1. (Exhaustive Search, 1 point for the code linting, 9 points for functional correctness, 5 points for space performance, 5 points bonus for time performance)

Suppose that a manufacturer takes orders for their product in custom amounts but fulfills those orders using boxes with standard sizes. Because it takes time to switch the packaging equipment from one box size to another, the manufacturer would prefer to use as few different box sizes as possible. For instance, if small, medium, and large boxes hold four, five, and six items, respectively, then the manufacturer would prefer to fulfill an order for 19 items with one small and three medium boxes ($1 \cdot 4 + 3 \cdot 5 = 19$). While the alternative—two small, one medium, and one large box—would also hold the correct number of items ($2 \cdot 4 + 1 \cdot 5 + 1 \cdot 6 = 19$), that plan would require three different box sizes instead of just two.

This problem of deciding how to best box an order can be solved by exhaustive search.

In `boxes.js`, the off-the-shelf helper function `partitions` takes a positive integer and a list of distinct positive integers. (Its third parameter, `forbiddenCount`, is only for internal use.) By tracing or running the function on small inputs, determine what it computes.

Then use `partitions` to complete the JavaScript function `chooseBoxes` further down in `boxes.js` so that `chooseBoxes` takes in an order quantity and a list of box sizes and returns a dictionary mapping box sizes to box counts with as few keys as possible. For example, `chooseBoxes(19, [4, 5, 6])` should evaluate to a map with two keys: the key `4` associated with the value `1` (signifying one small box) and the key `5` associated with the value `3` (signifying three medium boxes).

Unit tests to check functional correctness are provided in `boxes.test.js`, and you can run them using the `test-once:boxes:correctness` NPM script.

The current implementation of `partitions` uses memory very inefficiently when run on large values of `positiveInteger`. To earn the additional points for space performance, convert it to an equivalent generator function so that its memory usage only asymptotically depends on `allowedParts`, not `positiveInteger`.

A naïve exhaustive search that considers every possible way of boxing the items will, on average, run slowly on large inputs. To earn the bonus points for time performance, also find a way to arrange the search space so that the algorithm typically considers far fewer alternatives on large inputs. You may import and use any of the generator functions provided in `standardGenerators.js`.

Unit tests to check time performance (which also include specific bounds for problems of varying difficulty) are provided in `boxes.test.js`, and you can run them using the `test-once:boxes:performance` NPM script. (Be aware that you will forfeit your bonus points if you write code to hide basic operations from the performance tests or otherwise falsify your performance results.)

Finally, use the project's built-in linter to check your code style. Any `eslint-disable` directives will be ignored by the autograder, so do not use them to hide warnings in your local copy, or you may miss code style problems that will cost you points.

First we need to determine what `partitions` computes. Plugging in some likely values from the example above, the function call `partitions(19, [4, 5, 6])` gives us back a list with two maps, `{ 6 => 1, 5 => 1, 4 => 2 }` and `{ 5 => 3, 4 => 1 }`, which we can recognize as the two ways to package 19 items into boxes of size four, five, and/or six. In short `partitions` generates all of the candidate maps we will need for our search.

Based on that discovery, we can immediately write a find-one-optimizing search that optimizes map size, as shown on the next page.

```
export function chooseBoxes(quantity, boxSizes) {
  let best = undefined;
  for (const partition of partitions(quantity, boxSizes)) {
    if (best === undefined || partition.size < best.size) {
      best = partition;
    }
  }
  return best;
}
```

Next, to convert `partitions` to a generator function, we can mostly go through the usual process:

- We remove the declaration `const results = [];`. (This is a good choice for our first edit because the linter will highlight all uses of `results` so that we remember to modify them.)
- We convert every `return results;` to just `return;`. The `return` at the end of the function is now redundant, so we can remove it.
- We convert every call to `results.push` into a `yield` of the same value.
- We write a `*` after the keyword `function`.

