InformatiCup 2023: Profit!

Einreichung Team uwunauten

Joana Bergsiek
Hasso-Plattner-Institut, Universität Potsdam
joana.bergsiek@student.hpi.de

Dominik Meier
Hasso-Plattner-Institut, Universität Potsdam
dominik.meier@student.hpi.de

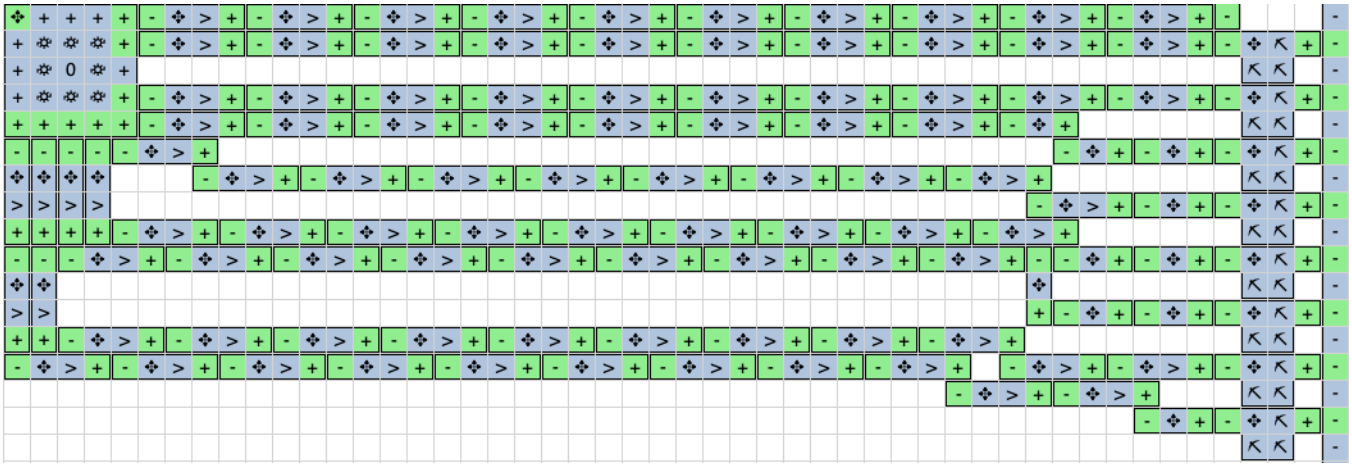

Abbildung 1: Ausschnitt aus einer Lösung

ZUSAMMENFASSUNG

Wir präsentieren unsere Lösung für den jährlichen InformatiCup der Gesellschaft für Informatik. Die Aufgabe „Profit!“ erfordert es, komplexe Teillösungen zu einer globalen Lösung für die Erstellung von Produkten in Fabriken zusammenzufügen. Durch Modellierung der dynamischen Bewegungsabläufe von Ressourcen zu stückweise linearen Funktionen, abstrahieren wir von einer rundenbasierten Simulation und sind somit in der Lage, die erreichbare Punktzahl einer Platzierung in konstanter Zeit abzuschätzen. Unsere Implementierung ist in modernem C++20 geschrieben, auf eine schnelle Verarbeitungsgeschwindigkeit optimiert, und kann somit tausende Platzierungen pro Sekunde explorieren. Zusammenhangskomponenten ermöglichen uns, mithilfe von *Divide and Conquer*, den Problemraum einzuschränken und Teilprobleme pro Komponente zu lösen. Dabei nutzen wir Eigenschaften, wie Distanzen, aus, um das Spielfeld aus verschiedenen Sichten zu visualisieren. Statt auf Anhieb eine feste Menge an Produkten und Verbindungen zu realisieren, verwenden wir verschiedene Greedy-Ansätze zur Auswahl einzelner Realisierungen. Durch eine Schleife fügen wir iterativ neue Verbindungen hinzu, bis die Blockaden weiteres Hinzufügen verhindern. Hat sich die erreichte Punktzahl verbessert, oder die Latenz verringert, werden die Änderungen vorgenommen. Damit erreichen wir eine Realisation, die mit Heuristiken lokal optimierte Produktionen baut.

1 EINLEITUNG

1.1 Vorstellung des Problems

Bei der Aufgabe „Profit!“ geht es darum, durch das Produzieren von Produkten eine möglichst hohe Punktzahl zu erreichen. Das Spielfeld ist ein Raster aus Quadraten. Es beinhaltet zu Beginn Minen und Hindernisse, welche die Landschaft definieren. Minen beherbergen Ressourcen, die einen Subtyp von eins bis acht haben können. Die Herstellung eines Produkts benötigt eine Kombination verschiedener Ressourcen-Subtypen mit unterschiedlicher Anzahl. Dabei erhöht die erfolgreiche Fertigstellung eines Produkts die Punktzahl. Verschiedene Produkte haben hierbei einen unterschiedlichen Wert.

Auch bei den Produkten begrenzen sich die möglichen Typen auf acht. Für die Herstellung von Produkten müssen Bauteile platziert werden. Minen bauen Ressourcen an einer Lagerstätte ab; Fabriken nehmen die Ressourcen in Empfang und realisieren einen der acht gegebenen Produkt-Typen. Verbinder, Förderbänder, oder weitere Minen können die abgebauten Ressourcen hierbei transportieren. „Runden“ sind eine zeitliche Einheit, welche die Bewegung der Ressourcen diktiert. Bauteile können ihre Aktion - Abbau, Transport, Produktion - nur einmalig pro Runde ausführen. Ressourcen sind ab ihrem Abbau demnach in einem dynamischen Ablauf, welcher sich pro Runde verändern kann, und auf die maximale Rundenzahl der Eingabe beschränkt ist. Gemeinsam mit einem Echtzeit-Limit für die Ausführung der Simulation, in der eine Lösung erwartet wird, ist die Ausführungsdauer beschränkt.

Die Schlüsselfrage ist also: Wie müssen die Bauteile platziert werden, um innerhalb der zeitlichen Grenzen eine hohe Punktzahl zu erreichen?

1.2 Teilprobleme

Bei der Analyse von „Profit!“ sind wir auf zwei übergeordnete Teilprobleme gestoßen, deren Lösungen für eine insgesamt gute Lösung zusammengefügt werden.

- **Produktbestimmung**: Welche Produkte können wir kreieren? Von welchen Produkten kann sich ein großer Gewinn erhofft werden?
- **Platzierungen**: Wie müssen die Bauteile auf dem Feld platziert werden, um die Produkte zu realisieren? Auf welche Art lässt sich das Feld weitreichend eingrenzen? Wie nutzen wir es möglichst gut aus?

Eine grundlegende algorithmische Herausforderung ist, dass beide Probleme eng miteinander verflochten sind. So ist es zum Beispiel nicht immer möglich, alle Lagerstätten zu verbinden. Hindernisse und existierende Verbindungen können den Weg blockieren. Zwar sind Hindernisse von Beginn an bekannt, doch Blockaden entstehen durch zugefügte Platzierungen. In einer idealen Welt würden die zugefügten Verbindungen zwischen Lagerstätten und Fabriken möglichst wenig nachfolgende Verbindungen blockieren.

1.3 Ähnlichkeiten zu bekannten Problemen und verwandte Arbeiten

Warum ist das Lösen von „Profit!“ eine besonders schwere Herausforderung? Die in 1.2 erwähnten Teilprobleme sind alleinstehend bereits komplex. Durch gegenseitige Abhängigkeiten der Teilprobleme wird es umso schwieriger, eine optimale Lösung zu finden.

Die Auswahl an Produkten ähnelt dem NP-schweren Knapsack Problem [11]. Hierzu betrachten wir die verfügbaren Ressourcen als Einschränkung dafür, wie viele Produkte wir bauen können. Statt im klassischen Knapsack, in dem wir ein maximales Gewicht definieren, haben wir in „Profit!“ pro Subtyp eine maximale Anzahl von Ressourcen. Die Aufgabe ist nun, eine Menge von Produkten zu finden, welche die Punktzahl maximiert, aber innerhalb der verfügbaren Ressourcen bleibt.

Um anschließend eine Auswahl an Produkten auf dem Spielfeld zu realisieren, muss entschieden werden, wo Fabriken platziert werden. Dabei sollte die Platzierung möglichst nah an allen benötigten Lagerstätten verfügbar sein, darf allerdings auch nicht wichtige Verbindungen blockieren. Dieses Platzierungsproblem erinnert an Floorplanning, welches zum Beispiel häufig beim Chipdesign auftritt[7]. Viele Varianten des Floorplannings sind NP-schwer, was bedeutet, dass wahrscheinlich kein effizienter Algorithmus zur Lösung existiert.

Nach dem Platzieren der Fabriken auf dem Spielfeld müssen die Lagerstätten mit den Fabriken verbunden werden. Das Verbinden erinnert an Probleme aus der Kategorie der Vehicle Routing Probleme. Diese sind auch in vielen Fällen NP-schwer [18].

Durch die Kombination aus verschiedenen, schwierigen Problemen, war unsere Lösungsstrategie nicht, einen optimalen Algorithmus zu finden. Stattdessen entwickelten wir Heuristiken, die schnell eine große Anzahl an Probleminstanzen lösen können. Sie sollen lokale Optima visualisieren, um Platzierungsentscheidungen zu erläutern. Mit dem Endresultat unserer Strategie bieten wir eine Basis für weitere, manuelle Optimierungen an.

Das Einsetzen von Heuristiken kann problematisch werden, da diese nicht immer eine global optimale Lösung auswählen. Als Strategie haben wir deshalb versucht, möglichst performante Implementierungen von

den Heuristiken umzusetzen. Das ermöglicht uns, während der Rechenzeit Teilprobleme nicht nur einmal zu lösen, sondern viele verschiedene Lösungen durchzuprobieren. Weil die Heuristiken nicht alle Beschränkungen beachten, kann es vorkommen, dass sich Teillösungen in tieferen Schritten als unmöglich herausstellen. In diesem Fall wird die Teillösung verworfen, zu einem Teilschritt zurückgesprungen, und eine neue Lösung probiert. Dieser Ansatz nennt sich Backtracking[14].

Bevor wir unseren Ansatz vorstellen, thematisieren wir zunächst unsere Abstraktion der Spezifikation, die das Spielfeld und seine dynamischen Abläufe aus einer datenorientierten Sicht betrachtet.

