Aufgabe 2
Missglückte Drohnenlieferung

Robert Hönig
Teilnahme-ID:5776

11. April 2016

Inhaltsverzeichnis

1 Lösung

1.1 Idee

1.1.1 Formalisierung

Bei dieser Aufgabe handelt es sich um ein Puzzle: Die Siedlung Amacity befindet sich mit ihrer jeweiligen Paketverteilung in einer bestimmten Konfiguration. Ziel ist es nun, diese Konfiguration in eine andere zu überführen, wobei bestimmte Operationen zur Verfügung stehen. Die Gesamtheit der für die jeweilige Überführung verwendeten Operationen bildet die Lösung. Diese Lösung ist genau dann optimal, wenn sie die geringste Menge an Operationen benötigt. Ein Algorithmus, der so eine optimale Lösung berechnen kann, wird "Gottes Algorithmus"genannt. Die Aufgabe ist es nun, so einen, oder zumindest eine Annäherung an so einen Algorithmus für das Paketverteilungsproblem zu finden, der sowohl von der Zeit- als auch von der Platzkomplexität anwendbar für eine Siedlung der Größe $10*10$ ist. Dafür ist es hilfreich, sich das Puzzle und seine Lösungswege als einen großen Baum vorzustellen. Jeder Knoten k stellt eine Paketverteilung der Siedlung dar, seine Kinder ergeben sich aus der Anwendung aller möglichen Operationen auf k. Diese Menge ist bei diesem Problem gigantisch: Da von fast jedem Feld (Dach) aus sowohl möglich ist, das Paket in einer Operation an eines der vier angrenzenden Felder zu werfen und von einem dieser auch ein Paket zu bekommen, als auch nichts zu tun, lässt sich als obere Schranke für die Menge an Kindern die Zahl $(4*4+1)^{10*10} \approxeq 10^{123}$ festsetzen. Um die untere Schranke zu bestimmen, kann man davon ausgehen, dass es zumindest für alle 50 horizontalen Zweierpaare der Felder möglich ist, entweder gegenseitig ihr Paket auszutauschen oder nicht. Deswegen hat k mindestens 2^{50} Kinder, was einen vollständigen Durchlauf des Baumes praktisch unmöglich macht. Eine andere Möglichkeit, nämlich einfach die Überführung der möglichen Konfigurationen ineinander zu berechnen, ist auch nicht möglich: Schließlich gibt es 100! verschiedene Möglichkeiten, die Pakete auf die Felder zu verteilen. Deswegen habe ich mich zunächst auf einen Algorithmus konzentriert, der überhaupt in der Lage ist, das Problem zu lösen und die Pakete richtig zu verteilen. Dieses Verfahren wird anschließend verbessert und durch eine Heuristik unterstützt.

1.1.2 Eine suboptimale Lösung

Die Idee dieser Lösung besteht darin, die Pakete nacheinander auf das richtige Feld zu bewegen. Dabei gehen wir so vor, dass ein sich noch nicht auf dem richtigen Feld f befindliches Paket p ausgewählt wird. Anschließend wird es über Swap-Operationen auf f bewegt. Eine Swap-Operation bezeichnet den gegenseitigen Austausch der Pakete zweier benachbarter Felder. Die Bewegung zu f folgt dabei einer Regel, die bestimmt, ob als nächstes der horizontale oder vertikale Unterschied von p zu f ausgeglichen werden werden soll. So hat eine Platzierung von p nördlich von f höchste Priorität. Darauf folgen eine östliche, südliche und schließlich westliche Platzierung. Deswegen bewegt sich das in Abbildung 1 dargestellte grüne Paket erst nach Süden und dann nach Osten, um

Bewegung eines Paketes auf das richtige Feld

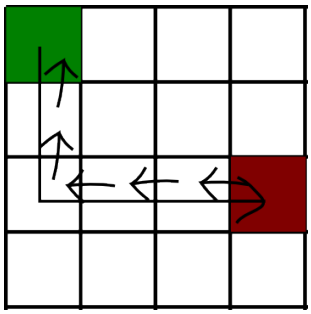


Abbildung 1: Um das gelbe Paket auf sein richtiges Feld zu bewegen, müssen alle auf dem Pfad liegenden Pakete um ein Feld verschoben werden.

Theorethisch infinite Kreisplatzierung

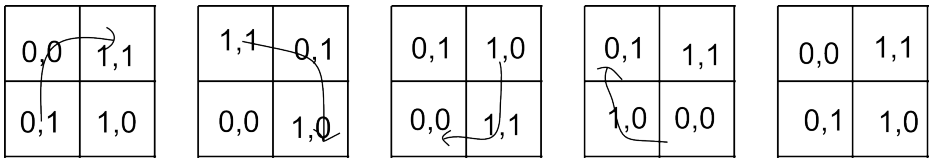


Abbildung 2: Da in diesem Beispiel jede richtige Platzierung eines Paketes die eines anderen verschiebt, wird eine Lösung niemals erreicht.

das rote Zielfeld zu erreichen. Diese Regel ist wichtig, um die dabei durch die kleinen Pfeile verdeutlichten Verschiebungen der Pakete auf dem Pfad wieder zurückbewegen zu können. Das gesamte Verfahren funktioniert ja nur unter der Annahme, dass einmal richtig platzierte Pakete nicht wieder verschoben werden können. Wäre das nicht der Fall, könnte die Platzierung aller Pakete wesentlich länger dauern oder sogar theoretisch nie enden, was das Beispiel in Abbildung 2 demonstriert. Zwar wird in jedem Schritt ein Paket über die für es kürzeste Manhattan-Distanz platziert, doch verschiebt seine Platzierung die Platzierung der anderen Pakete, sodass man letzten Endes wieder bei der Ausgangsposition angelangen kann. Damit also die verschobenen Pakete wieder „zurückverschoben“ werden, bewegt man einfach das rote Paket über denselben Pfad, den das grüne Paket genommen hat, auf das grüne Feld. Dadurch werden alle Swaps auf dem Pfad invertiert, also rückgängig gemacht. De facto wird nach der Ausführung beider Pfade nur das rote Paket verschoben, was jedoch keine Rolle spielt: Es hätte gar nicht richtig platziert sein können, da das grüne Paket auf sein Feld muss. Die Anzahl der benötigten Schritte vergrößert sich mit dem inversen Pfad höchstens um 1, da beide Pfade gleichzeitig begangen werden können. Um nun alle Pakete richtig zu platzieren, führen wir dieses Verfahren solange aus, bis kein Paket mehr auf dem flaschen Feld steht. Dieser Algorithmus ist determinierend, da nach jedem Durchlauf mindestens ein weiteres Paket zusätzlich

Mögliche Parallelisierung

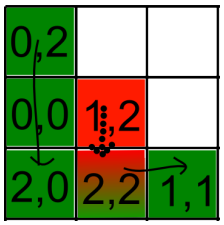


Abbildung 3: Die grünen Pfade können parallel ausgeführt werden, jedoch nicht zusammen mit dem roten Pfad.

platziert wurde. Um die Anzahl an benötigten Schritten zu verringern, können wir uns die Tatsache, dass beliebig viele Pakete innerhalb eines Schrittes das Feld wechseln können, zunutze machen. Wir können beliebig viele Pakete gleichzeitig auf ihr richtiges Feld bewegen, solange sich die Pfade nicht kreuzen. Würden sie sich kreuzen, bestünde die Gefahr, dass das Paket auf dem Kreuzungsfeld nicht wieder dahin zurückverschoben wird, da es zwei Mal verschoben wird. Unter Umständen würden die inversen Operationen dann ein anderes Paket zurückverschieben. Aus diesem Grund wählt der Algorithmus beliebig noch richtig zu platzierende Felder aus und markiert die von ihrem Pfad passierten Felder in einer Matrix. Das nächste unplatzierte Paket wird nur platziert, wenn der dafür nötige Pfad kein schon markiertes Feld enthält. Wurden alle gleichzeitig ausführbaren Pfade so gewählt, werden sie parallel abgelaufen. Eine mögliche Auswahl sieht man in Abbildung 3: Die Pfade von (0,0)->(0,2) und (1,2)->(2,2) können gleichzeitg ausgeführt werden, nicht jedoch noch zusätzlich (1,1)->(1,2). Nachdem der längste Pfad abgelaufen wurde, wird der Vorgang wiederholt, solange, bis alle Pakete richtig platziert sind.

1.1.3 Heuristik auf Basis dynamischer Programmierung

Das obige Verfahren erzeugt zwar eine Lösung, es ist jedoch noch nicht optimal. Mithilfe eines Greedy-Verfahrens kann man zumindest die Ausgangsbedingungen für das vorherige Verfahren verbessern. Greedy deswegen, da dieses Verfahren darauf basiert, in jedem Schritt die Manhattan-Distanz möglichst vieler Pakete zu ihrem Zielfeld zu verringern. Hierbei stehen zwei verschiedene Paket-Austauschverfahren zur Verfügung. Das einfachste ist ein einfacher Swap zweier Pakete zwischen zwei Feldern. Die zweite Variante ist ein Austauschkreis, in dem alle Pakete ein Feld weiter gereicht werden und der beliebig komplex werden kann. Beide Möglichkeiten sind in Abbildung 4 dargestellt. Da eine Generierung beliebiger Kreise die Laufzeit jedoch selbst bei einem $10 * 10$ Feld ins Unpraktikable nach oben schnellen lassen würde, wird im Folgenden nur ein Verfahren beschrieben, die Verteilung von ausschließlich Swap-Operationen mit der minimalen Summe der Manhattan-Distanzen aller Pakete zu ihrem Ziel zu berechnen. Um nicht alle verschiedenen möglichen Austauschvarianten durch-

