System sortowniczy w grze Minecraft

Mikołaj Kaźmierczak 254287 | Jędrzej Jamnicki 254290 | 27.01.2021

1. Obiekt badań:

Badany obiekt to mockup farmy surowca z gry komputerowej Minecraft.

Rzeczywista farma z reguły składa się z wielu modułów produkujących surowiec. Kiedy każdy moduł farmy będzie w pełni gotowy do zebrania, uruchamia się mechanizm zbierający wszystkie moduły jednocześnie. W naszej symulacji te moduły są zastąpione przez sztuczne dozowniki. Każdy z nich dawkuje stałą ilość surowca na sekundę.

Następnie surowiec jest transportowany przy użyciu wody. Jedna linia transportowa może też być wykorzystywana przez wiele różnych typów farm, czyli transportować różne typy surowców. Z tego względu na linii stosuje się sortowniki, które przechwytują jedynie pożądany surowiec.

Każdy z nich ma stałe ograniczenie prędkości sortowania. Dodatkowo ograniczony jest ilością surowca który może jednocześnie znajdować się w sortowniku, przez co jeśli wszystkie sortowniki będą w danym momencie zajęte, surowiec nie trafi do żadnego i kontynuuje podróż linią transportową na końcu której znajduje się niszczarka, która pozbywa się nadmiaru surowca. Jeśli natomiast surowca jest zbyt mało, niektóre sortowniki nie są używane, a każdy taki dozownik ma swój koszt budowy - z tego względu chcemy ograniczyć ilość niepotrzebnych dozowników.

Docelowo chcemy więc sprawdzić jaką minimalną ilość sortowników należy zbudować dla danej ilości dozowników, jednocześnie zachowując jak największą procentowo ilość surowca przetworzonego przez system.

2. Parametry obiektu:

Zmienne:

- **d** :ilość dozowników
- **s** :ilość sortowników
- **a** :ilość surowca z jednego dozownika
- **n** :łączna ilość surowca z jednego zbioru
- **position** :pozycja dozowanego surowca relatywna do dozownika

Stałe:

- **u** :prędkość dawkowania surowca
- **v** :prędkość jednego sortownika
- k :prędkość surowca w wodzie
- **w** :pojemność sortownika

Wyjściowe:

- $\mathbf{p_i}$:ilość posortowanego surowca przez i-ty sortownik
- **z** :łączna ilość zniszczonego surowca

3. Wskaźniki:

Procentowa ilość posortowanego surowca względem ilości wejściowej:

$Q([p_1...p_i],n) = sum(p_i)/n*100$

Procentowa ilość sortowników, które zostały wykorzystane w procesie sortowania:

$X([p_1...p_i],s) = sum([1 \text{ if } p_i!=0 \text{ else } 0 \text{ for } p_i \text{ in } p])/s*100$

4. Zakres:

$d = \{1, 2 ... \infty\}$

$s = \{1, 2 ... \infty\}$

$a = \{32, 64, ... 32*18\}$ *(zależna od ilości dozowników)*

$n = d*a$ *(zależna od ilości dozowników)*

position ~ N[0,1) (przy każdorazowym dozowaniu surowca losowana jest wartość jego pozycji relatywnej do jego położenia)

$u = 5$ i/s *(items per second)*

$v = 2.5$ i/s *(items per second)*

$k = 7.69$ m/s (*meters per second*) (za m przyjmuje się długość jednego bloku)

$w = 64-41 = 23$

Ze względu na sposób działania mechanik gry wszystkie parametry mają charakter deterministyczny oprócz parametru position, który jest probabilistyczny.

5. Cel:

Wyznacznik efektywności:

średnia z obu wskaźników : $avg(Q(p_i,n) + X(p_i,s))$

H: Czy zwiększenie ilości sortowników sprawi że system będzie bardziej efektywny?

6. Plan badań:

Dla każdej kombinacji zmiennych wejściowych należy wykonać symulację, a zwrócone zmienne wyjściowe oraz obliczone wartości wskaźników zwrócić do plików CSV.