1.4 Formalisierung

Zwar beschreibt die Spezifikation die finale Punktzahl als Ergebnis eines rundenbasierten Szenarios, doch können wir auf das Konzept von Runden vollständig verzichten. Da die Platzierung von Bauteilen bereits vor dem Start der ersten Runde feststeht, und nachträgliche Platzierungen unmöglich sind, reduziert sich das Problem auf Distanzen. Alle Bauteile transportieren Ressourcen in einer konstanten Geschwindigkeit von einer Runde. Deswegen ist die Ankunftszeit von Ressourcen an einem Ausgang gleich der Anzahl an Bauteilen einer Verbindung. Schauen wir uns dies an einem Beispiel dargestellt in Abbildung 2 an. Zwei Lagerstätten A und B sind mit einer Fabrik F verbunden. B mit Subtyp 1 hat mehrere Pipelines, die unterschiedlich lange benötigen, um erste Ressourcen an F zu liefern. A liefert in Runde fünf erste Ressourcen an F, B mit Verbindung $B1$ in fünf, mit $B2$ in sieben. A hat initial $capacity_A = 3 \cdot 3 \cdot 5 = 45$ Ressourcen, $capacity_B = 8 \cdot 3 \cdot 5 = 120$. Ein Produkt mit dem Subtyp 0 und den Anforderungen $[10 \cdot 0, 10 \cdot 1]$ ist zehn Punkte wert. Da Minen einen Durchsatz von bis zu drei Einheiten haben, können wir berechnen, dass A nach $t_A = \frac{45}{3} = 15$ Runden, und B nach $t_B = \frac{120}{3*2} = 20$ Runden keine Ressourcen mehr zur Verfügung haben. Vereint mit den Längen der Verbindungen $|p_{A1}|, |p_{B1}|, |p_{B2}|$ berechnen wir in Tabelle 1 die Anzahl zu F überbrachter Ressourcen. Die drei Einheiten Durchsatz von Minen seien $throughput$. Die Menge an ausgehenden Verbindungen einer Lagerstätte L bezeichnen wir als P_L.

Wir beobachten, dass der Abbau von Ressourcen und der Transport zu einer Fabrik stückweise lineare Funktionen sind. Für jede Verbindung, die wir im Folgendem

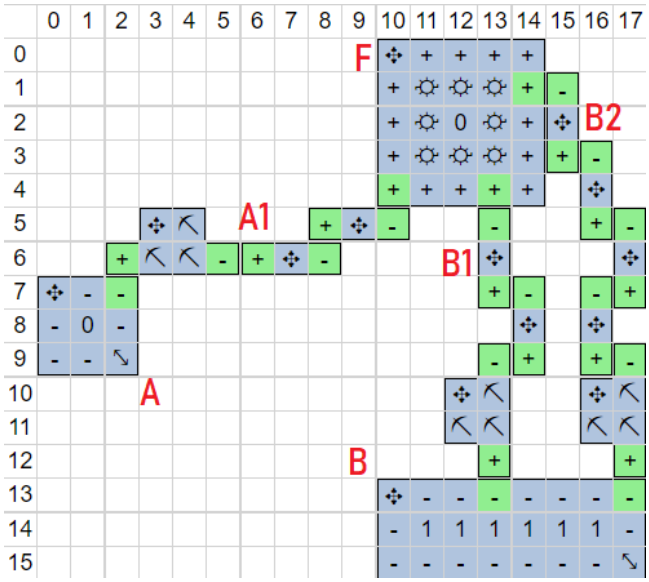

Abbildung 2: Beispielhafte Platzierung von Lagerstätten A, B, Verbindungen $A1$, $B1$, $B2$, und Fabrik F.

Pipeline nennen, werden so lange *throughput* Einheiten pro Runde und Pipeline abgebaut, bis die jeweilige Lagerstätte L leer ist. Angenommen, die Ressourcen werden fair über alle Pipelines von L aus verteilt, ist dieser Moment zum Zeitpunkt t_L (s. Gleichung 1) erreicht.

$$t_L = \left\lfloor \frac{capacity_L}{throughput \cdot |P_L|} \right\rfloor \tag{1}$$

Pipelines beliefern Fabriken erstmals in Runde $d_{p_{Li}} = |p_{Li}| + 1$. Die Anzahl der Bauteile definiert eine zu überwindende Distanz. Wir betrachten diese Beziehungen, Abbau $a_{p_{Li}}$ und Zulieferung $z_{p_{Li}}$, für eine Pipeline p_{Li} zu einer gegebenen Runde als folgende Funktionen in Gleichung 2 und Gleichung 3.

$$a_{p_{Li}}(R) = \begin{cases} throughput \cdot R & \text{if } R <= t_L \\ capacity_L & \text{if } R > t_L \end{cases} \tag{2}$$

$$z_{p_{Li}}(R) = \begin{cases} 0 & \text{if } R <= d_{p_{Li}} \\ t_L * throughput & \text{if } R >= t_L + d_{p_{Li}} \\ throughput \cdot (R - d_{p_{Li}}) & \text{else} \end{cases} \tag{3}$$

Tabelle 1: Die Menge an Ressourcen pro Pipeline, die Fabrik F bis zu Zeitpunkt R erreicht. Für jede Runde R berechnen wir $throughput \cdot (R - |p_{Li}| - 1), p_{Li} \in P_L$ mit Berücksichtigung, zu welchen Runden die ersten und letzten Ressourcen ankommen.

Von	$R = 5$	$R = 10$	$R = 20$	$R = 30$
$A1$	$3 \cdot (5 - 3 - 1) = 3$	$3 \cdot (10 - 3 - 1) = 18$	$3 \cdot 15 = 45$	$3 \cdot 15 = 45$
$B1$	$3 \cdot (5 - 3 - 1) = 3$	$3 \cdot (10 - 3 - 1) = 18$	$3 \cdot (20 - 3 - 1) = 48$	$3 \cdot 20 = 60$
$B2$	$3 \cdot 0 = 0$	$3 \cdot (10 - 5 - 1) = 12$	$3 \cdot (20 - 5 - 1) = 42$	$3 \cdot 20 = 60$

Mit dieser Vereinfachung lassen sich Punktzahlen ohne Simulation berechnen. Perfekt ist diese Variante gegenüber dem Simulieren jedoch nicht. Können die Ressourcen nicht gleich verteilt werden, muss eine Priorität unter den angrenzenden Pipelines zum Zeitpunkt t'_L, wie in Gleichung 4 definiert, existieren.

$$t'_L = \left\lceil \frac{capacity_L}{throughput \cdot |P_L|} \right\rceil \tag{4}$$

Die Priorisierung von Pipelines haben wir nicht implementiert. Annäherungen an die eigentliche Punktzahl sind ausreichend, um eine relative Ordnung zwischen Pipelines und ihrem Gewinn zu definieren. Die offizielle Implementierung von „Profit!“, welche die Lösungen auswertet, kann eine andere Punktzahl ausrechnen, die unsere Punktzahl gegebenenfalls nur weiter erhöht.

2 LÖSUNGSANSATZ

Im Folgenden wollen wir unsere Modellierungen und Überlegungen für beide in Unterabschnitt 1.2 erwähnten Probleme vorstellen. Wir setzen auf eine iterative Realisierung, die lokale Optima findet.

Als Vorbereitung benötigen wir vorab jedoch eine Repräsentation des Felds, welche die Platzierungen der Objekte widerspiegelt. Wenn wir über das Erkennen von Hindernissen, Kollisionen oder Platzieren von Objekten sprechen, lesen wir von, oder schreiben auf diese Repräsentation. Wir nennen diese Repräsentation die *OccupancyMap*. Die Karte kann an einer Zelle eine Belegung der folgenden Objekte annehmen.

- `EMPTY`
- `BLOCKED`
- `CONVEYOR_CROSSING`
- `DOUBLE_CONVEYOR_CROSSING`
- `INGRESS`
- `EGRESS`

2.1 Produktbestimmung

Zunächst werden die Produkte festgelegt. Dazu fangen wir mit einer räumlichen Analyse des Spielfelds an und teilen es in Zusammenhangskomponenten ein. Innerhalb dieser nutzen wir dann eine Heuristik, welche die räumliche Dimension ignoriert.

2.1.1 Erreichbarkeit von Zellen und Zusammenhangskomponenten. Die Produktbestimmung beginnt damit, herauszufinden, welche Zellen von den Lagerstätten aus erreichbar sind. Unerreichbare Zellen sollen in der Berechnung ignoriert werden und den Lösungsraum einschränken. Darüber hinaus gibt uns die Erreichbarkeit Auskunft darüber, welche Lagerstätten miteinander für die Produktherstellung kombiniert werden können. Diese Kombination an Lagerstätten soll einmalig ermittelt werden. Folgend könnten wir die Invariante garantieren, dass jede Untermenge an Lagerstätten innerhalb einer Kombination valide ist. Dies spart uns sich wiederholende Überprüfungen. Dafür eignen sich eine Breitensuche und eine Unterteilung des Feldes in zusammenhängende Lagerstätten, sie sich jeweils erreichen können.

Die generelle Idee ist, das Feld als Graph zu interpretieren, bei dem eine Zelle mit ihrer horizontalen und vertikalen Nachbarschaft verbunden ist. Die Breitensuche beginnt bei den Ausgängen einer Lagerstätte angrenzenden Zellen. Alle erreichten Zellen werden gespeichert. Im „Profit!“-Szenario erreichen sich zwei Zellen jedoch nur über Bauteile, wie zum Beispiel das Förderband. Die Suche fügt in einem Iterationsschritt demnach nicht die horizontale Nachbarschaft hinzu, sondern die möglichen Fortbewegungen mit den Bauteilen Mine, Förderband, und Verbinder. Die an den Ausgängen angrenzenden Zellen werden als erreicht markiert. Im ersten Schritt werden exklusiv Minen als Fortbewegung überprüft, da nur Minen-Eingänge an

Lagerstätten-Ausgängen sein dürfen. Mit der angepassten Breitensuche erhalten wir den Vorteil, nebenbei die minimalen Distanzen zu einem Feld zu errechnen. Für eine Lagerstätte tragen wir uns ihre Distanzen zu den anderen Zellen in einer *DistanceMap* ein, die wir als Heuristik in Unterunterabschnitt 2.2.1 verwenden. Abbildung 3 stellt diese veränderte Breitensuche mit der klassischen Variante gegenüber.

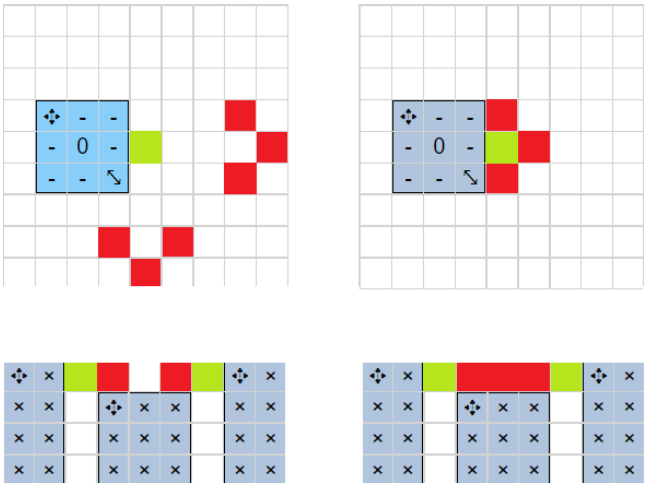

Abbildung 3: Iterationsschritt der angepassten Breitensuche links vs. Nachbarschaft rechts. Oben sehen wir die erreichbaren Felder in rot vom grünen Startfeld mit Distanz eins. Unten sehen wir die generell erreichbaren Felder der oberen, schmale Reihe in einem Szenario mit nah benachbarten Ecken. Die angepasste Suche grenzt den Lösungsraum realistischer ein.

