Computer Players in the *Star Chess* Game

Algorithms and Comparison to Chess

Christoph Nahr
christoph.nahr@kynosarges.org

Published 02 December 2016 on the Star Chess home page
http://kynosarges.org/StarChess.html

Abstract

This document examines the computer players in the *Star Chess* strategy game. The current versions of this document and the game itself are available at the *StarChess* home page. Please see there for system requirements and other information.

Online Reading. This document contains a "Bookmarks" navigation tree. Click on any tree node to jump to the corresponding section. Moreover, all phrases in blue color are clickable hyperlinks that will take you to the section or address they describe.

Colophon. This document was written in LaTeX using MiKTeX 2.9 with XeLaTeX, KOMA-Script, and various other packages. See LaTeX Typesetting with MiKTeX for details.

The UML diagrams were reverse-engineered from the compiled Java JAR files, using my free Class Diagrammer application, and embedded as PDF files.

Body text is set in Minion 12 pt from Adobe's Minion Pro collection, designed by Robert Slimbach. Subtitles and diagram text are set in various sizes of Myriad from Adobe's Myriad Pro collection, designed by Robert Slimbach and Carol Twombly.

Identifiers and code fragments outside of UML diagrams are set in Microsoft's Consolas, designed by Lucas de Groot. The font is artificially compressed by 20% to take up less space.

Date	Version	Game	Description
2016–12–02	1.4.2	2.0.4	Updated diagrams to Class Diagrammer 2.1.0
2016–05–01	1.4.1	2.0.4	Added UML diagrams for implementation
2014–05–16	1.4.0	2.0.0	Revised for rewritten Java program
2012–06–09	1.3.0	1.2.5	Changed typesetting to LaTeX with MiKTeX
2001–10–01	1.2.1	1.2.5	Some spelling and grammar corrections
2001–02–20	1.2.0	1.2.0	Revised for rewritten C program
1999–08–17	1.1.0	1.1.0	Score penalty doubled for stronger adversaries
1999–08–07	1.0.0	1.0.1	Initial release, using LaTeX with Y&Y TeX

Contents

Chapter 1

Introduction

Star Chess is a turn-based computer strategy game written by the author of this document. The game executable (requires Java 8 or later) and its source code are available for download at the *Star Chess* home page. A complete explanation of the game mechanics may be found in the game's help system. A short overview of the game rules is given in Appendix A.

This document focuses on the computer players, also called "artificial intelligence" or "AI players," that were built into *Star Chess*. Computer players can take the place of any of the four sides in the game, just like chess programs can play against human opponents.

1.1 Empire Building Games

We begin with a description of the game genre of which *Star Chess* is an example, and how games in this genre differ from well-researched traditional board games such as chess.

Star Chess is an "empire building game:" the player wins by building a successful empire composed of combat units (pieces) and cities or colonies (possession of certain squares). Although a far simpler game, it is similar in kind to computer games such as *Civilization* or *Master of Orion*. Generally speaking, *Star Chess* differs from chess in the following ways:

1. There are more than two players.
2. The initial board setup is partly random.
3. A player may move more than one unit per turn, or no unit at all.
4. Board locations may contain immobile player assets as well as mobile units.
5. More than one unit may occupy a single board location.
6. The outcome of attacks on units or immobile assets is determined by a nontrivial combat resolution system.

7. Additional units and immobile assets may be created throughout the game.

8. A resource management system determines which units or assets can be created.

Items 1–2 and 7–8 are unique to empire building games, while items 3–6 would constitute a pure wargame if taken by themselves. Commercial empire building games such as the classics mentioned above usually include these additional features:

9. A research system that creates various advantages in return for invested resources.

10. An initially unknown "map" (board) that each player must gradually uncover.

11. A diplomacy system regulating player co-operation, from armistice to alliances.

12. An espionage system to weaken other players or acquire their knowledge.

13. Beneficial or detrimental random events beyond the players' control.

All of the listed differences have the same important consequence: they make empire building games much more complex than traditional board games, in the sense that there is a much greater number of possible moves per turn.

1.2 Computer Players

Computer strategy games that are simply electronic versions of existing board games, e. g. chess programs, usually play under the exact same rules and restrictions as a human player would. However, computer strategy games that use an original game system, e. g. empire building games, usually include difficulty settings that allows their computer players to "cheat," i. e. habitually break the rules of the game in their favor. Many players are dissatisfied with the strength of most computer opponents unless the latter are allowed to "cheat" in this sense.

As we shall see, the lack of strong computer opponents playing by the rules is not a technical defect of contemporary empire building games, but the result of a mathematical complexity which far exceeds that of chess and similar games. Even *Star Chess,* although a rather simple game by today's standards, turns out to have more than *one million times* as many possibilities per player turn as chess has!

Nevertheless, the design goal for the computer players of *Star Chess* was just that: they should offer a credible challenge to human players while obeying the same rules. The implemented solution is a combination of a *minimax prediction tree,* similar to the one used in chess programs, and a *request generation scheme* that greatly reduces the number of possible moves fed to the prediction tree. While *Star Chess* itself is a relatively simple and uninteresting game, its complexity is great enough to illustrate the problems faced when designing computer players for empire building games, and perhaps offer some ideas that might prove useful for more complex game designs.

Chapter 2

Concepts

This section explains the concepts behind the *Star Chess* computer players. First we take a look at minimax algorithms as they are commonly used in computerized board games. As it turns out, this concept requires some modification in order to be used with empire building games because of the latters' immense complexity. So we propose a more sophisticated turn generation method that effectively "pre-prunes" the prediction tree.

2.1 Minimax Prediction Trees

The lecture notes by Charles R. Dyer and Jim Gast [1] provide an excellent introduction to prediction trees, minimax algorithms, and alpha-beta pruning. These topics are covered by a wide range of online and printed publications; some references are provided by Tony A. Marsland [2] and Aske Plaat [3]. We shall therefore not attempt any exhaustive explanation of these topics but rather assume that the reader is already broadly familiar with them.

2.1.1 Turn Generation and Evaluation

The situation is as follows. The computer player has just started its turn. The task is to select a move (in board games) or a sequence of commands (in empire building games) that will improve its standing. The basic solution is to generate *all* legal moves or command sequences so that we come up with a collection of all legal end-of-turn configurations for the current player. These configurations are then evaluated by a scoring function, and the particular move or command sequence that resulted in the highest score is finally executed as the computer player's turn. We therefore have a three-step operation:

1. Generate all possible end-of-turn configurations for the current player.
2. Evaluate each end-of-turn configuration with a scoring function.
3. Execute the commands that resulted in the configuration with the best score.

Note that real-world programs usually do not generate all configurations prior to evaluation but rather generate and evaluate one configuration at a time, saving across iterations the best evaluations and their generating sequences.

2.1.2 The Evaluation Function

A good evaluation function is crucially important since its results decide which move or command sequence is ultimately executed. Chess programs usually examine "material" as well as "positional" characteristics of a given configuration, i. e. they judge the number and values of the pieces on the board as well as their absolute and relative locations.

Star Chess is not quite as advanced, only taking into account a player's material assets but ignoring their placement on the board. Given the game's simplicity it is doubtful that a positional evaluation would be beneficial. However, more complex empire management games should probably account for the possession of important locations such as narrow land bridges or main traffic routes.

The evaluation function used in *Star Chess* is described in section 3.4. Basically, a player will receive higher scores the more ships and colonies he owns, and the fewer ships and colonies his opponents own.