$d = \{1, 2 ... 30\}$

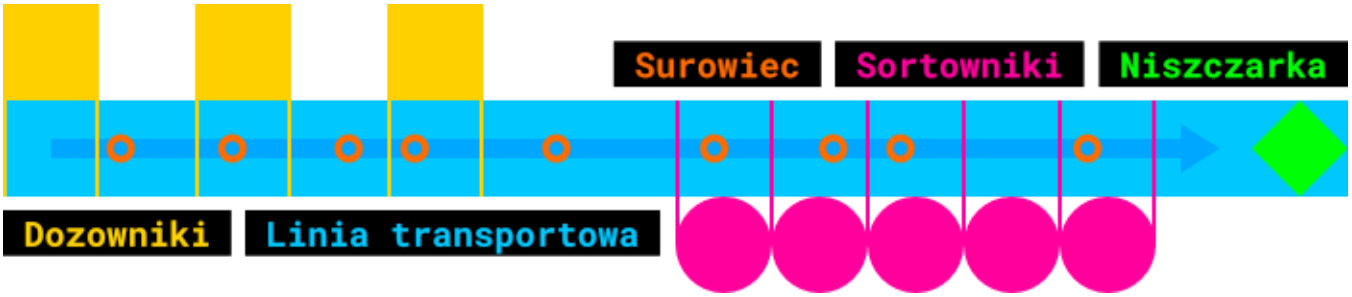
$s = \{1, 2 ... 30\}$

$a = \{32, 64, ... 32*18\}$

$n = d*a$

Ze względu na losowość symulację wykonano 9 razy.

7. Przebieg symulacji:



Na początek zaprojektowano obiekty dla 3 podsystemów obiektu badań: dozownika, sortownika, linii transportowej.

