GardenBuddy - Optimierungen

Qualifikationsschritt M2 (HSLU CAS MSED)

Gallus Bühlmann gallus.buehlmann@stud.hslu.ch
Pascal Kiser pascal.kiser@stud.hslu.ch

18.02.2024

Inhaltsverzeichnis

1 Ausgangslage

Im Rahmen der Transferarbeit für den CAS *«Modern Software Engineering & Development»* haben wir eine Webapplikation für Hobbygärtner entwickelt. Die Applikation soll den Gärtner insbesondere bei der Planung der zu beflanzenden Gartenbeete unterstützen. Die Idee ist grundsätzlich die folgende:

- Der Benutzer erfasst seinen Garten im App (Anzahl und Grösse der Beete)
- Daraus ergibt sich wie viele Plätze im Garten verfügbar sind, um Gemüse anzupflanzen
- Der Benutzer wählt anschliessend Gemüsesorten aus, bis alle Plätze besetzt sind. Wiederholungen sind erlaubt, d.h. der Benutzer kann eine Gemüsesorten auch mehrmals auswählen.
- Auf Knopfdruck gibt das System dann einen oder mehrere Vorschläge aus, wie die vom Benutzer ausgewählten Gemüsesorten idealerweise angeordnet werden sollten. Es sollen alle vom Benutzer gewählten Gemüsesorten in der vom Benutzer vorgegebenen Anzahl verwendet werden, auch wenn dadurch keine “gute” Lösung möglich ist. Das Resultat ist einfach die bestmögliche Anordnung basierend auf dem Input.

Die zur Auswahl stehenden Gemüsesorten und die Daten dazu kommen aus einer Datenbank und die Berechnung der optimalen Anordnung basiert auf verschiedenen vordefinierten Regeln, welche für den Benutzer nicht zwingend ersichtlich oder komplett nachvollziehbar sein müssen.

Ziel der Transferarbeit war es, mit wenigen Gemüsesorten und wenigen Regeln einen ersten, einfachen aber erweiterbaren Prototyp zu bauen, dem später weitere Regeln hinzugefügt werden können. Ziel dieser Arbeit ist es, das aktuell grösste Problem des Prototyps zu analysieren und einige Erkenntnisse zu gewinnen, wie damit umgegangen werden kann.

1.1 Grundlagen der Gartenplanung

Ein Gartenbeet besteht (in der Regel) aus mehreren Reihen, welche wiederum mit verschiedenen Gemüsesorten bepflanzt werden können. Das Wachstum der Pflanzen hängt dabei von zahlreichen verschiedenen Faktoren ab, wobei die meisten davon mit der Position innerhalb eines Gartenbeets zu tun haben:

1) Direkte Nachbarschaftbeziehungen:

Einige Pflanzen beeinflussen das Wachstum ihrer Nachbarn positiv, andere wirken sich negativ auf bestimmte Nachbarn aus. Die Lösung für dieses Problem sind sogenannte *Mischkulturen*: der gleichzeitige Anbau verschiedener Gemüsesorten, die sich idealerweise gegeneitig begünstigen.

Als Veranschaulichung soll hier ein Satz aus der Einleitung unserer Transferarbeit dienen:

> Die Karotte und die Zwiebel beispielsweise vertreiben mit ihrem Duft jeweils die Schädlinge der anderen Pflanze: die Möhrenfliege wird vom Zwiebelduft vertrieben und die Zwiebelfliege vom Möhrenduft.
>
> — *Bühlmann & Kiser (2024)*[1]

2) Nährstoffbedarf und Bodenqualität:

Verschiedene Gemüsesorten unterscheiden sich auch durch ihren Nährstoffbedarf. Wenn mehrere Jahre hintereinander Gemüsesorten mit ähnlichem Bedarf wachsen, nimmt die Bodenqualität ab und der Ertrag sinkt.

Die Lösung für dieses Problem wird als *Fruchtfolge* bezeichnet: verschiedene Pflanzengattungen werden in einer mehrjährigen Rotation an einer bestimmten Stelle angepflanzt, was dazu führt, dass der Boden sich besser erholen kann.