2.1.3 The Minimax Algorithm

In practice, chess programs are not content to look only at the moves available to the current player. They look ahead to a possible counter-move by the opposing player, then to any moves responding to the counter-move, and so on. The resulting structure is known as the prediction tree. Its purpose is to determine which move will turn out to produce the best evaluation *in the long run,* enabling the computer player to trade short-term losses for long-term benefits or to pick the best move among choices that seem equivalent at first glance.

For this prediction to work, we must take into account that subsequent players will try to optimize their scores just as the current one does. Chess programs commonly use a single evaluation function for both players, interpreting a high result as beneficial to white and detrimental to black, and vice versa for a low result. This means that in order to optimize their respective scores, white seeks high (maximal) evaluations and black seeks low (minimal) evaluations. Hence the name "minimax algorithm," although the concept really works just as well for games with more than two players (where the name is inappropriate since all players try to maximize their own scores).

Table 2.1 summarizes the levels of a "minimax" prediction tree for a game of P players, computed to a depth of $l_{\max}$ levels where each level represents one future player turn and $p(l)$ is the player active on level l. Prediction tree levels are usually called "plies" in the chess literature.

Positions[1] are only evaluated on the very last level. All previous levels merely examine the results returned by the next deeper level, and return the optimal result for the indicated

1. The term "position" is used interchangeably with "configuration" from now on.

Level	Positions	Player	Return position that optimizes…
1	n	1	Score for player 1 as returned by level 2
2	n^2	2	Score for player 2 as returned by level 3
⋮	⋮	⋮	⋮
P	n^P	P	Score for player P as returned by level $P+1$
$P+1$	n^{P+1}	1	Score for player 1 as returned by level $P+2$
⋮	⋮	⋮	⋮
l	n^l	$p(l)$	Score for player $p(l)$ as returned by level $l+1$
⋮	⋮	⋮	⋮
$l_{\max}$	$n^{l_{\max}}$	$p(l_{\max})$	Score for player $p(l_{\max})$ on this level

Table 2.1: Minimax Optimization

player to the calling level. When the tree is fully processed the caller will know which first-level position is likely to result in the best score for the current player after $l_{\max}$ player turns.

As shown in the "Positions" column, the number of positions that have to be generated and evaluated grows exponentially with the depth of the prediction tree. However, deepening the tree results in stronger computer players. Chess programs therefore employ various tree pruning schemes to deepen the tree while reducing the number of generated positions. The most fundamental pruning algorithm is known as alpha-beta pruning.

2.1.4 Alpha-Beta Pruning

Alpha-beta pruning relies on a very convenient feature of two-player zero-sum games: whenever one player's score rises, the other player's score is guaranteed to fall by the same amount. Since level l will always return a position that maximizes the score for player $p(l)$, it follows that the score for player $p(l-1)$ will be minimized at the same time.

Now if we already have at least one result on level l that has been handed back to level $l-1$, and we see that $p(l)$ has discovered a position that is *better* for $p(l)$ than this previous result, then we *know* that player $p(l-1)$ cannot expect any better score for himself of the current examination of level l because the current position will only be replaced by something *even* better for $p(l)$, which implies something *even* worse for $p(l-1)$. So we might as well return to level $l-1$ immediately, report failure, and restart level l with a new position.

This pruning method is particularly elegant because it is guaranteed to produce the same results as an unpruned tree. Unfortunately, it is completely useless for any game with more than two players, including most empire building games.

As soon as three or more players are present, the score of any one player will *by definition* no longer change in perfect synchronicity with that of any other player. Otherwise we would have a game of two alliances, and not one of three or more truly independent players. This in

turn means that a particularly good position for $p(l)$ is no longer guaranteed to be particularly bad for $p(l-1)$, or any other player $q \neq p(l)$. So a position favored by $p(l)$ is not certain to be rejected by any $q \neq p(l)$, destroying the basis for the alpha-beta pruning method.

2.1.5 The Complexity Gap

Although several heuristic pruning methods have been tested with *Star Chess,* all of them degrade the strength of the computer player. Therefore we delay their discussion and instead proceed to compare the complexity of mid-game situations in chess, with and without alpha-beta pruning, to those in *Star Chess,* without any pruning.

For the purpose of this discussion, "complexity" means the number of different end-of-turn positions that can be generated from a given start-of-turn position. Assuming for simplicity that this complexity is constant for all positions of a prediction tree, a game with a complexity of n positions and a prediction tree depth of $l_{\max}$ levels must generate $\sum_{l=1}^{l_{\max}} n^l$ positions and evaluate $n^{l_{\max}}$ positions each time the computer takes its turn.

Since the quality of the generated turns grows with increasing prediction tree depth, and tree depth is limited by the total number of positions that have to be generated and evaluated, low complexity results in a strong computer player, and vice versa.

Chess without pruning. According to Dyer & Gast, an average chess position allows for 38 possible moves. But we shall be more pessimistic and instead use Marsland's estimate of close to 80 moves in complex mid-game positions. We use a very modest prediction tree, looking but one full turn ahead. Since chess is a game of two players, one full turn is made up of two player turns, or "plies." So we have $n = 80$ and $l_{\max} = 2$. It follows that we must generate $\sum_{l=1}^{2} 80^l = 6480$ positions and evaluate $80^2 = 6400$ positions.

Chess with alpha-beta pruning. We use the empirical results of the *Deep Blue* team as reported by Dyer & Gast and estimate a reduced complexity of $n' = 13$ thanks to alpha-beta pruning. The number of generated positions is now down to $\sum_{l=1}^{2} 13^l = 182$, and the number of evaluated positions is $13^2 = 169$.

Star Chess. Alas, *Star Chess* has not been sufficiently researched (nor is it ever going to be, in all likelihood) to assert a reliable average mid-game complexity. We shall instead isolate a typical situation as it might, and frequently will, appear in the course of a *Star Chess* game. We disregard rarely used commands (Colonize, Terraform, Scrap Ships) in favor of movement and shipbuilding. Suppose the current player owns a colony in one of the four central sectors, as well as 20 ships in the same sector. Finally, suppose the colony can build 20 more ships.

According to the rules of *Star Chess* (see Appendix A), each of the 20 existing ships can either move to any of the eight neighboring sectors, or stay with the colony. All ships are identical which means that all positions that only differ in their generating command sequence

but not in the final number of ships per sector are counted as a single position. The total number of movement choices is staggering:[2]

$$C_{9,w}^{(20)} = \left(\begin{array}{c} 9+20-1 \\ 20 \end{array} \right) = \frac{28!}{20! \times 8!} = \frac{\prod_{i=21}^{28} i}{40,320} = 3,108,105$$

In addition, the colony can build 0–20 ships (which cannot move until the player's next turn) for a total of 21 choices. Movement and building choices are independent of each other and thus must be multiplied to obtain the total complexity $n = 65,270,205$.

Worse, *Star Chess* has four players instead of just two. Consequently, the prediction tree must be deepened to $l_{\max} = 4$ levels in order to cover one full turn. The resulting number of positions to be generated and evaluated is around 1.8×10^{31} each!

Comparison. The estimated prediction tree sizes for chess are almost trivial while those for *Star Chess* are astronomically large. Even the complexity of a single isolated *Star Chess* situation is ca. 815,878 times as high as a pessimistic estimate for a complex chess position without alpha-beta pruning. But a typical mid-game situation in *Star Chess* will feature several colonies and dozens of ships per player. Simple though it is when compared to commercial empire building games, even *Star Chess* can be expected to create situations at least one million times more complex than chess configurations.

Clearly, any attempt to use a conventional minimax algorithm for empire building games is doomed. Alpha-beta pruning does not work for these games, and even if it did it could not be expected to remove more than one order of magnitude which would still leave an insurmountable task. Since we cannot hope to generate all legal positions to choose purely by evaluation, we must somehow restrict the turn generation process in the first place.

