Aufgabe 1: Flohmarkt in Langdorf

Teilnahme-ID: 58138

Bearbeiter dieser Aufgabe:
Benedikt Kuder

19. April 2021

Inhaltsverzeichnis

Lösungsidee

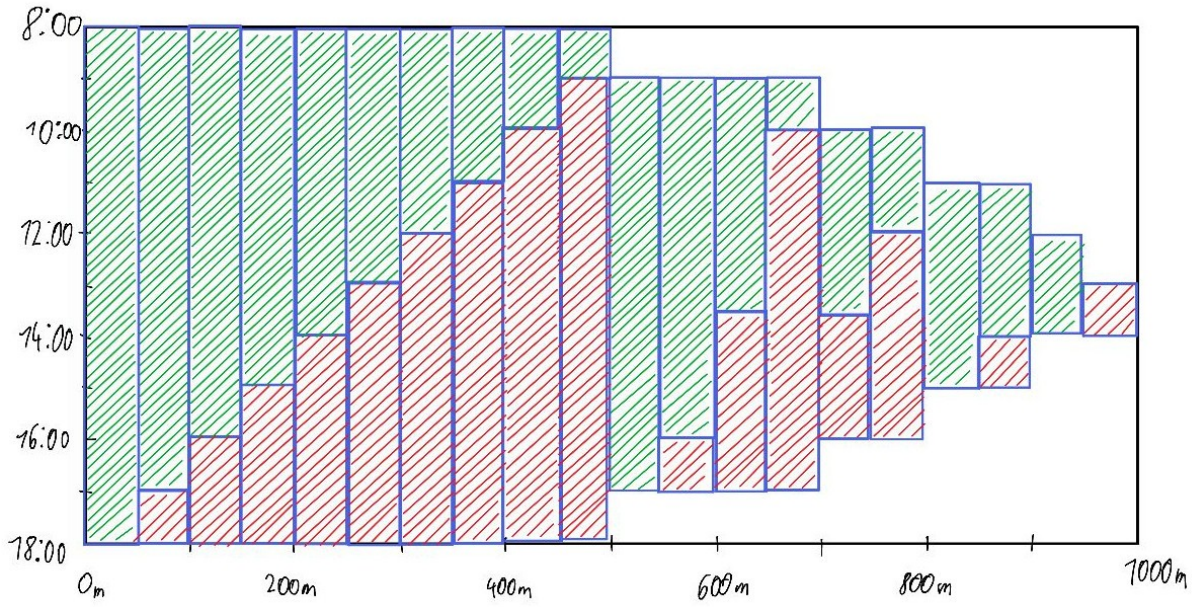
Zuerst soll das Problem besser veranschaulicht werden: Die Straße wird als großes Rechteck dargestellt und hat eine Breite von 1000m und eine Höhe von 10 Stunden (8h bis 18h). Die Anfragen sind kleinere Rechtecke mit der gewünschten Breite (in Metern) und einer Höhe von Endzeit – Anfangszeit. Da die Anfangszeit festgelegt ist, ist die Y-Koordinate auch als fest zu betrachten. Die X-Koordinate ist hingegen frei wählbar, vorausgesetzt, die Anfrage ist vollständig im Rechteck der Straße abbildbar. Damit ist die Aufgabe auf ein Packproblem reduziert worden.

Das Packproblem ist NP-Schwer („NP-Hard“) und damit nicht in angemessener Zeit vollständig lösbar.

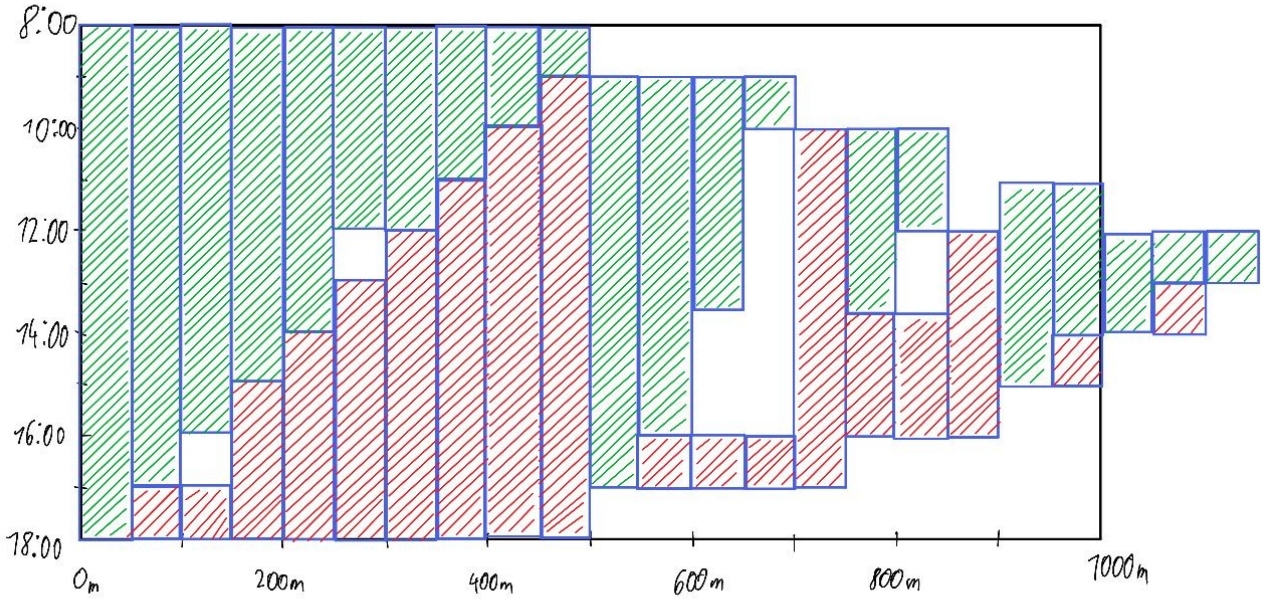
Deshalb wird sich der Lösung in zwei Schritten angenähert.

