Parallel Search for Masyu Puzzles

Francis Stephens

The Problem

Masyu puzzles first appeared in *Puzzle Communication Nikoli* a Japanese quarterly magazine publishing logic puzzles. The rules are simple.

A grid-like board is presented scattered with black or white pearls. The player must draw a closed circuit passing through each pearl following three basic rules.

1. No grid location may be visited more than once
2. When passing through a black pearl a right angle turn must be made
3. We must pass straight through a white pearl but there must be a right angle turn made either immediately before or after (or before and after) entering the pearl.

The Solution

Given the NP-complete nature of solving Masyu puzzles by computer I have chosen depth first search as my primary problem solving tool modelled after the popular DPLL sat solver for solving the canonical NP-complete problem, Boolean Satisfiability. A DPLL sat solver can be characterised roughly as a depth first search algorithm with search-space pruning.

Most depth first search algorithms have four important components.

1. A solution criteria
2. Branching decision points
    a. With optional decision heuristics
3. Search-space pruning
4. Backtracking

Our solution criteria is a complete circuit following the rules described above. Our branching decision points are the possible directions of movement that can be taken at each point in the grid while forming the desired circuit. The rules for the Masyu puzzle provide us with a powerful search space pruning tool as explicit constraints can be added to the search for each move. An example of this would be that moving onto a black pearl constrains the search by forcing us to either turn left or right next, we cannot decide to carry on straight

ahead. White pearls provide much more complicated constraints and combinations of pearls and movement decisions can be turned into general rules for constraining the search even further. Constraints are covered in detail below. We have chosen a simple decision heuristic by simply prioritising moves which take us closer to the nearest unvisited pearl.

The Sequential Algorithm

The search algorithm we have employed has a simple structure. Because the stack is a very important part of the parallelisation of my solution we will include it explicitly when describing a general depth first search algorithm.

1. From the current grid position push all possible directions onto the stack
    a. We push directions onto the stack in reverse order of how close they bring us to the nearest unvisited pearl
2. The choice at the top of the stack becomes our new direction
    a. The new direction is used to calculate a new grid position
    b. If the new direction or grid position is illegal we pop the direction stack and choose a new direction
    c. If all directions are popped for the current position the current position is removed and an untried direction is tried from the previous grid position
    d. If all positions are popped the puzzle is deemed unsolvable
3. Given a successful movement decision we analyse the puzzle under the conditions of that decision, adding any explicit constraints that are indicated by our rules.
4. If our last movement decision brings us back to our starting position we test to see if we have satisfied the closed pearl circuit.
    a. If we have then - success!
    b. Otherwise pop the last decision and continue searching

The Constraints

Movement Based Constraints

The most basic constraint is caused by a movement on the grid. Since our circuit cannot cross itself whenever we make a pair of moves, into a grid square and then out of it again, we know that no other movement can bring us back into that square. To express this we mark the two movement directions that were not taken on that square as forbidden.

The next simple constraint is activated when we move into a square containing a black pearl. We are not able to move out of this square in the same direction we moved into it. Therefore a constraint is added to express this fact.

The white pearl constraints have two parts. First we are not able to make a turn on a white pearl so perpendicular constraints are added to forbid this. The second white pearl constraint is activated when we enter a white pearl having not made a right angled turn immediately before hand. We are required to make one immediately after leaving the white pearl.

Static Constraints

In addition to the constraints that are added dynamically as we build our search path there are powerful constraints that can be added before the first move is made.

The first is the simple observation that a white pearl position in any corner of the grid makes it impossible to travel through directly. This means we can determine that the puzzle is unsolvable without any searching at all.

If a pair of adjacent white pearls appear along the edge of a grid we know that we must enter them parallel to that edge while making a right angled turn away from the edge immediately before and after the pair of pearls. If three or more white pearls appear in a row along the edge of the grid the puzzle is unsolvable.

If three or more pearls appear in a row anywhere in the the grid then it is not possible to move through them in the direction of the row. We must cross the pearls perpendicular to the row, often resulting in a weaving back and forth pattern.

Over Constrained Squares

It can be observed that to visit any square we must always make two movements. One into

the square and one out. Because we can't move backward the way we came there must be at least two unconstrained directions for moving across a square. This leads us to the rule that any move which causes an unvisited pearl's square to acquire three constraints is illegal.

Constraints Not Included

Due to time limitations there is further constraint analysis that could easily, and I think very profitably, be added to our solver. The details are not covered here but it is important to note that in the area that likely defines the efficiency of a depth first search algorithm more than anything else, search space pruning, there remain significant improvements to be made. One important consideration that is completely absent from my solution is constraint propagation. Under many circumstances constraints added by the simple rules above can lead us to uncover further constraints simply by looking that the constraints which exist. A kind of recursive constraint propagation algorithm would likely yield significant improvements in the search space pruning of this algorithm.

