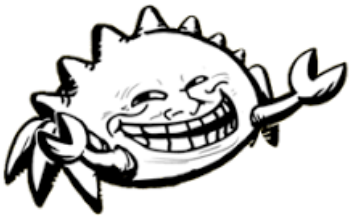
InformatiCup 2021 - Profit

Theoretische Ausarbeitung

RustEvangelismStrikeforce

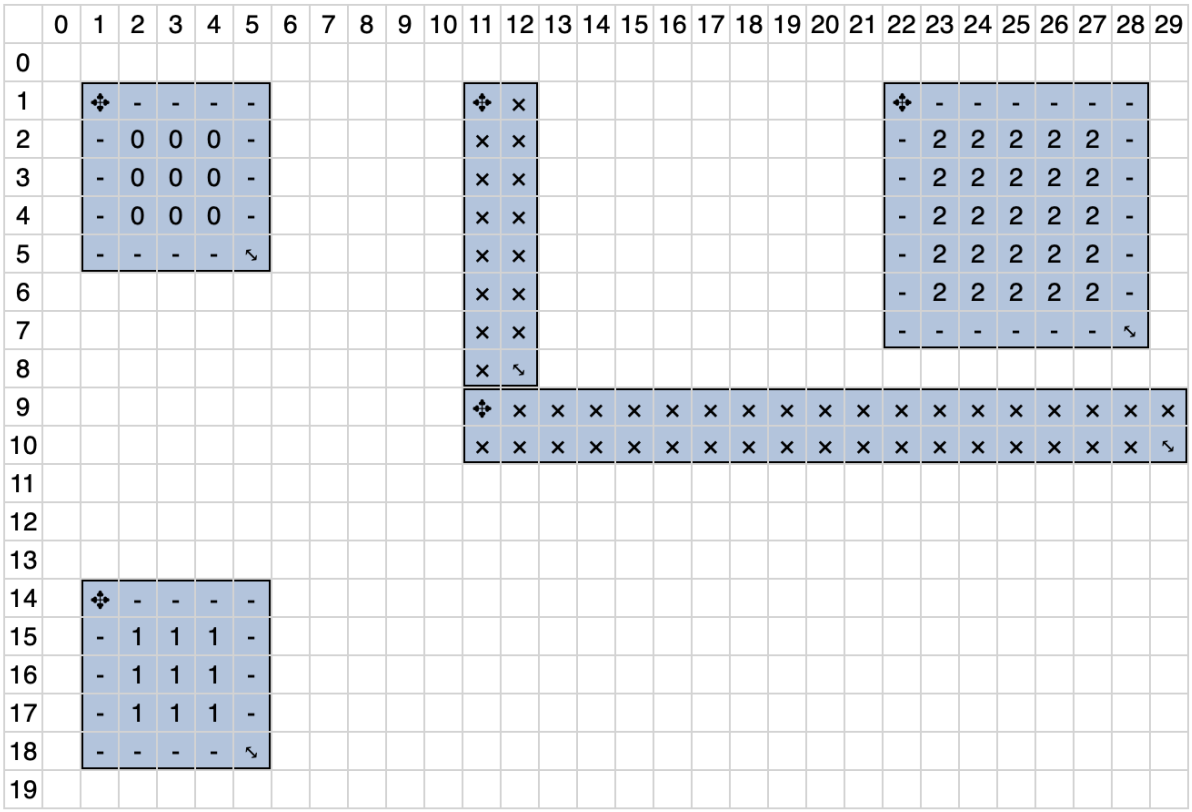


Tobias Schmitz
tobias.schmitz@student.uni-siegen.de
Maik Romancewicz
maik.romancewicz@student.uni-siegen.de

