Computing the Probability of Striking a Battleship

Jack Spalding-Jamieson

David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Canada

What is Battleship?

- 2-player pen and paper game.
- Sometimes played using special plastic boards too.

Battleship Game Rules

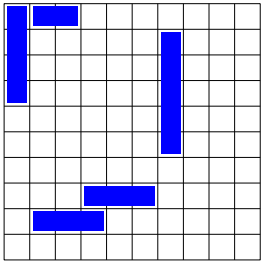
A quick summary of the rules of the game:

- Each player has a 2D grid, typically sized 10×10.

Battleship Game Rules

A quick summary of the rules of the game:

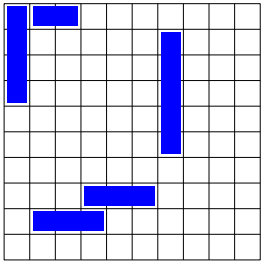
- Each player has a 2D grid, typically sized 10×10.
- Players start by placing 'ships' into hidden locations on their own grid. The ships can touch, but may not overlap. Typically, the ships have sizes $\{5, 4, 3, 3, 2\}$.



Battleship Game Rules

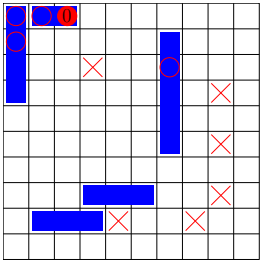
A quick summary of the rules of the game:

- Each player has a 2D grid, typically sized 10×10.
- Players start by placing 'ships' into hidden locations on their own grid. The ships can touch, but may not overlap. Typically, the ships have sizes $\{5, 4, 3, 3, 2\}$.



In general, we denote the number of ships as n.

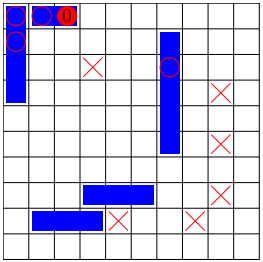
Battleship Game Rules (2)



Players now take turns attacking each other's grids.
During a turn:

- Attack a square on the opponent's grid.

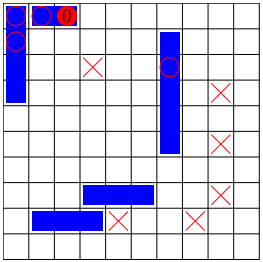
Battleship Game Rules (2)



Players now take turns attacking each other's grids.
During a turn:

- Attack a square on the opponent's grid.
- Three possible results:

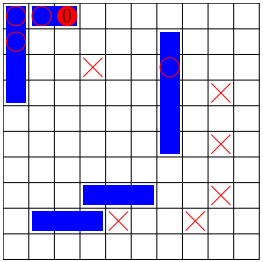
Battleship Game Rules (2)



Players now take turns attacking each other's grids.
During a turn:

- Attack a square on the opponent's grid.
- Three possible results:
  - Miss.

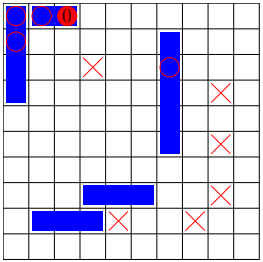
Battleship Game Rules (2)



Players now take turns attacking each other's grids.
During a turn:

- Attack a square on the opponent's grid.
- Three possible results:
  - Miss.
  - Hit.

Battleship Game Rules (2)



Players now take turns attacking each other's grids.
During a turn:

- Attack a square on the opponent's grid.
- Three possible results:
  - Miss.
  - Hit.
  - Sink! Not only was something sunk, but every other square of that ship was already hit. The attacking player also learns exactly which ship was sunk ("You sunk my battleship.").

Greedy Battleship AI

What makes a good battlesihp AI?

Greedy Battleship AI

What makes a good battlesihp AI?

One ‘easy’ idea (the ‘greedy’ approach): When attacking, try to choose the square that is most likely to have a ship on it.

Greedy Battleship AI

What makes a good battlesihp AI?

One ‘easy’ idea (the ‘greedy’ approach): When attacking, try to choose the square that is most likely to have a ship on it. It suffices to compute the probability that each square has a ship on it.