[1] Bühlmann G., & Kiser P. (2024), *Prototyp GardenBuddy eine Machbarkeitsstudie*, Transferarbeit CAS MSED HSLU

3) Weitere Faktoren:

Bestimmte Pflanzen haben noch zusätzliche Vorlieben, wenn es um ihre Position im Gartenbeet geht. Kürbisse beispielsweise tendieren dazu, gegen Süden zu wachsen und die dort angesiedelten Pflanzen zu verdrängen. Deshalb sollten sie vorzugsweise am südlichen Rand eines Beets gepflanzt werden.

1.2 Schwierigkeiten der Gartenplanung

Für Fruchtfolgen, Mischkulturen usw. gibt es zahlreiche verschiedene Modelle und Ansätze, wie verschiedene Gemüsesorten kombiniert werden sollen oder eben nicht. Wenn sich der Gärtner für ein entsprechendes Modell entschieden hat, oder seine eigene Kombination verschiedener Modelle ausgearbeitet hat, fängt die eigentliche Planungsarbeit erst richtig an: auf Papier werden Beete gezeichnet und Gemüse platziert, Nachbarschaftsbeziehungen in Büchern nachgeschlagen und Gemüse umplatziert, bis der erschöpfte Hobbygärtner mit seinem Plan zufrieden ist.

Ob er den optimalen Plan ausgearbeitet hat, wird der Gärtner auf diese Weise vermutlich nie herausfinden, sofern sein Garten nicht grösser ist als ein paar wenige Reihen. Grund dafür ist, dass die Zahl der möglichen Anordnungen (Permutationen) mit zunehmender Anzahl Elemente stark zunimmt.

Die Anzahl Permutationen für n Elemente lässt sich folgendermassen berechnen, wenn verschiedene Elemente (Gemüsesorten) mehrfach vorkommen (Permutation mit Wiederholung):

$$P(n) = \frac{n!}{k_1! \cdot ... \cdot k_s!}$$

P bezeichnet hier die Permutationen von n Elementen, von denen k identisch sind.

Oder, im schlechtesten Fall, also wenn alle Elemente unterscheidbar sind (Permutation ohne Wiederholung):

$$P(n) = n!$$

Bei zwei oder drei Beet-Reihen kann man sich die möglichen Kombinationen vielleicht noch im Kopf durchdenken, aber ab vier ist man vermutlich auf ein Papier angewiesen, ab fünf braucht man bereits die Rückseite und 6 oder mehr Reihen kann man ohne Hilfsmittel innerhalb einer Gartensaison nicht vernünftig planen.

Im Rahmen der Transferarbeit haben wir einen Prototyp für ein solches Hilfsmittel ausgearbeitet.

n	$n!$
1	1
2	2
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600
13	6227020800

1.3 Einschränkungen bezüglich Optimierungen

Eine naheliegende Optimierung wäre hier, die Anzahl der zu verarbeitenden Permutationen zu reduzieren, beispielsweise indem man gewisse Anordnungen bereits zu Beginn verwirft. In der aktuell umgesetzten Variante werden nur Nachbarschaftsbeziehungen berücksichtigt, was dazu führt, dass gespiegelte Anordnungen (`abc` und `cba`) auch immer die gleiche Punktzahl haben. Weil aber in Zukunft weitere Regeln dazukommen sollen, welche eben nicht symmetrisch sind (Nord-Süd-Ausrichtung, Fruchtfolge etc.) wurde darauf verzichtet. Falls zu einem späteren Zeitpunkt klar werden sollte, dass gewisse Kombinationen tatsächlich nie berücksichtigt werden müssen, können diese dann immer noch ausgefiltert werden. Aktuell gehen wir aufgrund der Erweiterbarkeit des Regelsets zur Punkteberechnung vom “Worst-Case” aus, d.h. alle möglichen Permutation müssen berücksichtigt werden.