The one tricky part is this line:

```
return positiveInteger === 0 ? [new Map()] : [];
```

where we have a `return`, but it is not returning `results`. Fortunately, if we refactor that code to show the computational steps one-by-one, we can get it into suitable form:

```
if (positiveInteger === 0) {
  const results = [];
  results.push(new Map());
  return results;
}
const results = [];
return results;
```

And this code can be converted using the process described above:

```
if (positiveInteger === 0) {
  yield new Map();
  return;
}
return;
```

which could be simplified to:

```
if (positiveInteger === 0) {
  yield new Map();
}
return;
```

After conversion, `partitions` looks like the code on the next page.

```
function*partitions(positiveInteger,
                    allowedParts,
                    forbiddenCount = 0) {
  if (forbiddenCount === allowedParts.length) {
    if (positiveInteger === 0) {
      yield new Map();
    }
    return;
  }
  const part = allowedParts[forbiddenCount];
  const maximumCount = Math.floor(positiveInteger / part);
  // This special case is not necessary for correctness,
  // but helps with performance:
  if (forbiddenCount === allowedParts.length - 1) {
    if (maximumCount * part === positiveInteger) {
      yield maximumCount === 0 ?
        new Map() :
        new Map([[part, maximumCount]]);
    }
    return;
  }
  for (let i = maximumCount; i >= 0; --i) {
    for (const completion of partitions(positiveInteger - i * part,
                                        allowedParts,
                                        forbiddenCount + 1)) {
      if (i > 0) {
        completion.set(part, i);
      }
      yield completion;
    }
  }
}
```

Finally, for the bonus, the instructions tell us that the search space can be rearranged to improve average-case performance, a reference to the trick from class where the search space is ordered by quality so that the search can break on the first solution it finds. So instead of investigating subsets of box sizes in the order found by `partitions`, we need to consider them from smallest to largest. Among the generator functions in `standardGenerators.js`, that means using `subsetsInSizeOrder`, like this:

```
import { subsetsInSizeOrder } from './standardGenerators.js';

export function chooseBoxes(quantity, boxSizes) {
  for (const subset of subsetsInSizeOrder(boxSizes)) {
    // ...
    if (/* ... */) {
      return /* ... */;
    }
  }
  return undefined;
}
```

Now the question is: Given a subset of box sizes, is there a way to quickly check whether it works for this order quantity? There is: we call `partitions(quantity, subset)`, and if we get back any answer at all, it must be an answer using all of the boxes sizes in our subset since the problem was not solvable with any smaller subset.

Since `partitions` is a generator function now, we do not even have to wait for it to find all solutions with our subset. We can just ask it to yield one value and then let the generator be garbage collected. The code looks like this:

```
export function chooseBoxes(quantity, boxSizes) {
  for (const subset of subsetsInSizeOrder(boxSizes)) {
    const { value, done } = partitions(quantity, subset).next();
    if (!done) {
      return value;
    }
  }
  return undefined;
}
```

where the `done` boolean tells us whether the generator found anything (if not, it will say that it is done), and `value` is the solution found if one exists.

2. (Exhaustive Search, 1 point for the code linting, 7 points for functional correctness, 7 points for performance)

An **arithmetic sequence** is a sequence of numbers where the difference between successive terms, the sequence's **stride**, is constant. For example, $[8, 19, 30, 41]$ is an arithmetic sequence with stride 11 because $19 - 8 = 30 - 19 = 41 - 30 = 11$.

Suppose that, given a list, you need to find a maximal arithmetic sequence that indexes strictly increasing terms. For instance, from the list $[70, 20, 60, 30, 40, 10, 0, 50]$, the values 20, 40, and 50 are three values in increasing order that are evenly spaced in the list (appearing at indices 1, 4, and 7), but there is no way to select four such terms. That means that $[1, 4, 7]$ is a longest suitable arithmetic sequence of indices. This problem can be solved by exhaustive search.

Complete the JavaScript function `findLongestSuitableSequence` in `subsequence.js` so that it takes in a list of numbers and returns a maximal arithmetic sequence that indexes strictly increasing terms from that list. For example, `findLongestSuitableSequence([70, 20, 60, 30, 40, 10, 0, 50])` should evaluate to `[1, 4, 7]`.

For your convenience, if you have a stride, an offset (an initial term), and a count of terms, you can turn those three numbers into an arithmetic sequence using JavaScript library functions as follows:

```
const arithmeticSequence = Array(count).fill().map(
  (_, index) => stride * index + offset,
);
```

Unit tests to check functional correctness are provided in `subsequence.test.js`, and you can run them using the `test-once:subsequence:correctness` NPM script.

A naïve exhaustive search that considers every possible arithmetic sequence will run slowly on most large inputs. To earn the additional points for performance, also find a way to reduce the search space so that the algorithm uses far fewer array accesses on typical inputs.

Unit tests to check performance (which also include specific bounds for problems of varying difficulty) are provided in `subsequence.test.js`, and you can run them using the `test-once:subsequence:performance` NPM script.