Für eine *vollständig* korrekte Menge an Lagerstätten-Distanzen, müssten wir Kollisionen und Selbstüberschneidungen zu den vorherigen Schritten beachten. Bezüglich der Distanzen genügt eine Annäherung ohne Überschneidungen, die unkompliziert in $O(Feldweite * Feldhoehe)$ berechnet wird. Jede Zelle in dem Feld kann nur einmalig der Warteschlange hinzugefügt und betrachtet werden. Wichtig ist die Korrektheit der Zusammenhangskomponenten.

Zusammenhangskomponenten sind in der Graphentheorie eine Submenge an Knoten, die sich durch einen Pfad erreichen können. In unserem Szenario Submengen an Lagerstätten. Jede erreichte Zelle der Lagerstätte A muss ebenfalls von Lagerstätte B erreicht werden können. Mithilfe von Zusammenhangskomponenten lösen wir die Aufgabenstellung nicht für das gesamte Feld, sondern grenzen den Raum an Möglichkeiten mit *Divide and Conquer* pro Komponente ein. In anderen Worten wird das Feld in Zusammenhangskomponenten partitioniert. Es bietet sich die Umsetzung mittels *Union-Find-Struktur* [5] an, da jede Lagerstätte einer Komponente zugeordnet wird. Die kleinstmögliche Zusammenhangskomponente ist eine Lagerstätte selbst, die größtmögliche Komponente beinhaltet alle Lagerstätten auf dem Feld. Die Vereinigung zweier Lagerstätten passiert neben der Breitensuche. Während der Breitensuche betrachten wir Lagerstätten vereinzelt - um eine Vereinigung durchführen zu können, benötigen wir eine Hilfsstruktur. Für diese Aufgabe definieren wir uns eine *ComponentMap*. Ist eine Zelle erstmals erreicht, trägt sich die dazugehörige Lagerstätte mit ihrer Komponente an der Position in der *ComponentMap* ein. Nachfolgende Lagerstätten vereinen sich mit der dort gefundenen Komponente. Durch eine Pfadkompression haben wir pro Iterationsschritt der Breitensuche $O(log|Lagerstaetten|)$ zusätzliche Laufzeit. Dies bringt die Laufzeit der Distanzberechnung auf:

$$O(Feldweite * Feldhoehe * log|Lagerstaetten|)$$

Ein Pseudocode ist in Algorithmus 1 aufgeführt.

Algorithm 1 BFS

```
Require: dep, occupancies, union, depId
  distances ← occupancies.dimensions
  queue ← ()
  for each dep egress, each rotation do
      if mine placement valid then      ▷ e.g. collisions
          distances, queue ← update reachability
      end if
  end for
  while queue not empty do
      reached ← queue.front
      union ← merge(reached, depId)
      for each rotation do
          if placeable valid then       ▷ e.g. conveyors
              distances, queue ← update reachability
          end if
      end for
  end while
  return distances
```

2.1.2 Produktauswahl. Ab der Produktauswahl betrachten wir das Problem nur noch innerhalb einer Zusammenhangskomponente. In diesem Schritt entscheiden wir uns für das Vernachlässigen der räumlichen Dimension, um dem Problem Komplexität zu entziehen. Hierdurch kann es uns passieren, dass wir Produkte auswählen, die unmöglich auf dem Spielfeld zu realisieren sind. Durch Backtracking können wir jedoch, falls wir in dieses Problem laufen, eine andere Produktauswahl ausprobieren, falls eine Pipeline-Realisierung für ein ausgewähltes Produkt erfolglos ist.

Die Vernachlässigung der räumlichen Komponente reduziert unser Problem auf die Verfügbarkeit verschiedener Ressourcen-Subtypen innerhalb einer Zusammenhangskomponente. Das Ziel ist nun, diese Ressourcen zu Produkten mit der besten möglichsten Punktzahl zusammenzufügen.

Wie in Unterabschnitt 1.3 angestoßen, ist diese Situation ähnlich zum klassischen Knapsack-Problem. Für die Reduzierung der Produktauswahl auf Knapsack betrachten wir die Produkte als einzupackende Gegenstände, und die Punktzahl von Produkten analog zu den Punktzahlen für Gegenstände. Durch die acht verschiedenen Ressource-Subtypen ergeben sich acht verschiedene Maximalgewichte. Diese Abwandlung des Knapsack-Problems nennt sich Multi-Dimensional Knapsack Problem (*MDKP*), und ist NP-vollständig [11]. Beide Probleme, der generelle Knapsack und MDKP, lassen sich mit dynamischer Programmierung in pseudopolynomieller Laufzeit lösen [11]. Im Fall von MDKP ist das jedoch deutlich aufwändiger als beim klassischen Knapsack, und kann, zum Beispiel, durch eine Reduktion auf ein Problem der linearen Programmierung geschehen [3].

Statt einer konkreten, besten Lösung, interessiert uns eine Rangfolge der möglichen Produkte. Sobald die räumliche Komponente im nächsten Schritt wieder einbezogen wird, und eine Verbindung erfolglos versucht wurde, können wir so Backtracking nutzen, und das nächste Produkt der Ordnung ausprobieren. Aufgrund des hohen Aufwandes der Implementierung und der großen Laufzeit haben wir somit keine exakte Lösung errechnet, sondern eine Heuristik aus der Literatur benutzt [1]. Unsere umgesetzte Heuristik wird *pech* genannt und ist ein iteratives Greedy-Verfahren. Es fügt immer mehr Elemente von dem Produkt hinzu, welches am meisten Punkte gäbe, wenn nur noch dieses Produkt zur aktuellen Lösung hinzugefügt werden würde. Nachdem alle Ressourcen aufgebraucht sind, bricht die Heuristik ab. Die Lösung ist eine gewichtete Auswahl der acht Produkt-Subtypen. Ein großer Vorteil der Heuristik ist dabei, dass diese effizient wiederholt ausgeführt werden kann. Die genaue Laufzeit hängt dabei von der konkreten Instanz ab, ist aber pseudopolynomiell [1]. Schlägt eine Realisierung der Pipeline fehl, entfernt der Algorithmus das jeweilige Produkt und führt die *pech* Heuristik erneut aus.

2.2 Platzierungen

Nach der Festlegung, welches Produkt durch die Fabriken schlussendlich produziert werden soll, geht es an die konkrete Realisierung. Es müssen Fabriken, Minen und Förderbänder auf das Feld platziert werden. Verbinder sind wie in Unterabschnitt 6.3 beschrieben nicht Teil unserer derzeitigen Lösung. Konkret verändern wir ab hier den Zustand der *OccupancyMap*. Ähnlich zur Produktauswahl fokussieren sich die Lösungen auf die Umsetzungen einzelner, und nicht mehrerer Verbindungen zu einem Zeitpunkt.

2.2.1 Platzieren von Fabriken. Fabriken werden mithilfe der zuvor erstellten *DistanceMaps* zuerst platziert. Sie sind eine gute Metrik, Positionen innerhalb einer Zusammenhangskomponente zu bestimmen, die von allen Lagerstätten in gleicher, beziehungsweise ähnlicher, Distanz erreicht werden können. Als Gegenstück zur Vernachlässigung des Raums in der Produktauswahl, vernachlässigen wir in diesem Schritt die Anzahl benötigter Ressourcen. Dies ist für Szenarien wie *Task3* nicht der optimale Weg. Benötigt ein Produkt bedeutend mehr von einer Ressource, als von den anderen, empfiehlt sich eine größere Nähe zu den dazugehörigen Lagerstätten. Im Fall von *Task 3* sollten Fabriken idealerweise näher zu der größeren Lagerstätte oben links platziert sein. Somit würde sich die anfängliche Ankunfts-Latenz der begehrten Ressource 0 verringern und es könnten mehr Produkte hergestellt werden. Wir streben jedoch eine Greedy-Lösung an, die alle Lagerstätten gleich priorisiert. Für *Task 3* bedeutet dies eine Platzierung, und somit Verschlechterung, in der Mitte des Felds. Szenario *Task 1*, welches ein Produkt mit gleichmäßiger Verteilung beschreibt, profitiert andererseits von dem Ansatz.

Unsere Idee zur Platzierung ähnelt der Erzeugung einer *HeatMap* [15] und dem Finden von *Hot Spots*. Ein

Hotspot ist ein Bereich in unserem Feld, welcher minimale Distanzen zu allen Lagerstätten aufweist. Eine HeatMap ist in unserer Anwendung eine vereinte *DistanceMap* mehrerer Lagerstätten. Das Vereinen mehrerer *DistanceMaps* setzt pro Zelle das Maximum aller Lagerstätten. Da die *DistanceMaps* die minimalen Distanzen zu Zellen beschreiben, repräsentieren sie in dem vereinten Ergebnis die Mindest-Distanz, bis alle Lagerstätten die Zelle erreichen. Abbildung 4 stellt die vereinten Distanzen für *Task 1* dar. Die Vierer- und Fünfer-Distanzen des Ergebnisses schließen wir mangels Platz für eine Fabrik als Hot Spots aus. Die Intuition wäre, eine Zelle mit der nächst-niedrigen Distanz von Sechs als Platzierung einer Fabrik auszuwählen. Allerdings ergibt nicht jede Sechs die niedrigste Mindest-Distanz. Eine an Position $(6, 6)$ platzierte Fabrik ist mit fünf Einheiten am weitesten von der oben rechten Lagerstätte entfernt. Platziert in $(11, 11)$ erhöht sich die Distanz auf sechs. Diese Inkonsistenz ist mit den Dimensionen einer Fabrik über eine Zelle hinaus begründet. Die Distanz zu einer Fabrik ist die kürzeste Distanz zu einem ihrer, am Rande liegenden, Eingängen. Wir passen das vereinte Ergebnis so an, dass Zellen die Mindest-Distanzen beschreiben, bis die Fabriken, beziehungsweise Objekten mit Dimensionen (w, h), an den Positionen erreicht werden. Algorithmus 2 beschreibt das Verfahren in Pseudocode.