Schritt eins

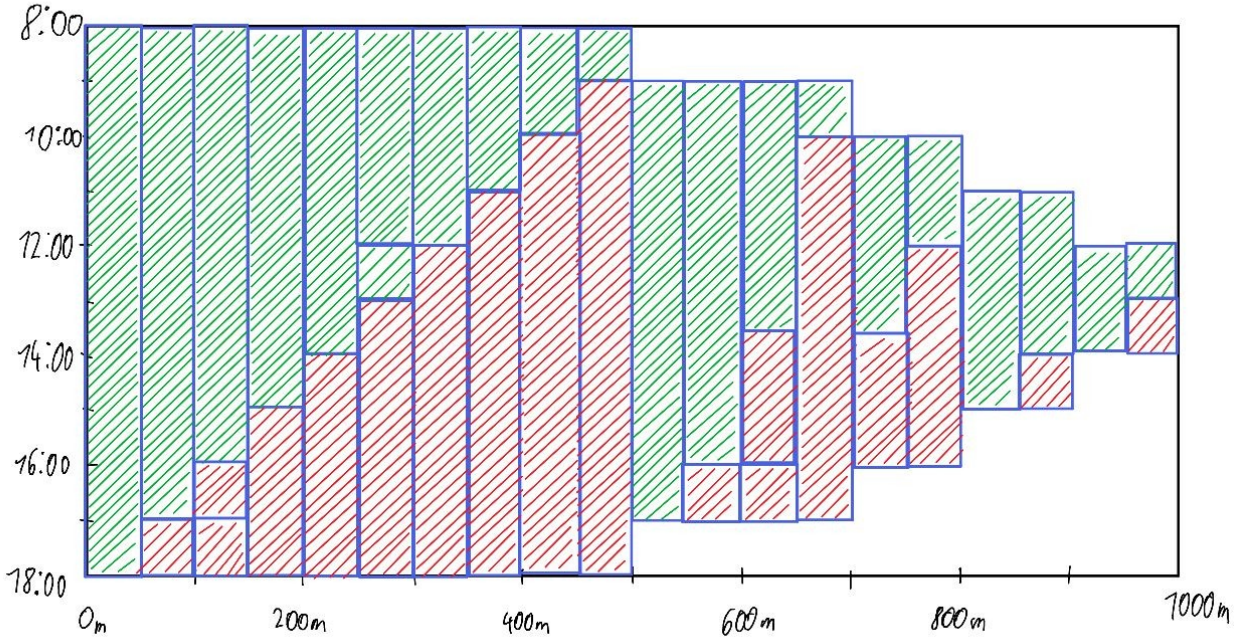
Es werden die Anfragen so sortiert, dass mehrere ‚Treppenmuster‘ entstehen, wobei die ‚Treppen‘ so angeordnet werden, dass jeweils zwei ineinander greifen (siehe folgende Abbildung).



Die jeweiligen ‚Treppen' sind dabei in grün und rot gekennzeichnet. Diese Abbildung zeigt einen Spezialfall, in welchem die ‚Treppen' perfekt aneinander passen. Dies ist nicht immer der Fall, wie in folgender Abbildung zu sehen ist.



In dieser Abbildung sind einige Lücken zu sehen. Daher passen nicht alle Anfragen in das Rechteck der Straße. Deshalb müssen die Lücken gefüllt werden. Dafür wird von links nach rechts die Lücke immer mit der Anfrage gefüllt welche in die Lücke passt und am wenigsten Platz frei lässt. Werden nach diesem Verfahren die Lücken gefüllt, sieht es danach folgendermaßen aus:



Schritt zwei

Nun gibt es allerdings auch Situationen, in welchem Schritt eins nicht zum besten Ergebnis führt. Dies ist vor allem der Fall, wenn nicht alle Anfragen angenommen werden können. Dafür gibt es einen zweiten Algorithmus. Es wurde dafür eine Abwandlung von Tabu-Search genutzt.

Tabu-Search ist ein Optimierungsalgorithmus welcher über mehrere Iterationen agiert. Dafür wird in jeder Iteration die aktuelle Belegung minimal durch das Tauschen zweier Anfragen abgeändert. Die n besten Variationen werden dann gespeichert. Dies wiederholt sich, bis eines der folgenden Abbruchkriterien eintritt.

Abbruchkriterien:

- Es wurden alle Anfragen untergebracht
- Es wurde die verfügbare Fläche vollständig genutzt
- Es gibt keine weiteren Variationen mehr bzw. die Cache ist leer
- Es wurde seit einer (festgelegten) Zeitspanne keine bessere Lösung mehr gefunden

Die ersten beiden Kriterien werden genutzt, um bei einer optimalen Verteilung abzubrechen, während die letzten beiden Kriterien auch eintreten können obwohl noch keine optimale Lösung gefunden wurde. Aufgrund der hohen Komplexität des Problems sind solche vorzeitigen Abbruchkriterien nötig, damit überhaupt ein Ergebnis verfügbar ist.

Umsetzung

Das Programm wurde in Rust geschrieben.
Zur graphischen Ausgabe wurde das crate/library ggez mit der Version 0.6.0-rc1 welche unter der MIT-Lizenz von „ggez-dev“ veröffentlicht wurde.

Datentypen

Road

Das *Road* Struct stellt die Straße dar. Es besteht aus drei *u32* welche Länge der Straße in Metern, Öffnungszeit in Stunden und Schließzeit in Stunden angibt.

Reservation

Das *Reservation* Struct stellt eine Anfrage nach einem Stellplatz dar. Es besteht aus drei *u32* welche Länge des Standes in Meter und Anfangs- und Endzeit in Stunden angibt. Ein weiteres Feld *color* repräsentiert die Farbe der Reservierung in der graphischen Ausgabe. Die Farbe wird bei Erstellung des Structes aus den anderen Feldern mit einer Hash Funktion bestimmt oder im monochrome Modus mit der Farbe Schwarz festgelegt. Der Datentyp *Color* ist aus dem crate ggez.

TimeTable

Das *TimeTable* Struct stellt eine Belegung der Straße dar und besteht aus einer *Road* und einem *Vec* welcher verwendet wird um Position und genutzte *Reservation*s darzustellen. Dafür wird ein Tuple aus den beiden Werten genutzt.

Solver

Das *Solver* Struct wird genutzt um ein Beispiel zu lösen. Es besteht aus einem *Slice* zu allen Reservierungen, einer Referenz zur genutzten *Road* und einem *Arc<Mutex<TimeTable>>* um das Ergebnis an die grafische Ausgabe zu übermitteln.

Neighbour

Das *Neighbour* Struct wird genutzt um eine Variation des Lageplans darzustellen.
Es besteht aus drei Feldern:

- einem *u32* welches die Flächennutzung wie in der Lösungsidee beschrieben repräsentiert
- einem *Vec<usize>* welche die Indizes der *Reservation*s im *Solver* Struct darstellt (dabei darf/muss jeder Index exakt einmal vorkommen)
- einer *usize* welche angibt bis zu welchem Index die *Reservation*s in das Rechteck der *Road* passen

View

Das *View* Struct wird ausschließlich zur graphischen Ausgabe verwendet und daher hier nicht weiter behandelt.