2.2 Restricted Turn Generation

Instead of generating all possible command sequences for a given situation, we must try to generate a reasonably-sized selection of promising sequences. Sequences are called "promising" if they have a positive effect on a player's score, compared to doing nothing or acting randomly. Generally speaking, a player's score will improve if he expands and protects his empire while destroying those of his opponents.

In the specific case of *Star Chess,* our task is to find command sequences that lead to the colonization of new worlds, successful defense of existing ones (especially the home world), and successful attacks on other player's ships and colonies (preferably their home worlds). Other games will set different tasks, according to their rules and evaluation function, but the primary goal is always to generate a turn that will improve the current player's standing.

2. $C_{9,w}^{(20)}$ means that 20 times, 1 out of 9 elements is chosen. Repetitions of elements ("Wiederholungen" in German, hence *w*) are obviously allowed. The order in which elements are chosen is not taken into consideration, only the number of times they are chosen.

2.2.1 Generating Turns by Prioritized Requests

A given position will usually pose several opportunities to improve a player's score but prevent the player from exploiting them all due to insufficient resources.[3] This means we have to decide how many resources should be expended on each opportunity, as we cannot afford to grasp them all as they come along. In other words, we need to start our turn generation process with a survey of all opportunities and then decide which are the most important ones, and therefore most deserving of our limited resources.

Our solution is to generate *one request per sector* for all sectors that contain either an opportunity or a threat to the current player. Each request is assigned one out of four *priorities* as well as the *minimal and ideal fleet sizes* required to fulfill the request. We then traverse all of our fleets,[4] from biggest to smallest, and dispatch each fleet to the most important request whose minimum requirement it fulfills. More ships may be dispatched to the same request, up to its ideal fleet size. Once a request is "ideally fulfilled" the remaining fleets try to serve requests of lower priority.

The actual request mechanism implemented in *Star Chess* is more complicated and also takes into account the distances of fleets to requests, among other things. In addition to ship movement, the prioritized sector request list also drives shipbuilding and colonization, while terraforming and scrapping of ships is handled by an independent formula. A detailed description of the turn generation mechanism is given in chapter 3.

2.2.2 Generating Alternatives by Request Masking

Now that we have generated a (more or less) clever command sequence for the current turn, we face a different problem. How can a prediction tree operate on this sequence? A prediction tree evaluates alternatives, but we do not presently have any alternatives to evaluate. Randomly changing some of the generated commands would in all likelihood result in exactly the flood of worthless choices that our turn generation method was trying to avoid.

Fortunately, the request mechanism lends itself to a more rational method of spawning alternatives. In real life, leaders are frequently faced with the choice of abandoning one of several objectives in order to save the others. We simulate this situation by *masking out one or two requests* before turn generation starts. If the prediction tree now delivers a better evaluation then we know that the initial request list was trying to do too many things at once, wasting forces on pointless endeavors. As it turns out, request masking decisively improves the quality of the *Star Chess* computer players, allowing them to concentrate their fleets where they are likely to have the most impact.

The complexity of a *Star Chess* prediction tree with request masking is now determined as follows. We always have the original, unmasked request list. With 16 sectors and one request

3. In this context, "resources" refers to ships and idle population as well as the two so-called "resources" in the *Star Chess* game, namely credits and materials.

4. In the *Star Chess* context, "fleet" simply refers to all the ships in one sector.

per sector, we have at most 16 possibilities to mask one request; less if we had fewer requests to begin with. Masking 2 out of 16 requests gives another 120 possibilities at worst. So the maximal complexity per position is $n = 1 + 16 + 120 = 137$. This is worse than chess but not substantially so, especially since the worst case is rare; it would require all sectors on the map not only to be reachable by the current player, but also to be either unpopulated or under threat by another player. Empirical values are usually less than half the maximal complexity.

2.3 Additional Pruning

Despite the significant reduction in complexity achieved by request-based turn generation, looking ahead multiple turns requires further pruning of the prediction tree to deliver acceptable computer player performance.

Assuming a typical mid-game complexity of at most 60 request combinations per position, the *Star Chess* prediction tree might generate $\sum_{l=1}^{4} 60^l = 13,179,660$ positions and evaluate $60^4 = 12,960,000$ positions for one full turn. On an Intel Core i7-4650U (1.7–3.3 GHz), the current Java 8 implementation (Oracle 64-bit HotSpot Server VM) generates up to 330,000 positions per second. So one full turn may incur a noticeable delay of tens of seconds, and looking ahead multiple turns would take minutes.

Lacking an "ideal" pruning method – such as alpha-beta pruning – that does not influence the quality of the prediction tree at all, we will have to look for a good compromise that combines significant time savings with a negligible degradation of quality.

2.3.1 Omitting the Prediction Tree

Unlike "brute force" turn generation, request-based turn generation does not *require* a prediction tree in order to produce any sensible output. The command sequence generated by the original, unmasked request list is of sufficient quality to provide a computer player that deserves this name. However, testing showed that such a computer player would frequently suffer "nervous breakdowns," sending minuscule forces hither and thither in an attempt to minimally cover all requests, rather than focusing on the most important ones.

As mentioned in subsection 2.2.2, request masking boosts a computer player's strength by allowing it to concentrate its forces rather than scatter them all over the galaxy. Therefore, a preferable way to minimize execution time is to find the optimal request mask for the current player without looking ahead to any other player's turn.[5]

2.3.2 Restricting Request Masking

Problem. Omitting the prediction tree as outlined in the previous section will generally produce a competent computer player but inhibits long-term considerations. A case in point is the

5. This is what happens when the "Prediction turns" option is set to zero for a computer player.

initial colonization phase of every *Star Chess* game in which all players must choose a planet for their first new colony. It is of utmost importance that this planet is barren (and not terran) so as to complement the terran home world's resource output. However, determining the resource production of a new colony requires a prediction tree depth of at least two full turns: one turn to move the colonization fleet to the desired sector, another turn to colonize the planet. The colony will start to produce and consume resources by the end of the second turn. At this point the evaluation function will reflect the altered resource flow, enabling the prediction tree to choose the best colonization request.

Of course, the particular case described above could be taken care of by special "colonization logic." However, it was the intention of the *Star Chess* project to avoid such tricks and rely on a *general* prediction method instead. A specialized function to determine a good place to set up a new colony would have to be rewritten with every change of the relevant game parameters, and it would be completely inapplicable and of no interest to other games. Pruning and consequently deepening the prediction tree, on the other hand, is a *general purpose method* to determine the long-term implications of a command sequence.

Solution. As it turns out, the nature of request-based turn generation allows for an effective pruning method. We saw that a computer player tends to play poorly unless request masking is applied, and concluded that all request masks have to be examined for the current player. However, the command sequences generated by unmasked request lists are still useful enough *to predict other players' turns!*

Turn prediction is never perfect, for human players anyway but also for computer players. Computer player *B*, taking its turn after computer player *A*, will look ahead one turn further than *A* did, and this increased horizon distance will change *B*'s turn from what *A* expected. This means that it is pointless to invest too much time in trying to precisely predict other players' actions. Getting a general idea of their intentions is usually sufficient, and the unmasked request list provides just that.

Consequently, the following pruning method based on *restricting request masking* was implemented. The prediction tree only branches out (with up to 137 branches) on levels that examine the current player. On all other levels, only a single position is generated or evaluated, reducing the prediction tree to a linear list in these places. To sum it up:

— Generate command sequences for all possible request masks on every level l with $p(l) = p(1)$, i. e. whenever the current player is under examination.

