ABSTRACT

This project modifies the endgame of a basic minimax checkers game by utilizing operant conditioning principals developed in the field of Psychology. In general terms, this project involved the design and implementation of a program that causes the computer's play to gradually improve over time by storing successful moves and accessing them in later games.

The operant conditioning principals of first-order association using positive and negative reinforcement, is applied to each stored board every time it is referenced by incrementing a rating counter field for those moves that participate in a successful game. Likewise, the counter field is decremented for those moves that participate in a loss. This operant conditioning practice extinguishes stored moves that prove to be unsuccessful over time and reinforces the existence of successful moves.

CONTENTS

INTRODUCTION

One criteria used to determine the existence of intelligent behavior is the ability to learn. For years, Psychologists have studied human learning, developing and performing numerous theories and experiments in this area. When programmers began their investigation into artificial intelligence one area that proved efficacious was machine learning. In 1959 Arthur L. Samuel published a paper that described a computer checkers game that had the ability to improve its performance over time. His efforts were focused on continuous honing of the heuristic with repeated play and advancing the search depth by referencing stored moves.[Samuel 1959] His latter efforts included book learning; adjusting the computer's game so that it would select the same move chosen by experts in recorded games.[Samuel 1967]

The purpose of this project was to develope a new machine learning technique for checkers. However, the underlying logic used to develope this new technique did not come from the traditional methods developed in artificial intelligence. Instead, the new concept was derived from studies of human learning as described by psychologists.

First it is necessary to discuss the components of a standard computer checkers game as well as critique its performance. In search of the best move, the computer applies a minimax search of the game tree to a fixed depth and applies a heuristic function to each frontier node.[Shannon 1950] Alpha-beta pruning is used to reduce the search time[Hart and Edwards 1963] and node ordering is used to expedite the alpha-beta pruning.[Kort 1988] The heuristic is composed of a few rudimentary evaluations and remains static throughout the entire game.

This standard algorithm plays an adequate opening and middle game, depending on the heuristic used and depth of search. However, once the computer reaches endgame it begins to falter, more often losing than any other outcome. (Endgame for this project is obtained when either player has three or fewer checkers remaining.) The heuristic

evidently lacks the proper emphasis to accurately terminate the opponent. It seemed obvious then, that the most prudent move would be to teach the computer to play a better endgame.

Next, it is important to describe those components of human learning research that were used to teach the computer to play a better endgame. The psychological theories and principals for this project were derived from first order operant conditioning. In operant conditioning a response is followed by a positive reinforcer such as a reward, or a negative reinforcer like an aversive stimulus.[Hill 1971] Thus, in operant conditioning, where a response is followed by a reward, there is a tendency for the response to be repeated. Each time this sequence is repeated, the probability is increased that response will be repeated.[Klausmeier 1971] The opposite holds true for the application of aversive stimuli, each application of the negative reinforcer decreases the likelihood that the behavior will be repeated.

In this project, a 100% schedule of positive reinforcement was used to reward proper responses and a 100% schedule of aversive stimuli was presented for inappropriate responses. Operant theory dictates that the continued rehearsal with contiguity of stimuli would gradually shape the subject's performance into the desired paradigm.

In order to implement these operant conditioning principals the project saves the moves used by both teams during endgame play. When the game is over, the program transfers the winning team's moves into the move library. In future games, once endgame has been reached, the move library is reference by the team attempting to make its next move. If this team is able to locate its exact situation in the move library, it copies its next move from the move library, regardless of this new moves heuristic value. At the end of the game every move that participated in the win is placed in the move library with its performance bit set at 1. If the move already exist in the move library its performance bit is simply incremented by one. In like manner, all the moves that participated in the losing teams effort are searched for in the move library. If they are located, their performance bit is decreased by one. Any move who's performance bit reaches negative one is eliminated from the move library.

At this point it is important to discuss the difference between this project and earlier work done by Arthur L. Samuel. In Samuel's first paper, he describes storing thousands of moves on tape and then having the program reference these moves throughout the game. In this aspect the two programs are similar. However, the use of the boards by the respective programs is quite different. Samuel's game stored moves throughout the game, not just in endgame. Also Samuel's game stored all boards without regarding the board's

participation in a winning team's effort or a losing team's effort. In Samuel's game the list of boards were referenced in the minimax phase of the game once the program had reached its maximum search depth. If a board, identical to the one that was reached at maximum search depth was located, it's stored associated heuristic score was used instead, effectively doubling the search depth.[Samuel 1959] In my project, the search through the move library is performed before minimax is performed. If a move is located, regardless of it's heuristic value, it is implemented Thus assuming it is a good move because it has already participated in a win.

