Задание 1

> Напишите MST на Deterministic Reservations с reserve и commit

Моя попытка реализовать парарллельного Краскалу 😀.

Надеюсь с комментами в этот раз будет попроще смотреть код :)

```
type Edge struct {
  u, v   int
  weight int64
  idx    int // позиция ребра в сортированном массиве ребер (приоритет)
}

func ParallelKruskal(edges []Edge, n int) []Edge {
  // как и в обычном краскале сначала сортируем ребра
  edges = ParallelSort(edges, func(e1, e2 Edge) bool {
     return e1.weight < e2.weight
   })

   // union-find/disjoint-set реализация будет ниже
   uf := UF{parent: make([]int32, n)}
   // шаги алгоритма с reserve/commit
   steps := make([]Step, n)

   // инициализируем uf и steps
   ParallelFor(0, n, func(idx int) {
     uf.parent[idx] = -1
     steps[idx] = Step{priority: -1}
   })

   mst := make([]Edge, n) // результат
   var mstPos int64 // счётчик записей в mst


   unprocessed := edges
   for len(unprocessed) > 0 {
     // delta размер блока который обрабатываем параллельно
     prefix := unprocessed[:delta]

     // Reserve стадия
     ParallelFor(0, delta, func(idx int) {
       edge := prefix[idx]
       // если концы уже в одной компоненте скипаем ребро
       if uf.Find(edge.u) == uf.Find(edge.v) {
         edge.idx = -1
         return
       }
       // пытаемся зарезервировать обе вершины
       // резервируем независимо, в коммите проверим, удалось ли хотя бы одной.
       ok1 := steps[edge.u].reserve(idx)
       ok2 := steps[edge.v].reserve(idx)
       // удалось резервировать, значит годен для коммита
       if ok1 || ok2 {
         edge.idx = idx
```

```
48        } else { // резервировать не удалось, пропускаем
49          edge.idx = -1
50        }
51      })
52
53      // Commit стадия
54      ParallelFor(0, delta, func(i int) {
55        edge := prefix[i]
56        if edge.idx < 0 {
57          return // не зарезервировали
58        }
59        // проверяем действительно ли зарезервировали данное ребро
60        uOk := steps[edge.u].check(idx)
61        vOk := steps[edge.v].check(idx)
62        // обе вершины перезаписались с более приоритетным ребром
63        if !(uOk || vOk) {
64          edge.idx = -1
65          return
66        }
67        // записываем ребре в mst, атомарно увеличивая счетчик
68        pos := atomic.AddInt64(&mstPos, 1) - 1
69        mst[pos] = edge
70        // объединяем компоненты в union-find
71        uf.Union(edge.u, edge.v)
72        // помечаем ребро как завершённое
73        edge.idx = -2
74      })
75
76      // сдвигаем неразрешённые ребра в начало массива
77      // для каждой позиции i создаём flag=true — ребро не было обработано
78      flags := make([]bool, delta)
79      ParallelFor(0, delta, func(idx int) {
80        flags[idx] = unprocessed[idx].idx == -1
81      })
82      // получаем индексы позиций в новом массиве
83      pos := Scan(flags)
84      // обновленный массив новых ребер
85      newUnprocessed := make([]Edge, delta)
86
87      ParallelFor(0, delta, func(idx int) {
88        if flags[idx] {
89          newUnprocessed[pos[idx]] = unprocessed[idx]
90        }
91        // если ребро уже обработано unprocessed[idx].idx==-2 — просто
отбрасываем
92      })
93
94      unprocessed = newUnprocessed
95      // очищаем резервации
96      ParallelFor(0, n, func(idx int) {
97        steps[idx].priority = -1
98      })
99    }
100    return mst[:mstPos] // отрезаем лишние нули, если они есть
101  }
```

Не знаю надо ли было писать реализации step и uf, но вот *по идее* их потокобезопасные имплементации

```
type Step struct {
  // храним только приоритет последней (успешной) резервации.
  priority int
}

// pwrite — приоритетная запись записывает v, если v < текущее значение.
func (rs *Step) pwrite(v int) {
  for {
    cur := atomic.LoadInt32(&rs.priority)
     if v >= cur { // уже лучший приоритет записан
       return
     }
     if atomic.CompareAndSwapInt32(&rs.priority, cur, v) {
       return
     }
   }
}

// reserve — попытка зарезервировать вершину с приоритетом p
func (rs *Step) reserve(p int) bool {
  // записываем только если p лучше (меньше) текущего.
  rs.pwrite(p)
  // после записи проверяем, действительно ли p стал новым приоритетом
  cur := atomic.LoadInt32(&rs.priority)
  return rs.check(p)
}

// check — проверка, зарезервировано ли p
func (rs *Step) check(p int) bool {
  return atomic.LoadInt32(&rs.priority) == p
}
```

Реализация системы непересекающихся множеств

```
type UF struct {
  parent []int
}

// Find с компрессией пути
func (uf *UF) Find(x int) int {
  for {
    p := atomic.LoadInt32(&uf.parent[x])
    if p < 0 {
       return x
     }
     grand := atomic.LoadInt32(&uf.parent[int(p)])
     atomic.CompareAndSwapInt32(&uf.parent[x], p, grand)
     x = p
   }
}

```

```
18 // Union объединяет два множества
19 func (uf *UF) Union(a, b int) {
20   for {
21     ra := uf.Find(a)
22     rb := uf.Find(b)
23     if ra == rb {
24       return
25     }
26     // попытка сделать ra ребёнком rb
27     if atomic.CompareAndSwapInt32(&uf.parent[ra], -1, int32(rb)) {
28       return
29     }
30     // если не получилось - кто-то уже изменил структуру, повторяем цикл
31   }
32 }
```