Implementierung von Schritt eins

Im Folgenden wird wiederholt von Reihenfolge von Reservierungen gesprochen. Die Reihenfolge von Reservierungen gibt die Lage im Rechteck der Straße an. Dafür kann man sich vorstellen, dass die Reservierung als Rechteck ganz rechts eingefügt wird, und dann solange nach links verschoben, bis sie an eine andere Reservierung angrenzt und daher nicht weiter verschoben werden kann. So bestimmt die Reihenfolge der Reservierungen auch deren Lage.

Schritt eins wird bei der Erstellung des *Solver* Structs von *Solver::new(reservations, road, target)* ausgeführt. Dafür wird *Solver::custom_sort(reservations, road)* aufgerufen.

Nun werden die *Reservation*s folgendermaßen sortiert, um die ‚Treppenform' zu erhalten: Alle *Reservation*s werden nach Startzeit und bei gleicher Startzeit nach Endzeit sortiert.
Darauf werden alle *Reservation*s welche nach der ersten *Reservation* starten, fallend nach Endzeit sortiert. Bei gleicher Endzeit wird fallend nach Öffnungszeit sortiert.
Diese zwei ‚Treppen' kann man nun erstmals als fest betrachten. Die verbliebenen Reservierungen werden weiter nach demselben Verfahren sortiert, bis alle Reservierungen sortiert wurden.

Diese Sortierung ergibt eine Zuteilung wie sie in den ersten beiden Abbildungen zu sehen ist.

Nun wird versucht die Lücken zu füllen.

Dafür wird die Funktion *Solver::find_gaps(road, reservations, start_i)* genutzt um eine Lücke zu finden.
Diese iteriert über alle *Reservation*s und speichert dabei immer die aktuellen Positionen der rechten Kante. Ist der aktuelle Index größer oder gleich dem übergebenen Startindex wird überprüft, ob durch das Einfügen der aktuellen *Reservation* eine freie Fläche entsteht. Wenn dies der Fall ist, wird der aktuelle Index und die Breiten pro Stunde der Fläche zurückgegeben. Sollte keine Lücke gefunden werden, wird *None* zurückgegeben.

Wurden die Werte zurückgegeben, wir *Solver::try_fill_gap(road, reservations, gaps, gaps_i)* mit diesen aufgerufen.
In dieser Funktion wird über alle *Reservations* mit einem höheren Index als der, der *Reservation* welche die Lücke von rechts einschließt (*gaps_i*) iteriert. Dabei wird für jede *Reservation* die Funktion *Solver::reservation_fits_gaps(…)* aufgerufen, welche einen Wert zurück gibt wie viel Platz beim füllen der Lücke über, links vom und unter der *Reservation* übrig bleiben würde. Es wird die *Reservation* mit dem niedrigsten Wert übernommen. Sollten alle *Solver::reservation_fits_gaps(…)* Aufrufe *None* zurückgeben, weil z.B. keine *Reservation* in die Lücke passt, wird *None* zurückgegeben. Andernfalls wird die *Reservation* mit dem niedrigsten an die entsprechende Stelle eingefügt. Dafür wird sie mit der *Reservation* mit dem höchsten Index getauscht und darauf alle *Reservations* hinter der Lücke um eins nach rechts rotiert.

Die Funktion *Solver::reservation_fits_gaps(road, reservation, gaps)* überprüft, ob der kleinste Wert von *gaps* im Zeitbereich von *reservation* größer gleich der Breite von *reservation* ist. Ist dies nicht der Fall, wird *None* zurrückgegeben. Andernfalls wird die Summe an freien Flächen über, links von und unter der *Reservation* zurückgegeben. Diese Art zur Berechnung sorgt dafür, dass Lücken

welche nicht ideal gefüllt werden können, so gefüllt werden, dass die verbliebene Fläche noch möglichst gut durch andere Reservierungen gefüllt werden kann.

Implementierung von Schritt zwei

Der zweite Schritt wurde in der Methode *Solver::solve()* umgesetzt. Dafür wird die erste Permutation der *Reservation*s in *Solver::generate_first()* erstellt. Diese ist die Reihenfolge wie sie im vorherigen Schritt sortiert wurde.

Es werden mehrere Variablen angelegt: *tabu* ist ein *BTreeSet<Neighbour>* welches bereits generierte Permutationen speichert, und *cache* ein *VecDeque<Neighbour>* welcher die besten zuletzt generierten *Neighbours* speichert. Die erste Permutation wird darauf in *cache* und in einer weiteren Variable *best_neighbour* gespeichert.

Nun wird in einer while-Schleife solange *cache* nicht leer ist folgendes Ausgeführt:

1. Es wird der *Neighbour* mit der besten Flächenabdeckung, welcher an letzter Stelle in *cache* ist, in die Variable *current* gespeichert.
2. Wenn eines der ersten beiden Abbruchkriterien aus der Lösungsidee erfüllt ist, wird das Ergebnis ausgegeben und die Schleife abgebrochen.
3. Wenn seit einer vorher angegebenen Zeit kein Ergebnis gefunden wurde wird die Schleife abgebrochen.
4. Wenn *current* in *tabu* existiert wird die Schleife mit dem nächsten Element in *cache* wiederholt.
5. Wenn *current* mehr Fläche abdeckt als *best_neighbour* wird *best_neighbour* ersetzt und das aktuelle Ergebnis ausgegeben.
6. Wenn *cache* mehr als 1000 Elemente enthält, wird solange das erste Element entfernt, bis dies der Fall ist.
7. Es werden mit *Solver::generate_neighbours* Abänderungen von *current* erzeugt, und alle deren abgedeckte Fläche größer gleich der von *current* ist werden *cache* mit steigender Flächennutzung gespeichert.

Schritt 6 wird benötigt, da sonst der Arbeitsspeicher voll laufen würde. Es werden dabei Permutationen verworfen, was dazu führt, dass nicht garantiert das beste Ergebnis berechnet wird. Es wird sich eher einem lokalem Optimum angenähert. Der Algorithmus wird sehr selten eine schlechtere Variation in Betracht ziehen, um im nächsten Schritt auf ein höheres globales Optimum zu kommen.