Another difference in the treatment of the boards can be found in how they are culled. In order to limit the number of boards saved, Samuel's program used two techniques. The first technique involved maintaining an age variable associated with each board. Every time a board was accessed it's age variable was refreshed. If a board reached an arbitrary maximum age it was expunged from the record. Samuel labeled this a form of forgetting. The second technique he used was to simply lop off the lowest-ply moves when the maximum record size had been reached.[Samuel 1959] In contrast, my program only forgets a move once it has participated in more losing engagements than winning engagements proving that it is not a worthy move.

Finally a discussion on the merits of this project. Extensive amounts of time and energy have been spent examining human learning. As computer languages become more abstract it becomes easier to program a concept rather than specific bits and bytes. The next logical step is to apply those principals that have been developed in human learning studies to a computer program in order to extract the desired advanced learned behavior from the computer. This project is an attempt to do just that.

DETAILS OF DESIGN

The actual code for this project was influenced by two previous programs. The first influence was a checkers game written in Basic by John Krutch. Features imitated from Krutch's program included a fundamental evaluation function, and the representation of the checker board using a two dimensional array. The second program that influenced this project was a program I wrote that was a much simpler form of this project. Features imitated from this program include minimax, the existence of kings, unlimited jumping capability, the acceptance of moves from a human player and an enhanced graphic representation of the checkers and board. This project is programmed for the IBM PC, XT, PS/2 family and compatible in the language Basic. The program is compiled using Microsoft Basic Professional Version 7.0. The code for this project breaks down into two major areas, User Interface and Computer Move.

USER INTERFACE

The main reason that this program was written in the language Basic is because the program relies heavily on video graphics to interface with the user. Basic is not highly regarded as an ardent choice for series programing. However, with the refinements engineered into the latest versions of Basic, in the areas of program structure and the ability to compile the completed code, it becomes a minor goal of this project to evaluate the worthiness of Basic to be used in a project such as this.

Throughout the game the user is prompted for information or presented with continually updated visual information. This next section is a breakdown of all of the information that is presented to the user during pregame, during the game, and after the completion of the game.

In the pregame phase of the game the user is presented with a number of screens controlled by the procedure Gameselect.[Page# A-2 & B-12]* The first screen the user views is a menu requesting the user to define their machine's video capabilities. This allows a user to take advantage of advanced VGA graphics without limiting other users. The next screen the user is presented with allows the user to determine if a human player will control either or both of the teams or if the computer will control either or both of the teams. If the user specifies that one or both of the teams are to be controlled by the computer, the next screen prompts the user for three playing characteristic that the user has control over. The characteristic are search depth, choice of available heuristic, and access to the move library. The next screen asks if a new game is to be played or if an existing game is to be restarted. If the choice is for a game to be recalled, the name of that game is requested. The final pregame screen presented asks the user if they would like the screen to remain blank or for the screen to be refreshed after each move. Leaving the screen blank speeds computer play and saves wear on the monitor.

Once the contest begins a number of procedures assign information to the screen, keeping the user updated. The first procedure to do this is Drawboard.[Page# A-4 & B-7] It is Drawboard's function to call all of the procedures necessary to present the viewer with a graphic presentation of the current board on the screen. Drawboard is called every time a new move is executed. The second procedure to assign information to the screen is Initplayer[Page# A-5 & B-21] called from Playermove.[Page# A-5 & B-41] Initplayer places a menu beside the checkerboard on the screen. The menu instructs the human player to enter a move or call up a help menu. If the help menu is called for, the procedure Menu[Page# A-5 & B-29] presents the user with options like save current game, restore old game, continue current game, and terminate current game.

If the user has selected one or both teams to be controlled by the computer, the procedure Inform[Page# A-6 & B-20] acquaints the user with the current phase that the computer is executing. Inform's choices are testing for endgame, searching the move library, locating a move in the move library, testing for a jump, and executing minimax.