(Yes, it is possible to hide basic operations from the performance tests, for instance by copying the input list and working on the copy instead of the original. Do not do that; the graders will give you a zero for performance if they find that you have falsified performance results.)

Finally, use the project's built-in linter to check your code style. Any `eslint-disable` directives will be ignored by the autograder, so do not use them to hide warnings in your local copy, or you may miss code style problems that will cost you points.

For the main search, the search space is the possible arithmetic sequences of indices, and we can describe each candidate arithmetic sequence by its stride, offset (that is, its initial term), and number of terms (which we will call a "count" rather than a "length" to avoid any confusion with the list length). Without optimizing at all yet, the stride can be as small as one or as large as the list length (like when the sequence starts at index zero and then skips over everything else). Similarly, the offset could be any valid index into the list, and the count can range from zero (if the list is empty) to the length of the list. Hence, we can code the "generate" step of an exhaustive search with three nested `for` loops based on these ranges.

As for the "check" step, we have a mix of a "find-one satisfying" problem and a "find-one optimizing" problem: we are only looking for sequences that index increasing elements, but among those sequences we want the longest. So our conditional has two tests: does the sequence index increasing elements, and does it have more terms that the best found so far?

Altogether, using the suggested library functions for the return value, a completely unoptimized exhaustive search looks like this:

```
let bestStride = undefined;
let bestOffset = undefined;
let bestCount = 0;
for (let stride = 1; stride <= list.length; ++stride) {
  for (let offset = 0; offset < list.length; ++offset) {
    for (let count = 0; count <= list.length; ++count) {
      if (isIncreasing(stride, offset, count, list) &&
          count > bestCount) {
        bestStride = stride;
        bestOffset = offset;
        bestCount = count;
      }
    }
  }
}
return Array(bestCount).fill().map(
  (_, index) => bestStride * index + bestOffset,
);
```

But we still require an implementation of `isIncreasing`. As is typical with exhaustive search, our "check"-step is itself an exhaustive search: we are looking for any flaws in the candidate sequence, where a flaw could either be running out of list elements by trying to index past the end of the list or else indexing an element that is not an increase over the prior element. The "generate" step is therefore a loop over each index into the sequence (from which we can compute an index into the list by using the stride and offset as a slope and intercept, respectively), and the "check" step has two tests: one for going out-of-bounds and another for finding a decrease. We do need another loop variable to track the last-seen element in order to write the second test.

In JavaScript:

```
function isIncreasing(stride, offset, count, list) {
  let previous = -Infinity;
  for (let i = 0; i < count; ++i) {
    const index = offset + stride * i;
    if (index >= list.length || list[index] <= previous) {
      return false;
    }
    previous = list[index];
  }
  return true;
}
```

Now there are many optimization opportunities. From class, one trick in a find-one-optimizing search is to order the search space by quality so that the first-found solution is necessarily the best, allowing the search to break early, which can improve average-case performance. Here we could apply that trick by pulling the `count` loop to the very outside and running it in reverse order, from high (better) counts to low (worse) counts. However, we can do even better and actually improve even the worse case. As discussed in class, the way to do that is to reduce the size of the search space. In particular, can we narrow the range of some loop variable when we know values for the others?

We can; the first key insight is that there is no need to specify a count up-front. Since `isIncreasing` can already detect where the sequence goes bad, we can rework it to give the best possible count for any stride/offset combination, a revision shown below:

```
function getCount(stride, offset, list) {
  let previous = -Infinity;
  let count = 0;
  while (stride * count + offset < list.length &&
         list[stride * count + offset] > previous) {
    previous = list[stride * count + offset];
    ++count;
  }
  return count;
}
```

That lets us remove a whole layer of looping in the outer search:

```
for (let stride = 1; stride <= list.length; ++stride) {
  for (let offset = 0; offset < list.length; ++offset) {
    const count = getCount(stride, offset, list);
    if (count > bestCount) {
      // ...
    }
  }
}
```

Unfortunately, that improvement, while better, is still not enough to pass all of the performance tests. We cannot completely remove either of the surviving loops, so the question is whether we can narrow their bounds. *A priori*, not really. But while the search is running, we do have a little bit more information—we know the best count found so far.