Das Vereinen der *DistanceMaps* ist eine teure Operation. Für jede Zelle betrachten wir die $2 \cdot w + 2 \cdot (h - 2)$ Randzellen des zu platzierenden Objekts der Breite w und der Höhe h. Dies tun wir für jede *DistanceMap* einer Zusammenhangskomponenten, deren Anzahl $|Z|$ sei. Es ergibt sich eine Zeitkomplexität von:

$$O(Feldweite \cdot Feldhoehe \cdot w \cdot h \cdot |Z|) \qquad (5)$$

In unserem Fall ist das Objekt eine Fabrik mit 16 Randzellen. w, h kürzen sich in der Notation als Konstanten dann zwar heraus, ergeben in einem maximal großen Feld trotzdem $100 \cdot 100 \cdot 16 = 160\,000$ Zugriffe pro *DistanceMap*. Viele Lagerstätten treiben diese Zahl schnell in den Millionenbereich. Wir führen das Vereinen dementsprechend pro Zusammenhangskomponente nur einmal aus und speichern uns das Ergebnis für die kommenden Platzierungen. Durch die seltene Ausführung akzeptieren wir die exzessivere Menge an Operationen.

In dem vereinten Ergebnis wissen wir noch nicht, ob ein Objekt bei Platzierung an einer Zelle mit seiner Umgebung kollidiert. Es werden nur die Zellen als nicht

Algorithm 2 Merge Distance Maps

```
Require: distances, reachable = (w, h)
  result ← distances[−].dimensions      ▷ any suffices
  for each cell in result do
      max ← distances[−][cell]
      for each map do
  max ← max(max, min_to_at(object, cell, map))
      end for
  end for
  return result

  procedure MIN_TO_AT(reachable, handle, map)
      if map[handle] == NOT_REACHABLE then
          return NOT_REACHABLE
      end if
      min ← map[handle]
      for each egress of reachable at handle do
  min ← min(min, map[egress])
      end for
      return min
  end procedure
```

erreichbar gesetzt, welche es schon vorher waren. Für die Bestimmung geeigneter Platzierungen definieren wir eine *PlacementMap*. Sie ist eine Bitmap, die für eine Zelle aussagt, ob das Objekt ohne Kollisionen platziert werden kann. Sie funktioniert durch einen horizontalen und einen vertikalen Scan für je w, h nachfolgende Zellen. Dies bleibt für Fabriken in linearer Abhängigkeit zu der Feldgröße und wird für jede Platzierung erneut berechnet. Es wird die minimale Distanz in der vereinten *DistanceMap* gewählt, die laut *PlacementMap* ebenfalls erreichbar ist. Falls eine geeignete Zelle gefunden wird, wird die *OccupancyMap* aktualisiert. Für Szenarien wie *Task 2* finden wir zuverlässig diejenigen Zellen, in der eine Fabrik für eine Produktion platziert werden *muss*. Das Szenario begründet unsere Motivation, Fabriken zuerst zu platzieren. Die Verbindungen folgen darauf.

2.2.2 Platzieren von Pipelines. Die Platzierung von Bauteilen funktioniert ähnlich wie die Breitensuche nach erreichbaren Feldern in Kapitel 2.1.1. Zum Verbinden nutzen wir das Konzept einer Pipeline. Diese ist immer eine Verbindung von genau einer Lagerstätte zu einer Fabrik. Zum Realisieren einer Pipeline werden zuerst alle Zielfelder in einer *TargetMap* markiert. Das sind

```
n,  n,  n,  n,  n,  n,  4,  4,  4,  5,  5,  5,  5,  6,  6,  6,  6,  7,  7,  7,  7,  8,  n,  n,  n,  n,  n,  n,  n,  n,
n,  n,  n,  n,  n,  n,  n,  4,  4,  4,  5,  n,  n,  6,  6,  6,  7,  7,  7,  7,  8,  n,  n,  n,  n,  n,  n,  n,  n,  n,
n,  n,  n,  n,  n,  n,  n,  5,  4,  4,  4,  n,  n,  7,  7,  7,  7,  7,  7,  8,  8,  n,  n,  n,  n,  n,  n,  n,  n,  n,
n,  n,  n,  n,  n,  n,  n,  5,  5,  4,  4,  n,  n,  7,  7,  7,  7,  7,  8,  8,  8,  n,  n,  n,  n,  n,  n,  n,  n,  n,
n,  n,  n,  n,  n,  n,  n,  5,  5,  5,  4,  n,  n,  7,  7,  7,  7,  8,  8,  8,  8,  n,  n,  n,  n,  n,  n,  n,  n,  n,
n,  n,  n,  n,  n,  n,  n,  5,  5,  5,  5,  n,  n,  7,  7,  7,  8,  8,  8,  8,  9,  n,  n,  n,  n,  n,  n,  n,  n,  n,
8,  n,  n,  n,  n,  n,  6,  6,  5,  5,  5,  n,  n,  8,  8,  8,  8,  8,  8,  9,  9,  n,  n,  n,  n,  n,  n,  n,  n,  n,
8,  7,  7,  7,  7,  6,  6,  6,  6,  5,  5,  n,  n,  8,  8,  8,  8,  8,  9,  9,  9,  n,  n,  n,  n,  n,  n,  n,  n,  n,
8,  8,  7,  7,  7,  7,  6,  6,  6,  6,  5,  n,  n,  8,  8,  8,  8,  9,  9,  9,  9, 10,  n,  n,  n,  n,  n,  n,  n,  n,
8,  8,  8,  7,  7,  7,  7,  6,  6,  6,  6,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,
8,  8,  8,  8,  7,  7,  7,  7,  6,  6,  6,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,  n,
9,  8,  8,  8,  8,  7,  7,  7,  7,  6,  6,  6,  7,  7,  7,  7,  8,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11,
9,  9,  8,  8,  8,  8,  7,  7,  7,  7,  6,  7,  7,  7,  7,  8,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11,
9,  n,  n,  n,  n,  n,  8,  7,  7,  7,  7,  7,  7,  7,  8,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11,
n,  n,  n,  n,  n,  n,  n,  8,  7,  7,  7,  7,  7,  8,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 12,
n,  n,  n,  n,  n,  n,  n,  8,  8,  7,  7,  7,  8,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 12, 12,
n,  n,  n,  n,  n,  n,  n,  8,  8,  8,  7,  8,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 12, 12, 12,
n,  n,  n,  n,  n,  n,  n,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 12, 12, 12, 12,
n,  n,  n,  n,  n,  n,  n,  9,  8,  8,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 12, 12, 12, 12, 13,
n,  n,  n,  n,  n,  n,  9,  9,  9,  8,  8,  8,  9,  9,  9,  9, 10, 10, 10, 10, 11, 11, 11, 11, 12, 12, 12, 12, 13, 13
```

Abbildung 4: Eine vereinte *DistanceMap* für die Zusammenhangskomponente aus den Lagerstätten von *Task 1*. *n* bedeutet, dass die jeweiligen Zellen nicht erreicht werden können.

die Randfelder der Fabrik, jedoch auch mögliche Eingänge einer schon vorhandenen Pipeline, an die sich die Pipeline anschließen kann.

Damit wir nach dem Erreichen eines Zielfeldes zurückrechnen können, welche Bauteile wir benutzt haben, speichern wir uns beim Setzen jedes Bauteils die Verbindungen zu vorherigen Feldern. Abbildung 5 zeigt dafür ein Beispiel. Die blauen Pfeile markieren Eingänge und Ausgänge von Bauteilen, die orangen Pfeile Verbindungen von Eingängen und Ausgängen zwischen Bauteilen. Erreichen wir beim Ausführen der Breitensuche ein Zielfeld, können wir durch das Zurückgehen der Pfeile rekonstruieren, welche Bauteile platziert werden müssen, um somit die Pipeline wie in Abbildung 5 dargestellt zu konstruieren.

Die Breitensuche speichert sich nur die erreichten Felder, ohne Teile zu platzieren. Demnach kann es passieren, dass sich Bauteile entlang der Pfeile in Abbildung 5 überschneiden. Es würde keine gültige Pipeline gebaut werden. Diese Fälle müssen wir speziell durch die Strategie in Algorithmus 3 behandeln.

Die Behandlung für Selbst-Überschneidungen beginnt in Zeile 5. Tritt eine Überschneidung auf, so werden iterativ die schon platzierten Objekte, welche noch keine Überschneidungen hatten, entfernt. Nach jedem Entfernen wird die Breitensuche erneut ausgeführt (Zeile 9,) mit dem Ziel, die schon erreichten Bauteile zu erreichen. Dadurch, dass die Anfangsobjekte jetzt schon auf der Karte platziert sind, können in der erneuten Breitensuche Kollisionen mit diesen Objekten verhindert werden. Es gibt weiterhin die Möglichkeit, dass der Versuch, eine überschneidungsfreie Pipeline zu bauen, nicht funktioniert. In diesem Fall geben wir den Misserfolg zurück, und sorgen durch Backtracking für eine neue Ausführung der Produktauswahl.

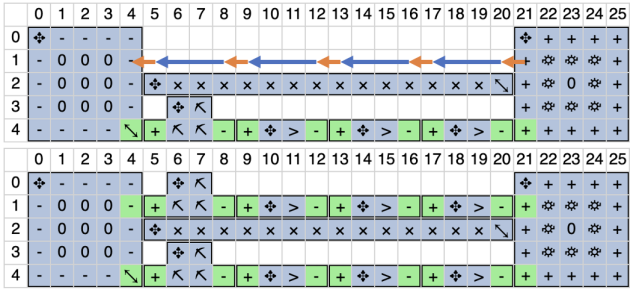

Abbildung 5: Darstellung der Vorgänger-Felder aus denen die Pipeline realisiert werden kann oben und fertig realisierte Pipeline unten

3 EVALUATION

In diesem Kapitel stellen wir verschiedene Eingaben vor, in denen unser Algorithmus eine besonders gute

Algorithm 3 Place Pipeline

Require: $targetfields, deposit, factory$
1: $predecessors \leftarrow bfs\,(deposit, target\_fields)$
2: $objects \leftarrow calculate\_path(predecessors)$
3: **for** $object$ *in* $objects$ **do**
4: $place(object)$
5: **if** $object$ collides **then**
6: **while** objects not empty and connection fails **do**
7: Remove $object$
8: $targetfields \leftarrow$ ingress of last object
9: $predecessors \leftarrow bfs\,(deposits, target\_fields)$
10: $objects \leftarrow calculate\_path(predecessors)$
11: **end while**
12: **end if**
13: **end for**

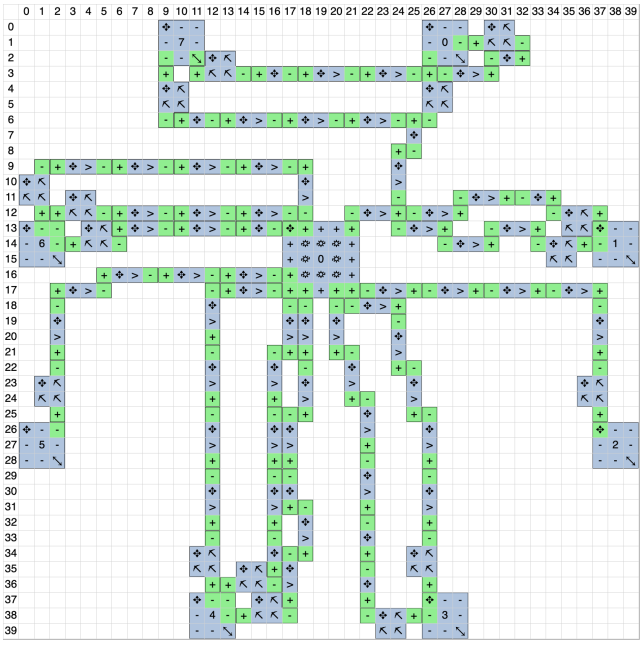

Abbildung 6: Beispiel für Lösungen des Algorithmus bei vielen, ähnlichen Lagerstätten

Lösung liefert, oder in der unser Algorithmus nicht gut funktioniert. Zudem haben wir die Laufzeit des Algorithmus auf den Beispieleingaben und einer selbst erstellten, großen Eingabe gemessen.