Inhaltsverzeichnis

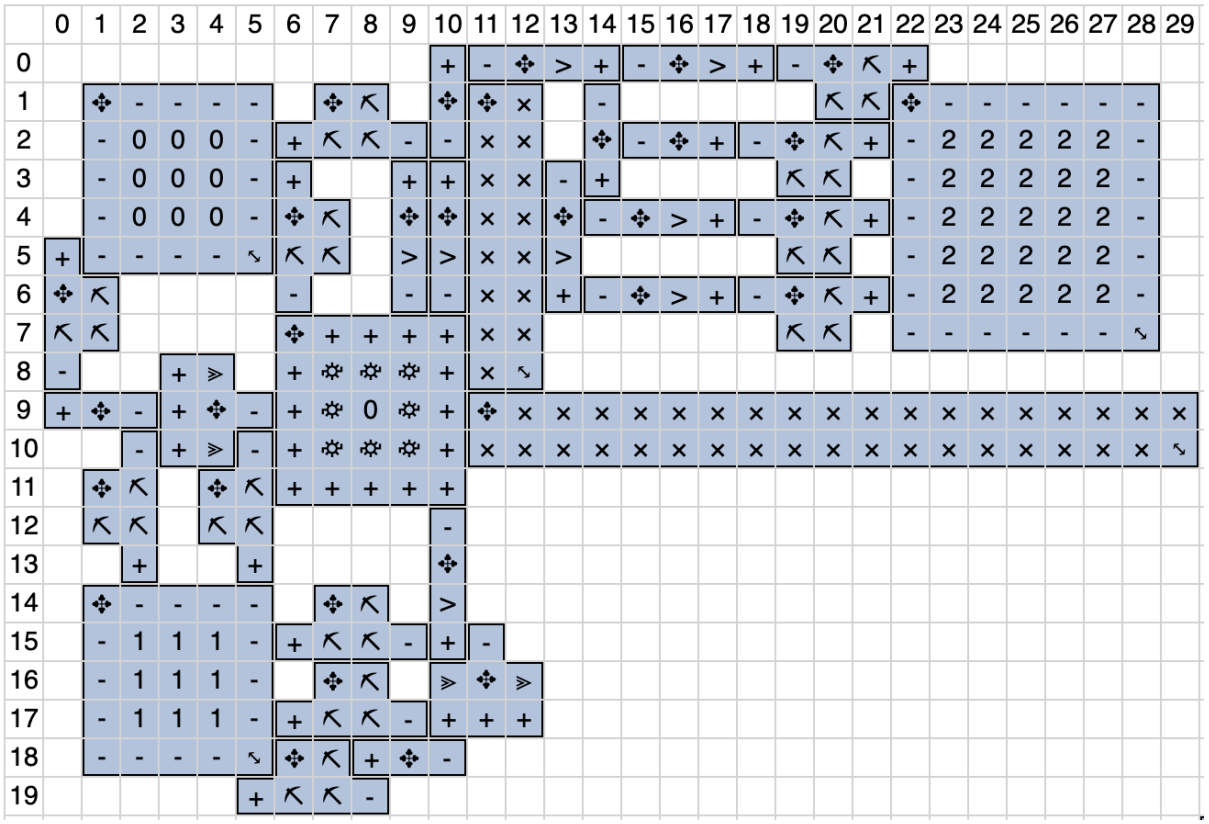
1 Einleitung

Beim InformatiCup 2023 war die Aufgabe, im Rahmen einer rundenbasierten Simulation einen Produktionsprozess zu optimieren. Als Eingabe für das zu entwickelnde Programm erhält man ein 2-dimensionales Feld mit bereits platzierten Deposits und Obstacles, eine Liste von Produkten deren Produktion gewisse Ressourcen (aus den Deposits) benötigt und deren Produktion eine gewisse Punktzahl erzielt als auch ein Rundenlimit in dem die Simulation abläuft. Auf dem besagten Feld gilt es unterschiedliche Bauteile zu platzieren welche Ressourcen abbauen, diese transportieren und schließlich Produkte herstellen um Punkte zu erzielen. Ziel war es ein Programm zu entwickeln welches innerhalb einer vorgegebenen Zeit eine Liste an zu platzierenden Bauteilen generiert welche möglichst viele Punkte erzielt, in möglichst wenigen Runden.



Eine ausführliche Beschreibung der Aufgabe ist auf der Website bzw. im Git Repository des Informaticups 2023 zu finden.

Wir werden in der folgenden Ausarbeitung anhand des obigen Beispiels erklären wie unsere Lösung funktioniert, welche Gedanken wir uns dazu gemacht haben und wie unser Programm letztendlich zu folgender Lösung kommt:



2 Theoretischer Ansatz

Im Laufe des Wettbewerbs hatten wir natürlich zahlreiche unterschiedliche Ideen wie man dieses Problem angehen kann und haben auch einige davon länger verfolgt. Letzten Endes sind wir zum Entschluss gekommen, dass wir die Aufgabenstellung in unterschiedliche Teilaufgaben unterteilen und versuchen diese einzeln anzugehen bevor wir die Erkenntisse die wir aus den Teilaufgaben gewonnen haben zu einer Gesamtlösung zusammenführen. Konkret bedeutet das, dass wir die Aufgabe in folgende Punkte unterteilt haben:

1. Produktauswahl treffen
2. Factory Platzierung
3. Minen Platzierung
4. Minen und Fabriken verbinden

Die Idee hinter dieser Aufteilung war es, dass wir unsere Lösungen soweit Laufzeittechnisch optimieren, dass wir uns nicht auf einzelne Heuristiken verlassen müssen, sondern wir möglichst viele unterschiedliche Konfigurationen der Lösungen von den Teilaufgaben ausprobieren können. Dazu bringen wir die Lösungen der einzelnen Teilaufgaben in eine Rangliste und testen diese dann nacheinander kombiniert mit den Lösungen der anderen Teilaufgaben. Dazu brauchten wir also eine Simulationsumgebung mit der wir gefundene Lösungen testen können, Lösungsverfahren für die Einzelprobleme und eine sinnvolle Methode diese Lösungen miteinander zu kombinieren.

3 Implementierungen

3.1 Simulationsumgebung

Das erste was wir zu Beginn des Wettbewerbs implementiert haben, ist eine Simulationsumgebung, die anhand einer Liste an Objekten die Simulation durchführt und uns die Punktzahl und die dafür benötigten Runden zurückgibt. (Angelehnt also an die letztendliche Abgabe unserer Lösung)

Damit haben wir uns zum Einen von der Simulation die uns für den Wettbewerb zur Verfügung gestellt wurde unabhängig gemacht und haben eine Simulationsumgebung die um ein Vielfaches schneller ist als die Web-basierte Implementierung auf der profit.phinau.de Website. Ursprünglich ist die Simulationsumgebung entstanden um als Supervision für Machine-Learning basierte Methoden (z.B als Reinforcement Learning environment) zu dienen. Bei diesen wäre es natürlich sehr wichtig gewesen, dass wir ggf. tausende Inputs schnell auswerten können, um unser Model zu trainieren. Allerdings haben wir unseren Ansatz im Laufe des Wettbewerbs geändert, trotzdem war die Simulationsumgebung ein integraler Bestandteil unserer Methodik. Mithilfe unserer Umgebung können wir einen gegebenen Input im Bruchteil einer Millisekunde auswerten und haben ein Feedback darüber wie gut die platzierten Bauteile die Aufgabe lösen.

3.2 Regionen

Beim Ausführen unserer Lösung wird als Erstes die Umgebung in unterschiedliche Regionen unterteilt für den Fall, dass das Feld in mehrere Bereiche unterteilt wurde, die man nicht miteinander verbinden kann.

Um diese Regionen zu finden laufen wir mithilfe eines rekursiven Path-finding Algorithmus einmal über das Feld und überprüfen für jedes Feld ob sich bereits ein Deposit oder ein Obstacle darauf befindet, falls wir ein Objekt finden werden die Felder mit einem Objekt aus der Eingabe gematched. Wenn alle Felder besucht wurden brechen wir ab, falls noch nicht alle Felder besucht wurden suchen wir an den umliegenden Stellen rekursiv weiter.

3.3 Distance Maps

Als nächstes generieren wir für jedes Deposit eine Distance Map. Eine Distance Map ist eine Map die für jede Zelle die Manhattan Distanz zu einem bestimmten Gebäude enthält.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	14	15	16	17	16	15	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37
1	13	.	.	.	.	.	13	14	15	16	17	.	.	22	23	24	25	26	27	28	29	30	.	.	.	.	.	.	.	38
2	12	.	.	.	.	.	12	13	14	15	16	.	.	23	24	25	26	27	28	29	30	31	.	.	.	.	.	.	.	39
3	11	.	.	.	.	.	11	12	13	14	15	.	.	24	25	26	27	28	29	30	31	32	.	.	.	.	.	.	.	40
4	10	.	.	.	.	.	10	11	12	13	14	.	.	25	26	27	28	29	30	31	32	33	.	.	.	.	.	.	.	41
5	9	.	.	.	.	.	9	10	11	12	13	.	.	26	27	28	29	30	31	32	33	34	.	.	.	.	.	.	.	42
6	8	7	7	7	7	7	8	9	10	11	12	.	.	27	28	29	30	31	32	33	34	35	.	.	.	.	.	.	.	43
7	7	6	6	6	6	6	7	8	9	10	11	.	.	28	29	30	31	32	33	34	35	36	.	.	.	.	.	.	.	44
8	6	5	5	5	5	5	6	7	8	9	10	.	.	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
9	5	4	4	4	4	4	5	6	7	8	9	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
10	4	3	3	3	3	3	4	5	6	7	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
11	3	2	2	2	2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
12	2	1	1	1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
13	1	0	0	0	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
14	0	.	.	.	.	.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
15	0	.	.	.	.	.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
16	0	.	.	.	.	.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
17	0	.	.	.	.	.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
18	0	.	.	.	.	.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
19	1	0	0	0	0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24