Austauschmöglichkeiten

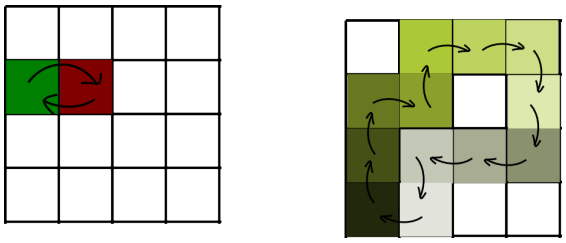


Abbildung 4: Links sieht man den Swap, rechts einen möglichen Austauschkreis

probieren zu müssen, verwenden wir dynamische Programmierung und speichern die Ergebnisse für die jeweils optimale Austauschvariante für jedes Feld. Feld (r,c) enthält diese Informationen dann über eine Siedlung, die aus den Reihen 0 - r besteht, wobei die letzte Reihe jedoch nur bis Spalte c geht. Möchte ich nun diese optimale Austauschvariante für Feld (r, c) berechnen, muss ich aus drei Möglichkeiten die mit der geringsten Manhattan-Distanz, bzw. der maximalen Verringerung mV der Manhattan-Distanz auswählen. Möglichkeit 1 ist, Feld (r,c) nicht in einen Swap einzubauen. In diesem Fall ist mV von (r,c) gleich dem mV seines Vorgängers, ein Wert, der einfach übernommen werden kann. Möglichkeit 2 und 3 bestehen darin, das Paket auf Feld (r,c) entsprechend mit dem Paket auf (r, c-1) bzw. (r-1, c) zu tauschen. mV wäre dann gleich der gewonnen Verringerung aus dem Swap + die mV der Siedlung von (r,c-1) ohne das Feld, mit dem getauscht wird. Dieses darf ja schließlich nicht bereits verwendet werden. Die drei Möglichkeiten werden in Abbildung 5 dargestellt. Um diese Information zu bestimmen, enthält das DP-Array für jedes Feld einen Vektor mit $2^(n * 2)$ Einträgen, wobei n die Seitenlänge angibt und in unserem Fall gleich 10 ist. Die beim Index eines Eintrags gesetzten Bits geben mir an, welche der Felder aus den letzten zwei Reihen der Siedlung vorhanden sein sollen. Die Berechnung all dieser Einträge für jedes Feld verläuft analog zur vorhin beschriebenen Berechnung; es wird einfach das mV des Eintrags des vorherigen Feldes angeschaut, der die gleichen Bits wie der aktuell zu berechnende Eintrag gesetzt hat. Eine Ausnahme bilden die Bits, die die an (r,c) angrenzenden Felder symbolisieren. Sind sie nicht gesetzt bedeutet das, dass die Felder nicht verwendet werden, hier gibt es keine Veränderung zum normalen Verhalten. Sind sie jedoch gesetzt, muss man diese Bits für das Nachschlagen der vorherigen Einträge auf 0 setzen, da diese Einträge die Felder dann nicht verwenden dürfen (sie müssen ja für den Swap zur Verfügung stehen). Es stellt sich nun die Frage, wieso es notwendig ist, Einträge für alle Kombinationen der Felder der letzten zwei Reihen zu berechnen. Die Notwendigkeit der Speicherung der gesamten vorherigen Reihe ergibt sich daraus, dass sie erst mit dem letzten Eintrag der aktuellen Reihe nicht mehr benötigt wird: Auch hier muss noch der Fall überprüft werden, dass das letzte Feld einen Swap nach Süden (r-1,c) macht und das dortige Feld

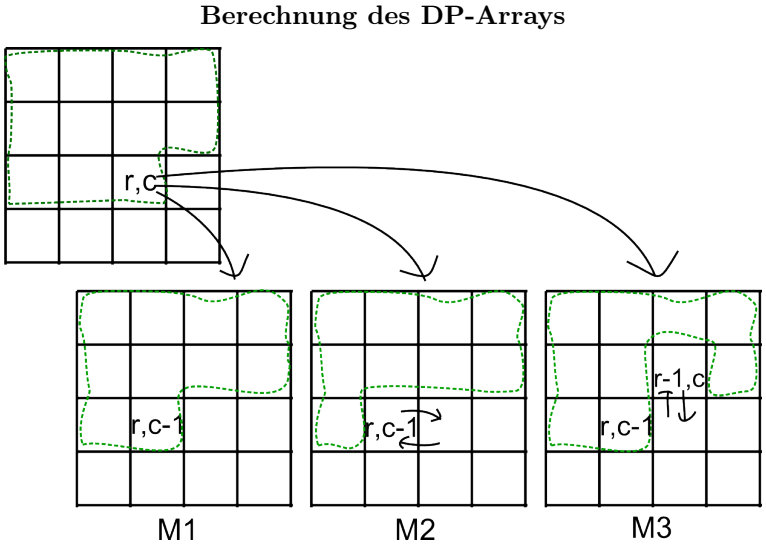


Abbildung 5: Zur Berechnung der besten Austauschvarianten für Feld (r,c) muss man drei verschiedene Fälle berücksichtigen. M1 geht davon aus, dass (r,c) nicht verwendet wird. M2 macht einen Swap zwischen (r,c) und (r,c-1). M3 macht einen Swap zwischen (r-1, c).

nicht benutzt worden sein darf. Die Speicherung aller möglichen Kombinationen und nicht nur der einzelnen Aktivierung oder Deaktivierung eines Feldes ergibt sich aus der Abhängigkeit der Berechnungen: Will ich die beste Austauschvariante für den Fall wissen, dass das Feld (r,c-2) deaktiviert ist, muss ich wie sonst auch sowohl einmal (r,c-1) und einmal (r-1, c) deaktivieren (M2 und M3 müssen ja überprüft werden). Für die beste Austauschvariante dieser Deaktivierung von zwei Feldern müsste ich drei Felder deaktivieren, usw. Schließlich ergibt sich noch die Speicherung der aktuellen Reihe aus dem Grund, dass diese Informationen für die Berechnungen der kommenden Reihe benötigt werden. Ein Spezialfall tritt ein, wenn man die Aktivierungskombinationen des ersten Feldes einer neuen Reihe ermitteln will. Jedes Feld der letzten Reihe weißt ja für jede mögliche Aktivierungskombination mit anderen Feldern dieser Reihe genau 2^n Möglichkeiten auf, diese Kombination mit Aktivierungskombinationen der vorletzten Reihe zu kombinieren. Diese müssen alle überprüft werden, um die Kombination mit der maximalen Verringerung zu nehmen. Deswegen sind hier insgesamt 2^{2*n+1} Durchläufe notwendig: Eine Potenz von dem aktuellen Feld, n Potenzen von der letzten und n Potenzen von der vorletzten Reihe. Für jeden Eintrag wird noch die Information gespeichert, welche der drei Austauschmöglichkeiten die beste ist und verwendet wird. Dadurch ist es nach der Erstellung des DP-Arrays möglich, die beste Austauschvariante zu rekonstruieren: Man nimmt einfach die beste Austauschmöglichkeit für das letzte Feld, trägt sie in den aktuellen Schritt ein und wiederholt den Prozess für das nächste

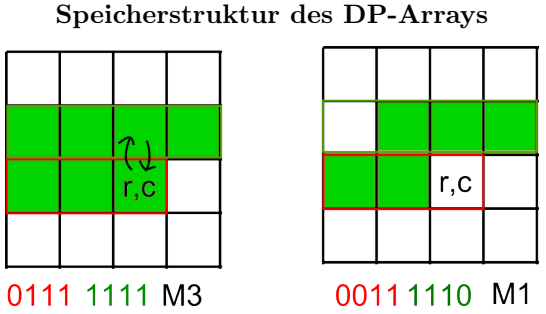


Abbildung 6: Zwei der 128 Einträge des Feldes (r,c) sind zu sehen. Beim ersten sind alle Felder aktiviert, als beste Austauschmöglichkeit hat sich M3 herausgestellt. Beim zweiten sind sowohl das letzte, als auch das Feld (r,c) selbst deaktiviert. Die beste Austauschmöglichkeit kann deswegen nur M1 sein.

Feld, wobei man den der gewählten Austauschmöglichkeit entsprechenden Eintrag wählt. Ein Schema der Datenstruktur enthält Abbildung 6. Nachdem die beste Austauschvariante rekonstruiert wurde, wird sie umgesetzt und das Verfahren auf die daraus resultierende Paketverteilung angewendet. Das geschieht solange, bis mV einen bestimmten Schwellwert unterschreitet, ab dem es sich nicht mehr lohnt, einen ganzen Schritt für nur wenige Swaps zu verwenden, oder sogar überhaupt keine Verringerung der Manhattan-Distanz mehr eintritt.

1.1.4 Komplexität

Beim ersten beschriebenen Verfahren wäre das Worst-Case Szenario, dass maximal ein Paket pro Durchlauf richtig platziert werden kann, da sich sonst die Pfade überschneiden würden. Bei einem Feld der Größe $n * n$ beträgt die maximale Manhattan-Distanz für ein Paket $2 * n - 1$. Da das Verfahren die Lösung Schritt für Schritt generiert, haben wir eine Laufzeit von $O(n*n*(2*n-1)) = O(n^3)$. Die Speicherkomplexität beläuft sich auf $O(n^2)$, wir müssen ja lediglich die Matrix der Siedlung abspeichern und updaten. Beide Komplexitäten sind in der Praxis bis zu $n = 1000$ ausreichend, der gegebene Fall mit $n = 10$ ist also ausführbar. Beim DP-Verfahren speichern wir zunächst die komplette Siedlung, eine Matrix aus $n * n$ Feldern. Jedes Feld enthält Einträge für alle möglichen Aktivierungskombinationen der letzten zwei Reihen, also $2^(2 * n)$ Einträge, von denen jeder aus konstant zwei Werten besteht. Alle benötigten Werte werden im DP-Array gespeichert, wodurch zunächst eine Speicherkomplexität von $O(n^2 * 2^{2*n})$ entsteht. Bei $n = 10$ hätten wir somit 104.857.600 Speichereinheiten, was bei zwei Werten pro Eintrag in 1,6 GByte realem Speicher resultieren kann. Diese Menge kann durch das Verwenden von zwei char-Werten anstelle zweier Integer auf 400 MByte resultiert werden, zur Laufzeit löschen kann man jedoch keine Reihen: Diese werden alle für die spätere Rekonstruktion benötigt. Der Algorithmus geht jeden Eintrag jedes Feldes durch, und überprüft für diesen konstante 3 Einträge. Hier muss jedoch noch der Durchlauf des Spezialfals beachtet werden. Zu-

dem wird das DP-Array sooft generiert, wie es Verringerungen der Manhattan-Distanz herbeiführen kann. Im Worst-Case führen wir eine Swap-Operation pro Durchlauf aus, die die Manhattan-Distanz um 2 Felder verringert. Bei einer maximalen gesamten Manhattan-Distanz von $O(n^3)$ ergibt sich also eine Laufzeit von $O(n^5 * 2^{2*n})$. Die bei $n = 10$ resultierenden 209.715.200.000 Zeiteinheiten könnten zwar einige Minuten Rechenzeit kosten, doch ist eine normale Paketverteilung weit entfernt von diesem Worst-Case-Szenario. Die Komplexität der DP-Strategie bildet auch die Gesamtkomplexität des Programms.