Greedy Battleship AI

What makes a good battlesihp AI?

One ‘easy’ idea (the ‘greedy’ approach): When attacking, try to choose the square that is most likely to have a ship on it. It suffices to compute the probability that each square has a ship on it. This is the approach we will use here.

Greedy Battleship AI

What makes a good battlesihp AI?

One ‘easy’ idea (the ‘greedy’ approach): When attacking, try to choose the square that is most likely to have a ship on it. It suffices to compute the probability that each square has a ship on it. This is the approach we will use here.

Another idea: Minimize the maximum number of attacks necessary. Fiat and Shamir made some progress on a special case of this.

Probabilities

In order for the 'probability' that a square has a ship on it to even be well-defined, we must first make a key assumption:

- It is assumed that the opponent's configuration of ships is chosen **uniformly at random from all legal configurations**.

Probabilities

In order for the 'probability' that a square has a ship on it to even be well-defined, we must first make a key assumption:

- It is assumed that the opponent's configuration of ships is chosen **uniformly at random from all legal configurations**.

Ultimate Goal: Compute these probabilities in about the same amount of time a human would take to make a move (up to about 10 seconds).

The Battleship Probability Problem

In summary, the exact problem we attempt to solve is:

The Battleship Probability Problem

In summary, the exact problem we attempt to solve is:

- Assume your opponent in a game of battleship chose a ship configuration uniformly at random.

The Battleship Probability Problem

In summary, the exact problem we attempt to solve is:

- Assume your opponent in a game of battleship chose a ship configuration uniformly at random.
- Given: The attacks you have made so far, and their results (miss, hit, sink).
- For each square s on the opponent's board, output the probability that s has a ship on top of it.

Related Work

This is not the first attempt at a variation of this problem. C. Liam Brown attempted this problem as well, but his full implementation of this problem takes about a day to run. He instead used MCMC methods to estimate the probabilities.

Related Work

This is not the first attempt at a variation of this problem. C. Liam Brown attempted this problem as well, but his full implementation of this problem takes about a day to run. He instead used MCMC methods to estimate the probabilities.

Our solution does not employ a significantly different approach, but rather uses a long series of optimizations and changes to drastically improve the performance overall (success by a thousand cuts).

Related Work

This is not the first attempt at a variation of this problem. C. Liam Brown attempted this problem as well, but his full implementation of this problem takes about a day to run. He instead used MCMC methods to estimate the probabilities.

Our solution does not employ a significantly different approach, but rather uses a long series of optimizations and changes to drastically improve the performance overall (success by a thousand cuts).
We eventually achieve an implementation that can run in just a few seconds.

Placement Graph

Create a graph (the 'placement graph' G_1):

- Vertex for every ship + placement pair.

Placement Graph

Create a graph (the 'placement graph' G_1):

- Vertex for every ship + placement pair.
- Edge for every pair of placements of distinct ships that do not overlap

Placement Graph

Create a graph (the 'placement graph' G_1):

- Vertex for every ship + placement pair.
- Edge for every pair of placements of distinct ships that do not overlap (i.e. it is legal to use both placements simultaneously in a configuration).

Placement Graph

Create a graph (the 'placement graph' G_1):

- Vertex for every ship + placement pair.
- Edge for every pair of placements of distinct ships that do not overlap (i.e. it is legal to use both placements simultaneously in a configuration).

This is a multi-partite graph where each ship forms a part, so in particular it is an n-partite graph.

Placement Graph

Create a graph (the 'placement graph' G_1):

- Vertex for every ship + placement pair.
- Edge for every pair of placements of distinct ships that do not overlap (i.e. it is legal to use both placements simultaneously in a configuration).

This is a multi-partite graph where each ship forms a part, so in particular it is an n-partite graph.

The *n-cliques* in this graph correspond to legal configurations at the start of the game.