Następnie zaimplementowano symulację dla danej kombinacji zmiennych wejściowych. Do tego potrzebne było zaimplementowanie sztucznej pętli gry. W grze jedna sekunda w świecie rzeczywistym dzielona jest na 20 ticków. W każdym ticku wykonywane są liczne operacje. Między innymi obliczana jest fizyka (grawitacja, kolizje, prędkość, ...) i określane są następne operacje obiektów (bloków i innych elementów gry).

```
def simulate_combination(dsa):
  '''Przeprowadzenie symulacji dla danej kombinacji zmiennych wejściowych.'''

  # zmienne wejściowe obecnej kombinacji
  d,s,a = dsa
  n = d*a

  # utworzenie obiektów dozowników, sortowników i linii transportowej
  dispensers = [Dispenser(a=a, position=-i) for i in range(0,d*2, 2)]
  sorters = [Sorter(area=(i,i+1)) for i in range(s)]
  line = Line()

  # symulacja pętli gry
  ticks = 0
  while 1:
    for dis in dispensers:
      dis.dispense(line) # dozowanie surowca (co 2 ticki)
    line.flow(sorters)   # symulacja przepływu surowca w wodzie
    for sor in sorters:
      sor.collect(line)  # próba przejęcia surowca z linii transportowej
      sor.push()         # próba przesortowania surowca (co 4 ticki)
    # zakończenie symulacji gdy dozowniki i linia transportowa są puste
    if sum([dis.a for dis in dispensers])==0 and len(line.line)==0:
      break
    ticks += 1

  # obliczenie zmiennych wyjściowych i wskaźników
  p = [sor.p for sor in sorters]
  Q = sum(p)/n*100
  X = sum([1 if p_i!=0 else 0 for p_i in p])/s*100
  avgQX = (Q+X)/2
  T = ticks/20 # czas w sekundach

  # podsumowanie i zwrócenie wyników
  if SINGLE: summary(d,s,a,n,p,line,ticks,T,Q,X,avgQX)
  return (d,s,n,round(Q,4),round(X,4),round(avgQX,4),T)
```

```
class Dispenser():
  '''Obiekt dozownika.'''

  def __init__(self, a, position):
    '''Parametry:
    a:int -- ilość surowca w wewnętrznym magazynie
    position:int -- abstrakcyjna pozycja dozownika
    '''
    self.position = position
    self.a = a
    # 5i/s -> 1i/4t -> 1i/2rt (1s=20t=10rt)
    self.u = 2     # częstotliwość działania (ilość ticków)
    self.time = 0  # zegar odmierzający czas działania

  def dispense(self, line):
    '''Dozuj jedną jednostkę surowca.
    line:Line -- obiekt linii transportowej
    '''
    if self.a > 0:             # jeśli surowiec jest w magazynie
      self.time += 1           # odmierz czas i dozuj surowiec jeśli
      if self.time == self.u:  # upłynęła już odpowiednia ilość czasu
        # losuj wyjściową pozycję i umieść dozowany surowiec w linii transportowej
        line.catch(self.position - random.uniform(0,1))
        self.a -= 1    # usuń jedną jednostkę surowca
        self.time = 0  # i resetuj zegar
```

```
class Sorter():
  '''Obiekt sortownika.'''

  def __init__(self, area):
    '''Parametry:
    area:(int,int) -- przestrzeń na której pracuje dozownik
    '''
    self.area = area
    self.w = 23  # maksymalna ilość surowca w sortowniku
    self.a = 0   # ilość surowca w sortowniku
    # 2.5i/s -> 1i/8t -> 1i/4rt (1s=20t=10rt)
    self.v = 4     # częstotliwość działania (ilość ticków)
    self.time = 0  # wewnętrzny zegar odmierzający czas działania
    self.p = 0  # łączna ilość posortowanego surowca

  def collect(self, line):
    '''Spróbuj przejąć te jednostki surowca, które znajdują się obszarze pracy sortownika.
    line:Line -- obiekt linii transportowej
    '''
    # sprawdź z góry czy sortownik może przyjąć surowiec
    # (jeśli nie może to unikamy zasobożernych operacji)
    if self.a < self.w:
      area_products = []  # identyfikatory jednostek surowca w obszarze pracy sortownika
      # przejdź po wszystkich jednostkach surowca w linii transportowej
      # i dodaj do listy te, które są w obszarze pracy sortownika
      for i, product in enumerate(line.line):
        if product >= self.area[0] and product < self.area[1]:
          area_products.append(i)
      # próbuj usunąć surowiec z linii i przejąć przez sortownik
      for i in area_products[::-1]:
        if self.a < self.w:  # sprawdź czy sortownik może przyjąć surowiec
          line.remove(i)
          self.a += 1
```

[cd. na następnej stronie]

```
  def push(self):
    '''Próbuj przesortować jedną jednostkę surowca.'''
    if self.a > 0:    # jeśli surowiec jest w sortowniku
      self.time += 1  # odmierz czas
      # jeśli upłynęła odpowiednia ilość czasu
      # usuń surowiec z sortownika
      if self.time == self.v:
        self.a -= 1
        self.p += 1
        self.time = 0  # resetuj zegar
```

```
class Line():
  '''Obiekt linii transportowej.'''

  def __init__(self):
    self.line = []  # lista pozycji każdego wydozowanego surowca
    # 7.69m/s -> 0.385t/s (1s=20t)
    self.k = 0.385  # prędkość surowca w wodzie
    self.destroyed = 0  # ilość zniszczonego surowca

  def catch(self, position):
    '''Dodaj pozycję surowca do listy.
    position:float -- pozycja surowca
    '''
    self.line.append(position)

  def flow(self, sorters):
    '''Zmień pozycję wszystkich jednostek surowca w zależności od prędkości wody.
    sorters:list[Sorter] -- lista obiektów sortowników
    '''
    # przejdź po wszystkich pozycjach jednostek surowca
    for i, product in enumerate(self.line):
      self.line[i] += self.k  # zmień pozycję
      # jeśli jednostka surowca dotarła do końca linii - zniszcz ją
      if product >= sorters[len(sorters)-1].area[1]:
        self.destroy(i)

  def remove(self, i):
    '''Usuń jednostkę surowca z linii.'''
    del self.line[i]

  def destroy(self, i):
    '''Usuń jednostkę surowca z linii i zwiększ ilość zniszczonego surowca.'''
    self.remove(i)
    self.destroyed += 1
```