1.1.5 Kombination zweier Strategien

Für das beste Resultat müssen wir beide Strategien kombinieren. Zunächst wird die DP-Heuristik ausgeführt. Sobald sich keine hinreichenden Verbesserungen mehr einstellen, wenden wir das erste Verfahren auf die Siedlung an. Dieses verteilt dann alle noch falsch platzierten Pakete an die richtige Stelle.

Stärken Das vorgestellte Verfahren zeichnet sich dadurch aus, dass durch die DP-Strategie zumindest von einem Schritt auf den nächsten die größtmögliche Verbesserung erreicht wird; für einen Schritt ist das Verfahren optimal. Weiterhin werden in der Praxis für unsere Siedlung Resultate in Sekunden ermittelt.

Schwächen Das Verfahren weist alle typischen Schwächen einer Heuristik auf. Da das DP-Verfahren nicht zwingend eine Verringerung der Schrittanzahl herbeiführt und eventuell sogar eine Erhöhung herbeiführen könnte (wenn das auch sehr unwahrscheinlich ist), spielt Glück bei der Lösungsgenerierung immer eine Rolle. Dem entgegenwirken könnte man, in dem man eine Zufallsgröße in den Algorithmus einführt und diesen dann mehrmals durchläuft, um das beste Ergebnis zu verwenden. So könnten in einigen Schritten des DP-Verfahrens eigentlich die Manhattan-Distanz vergrößernde Swaps eingebaut werden. Diese könnten unter Umständen andere, bessere Swaps erst ermöglichen. Weiterhin ist der tatsächliche Speicherbedarf bei einem $10 * 10$ Feld von 450 MByte grenzwertig.

1.1.6 Erweiterung

Um die Heuristik zu verbessern, gibt es viele Möglichkeiten. So könnte man z.B. auch die simplen Austauschkreise aus vier Feldern in das DP-Verfahren mit einbeziehen und diese simplen Kreise anschließend auf in dem Schritt noch nicht verwendete Felder ausdehnen. Auch möglich wäre es, das DP-Verfahren zusammen mit dem ersten Verfahren für jeden Schritt auszuführen. Ich habe mich für zwei Verbesserungen entschieden.

Erste Verbesserung Zunächst wird das erste Verfahren verbessert, indem nicht mehr auf die Beendigung des längsten Pfades gewartet wird, bis neue Pfade hinzugefügt werden. Stattdessen werden dynamisch nach jedem Schritt

bereits abgelaufene Pfade entfernt und die durch sie besetzten Felder in der Matrix **B** wieder als frei markiert. Anschließend werden wieder so viele Pfade wie möglich hinzugefügt, um mehr Pakete gleichzeitig richtig platzieren zu können. Mag sich diese Idee zunächst trivial anhören, birgt ihre Umsetzung doch einige Tücken; so wird jeder Pfad zweimal begangen (von dem zu platzierenden und dem inversen Paket), weswegen die vorher noch boolsche Matrix **B** umgebaut werden muss, sodass sie Zähler enthält. Sobald einer der Pfade abgelaufen ist, wird der Zähler dekrementiert. Ist er 0, kann das Feld benutzt werden.

Zweite Verbesserung Die zweite Verbesserung zielt auf die bereits angesprochene Zufallsgröße ab. Statt diese jedoch in das DP-Verfahren einzubauen, bewirkt sie, teils zufällige, Swaps während des ersten Verfahrens. Konkret werden nach jedem Schritt alle in **B** als unbenutzt markierten Felder untersucht. Ein Swap zwischen zweier dieser unbenutzten Felder wird ausgeführt, wenn er die Manhattan-Distanz beider zu ihren Zielen verringert. Zudem kann dem Programm beim Start ein prozentualer Parameter mitgegeben werden. Sollte ein Swap die Manhattan-Distanz weder verringern, noch erhöhen, bestimmt dieser Parameter die Wahrscheinlichkeit, mit dem er trotzdem ausgetauscht wird. Da die Ergebnisse des Programms über mehrere Durchläufe hinweg bei einem Prozentwert über 0 variieren, lässt es sich beliebig oft wiederholen, um bis zu einem bestimmten Grad bessere Resultate zu erzielen.

1.2 Implementierung

Globale Datenstrukturen

```
vector <vector <PII> > fields; //1,2 indizes; PII = wo will dieses Pa
vector <vector < vector <PII> > > stepsV; //alle Schritte
vector <vector < vector <pair <unsigned char,unsigned char> > > > dp;
vector <vector <string> > result; //1,2 indizes; string = Befehlsabfo
ifstream fieldIn ("drohnen_eingabe_amacity.txt");
int wLength ,
    tLength = 20;
int steps = 0;
int elC;
Dimensions dimensions(wLength*tLength, wLength*(tLength+10)*steps);
Document doc("result.svg", Layout(dimensions, Layout::TopLeft));
```

Die Eingabe wird über eine Text-Datei realisiert, in der sich die Siedlung im vorgegebenen Format befindet (hier drohnen_eingabe_amacity.txt). Die graphische Ausgabe erfolgt über die Bibliothek simple-svg von Mark Turney, die zur Erstellung von SVG-Dateien benutzt werden kann. Diese graphische Ausgabe wird auch in den Beispielen verwendet. Der Plan zur Paketverteilung wird später in die Datei paketverteilung.txt ausgegeben. Dabei verwendet sie die in der Matrix *result* gespeicherten Zeichenketten; $result[i][j]$ enthält die Zeichenkette für das Feld in der i-ten Reihe und j-ten Spalte. Diesen Matrixaufbau haben auch $fields$, $stepsV$ und dp. $fields$ enthält die aktuelle Paketverteilung

und speicher in einem $pair < int, int >$ das Zielfeld. $stepsV$ enthält die gesamte Paketverteilung, also $fields$ über alle Schritte hinweg. dp ist der Speicher, der für das DP-Verfahren notwendig ist. Der innerste Vektor speichert alle elC verschiedenen Aktivierungskombinationen, von denen jeder die Werte mV und die genutzte Austauschmöglichkeit angibt. Die Verwendung von zwei unsigned chars führt zu keinem Overflow und reduziert die absolute Speichermenge effektiv auf ein Viertel im Vergleich zu Integers.

Das DP-Verfahren

```
for (int r = 0; r < wLength; ++r) {
    for (int c = 0; c < wLength; ++c) {
        if (r == 0 && c == 0)   continue; //Das erste Feld unbeacht
        pair <int, int> prevP = getPrev (r, c), //Das vorherige Pa
                        prevV = fields[prevP.first][prevP.second],
                        nowV = fields[r][c]; //Das Ziel des jetzig
        int elsToCheck = getIndex(r > 0 ? wLength+c : c); //Alle Bi

        for (int comb = 0; comb < elsToCheck; ++comb) {
```

Das DP-Verfahren geht zeilen- und feldweise die dp-Matrix durch. Das Feld [0,0] kann für sich keine Swaps ausführen, deswegen kann man es immer mit 0 initialisiert lassen (continue-Anweisung). Die Funktion $getPrev(r, c)$ liefert das Feld zurück, das direkt vor [r,c] untersucht wird. Befindet man sich auf dem ersten Feld einer Reihe, ist dies das letzte Feld der vorherigen Reihe. elsToCheck ist die größte Aktivierungskombination, die man überprüfen muss. In ihr sind alle Felder der vorigen Reihe, sowie alle Felder der aktuellen Reihe bis zum aktuellen Feld aktiviert. $wLength$ gibt dementsprechend die Länge einer Reihe an, $getIndex(n)$ liefert für n Felder den passenden Wert. Sind wir in der ersten Reihe gibt es keine vorige Reihe, Swaps nach Norden sind dann auch nicht möglich. Die Kombinationen werden dann alle berechnet.

```
int offset = r > 0 ? wLength : 0;
if (c == 0) {
    int combPrev = (comb & ~(1 << (wLength))); //Die vorherige Kombination äquivalent zu die
    if (r>1)
         combPrev <<= wLength;
    int maxVAll = 0;
    for (int cToChk = combPrev; cToChk < (combPrev+ (r > 1 ? getIndex(wLength) : 1));
    ++cToChk) {
        int bindingNothing = dp[prevP.first][prevP.second].at(cToChk).first;//Der Wert des v
        int maxV = bindingNothing;
        int predecessor = 0; //0 = unbesetzt, 1 = bund mit links, 2 = bund mit oben
        int oldMax = maxV;
        if (isZero(comb, offset) == false) { //wenn das eigene Feld besetzt werden darf
            if (r > 0 && isZero (comb, c) == false) { //wenn das oben angrenzende Feld beset
                pair <int, int> upV = fields[r-1][c];
                int bindingUp = (upV.first >= r ? 1 : -1) + (nowV.first <= (r-1) ? 1 : -1);
                maxV = max (maxV, bindingUp + dp[r-1][c][cToChk & ~(1 << (wLength))].first);
                predecessor = oldMax == maxV ? predecessor : 2;
            }
        }
        if (maxV > maxVAll) {
            dp[r][c][comb] = make_pair (maxV, predecessor);
            maxVAll = maxV;
        }
    }
```

Zunächst muss man den Spezialfall behandeln, dass man sich im ersten Feld einer Reihe befindet. In so einem Fall muss man ja auch die Kombinationsmöglichkeiten der vorletzten Reihe durchgehen, und von diesen jeweils die beste auswählen. Das geschieht in der hier abgebildeten Schleife. Mit Bit-Operatoren wird das aktuelle Feld aus der Aktivierungskombination *combPrev* herausgerechnet, damit die Einträge des vorigen Feldes in der for-Schleife überprüft werden können. Der in *maxVAll* gespeicherte beste Wert wird dann für den aktuellen Eintrag verwendet. Da man sich im ersten Feld einer Reihe befindet, gibt es zwei Möglichkeiten: Entweder kein Swap des aktuellen Feldes, was in *bindingNothing* gespeichert wird. Oder aber ein Swap nach Norden, was möglich ist, wenn das aktuelle Feld aktiviert und das darüber ist (isZero()==false). *bindingUp* enthält dann die minimale Manhattan-Distanz-Verringerung, die sich aus dem Swap ergeben würde. *predecessor* ist dann die Austauschmöglichkeit mit dem größten mV.

```
if (c > 0 && isZero (comb, c-1+offset) == false) { //wenn das links angrenzende Feld besetzt
    int bindingLeft = (prevV.second >= c ? 1 : -1) + (nowV.second <= prevP.second ? 1 : -1);
    maxV = max (maxV, bindingLeft + dp[r][c-1][combPrev & ~(1 << (c-1+offset))].first);
    predecessor = oldMax == maxV ? predecessor : 1;
    oldMax = maxV;
}
```