In dieser Distance Map zum Beispiel sehen wir die Distanzen von jeder Zelle zum Deposit unten links. Alle Zellen die als Punkt angezeigt werden sind mit Objekten belegt. Diese Distance Maps verwenden wir sowohl um die Factory zu platzieren, damit diese möglichst nah an bzw. möglichst gut zwischen den relevanten Deposits platziert wird und auch später um die kürzesten Wege von A nach B zu finden.

3.4 Produkte filtern

Als nächstes werden für jede Region anhand von einfachen Kriterien die Produkte gefiltert. Produkte die mehr Ressourcen oder andere Ressourcen benötigen als in einer Region vorhanden sind, können in dieser Region selbstverständlich nicht produziert werden.

3.5 Deposits gewichten

Nicht jede Ressource ist zur Produktion eines Produkts gleich wichtig. Wie viele Deposits von einer Ressource zur Verfügung stehen, wie viele Ressourcen zur Produktion eines Produkts benötigt werden und auch wie viele Ressourcen pro Deposit zur Verfügung stehen muss mit in Betracht gezogen werden um eine sinnvolle Gewichtung der Deposits zu erstellen.

Diese Gewichtung verwenden wir im nächsten Schritt um die Factory zu platzieren welche dann das gewünschte Produkt herstellt.

3.6 Factory platzieren

Zuerst werden alle Felder auf denen keine Factory platziert werden kann ausgefiltert und auch alle Factory Platzierungen die nicht alle nötigen Ressourcen rechtzeitig erhalten würden. Danach werden die möglichen Felder mithilfe der Deposit Gewichtung in eine Rangliste gebracht. Folgende Scores werden berechnet, gewichtet und aufsummiert um eine Gesamtpunktzahl für die potenziellen Factory Positionen zu erhalten:

1. Gesamtdistanz zu den Deposits
2. Gewichtete Distanz zu den Deposits
3. Differenz der Distanz zum nächsten Deposit und zum am weitesten entfernten Deposit
4. Abschätzung wie viele Produkte an der Factory Position hergestellt werden können

An dieser Stelle könnte unsere Lösung noch weiter optimiert werden um über möglichst viele Aufgaben hinweg ein besonders gutes Ranking zu erstellen. Perfekt wäre ein Ranking welches immer an erster Stelle eine 'perfekte' Lösung hat. In diesem Fall wäre das also eine Factory Position mit der unsere restliche Lösung die maximale Punktzahl erreicht, die über alle Factory Positions hinweg erreicht werden kann. Vermutlich gibt es keine Gewichtung der vier oben genannten Kriterien, die für jede mögliche Aufgabe immer eine perfekte Factory an erster Stelle platziert. Allerdings kann man sich natürlich einer möglichst guten Gewichtung annähern in dem man über viele Aufgaben optimiert.

3.7 Connection Tree

Im nächsten Schritt passiert der Großteil unserer Lösung. Das Verbinden von Deposits und Factories mithilfe von Minen und den Verbindungsbausteinen. Wie bereits oben beschrieben basiert unsere Lösung in sehr großen Teilen möglichst effizient viele Möglichkeiten auszuprobieren und mit vielversprechenden Varianten weiter zu arbeiten. Vor Allem im Falle der Verbindungen ist dies sehr zum Tragen gekommen.

Unsere Lösung baut zunächst einen Baum aus Minen-Platzierungen und daran geknüpfte Verbindungenn mit einer Suchtiefe n auf. Der beste Pfad in diesem Baum wird nach der Distanz des letzen Pfadsegments zu der Factory ausgewählt. Anschließend wird das erste Pfad Segment des besten Pfades platziert und von dort aus wird ein neuer Baum mit der Suchtiefe n generiert.

Dazu iterieren wir über zahlreiche unterschiedliche Reihenfolgen unserer Deposits die wir mit unserer Factory verbinden wollen, platzieren wenn möglich eine Mine am Deposit und versuchen diese dann mithilfe unserer Distance Map Schritt für Schritt mit der Factory zu verbinden.