Die Funktion *Solver::generate_neighbours(index, selected)* generiert Abwandlungen der aktuellen Permutation welche in *index* gespeichert ist, indem sie zwei Einträge tauscht. Der erste der beiden Einträge muss dabei in der aktuellen Verteilung einen Standplatz haben und der zweite Eintrag muss einen höheren Index haben als der erste.

Um dies zu erreichen wurde zwei verschachtelte for-Schleifen genutzt. Die erste Schleife iteriert von 0 bis *selected* und speichert den Wert in *a*. Die zweite Schleife iteriert von *a* bis zum Index des letzten Elements in *index* und speichert den Wert in *b*. In jeder Iteration inneren Schleife werden die Einträge mit den Indizes *a* und *b* in *index* getauscht. Darauf wird mit *Solver::calculate_positions(index)* die Flächennutzung und die Anzahl an Reservierungen mit Stellplatz berechnet. Aus diesen Werten und *index* wird ein neuer *Neighbour* erstellt. Die 100 *Neighbour*s mit der höchsten Flächennutzung werden in einem *BTreeSet<Neighbour>* gespeichert und zurückgegeben.
Da nur die 100 besten *Neighbours* zurückgegeben werden, ist es möglich, dass die optimale Lösung nie berechnet wird.

Beispiele

Das Programm wurde für Windows und Linux kompeliert. Die Windows-Version ist nicht vollständig getestet. Bei8 Fehlern ist auf die Linux-Version zurückzugreifen.

Das Programm muss mit dem Dateipfad als ersten Parameter aufgerufen werden. Zusätzlich sind folgende Parameter möglich:

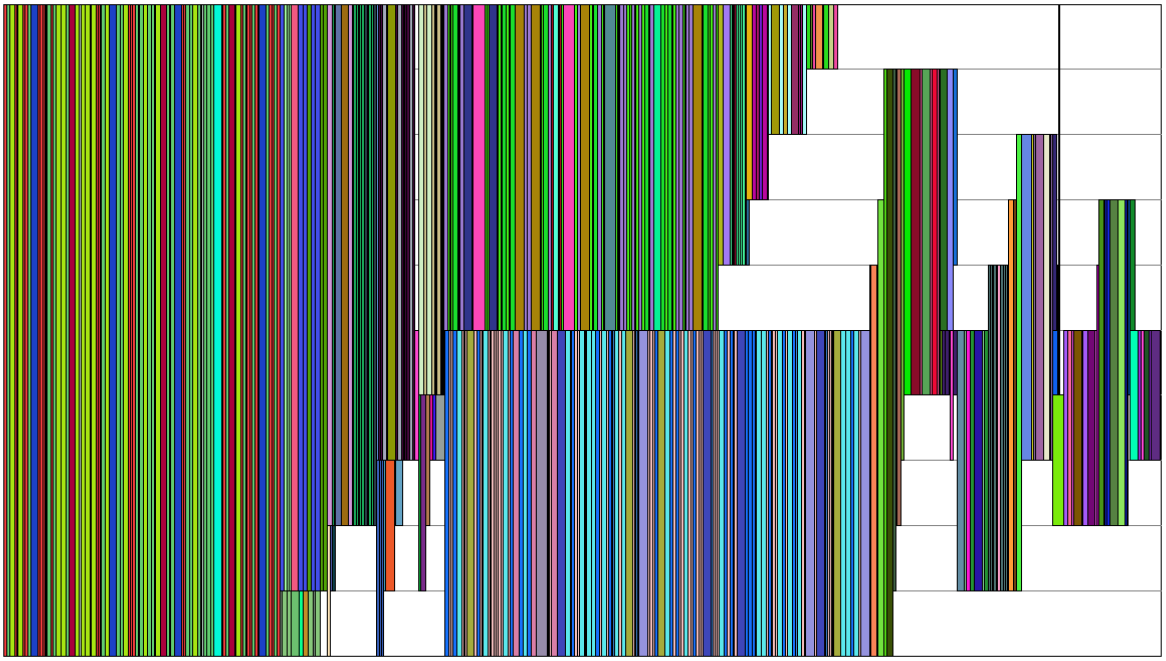
- „-custom“ Ermöglöicht das Einlesen von Dateien welche in der ersten Zeile mit Leerzeichen getrennt Öffnungszeit, Schließzeit und Länge der Straße angeben. Die Restlichen Zeilen müssen wie gehabt formatiert sein.
- „-no-ui“ Das Programm nutzt keine graphische Ausgabe
- „-monochrome“ Alle Anfragen werden schwarz dargestellt.
- „-timeout <Time>“ Gibt an, wie lange das Programm seit der letzten Verbesserung des Lageplans weiter nach Lösungen sucht, bevor es abbricht.

Im Folgenden wird, solange nicht anderst erwehnt das Programm mit „./aufgabe1 <Pfad zum Beispiel> -timeout 45“ aufgerufen.

Da die Textausgaben sehr lange sein können, wird hier ledigglich eine gekürzte Version abgebildet. Die vollständigen Ausgaben sind der Dokumentation als einzelne Dateien beigelegt.

Die graphische Ausgabe stellt die Anfragen wie in der Lösungsidee beschrieben als Rechtecke dar.

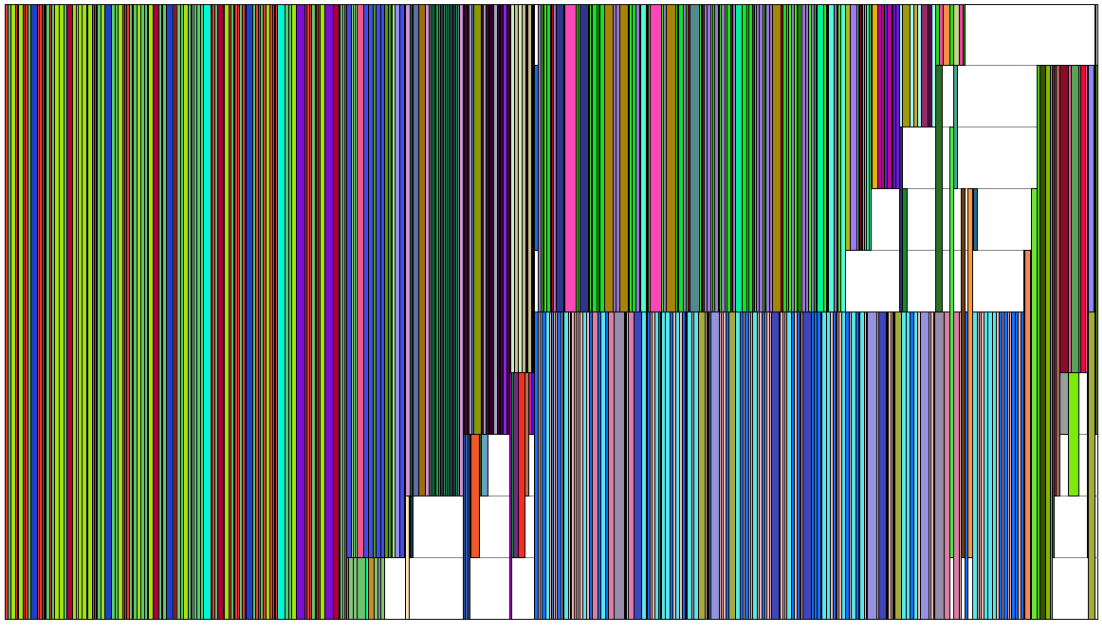
Beispiel flohmarkt1.txt



```
Publish| usage: 8028 used_reservations: 490/490
Completely Solved!
Zeitplan Flohmarkt, Start: 8, Ende: 18, Breite: 1000, akzeptierte
Reservierungen: 490
```