Falls man sich nicht im ersten Feld der Reihe befindet, kommt die im obigen Code dargestellte Möglichkeit hinzu, einen Swap nach Westen zu machen. Ansonsten beträgt der offset für die Werte der aktuellen Reihe $wLength$, also eine komplette Reihe, da die Werte der vorigen Reihe jetzt genau festgelegt werden können: So sagt $combPrev\& \ (1 << (c - 1 + offset))$, dass alle Felder der aktuellen und letzten Reihe aktiviert sind, außer das vorige im Westen. Dies ist notwendig, damit wir mit dem Feld im Westen swappen können und dabei sicher gehen können, dass es noch nicht in einen Swap involviert ist.

```
void moveField () { //Wirft die Packete nach dem durch DP vorgerechneten Plan
    int conditionMask = elC;
    vector <vector <bool> > isHandled (wLength, vector <bool> (wLength, false));
    for (int r = wLength-1; r >= 0; --r) {
        for (int c = wLength-1; c >= 0; --c) {
            int vMax = 0;
            int idxMax = 0;
            int elsToCheck = getIndex(r > 0 ? wLength+c : c);
            for (int comb = 0; comb < elsToCheck; ++comb) { //Die beste Kombinat
                int combToUse = comb & conditionMask; // Unter der Vorbedingung,
                int vNow = dp[r][c][combToUse].first;
                if (vNow > vMax) {
                    vMax = vNow;
                    idxMax = combToUse;
                }
            }
```

Nachdem die beste Austauschvariante für das gesamte Feld generiert wurde, müssen wir aus dem DP-Array die dafür benötigten Swap-Operationen extrahieren. Die for-Schleifen starten dafür vom letzten Feld. In jedem Feld gehen sie durch alle Aktivierungskombinationen und wählen die mit der maximalen Verringerung aus, die zudem die conditionMask-Bits gesetzt hat. Diese aktivierten Felder ergeben sich aus den bereits von vorausgehenden Swaps verwendeten Feldern. Die Matrix *isHandled* gibt an, welche Felder schon für Swaps verwendet wurden.

```
if (isHandled[r][c] == false) {
    switch (dp[r][c][idxMax].second) { //Abhängig vom besten Wert die Swaps ausführe
    case 0: conditionMask = elC; result[r][c] += '_'; break;
    case 1: swap (fields[r][c-1], fields[r][c]);
        conditionMask = ~(elC & (1 << c));  result[r][c] += 'W';
        result[r][c-1] += 'O'; isHandled[r][c-1] = true; break;
    case 2: swap (fields[r-1][c], fields[r][c]);
        conditionMask  = ~(elC & (1 << (r+c-1)));
        result[r][c] += 'N'; result[r-1][c] += 'S'; isHandled[r-1][c] = true; break;
    }
}
```

Diese Information wird benötigt um zu wissen, ob man für das aktuelle Feld überhaupt noch einen Swap auswählen könnte. Ist das der Fall, wird die beste gefundene der drei möglichen Austauschmöglichkeiten umgesetzt. Dafür muss in der conditionMask das Feld eventuell als aktiv gesetzt werden, die Möglichkeit zur Zeichenkette des Feldes hinzugefügt werden und das Feld, wenn verwendet, in *isHandled* als verwendet markieren.

```
} while (improved() == true);
```

Der Vorgang wird wiederholt, solange eine merkliche Verbesserung erkennbar ist. *improved*() nimmt dabei die Gesamtverringerung (der maximale Eintrag im letzten DP-Feld) und vergleicht sie mit einen Schwellwert.

Die Lösungsgenerierung

```
list <vector <pair <int, int> > > displaced; //Die verrückten Packete
vector <vector <int> > isAdded (wLength, vector <int> (wLength, 0)); //Wird das Pack
int ctr = 0;
int moveCtr = 0;
while (isFinished() == false) { //Alle hinzugefügten Packete platzieren, ein Swap pr
    for (int r = 0; r < wLength; ++r) {
        for (int c = 0; c < wLength; ++c) {
            pair <int, int> goTo = fields[r][c];
            if (goTo.first != r || goTo.second != c) { //Wenn das Packet noch nicht
                //Und wenn sein Verteilungsweg noch nicht verwendet wird...
                bool fieldOcc = false;
                for (int i = min (r, goTo.first); i <= max (r, goTo.first); ++i) {
                    fieldOcc |= isAdded[i][min(goTo,make_pair (r,c)).second] > 0;
                }
                for (int i = min (c, goTo.second); i <= max (c, goTo.second); ++i) {
                    fieldOcc |= isAdded[max(goTo,make_pair (r,c)).first][i] > 0;
                }
```

Nachdem DP-Verfahren wird das erste beschriebene Verfahren verwendet, um zu einer endgültigen Lösung zu gelangen. Der Vektor *displaced* enthält dabei die zu platzierenden Pakete, jeweils mit Start- aktuellem und Zielfeld. *isAdded* gibt für jedes Feld an, ob es bereits von einem Pfad verwendet wird. Dann werden solange Pakete platziert, wie noch nicht alle Pakete auf ihrem designierten Feld stehen (*isFinished*). Zunächst werden dafür in den beiden äußeren for-Schleifen alle neuen Pfade ermittelt, die platziert werden können. Die beiden inneren for-Schleifen testen, ob ein Feld des potentiellen Pfades schon besetzt ist, die erste den vertikalen, die zweite den horizontalen Teil des Pfades. Durch die min- und max-Operationen wird für zwei Punkte immer derselbe Pfad erstellt, egal, in welcher Reihenfolge sie übergeben werden.

```
if (fieldOcc == false) {
    for (int i = min (r, goTo.first); i <= max (r, goTo.first); ++i) {
        isAdded[i][min(goTo,make_pair (r,c)).second] = 2;
    }
    for (int i = min (c, goTo.second); i <= max (c, goTo.second); ++i) {
        isAdded[max(goTo,make_pair (r,c)).first][i] = 2;
    }
    displaced.push_back (vector <pair <int, int> >
            {make_pair (r, c), goTo, make_pair (ctr, 0), make_pair (r, c)});
    displaced.push_back (vector <pair <int, int> >
            {goTo, make_pair (r, c), make_pair (ctr, 0), goTo});
    ++ctr;
}
```

Ist noch kein Feld besetzt, besetzen zwei fast identische for-Schleifen den Pfad und fügen die entsprechenden Pakete *displaced* hinzu. Da auch der umgekehrte Pfad gegangen werden muss, gibt es zwei pus_back-Anweisungen und der Zähler in *isAdded* wird auf zwei gesetzt.

```
for (auto it = displaced.begin(); it != displaced.end();) {
    pair <int, int> pNow = (*it)[0],
                    goTo = (*it)[1];
    //In der Richtigen Reihenfolge swappen, damit sowohl Packet als auch Umkehrpac
    if (goTo.first > pNow.first) {
        swap (fields[pNow.first][pNow.second], fields[pNow.first+1][pNow.second]);
        result[pNow.first][pNow.second] += 'S';
        result[pNow.first+1][pNow.second] += 'N';
        ++(*it)[0].first;
```

Nun werden alle zu platzierenden Pakete in die entsprechende Richtung bewegt. Die hier gezeigte if-Bedingung ist eine von vier, eine pro Himmelsrichtung. Sie swappen das Paket zunächst in die jeweilige Richtung, fügen die Bewegung zur Zeichenkette hinzu und updaten die Position des Paketes in *displaced*.

```
else {
  //unadd element
  pair <int, int> origin = (*it)[3];
  int subtracted = isAdded[origin.first][origin.second]-1;
  for (int i = min (origin.first, goTo.first); i <= max (origin.first, goTo.first); ++i) {
      isAdded[i][min(goTo,origin).second] = subtracted;
  }
  for (int i = min (origin.second, goTo.second); i <= max (origin.second, goTo.second); ++i) {
      isAdded[max(goTo,origin).first][i] = subtracted;
  }
  it = displaced.erase(it);
  continue;
```

Falls sich das Paket bereits auf dem richtigen Feld befindet, wird der Pfad aus *displaced* gelöscht und in *isAdded* dekrementiert, also eventuell wieder freigegeben.

```
if (next(it) != displaced.end() && (*it)[2].first == (*next(it))[2].first &&
    (*it)[2].second == 0) {
    int dist = (abs ((*next(it))[0].first-pNow.first)+
                abs ((*next(it))[0].second-pNow.second));
    if (dist <= 2) {
        if (dist == 1) {
            assert ((*next(it))[0].second < 10 && (*next(it))[0].first < 10);
            switch (result[pNow.first][pNow.second].back()) {
            case 'S':  --(*next(it))[0].first; break;
            case 'N':  ++(*next(it))[0].first; break;
            case 'W':  ++(*next(it))[0].second; break;
            case 'O':  --(*next(it))[0].second; break;
            }
            ++(*it)[2].second;
            ++(*next(it))[2].second;
        }
        ++it;
```

Schließlich muss noch der Spezialfall behandelt werden, dass die sich beide auf dem selben Pfad befindlichen Pakete, von denen das eine den Umkehrpfad abgeht, kreuzen. Sind sie zwei Felder voneinander entfernt, muss das zweite Paket warten, da sie ja nicht beide auf das selbe Feld swappen können. Deswegen

wird hier der Iterator einmal zusätzlich inkrementiert. Sind sie hingegen auf gegenüberliegenden Feldern, hat das erste Paket das zweite bereits bewegt. Die if-Bedingung führt deshalb noch die notwendigen Änderungen zum zweiten Paket hinzu, und überspringt dieses dann ebenfalls. Nachdem nun insgesamt ein Schritt hinzugefügt wurde, wird, der Erweiterung gemäß, wieder von vorne begonnen und überprüft, welche weiteren Pakete platziert werden können.

```
vector <pair <int, int> > throwM = {make_pair (0,1),
                                    make_pair (1,0), make_pair (0,-1), make_pair(-1,0)}; //
vector <char> throwMS = {'O', 'S', 'W', 'N'};   //Wurfmatrix-Zeichen
vector <vector <bool> > usedNow (wLength, vector <bool> (wLength, false));
for (int i = 1; i < wLength-1; ++i) {
    for (int j = 1; j < wLength-1; ++j) {
        for (int k = 0; k < throwM.size(); ++k) {
            int iN = i+throwM[k].first,
                jN = j+throwM[k].second;
             int sum = mDist (fields[i][j],
                    make_pair (i,j)) > mDist (fields[i][j], make_pair (iN, jN)) ? 1 : -1;
             sum += mDist (fields[iN][jN],
                    make_pair (iN,jN)) > mDist (fields[iN][jN], make_pair (i, j)) ? 1 : -1;
```