---

*[Page# ...] Denotes the location of this procedure in appendix A and appendix B.

Three other pieces of information are presented to the user during the game. The first is a sign that indicates whose turn it is, updated from the procedure Inform[Page# A-6 & B-20] and the procedure Initplayer.[Page# A-5 & B-21] The second and third signs indicate how many moves have been completed and the amount of time that has elapsed so far in the contest. Both of these indicators are generated in the procedure Movetimer.[Page# A-8 & B-37]

At any time during the game the user can strike the F1 key and create a user interrupt. When this key is hit, the procedure Hotkeysub[Page# A-10 & B-19] presents a menu to the user which allows the user to blank or unblank the screen, terminate play, or declare the game a draw. This button is especially useful for computer vs computer games.

The last group of visual information is presented to the user after the game is finished. This is done by the procedures Displaywinner[Page# A-8 & B-6] and Updatestorage.[Page# A-9 & B-52] Displaywinner informs the user which team won and how many moves found in the move library participated in the win. Updatestorage indicates to the user that the boards saved in the past game are being placed in the move library and are being stored on disk.

COMPUTER MOVE

When the computer is asked to generate a move it goes through three phases. First it searches the move library for a possible match and solution to its current situation. Second the computer program tests for a possible jump. Third, the computer finally is forced to generate a move on its own. If all three of these methods fail to produce a move, the computer is forced to concede and the other team is declared the winner.

When the computer is forced into phase three, generate its own move, it uses a minimax search. The specific details of this search are covered later in this paper. However, for those readers that would like a more detailed description of the basic elements of minimax, appendix C contains a technical description of the fundamental ideology that makes up minimax as well as an example search tree employing alpha-beta pruning and node ordering.

The following sections are a description of the three phases of the computer move. They are broken down into the section headings Move Library, Test For Jump, and Minimax. A fourth section, Heuristic gives a description of the board-evaluating procedure that is called by minimax.

MOVE LIBRARY

The move library represents the heart of this project. It is the storage and access of these moves that make this project unique. As described earlier, the only way a move can be placed in storage is for it to participate in a winning game. Once a move is placed in storage it can participate in future wins and have its performance bit incremented, thus strengthening its chances for surviving; it can participate in losing engagements and have its performance bit diminished, possibly diminished into extinction; or it can remain forever unaccessed.

As more winning boards are stored the computer player gains more experience. The computer is not limited to its own computer generated moves. Every time the game is played, by either humans or by the computer, the winning team's moves are stored. Thus, the computer player can gain just by observing two human checker experts engaged in a game being played on the computer. More likely past contests described in literature can be reenacted on the computer with the results being the same.

Like the role played by the user interface procedures, the move library procedures are accessed before, during, and after the contest. After the computer has prompted the player with all of the pregame questions, the procedure Loadarray[Page# A-3 & B-25] loads all of the moves in the move library, currently residing on disk, into main memory. All throughout the endgame it is these moves that are referenced in the procedure Lookforboard.[Page# A-6 & B-51] When Lookforboard is requested to find a move it first calls the procedure Hash[Page# A-6 & B-51] which scans the current board and generates a nine digit number that identifies the board. The factors used to identify the board are the number of kings, the number of checkers remaining on each team, and the position of those remaining checkers. The procedure Search[Page# A-6 & B-46] is then called and goes looking for a match to this nine digit key. Upon locating this key it does a square for square comparison to verify that the two

boards are identical. At this point the next move, stored with the current board in memory is loaded up as the computers next move.

Throughout endgame all of the moves are being saved for both teams. Whether it is a human player or a computer player the first procedure called is Savemove[Page# A-5, A-6 & B-42] which stores the current board into memory. After the next move has been generated and before control has been passed on to the next team, Savenextmove[Page# A-5,A-6 & B-45] is called which saves this newly generated move and links it with the current board saved at the beginning of this move.

Finally it is these boards that have been temporarily stored in memory that are reviewed at the end of the game. Once a winner has been determined the procedure Updatestorage[Page# A-9 & B-52] takes the same steps, first getting an identifying key by calling Hash2[Page# A-9 & B-16] and second, searching for these moves in the move library using the procedure Search2.[Page# A-9 & B-47] Winning boards located are incremented. Losing boards located are decremented and possibly removed. And first time winning boards are placed into the move library with their performance bits set at one.