Parallelisation

Stack Sharing (a.k.a. Work Stealing)

The key insight to my approach to parallelising the sequential algorithm described above was that when the work stack, chiefly the stack of chosen directions, was popped to facilitate backtracking the new state of the algorithm worked despite being entirely ignorant of what values had been popped off the stack. This meant that the stack could, in principle, have portions of itself shipped out to other cooperating processes. The process receiving a portion of a work-stack could continue work on that stack without penalty. Furthermore, if the process which had shared this work out knew which parts of the stack had been shared it would not need to explore these parts of the stack, but could pop them without concern during backtracking, knowing that another process was exploring that part of the stack.

This general approach to parallelising sharable workloads is often called work-stealing. It has been written about extensively. A concrete implementation, for a slightly different problem type, appears in the Fork-Join framework by Doug Lea which will be included in Java 7.

Breadth First, Depth First and Serendipity

For this project we tried two different approaches to sharing out the work. We have named them breadth first and depth first work sharing.

Depth First

For depth first work sharing we share the stack by alternately sharing and retaining sharable stack elements. In a scenario with two cooperative workers sharing a stack would result in both workers searching in roughly the same area of the search tree. Intuitively this approach works well if we believe our early movements are unlikely to be backtracked over and that our decision heuristic is of a good quality. This work sharing prioritises existing decisions efficiently searching their consequences but makes early mistakes very expensive as there will be a lot of work done before early decisions are backtracked.

An illustration of depth first work sharing

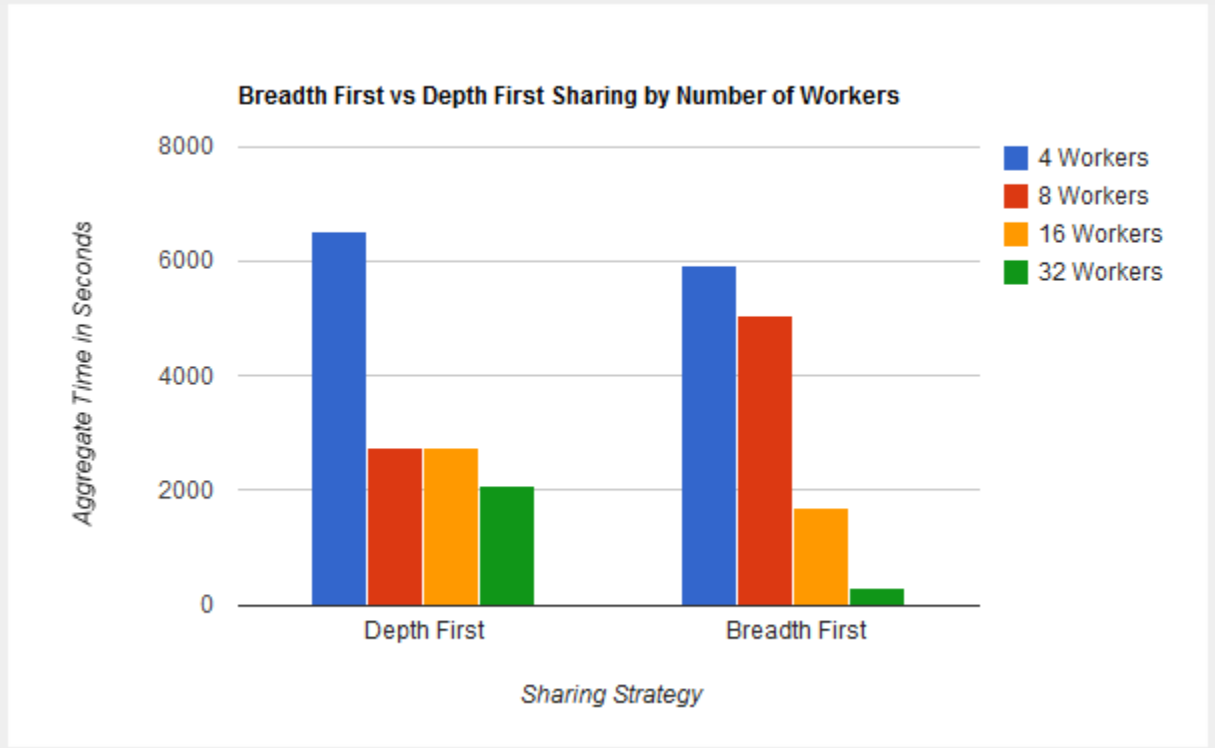
Breadth First

For breadth first work sharing we share the stack by sharing the first half of sharable elements and retaining the rest. This results in two workers exploring very different parts of the search tree. Intuitively this approach works well if we believe that early decisions are likely to be backtracked over and that our decision heuristic is of a low quality. This work sharing strategy prioritises trying alternate decisions early in the search stack but existing deep searches which are of a high quality will not be assisted by other workers. This search works best by attempting to increase the chance of a lucky encounter with a high quality decision branch.

```
| Retained |
| Retained |
| Retained |
|  Shared  |
|  Shared  |
|  Shared  |
------------
```

An illustration of breadth first work sharing

We have chosen breadth first work sharing based on test runs indicating that my heuristic wasn't a particularly good one.