Schließlich wird noch die zweite Erweiterung umgesetzt. Die drei for-Schleifen gehen jedes nicht am Rand liegende Feld durch und testen in jede Richtung, welche mV ein Swap mit diesem Feld hätte (sum).

```
if ((isAdded[i][j] == 0 && isAdded[iN][jN] == 0) &&
    usedNow[i][j]==false && usedNow[iN][jN]==false &&
    ((sum == 2 && randomFac > (rand()%10)) || (sum==0 && randomFac > (rand()%100))) &&
    (fields[i][j].first != i || fields[i][j].second != j) &&
    (fields[iN][jN].first != iN || fields[iN][jN].second != jN)
    ) {
   result[i][j].back() = throwMS[k];
   result[iN][jN].back() = throwMS[(k+2)%4];
   swap (fields[i][j], fields[iN][jN]);
   usedNow[i][j] = true;
   usedNow[iN][jN] = true;
}
```

Wenn dann mehrere Bedingungen erfüllt sind (das Feld darf gerade nicht in einem Pfad enthalten sein, es darf noch nicht in einen anderen Swap eingebaut sein, der Swap darf die Manhattan-Distanz nicht vergrößern und keines der beiden Pakete darf bereits richtig platziert sein), kommt die Wahrscheinlichkeit ins Spiel. Der beim Programmstart übergebene Wert $randomFac$ gibt die Wahrscheinlichkeit in Prozent an, mit der ein von der Manhattan-Distanz her neutraler Swap stattfindet. Wird die Distanz verringert, ist die Wahrscheinlichkeit zehnmal so hoch. Ist der generierte Zufallswert kleiner als $randomFac$, wird der Swap ausgeführt und den Zeichenketten hinzugefügt.

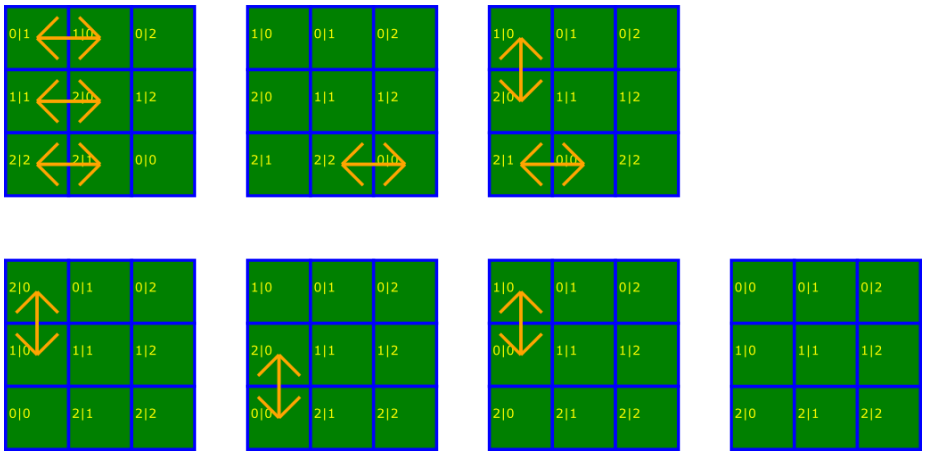
Ausgabe Wurden alle Pakete richtig platziert, wird die Lösung einmal in die Textdatei und dann noch als SVG-Datei ausgegeben.

Monte-Carlo-Methode Um durch mehrmaliges Durchlaufen des Algorithmus mithilfe des Zufallsfaktors bessere Ergebnisse erhalten zu können, habe ich den Batch-Skript montecarlo.bat geschrieben. Er lässt das Programm 1000 Mal laufen und speichert alle Ergebnisse. Die benötigten Schritte gibt er aus. Da die Windows-Konsole standardmäßig keine 1000 Zeilen gebuffert hat, empfiehlt es sich, die Ausgabe in eine Textdatei umzulenken.

1.3 Beispiele

Die Namen der Eingabedateien wurden entsprechend angepasst. Das Programm ließt nur von einer Datei namens ßiedlung.txt". Verifizierung durch die Juroren kann also nur erfolgen, indem sie das zu überprüfende Beispiel in ßiedlung.txtümbenennen.

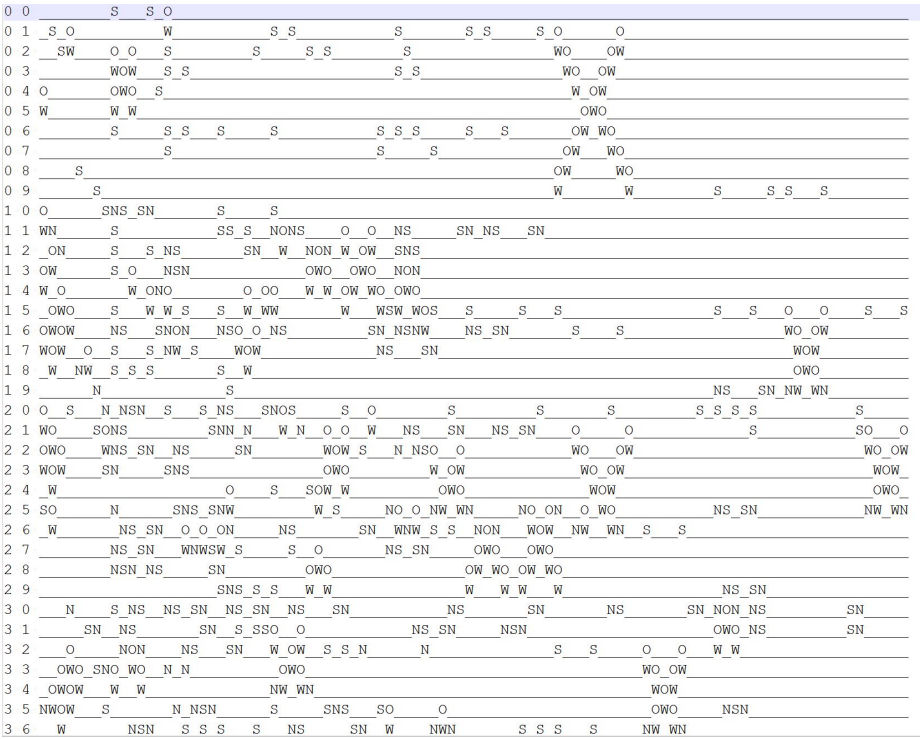
Beispiel 1 Datei: "drohnen_eingabe_beispiel.txt"
Ausgabe: "beispiel1Out.txt"Benötigte Schritte zur Lösung, einfache Berechnung: 6
Benötigte Zeit: 0,019s
Benötigter Speicher: 1.7MByte
Beste Lösung nach 100 zufälligen Durchläufen: 6

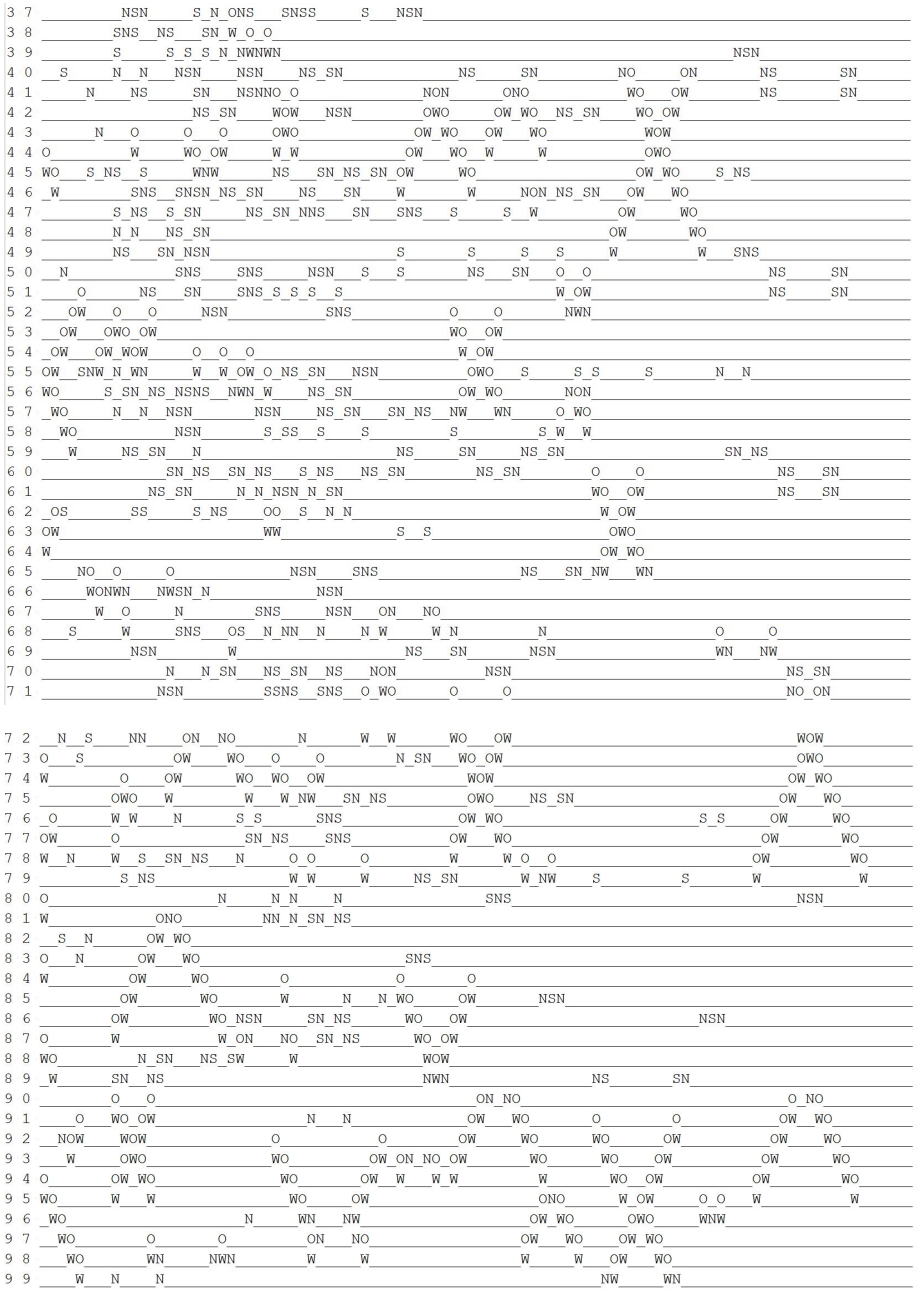


Hinweis: Die originale SVG-Datei enthält die Schritte vertikal geordnet. Aus Platzgründen habe ich sie hier manuell horizontal angeordnet.

```
0 0 O_SS_S
0 1 W_____
0 2 _____
1 0 O_NNSN
1 1 W_____
1 2 _____
2 0 O_O_N_
2 1 WOW___
2 2 _W____
```