Thus, the second key insight is that we can tell ahead of time, before calling `getCount`, what strides and offsets have no hope of fitting enough terms inside the list to beat our current best count. For example, if the stride is 2, the offset is 5, and the best count found so far is 3, then the shortest arithmetic sequence we care about is $[5, 7, 9, 11]$, where the last index is $2 \cdot 3 + 5 = 11$. If the list does not have at least 12 elements, then we need not bother looking. More generally, a stride/offset combination is worth checking if and only if `stride * bestCount + offset` is a valid index into list, which is to say that it is less than `list.length`.

Now, what we do not want to do is to check this condition inside the loop body because that will not actually cut down on the number of loop iterations. Instead, the loop termination conditions themselves should stop the loop when the stride or offset becomes too large, like this:

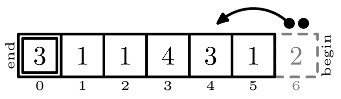
```
for (let stride = 1;
     stride * bestCount < list.length;
     ++stride) {
  for (let offset = 0;
       stride * bestCount + offset < list.length;
       ++offset) {
    // ...
  }
}
```

(We do not have an offset in the outer loop's termination condition, so we write the test to keep the loop running if any offset, even zero, could work.)

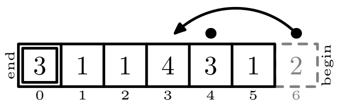
With this revision, the code performs well enough to pass the entire test suite.

3. (Graph Search, 1 point for the code linting, 9 points for functional correctness, 5 points for performance)

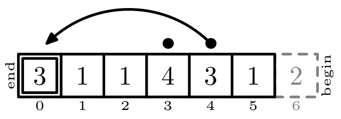
Consider a puzzle in the Backgammon family of board games where the goal is to move all of one's pawns exactly to the final space on the board in as few turns as possible, and the number of steps that a pawn takes each turn is determined by the square last landed on. For instance, if the board is marked with the numbers $[3, 1, 1, 4, 3, 1]$ from the end square back to the beginning, and the player begins with two pawns off the board and a two-step move, they can win in four turns by first advancing a pawn two steps:



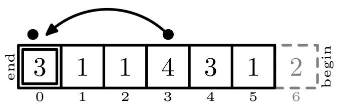
to land on a 3, then advancing the other pawn three steps:



to land on a 4, then advancing the first pawn four steps:



to land on the 3 at the end of the board, and finally advancing the remaining pawn three steps:



Such puzzles are planning problems and can therefore be solved by graph search.

Complete the `incidences` getter in the `Vertex` class in `boardGame.js` so that `solveBoardGame` takes in a list of board markings, a starting step count, and the number of pawns and returns a shortest solution as a list of indices to move pawns from. For example, `solveBoardGame([3, 1, 1, 4, 3, 1], 2, 2)` should evaluate to the list `[6, 6, 4, 3]` to describe the first pawn moving from index 6 (from off the board), the second pawn moving from index 6 (from off the board), the first pawn moving from index 4, and finally the second pawn moving from index 3.

Unit tests to check functional correctness are provided in `boardGame.test.js`, and you can run them using the `test-once:boardGame:correctness` NPM script.

The graph search can be sped up considerably by removing redundant graph edges and then using heuristic search. To earn the points for performance, revise the `incidences` getter so that it deduplicates edges, implement the `heuristic` getter so that it returns a more informative but still admissible heuristic, and update `solveBoardGame` to use that heuristic in its search.

Unit tests to check performance (which also include specific bounds for problems of varying difficulty) are provided in `boardGame.test.js`, and you can run them using the `test-once:boardGame:performance` NPM script. (Again, you will earn zero points if you hide basic operations from performance tests or otherwise falsify performance results.)

Finally, use the project's built-in linter to check your code style. Any `eslint-disable` directives will be ignored by the autograder, so do not use them to hide warnings in your local copy, or you may miss code style problems that will cost you points.

In this planning problem, each edge (action) is a choice of a space to move a pawn from, and the vertices (situations), as suggested by the starter code, are arrangements of pawns on the board paired with information about how many steps the next move must take.