Grundsätzlich können “schlechte” Paare, also Kombinationen innerhalb einer Permutation mit besonders negativen Beziehungen untereinander können auch nicht einfach aussortiert werden. Eventuell gibt es ja in dieser Benutzerauswahl nur schlechte Beziehungen, trotzdem sollte das bestmögliche Resultat angezeigt werden. Oder das “schlechte Pärchen” ist nötig, um ein anderes, “gutes” Pärchen zu ermöglichen. Allenfalls liessen sich die einzelnen Permutationen anders priorisieren, d.h. Permutation mit schlechten Kombinationen weiter nach hinten in die Verarbeitungsqueue geschoben. Die effektive Anzahl zu verarbeitender

Permutation lässt sich so aber vermutlich nicht sinnvoll reduzieren, ohne dass die Erweiterbarkeit der Regeln für die Punkteberechnung beeinträchtigt wird.

Aus diesen Gründen wird nachfolgend von der Annahme ausgegangen, dass jeweils alle Permutationen betrachtet werden müssen.

2 Implementierung

Der oben erwähnte Prototyp kann bereits einiges. Ein Benutzer kann sich registrieren und einloggen, Gemüsebeete erfassen, die Anzahl Reihen pro Beet konfigurieren und verschiedene Gemüsesorten auswählen. Per Knopfdruck wird dann die ideale Zusammenstellung berechnet und dem Benutzer angezeigt.

Aktuell werden zwar nur die Nachbarschaftsbeziehungen berücksichtigt, aber das Datenmodell ist so konzipiert, dass später weitere Faktoren relativ einfach ergänzt werden könnten.

Aufgrund der Implementierung kommt das System aber relativ schnell an seine Grenzen. Je nach Konfiguration der Umgebung können momentan etwa 9 bis 10 Beetreihen berechnet werden. Grund dafür ist die Art, wie die Berechnung des optimalen Beets aktuell gemacht wird. In Pseudocode sieht das in etwa so aus:

```
function getOptimalBed(input)
    allBeds = permutations(input)
    foreach bed in allBeds
        score = 0
        foreach veggie in bed
            score += calculateScore(veggie)
    sort(allBeds)
    return allBeds[0]
```

Die Erstellung einer Liste mit allen Permutation kann relativ einfach rekursiv gelöst werden:

```
function permutations(lst):
    if length(lst) == 0:
        return [[]]
    result = []
    for i from 0 to length(lst) - 1:
        current_element = lst[i]
        remaining_elements = concatenate(lst[0:i], lst[i+1:])
        for p in permutations(remaining_elements):
            result.append(concatenate([current_element], p))
    return result
```

2.1 Variante 1: PHP

Das Backend der Webapplikation ist mit dem PHP-Framework Laravel umgesetzt, weshalb die erste Version direkt als Laravel Controller in PHP geschrieben wurde. Die Input-Daten kommen über einen POST Request (als Liste von Gemüse-IDs) rein und der Controller gibt als Response die Permutation mit der höchsten Punktzahl zurück.

Die Funktion zum erstellen aller Arrays sieht im Projekt etwa so aus:

```php
function permRec($array)
{
    if (count($array) ⩽ 1) {
        return [$array];
    }
    $result = [];
    foreach ($array as $key ⇒ $element) {
        foreach (permRec(array_diff_key(
                    $array,[$key ⇒ $element])) as $perm) {
            $result[] = array_merge([$element], $perm);
        }
    }
    return $result;
}
```

PHP hat eine Einstellung `max_nesting_level`, welche hier relativ schnell zu einem Problem wurde. Diesen Wert kann man in der Konfiguration natürlich beliebig erhöhen. Dies ist aber nur bedingt eine Lösung.