Bei der Implementierung wurde darauf geachtet, dass nur einmal zu Beginn ein ConnectionTree allocated wird, der Baum ist hierbei nicht Hierarchich strukturiert, sondern besteht im Grunde nur aus einem linearen Array, bestehend aus Nodes, welche die Indizes ihrer Children Nodes enthalten. Hierdurch ist die Implementierung sehr Cache-Freundlich.

4 Verwendete Technologien

4.1 Rust

Unsere Lösung haben wir vollständig in Rust implementiert. Da wir vor allem auf Performance gesetzt haben und alle Komponenten soweit wie möglich optimieren wollten und unsere Lösung natürlich innerhalb eines Zeitlimits fertig werden musste, waren Sprachen mit Garbage-Collection ausgeschlossen. Rust hat es uns ermöglicht Code zu schreiben der das vorgegebene Problem effizient und schnell löst.

4.2 GitHub

Wir haben GitHub zum einen natürlich für die Zusammenarbeit während des Wettbewerbs verwendet, als auch um die Möglichkeiten von GitHub Actions zu nutzen.

5 Softwarearchitektur

Unsere Software Architektur legt Wert auf Wiederverwendbarkeit der einzelnen Komponenten. So sind die Komponenten, welche die eigentliche Logik zur Simulation und Lösung der Aufgaben, als Rust 'crates' definiert, und das CLI welches im Wettbewerb die Eingabe und Ausgabe behandelt, ruft diese bloß auf. Somit könnte ohne weitere Umstände eine GUI, oder ein Web-Server auf die gleichen Logikbausteine zurückgreifen ohne diese neu strukturieren oder kopieren zu müssen.

Die einzelen 'Crates' legen viel Wert auf Data-Driven Design statt durch undurchsichtige Objekt-Strukturen, den Fluss der Daten zu verschleiern. Obwohl für einzelne Datenstrukturen Methoden benutzt wurden, meist für simplere Zugriffe oder Berechnungen, sind die eigentlich tragenden Algorithmen meist in einfachen Funktionen umgesetzt und ein Blick auf deren Signatur lässt erkennen welche Daten benötigt und zurückgegeben werden.

6 Software Testing

Beim Testing greifen wir auf das in Rust standardmäßig enthaltene Testing-Framework zurück. Es wurden sowohl Unit-Tests für einzelne Komponenten, als auch Integrations-Tests wie z.B. für die ganze Simulation geschrieben. Viele dieser Tests enstanden aus Bug-Hunts und verhindern Regressionen beim Refactoring des existierenden Codes. Diese Tests werden zusätzlich auf GitHub via CI Actions automatisch bei jedem 'Push' ausgeführt.

7 Coding Conventions

Die Coding Conventions lehnen sich hauptsächlich an die in großen Teilen der Rust-Community etablierten Standards an. So wurde zur Formattierung des Codes z.B. 'rustfmt' genutzt. Um mögliche Bugs durch Verwechslung zu vermeiden wurde für häufig verwendete Indizes das New-Type Pattern angwendet. Die Indizes in den Bauteil Arrays der Simulation sind ein Tupel-Struct welches einen 16bit Integer enthält, Da beim Zugreifen auf dieses Array der genaue Datentyp erfordert wird, kann es nicht zu Verwechslungen mit zu anderen Datenstrukturen zugehörigen Indizes kommen. Da von Anfang an Performance im Mittelpunkt stand, wurde bei allen Datentypen darauf geachtet so wenig wie möglich Speicher in Anspruch zu nehmen, um dem Compiler bei der automatischen Vektorisierung durch SIMD Instruktionen in die Hände zu spielen und die größtmögliche Menge an relevanten Daten im Cache bereitstehen zu haben. So ist der Koordinaten Datentyp nur 2 Byte groß, da dies bereits das maximal 100x100 große Spielfeld abdeckt. Ressourcen sind in einem 8-elementigen Array bestehend aus 16bit Integern gespeichert, so können simple arithmetische Operationen von zwei Resourcen-Arrays mit einer einzigen MMX Instruktion durchgeführt werden.

8 Fazit

Mithilfe unserer theoretischen Ansätze und unserer performanten Implementierung dieser sind wir unserer Ansicht nach zu einer sehr guten Lösung der Aufgabe gelangt. Uns sind natürlich noch immer Schwächen unserer jetzigen Lösung bewusst. Allerdings ist unsere Abgabe in der Lage für verschiedenste Inputs eine gute Lösung bzw. teilweise auch eine optimale Lösung zu finden.