— Generate only the command sequence for the unmasked request list on every other level.

For comparison, this is the typical complexity of a *Star Chess* prediction tree *without* the described pruning method:

$$\sum_{l=1}^{l_{\max}} 60^l \text{ positions generated}, 60^{l_{\max}} \text{ positions evaluated.}$$

Restricting request masking brings the position counts down to these values:

$$\sum_{l=1}^{l_{\max}/P} (60^l \times P) \text{ positions generated, } 60^{l_{\max}/P} \text{ positions evaluated.}$$

Note that predicting a given number of *full* turns *with* pruning requires the same amount of work as predicting the same number of *player* turns *without* pruning, multiplied by the number of players P. This is a particularly attractive feature of this pruning method as it means that adding more players will no longer exponentially increase computation time.

In practice, restricting request masking enables *Star Chess* to predict 2–3 full turns with ease, equivalent to 8–12 levels (or "plies") in a four-player game. Two full turns typically require the generation of $\sum_{l=1}^{2}(60^l \times 4) = 14,640$ positions and the evaluation of $60^2 = 3,600$ positions. Predicting three full turns increases these numbers to $\sum_{l=1}^{3}(60^l \times 4) = 878,640$ and $60^3 = 216,000$ positions, respectively. This is more than an order of magnitude cheaper than predicting even one full turn without pruning! [6,7]

2.3.3 Seeking Short-Term Gains

Another pruning method was tested but eventually dropped. Depth searches were inhibited for positions whose evaluations on the current level were worse (or alternatively, no better) than the best result found in previous depth searches. However, this rather crude heuristic method missed too many good positions that led to long-term advantages despite short-term setbacks, effectively negating the whole purpose of the prediction tree.

6. The "Evaluate all masks" option disables this pruning method for a computer player.

7. These estimates are conservative averages. For a complex mid-game situation, I observed a computer player generate over 8 million positions while looking three full turns ahead. Such outliers are the reason why prediction tree depths default to only two full turns in the current *Star Chess* version.

Chapter 3

Implementation

This section describes the actual implementation of the algorithms outlined in chapter 2. The classes implementing the *Star Chess* game engine and computer players are located in the Java package `org.kynosarges.starchess.core`. We'll give a brief overview of the actions performed to generate a computer player's turn. Please consult the extensively commented source files for more detailed information.

3.1 Game State and Engine

One complete *Star Chess* game state or position equals one `Galaxy` instance. Each `Galaxy` holds the following objects which contain the actual game data:

- Four `Faction` instances, each holding a number of integer values tracking the possessions of the corresponding player, and a boolean flag indicating defeat.
- Sixteen `Sector` instances, each holding one `Fleet` instance and one `Planet` instance.
- `Fleet` tracks the owner, number, and availability of all ships in the sector. An absent fleet is represented as an empty `Fleet` instance, not a null pointer.
- `Planet` tracks the type, owner, population, and resource status of the sector's planet. Each sector always contains exactly one planet.

During each game, one single `Engine` instance holds one `Galaxy` instance to represent the current game state. Other notable `Engine` data include a `History` instance containing all `Command` objects executed so far, a `Settings` object for user preferences, and a `ViewAdapter` that represents the attached user interface.

The `ViewAdapter` interface enables total separation of game engine and display output. The `StarChess.Core` project comes with a simple console runner, useful for testing game mechanics and computer players. The JavaFX GUI is realized as an entirely separate project, and could be easily swapped out for a different GUI without affecting the engine.

Figure 3.1 shows `Galaxy` and related classes that constitute a game state, and Figure 3.2 shows `Engine` and related classes that control the game progress.

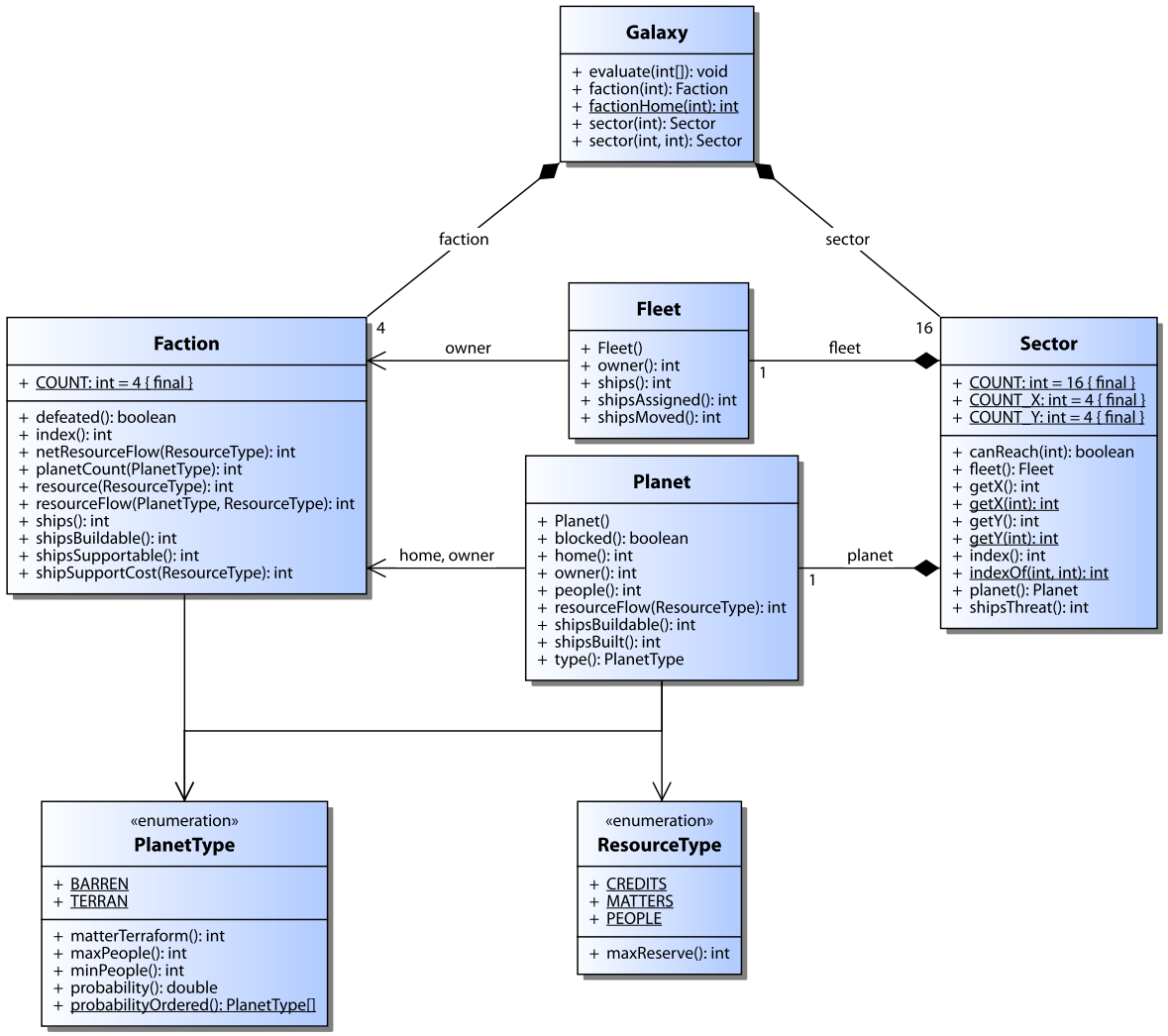

Figure 3.1: Galaxy Overview

3.1.1 Computer Player Operation

Computer players are implemented entirely separately (see section 3.5) as an alternative means to generate commands, at the discretion of the game client. The `Engine` does not know whether any given `Command` instance originated with a human or computer player.