8. Wyniki symulacji:

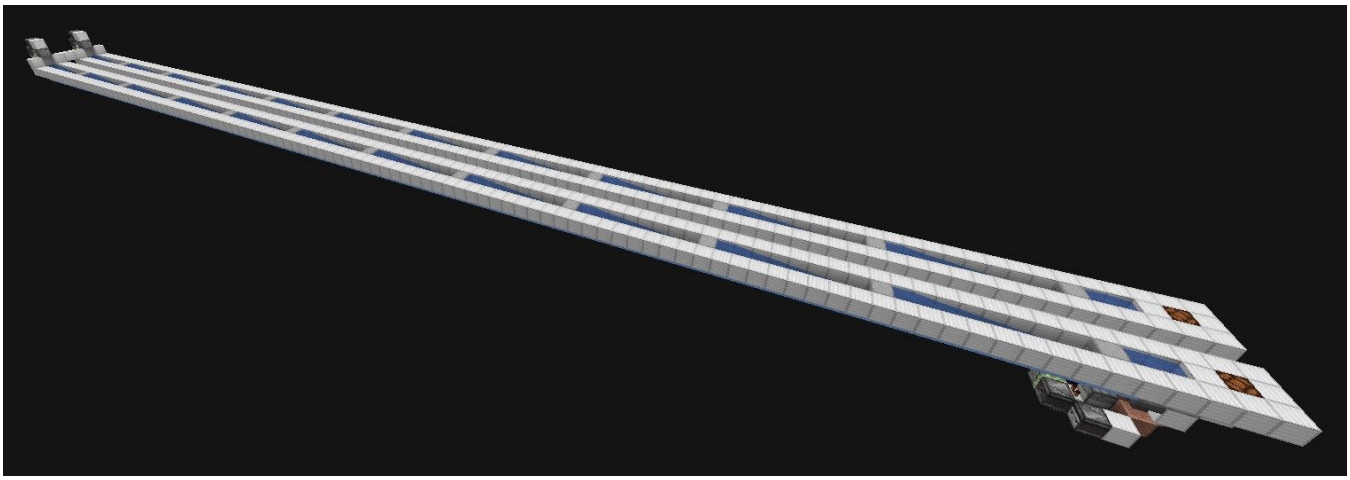
Wyniki 9 symulacji scalono i uzyskano 145800 krotek surowych danych:

	n	d	s	d/s	Q	X	avgQX	T
0	32	1	1	1.0	100.0000	100.0	100.0000	3.25
1	64	1	1	1.0	85.9375	100.0	92.9688	6.55
...	...	...	...	...	...	...	...	...
145798	16320	30	30	1.0	56.1826	100.0	78.0913	65.25
145799	17280	30	30	1.0	55.8507	100.0	77.9253	68.60

9. Walidacja danych:

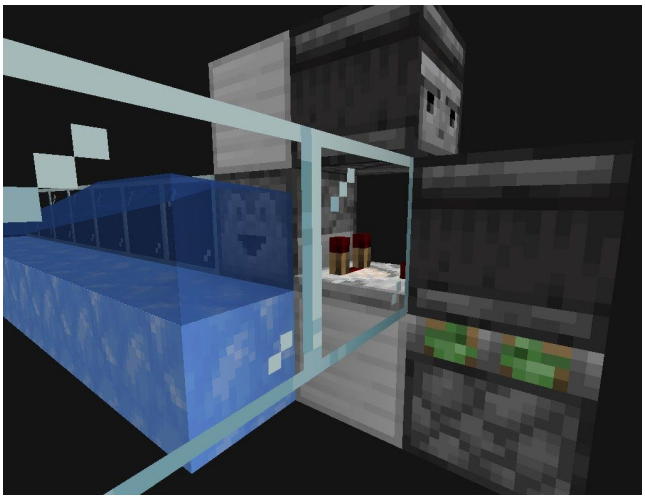
Aby przeprowadzić walidację danych otrzymanych podczas symulacji, zbudowano bezpośrednio w świecie gry jeden z wariantów opisywanego obiektu badań. W poniższym opisie pomijane są detale związane z działaniem gry, sposobem projektowania użytych mechanizmów oraz ich dokładnego działania.