TEST FOR JUMP

Testforjump[Page# A-6 & B-49] is the procedure called by Computermove to verify that there are no jumps available before the procedure Minimax is called to generate a new move. If in fact a jump is available, then, following the rules of checkers, Testforjump executes the jump by calling the procedure Mustjump.[Page# A-6 & B-38] Testforjump handles all double, triple, and multiple jumps through a recursive process. Testforjump first advances the current position to the new position just derived from the latest jump and then calls itself recursively, thus generating a search for the next jump. If two or more jumps are available then Mustjump executes all jumps and then calls the procedure Heuristic[Page# A-6 & B-1?] to evaluate which jump leaves the team in the best position. Once a jump has been chosen the minimax procedure is skipped all together and Computermove passes along the move generated in Testforjump.

MINIMAX

The procedure Minimax[Page# A-6, A-7 & B-31] is the move generator. Once the procedures Lookforboard and Testforjump have failed to produce the next move it is up to Minimax to produce the best possible move available. Minimax begins its work by calling the procedure Generatemove.[Page# A-7 & B-13] Generatemove scans the current board in a fashion that generates the best moves first, this is called node ordering. As soon as Generatemove locates a friendly checker it calls Movegenerator[Page# A-7 & B-35] with the checker's coordinates. Movegenerator scans for a possible move or jump and executes that move if it finds one. If no move is available control returns to Generatemove which continues to locate friendly checkers until Movegenerator indicates a move has been made. Control is returned to Minimax which in turn calls itself recursively until the desired search depth has been reached.

Once the desired search depth has been reached Minimax calls the procedure Heuristic[Page# A-7 & B-17] which returns a numeric evaluation of the board. Minimax then checks to see if the value returned by the heuristic is greater than or less than, depending on the depth at which the Heuristic is applied, the alphabeta pruning value. If the value indicates a prune, further exploration of this branch is discontinued. If the value does not indicate pruning is necessary, this new value is compared with existing values to determine if it is the best move allowed. If this is the case the move is stored and the search is continued, otherwise the move is simply discarded. As the best moves are transversed up the search tree, it is these few select moves that are assigned as the best moves and finally the one best move that survives the last comparison made at the top of the search tree. It is this best move that Minimax returns as the next move to be made.

The entire Minimax procedure including alpha-beta pruning was first written in a simplified form. Once this simple algorithm executed properly, it was repeatedly tested using example search trees found in Samuel's work and example search trees found in texts written by Winston and Nilsson. As soon as the algorithm executed all of these example search trees flawlessly, the algorithm was implemented into the program. Further testing and modifications were then made so that it functioned properly in this new environment.

HEURISTIC

The procedure Heuristic[Page# A-6, A-7 & B-17] is called by both Testforjump[Page# A-6 & B-49] and Minimax[Page# A-7 & B-17] and is responsible for assigning a numerical value that represents the value of each board to the team currently looking for a move. There is an endless combination of aspects that can be scored during a checkers game and a number of phases that are passed through as a game progresses. An aspect, like center control, might be important at one point in the game and actually harmful at a different point in the same game. One of the most difficult aspect in the creation of a heuristic is to encode a numerical representation of a move or series of moves that a checkers player would normally only execute in a rare or unique situation. However, no matter how difficult it is to perfect, it is the heuristic in the end that will dictate how well the game is be played.

This project has approached the task of creating a heuristic from two sides. The first approach is to create a heuristic with only four or five variables in its equation. Some examples of variables would be piece count, center control, reward for advance, and a reward for keeping a few checkers at home to defend. The second method attempted was an imitation of Samuel's method of combining the simple terms into nonlinear equations. An example of this would be to reward a team if all three variables like center control, move options and denial of opponents moves, all were true. In both cases the project applied the heuristic only once at the bottom of the search depth regardless of which team has just moved or was about to move.

CONCLUSION

The conclusion is divided into three section. The first section describes and reviews the tests performed to evaluate this program and the results they obtained. The second section evaluates how well this program accomplished its stated goal. And the third section makes recommendations for enhanced performance for future projects.

TEST RESULTS

Testing for this project was accomplished by repeatedly playing the computer program against itself and human opponents and recording the outcome of every tenth game. The Heuristic procedure was continually varied in a quest for a better game, but on every tenth game the computer was pitted against itself using an identical heuristic procedure that had not been previously used by either side. In every tenth recorded contests, one team was allowed to access the move library and the other team was not.