3.1 Ähnliche Requirements

Die Heuristik für das Platzieren von Fabriken funktioniert gut, wenn es ähnlich verteile Anforderungen für Produkte gibt. Fabriken werden so platziert, dass der Abstand zu allen Lagerstätten gleich kurz ist. Ein Beispiel ist demonstriert in Abbildung 6. Durch die Platzierung in der Mitte wird die Fabrik von allen benötigten Lagerstätten zeitgleich erreicht. Sie erhält pro Runde eine gleichmäßige Verteilung, mit der sie in optimaler Frequenz produziert.

3.2 Knifflige Engpässe

Abbildung 7 zeigt zwei Beispiele für Lösungen. Im oberen Beispiel konnten keine Verbindungen zu den platzierten Fabriken generiert werden. Im unteren Beispiel wurde eine Verbindung mit der korrekten Länge erzeugt. Die Ergebnisse zeigen, dass die Breitensuche für das Verbinden einer einzelnen Lagerstätte mit einer Fabrik sehr leistungsstark ist, jedoch bei Engpässen, an denen mehrere Verbindungen benötigt werden, schnell die Probleme Selbstüberschneidung und Selbstblockade auftreten. Unsere aktuellen Strategien, diese zu vermeiden, könnten noch verbessert werden. Einen möglichen Ansatz dazu haben wir in Unterabschnitt 6.1 vorgestellt. Zudem sind manche Probleme nicht ohne den Einsatz eines Verbinders lösbar, den wir aktuell nicht benutzen. Durch diesen könnte man an Engpässen Platz sparen, der dann wieder für andere Bauteile benutzbar wird.

3.3 Unterteilte Karten

Bei Karten, die durch Hindernisse in mehrere Bereiche aufgeteilt sind, funktioniert unsere *Divide and Conquer*-Strategie sehr gut. Das Lösen einzelner Zusammenhangskomponenten hintereinander ermöglicht es, auch große Karten effizient zu berechnen.

3.4 Auswertung der Beispiele

Um die Effizienz und Ergebnisse unseres Algorithmus zu testen, haben wir seine Laufzeit auf verschiedenen Eingaben getestet. Die Laufzeit wurde auf einem MacBook Pro[1], und sind Mittelwerte von zehn Messungen. Die Ergebnisse befinden sich in Tabelle 2.

[1] MacBook Pro mit M1 Pro und 16 GB Arbeitsspeicher, MacOS 13.1, clang version 15.0.6 mit **-O3** und **-std=gnu++20**

Tabelle 2: Zeitverbrauch und Ergebnisse für verschiedene Eingaben. Screenshots von den verwendeten Aufgaben befinden sich in Anhang A.

Aufgabe	Dimensionen	Gemessene Zeit	Score	Letzte Produktion in Runde
Task 1	30x20	0,00s	410	48
Task 2	26x5	0,00s	90	20
Task 3	40x40	0,01s	10	12
Task 4	29x23	0,00s	0	1
Circle	40x40	0,03s	40	21
DivideAndConquer	100x100	0,07s	2150	100

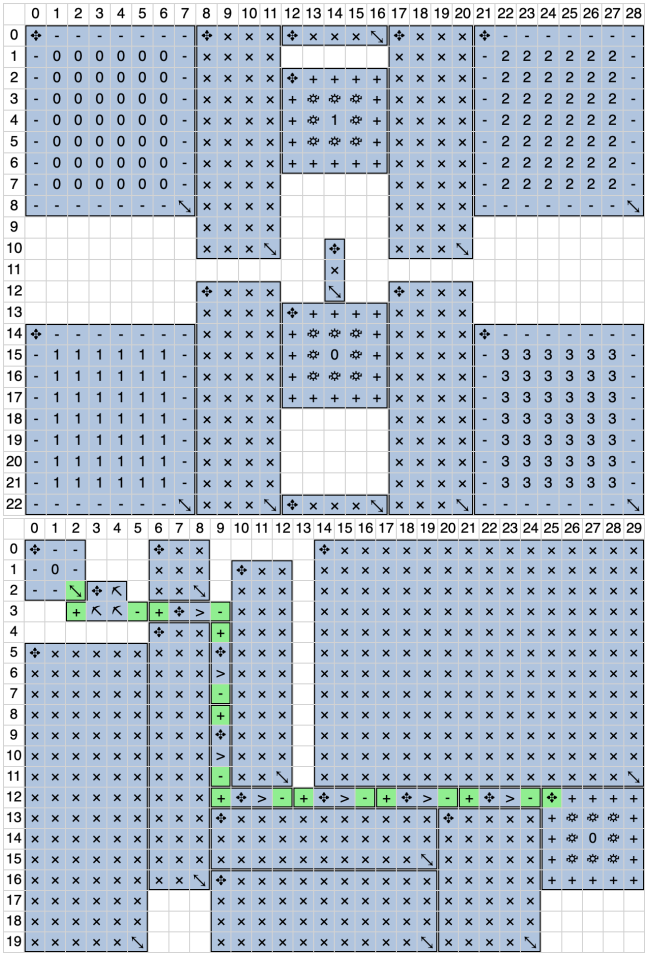

Abbildung 7: Beispiele für Lösungen des Algorithmus an Engpässen

Der Algorithmus produziert für alle Aufgaben eine valide Lösung, auch wenn diese bei *Task 4* keine Pipelines, sondern nur zwei Fabriken enthält.

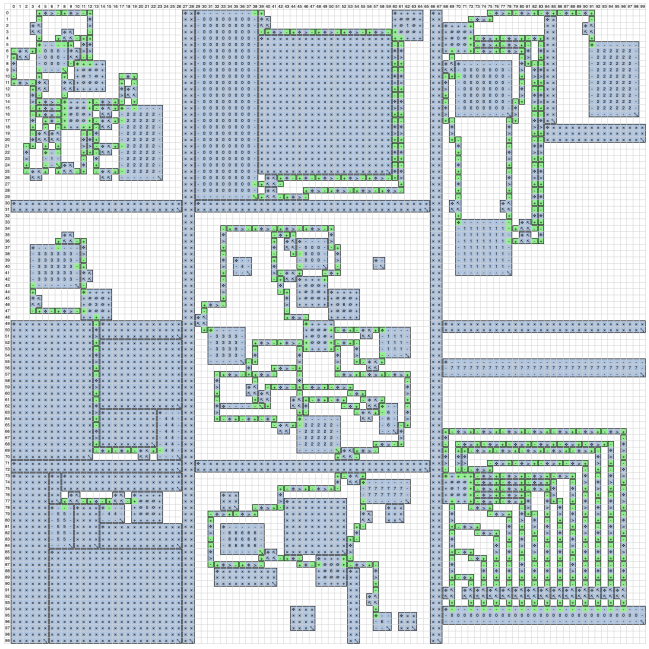

Abbildung 8: Beispiele für Lösungen des Algorithmus bei unterteilten Karten. Eine große Abbildung findet sich in Anhang A

An den gemessenen Zeiten ist ersichtlich, dass unser Algorithmus sehr schnell terminiert. Auf keiner der Eingaben benötigt er länger als 0.1s. Dies zeigt, dass es uns gelungen ist, eine sehr effiziente Implementierung unserer Heuristiken zu erstellen, die gleichzeitig energiesparend ist. Durch die schnelle Laufzeit besteht viel Potenzial, noch weitere Optimierungen zu integrieren, bis das Zeitlimit erreicht ist.

4 ARCHITEKTUR

Unser Programm lässt sich grob in passive Datenstrukturen und aktive Akteure unterteilen. Die Datenstrukturen werden von den Akteuren für die Lösung ihrer Probleme verwendet. Manche der Strukturen sind suggerierend *structs*, andere mit wenig Verhalten *classes*, wiederum andere ein Konzept vereint in einer gemeinsamen Datei. Am Ende thematisieren wir die enthaltenen Tests.

4.1 Datenstrukturen

Die Datenstrukturen lassen sich zu verschiedenen Paketen zusammenfassen.

IO. Zu Beginn und Ende unserer Ausführung müssen wir JSON parsen und schreiben. Wir verwenden dazu die JSON-Bibliothek von Niels Lohmann[12], die als Header-Datei eingebunden ist. *Input* ist unser Interface für die geparsten Daten. Ergebnisse werden in einem *struct* gespeichert, welches die Objekte zu JSON serialisiert.

Geometry. Enthält grundlegende geometrische Objekte wie zweidimensionale Vektoren, Rotationen, und Rechtecke. Durch die Begrenzung des Feldes auf 100 Felder pro Dimension, genügt uns ein Byte pro x bzw. y Koordinate. *Coordinate* ist im Sinne konsistenter Handhabung des Typen unsere Wrapper-Struktur. *Vec2* verwenden zwei *Coordinates* und bieten grundlegende mathematische Operationen an. Minen und Fabriken sind Rechtecke, für die *Rectangle* Hilfsmethoden wie Iteration belegter Zellen, und Randerkennung anbietet.

Objects. Die Produkte, Landschafts- und Bauteile aus der Spezifikation finden sich analog in *Product*, *LandscapeObject*, und *PlaceableObject* wieder. *PlaceableObjects* haben Methoden zur Erstellung mit einer gegebenen Rotation, Abruf der besetzten Zellen, und zum Eingang (*upstream_egress_cells*) oder Ausgang (*downstream_ingress_cells*) angrenzender Zellen.

Fields. In Abschnitt 2 erwähnen wir des Öfteren verschiedene Repräsentationen des Feldes, *OccupancyMap*, *DistanceMap*, *PlacementMap*, et cetera. Sie sind konkrete Template-Instanziierungen einer *Field* Klasse, die ein zweidimensionales Feld repräsentiert. Um Überprüfungen für Grenzeinhaltungen in unseren Algorithmen zu umgehen, fängt *Field* intern ungültige Koordinaten ab. Die Überprüfung geschieht derzeitig manuell, doch könnte ein Padding des Feldes den Zugriff auf Felder durch weniger Operationen beschleunigen.