Na początek należało przeprowadzić pomiar prędkości wody, który później wykorzystywano w symulacji. W tym celu zbudowano tor pomiarowy, gdzie wrzucano surowiec na jednym końcu, a na drugim po zakończeniu podróży surowca włączała się dioda sygnalizująca koniec pomiaru. Taki pomiar przeprowadzono 9 razy i uśredniono, uwzględniając przy tym niepewności pomiarowe związane z czasem reakcji człowieka:

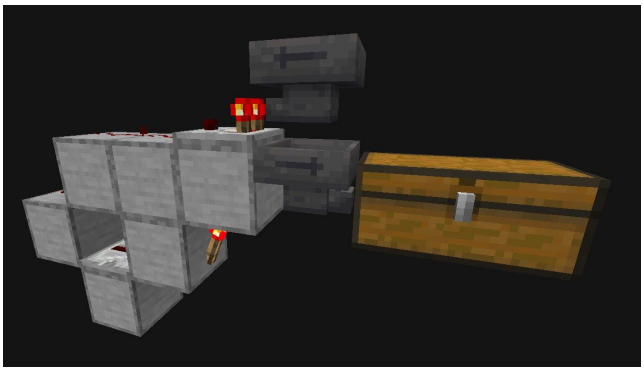


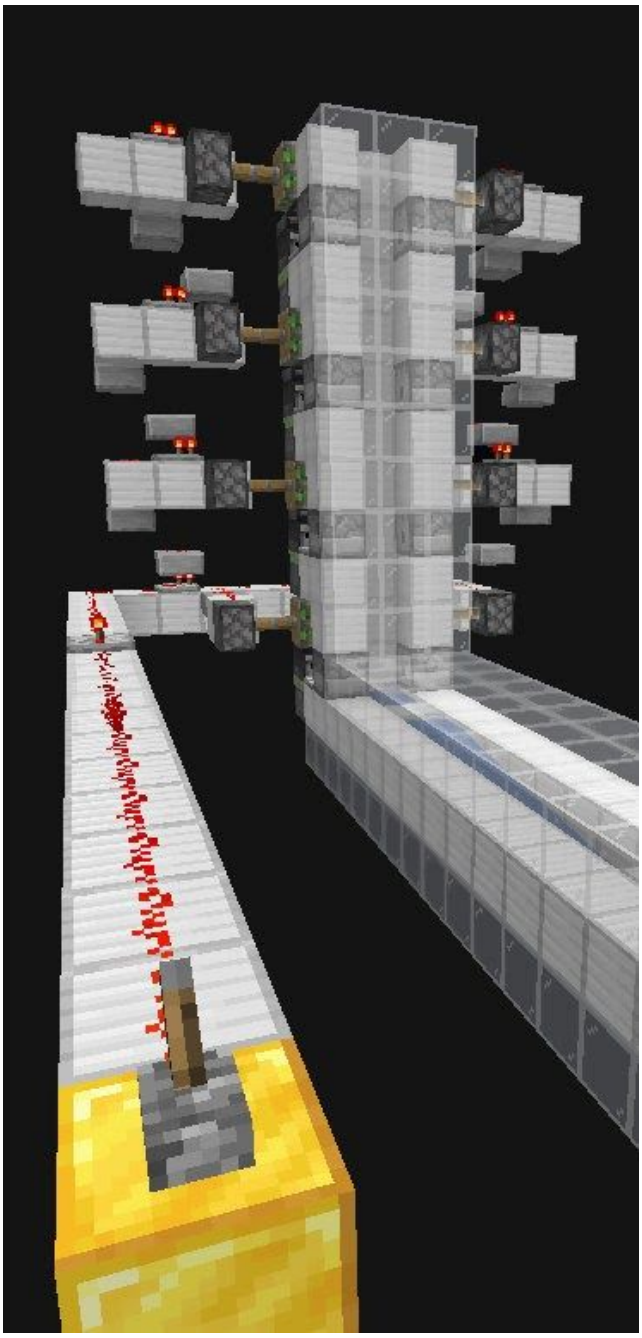
W następnej kolejności należało użyć znanych już w społeczności gry modułów do dozowania przedmiotów (symulującego np. moduł farmy) oraz sortowniczego.