Beispiel 2 Datei: "drohnen_eingabe_amacity.txt"
Ausgabe: "beispiel2Out.txt"Benötigte Schritte zur Lösung, einfache Berechnung: 112
Benötigte Zeit: 5.445 s
Benötigter Speicher: 400.8 MByte
Beste Lösung nach 100 zufälligen Durchläufen: 101
Beste Lösung nach 1000 zufälligen Durchläufen: 98
Lösung mit 98 Schritten:





Beispiel 3 Dieses Beispiel wird ohne die Erweiterungen ausgeführt, um deren Leistungsfähigkeit zu demonstrieren. Den dazugehörigen Code mit Anwendung findet man im Ordner "Programm_ohne_Erweiterungen". Datei: "drohnen_eingabe_amacity.txt"

Ausgabe: "beispiel3Out.txt"Benötigte Schritte zur Lösung, einfache Berechnung: 167
Benötigte Zeit: 5.903 s
Benötigter Speicher: 401.9 MByte

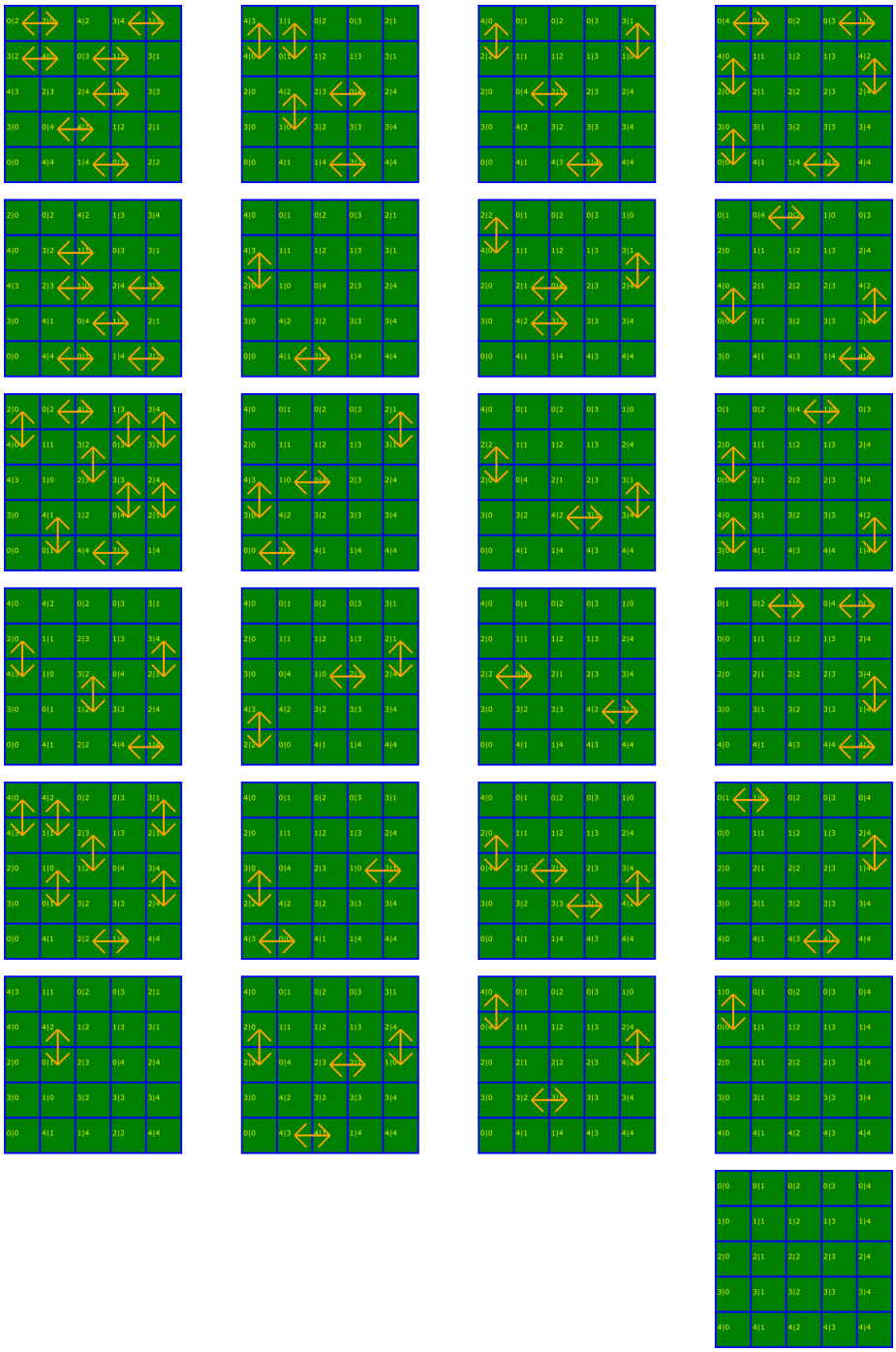
Beispiel 4 Datei: "drohnen_eingabe_amacity.txt"
Ausgabe: "beispiel4Out.txt"Dieses Beispiel wird sowohl ohne die Erweiterungen, als auch ohne das DP-Verfahren ausgeführt, um dessen Leistungsfähigkeit zu demonstrieren. Den dazugehörigen Code mit Anwendung findet man im Ordner "Programm_ohne_alles"
Benötigte Schritte zur Lösung, einfache Berechnung: 218
Hinweis: Zeit und Speicher sind hier nicht angegeben, da der DP-Speicher trotzdem alloziiert wird. Somit würde die Angabe keinen Sinn ergeben.

Beispiel 5 Datei "beispiel5.txt"
Ausgabe: "beispiel5Out.txt"
Benötigte Schritte zur Lösung, einfache Berechnung: 0
Benötigte Zeit: 0.016 s
Benötigter Speicher: 1.7 MByte
In diesem Beispiel sind bereits alle Pakete richtig zugeordnet.

```
2
0 0 0 0
0 1 0 1
1 0 1 0
1 1 1 1
```

Beispiel 6 Datei "beispiel6.txt"
Ausgabe: "beispiel6Out.txt"
Benötigte Schritte zur Lösung, einfache Berechnung: 24
Benötigte Zeit: 0.066 s
Benötigter Speicher: 1.7 MByte
Dieses Beispiel ist eine gewöhnliche Siedlung, mit einer Seitenlänge von 5.

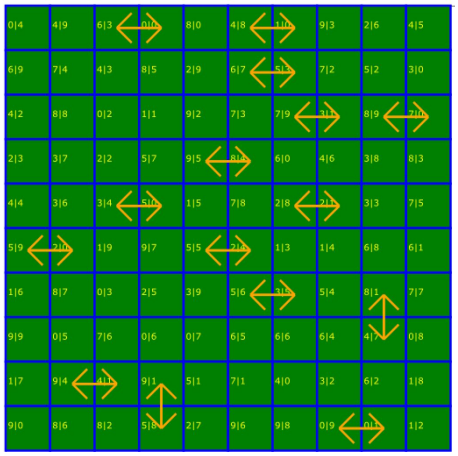
```
0 0 O_S_S_S_____SS___SO___OS
0 1 W_O_S_S_____________WO_OW_
0 2 __W_____________WOW__
0 3 O_S_____________O_WO__
0 4 W_S_S___S___S_____W__W__
1 0 O_NSN_NS___SNNS_SNS_S__N
1 1 WO__NSN_____________
1 2 OWS_S_____________
1 3 W_N_____________
1 4 __NSN___NS_SNS___SS___S_
2 0 ___N___NS_SN__NON_NSN___
2 1 _O__SNS_O___OO_WO_____
2 2 OWNSN_O_WO_OWW__W_____
2 3 WOS___W__WOW_____
2 4 _WSNS____NWN_NS_SNNS_SN_
3 0 _____NSN_____SNS___
3 1 O_S_N_N_____O___O_____
3 2 WO_N_____WO_OW_____
3 3 _WN_____WOW_____
3 4 __N_N_____NWN__NSN__
4 0 _____ONO_____N_N___
4 1 _ON___OW_WO_____
4 2 OWO_O_OW___WO_____O___O_
4 3 WOWOW_W_____W_____WO_OW_
4 4 _W_W_____WNW__
```



Hinweis: Diesmal sind die Schritte vertikal geordnet.

Beispiel 7 Datei "beispiel7.txt"
Dieses Beispiel ist eine gewöhnliche Siedlung, mit einer Seitenlänge von 15.
Das Programm wirft eine bad_alloc - Exception, da nicht genügend Speicherplatz zur Verfügung steht.

Beispiel 8 Datei "beispiel8.txt"
Ausgabe: "beispiel8Out.txt"
Benötigte Schritte zur Lösung, einfache Berechnung: 24
Benötigte Zeit: 9.458 s
Benötigter Speicher: 400.8 MByte
Dieses Beispiel ist eine gewöhnliche Siedlung, mit einer Seitenlänge von 10.
Das Bild zeigt den Grundkonfiguration mit dem ersten Schritt.