Wenn man die Permutationen nicht mehr rekursiv, sondern iterativ erstellt, sieht das in PHP etwa so aus[2]:

[2] Wikipedia. (2024, Januar 31). Heap's algorithm. https://en.wikipedia.org/wiki/Heap%27s_algorithm

```php
function permIt($array)
{
    $result = [];
    $count = count($array);
    $c = array_fill(0, $count, 0);
    $result[] = $array;

    $i = 0;
    while ($i < $count) {
        if ($c[$i] < $i) {
            if ($i % 2 == 0) {
                $temp = $array[0];
                $array[0] = $array[$i];
                $array[$i] = $temp;
            } else {
                $temp = $array[$c[$i]];
                $array[$c[$i]] = $array[$i];
                $array[$i] = $temp;
            }

            $result[] = $array;
            $c[$i]++;
            $i = 0;
        } else {
            $c[$i] = 0;
            $i++;
        }
    }
    return $result;
}
```

Das ist schon wesentlich schwerer zu lesen, aber führt nicht mehr zu Problemen mit der Rekursionstiefe. Der limitierende Faktor ist jetzt der Arbeitsspeicher: PHP hat auch dazu eine Konfiguration (`memory_limit`).

Verschiedene Versuche mit dem iterativen und rekursiven Ansatz können mit dem Skript `source/php/perm.php` nachvollzogen werden.

Auf Zeilen 2 bis 5 sind die oben erwähnten Konfigurationen zu Nesting Level und Memory Limit:

```php
<?php
//ini_set('xdebug.max_nesting_level', 100);
ini_set('xdebug.max_nesting_level', -1);
ini_set('memory_limit', '256M');
// ini_set('memory_limit', '4096M');
```

Das Skript kann folgendermassen aufgerufen werden:

```
php perm.php MODE SIZE
```

- `MODE`: Entweder `R` für die rekursive Variante, oder `I` für iterativ.
- `SIZE`: Grösse des Input-Arrays

Zum Beispiel:

```
$ php perm.php R 9
R: 362880 permutations in 0.107521011 seconds
$ php perm.php I 9
I: 362880 permutations in 0.517928187 seconds
$ php perm.php I 10
I: 3628800 permutations in 5.956723439 seconds
```

Wenn das Memory Limit hoch genug gesetzt wird, dann steigt naheliegenderweise die Verarbeitungzeit, was irgendwann zu einem Timeout beim POST Request in Laravel führt.

2.2 Variante 2: Rust

Eine einfache Lösung wäre die Verwendung einer Programmiersprache, die etwas performanter ist als PHP, wie beispielsweise Rust. Die Resultate sind durchaus interessant, aber nur bedingt hilfreich:

```
$ ./perm R 10
R: 3628800 permutations in 9.187155496 seconds
$ ./perm I 10
I: 3628800 permutations in 0.330772779 seconds
$ ./perm I 11
I: 39916800 permutations in 3.913299555 seconds
$ ./perm I 12
I: 479001600 permutations in 60.636729915 seconds
```

Der iterative Ansatz schneidet hier besser ab, was vermutlich auf bessere Compiler-Optimierungen zurückzuführen ist. Grundsätzlich scheint das aber noch nicht die Lösung für unser Problem zu sein.

Der Code dazu ist im Verzeichnis `source/rust` abgelegt.

Die Erkenntnisse aus Variante 1 und 2 sind somit:

- Mit der Zuweisung von mehr Arbeitsspeicher (Variante 1) oder der effizienteren Verwendung des Arbeitsspeichers (Variante 2) können nur minimale Verbesserungen erzielt werden.
- Die maximale Grösse des Inputs ist limitiert durch die Grösse des resultierenden Arrays, welches alle Permutation enthält.
- Mit grösseren Inputs steigt die Verarbeitungszeit stark an, weshalb eine synchrone Verarbeitung des Requests vermutlich kein geeigneter Ansatz ist.

Für unsere Implementierung bedeutet das, dass wir an verschiedenen Punkten ansetzen können:

1. Parallelisierung, was aber bei diesem Algorithmus nicht ganz einfach ist, aufgrund der Abhängigkeiten zwischen den Iterationen.
2. Es müssen nicht zwingend alle Permutationen auf einmal erstellt und in einer Variable abgelegt werden. Die einzelen Permutation sollten einzeln oder Batch-weise erzeugt und verarbeitet werden. Dadurch ergeben sich auch wieder weitere Möglichkeiten zur Palallierung.