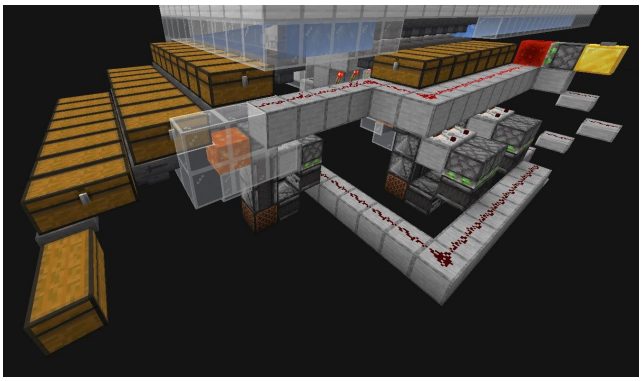
Użyty moduł dozowniczy:



Użyty moduł sortowniczy:





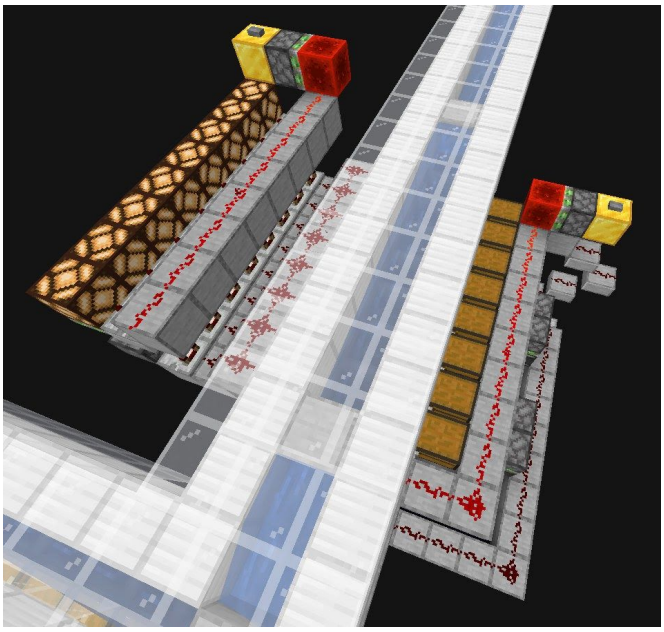


Następnie zaprojektowano system dozowniczy, który uruchamia wszystkie moduły dozownicze jednocześnie (po lewej).