Computer players work on a copy of the current `Galaxy`, so as to not disturb the `Engine` or the client display. The turn prediction algorithm also requires one additional copy for each level of the prediction tree. To allow such copies to be made safely and efficiently, we only use

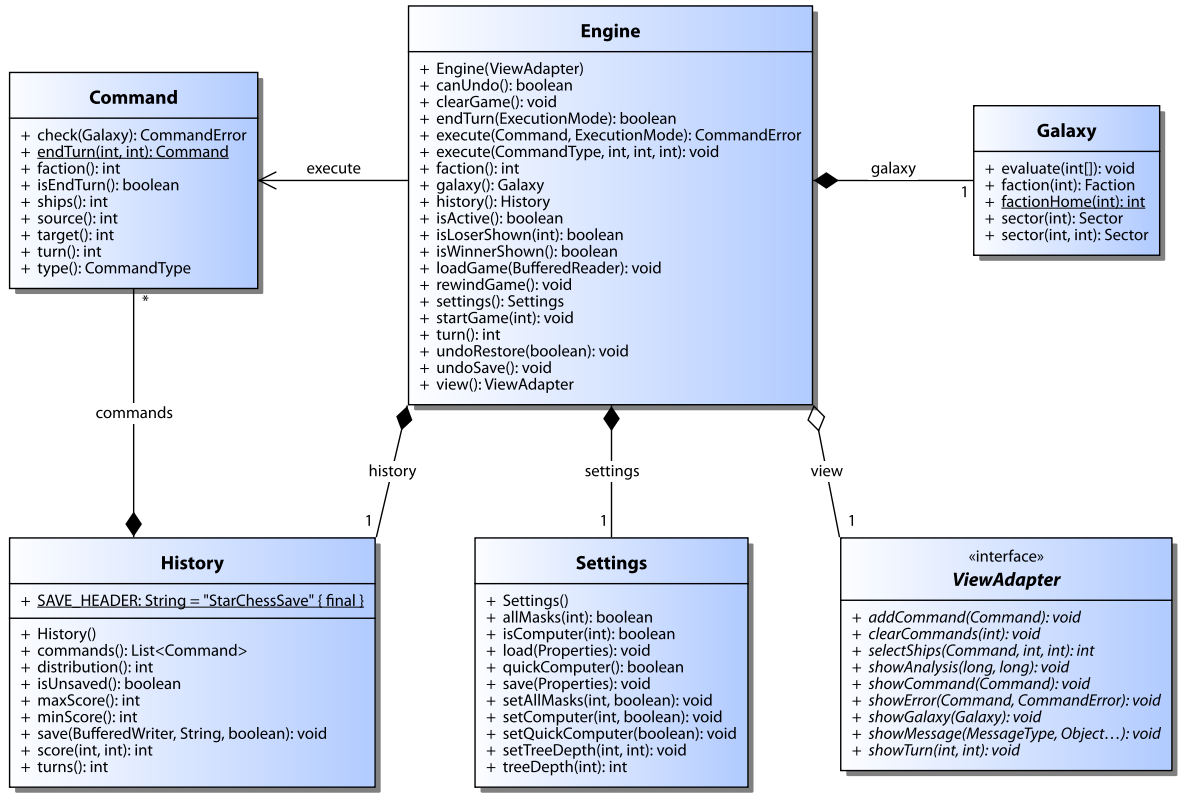

Figure 3.2: Engine Overview

a minimal number of Java object references within a game state. For example, the owner of a `Fleet` is stored as an index into the `Faction` array of the containing `Galaxy`, rather than as a direct reference. This requires an additional array lookup whenever the `Faction` object is required, but lets us perform a simple integer copy up front.

Moreover, the turn prediction algorithm preallocates `Galaxy` objects for all levels of the prediction tree. This enables copying the current game state into an existing permanent `Galaxy` instance, minimizing garbage collection activity during turn prediction. The `Galaxy` object tree has no components that require a new allocation to represent a different game state. Only enumeration and primitive values (mostly integers) need to be replaced.

Figure 3.3 shows `MaskPredictor` and related classes for computer player functionality.

3.2 Sector Processing Order

When generating a computer player's turn, we frequently have to traverse multiple sectors in order to find out how to allocate the available resources. The problem is that the *order* in which the sectors are traversed can *influence the allocation pattern.* If we have insufficient resources (as usual) to fulfill two requests of equal priority, then the available resources will be spent on

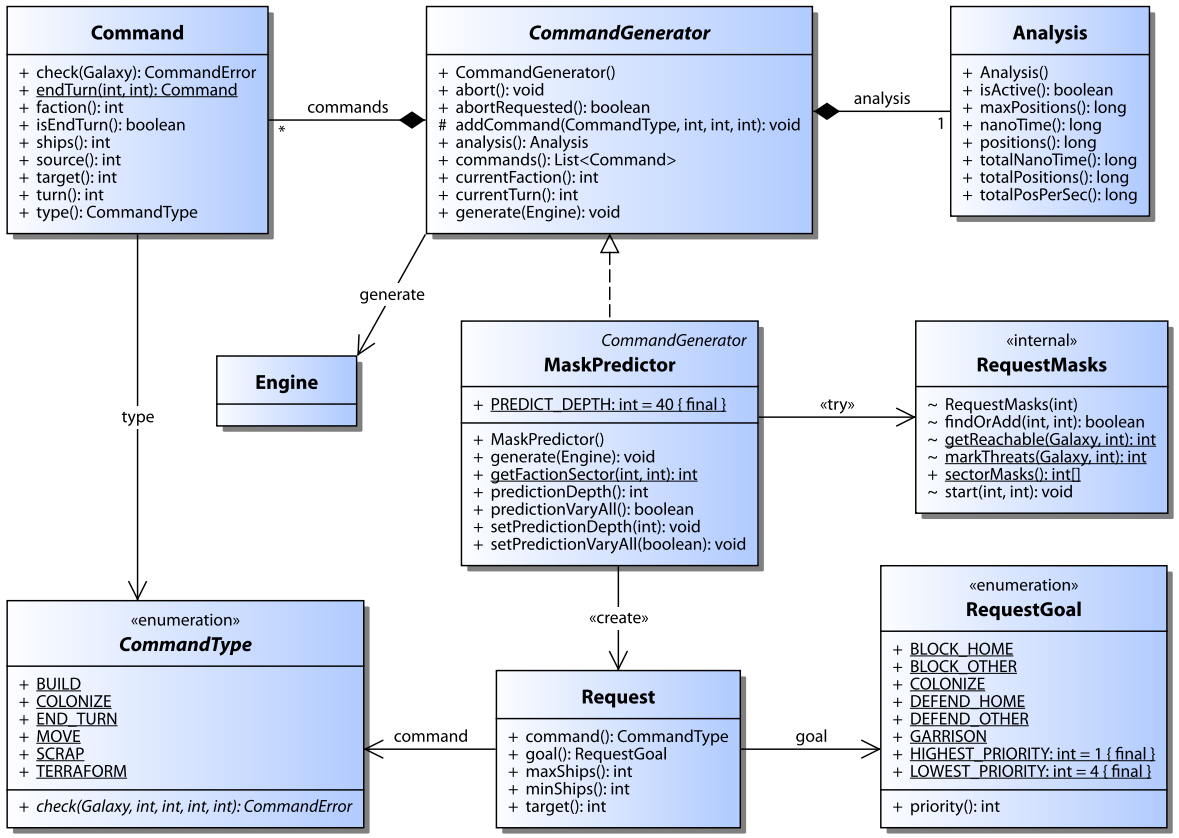

Figure 3.3: MaskPredictor Overview

the request whose sector we visit first while the other request will remain unfulfilled.[1]

The naïve sector processing orders[2] of A1–A4, B1–B4, C1–C4, D1–D4 or A1–D1, A2–D2, A3–D3, A4–D4 will therefore lead to a "race for the lower left corner" where all computer players, whenever given the chance, will allocate their resources towards sector A1 – even though the home worlds of players 2–4 are in the three other corners of the galaxy. This will make the game unfairly difficult for player 1 (whose home world is in sector A1) while leaving the other three players wide open to attacks on their respective home worlds.