Placement Graph (Example)

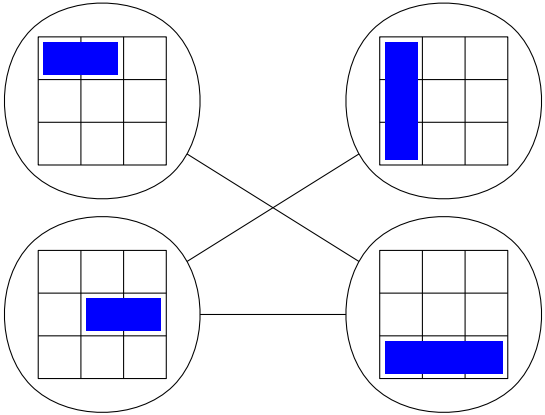


Figure: For a 3x3 board with ship sizes $\{2, 3\}$, the above diagram represents a subgraph of the board-induced graph.

Valid Placement Subgraph

Given the attacks and results so far, we take an induced subgraph of the placement graph (the *valid placement graph* $G_2 \subset G_1$) using vertices corresponding to placements that still could be possible.

Valid Placement Subgraph

Given the attacks and results so far, we take an induced subgraph of the placement graph (the *valid placement graph* $G_2 \subset G_1$) using vertices corresponding to placements that still could be possible.
Now, the remaining legal configurations correspond to cliques in G_2 that cover all squares that have received *hit* results from attacks.

Using Clique Iteration

The most obvious solution is now to iterate through all cliques in G_2, count up how many times each vertex appears in the cliques, and then compute how many times each square appears.

Using Clique Iteration

The most obvious solution is now to iterate through all cliques in G_2, count up how many times each vertex appears in the cliques, and then compute how many times each square appears.

In the most common variant of the game with a 10×10 board and ships of sizes $\{5, 4, 3, 3, 2\}$, there are $30093975536 \approxeq 3 \times 10^{10}$ cliques in G_1.

Using Clique Iteration

The most obvious solution is now to iterate through all cliques in G_2, count up how many times each vertex appears in the cliques, and then compute how many times each square appears.

In the most common variant of the game with a 10×10 board and ships of sizes $\{5, 4, 3, 3, 2\}$, there are $30093975536 \approxeq 3 \times 10^{10}$ cliques in G_1.
Modern CPUs only run at about 3×10^9 Hz, and even basic operations often take a few dozen clock cycles.

Using Clique Iteration

The most obvious solution is now to iterate through all cliques in G_2, count up how many times each vertex appears in the cliques, and then compute how many times each square appears.

In the most common variant of the game with a 10×10 board and ships of sizes $\{5, 4, 3, 3, 2\}$, there are $30093975536 \approxeq 3 \times 10^{10}$ cliques in G_1.
Modern CPUs only run at about 3×10^9 Hz, and even basic operations often take a few dozen clock cycles.

In order to attain our goal of a few seconds per input, we're going to need to perform **lots** of performance optimizations.

Clique Iteration Background

Bron and Kerbosch have a well-known method for clique iteration for general graphs, but a more naive approach works much better in our case:

```
For each placements p_1 of ship s_1:
  For each placements p_2 of ship s_2 compatible with the previous ones:
    For
      ...
        Record the clique if it corresponds to a valid configuration.
```

Note: `placements p_i of ship s_i` corresponds exactly to the vertices in part i of the graph.

Benchmarks

Benchmark methodology:

- Query-less board has the most cliques, so used for benchmark.
- All other inputs have a subset of the cliques of the empty board.

Benchmarks ran on an AMD Ryzen 5 1400 CPU and an NVIDIA RTX 3080 GPU.

We test on the most common variant of a 10×10 board with ships of sizes $\{5, 4, 3, 3, 2\}$.

Algorithm Engineering: Ship Order

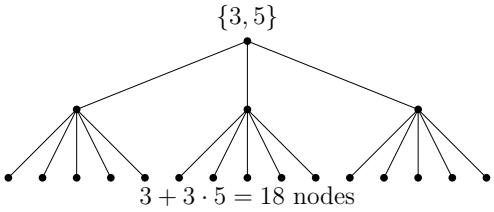
Simple optimization: Order the ships from *largest to smallest*.

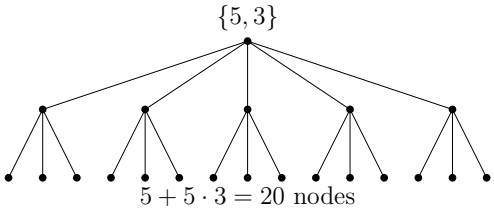
Algorithm Engineering: Ship Order

Simple optimization: Order the ships from *largest to smallest*.
Smaller ships have more associated vertices.