The Means of Production

A common way to facilitate the efficient sharing of work across a pool of workers is via some kind of orchestrating master process. While this approach is simple and easily understood the master process is a natural bottle neck and risks increasing the chattiness of our parallel algorithm to unacceptable levels. I opted instead to employ a peer to peer approach instead.

Worker Chains

In our solution each worker in our pool is connected to two adjacent workers in a chain which connects all workers to each other, transitively, and loops around on itself. All messages are passed along the chain from worker to worker via these links, except for work sharing which is done directly.

Types of Messages

There are six kinds of messages which may be passed along the worker chain.

1. Work Requests
    a. These travel from worker to worker until either
        i. Work is provided - see below - and the message is discarded
        ii. The message loops back to its sender - casing the sender to hibernate periodically before again requesting work
2. Work Responses
    a. These messages are passed from the responder directly back to the requester and contain a stack of shared work
3. Wake-Up messages
    a. These messages indicate that a worker has woken up from hibernation
    b. This message must pass through every single worker until it returns to the sender and is discarded
    c. Receipt of this message will cause the receiving worker to increase its private count of the number of active workers
4. Hibernation messages
    a. These messages indicate that a worker has just entered hibernation
    b. This message must pass through every single worker until it returns to the sender and is discarded
    c. Receipt of this message will cause the receiving worker to decrease its private count of the number of active workers
5. Success Message
    a. This message is sent from a worker, potentially several simultaneously, to an external process who waits for this message alone.
6. Failure Message
    a. This message is sent from a worker, potentially several simultaneously, who waits for this message alone.

All messages must be appear to the receiver in the order in which they were sent. [1]

[1]This approach to work sharing was first prototyped in Erlang which provides these message delivery guarantees in a distributed as well as SMP context.

Additionally all messages must be forwarded on in the order in which they were received. This ensures a crucial system property that no worker process should receive a hibernation message that was sent before the wake-up message that must have preceded it.

Initial State

When we set our worker pool running it contains a fixed number of workers, each of which is mapped to a Java thread. A single worker, primary worker, is set to begin work on the problem immediately and the remaining workers, secondary workers, are set to immediately request work. Each worker has its private active worker count set to the total initial number of workers.

Let us imagine an artificial scenario where each of the secondary workers is given CPU time immediately and finding they have no work available immediately make a work request. These requests are passed around the worker pool chain and each time the primary worker receives a request he forwards it on, deeming that he doesn't have enough work to share. In this scenario each of the secondary workers will receive his own work request and immediately go into hibernation. Before entering hibernation each secondary worker will send a hibernation message that will pass around the entire network. It is easy to see that when this process is complete each worker will believe that there is exactly one worker, the primary worker, still active. This scenario serves to illustrate that each workers private active worker count is eventually consistent.

Success and Unsolvable Puzzles

When a worker finds a satisfying solution to the Masyu puzzle it passes the solution back to an external waiting process, often the main thread, and the worker chain is shutdown. This is simple and straightforward but unsolvable puzzles give us a problem - when every worker has run out of work how do we know and how can we be sure that this condition isn't triggered spuriously?

The crucial property of the worker chain is that no worker will ever believe that there are zero active workers unless this is globally true. A rigorous proof of this is beyond this report but we can provide an indication of why this is true.

Beginning from the initial state, with every worker believing that all workers are active, the first few processing rounds usually result in the majority of workers hibernating. We have seen that this results in eventual consistency, but the system will probably not remain in a stable state long enough for all workers to have a consistent view to the number of active workers. We can see that any worker's active worker count may be less or more than the actual number. However, in order for any worker's active worker count to fall to zero it must have received at least n (n being the number of workers in the chain) hibernation messages. Since, in this scenario, there have been n hibernation messages sent, and not all workers are

hibernating, there must have been some workers who hibernated, woke up and hibernated again. In this case we know that our receiving worker must necessarily have received the wake-up messages before the subsequent repeat hibernations. This ensures that the receiving worker's active worker count will not reach zero. Although this scenario illustrates only a tiny fraction of the possible combinations of messages I hope that it provides a good intuition for the robustness of this message passing system.

Heuristics Preprocessing

Above it was mentioned that we prioritise movement decisions which take us closest to the nearest unmarked pearl. It isn't a cheap operation to determine the nearest pearl and this information is precomputed and provided to each worker as a global data-structure. As each pearl requires an ordered list of its nearest neighbours we can easily parallelise this work at the outer-loop level, one of the happiest parallel tasks there is. This approach yields a very near to linear speedup and using merge sort on a quad core i7 processor puzzles containing 2000 pearls can be preprocessed in this way in a handful of seconds.

Conclusion

In conclusion it appears that while the worker chain approach is a fair approach to work sharing for this problem domain the sequential algorithm itself is the weakest part of this implementation. Given the exponential growth of possible search paths any number of processors, be it 40 or 1024, will not make up for an under constrained search space.