Miscellany. Eine Sammlung von diversen Helfern, wie zum Beispiel Debug-Asserts.

4.2 Akteure

Die restlichen Klassen spiegeln größtenteils die besprochenen Algorithmen wider, welche wir bereits erläutert haben. Wir fokussieren uns kurz auf die nebensächlichen Akteure, die wir zur Ausführung benötigen.

Field State. Platzierungen erfolgen nicht nativ auf der *OccupancyMap*, sondern geschehen über *FieldState*. Die Klasse garantiert valide Platzierungen und verwaltet das Wissen aller platzierten Objekte.

Solver. Solver setzen eine Strategie zur Lösung um. Sie kümmern sich um das Setup und die Verbindung der einzelnen Schritte. Für die iterative Wiederholung an Schritten sind sie verantwortlich. Zukünftige Lösungsstrategien würden neue Solver hinzufügen und in *Worker* austauschen.

Nebenläufigkeit. Um den zeitlichen Rahmen der Eingabe einzuhalten, benutzen wir mehrere Threads. Wir wollen Lösungen mit höherer Punktzahl mit der alten ersetzen, und am Ende der Zeit, oder der Berechnungen, die beste Lösung ausgeben. Ein Thread kümmert sich um die Berechnung einer Lösung, der andere wacht kurz vor Ende der Zeit auf, und beendet notfalls die Berechnung. Die geteilte Ressource der besten Lösung ist ein geteilter Mutex.

4.3 Tests

Neben der Entwicklung einer Komponente fügten wir fortlaufend Tests zu. Neben einer CMake Konfiguration für unsere Lösung gibt es im Unterordner *tests* eine zweite, die mit GoogleTest [10] eine ausführbare Test-Suite baut. Für den Inhalt der Tests gab es keine strikten Regeln, sondern eher grobe Richtlinien.

- Behobene Fehler in der Anwendung sollten stets von einem Regressionstest gefolgt werden.
- Die Funktionalität wird mit den gegebenen *Tasks* der GI getestet
- Für das Problem interessante Felder werden *example_tasks.hpp* hinzugefügt, eine Sammlung an Eingaben für schnelles Setup in Tests

- Tests für verschiedene *Fields* stellen das Feld veranschaulicht dar

Wir setzten keinen Fokus auf eine modulare Struktur der Tests, sie dürfen sich in ihrem Setup derzeit duplizieren. Neben der niedrigeren Hemmschwelle, bleiben die Beispiel-Benutzungen zur Erklärung von Methoden lokal in einer Datei. Die Tests sind eine Mischung aus Unit- und Integrationstests. *solver_test.cpp* enthält Systemtests.

5 SOFTWAREQUALITÄT

Über die Laufzeit des Projekts hinweg haben wir besonderen Wert auf die Einhaltung von Softwarepraktiken gelegt. Wir vereinten Ansätze des Agile Developments, Lean Software Development, und diverse Industriestandards [6].

5.1 In Sachen Code

Das Projekt wurde mit gezielter Intention mit C++20 umgesetzt. Für die gegebene Eingabezeit wollen wir jede Sekunde effizient nutzen. Versteckten Overhead in Form von Garbage Collection, übermäßiges Kopieren oder Virtualisierung, wie zum Beispiel in Java und Python üblich [8, 13], vermeiden wir. Durch die Nähe zur Hardware definieren wir in C++ nur das Nötigste, was wir für die Aufgabe brauchen. Mithilfe von wichtigen Keywords wie *constexpr* und dem berühmten gcc *-O3* Flag, optimieren wir zur Kompilierungszeit. Die einzige Abhängigkeit unseres Projekts ist die Einbindung einer JSON-Bibliothek von Niels Lohmann, welche wir als Header Datei eingebunden haben [12].

Neben der generellen Einhaltung der gut dokumentierten Core Guidelines [16] haben wir Googles Style umgesetzt [9]. *Clang-Tidy*, *CPPLINT* als Linter und *Clang-Format* als Formatter sind essenzielle Bestandteile der Konsistenzerhaltung. Deswegen sind sie nicht nur Teil des Projekt-Setups, sondern auch dazugehörige GitHub Aktionen in unserer CI (siehe Unterabschnitt 5.2).

Sprachenunabhängig ist in unserem Code auffällig, dass wir sparsam mit Kommentaren und Dokumentation umgehen. Dies resultiert aus der Motivation heraus, stets selbsterklärenden Code zu erzwingen, der simple, erkennbare Intentionen kommuniziert. Dokumentation hat nämlich oft den Nachteil, dass sie nicht mitgewartet wird bzw. zeitintensiv werden kann. Änderungen im Code lassen den einen oder anderen Kommentar veraltet zwischen den Zeilen herumstehen, welcher dann schlichtweg falsche Informationen enthält.

Eine bessere Möglichkeit, Verhalten von konkreten Methoden zu erläutern, sind Tests. Wir haben Unit- und Integrationstests, welche den Gebrauch und das gewünschte Ergebnis in deskriptiven Situtationen beschreiben. Umgesetzt wurden diese mit dem Google Testing Framework *GTest* [10]. Besonders die Tests zu den verschiedenen Felder-Repräsentationen visualisieren ihre Idee leicht verständlich. Zum Verständnis auf abstraktem Level dient weiterhin diese Ausarbeitung.

5.2 Projektmanagement

Unser Projekt ist auf GitHub gehosted. Git als Versionsverwaltungssystem ist weitbekannter Standard - GitHub ist mit seinen Issues, Actions und den relativ neuen Projektboards eine zentrierte Plattform mit allem, was wir benötigen.

Das Projekt verfolgt den offiziellen Feature Branch Workflow [2] und Continuous Integration, um stets funktionierende Prototypen auf *main* zu garantieren. Merge Requests wurden nur unter folgenden Bedingungen in *main* akzeptiert:

- Mindestens ein Team-Mitglied hat es überprüft und akzeptiert.
- Die Pipeline, bestehend aus GitHub Actions Tests, Linter, und Formatter, ist erfolgreich durchgelaufen.
- Es wurden Tests geschrieben.
- Offene Probleme und Aufgaben wurden zu Issues umgewandelt.

Im Sinne des Collective Code Ownerships sind wir nicht zu streng damit, ob die Merge Requests thematisch eingegrenzt sind. Falls Verbesserungen gefunden werden, sollte die Überwindung zur Umsetzung gering sein. Das Makefile bietet verschiedene Konfigurationen an, in der das Programm gebaut werden kann. Lokal programmieren wir in der *Debug*-Version, welche zusätzliche Assertions und explizite Fehlermeldungen beinhaltet. Die Pipeline ist mit *Release* konfiguriert und garantiert somit, dass unsere Abgabeversion stets funktioniert.

Weitreichende Aufgaben (bzw. EPICs) sind als Tickets auf unserem Board beschrieben, konkrete Aufgaben als Issues. In wöchentlichen Treffen wurden Algorithmen gemeinsam konzipiert und im Pair Programming entwickelt. Falls allein weiterentwickelt wurde, gab es

stets eine kurze Zusammenfassung als Textnachricht, oder einen kurzen Anruf. So haben wir eine gleichmäßige Wissensverteilung sichergestellt und haben das Zwischenmenschliche nicht zu kurz kommen lassen, welches in der Softwareentwicklung für den Spaß an der Sache natürlich nicht fehlen darf.

6 DISKUSSION

Unser derzeitiger Stand hat noch Raum für Optimierungen und ist auch von seinem Kerngedanken hinsichtlich Effizienz kritisch zu betrachten. In diesem Kapitel thematisieren wir kurz, welche möglichen Erweiterungen des Projekts umgesetzt werden könnten.

6.1 Wigglen der Pipelines

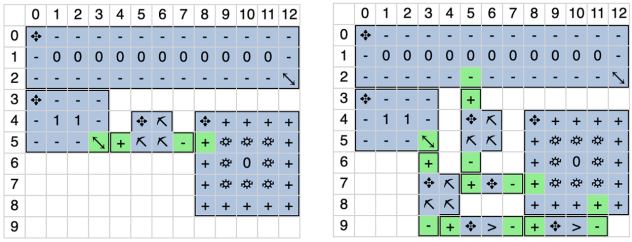

Abbildung 9: Blockierende Verbindung von Depot 1 mit der Fabrik links und mögliche Lösung rechts.

Durch die iterative Erstellung einer Lösung kann es vorkommen, dass vorhandene, lokal optimale Verbindungen, eine global bessere Lösung blockieren. Ein mögliches Beispiel dafür befinde sich in Abbildung 9. Auf dem linken Spielfeld sieht man, dass beim Verbinden von Lagerstätte 1 und der Fabrik auf dem kürzesten Weg nur eine Mine benötigt wird. Diese Mine einzusetzen verhindert allerdings, dass die Lagerstätte 0 mit der Fabrik verbunden werden kann. Eine bessere Lösung ist auf der rechten Seite zu sehen. Hier gehen beide Minen nach unten weg und beide Lagerstätten sind verbunden.

Wir haben über verschiedene Möglichkeiten nachgedacht, Situationen wie diese zu verhindern. Grob einteilen kann man unsere Ideen in lokale versus globale Optimierungen. Eine globale Optimierung versucht, für geplante Pipelines vorher Bereiche auf dem Spielfeld zu reservieren, die anschließend von anderen Pipelines gemieden werden. Dadurch, dass jede Pipeline ihre eigenen Felder zugeordnet bekommt, treten keine Konflikte auf. Diese Optimierung würde jedoch gute Heuristiken für den Platzbedarf von Pipelines voraussetzen, die durch die verschiedenen Einschränkungen von „Profit!“ schwierig sind. So sind zum Beispiel Förderbänder nur in der Länge von drei und vier vorhanden, was es teilweise notwendig macht, benötigte Längen mit einer Hilfskonstruktion zu erreichen, die viel Platz benötigen können.

Für die Weiterentwicklung schlagen wir deswegen eine lokale Optimierung vor, welche jede Pipeline zuerst nach der kürzesten Verbindung realisiert, und bei auftretenden Konflikten versucht, diese lokal zu lösen. Zentral ist dabei die Möglichkeit, dass wir eine Pipeline als eine konzeptionelle Verbindung von Lagerstätte und Fabrik modelliert haben, die aber nicht direkt aus platzierten Objekten besteht, sondern diese Objekte erst erhält, wenn sie realisiert wird. Dadurch können wir platzierte Objekte von Pipelines flexibel vom Spielfeld entfernen, wenn wir vermuten, dass diese uns stark einschränken.