Das Beispiel wurde in weit unter einer Sekunde gelöst. Der Umsatz liegt bei 8028€. Es wurden alle Reservierungen angenommen. Schritt zwei hat nur eine Iteration gebraucht um das Ergebnis aus Schritt eins zu perfektionieren.

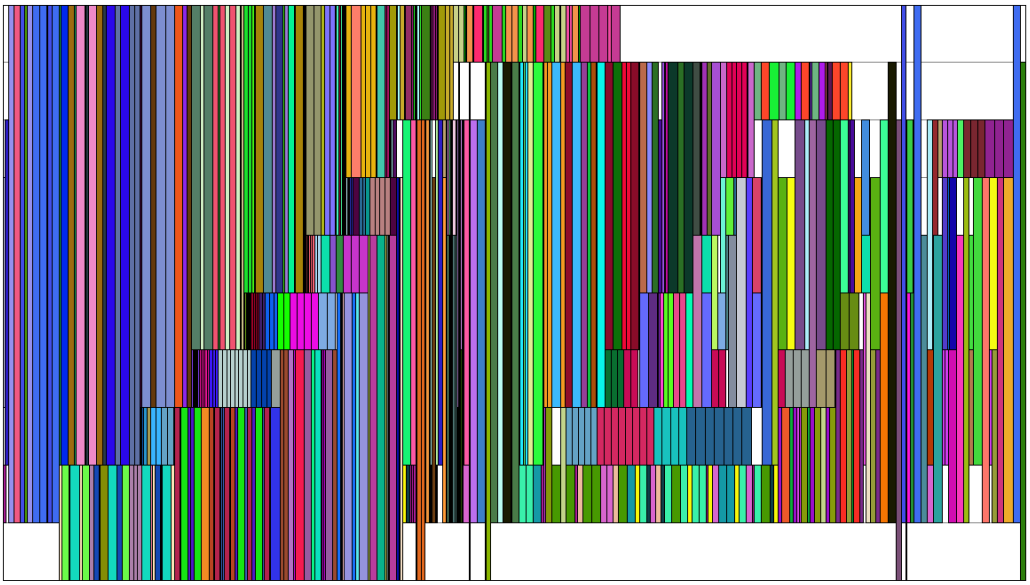
Beispiel flohmarkt2.txt



```
Time: 8.579577146
Publish| usage: 9075 used_reservations: 519/603
Timeout of 45 seconds reached
Zeitplan Flohmarkt, Start: 8, Ende: 18, Breite: 1000, akzeptierte
Reservierungen: 519
```

Diese Lösung wurde nach ca. 8,6 Sekunden erreicht. Nach 45 weiteren Sekunden wurde abgebrochen, da keine besseren Verteilungen mehr gefunden wurden. Der Umsatz liegt bei 9075€

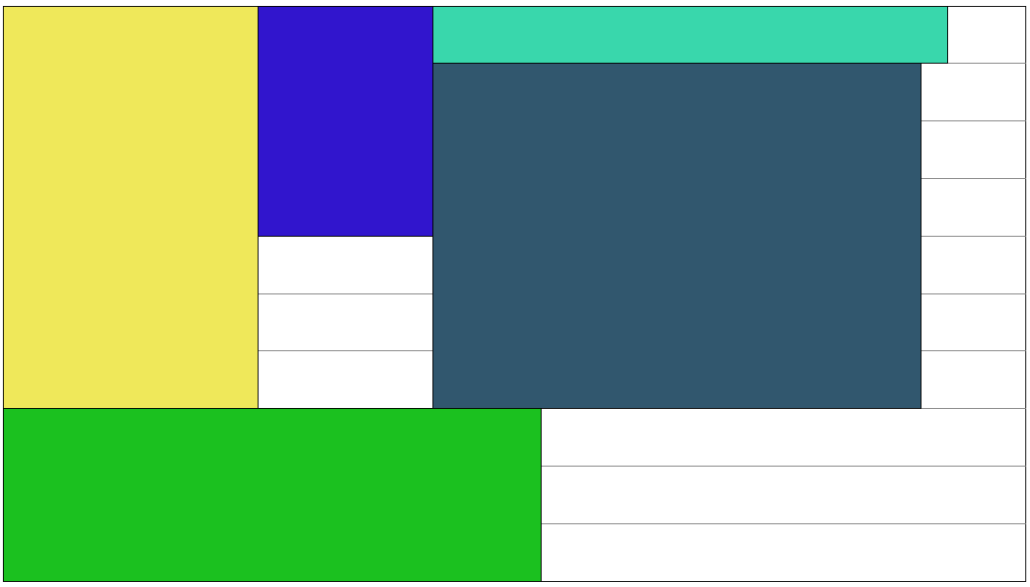
Beispiel flohmarkt3.txt



```
Time: 118.391637367
Publish| usage: 8641 used_reservations: 636/735
Timeout of 45 seconds reached
Zeitplan Flohmarkt, Start: 8, Ende: 18, Breite: 1000, akzeptierte
Reservierungen: 636
```

Diese Lösung wurde nach ca. 118 Sekunden erreicht. Nach 45 weiteren Sekunden wurde abgebrochen, da keine besseren Verteilungen mehr gefunden wurden. Der Umsatz liegt bei 8641€.