From a game situation `this`, we can move any one pawn `this.stepCount` steps towards the final space, but with the restriction that we must not overshoot the end of the board. So the structure of a method to enumerate these possibilities will be a "find-all-satisfying" exhaustive search:

```
get incidences() {
  const results = [];
  for (const pawnLocation of this.pawnLocations) {
    if (pawnLocation >= this.stepCount) {
      // ...
      results.push(/* ... */);
    }
  }
  return results;
}
```

To compute a description of the edge, we need to know the three data listed in the starter-code comment: the action taken, the destination vertex, and the edge weight. The action taken is already in the variable `pawnLocation`, and every edge costs the same amount, one turn. But for the destination vertex, we need to give three values to the `Vertex` constructor: the board, the updated pawn locations, and the new step count. The board is unchanged, so we can just use `this.board`, but the other two values will need to be computed:

```
results.push({
  action: pawnLocation,
  child: new Vertex(this.board, /* ... */, /* ... */),
  cost: 1,
});
```

For the new pawn locations, we can make a copy of the current pawn locations, and reduce the old pawn location by `this.stepCount`. With that updated value, we can look up the corresponding number on the board:

```
const newPawnLocations = [...this.pawnLocations];
newPawnLocations[i] = pawnLocation - this.stepCount;
const newStepCount = this.board[newPawnLocations[i]];
```

However, this code does require us to know the index `i` of the pawn we are moving, so we have to convert the surrounding loop from a more natural for-each loop to a counter-based loop:

```
for (let i = 0; i < this.pawnLocations.length; ++i) {
  const pawnLocation = this.pawnLocations[i];
  // ...
}
```

Putting these parts together gives this basic `incidences` getter:

```
get incidences() {
  const results = [];
  for (let i = 0; i < this.pawnLocations.length; ++i) {
    const pawnLocation = this.pawnLocations[i];
    if (pawnLocation >= this.stepCount) {
      const newPawnLocations = [...this.pawnLocations];
      newPawnLocations[i] = pawnLocation - this.stepCount;
      const newStepCount = this.board[newPawnLocations[i]];
      results.push({
        action: pawnLocation,
        child: new Vertex(this.board, newPawnLocations, newStepCount),
        cost: 1,
      });
    }
  }
  return results;
}
```

Now, as suggested in the problem description, when multiple pawns are on the same space, it does not matter which of them we choose to move, so returning a distinct edge for each of them is wasteful. An easy fix, since the `Vertex` constructor sorts the pawn locations, is to remember the last pawn location used in an edge and refuse to create another edge whenever we would be duplicating a result:

```
get incidences() {
  const results = [];
  let previousPawnLocation = undefined;
  for (let i = 0; i < this.pawnLocations.length; ++i) {
    const pawnLocation = this.pawnLocations[i];
    if (pawnLocation !== previousPawnLocation &&
        pawnLocation >= this.stepCount) {
      previousPawnLocation = pawnLocation;
      // ...
    }
  }
  return results;
}
```

As for the heuristic, we can incorporate it into the graph search by switching from a BFS to a best-first search, giving a heuristic arrow function as the second argument to match the signature in `graphSearch.js`:

```
const edges = bestFirst(
  new Vertex(board, Array(pawnCount).fill(board.length), initialStepCount),
  (vertex) => vertex.heuristic,
  (vertex) => vertex.isDestination,
);
```

And then we just need to be able to compute a quantity that will approximate but never overestimate the number of moves still left to play. One simple idea is that each pawn not on the final space still needs at least one move, so we could count the number of pawns not yet at index zero:

```
get heuristic() {
  let result = 0;
  for (const pawnLocation of this.pawnLocations) {
    result += pawnLocation > 0 ? 1 : 0;
  }
  return result;
}
```

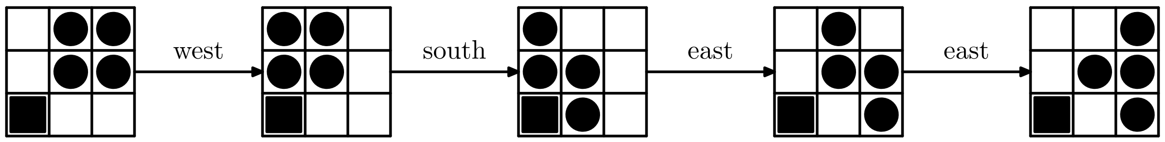
Unfortunately, this only works on some performance test cases, and it especially has trouble when the numbers on the board are much less than the length of the board—when most of the pawns will need multiple moves to finish. Therefore, a slightly more accurate heuristic would compute for each pawn how many moves it needs if by luck it can move as fast as possible given the numbers on the board:

```
get heuristic() {
  const fastest = Math.max(...this.board);
  let result = 0;
  for (const pawnLocation of this.pawnLocations) {
    result += Math.ceil(pawnLocation / fastest);
  }
  return result;
}
```

Other ideas are possible, but this heuristic is admissible and does indeed pass the performance tests.