2 Anhang

2.1 Quelltext

main.cpp

```
#include <iostream>
#include <string>
#include <fstream>
#include "simple_svg_1.0.0.hpp"
#include <sstream>
#include <utility>
#include <vector>
#include <queue>
#include <list>
#include <bitset>
#include <cmath>
#include <cassert>
#include <stdlib.h>
```

```
#include <time.h>

using namespace std;
using namespace svg;

#define PII pair<int, int>

vector <vector <PII> > fields; //1,2 indizes; PII = wo will dieses Paket
    hin?
vector <vector < vector <PII> > > stepsV; //alle Schritte
vector <vector < vector <pair <unsigned char,unsigned char> > > > dp; //
    1, 2: indizes, 3: welche reihe, 4: die felderkombination
vector <vector <string> > result; //1,2 indizes; string = Befehlsabfolge
    fuer die werfende Person an diser Stelle
list <vector <pair <int, int> > > displaced; //Die verrueckten Packete
vector <vector <int> > isAdded; //Wird das Packet gerade verwendet?
ifstream fieldIn ("siedlung.txt"); //Eingabedatei
int wLength ,  //Laenge einer Reihe
   tLength = 20; //SVG-Ausgabe, Pixel eines Feldes
int steps = 0; //Wie viel Schritte sind bereits vergangen?
int elC; // Alle Felder in zwei Reihen aktiviert.
int ctr; //ID des displaced-Pfades
Dimensions dimensions(wLength*tLength, wLength*(tLength+10)*steps);
Document doc("result.svg", Layout(dimensions, Layout::TopLeft));

void paintArrow (Point a, Point b) { //Austauschpfeil von a nach b
    zeichnen
    Stroke s = Stroke (1, Color::Orange);
    doc << Line (a, b, s);
    doc << Line (b, Point (b.x+(a.y-b.y)/3+(a.x-b.x)/3,
        b.y+(a.x-b.x)/3+(a.y-b.y)/3), s);
    doc << Line (b, Point (b.x-(a.y-b.y)/3+(a.x-b.x)/3,
        b.y-(a.x-b.x)/3+(a.y-b.y)/3), s);
}

bool improved () { //Testet, ob das DP eine Verbesserung gebracht hat.
    for (int i = 0; i < elC; ++i) {
           if (dp.back().back()[i].first > 8) {
               return true;
           }
    }
    return false;
}

bool isFinished () { //Testet, ob alle Pakete richtig platziert sind
    for (int i = 0; i < wLength; ++i) {
        for (int j = 0; j < wLength; ++j) {
            if (fields[i][j].first != i || fields[i][j].second != j)
                return false;
        }
```

```
    }
    return true;
}

int mDist (pair <int, int> a, pair <int, int> b) { //Liefert die
    Manhattan-Distanz zwischen zwei Punkten zurueck
    return abs(a.first-b.first) + abs(a.second-b.second);
}

pair <int, int> getPrev (int r, int c) { //Liefert den Vorfolger von [r,
    c] zurueck
    return make_pair ((c==0?r-1:r), (c==0?wLength-1:c-1));
}

int getIndex (int num) { //liefert einen Zaehler zurueck, der Ueber alle
    Kombinationen bis zu diesem hin geht
    return 1<<(num+1);
}

bool isZero (int idx, int num) { //Testet, ob das Feld num in idx 0 ist
    return !((idx >> num) & 1);
}

void moveField () { //Wirft die Packete nach dem durch DP vorgerechneten
    Plan
    int conditionMask = elC; //Welche Felder aktiviert sein muessen
    vector <vector <bool> > isHandled (wLength, vector <bool> (wLength,
        false)); //Welche Felder schon benutzt wurden
    for (int r = wLength-1; r >= 0; --r) {
        for (int c = wLength-1; c >= 0; --c) {
            int vMax = 0;
            int idxMax = 0;
            int elsToCheck = getIndex(r > 0 ? wLength+c : c);
            for (int comb = 0; comb < elsToCheck; ++comb) { //Die beste
                Kombination finden
                int combToUse = comb & conditionMask; // Unter der
                    Vorbedingung, dass bestimmte Felder ungebunden sind
                int vNow = dp[r][c][combToUse].first;
                if (vNow > vMax) { //Besten Wert aktualisieren
                    vMax = vNow;
                    idxMax = combToUse;
                }
            }
            if (isHandled[r][c] == false) {
                switch (dp[r][c][idxMax].second) { //Abhaengig vom besten
                    Wert die Swaps ausfuehren
                case 0: conditionMask = elC; result[r][c] += '_'; break;
                case 1: swap (fields[r][c-1], fields[r][c]);
                    conditionMask = ~(elC & (1 << c)); result[r][c] += 'W';
                    result[r][c-1] += 'O'; isHandled[r][c-1] = true; break;
```

```
                case 2: swap (fields[r-1][c], fields[r][c]);
                    conditionMask = ~(elC & (1 << (r+c-1)));
                    result[r][c] += 'N'; result[r-1][c] += 'S';
                        isHandled[r-1][c] = true; break;
                }
            }
        }
    }
}

int main(int argc, char* argv[])
{
    string resFileName = "result";
    srand(time(NULL));
    int randomFac = 0;
    if (argc > 1) {
        sscanf (argv[1], "%i", &randomFac);
        resFileName = argv[2];
    }
    doc = Document(resFileName+".svg", Layout(dimensions,
        Layout::TopLeft));
    //Eingabe Beginn
    fieldIn >> wLength;

    fields.resize (wLength, vector <PII>(wLength));
    result.resize (wLength, vector <string> (wLength));
    elC = getIndex (2*wLength)-1;

    for (int i = 0; i < wLength*wLength; ++i) { //Initialisierung der
        Siedlung
        int col, row, colD, rowD;
        fieldIn >> row >> col >> rowD >> colD;
        fields[row][col] = make_pair (rowD, colD);
    }
    if (isFinished() == true) { //Spezialfall: Alle Pakete sind bereits
        richtig platziert
        stepsV.push_back(fields);
        goto alreadyFinished;
    }
    //DP Beginn
    dp.resize (wLength, vector <vector <pair <unsigned char,unsigned
        char> > > (wLength, vector <pair <unsigned char,unsigned char> >
        (elC, make_pair (0,0)))); //Neu fuer jede neue Zeiteinheit
    do {
        if (steps > 0) {
            moveField();
        }
        stepsV.push_back(fields);
        ++steps;
        for (int i = 0; i < wLength; ++i) {
```

```
    for (int j = 0; j < wLength; ++j) {
        fill(dp[i][j].begin(), dp[i][j].end(), make_pair (0,0));
            //dp resetten um neuen Durchgang starten zu koennen
    }
}

for (int r = 0; r < wLength; ++r) {
    for (int c = 0; c < wLength; ++c) {
        if (r == 0 && c == 0) continue; //Das erste Feld
            unbeachtet lassen, wurde schon initialisiert
        pair <int, int> prevP = getPrev (r, c), //Das vorherige
            Packet
                    prevV = fields[prevP.first][prevP.second],
                        //Das Ziel des vorherigen Packets
                    nowV = fields[r][c]; //Das Ziel des
                        jetzigen Packets
        int elsToCheck = getIndex(r > 0 ? wLength+c : c); //Alle
            Bindungsmoeglichkeiten mit dem aktuellen Packet
            berechnen

        for (int comb = 0; comb < elsToCheck; ++comb) { //Alle
            Aktivierungsmoeglichkeiten der letzten zwei Reihen
            durchgehen
           int offset = r > 0 ? wLength : 0;
           if (c == 0) { //Spezialfall, Feld befindet sich am
               westlichen Rand
              int combPrev = (comb & ~(1 << (wLength))); //Die
                  vorherige Kombination Aequivalent zu dieser
                  Kombination
              if (r>1)
                   combPrev <<= wLength;
              int maxVAll = 0;
              for (int cToChk = combPrev; cToChk < (combPrev+ (r
                  > 1 ? getIndex(wLength) : 1));
              ++cToChk) { //Beste Aktivierungsmoeglichkeit der
                  letzten UND VORLETZTEN Reihe
                 int bindingNothing =
                     dp[prevP.first][prevP.second].at(cToChk).first;//Der
                     Wert des vorherigen Feldes zu dieser
                     Kombination
                 int maxV = bindingNothing;
                 int predecessor = 0; //0 = unbesetzt, 1 = bund
                     mit links, 2 = bund mit oben
                 int oldMax = maxV;
                 if (isZero(comb, offset) == false) { //wenn
                     das eigene Feld besetzt werden darf
                    if (r > 0 && isZero (comb, c) == false) {
                        //wenn das oben angrenzende Feld
                        besetzt werden darf
                       pair <int, int> upV = fields[r-1][c];
```

```
                    int bindingUp = (upV.first >= r ? 1 :
                        -1) + (nowV.first <= (r-1) ? 1 :
                        -1);
                    maxV = max (maxV, bindingUp +
                        dp[r-1][c][cToChk & ~(1 <<
                        (wLength))].first);
                    predecessor = oldMax == maxV ?
                        predecessor : 2;
                }
            }
            if (maxV > maxVAll) {
                dp[r][c][comb] = make_pair (maxV,
                    predecessor);
                maxVAll = maxV;
            }
        }

    } else {
        int combPrev = comb & ~(1 << (c+offset)); //Die
            vorherige Kombination Aequivalent zu dieser
            Kombination
        int bindingNothing =
            dp[prevP.first][prevP.second][combPrev].first;//Der
            Wert des vorherigen Feldes zu dieser
            Kombination

        int maxV = bindingNothing;
        int predecessor = 0; //0 = unbesetzt, 1 = bund mit
            links, 2 = bund mit oben
        int oldMax = maxV;
        if (isZero(comb, c+offset) == false) { //wenn das
            eigene Feld besetzt werden darf
            if (c > 0 && isZero (comb, c-1+offset) ==
                false) { //wenn das links angrenzende Feld
                besetzt werden darf
                int bindingLeft = (prevV.second >= c ? 1 :
                    -1) + (nowV.second <= prevP.second ? 1
                    : -1);
                maxV = max (maxV, bindingLeft +
                    dp[r][c-1][combPrev & ~(1 <<
                    (c-1+offset))].first);
                predecessor = oldMax == maxV ? predecessor
                    : 1;
                oldMax = maxV;
            }
            if (r > 0 && isZero (comb, c) == false) {
                //wenn das oben angrenzende Feld besetzt
                werden darf
                pair <int, int> upV = fields[r-1][c];
```

```
                        int bindingUp = (upV.first >= r ? 1 : -1) +
                            (nowV.first <= (r-1) ? 1 : -1);
                        maxV = max (maxV, bindingUp +
                            dp[r-1][c][combPrev & ~(1 <<
                            (c))].first);
                        predecessor = oldMax == maxV ? predecessor
                            : 2;
                    }
                }
                dp[r][c][comb] = make_pair (maxV, predecessor);