Algorithm Engineering: Ship Order

Simple optimization: Order the ships from *largest to smallest*.
Smaller ships have more associated vertices.





Algorithm Engineering: Adjacency Masks

We can precompute the *compatible with the previous ones* filter.

Algorithm Engineering: Adjacency Masks

We can precompute the *compatible with the previous ones* filter.

For every pair of ships s_1 and s_2,

Algorithm Engineering: Adjacency Masks

We can precompute the *compatible with the previous ones* filter.

For every pair of ships s_1 and s_2, and every placement p_1 of s_1,

Algorithm Engineering: Adjacency Masks

We can precompute the *compatible with the previous ones* filter.

For every pair of ships s_1 and s_2, and every placement p_1 of s_1, compute a set of placements p_2 of s_2 that do not intersect p_1.

Algorithm Engineering: Adjacency Masks

We can precompute the *compatible with the previous ones* filter.

For every pair of ships s_1 and s_2, and every placement p_1 of s_1, compute a set of placements p_2 of s_2 that do not intersect p_1.
This pre-computation turns out to take a negligible amount of time compared to the clique iteration.

Algorithm Engineering: Adjacency Masks

We can precompute the *compatible with the previous ones* filter.

For every pair of ships s_1 and s_2, and every placement p_1 of s_1, compute a set of placements p_2 of s_2 that do not intersect p_1.
This pre-computation turns out to take a negligible amount of time compared to the clique iteration.

We can now maintain a *currently valid* set for each part of the graph:

Algorithm Engineering: Adjacency Masks

We can precompute the *compatible with the previous ones* filter.

For every pair of ships s_1 and s_2, and every placement p_1 of s_1, compute a set of placements p_2 of s_2 that do not intersect p_1.
This pre-computation turns out to take a negligible amount of time compared to the clique iteration.

We can now maintain a *currently valid* set for each part of the graph:When choosing a vertex corresponding to (s_1, p_1) in each of the for loops, intersect the generated sets above with the currently valid sets for every choice of s_2.

Algorithm Engineering: Bitsets

To speed up the previous improvement, we can plug in a bitset to represent the sets.

Algorithm Engineering: Bitsets

To speed up the previous improvement, we can plug in a bitset to represent the sets.

The important aspect of bitsets are that they very much speed up intersection, which is the core operation used for sets in the *critical path* of the algorithm (along with iteration).

Algorithm Engineering: Bitsets

To speed up the previous improvement, we can plug in a bitset to represent the sets.

The important aspect of bitsets are that they very much speed up intersection, which is the core operation used for sets in the *critical path* of the algorithm (along with iteration).

The C++ STL *std::bitset* has extremely well-optimized implementations for our purposes,

Algorithm Engineering: Bitsets

To speed up the previous improvement, we can plug in a bitset to represent the sets.

The important aspect of bitsets are that they very much speed up intersection, which is the core operation used for sets in the *critical path* of the algorithm (along with iteration).

The C++ STL *std::bitset* has extremely well-optimized implementations for our purposes,but it requires the next optimization.

Algorithm Engineering: Compile-Time Values

Since our goal is an implementation that can solve the battleship probability problem within a few seconds, it can help a lot to fix a variant of battleship at compile-time.

Algorithm Engineering: Compile-Time Values

Since our goal is an implementation that can solve the battleship probability problem within a few seconds, it can help a lot to fix a variant of battleship at compile-time.

In addition, this actually allows us to remove all heap allocations from the critical path of our code **(zero-alloc code!)**.

Algorithm Engineering: Compile-Time Recursion Unrolling

Normally, to implement a variable number of embedded for-loops, we would use a recursive function. However,

Algorithm Engineering: Compile-Time Recursion Unrolling

Normally, to implement a variable number of embedded for-loops, we would use a recursive function. However, function calls are expensive.

Algorithm Engineering: Compile-Time Recursion Unrolling

Normally, to implement a variable number of embedded for-loops, we would use a recursive function. However, function calls are expensive.

