Dokumentacja wstępna projektu z przedmiotu Analiza Algorytmów[AAL] Tron Yertle

Paweł Kamiński

05-12-2015

1 Opis problemu

W jakiej kolejności należy ustawić żółwie jeden na drugim, aby utworzyć tron króla Yertle. Każdy z 5607 żółwi ma inną wagę i inną wytrzymałość. Zadaniem jest zbudować najwyższy możliwy stos żółwi.

2 Dane

2.1 Dane wejściowe

Kolekcja danych o parametrach co najwyżej 5607 żółwi (wytrzymałość, waga). Wytrzymałość oznacza ile żółw może udźwignąć włączając w to jego własną wagę.

2.2 Wynik

Liczba całkowita oznaczająca maksymalną liczbę żółwi, jakie można ułożyć na stosie, nie przekraczając przy tym wytrzymałości żadnego z żółwi.

3 Metody rozwiązania problemu

We wszystkich rozwiązaniach Tron Yertle jest układany od góry - najpierw wrzucamy żółwie, które będą na samej górze, a następnie szukamy kolejnych żółwi, które będą w stanie utrzymać już do tej pory ułożony stos.

3.1 Programowanie dynamiczne

Na samym początku algorytmu żółwie są sortowane. Operator porównania jest zdefiniowany następująco:

```
bool Turtle::operator<(const Turtle& t) const
{
  if (getWeight() == t.getWeight())
    return getStrength() < t.getStrength();
  return getWeight() < t.getWeight();
}
```

Następnie tworzona jest tablica dwuwymiarowa o rozmiarze N x N(N - liczba dostępnych żółwi), w której są umieszczane aktualnie znalezione optymalne rozwiązania. Celem optymalizacji jest minimalizacja wagi tronu przy jak największej liczbie użytych do jego zbudowania żółwi.

1				
1	3			
1	3	6		
1	3	6	10	
1	3	6	10	15

Rysunek 1: Przykładowa tabela stworzona przez algorytm dla żółwi o wartościach (waga, wytrzymałość): (1,1) (2,3) (3,6) (4,10) (5,15). Widać, że z podanych żółwi możemy zbudować tron o wysokości 5 i wadze 15.

Każda kolejna iteracja algorytmu to kolejny wiersz tabeli. Do każdej kolejnej kolumny tabeli wpisywana jest wartość minimalna z porównania:

```
min(table[i][j], table[i - 1][j])
```

Jeśli żółw może udźwignąć wartość z poprzedniej iteracji, to w aktualne pole wybieramy:

```
min(table[i][j], table[i-1][j-1]+turtles[i].getWeight())
```

gdzie i, j to odpowiednio aktualny wiersz i kolumna, a turtles[i] to aktualnie rozpatrywany żółw.

```
table[0][1] = turtles[0].getWeight();
for (unsigned int i = 1; i < N; ++i)
{
  for (unsigned int j = 1; j <= i + 1; ++j)
  {
    table[i][j] = min(table[i][j], table[i - 1][j]);
```

```
    if (turtles[i].getCapacity() >= table[i - 1][j - 1])
    {
      table[i][j] = min(table[i][j],
                        table[i - 1][j - 1]+turtles[i].getWeight());
    }
  }
}

// looking for index of last element in last row(height of Throne)
for (unsigned int i = N; i >= 0; --i)
{
  if (table[N - 1][i] != numeric_limits<unsigned int>::max())
  {
    return i;
  }
}
```

3.2 Algorytm naiwny

Rozwiązanie polega na przejściu w pętli tyle razy ile jest żółwi, za każdym razem wybierając najlżejszego żółwia(jeśli żółwie mają taką samą wagę, to wybieramy żółwia z mniejszą wytrzymałością). Znalezionego żółwia następnie umieszczamy na stos pod warunkiem, że jest w stanie go unieść. Jeśli żółw, który jest w danej chwili na szczycie stosu może unieść więcej niż aktualnie znaleziony żółw i da radę unieść także nowego, to zostaje on na szczycie stosu, a pod niego wkłada się nowego żółwia.

```
// putting first turtle on stack
auto firstTurtle = min_element(turtles.begin(), turtles.end());

// if there are no turtles in vector, return 0
if (firstTurtle == turtles.end())
        return 0;

unsigned int stackWeight = firstTurtle->getWeight();
unsigned int stackHeight = 1;

Turtle lastTurtle = *firstTurtle;

// erase first turtle from vector
turtles.erase(firstTurtle);

for (int i = 0; i < turtles.size(); ++i)
{
```

```
  // looking for the lightest turtle
  auto it = min_element(turtles.begin(), turtles.end());

  // if lastTurtle in stack has got better capacity than turtle
  // that we found and new turtle can hold stack -
  // we can maybe swap them on the top of stack
  if (lastTurtle.getCapacity() > it->getCapacity()
    && it->getCapacity() >= stackWeight)
  {
    // if lastTurtle can hold the stack with new turtle
    if(lastTurtle.getStrength() > stackWeight + it->getWeight())
    {
      stackWeight += it->getWeight();
      ++stackHeight;

      // erasing current turtle from vector
      turtles.erase(it);
      continue;
    }
  }

  // if new turtle can hold current stack - push it on it
  if (it->getCapacity() >= stackWeight)
  {
    ++stackHeight;
    stackWeight += it->getWeight();
    lastTurtle = *it;
  }

  // erasing current turtle from vector
  turtles.erase(it);
}

return stackHeight;
```

3.3 Algorytm z presortowaniem

Algorytm bardzo przypomina przedstawione powyżej podejście naiwne. Na samym początku kolekcja żółwi jest sortowana według wagi. Następnie algorytm przechodzi po wszystkich żółwiach w pętli i próbuje je wrzucać na stos(sprawdzanie czy można wrzucić na stos jest analogiczne do algorytmu naiwnego).

```
sort(turtles.begin(), turtles.end());
```

```
unsigned int stackWeight = 0, stackHeight = 0;
auto lastTurtle = turtles.begin();
for (auto it = turtles.begin(); it != turtles.end(); ++it)
{
  if (lastTurtle->getCapacity() > it->getCapacity()
    && it->getCapacity() >= stackWeight)
  {
      if (lastTurtle->getStrength() > stackWeight + it->getWeight() )
      {
        stackWeight += it->getWeight();
        ++stackHeight;
        continue;
      }

    }
    if (it->getCapacity() >= stackWeight)
    {
      ++stackHeight;
      stackWeight += it->getWeight();
      lastTurtle = it;
    }
}
return stackHeight;
```