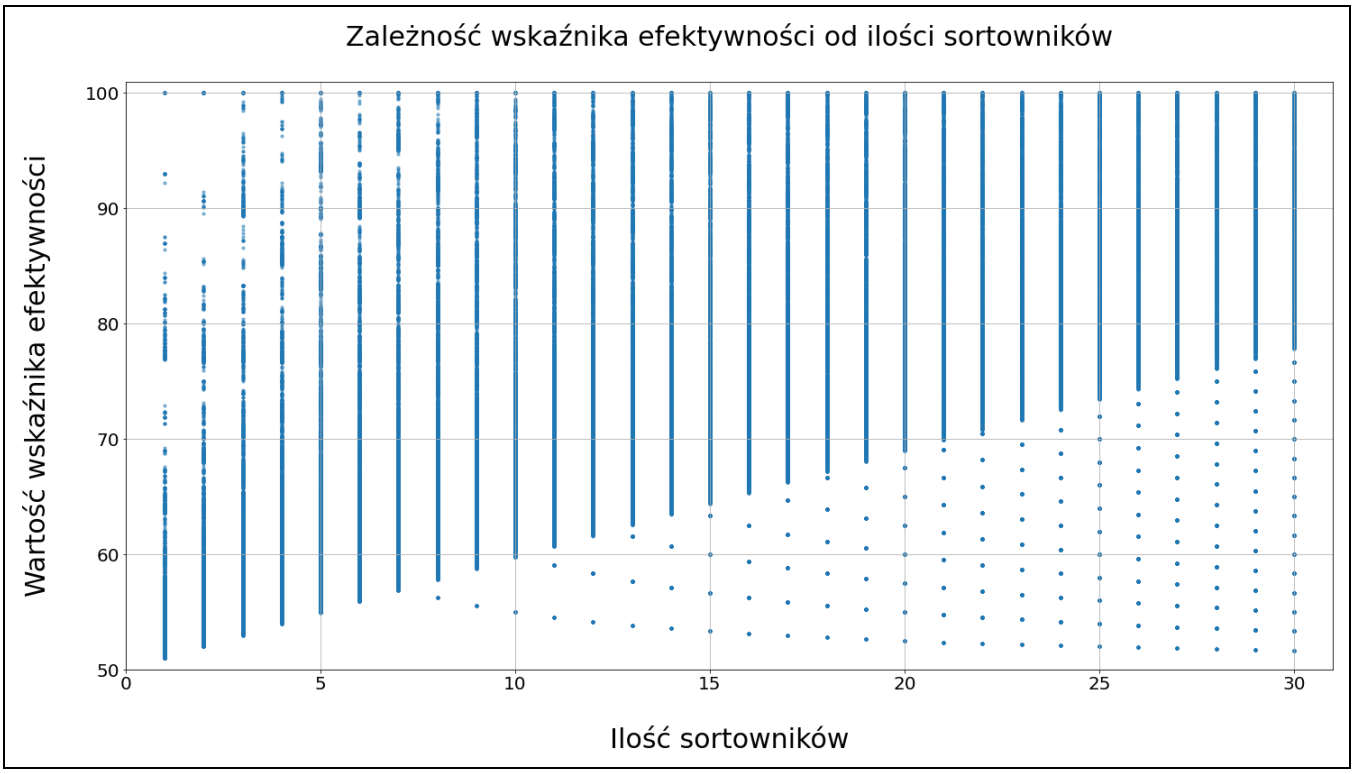
Zaimplementowano również system sortowniczy, gdzie moduły sortownicze są ustawione jeden obok drugiego. Pomiędzy systemami dozowniczym i sortowniczym jest linia transportowa, gdzie surowiec jest przesuwany wykorzystując wodę.