The solution is to create an *individual sector processing order* for each player, starting with the player's home world sector and continuing outward from there. So method `MaskPre-`

1. Note that this problem is unique to games which allow multiple actions per player turn, and where certain equally desirable actions are mutually exclusive, e. g. due to lack of resources. Chess-like games which allow only one action per player turn, or hypothetical games where all possible actions are completely independent of each other, are not affected by this problem.

2. As in the *Star Chess* game itself, sector coordinates are given in a chess-like notation of *Xn* where *X* is a letter from A to D designating a board column from left to right, and *n* is a number from 1 to 4 designating a board row from bottom to top.

`dictor.createFactionOrders` creates the following processing orders for the four players:

1. A1, A2, B1, C1, B2, A3, A4, B3, C2, D1, D2, C3, B4, C4, D3, D4
2. A4, B4, A3, A2, B3, C4, D4, C3, B2, A1, B1, C2, D3, D2, C1, D1
3. D4, D3, C4, B4, C3, D2, D1, C2, B3, A4, A3, B2, C1, B1, A2, A1
4. D1, C1, D2, D3, C2, B1, A1, B2, C3, D4, C4, B3, A2, A3, B4, A4

Each pattern progresses from the player's home world to the opposite home world, alternating the direction in which each diagonal is taken. The processing pattern of player $n+1$ is identical to that of player n, rotated clockwise by 90 degrees. The result can be visualized as a "snake curve" across the board. These processing orders ensure that threats and opportunities close to the player's home world are dealt with before any resources are put into more remote endeavors.

3.3 Threat Count and Mask Generation

To create requests, we must know which sectors could possibly require our attention. To this end, method `RequestMasks.markThreats` sums up and stores the size of all enemy fleets that are either present in a sector, or are able to reach the sector in one turn.[3] If a sector has a friendly colony and a threat count of zero, it is considered *safe*.

The standard request mask is now determined to enable all possible requests. Requests are possible for all sectors that are reachable (i. e. within one sector of a friendly colony – see `RequestMasks.getReachable`) but *not* safe. So the standard request mask allows the generation of requests for all sectors that contain a threat or an opportunity.

3.4 Evaluation Function

The evaluation function `Galaxy.evaluate` calculates and returns a separate score for each of the four players. The scores shown on the screen are the same as those used by the computer players to evaluate a position – the higher the better. Scores are computed as follows:

1. Let P be the number of players, p with $1 \leq p \leq P$ a player index, $S(p)$ the number of ships and $C(p)$ the number of colonies owned by player p, respectively.
2. For all players p, compute a basic score of $B(p) = S(p) + 10\ C(p)$.
3. Let p_S be the player whose score we wish to compute. For all players $p \neq p_S$ with $B(p) > B(p_S)$, double the basic score: $B(p) \leftarrow 2\ B(p)$.

3. I also attempted to increase the threat count by the number of enemy ships that might be built in a sector, but this made the computer players too timid in practice.

4. Player p_S receives a final evaluation of

$$E(p_S) = (P-1)\,B(p_S) - \sum_{p=1}^{P} B(p), p \neq p_S$$

Although the evaluation function is most sensitive to changes in the possessions of the currently evaluated player p_S, there is an additional (inverse) influence by changes in the possessions of any other player. A stagnant empire's score will climb while several opponents are busy eliminating each other, but it will fall while other players are expanding their realms. This ensures that the computer players are neither complacent nor meddlesome.

Step 3 is an improvement to the original evaluation function, suggested by Martin Leslie Leuschen. Before computing the final evaluation, we double all basic scores that surpass that of the current player p_S. This makes the final evaluation more sensitive to changes in stronger players' possessions. As a result, the computer players prefer focusing on the strongest opponent rather than wasting forces trying to eradicate a weaker player, weakening themselves in the process. The change also causes the evaluation function to react in a nonlinear way to changes in the balance of power, which helps to avoid stalemates – situations where no computer player dares to make a move because all players are of nearly equal strength.[4]

3.5 Turn Generation

Computer players are exposed through the abstract base class `CommandGenerator` which manages input situation, output commands, and auxiliary features such as performance analysis. The concrete algorithm described in this document is implemented by class `MaskPredictor`. Entry method `generate` first evaluates the prediction tree to find the best request mask, and then uses that mask to generate the output command sequence.

3.5.1 Eliminated Players

One noteworthy point is the treatment of eliminated players. These are still included in the prediction tree, simply generating an empty command sequence for their tree levels. The older C version of *Star Chess* offered an optional macro `SKIP_DEAD` that would remove eliminated players from the prediction tree, filling their tree levels with live players instead. This deepened the tree in terms of *full turns* as the number of surviving players diminished, thus enabling the computer players to provide a stronger endgame.

Unfortunately, this option also caused a massive increase in complexity as weak and nearly inactive players who contributed little to the total position count were eventually replaced with strong and highly active players. As a result, the computer player algorithm was

4. Certain planet type distributions will still cause stalemates in early mid-game, however. *Star Chess* does not attempt to combine fleets from different sectors in order to match minimum request sizes, so small fleets may simply remain on garrison duty and uselessly consume support indefinitely.

slowed down to such an extent that the SKIP_DEAD option was disabled by default, and removed entirely in the current Java version.

3.5.2 Tree Prediction

The recursive method MaskPredictor.optimize builds the prediction tree proper, up to a depth of $l_{\max}$ = predictionDepth player turns, and returns the request mask that eventually generated the best evaluation for player $p(1)$ = currentFaction to the caller.

The standard mask fullMask is always constructed and evaluated. Alternative masks, called tryMask, disable one or two requests that were enabled by the standard mask. RequestMasks.findOrAdd ensures that we don't try identical masks repeatedly. Alternative masks are only examined if tree pruning is disabled in the *Preferences* dialog, i. e. predictionVaryAll is true, or on any level l with $p(l) = p(1)$, i. e. faction = currentFaction.

On each tree level, createAll builds and executes the Command sequence for the current level and request mask. optimize then recursively descends to the next tree level while above predictionDepth. On that lowest level, Galaxy.evaluate scores the game state. The best evaluation for $p(l)$ = faction is returned to the previous tree level, and eventually to the original caller, along with the request mask that produced this evaluation.

3.5.3 Command Generation

Method MaskPredictor.createAll builds the Command sequence corresponding to a request mask. For any given Galaxy and mask, this sequence and its effects are fully deterministic. After initializing the current tree level, createAll first calls createRequests which, surprisingly enough, create requests for all sectors enabled by the request mask.

The resulting Request list is then used to generate fleet movement, ship building, and colonization commands, in this order. Terraforming and additional construction or demolition of ships is then performed as determined by available resources, independently of the request list. Finally, Galaxy.turnFaction performs regular end-of-turn calculations for the active faction.

3.5.4 Request Creation

Method MaskPredictor.createRequests creates one request per Sector for all sectors that are enabled by the supplied request mask. All other sectors do not emit any requests. Each Request contains the following data:

— One out of four priorities, ranging from 1 (highest) to 4 (lowest). Priorities are assigned to possible goals in the following order:

1. Defend the home world colony against enemy ships in the same sector.
2. Defend other colonies against enemy ships in the same sector, or attack an enemy home world colony.