In the first thirty contests that the project participated in, the computer was pitted against itself on twenty seven occasions and was pitted against a human opponent on three occasions. At no time did the computer observe a human vs human contest. After thirty games, the move library contained less than two hundreds boards. The only time the move library was accessed was when the same two heuristic were matched in two consecutive

games and produced the same board situations upon entering endgame. At this point the Lookforboard procedure took over and executed the referenced moves for the winning team in under five seconds per move, exactly as they had been saved in the previous game. On the tenth, twentieth and thirtieth contests the team allowed to access the move library was unable to locate any similar moves, and in all three cases lost the contest.

Finally in the thirty fifth contest a move was accessed by the winning team. By the fortieth contest there were two hundred and fifty stored boards. However, the computer that had access to these boards did not find any matches but subsequently went on to win the contest. Between the thirty fifth and fiftieth contests a few of the winning teams accessed one or two stored boards. By the end of the fiftieth game the computer had contested itself forty four times and had a human opponent six times. The results of the fiftieth game were the same as the previous attempt with zero boards being accessed and the team with access to the boards losing the contest. At the time of the fiftieth contest there were three hundred and sixteen moves in the move library.

EVALUATION

In Arthur Samuel's 1959 article his purpose was to prove that a computer could learn. He proposed that if a computer could improve its play in the game of checkers over time, this would indicate that the computer was learning. He describes the computer's playing abilities in the first seven games as "extremely poor." During the next seven games his rating went up to "rather bad." In games fifteen through twenty one, the program was evaluated as "tricky but beatable." And in the next seven games it approached a quality of play described as "a better-than-average player." In twenty eight games Samuel had proved his theory and the computer was beating opponents left and right.

In this project the computers level of play started out just above "rather bad" but still below "tricky but beatable". After fifty games the heuristic had been sharpened to a level of play the fell into the "tricky but beatable" category. But in the first fifty games this improvement had very little to do with the boards being stored, instead it was do to the reshaping of the heuristic.

After observing these fifty games it has become obvious that the currently stored boards will do little to improve the computer's future play. This is because ninety percent of the boards were generated in computer vs computer games. If the computer continues to play itself, the moves held in storage will be the exact same moves the heuristic will generate itself when placed in the exact same situation it was in when it stored the board in the first place. The only improvement these boards will give to the game will be the quicker execution of the same move the heuristic would generate anyway.

In order for this program to learn to play better checkers the contests that were stored in its move library would have to be moves other than its own. In fact, the highest level of play would be produced by a move library completely filled with winning moves made by the best players of the game. Suddenly we have an entirely different scenario, the computer is no longer required to "learn" new moves. Instead the computer is only "looking up" the next move. Certainly this "looking up" method exist completely outside of the field of learning and this project can no longer be considered a computer learning method.

In the introduction of this paper it was stated that the purpose of this project was to teach the computer the proper way to play a game by rewarding its good moves and punishing bad moves. After experimentation, it has been determine that this method of reward and punishment will not significantly change the behavior of the computer as it is applied here. A better place for this operant conditioning would be to use it in the shaping of the heuristic. This project has indicated that endgame performance of a computer's play can be improved by accessing a library of superior moves, however, this type of performance modifier cannot be labelled as a learning process.

The only hope that this project has left of actually being considered a learning procedure is described in this next possible scenario. The computer has a move library of past winning moves made by expert checkers players. In the same endgame, the computer accesses different boards from the move library that were stored from different players. The computer wins the current contest and links these boards together. Later in another contest, the computer accesses and executes this new combination. The question remains whether this can be classified as learning; the execution of a new series of moves formed from the moves of two or more previously observed games. The argument against this theory is that if the computer located these two moves the first time, given the same circumstances, would not the program find these two moves again even without them being linked together. The argument would seem to support the idea that this project does not

employ an effective learning technique, but instead uses a reference technique. Albeit, a reference technique that if loaded with the proper moves could improve the computer's play significantly.

RECOMMENDATIONS

This section is divided into to two areas. The first area describes specific problems that the programs has and then gives a recommendation to improve play. The second area suggest ideas that were not incorporated in this project but would enhance this projects performance.