Eine interne Datenstruktur könnte für jede Zelle des Spielfeldes abspeichern, welche Pipeline sich dort befindet. Bei einer Kollision mit einer anderen Pipeline ermöglicht das effizient nachzuschlagen, von welcher Pipeline man gerade blockiert wird. Im Fall von 9 würde somit erkannt werden, dass wir beim Verbinden von Lagerstätte 0 mit der Fabrik von einer Pipeline blockiert werden, die von Lagerstätte 1 zur Fabrik führt. Lösen können wir den Konflikt durch das Entfernen von den Bauteilen der im Konflikt stehenden Pipeline. Anschließend realisiert sich die Verbindung von Lagerstätte 0 zur Fabrik, und erneutem Realisieren von der vorher entfernten Pipeline. Das könnte zum Beispiel zur Lösung auf der rechten Seite in 9 führen. Nicht alle Probleme lassen sich mit dieser Strategie lösen, weshalb unser Algorithmus noch andere Strategien nutzt, die in dieser Situation helfen könnten. Zum Beispiel das Platzieren von der Fabrik an verschiedenen Stellen, die eventuell eine bessere Konnektivität ermöglichen, oder den Fokus auf die Produktion eines anderen Produktes führt, das realisiert werden kann.

6.2 Implementierung in den Header Dateien

Die meisten Funktionalitäten sind direkt in den Header Dateien implementiert, welches für den größeren Projektrahmen schlechter Stil ist. Der Grund für die direkte

Einbindung ist die Veränderung der Methoden über den Projektzeitraum hinweg. Umbenennungen mussten im Hinterkopf behalten werden, oder wurden bei Ausführung mit Visual Studio Code nicht konsistent durchgesetzt. Dies ist also eine bewusste technische Schuld unsererseits, die sich mit detaillierterer Planung hätte vermeiden können.

6.3 Fehlende Verbinder

Verbinder sind ein großartiges Bauteil, um Pipelines in ihrer Ausbreitung zu verringern. Neben dem in Unterabschnitt 6.4 erwähnten Vorteil gegen Engpässe könnten sie dem Wigglen von Pipelines mehr Freiraum verschaffen. Sie könnten eingebunden werden, indem sich Pipelines die Bauteile ab einem Verbinder teilen. Für die Einbindung hat uns jedoch die Zeit gefehlt.

6.4 Engpässe

Die Platzierungen der Pipelines ignorieren den Vorteil von Verbindern an Engpässen. Statt den kürzesten Weg zu wählen, sollten enge Durchgänge erkannt werden und Verbinder prioritisieren, um die Latenz insgesamt zu verringern. Einen Ein-Zell breiten Durchgang mittels Nachbarschaftsinformationen zu finden ist zwar unkompliziert, aber ein Zwei-Zell breiter, Drei-Zell breiter, etc. Durchgang führen die selben Probleme mit sich. Eine allgemeine Lösung des Problems könnte Engpässe mithilfe einer Formalisierung als Max Flow Graph Problem heuristisch vermeiden. Max Flow = Min Cut [4] gäbe uns die Anzahl der verschiedenen Wege von Lagerstätte A nach Lagerstätte B mit Identifizierung von Engpässen unabhängig von ihrer Breite.

6.5 Energieverbrauch

In Tabelle 2 ist zu sehen, dass wir nur einen Bruchteil der verfügbaren Bearbeitungszeit für das Herausfinden der Lösung benutzen. Man könnte unseren Algorithmus erweitern, indem er deutlich mehr Platzierungen testet. Dadurch könnten insbesondere an schwierigen Engpässen, an denen unsere Heuristiken nicht gut funktionieren, trotzdem Lösungen gefunden werden. Der generelle Ansatz des bloßen Ausprobierens verschiedener Lösungen, bis die Zeit endet, sollte jedoch hinsichtlich Energy-Aware Computing [17] kritisch betrachtet werden. In dem zeitlich begrenzten Rahmen der Wettbewerbs-Aufgaben sind die Auswirkungen zwar nicht gravierend, trotzdem verschwendet die Ausführung Energie. Das potenziell endlos lange Berechnen in der Hoffnung auf höhere Punktzahlen könnte dazu führen, Rechengeräte nebenbei, wie zum Beispiel in der Nacht, laufen zu lassen. Dabei ist mit vergangener Zeit umso ungewisser, wann die nächste Optimierung gefunden wird. Außerhalb des Wettbewerbs sollten die Lösungen ein maximales Zeitlimit festlegen, oder nach einer bestimmten Anzahl Runden ohne Verbesserung automatisch abbrechen. Derweil sollte das Zeitlimit auf ein paar Minuten beschränkt werden. In realen Anwendungen um Konstruktionen herum, kann unsere Software eine Orientierung geben, die gegebenenfalls manuell optimiert werden kann.

7 FAZIT

Die Aufgabenstellung konfrontierte uns mit mehreren NP-vollständigen Problemen. Wir entwickelten eine datenbasierte Lösung, die mehrere Greedy-Ansätze iterativ miteinander verbindet. In Szenarien mit gleichmäßigen Ressourcen-Anforderungen, die keine bis wenig Engpässe hat, findet sie effizient gute Lösungen. Durch die Aufteilung des Problemraums in Zusammenhangskomponenten werden auch große Felder effizient in weniger als einer Sekunde gelöst. Falls die Szenarien Präferenzen von Ressourcen beschreiben, oder nur Verbinder eine Pipeline realisieren können, fällt die Lösung zurück. Während der Umsetzung haben wir Techniken der agilen Entwicklung genutzt, um die Softwarequalität sicherzustellen. Mit knapp 100 Tests garantieren wir die Funktionalität.

LITERATUR

[1] Yalçin Akçay, Haijun Li, and Susan H. Xu. 2007. Greedy algorithm for the general multidimensional knapsack problem. *Ann. Oper. Res.* 150, 1 (2007), 17–29. https://doi.org/10.1007/s10479-006-0150-4

[2] Atlassian Bitbucket. 2023. *Git Feature Branch Workflow.* Retrieved 13th January, 2023 from https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

[3] Vincent Boyer, Didier El Baz, and Moussa Elkihel. 2010. Solution of multidimensional knapsack problems via cooperation of dynamic programming and branch and bound. *European Journal of Industrial Engineering* 4, 4 (2010), 434–449.

[4] Diverse. 2023. *Max-flow min-cut theorem.* Retrieved 14th January, 2023 from https://en.wikipedia.org/wiki/Max-flow_min-cut_theorem

[5] Diverse. 2023. *Union-Find-Struktur.* Retrieved January 13, 2023 from https://de.wikipedia.org/wiki/Union-Find-Struktur

[6] Kent Beck et al. 2001. *Manifesto for Agile Software Development.* Retrieved 13th January, 2023 from https://agilemanifesto.org/
[7] Hussein Etawil, Shawki Areibi, and A. Wannelli. 1999. Attractor-repeller approach for global placement. 20–24. https://doi.org/10.1109/ICCAD.1999.810613
[8] Python Software Foundation. 2023. *General Python FAQ.* Retrieved 13th January, 2023 from https://docs.python.org/3/faq/general.html#what-is-python
[9] Google. 2023. *Google C++ Style Guide.* Retrieved 13th January, 2023 from https://google.github.io/styleguide/cppguide.html
[10] Google. 2023. *GoogleTest - Google Testing and Mocking Framework.* Retrieved 13th January, 2023 from https://github.com/google/googletest
[11] H. Kellerer, U. Pferschy, and D. Pisinger. 2004. *Knapsack Problems.* Springer. https://books.google.de/books?id=u5DB7gck08YC
[12] Niels Lohmann. 2022. *JSON for Modern C++.* Retrieved 13th January, 2023 from https://github.com/nlohmann/json
[13] Oracle. 2023. *Java Garbage Collection Basics.* Retrieved 13th January, 2023 from https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
[14] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms, 4th Edition.* Addison-Wesley.
[15] Shrashti Singhal. 2020. *All About Heatmaps.* Retrieved 14th January, 2023 from https://towardsdatascience.com/all-about-heatmaps-bb7d97f099d7
[16] Bjarne Stroustrup. 2022. *C++ Core Guidelines.* Retrieved 13th January, 2023 from https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c-core-guidelines
[17] Anne E. Trefethen and Jeyarajan Thiyagalingam. 2013. Energy-aware software: Challenges, opportunities and strategies. *Journal of Computational Science* 4, 6 (2013), 444–449. https://doi.org/10.1016/j.jocs.2013.01.005 Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing.
[18] Thibaut Vidal, Teodor Gabriel Crainic, Michel Gendreau, and Christian Prins. 2013. Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. *Eur. J. Oper. Res.* 231, 1 (2013), 1–21. https://doi.org/10.1016/j.ejor.2013.02.053