            }
        }
    }
} while (improved() == true);
--steps;

ctr = 0;
isAdded = vector <vector <int> >(wLength, vector <int> (wLength, 0));
while (isFinished() == false) { //Alle hinzugefuegten Packete
    platzieren, ein Swap pro Zeiteinheit
    for (int r = 0; r < wLength; ++r) {
        for (int c = 0; c < wLength; ++c) {
            pair <int, int> goTo = fields[r][c];
            if (goTo.first != r || goTo.second != c) { //Wenn das
                Packet noch nicht an der richtigen Stelle ist...
                //Und wenn sein Verteilungsweg noch nicht verwendet
                    wird...
                bool fieldOcc = false;
                for (int i = min (r, goTo.first); i <= max (r,
                    goTo.first); ++i) {
                    fieldOcc |= isAdded[i][min(goTo,make_pair
                        (r,c)).second] > 0;
                }
                for (int i = min (c, goTo.second); i <= max (c,
                    goTo.second); ++i) {
                    fieldOcc |= isAdded[max(goTo,make_pair
                        (r,c)).first][i] > 0;
                }
                //Dann markiere seinen Verteilungsweg und fuege das
                    Packet in die Platzierungsliste ein
                if (fieldOcc == false) {
                    for (int i = min (r, goTo.first); i <= max (r,
                        goTo.first); ++i) {
                        isAdded[i][min(goTo,make_pair (r,c)).second] =
                            2;
                    }
                    for (int i = min (c, goTo.second); i <= max (c,
                        goTo.second); ++i) {
```

```
                    isAdded[max(goTo,make_pair (r,c)).first][i] =
                        2;
                }
                //Sowohl Pfad als auch umgekehrten Pfad hinzufuegen
                displaced.push_back (vector <pair <int, int> >
                        {make_pair (r, c), goTo, make_pair (ctr,
                            0), make_pair (r, c)});
                displaced.push_back (vector <pair <int, int> >
                        {goTo, make_pair (r, c), make_pair (ctr,
                            0), goTo});
                ++ctr;
            }
        }
    }
}
for (auto it = displaced.begin(); it != displaced.end();) {
    pair <int, int> pNow = (*it)[0],
                    goTo = (*it)[1];
    //In der Richtigen Reihenfolge swappen, damit sowohl Packet
        als auch Umkehrpacket den gleichen Weg verwenden
    if (goTo.first > pNow.first) {
        swap (fields[pNow.first][pNow.second],
            fields[pNow.first+1][pNow.second]); //Austauschen
        result[pNow.first][pNow.second] += 'S'; //Zeichenkette
            aktualisieren
        result[pNow.first+1][pNow.second] += 'N';
        ++(*it)[0].first; //Position des Paketes aktualisieren
    } else if (goTo.second < pNow.second) {
        swap (fields[pNow.first][pNow.second],
            fields[pNow.first][pNow.second-1]);
        result[pNow.first][pNow.second] += 'W';
        result[pNow.first][pNow.second-1] += 'O';
        --(*it)[0].second;
    } else if (goTo.first < pNow.first && (goTo.second <=
        pNow.second)) {
        swap (fields[pNow.first][pNow.second],
            fields[pNow.first-1][pNow.second]);
        result[pNow.first][pNow.second] += 'N';
        result[pNow.first-1][pNow.second] += 'S';
        --(*it)[0].first;
    } else if (goTo.second > pNow.second) {
        swap (fields[pNow.first][pNow.second],
            fields[pNow.first][pNow.second+1]);
        result[pNow.first][pNow.second] += 'O';
        result[pNow.first][pNow.second+1] += 'W';
        ++(*it)[0].second;
    } else {
        //unadd element
        pair <int, int> origin = (*it)[3];
        int subtracted = isAdded[origin.first][origin.second]-1;
```

```
            //Gesamten Pfad loeschen
            for (int i = min (origin.first, goTo.first); i <= max
                (origin.first, goTo.first); ++i) {
                isAdded[i][min(goTo,origin).second] = subtracted;
            }
            for (int i = min (origin.second, goTo.second); i <= max
                (origin.second, goTo.second); ++i) {
                isAdded[max(goTo,origin).first][i] = subtracted;
            }
            it = displaced.erase(it);
            continue;
        }
        //Spezialfallbehandlung, dass die Pakete von Pfad und
            umgekehrten Pfad sich treffen
        if (next(it) != displaced.end() && (*it)[2].first ==
            (*next(it))[2].first &&
            (*it)[2].second == 0) {
            int dist = (abs ((*next(it))[0].first-pNow.first)+
                       abs ((*next(it))[0].second-pNow.second));
            if (dist <= 2) { //Wenn noch ein Paket dazwischen, dann
                soll der umgekehrte Pfad warten
                if (dist == 1) { //Sonst wurde das Paket des
                    u8mgekehrten Pfades bereits geswappt, das muss
                    markiert werden
                    assert ((*next(it))[0].second < 10 &&
                        (*next(it))[0].first < 10);
                    switch (result[pNow.first][pNow.second].back()) {
                    case 'S': --(*next(it))[0].first; break;
                    case 'N': ++(*next(it))[0].first; break;
                    case 'W': ++(*next(it))[0].second; break;
                    case 'O': --(*next(it))[0].second; break;
                    }
                    ++(*it)[2].second;
                    ++(*next(it))[2].second;
                }
                ++it;
            }
        }
        ++it;
    }
    if (displaced.size() > 0) { //Schritt zur Gesamtloesung
        hinzufuegen
        ++steps;
        stepsV.push_back(fields);
    }

    for (int r = 0; r < wLength; ++r) {
        for (int c = 0; c < wLength; ++c) {
            if (result[r][c].length() < steps)
                result[r][c] += '_';
```

```
        }
    }

    vector <pair <int, int> > throwM = {make_pair (0,1),
                                    make_pair (1,0), make_pair
                                        (0,-1), make_pair(-1,0)};
                                        //Wurfmatrix
    vector <char> throwMS = {'O', 'S', 'W', 'N'};
        //Wurfmatrix-Zeichen
    vector <vector <bool> > usedNow (wLength, vector <bool>
        (wLength, false)); //Ist das Feld bereits in einen Swap
        involviert?
    for (int i = 1; i < wLength-1; ++i) {
        for (int j = 1; j < wLength-1; ++j) {
            for (int k = 0; k < throwM.size(); ++k) {
                int iN = i+throwM[k].first,
                    jN = j+throwM[k].second;
                 int sum = mDist (fields[i][j], //Welche Auswirkungen
                     haette ein Swap auf die Manhattan Distanz?
                       make_pair (i,j)) > mDist (fields[i][j],
                           make_pair (iN, jN)) ? 1 : -1;
                 sum += mDist (fields[iN][jN],
                       make_pair (iN,jN)) > mDist (fields[iN][jN],
                           make_pair (i, j)) ? 1 : -1;
                 if ((isAdded[i][j] == 0 && isAdded[iN][jN] == 0) &&
                     usedNow[i][j]==false && usedNow[iN][jN]==false &&
                     ((sum == 2 && randomFac > (rand()%10)) || (sum==0
                         && randomFac > (rand()%100))) &&
                     (fields[i][j].first != i || fields[i][j].second
                         != j) &&
                     (fields[iN][jN].first != iN ||
                         fields[iN][jN].second != jN)
                     ) { //Wurde das aktuelle Paket noch nicht
                         benutzt, ist es noch nicht auf seinem Ziel
                         und verringert ein Swap nicht die
                         Manhattan-Distanz?
                    //Dann Swappe, falls der Zufall zustimmt
                    result[i][j].back() = throwMS[k];
                    result[iN][jN].back() = throwMS[(k+2)%4];
                    swap (fields[i][j], fields[iN][jN]);
                    usedNow[i][j] = true;
                    usedNow[iN][jN] = true;
                 }
            }
        }
    }
}
alreadyFinished:
ofstream fieldOut (resFileName+".paketverteilung.txt");
for (int r = 0; r < wLength; ++r) {
```

```
    for (int c = 0; c < wLength; ++c) {
        fieldOut << r << ' ' << c << ' ' << result[r][c] << endl;
    }
}
fieldOut.close();

///GRAPHIKERZEUGUNG!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
cout << steps << endl;
//Felder ausgeben
for (int i = 0; i <= steps; ++i) {
    for (int j = 0; j < wLength; ++j) {
        for (int k = 0; k < wLength; ++k) {
            doc << Rectangle(Point(j*tLength,
                i*(tLength*wLength+10)+k*tLength), tLength, tLength,
                Color::Green, Stroke(1, Color::Blue));
        }
    }
}
//Texte ausgeben
for (int i = 0; i <= steps; ++i) {
    for (int j = 0; j < wLength; ++j) {
        for (int k = 0; k < wLength; ++k) {
            ostringstream sX, sY;
            sX << stepsV[i][j][k].first;
            sY << stepsV[i][j][k].second;
            string sCoords = sX.str()+ "|" + sY.str();
            doc << Text(Point(k*tLength+1,
                10+i*(tLength*wLength+10)+j*tLength), sCoords,
                Fill(Color::Yellow), Font (4));
        }
    }
}

// Red image border.
Polygon border(Stroke(1, Color::Red));
border << Point(0, 0) << Point(dimensions.width, 0)
    << Point(dimensions.width, dimensions.height) << Point(0,
        dimensions.height);
doc << border;

//Pfeile ausgeben
for (int j = 0; j < steps; ++j) {
    for (int r = 0; r < wLength; ++r) {
        for (int c = 0; c < wLength; ++c) {
            int xA = c*tLength+(tLength/2),
                yA = j*(tLength*wLength+10)+r*tLength+(tLength/2);
            switch (result[r][c][j]) {
            case 'S': paintArrow (Point (xA, yA), Point (xA,
                yA+tLength)); break;
```

```
                case 'N': paintArrow (Point (xA, yA), Point (xA,
                    yA-tLength)); break;
                case 'W': paintArrow (Point (xA, yA), Point
                    (xA-tLength,yA)); break;
                case 'O': paintArrow (Point (xA, yA), Point
                    (xA+tLength,yA)); break;
                }
            }
        }
    }
    doc.save();
}
```

montecarlo.bat

```
ECHO OFF
FOR /L %%A IN (1,1,1000) DO (
"Missglueckte Drohnenlieferung.exe" 9 result%%A.svg
```

2.2 Quellen

Zur Generierung der SVG-Datei habe ich die Bibliothek simple-svg von Mark Turney benutzt: `https://code.google.com/archive/p/simple-svg`