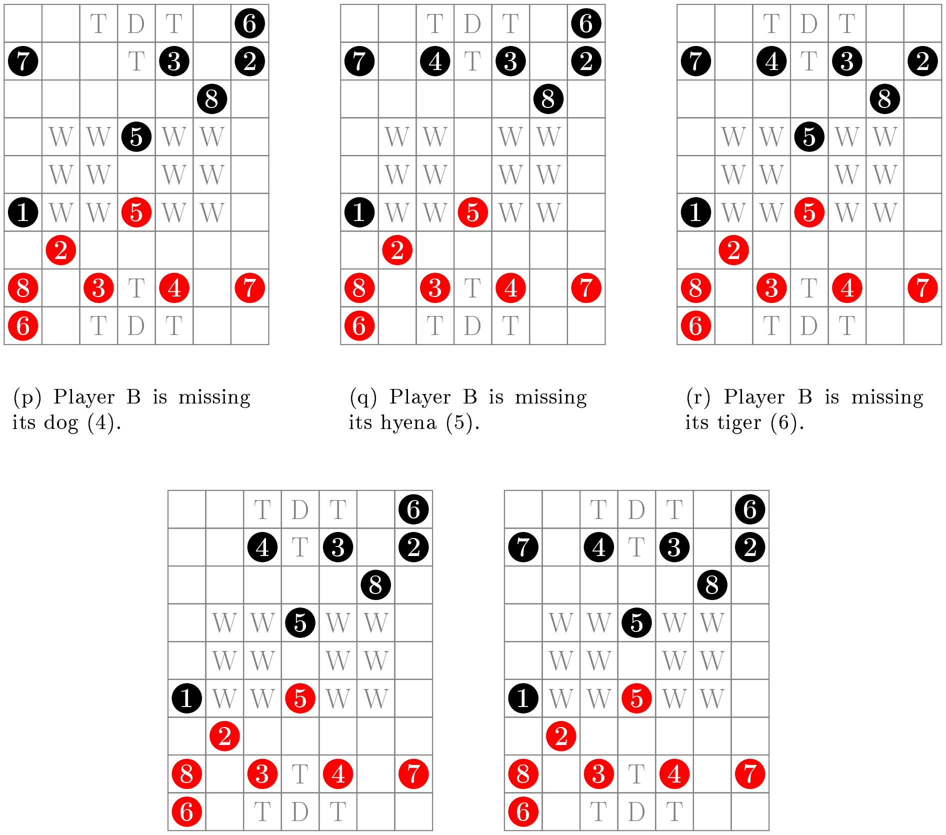
(m) Player B is missing its mouse (1).



(n) Player B is missing its cat (2).



(o) Player B is missing its wolf (3).



(p) Player B is missing its dog (4).

(q) Player B is missing its hyena (5).

(r) Player B is missing its tiger (6).

(s) Player B is missing its lion (7).

(t) Player B is missing its elephant (8).

References

[1] AnchientChess.com. Dou shou qi. http://ancientchess.com/page/play-doushouqi.htm. Accessed 3/19/2010.

[2] Elwyn Berlekamp and David Wolfe. *Mathematical Go.* A K Peters, Ltd, 1994.

[3] Vincent de Boer. *Invincible: A Stratego Bot.* PhD thesis, Delft University of Technology, 2007.

[4] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *Journal of Algorithms*, 17:237–250, 1994.

[5] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees. *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, 1:252–257, 1983.

[6] John Hughes. Why functional programming matters. *Chalmers memo*, 1984.

[7] David Levy and Monty Newborn. *How Computers Play Chess.* Computer Science Press, 1991.

[8] Joe Malkevitch. Taxi. http://www.ams.org/featurecolumn/archive/taxi.html. Accessed 3/19/2010.

[9] Wolfram Mathworld. Voronoi diagrams. http://mathworld.wolfram.com/VoronoiDiagram.html. Accesses 3/19/2010.

[10] Chiara Casolino Michele Florina, M. Concepción Cerrato Oliveros and Monica Casale. Minimum spanning tree: ordering edges to identify clustering structure. *Analytica Chimica Acta*, 515:43–53, 2004.

[11] Guillermo Owen. *Game Theory, Third Edition.* Academic Press, 1995.

[12] Ludek Pachman. *Modern Chess Strategy.* Dover Publications, Inc., 1963.

[13] Anatol Rapoport. *Two-Person Game Theory.* The University of Michigan Press, 1966.

[14] Celso C. Ribeiro and Rodrigo F. Toso. Experimental analysis of algorithms for updating minimum spanning trees on graphs subject to changes in edge weights. *Lecture Notes in Computer Science*, Experimental Algorithms:393–405, 2007.

[15] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001.

[16] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior.* Princeton University Press, 1953.

[17] Wikipedia. Minimax. http://en.wikipedia.org/wiki/Minimax. Accessed 3/19/2010.