Since the number of ships is now fixed at compile-time, we can use meta-programming to generate these for-loops as one big function.

Algorithm Engineering: Compile-Time Recursion Unrolling

Normally, to implement a variable number of embedded for-loops, we would use a recursive function. However, function calls are expensive.

Since the number of ships is now fixed at compile-time, we can use meta-programming to generate these for-loops as one big function.
Variadic C++ Templates make this easy!

Algorithm Engineering: GPU Computation

We removed heap allocations, so it's very easy to run our existing code on a GPU.

Algorithm Engineering: GPU Computation

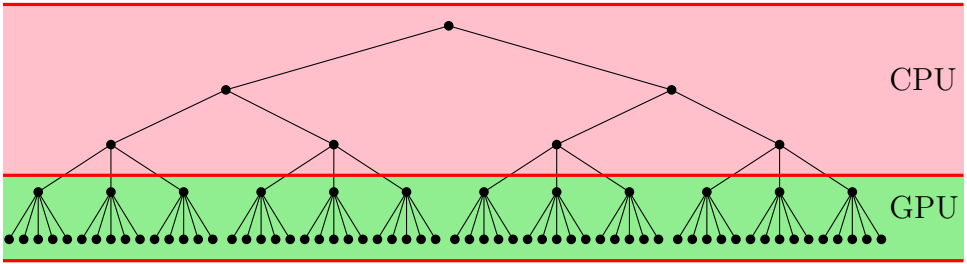
We removed heap allocations, so it's very easy to run our existing code on a GPU.

To make use of the parallelism, we run the innermost for-loops in parallel:

Algorithm Engineering: GPU Computation

We removed heap allocations, so it's very easy to run our existing code on a GPU.

To make use of the parallelism, we run the innermost for-loops in parallel:



Final Benchmarks

Final Benchmarks

Single-threaded CPU implementation: 146.37 seconds.

Final Benchmarks

Single-threaded CPU implementation: 146.37 seconds.

Multi-threaded CPU implementation (8 threads): 36.36 seconds.

Final Benchmarks

Single-threaded CPU implementation: 146.37 seconds.

Multi-threaded CPU implementation (8 threads): 36.36 seconds.

GPU implementation: 5.70 seconds.

Final Benchmarks

Single-threaded CPU implementation: 146.37 seconds.

Multi-threaded CPU implementation (8 threads): 36.36 seconds.

GPU implementation: 5.70 seconds.

This certainly meets our "few seconds" goal.

Final Benchmarks

Single-threaded CPU implementation: 146.37 seconds.

Multi-threaded CPU implementation (8 threads): 36.36 seconds.

GPU implementation: 5.70 seconds.

This certainly meets our "few seconds" goal.

Source code:
https://github.com/jacketsj/gpgpu-fun/tree/master/battleship/rewrite

Probabilities

If you were curious what we were computing for the benchmark, the most common variant has the following probabilities for the blank board:

0.080	0.115	0.143	0.159	0.167	0.167	0.159	0.143	0.115	0.080
0.115	0.143	0.166	0.178	0.184	0.184	0.178	0.166	0.143	0.115
0.143	0.166	0.184	0.194	0.199	0.199	0.194	0.184	0.166	0.143
0.159	0.178	0.194	0.203	0.208	0.208	0.203	0.194	0.178	0.159
0.167	0.184	0.199	0.208	0.214	0.214	0.208	0.199	0.184	0.167
0.167	0.184	0.199	0.208	0.214	0.214	0.208	0.199	0.184	0.167
0.159	0.178	0.194	0.203	0.208	0.208	0.203	0.194	0.178	0.159
0.143	0.166	0.184	0.194	0.199	0.199	0.194	0.184	0.166	0.143
0.115	0.143	0.166	0.178	0.184	0.184	0.178	0.166	0.143	0.115
0.080	0.115	0.143	0.159	0.167	0.167	0.159	0.143	0.115	0.080

Applicability to Extensions of Battleship

This approach also extends perfectly to other variations of Battleship, e.g.:

- The variation where no two ships can touch.
- Battleship Outer Space
- Hexagonal Battleship

The Last Slide

Fin.