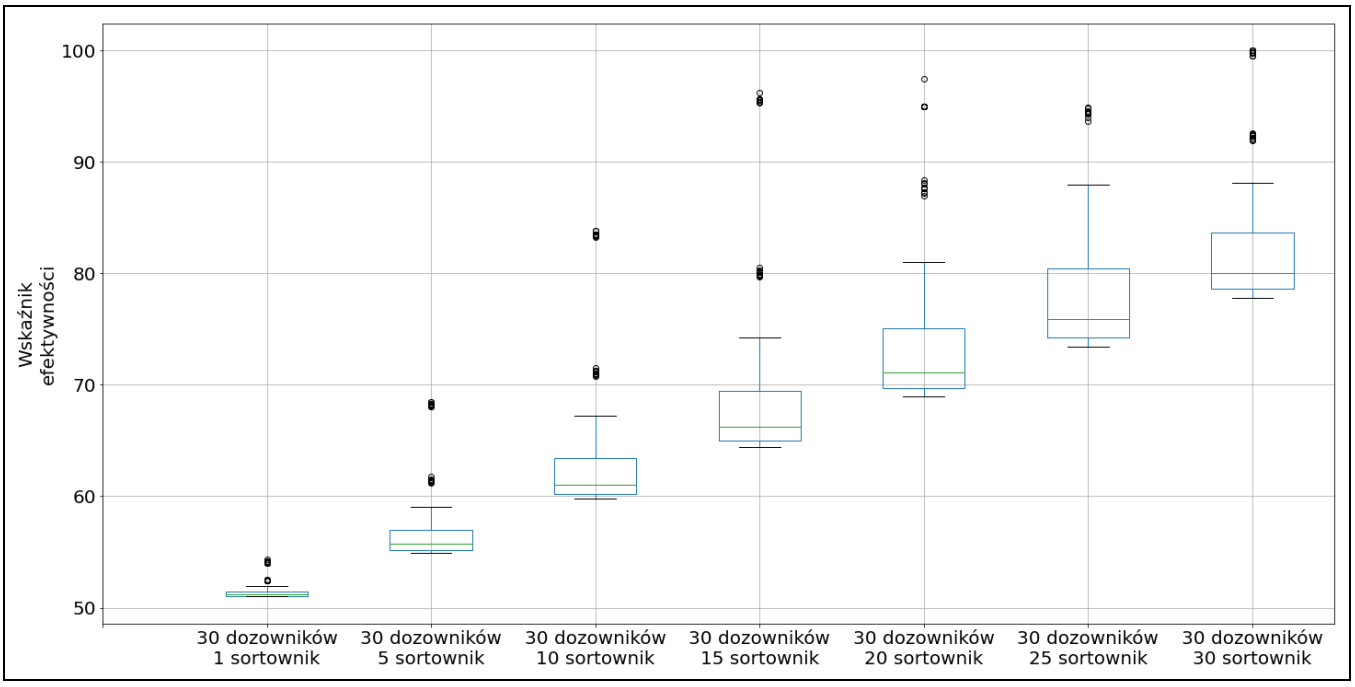
Posortowany surowiec trafia do magazynów, gdzie można sprawdzić ile surowca przetworzył każdy sortownik, a ile trafiło do niszczarki.

Jako pomoc w sprawdzeniu jaka ilość sortowników została wykorzystana przy sortowaniu zaprojektowano też system diod, gdzie jedna dioda odpowiadająca jednemu sortownikowi zapala się w momencie rozpoczęcia przez niego pracy:



10. Prezentacja wyników





11. Analiza wyników

Test Jonckheere-Terpstra dla trendu:

Test przeprowadzono za pomocą języka programowania R w środowisku RStudio wykorzystując bibliotekę DescTools.

Na podstawie wykresu pudełkowego przedstawionego powyżej założono, że trend populacji jest rosnący.

By można było przeprowadzić analizę trendu, należy wskazać oczekiwaną kolejność populacji przypisując im kolejne liczby naturalne.

Wiersze tabeli uporządkowano względem ilości sortowników w kolejności rosnącej. Następnie według nowej kolejności wybrano wartości wyznacznika efektywności i przypisano im liczby naturalne $\mathbb{N} = \{1, 2, 3, ..., 145800\}$

Przy znanym oczekiwanym kierunku trendu, hipoteza alternatywna jest jednostronna i interpretacji podlega jednostronna wartość p.

H_0: w badanej populacji wraz ze wzrostem ilości sortowników nie rośnie efektywność systemu

H_1: w badanej populacji efektywność systemu rośnie wraz ze wzrostem ilości sortowników

Test przeprowadzono na poziomie istotności $\alpha = 0.05$:

$p = 0.01$

Wyznaczoną na podstawie statystyki testowej wartość p porównujemy z poziomem istotności α:

jeżeli $p \leq \alpha \implies$ odrzucamy H_0 przyjmując H_1,

jeżeli $p > \alpha \implies$ nie ma podstaw, aby odrzucić H_0.

Uzyskana jednostronna wartość p jest mniejsza niż zadany poziom istotności $\alpha = 0.05$ więc odrzucono hipotezę H_0 przyjmując hipotezę H_1.

12. Podsumowanie

Z wyników symulacji i przeprowadzonych badań wynika, że im więcej sortowników względem danej ilości dozowników tym większa będzie efektywność systemu.