4. (Graph Search, 1 point for the code linting, 14 points for functional correctness)

Consider a grid-movement puzzle where each turn the player moves all unblocked pawns one step in one of the four cardinal directions (north, south, east, or west), and the goal is to arrange them in a specific shape. For instance, on a 3×3 grid with the southwest corner blocked off, a square of four pawns in the northeast can be rearranged into a left-facing turnstile in the east by playing four moves: "west", "south", "east", and "east" again:



(Notice how pawns do not move when the edge of the grid, a block, or another blocked pawn is in the way.)

Such puzzles are planning problems and can therefore be solved by graph search.

Using the provided helper constants, function, and methods, complete the `incidences` getter in the `Vertex` class in `gridGame.js` so that `solveGridGame` takes in initial and final grids as strings and returns the solution as a list of strings. For example, `solveGridGame('□●●\n□●●\n■□□', '□□●\n□●●\n■□●')` should evaluate to the five-element list `['□●●\n□●●\n■□□', '●●□\n●●□\n■□□', '●□□\n●●□\n■●□', '□●□\n□●●\n■□●', '□□●\n□●●\n■□●']` where each string corresponds to one of the grids in the diagram on the previous page.

For your convenience, to make a new grid of cells by removing all pawns from `this.cells`, you can use JavaScript library functions as follows:

```
const newCells = this.cells.map(
  (row) => row.map(
    (cell) => cell === WALL ? WALL : FLOOR,
  ),
);
```

Unit tests to check functional correctness are provided in `gridGame.test.js`, and you can run them using the `test-once:gridGame:correctness` NPM script.

Finally, use the project's built-in linter to check your code style. Any `eslint-disable` directives will be ignored by the autograder, so do not use them to hide warnings in your local copy, or you may miss code style problems that will cost you points.

In this planning problem, each edge (action) is a choice of a direction to move all pawns, and the vertices (situations) are arrangements of pawns on the board. So the structure of a method to enumerate edges out of a game situation `this` will be a loop over directions where, for each direction, we clear the board, place the updated pawns, and then push a result object:

```
get incidences() {
  const results = [];
  for (const direction of DIRECTIONS) {
    const newCells = this.cells.map(
      (row) => row.map(
        (cell) => cell === WALL ? WALL : FLOOR,
      ),
    );
    // ...
    results.push({
      child: new Vertex(newCells),
    });
  }
  return results;
}
```

To place the updated pawns, we first have to be able to loop over the original pawns. Although there is no helper function to do so, there is a helper method `getLocations` that generates all locations in the grid and another helper method `at` that lets us check what is at a location, so we can write code to test for pawns:

```
for (const location of this.getLocations()) {
  if (this.at(location) === PAWN) {
    // ...
  }
}
```

In ordinary circumstances, a pawn will be free to move, so its new location will be its old location shifted by the direction of movement:

```
const movedLocation = shift(location, direction);
```

But if it will bump into a pawn that will bump into a pawn that will ... that will bump into a wall, then the pawn cannot move. So we need a loop to trace through a line of pawns and find the location the line is trying to push into:

```
let candidate = movedLocation;
while (this.at(candidate) === PAWN) {
  candidate = shift(candidate, direction);
}
```

Then, based on whatever is at that location, we can choose to keep the pawn in place or move it to `movedLocation`. Based on our choice we destructure out x and y coordinates and update `newCells` accordingly:

```
const [newX, newY] = this.at(candidate) === WALL ?
  location : movedLocation;
newCells[newY][newX] = PAWN;
```

(Notice that the y coordinate goes first because our cells array is indexed by row, then column, and the y coordinate chooses a row.)

Putting everything together, the following incidences getter solves the problem:

```
get incidences() {
  const results = [];
  for (const direction of DIRECTIONS) {
    const newCells = this.cells.map(
      (row) => row.map(
        (cell) => cell === WALL ? WALL : FLOOR,
      ),
    );
    for (const location of this.getLocations()) {
      if (this.at(location) === PAWN) {
        const movedLocation = shift(location, direction);
        let candidate = movedLocation;
        while (this.at(candidate) === PAWN) {
          candidate = shift(candidate, direction);
        }
        const [newX, newY] = this.at(candidate) === WALL ?
          location : movedLocation;
        newCells[newY][newX] = PAWN;
      }
    }
    results.push({
      child: new Vertex(newCells),
    });
  }
  return results;
}
```