PROBLEMS

As mentioned earlier, the number one enhancement for this project should be the replacement of all boards in the current move library with checkers games recorded from expert players. The higher quality moves stored initially would enhance the computer's play sooner and to a higher degree. The higher the quality of game the computer observes, the higher the quality of output it would produce.

A second area that needs work is the Heuristic procedure. In this project I grossly underestimated the importance of the heuristic. A prime example of how a poor heuristic causes difficulties is when a contestant enters endgame at a decided disadvantage in the number of checkers remaining, say three checkers to eight checkers. If one team enters endgame this far behind its opponent due to a poor heuristic, it is virtually impossible to win in the little time left the three checkers team. Thus, for the stored boards to have a chance the heuristic must play at an above average level. The heuristic would be the prime location to instigate an operant behavior influencing scheme.

A third area that would improve play is the choice of language and structure of the

program. Basic performed well in the area of video representation. Basic also proved to be an excellent choice for its readability which became useful in designing, editing and presenting the code. However, Basic fell far short in its speed of execution and when the program was structured using multiple recursive calls Basic bogged down even further. The average four ply move requires longer than seven minutes to execute. Selecting a faster language and structuring the program for speed would be an improvement on the project.

IDEAS

The first idea that might be worthy of investigation is the modification of the definition of the endgame. The idea is to expand the definition of the endgame from three checkers remaining on one team, to five checkers. If the game is lopsided, for example, three checkers to eight checkers, it is very difficult to come back against the odds for the victory. But if the endgame was defined as five checkers remaining on one team, the stored boards have a greater amount of playing time to influence the outcome of the game. However, with this expansion of the definition it is possible that the process would require so many stored boards that it would become impractical.

A second enhancement that might improve play would be an addition of a random choice when minimax presents three of four moves of equal merit. Currently the program always selects the first move and passes over the rest, giving the program a predictable pattern. However, a simple random choice between these boards would add variety to the computer's playing style. The outcome of the game might not be unduly influence when playing a computer component. However, without this randomness of play against a human competitor, the computer play becomes too predictable and the computer becomes easy prey to the human player. This random element would reduce the predictability of the computer's game.

A final idea that would help the heuristic would be to store opening moves and have the computer use standard opening moves used by checker experts. Like this project's move library routine, the computer would be able to access stored opening moves and possibly execute a proven winning opening strategy.

REFERENCES

Barr, Avron and Feigenbaum, Edward A.,1981. The Handbook of Artificial Intelligence. Volume 1. Heuristech press, Standford,California.

Ellzey, Roy S., 1989. Data Structures for Computer Information Systems. Second ed. Chicago: Science Research Associates.

Hart, T.P. and Edwards, D.J., 1963. *Artificial Intelligence Project Memo.* Massachusetts Institute of Technology, Cambridge, Massachusetts.

Hill, Winfred F., 1971. Learning, a Survey of Psychological Interpretations. London: Chandler Publishing Company.

Klausmeier Herbert J., 1971. Learning and Human Abilities. New York: Harper & Row Publishers.

Korf Richard E., 1988. *Search: A Survey of Recent Results.* Los Angeles: University of California.

Krutch John. 1980. *Experiments in Artificial Intelligence for Microcomputers.* H. Sams Publisher.

Luger, George F. and Stubblefield, William A.,1989. Artificial Intelligence and the Design of Expert Systems. The Benjamin/Cummings Publishing Company,Inc. Redwood City, California.

Nilsson, Nils J.,1980. Principles of Artificial Intelligence. Palo Alto, California: Tioga Publishing.

Pearl, Judea,1984. Heuristics: Intelligent Search Strategies for Computer Problem Solving. University of California, Addison-Wesley Publishing Company.

Samuel, Arthur L., 1959. Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of R&D, vol.3.

Samuel, Arthur L., 1967. Some Studies in Machine Learning Using the Game of Checkers II--Recent Progress. IBM Journal of R&D, vol.11.

Schneider David I., 1988. Handbook of Basic. Third Edition. New York: Brady.

Shannon C.E., 1950. *Programming a Computer for Playing Chess.* Philosophical Magazine **41**:256-275.

Winston, Patrick H.,1984. Artificial Intelligence. Second Edition. Addison-Wesley Publishing Company. Reading, Massachusetts.