Beispiel flohmarkt4.txt

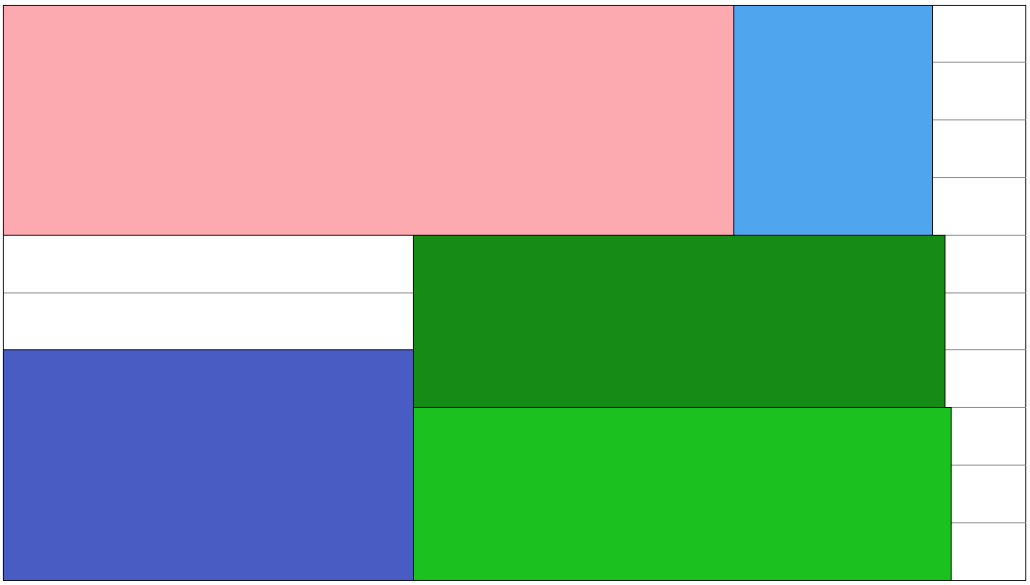


```
Time: 0.000088101
Publish| usage: 7370 used_reservations: 5/7
```

```
Zeitplan Flohmarkt, Start: 8, Ende: 18, Breite: 1000, akzeptierte
Reservierungen: 5
```

Diese Lösung wurde in weit unter einer Sekunde errechnet. Der Umsatz liegt bei 7370€. Es wurde abgebrochen, da die cache leer wurde.

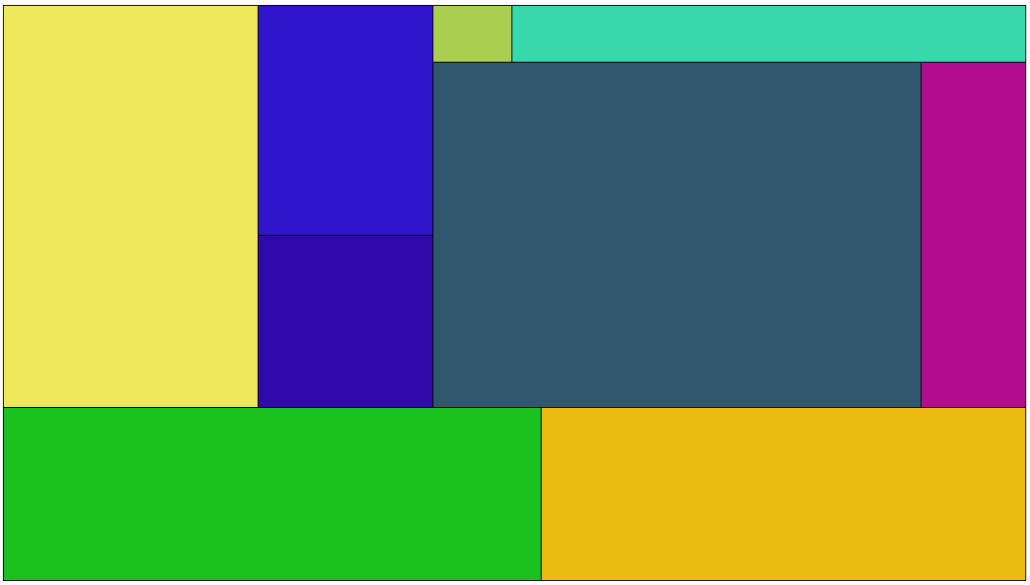
Beispiel flohmarkt5.txt



```
Time: 0.002213299
Publish| usage: 8374 used_reservations: 5/25
Zeitplan Flohmarkt, Start: 8, Ende: 18, Breite: 1000, akzeptierte
Reservierungen: 5
```

Diese Lösung wurde in weit unter einer Sekunde errechnet. Der Umsatz liegt bei 8374€. Es wurde abgebrochen, da die cache leer wurde.

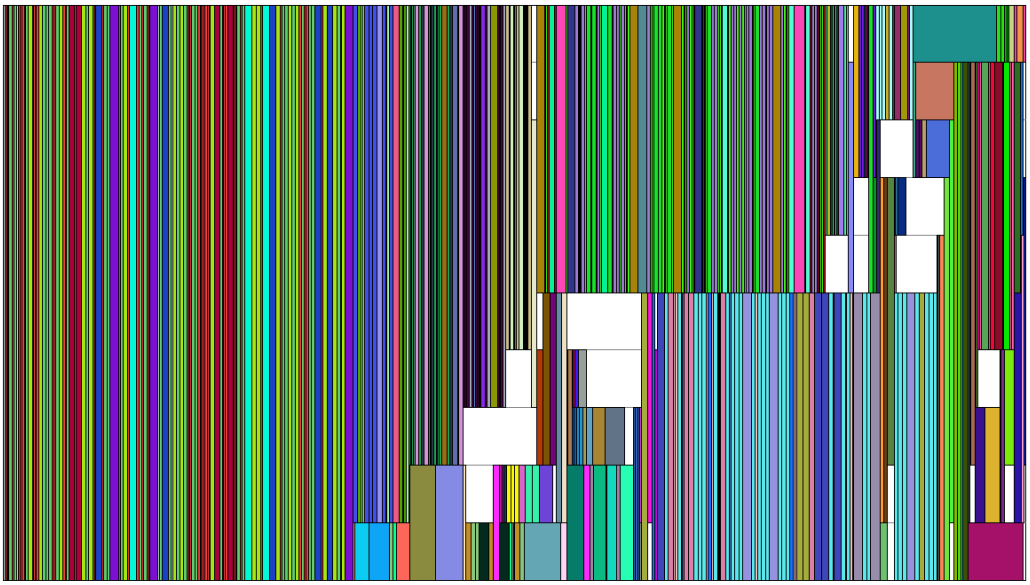
Beispiel flohmarkt6.txt



```
Publish| usage: 10000 used_reservations: 9/9
Completely Solved!
Zeitplan Flohmarkt, Start: 8, Ende: 18, Breite: 1000, akzeptierte
Reservierungen: 9
```

Diese Lösung wurde in weit unter einer Sekunde errechnet. Der Umsatz liegt bei 10000€. Es wurde abgebrochen, da sowohl 1000% der verfügbaren Fläche gnutzt wird als auch alle Anfragen berücksichtigt wurden.

