Robots on a Chessboard

Fleury, S; Griffin, L; Paterson, H; Scarlet, A.

Group Two was tasked to develop a basic robot AI system to navigate a robot over a checkboard-like tile pattern. The robot's goal was to locate and move a weighted tower. We developed the robot by working as a single software team to develop each system of our AI in sequence. Our software team approach allowed us to leaver our respective strengths to quickly resolve each stage of development. We iterated through several, increasingly sophisticated, algorithms for each stage of the task, developing new ones in response to the difficulties we encountered during our ongoing testing. We encountered several difficulties during the task, related to our project management was well as more concrete technical issues.

Our robot divides the task into three distinct phases: First, advancing along the 'chessboard,' while counting tiles; Second, approaching the tower; Third, moving the tower off the tile it's located on. Intermediate tasks which didn't clearly belong in a phase, such as turning right after phase one before we could start phase two, were arbitrarily assigned to a phase in our initial design.

In phase one, we used a proactive correction algorithm to navigate the tiles. Our initial algorithms assumed course corrections would not be required because the robot would drive straight. Our testing showed this algorithm caused the robot deviate from the expected straight course and the error accumulation was difficult to correct. The algorithm currently stops after locating each black tile and computes a course correction. Our algorithm computes the correction by multiplying a hardcoded adjustment factor (based on the geometry of the tiles in the test environment) with the weighted average of the angle to each side, perpendicular to the direction of movement, of the black tile. Our robot is reset to its original heading, and the course correction is applied before the robot drives forward searching for the next black tile.

We detect the edges of tiles with by considering the difference of the recent time averages in an independent thread. The task specifications suggested that the lighting conditions in the test environment were variable - additionally, the dirt and speckled tiles introduce additional sensor noise. We use the average light-intensity reading over a short period, currently 0.1 seconds, to eliminate the environmental noise. We realised we could not use absolute or hardcoded values to detect the different tile colours because the detected readings would vary with the ambient lighting. We detect changes in tile colour by monitoring the difference of the time-averaged readings, relative to a threshold factor. Our algorithm determines colour of the new tile can be determined from the sign of the difference: A negative difference

indicates the drop of intensity as the robot moves onto a black tile.

Phase two starts assuming the robot is facing approximately towards the tower. Initially the algorithm drives forward under dead reckoning, because the tower is outside the maximum range of the ultrasound sonar. The robot enters a search and advance loop when it arrives within sonar range. The bot's search and advance loop searches forty-five degrease to each side of the robot for sonar returns, orientates the robot toward the minimum sensor reading, and drives a set distance forward. The robot will maintain the search loop until the robot is immediately in front of the tower, when at least one of the two exit conditions is met: The sonar return is less than a prescribed distance, or a touch sensor is triggered. The robot finally once aligns itself straight with respect to the tower and is ready to commence phase three.

The robot starts phase three assuming the tower is less than two centimetres from the robot in an approximately forward direction. The robot reverses briefly to provide itself with room to accelerate, then moves forward at full power to push the tower off the black tile. The robot uses dead reckoning to determine when the tower has been sufficiently moved. Our use of dead reckoning to compute the bot's final movement can lead the robot terminating in an erratic final position, but acceptable considering that the specification how far the tower should be moved were also imprecise. The robot needs to reverse before ramming the tower to accommodate its acceleration: The friction coefficient of an object is higher when it is static than in motion, so the robot needs to reach a higher speed than normal to start the tower moving.

As we developed our algorithms we encountered several difficulties that causes us to adjust our code or, in extreme cases, switch to using a different algorithm for a task. Our examples include the evolution of the bot's phase one algorithm in response to difficulties in the environment, unreliable sonar readings in the second phase, occurrences of an unrecoverable crash cause by lazily written multi-threaded code, and a misunderstanding of the project specifications. We also encountered project management difficulties when our member's timetables proved unaccommodating to group meetings.

Our initial approach to phase one was to drive the robot forward continuously, until the count of tile passed reached a specified value. Our initial code used the same multi-threaded tile counter as our final version. We found the robot tended not to drive straight, veering off the path of tiles, when we tested the robot with this algorithm. We assume this deviation was caused by a combination of asymmetric motor conditions, and slight undulations in the test surface. We were initially uncertain how to correct this behaviour so tried to move the robot in a zig-zagging course to 'average out' under the law of large numbers. This zig zagging also proved unsuccessful because of the floor slopes. We eventually mocked up the actions of the

robot, using a match-box car, so we could simulate the 'though process' of the robot as a group. Our childish approach to brainstorming earned us more than a few odd looks from passers-by in the computer science lobby but led us to our current algorithm of correcting the bot's course on each black tile. During phase two we encountered a comparable difficulty with an initial algorithm that tried to scan for the tower while the robot was under way. We used to same brainstorming and modelling approach to develop our current algorithm which stops to perform each sonar scan.

We occasionally observed an out-of-range exception being thrown as we tested various unrelated parts of our code. The exception was being thrown in our tile counting thread when two different function attempted to manipulate a list used to compute average readings from the light sensor. The bug was a classic data race caused by the tile counter thread deleting old values from the list before replacing them with a new one, while another thread computed the average value of this array, occasionally trying to access an indexed reading that had been deleted bug not yet replaced. As only one member of our group was familiar with multi-threading, the problem was referred to him for a solution: He had initially written the threading code in a hurry and had foreseen the chance of the data race occurring but deemed it too unlikely to justify researching the Python threading API. When we realised that the bug was a practical possibility, he implemented a lock on the shared data structure.

During the final phase of testing, we reviewed the specifications and realised he had misunderstood the position the robot would start in. The bot's algorithms assumed it would start on the first of the fifteen black tiles, facing down the line of tiles. We panicked when we noticed the actual requirement, during one of our final group testing session. We implemented a very naive algorithm to handle the 'new' starting requirement, because we decided we didn't have time to develop a more sophisticated approach: The robot handles a special case on the first black tile it finds, which turns itself right by ninety degrees and reverses slightly, before continuing with our existing algorithm.

Finally, we encountered some softer difficulties in our timetables. We would have preferred to approach this project by meeting and working together with a test robot. Our schedules prevented more than two of us meeting at a time, so we had to find another approach. We realised that only one of us, at any time, would be able to test the robot in the labs and correct bugs. We decided to use our poor scheduling to our advantage and use a 'programming team' model, where we concentrated our respective talents on the same part of the project until it was finished. Under this system, our strongest Python programmer took responsibility for writing the code, two algorithmists provided theroycrafted the algorithms, and one approver pro-actively reviewed identified possible bugs in the code. We used this system because we did not need to be in the lab at the same time to preform our tasks: The programmer could write and test at his will, the algorithmists could view the

test averment at their leisure, and the reviewer could provide a fresh pair of eyes from anywhere he could access the code. We also were able to develop our robot very quickly and reliably with this approach

Throughout our development of the robot we needed to solve several technical and non-technical problems. We were each challenged by our lack of foresight, lazy programming, and personal schedules. Individually, these challenges would have caused us significant difficulty. Collectively, we were able to pool our abilities and resolve our challenges efficiently. Our collaboration produced a series of simple, but robust algorithms that allow our robot to complete the task in a reasonable manner.