2.3 Variante 3: Haskell

Eine Möglichkeit, Werte aus einer langen (oder sogar unendlichen) Liste zu verarbeiten, ist *Lazy Evaluation*. Gemeint ist damit eine Art der Auswertung von Ausdrücken, bei denen das Ergebnis nur und erst dann berechnet wird, wenn es benötigt wird. Dadurch wird es beispielsweise möglich eine unendlich lange Liste zu verarbeiten.

Eine Programmiersprache, die bekannt dafür ist, Lazy Evaluation einzusetzen ist Haskell.

Der interessante Teil des Codes sieht so aus:

```
import Data.List (permutations)
import Data.Time.Clock (getCurrentTime, diffUTCTime)
import Text.Printf

-- Define a lazy stream of permutations
permStream :: [a] → [[a]]
permStream = permutations

let inputList = [1..lengthInput]
let allPerms = permStream inputList
let count = length allPerms
putStr $ "λ: " ++ show count ++ " permutations"
stop <- getCurrentTime
let delta = diffUTCTime stop start
let timeTaken = realToFrac delta :: Double
putStrLn $ " in " ++ printf "%.8f" timeTaken ++ " seconds"
```

Die Umsetzung war erstaunlich einfach, sogar für jemanden der bisher noch nie mit Haskell gearbeitet hat. Das einzige, was nicht auf Anhieb funktionierte, war die Messung der Zeit in Sekunden. Wenn die End-Zeit wie bei den anderen Implementation direkt nach dem Erstellen der Permutationen gemessen wird, dann bekommt man kein sinnvolles Resultat. Grund dafür ist, dass hier der Stream eben nur für die Ausgabe von `count` benötigt wird und deshalb erst dann ausgewertet.

Die Resultate sind insofern interessant, als dass es nicht mehr zu einem Abbruch kommt:

```
$ ./perm 11
λ: 39916800 permutations in 0.79687400 seconds
$ ./perm 12
λ: 479001600 permutations in 8.75207300 seconds
$ ./perm 13
```

```
λ: 6227020800 permutations in 121.11810000 seconds
$ ./perm 13
λ: 87178291200 permutations in 1871.11263400 seconds
```

Egal wie gross der Input ist, das Progamm wird nicht mehr abstürzen wie bisher. Damit ist die Verarbeitung zwar nicht schneller als vorher, aber zumindest in der Länge des Inputs theoretisch nicht mehr durch Memory-Limitationen beschränkt.

3 Fazit

Die Limitation betreffend Rekursionstiefe, Arbeitsspeicher und Verarbeitungszeit sind im Kontext von Laravel und PHP durchaus sinnvoll. Ein einzelner Request ans Backend sollte idealerweise nicht alle Ressourcen des Servers beanspruchen können. Eine synchrone Verarbeitung des Requests ist aufgrund der langen Verarbeitungszeit ebenfalls nicht ideal.

Mit Lazy Evaluation oder einer Art Generator-Funktion, welche nur die gerade benötigen Permutation erzeugt und zur Verarbeitung übergibt, können Probleme mit begrenztem Arbeitsspeicher gelöst werden, aber die Verarbeitungszeit ist bei grösseren Input immer noch hoch.

Durch die Verarbeitung der Permutationen als Lazy List oder als Stream ergeben sich aber einige Möglichkeiten, um die User Experience für den auf Resultate wartenden Gärtner zu verbessern. Resultate können beispielsweise in einen Key-Value-Store geschrieben werden und und dem User alle paar Sekunden die aktuell beste Lösung angezeigt werden. Wenn eine Lösung “gut genug” ist, kann die Verarbeitung auch durch den Benutzer beendet werden.

Die Verarbeitung längerer Listen auf diese Art bietet auch weitere Optimierungsmöglichkeiten, indem die vorzu erzeugten Permutation beispielsweise in eine Queue geschrieben und von mehreren Consumern parallel verarbeitet werden (den Score berechnen). Dadurch lässt sich zumindest ein Teil der Berechnung sehr einfach parallelisieren.

Insgesamt scheint das ein interessanter und vielversprechender Ansatz zu sein, welcher für diese Art von Problem zumindest vielversprechend ist.