Beispiel flohmarkt7.txt

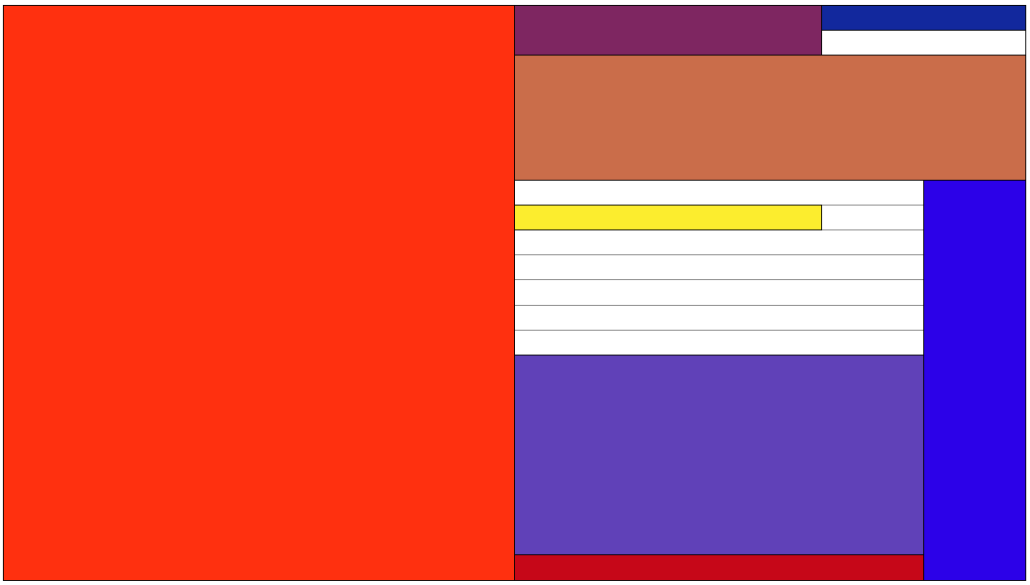


```
Time: 26.139606819
Publish| usage: 9479 used_reservations: 507/566
Timeout of 45 seconds reached
Zeitplan Flohmarkt, Start: 8, Ende: 18, Breite: 1000, akzeptierte
Reservierungen: 507
```

Diese Lösung wurde nach ca. 26 Sekunden errechnet. Nach 45 weiteren Sekunden wurde abgebrochen, da keine besseren Verteilungen mehr gefunden wurden. Der Umsatz liegt bei 9479€.

Beispiel flohmarkt99.txt

Dieses Beispiel nutzt eine eigene Beispieldatei und wurde mit folgendem command aufgerufen: „./aufgabe2 flohmarkt99.txt -custom“



```
Time: 0.000147282
Publish| usage: 203 used_reservations: 8/13
Zeitplan Flohmarkt, Start: 1, Ende: 24, Breite: 10, akzeptierte
Reservierungen: 8
```

Diese Lösung wurde in weit unter einer Sekunde berechnet. Es wurde abgebrochen, da die cache leer gelaufen ist. Der Umsatz liegt bei 203€.

Dieses Beispiel zeigt, dass das Programm auch mit anderstförmigen Straßen arbeiten kann. In diesem Beispiel ist die Straße nur 10 Meter lang. Zusätzlich hat in diesem Beispiel der Flohmarkt von 1 Uhr bis 24 Uhr offen.