3. Attack enemy colonies other than home worlds.
4. Garrison friendly colonies threatened by enemy ships in adjacent sectors, or colonize an unpopulated planet (whose sector may contain enemy forces).

— One out of two commands: move ships or colonize planet. This determines whether a fleet should set up a colony in its current sector, or whether it is merely in transit.

— Minimal and maximal (ideal) "request size," i. e. the smallest and largest number of ships that should be sent to fulfill the request.

— Sector targeted by the request, in case a `Request` instance is passed to a method that does not know its originating `Sector`.

In general, the minimal request size is set to the number of enemy ships threatening the request sector (`Sector.shipsThreat`) while the maximal size is set to $3 \times$ `shipsThreat`. This assures that the "minimal fleet" can defeat all enemy ships in and around a sector, although it might itself be destroyed in the process, while the "maximal fleet" can defeat all enemies without suffering any losses. Request sizes are then adjusted depending on the specific action taken.

— Defend friendly colony: due to the importance of this task, the minimal fleet size is lowered to $\texttt{shipsThreat}/3 + 1$, i. e. the fleet size required to destroy at least one enemy ship, no matter what the costs in terms of friendly ships.

— Garrison friendly colony: the minimal fleet size is once again lowered to $\texttt{shipsThreat}/3 + 1$, this time to avoid putting too many ships on unproductive garrison duty.

— Attack enemy colony: minimal fleet size is $\texttt{shipsThreat} + 1$ so that at least one blockade ship survives combat, and maximal fleet size is raised to at least 50 ships for optimal bombardment effect.

— Colonize unpopulated planet: minimal and maximal fleet sizes are both increased by the number of ships required to set up a colony.

Once all requests have been generated, they are sorted into a request list by priority. Within the same priority, requests are sorted by the current player's sector processing order. All methods described in the next section operate on this request list.

3.5.5 Request-Driven Actions

Move Ships. Methods `moveToRequest` and `moveToSector` are fairly complex, so we defer discussion of ship movement to section 3.6.

Build Ships. Method `buildToRequest` attempts to build enough ships to fulfill all requests that were not satisfied by `moveToRequest`. Requests are processed in the order determined by the sorted request list, i. e. by priority and by sector processing order.

On blockaded colonies, ships are built regardless of the current player's support capacity because the newly built ships either won't survive combat against the blockade fleet anyway, or else the (now liberated) colony will resume its resource production, rendering our previous support calculations obsolete. Other requests only cause as many ships to be built as can be supported by current resource income.

Colonize. Method `colonize` attempts to fulfill colonization requests. If a fleet with a sufficient number of active ships is located in a sector with a colonization request *and* the total fleet size meets the minimal request size, 20 ships are expended to set up a new colony.

It is important that a new colony is only set up if the minimal request size is met, or else some of the enemy ships that figured into `shipsThreat`, the basis for calculating minimal request sizes, would eradicate the fledgling colony in short order! Observing the minimal request size ensures that enough friendly ships are present to defend the colony against such threats.

3.5.6 Resource-Driven Actions

The final part of a computer player's turn consists of several actions that are not directly caused by requests. Their execution does, however, depend on the situation created by the previous movement, shipbuilding, and colonization actions which in turn *were* caused by requests.

Terraform. If the current `Faction` owns at least two barren colonies, at most twice as many terran colonies as barren ones, and can afford the 1, 000 materials required to terraform a barren planet, method `terraform` transforms the most populous unblockaded barren colony into a terran world. As a precaution against toppling the delicate balance of planet classes, only one planet is terraformed at a time.

Build Ships. If the current support capacity exceeds the current number of ships, method `buildToSupport` builds additional ships "just in case," up to the limit set by the support capacity. Ships are always built on the most productive unblockaded planets, sought in the player's sector processing order.

Scrap Ships. Conversely, if the current number of ships exceeds the current support capacity, method `scrapToSupport` scraps ships that are not currently assigned to any request, bringing the total number of ships down to the limit set by the support capacity. Unassigned ships are sought in the player's sector processing order. If there are not enough unassigned ships to scrap, the remaining number of unsupported ships will be taken care of by the automatic demolition routine during end-of-turn calculations.

3.6 Ship Movement

Movement processing is split into method `moveToRequest` which determines the best possible fleet movement, and method `moveToSector` which attempts to perform that movement.

3.6.1 Matching Fleets to Requests

First, all fleets receive their own version of the request list. For each fleet, the request list, already sorted by priority and by sector processing order, is once again sorted by the fleet's distance[5] to each request, but only within the same request priority. Next, all fleets are sorted by size into a fleet list. Fleets of the same size are sorted by sector processing order.

So prepared, we can now assign fleets to requests. We process fleets in the order of the fleet list we just generated, i. e. largest fleets first. Each fleet looks at requests in the order of its individual request list, i. e. closest and most important requests first.

If the number of ships in a fleet that can fulfill a request (either all present ships, or present and active ships only) is too small to satisfy the minimal request size, the request is skipped and the next request is examined. Otherwise either the entire fleet, or a partial fleet large enough to meet the maximal request size, is assigned to the current request and immediately executes the requested action. If unassigned ships are left in the current fleet they continue to look for unfulfilled requests; otherwise the next fleet is examined.

Whenever a fleet is assigned to a request, the request's minimal and maximal fleet size are reduced by the number of ships assigned. This means that a small fleet that could not have met a minimal request size by itself may still be used to supplement a larger fleet which previously dropped the minimal request size to zero. Supplementary fleets are assigned until the maximal request size has been reduced to zero.

Special Considerations. If an enemy fleet is present in the target sector of a request with a minimal fleet size of zero, we must consider the possibility that the larger fleet previously assigned to this request has not yet arrived. In this case a smaller supplementary fleet would promptly commit suicide against an overwhelming enemy! Therefore fleet sizes are always checked against the number of enemy ships in the target sector, even if the minimal request size has already dropped to zero.

Another point concerns the divergence of theory and practice. In theory, small partial fleets "left over" in the request assignment process should re-examine their request list once all other fleets had a chance to minimally fulfill a request. However, in practice it is preferable to allow partial, unassigned fleets to serve as garrisons wherever they are, rather than sending them back and forth on dangerous trips across the galaxy.

5. On the chess-like board of *Star Chess,* diagonal movements of n squares are no costlier than horizontal or vertical movements of n squares. This means that the distance between two squares (or sectors) is either their horizontal or their vertical distance, whichever is greater.

3.6.2 Moving Towards a Target

Method `moveToRequest` eventually calls method `moveToSector` to move a (full or partial) fleet one step towards its target sector. If the target sector is adjacent to the fleet's position, we simply execute the standard movement command without further considerations.

Things get more difficult if the distance to the target is greater than one sector. In the chess-like movement system of *Star Chess,* there are 2–3 alternative first steps that result in routes of equal length for any movement of two or more sectors, except if the line connecting source and target sector is strictly diagonal. We exploit this fact to evade enemy fleets that block one of our possible "first step" sectors and choose another alternative instead, provided that the fleet is allowed to move into that sector. If no unblocked legal "first step" sector can be found, the movement request is ignored and the fleet stays where it is.

The rationale is that if the fleet was supposed to fight for a certain sector, its request would have sent it right there rather than somewhere else. So we conclude that we are supposed to evade enemy fleets in any sector other than the target sector. This complication does not apply to adjacent targets, as we can assume that any combat is intentional.