A BEISPIELEINGABEN

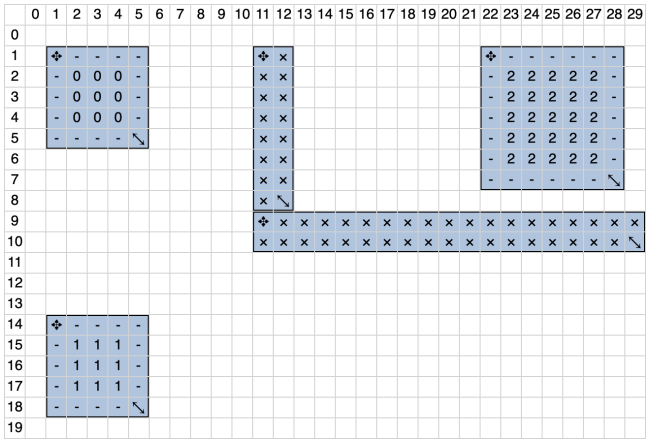

Abbildung 10: Beispieleingabe *Task 1*

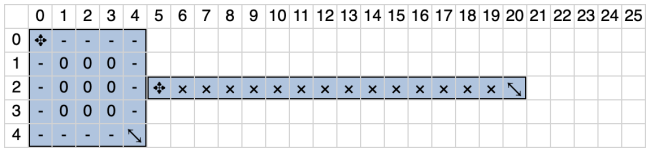

Abbildung 11: Beispieleingabe *Task 2*

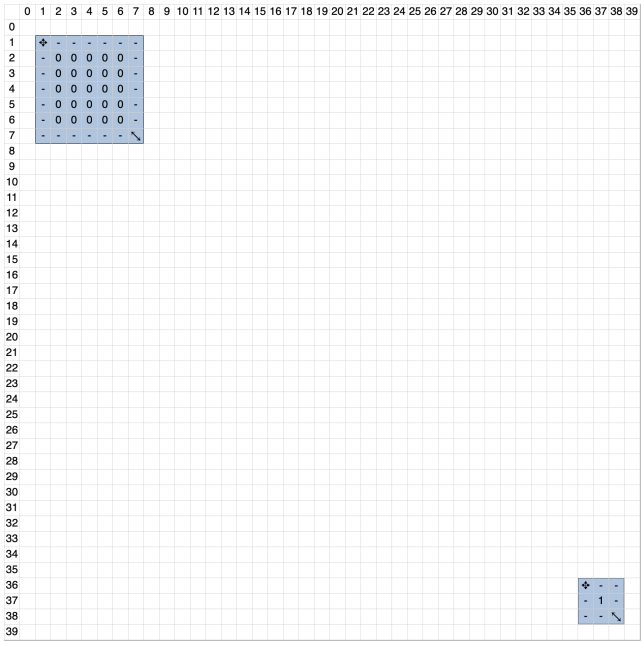

Abbildung 12: Beispieleingabe *Task 3*

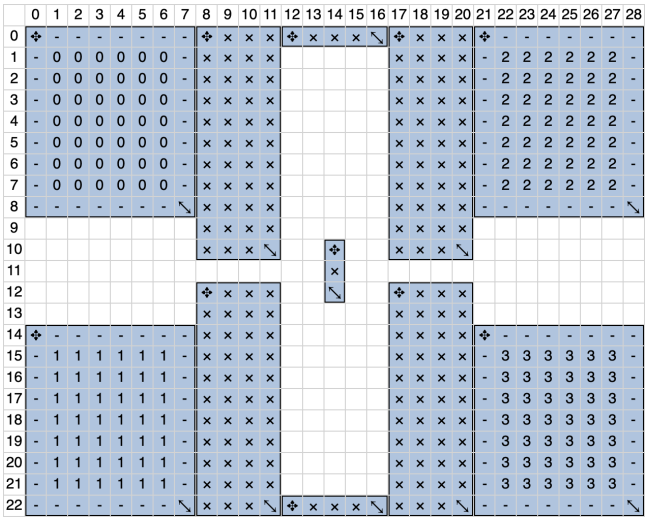

Abbildung 13: Beispieleingabe *Task 4*

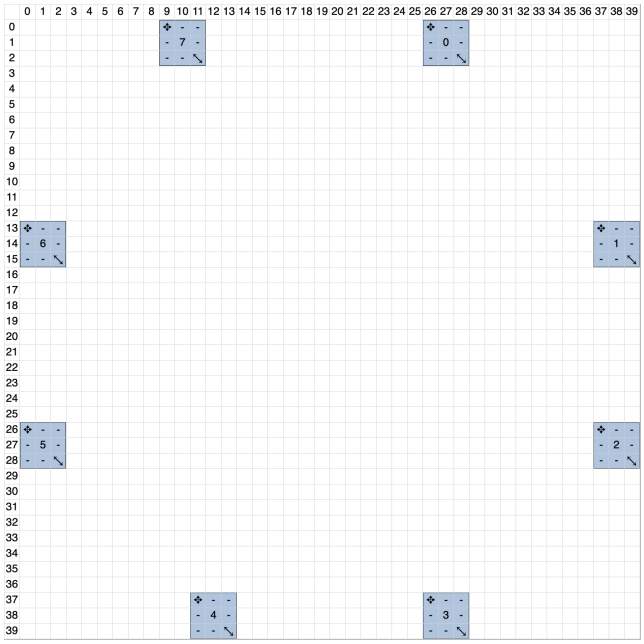

Abbildung 14: Beispieleingabe *Circle*

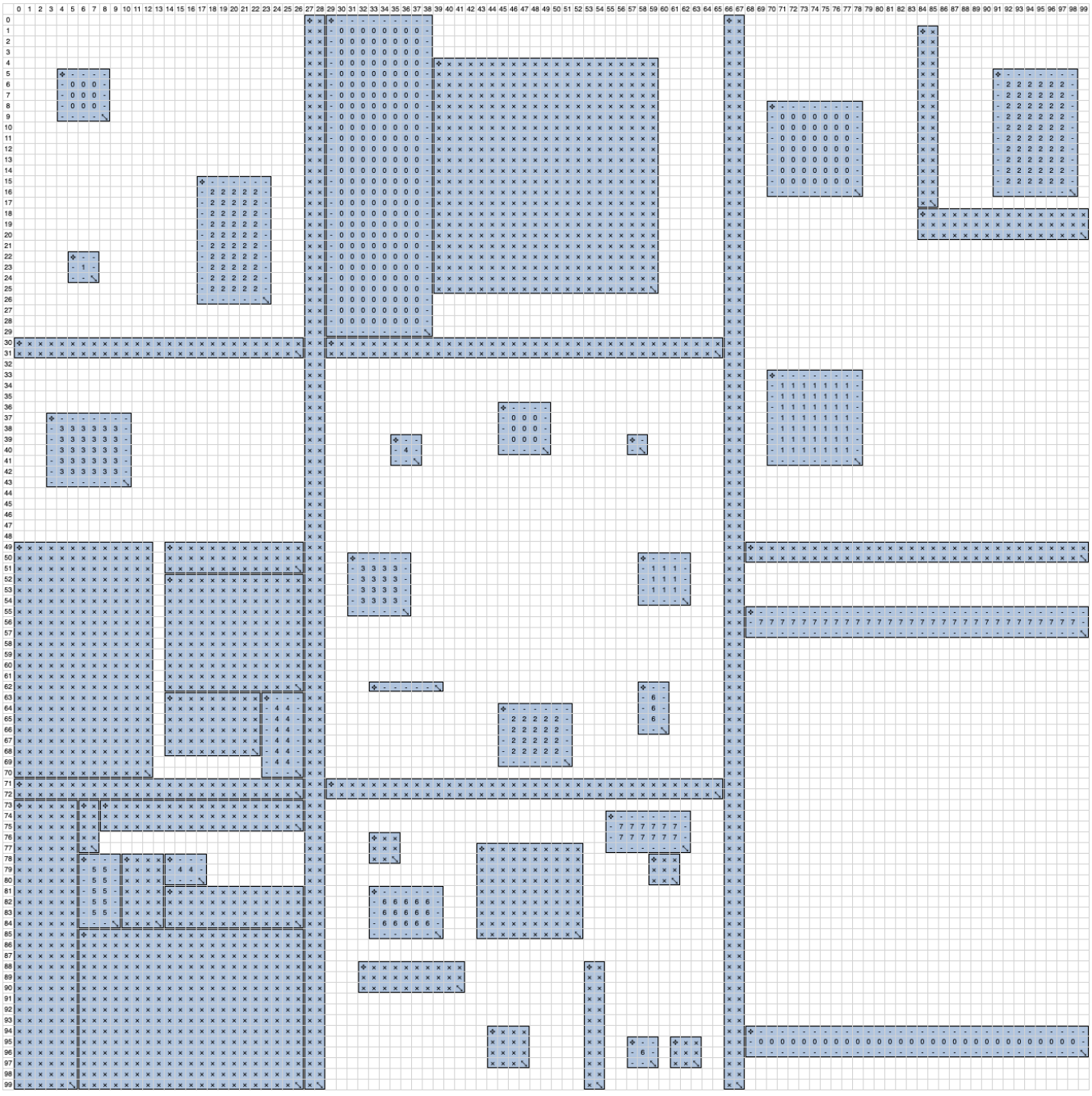

Abbildung 15: Beispieleingabe *DivideAndConquer*

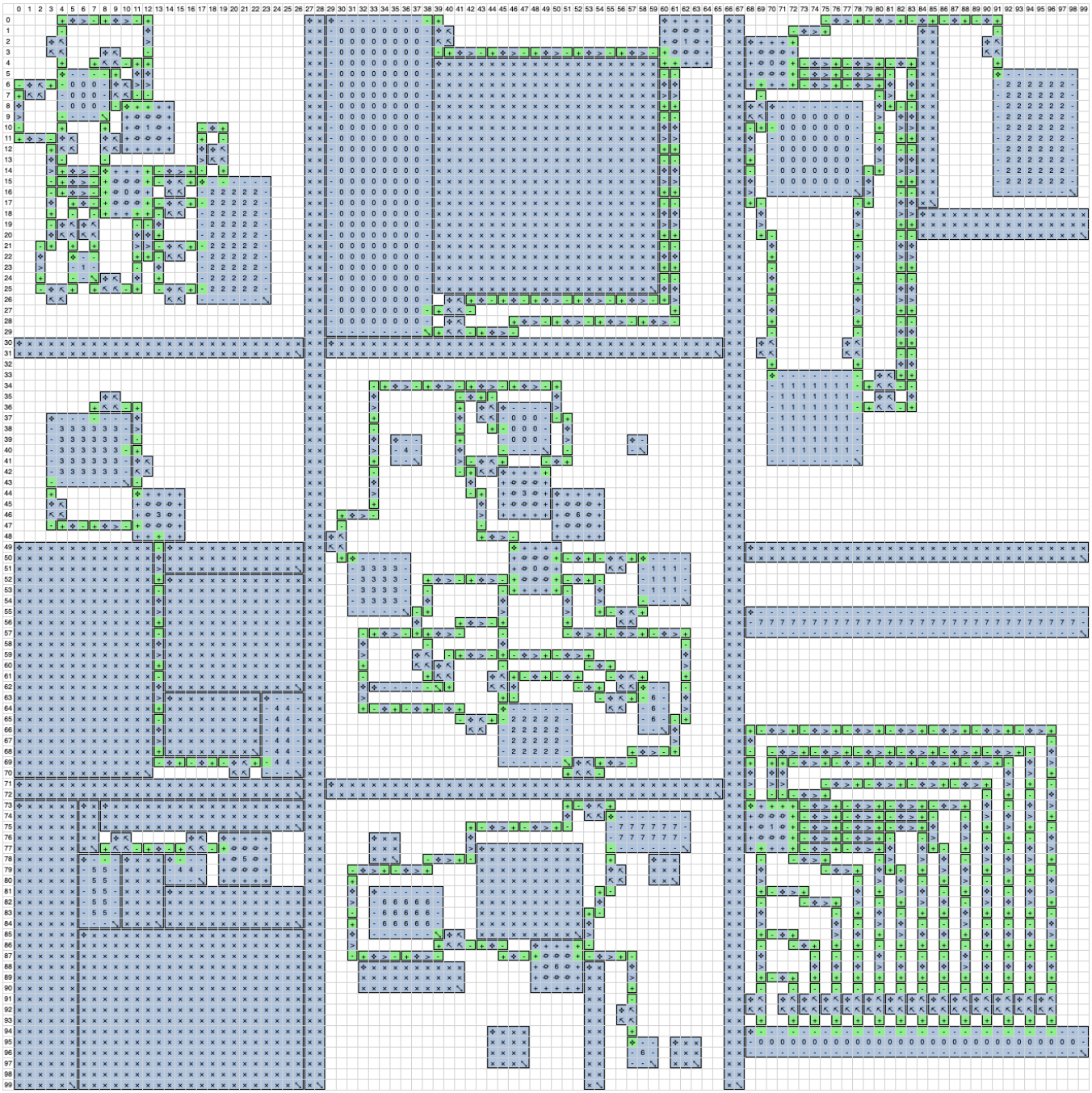

Abbildung 16: Lösung von *DivideAndConquer*