Quellcode

```
pub struct Solver<'res> {
    reservations: &'res[Reservation],
    road: &'res Road,
    target: Arc<Mutex<TimeTable>>,
}

    fn custom_sort(reservations: &mut [Reservation], road: &Road) {
        let mut start_seq = 0;
        let mut borders = vec![1];
        //Create 'stairs'
        for i in 0..road.duration() {
            let part = reservations.split_at_mut(start_seq).1;

            if i%2 == 0 {
                //Sort by start time
                part.sort_by(|a, b| a.start().cmp(&b.start()).then(b.end().cmp(&a.end())));

                let start = part.first().and_then(|e| Some(e.start())).unwrap_or(0);
                while start_seq < reservations.len() && reservations[start_seq].start() == start {
                    start_seq += 1;
                }
            } else {
                //sort by end time
                part.sort_by(|a, b| b.end().cmp(&a.end()).then(b.start().cmp(&a.start())));

                let end = part.first().and_then(|e| Some(e.end())).unwrap_or(0);
                while start_seq < reservations.len() && reservations[start_seq].end() == end {
                    start_seq += 1;
                }
            }
            borders.push(start_seq);
        }

        let mut i = 0;
        while i < reservations.len() {
            if let Some((gaps_i, gaps)) = Self::find_gap(road, reservations, i) {
                let changed = Self::try_fill_gap(road, reservations, &gaps, gaps_i);

                if !changed {
                    i = gaps_i +1;
                }
            } else {
                break;
            }
        }
    }
```

```
    fn find_gap(road: &Road, reservations: &[Reservation], start_i: usize) -> Option<(usize,
Vec<Position>)> {
        let mut positions = vec![0; road.duration() as usize];
        for res in 0..reservations.len() {
            let reservation = &reservations[res];
            debug_assert!(reservation.start() >= road.start_time());
            debug_assert!(reservation.end() <= road.end_time());

            let mut position = 0;
            for i in (reservation.start() - road.start_time()) as usize..(reservation.end() - road.start_time()) as
usize {
                position = max(position, positions[i]);
            }

            let mut gaps = vec![0; positions.len()];
            for i in (reservation.start() - road.start_time()) as usize..(reservation.end() - road.start_time()) as
usize {
                gaps[i] = position - positions[i];
                positions[i] = position + reservation.length();
            }

            if res >= start_i {
                if gaps.iter().find(|e| e != &&0).is_some() {
                    return Some((res, gaps));
                }
            }
            gaps.fill(0);
        }
        return None;
    }

    fn try_fill_gap(road: &Road, reservations: &mut [Reservation], gaps: &[Position], gaps_i: usize) -> bool {
        let reservations = &mut reservations[gaps_i..];
        let mut best_fit_i = None;
        let mut best_fit_waste = u32::MAX;
        for i in 1..reservations.len() {
            let waste = Self::reservation_fits_gaps(road, &reservations[i], gaps);
            if let Some(waste) = waste {
                if waste < best_fit_waste {
                    best_fit_waste = waste;
                    best_fit_i = Some(i);
                }
            }
        }

        if let Some(best_fit_i) = best_fit_i {
            reservations.swap(best_fit_i, reservations.len()-1);
            reservations.rotate_right(1);
            return true;
        } else {
            return false;
        }
    }
```

```
    fn reservation_fits_gaps(road: &Road, reservation: &Reservation, gaps: &[Position]) -> Option<u32> {
        let mut smallest = Position::MAX;
        for i in (reservation.start() - road.start_time())as usize..(reservation.end() - road.start_time()) as usize
{
            if reservation.length() > gaps[i] {
                return None;
            } else {
                smallest = min(smallest, gaps[i] - reservation.length());
            }
        }

        let mut waste = 0;
        for i in 0..(reservation.start() - road.start_time())as usize {
            waste += gaps[i];
        }
        for i in (reservation.start() - road.start_time())as usize..(reservation.end() - road.start_time())as usize
{
            waste += gaps[i] - smallest;
        }
        for i in (reservation.end() - road.start_time())as usize..gaps.len() {
            waste += gaps[i];
        }
        return Some(waste);
    }

    //Generates first permutation which is just [0, 1, 2, ... Reservations.len()-1]
    fn generate_first(&self) -> Vec<usize>{
        let mut index = Vec::with_capacity(self.reservations.len());
        for i in 0..self.reservations.len() {
            index.push(i);
        }
        return index;
    }

    //Generates permutations of current distribution while returning only the GENERATE_NUM_NEIGHBOURS
best ones
    fn generate_neighbours(&self, index: &mut Vec<usize>, selected: usize) -> BTreeSet<Neighbour> {
        let mut neightbours = BTreeSet::new();
        for a in 0..selected {
            for b in a..index.len() {
                if self.reservations[a] != self.reservations[b] {
                    index.swap(a, b);

                    let (_, usage, selected) = self.calculate_positions(&index);
                    if neightbours.len() < Self::GENERATE_NUM_NEIGHBOURS {
                        neightbours.insert(Neighbour{
                            usage,
                            index: index.clone(),
                            selected,
                        });
                    } else if usage > neightbours.first().unwrap().usage {
                        neightbours.pop_first();
                        neightbours.insert(Neighbour{
                            usage,
                            index: index.clone(),
                            selected,
                        });
                    }

                    index.swap(a, b); //swap back for next iteration
                }
            }
        }
        return neightbours;
```

```
    }

    //Calculates right edge positions, usage of surface and number of used reservations
    fn calculate_positions(&self, index: &[usize]) -> (Vec<Position>, Position, usize) {
        let mut positions = vec![0; self.road.duration() as usize];
        let mut usage = 0;
        let mut selected = 0;

        for i in index {
            let reservation = &self.reservations[*i];
            debug_assert!(reservation.start() >= self.road.start_time());
            debug_assert!(reservation.end() <= self.road.end_time());

            let mut position = 0;
            for i in (reservation.start() - self.road.start_time()) as usize..(reservation.end() -
self.road.start_time()) as usize {
                position = max(position, positions[i]);
            }

            if position + reservation.length() > self.road.length() {
                return (positions, usage, selected);
            }

            selected += 1;
            usage += reservation.surface();

            for i in (reservation.start() - self.road.start_time()) as usize..(reservation.end() -
self.road.start_time()) as usize {
                positions[i] = position + reservation.length();
            }
        }
        return (positions, usage, selected);
    }

    pub fn solve(mut self) {
        //inicialize time mesurement
        let instant = std::time::Instant::now();
        let mut last_changed = std::time::Instant::now();

        //Create first permutation
        let first = self.generate_first();

        //Display
        self.publish(&first);

        //Create first Neighbour struct
        let (_, first_usage, selected) = self.calculate_positions(&first);
        let mut cache = VecDeque::new();
        cache.push_back(Neighbour{
            usage: first_usage,
            index: first,
            selected,
        });

        let mut best_neighbour = cache[0].clone();

        let mut tabu = BTreeSet::new();
        while !cache.is_empty() {
            let mut current = cache.pop_back().unwrap();
            let current_usage = current.usage;

            if current.usage == self.road.duration() * self.road.length() || current.selected ==
self.reservations.len() {
                //If 100% of reservations or 100% of the available surface are used than return result.
```

```
                self.publish(&current.index);
                println!("Completely Solved!");
                break;
            }
            if last_changed.elapsed().as_secs() > *crate::TIMEOUT.deref() {
                //If for more than TIMEOUT seconds no new solution was found than abort and return result.
                println!("Timeout of {} seconds reached", *crate::TIMEOUT.deref());
                break;
            }


            if tabu.contains(&current) {
                //If current combination was already used, then use next one
                continue;
            }

            //If current Neighbour is better than best, replace best and display new one.
            if current.usage > best_neighbour.usage {
                println!("Time: {}", instant.elapsed().as_secs_f64());
                self.publish(&current.index);
                best_neighbour = current.clone();

                //Clear tabu as no entry worse than the new one could be generated
                tabu.clear();
                //Reset Timeout
                last_changed = std::time::Instant::now();
            }
            while cache.len() > Self::CACHE_SIZE {
                //Remove worst entry while cache is full
                cache.remove(0);
            }

            //Generate new Neighbours and put those in cache which use at least as much space as the
current one.
            self.generate_neighbours(&mut current.index, current.selected).into_iter()
                .filter(|e| e.usage >= current_usage)
                .for_each(|e| cache.push_back(e));
            tabu.insert(current);
        }
        println!("{}", self.target.lock().unwrap().deref());
    }
}

#[derive(Debug, Eq, PartialEq, Ord, PartialOrd, Clone)]
struct Neighbour {
    usage: Position,
    selected: usize,
    index: Vec<usize>,
}
```