Appendix A

Star Chess Rules

This appendix presents, as concisely as possible, the rules of the *Star Chess* game. Please see the help system for details and explanations of these rules. The chess equivalents of certain *Star Chess* terms are noted where applicable.

1. Players
   - — There are four players.
   - — Players take turns in a fixed sequence.
   - — Four player turns (chess: half turns, "plies") constitute one full turn.
   - — A player turn consists of zero or more commands given by the player. Commands include Terraform, Colonize, Build Ships, Move Ships, Scrap Ships.
   - — Each player starts with 20 ships and one terran colony. The planet hosting this colony is called the player's home world.
   - — A player who loses his home world colony is defeated, and all of his possessions are immediately eliminated from the game.
   - — When three players have been defeated the remaining player wins.
2. Resources
   - — Each player holds a reserve of two resources: materials and credits.
   - — Initially all reserves are zero. They may never drop below zero.
   - — Resources are produced by colonies and consumed by colonies and ships.
   - — At the end of a player's turn, the player's reserves are increased by colony production and decreased by colony consumption and fleet support.
   - — If ship support costs would drive a reserve below zero, ships are automatically scrapped in a sufficient number to avoid this.

— Unaffordable colony consumption causes any existing ships to be scrapped but has otherwise no ill effects on the colony or the player.

3. Sectors
   — The galaxy (chess: board) consists of 4×4 sectors (chess: squares).
   — Every sector has exactly one planet.
   — Every sector may host 1 to 999 ships (chess: pieces) owned by one player.
   — The ships in one sector are collectively referred to as a fleet.
4. Planets
   — Every planet falls into one of two classes: terran or barren.
   — Every planet may host one colony owned by one player.
   — **Terraform:** A player who owns a colony on a barren planet may change the planet class to terran for a price of 1,000 materials.
   — If a colony on a terran planet is lost to bombardment, the planet class changes to barren.
5. Colonies
   — **Colonize:** A player may exchange 20 ships for a new colony on an empty planet in the same sector. The ships must not have moved in the same turn.
   — All colonies start out with 20 inhabitants.
   — A colony that drops below 20 inhabitants due to bombardment is eliminated.
   — Colony parameters change with the class of the hosting planet.
   — Barren colonies produce 5 materials (maximum 1,000) and consume 1 credit (maximum 200) per inhabitant and turn. They grow to a maximum population of 200 inhabitants, at a rate of 10% per turn.
   — Terran colonies produce 1 material (maximum 200) and 1 credit (maximum 1,000) per inhabitant and turn. They grow to a maximum population of 1,000 inhabitants, at a rate of 20% per turn.
   — **Build Ships:** A colony may build ships in its sector at a cost of 20 materials and 40 credits per ship. In addition, each ship requires a workforce of 10 local colonists who have not worked on another ship in the same turn.
6. Ships
   — **Move Ships:** A player may move all or part of the ships in a sector to an adjacent sector that either contains a friendly colony or is adjacent to a sector with a friendly colony.

— Ships that end their owner's turn more than one sector away from any friendly colony (because all nearer colonies were destroyed by bombardment) are considered out of support and automatically scrapped.

— Ships that have been built in a given turn cannot move in the same turn.

— Ships that have moved in a given turn cannot move again or colonize a planet in the same turn.

— Ships moving into a sector with enemy ships automatically commence combat. One or both player's fleets are completely destroyed in the combat.

— Ships ending their turn in a sector with an enemy colony automatically blockade and bombard the colony.

— Blockades stop all local resource production and consumption.

— Bombardments reduce the colony population by a percentage value equal to the number of bombarding ships, up to a maximum of 50%.

— Every ship consumes 10 materials and 20 credits per turn as support.

— If a lack of reserves causes a number of ships to be scrapped, the number is distributed among all the player's fleets so that the size of each fleet is reduced by approximately the same percentage value.

— **Scrap Ships:** A player may manually scrap ships to avoid support shortages.

7. Combat

— Combat results depend on the relative sizes $S(F_1), S(F_2)$ of the two fleets.

— Two fleets of equal size completely annihilate each other.

— In a combat of fleets of different size, the larger fleet F_L always wins and the smaller fleet F_S is always annihilated. Losses are calculated as follows:

- If $S(F_L) < 2\ S(F_S)$ then F_L loses $S(F_S)$ ships.
- If $2\ S(F_S) \leq S(F_L) < 3\ S(F_S)$ then F_L loses $S(F_S)/2$ ships.
- If $S(F_L) \geq 3\ S(F_S)$ then F_L suffers no losses at all.

Appendix B

Notable Changes

This appendix notes some important changes between the 2001 C version and the 2014 Java version of *Star Chess*. Please see the `WhatsNew` file for a complete version history.

B.1 File Management

Star Chess no longer ships with predefined save games for galaxies with only barren or only terran planets (`StartBarren.SCA`, `StartTerran.SCA`). Instead, you can directly select these galaxy types from the "New Game" dialog.

Moreover, the format of the settings file (`StarChess.ini`) has changed and now uses the serialization provided by `java.util.Properties`. All these files are no longer stored in the *Star Chess* installation directory, either, but rather in a separate user-specific folder. Please see the `ReadMe` file for details.

B.2 Computer Players

Setting the depth of a computer player's prediction tree to zero would cause *Star Chess* 1.2 to generate invalid commands. These would be initially executed (`?`) without error, but attempting to load saved games containing such commands would abort with an error message.

Tree depth zero should now work correctly, although it is still prone to stalemates where all computer players have exhausted their ship support and simply stare at each other, none able to move without losing a colony.

To make such stalemates less likely in general, garrison priority dropped from 3 to 4, the same as colonization. The new algorithm will also sometimes choose differently between equally valued options than the old one, resulting in divergent game states from that point onward. I haven't bothered to analyze the C code to find out exactly why this is the case.

The `SKIP_DEAD` option was removed entirely. It was disabled by default and not accessible through the user interface, and it complicated the turn prediction algorithm.

B.3 Source Code

The state of a planet (normal or blockaded) used to be implemented as an enumeration, but is now simply a boolean flag to indicate blockades. I could not think of any third value that made sense, so the enumeration was pointless.

Players have been internally renamed to factions, so as to distinguish them from the (human or computer) player that controls them. The user interface and help system still refer to all sides as "players," though.

The help system itself is based on the same HTML pages as before. However, they now ship as loose files rather than a compiled Microsoft HTML Help package. I use a JavaFX `WebView` to show them in a simple frame set.

Bibliography

[1] Charles R. Dyer and Jim Gast, *CS 540 Lecture Notes: Game Playing*. University of Wisconsin, Madison, USA, 23 June 1999. — This lecture describes minimax prediction trees and alpha-beta-pruning for chess-like games.

[2] Tony A. Marsland, *The Anatomy of Chess Programs*. University of Alberta, Edmonton, Canada. — This overview describes implementations and refinements of the basic algorithm explained in Dyer and Gast. A short bibliography is also included.

[3] Aske Plaat, *MTD(f): A Minimax Algorithm faster than NegaScout*. Vrije Universiteit Amsterdam, The Netherlands. — I make no use of the refined chess algorithm presented here, but this page contains valuable links to other sites discussing chess algorithms. Moreover, Plaat's *MTD(f)* algorithm nicely demonstrates the level of sophistication reached by contemporary chess research.

Original Sources. As of June 2012, the documents listed above have all vanished from their original Internet addresses. The following list shows the sources as I found them in 1999.

1. http://www.cs.wisc.edu/~jgast/cs540/notes/games.html
2. http://www.cs.ualberta.ca/~tony/ICCA/anatomy.html
3. http://www.cs.vu.nl/~aske/mtdf.html