

Задание 1

Напишите MST на Deterministic Reservations с reserve и commit

Моя попытка реализовать параллельного Краскала 😊.

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

```
1 type Edge struct {
2     u, v    int
3     weight  int64
4     idx     int // позиция ребра в сортированном массиве ребер (приоритет)
5 }
6
7 func ParallelKruskal(edges []Edge, n int) []Edge {
8     // как и в обычном краскале сначала сортируем ребра
9     edges = ParallelSort(edges, func(e1, e2 Edge) bool {
10         return e1.weight < e2.weight
11     })
12
13     // union-find/disjoint-set реализация будет ниже
14     uf := UF{parent: make([]int32, n)}
15     // шаги алгоритма с reserve/commit
16     steps := make([]Step, n)
17
18     // инициализируем uf и steps
19     ParallelFor(0, n, func(idx int) {
20         uf.parent[idx] = -1
21         steps[idx] = Step{priority: -1}
22     })
23
24     mst := make([]Edge, n) // результат
25     var mstPos int64 // счётчик записей в mst
26
27
28     unprocessed := edges
29     for len(unprocessed) > 0 {
30         // delta размер блока который обрабатываем параллельно
31         prefix := unprocessed[:delta]
32
33         // Reserve стадия
34         ParallelFor(0, delta, func(idx int) {
35             edge := prefix[idx]
36             // если концы уже в одной компоненте пропускаем ребро
37             if uf.Find(edge.u) == uf.Find(edge.v) {
38                 edge.idx = -1
39                 return
40             }
41             // пытаемся зарезервировать обе вершины
42             // резервируем независимо, в коммите проверим, удалось ли хотя бы одной.
43             ok1 := steps[edge.u].reserve(idx)
44             ok2 := steps[edge.v].reserve(idx)
45             // удалось резервировать, значит годен для коммита
46             if ok1 || ok2 {
47                 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

```

1  type Step struct {
2      // храним только приоритет последней (успешной) резервации.
3      priority int
4  }
5
6  // pwrite – приоритетная запись записывает v, если v < текущее значение.
7  func (rs *Step) pwrite(v int) {
8      for {
9          cur := atomic.LoadInt32(&rs.priority)
10         if v >= cur { // уже лучший приоритет записан
11             return
12         }
13         if atomic.CompareAndSwapInt32(&rs.priority, cur, v) {
14             return
15         }
16     }
17 }
18
19 // reserve – попытка зарезервировать вершину с приоритетом p
20 func (rs *Step) reserve(p int) bool {
21     // записываем только если p лучше (меньше) текущего.
22     rs.pwrite(p)
23     // после записи проверяем, действительно ли p стал новым приоритетом
24     cur := atomic.LoadInt32(&rs.priority)
25     return rs.check(p)
26 }
27
28 // check – проверка, зарезервировано ли p
29 func (rs *Step) check(p int) bool {
30     return atomic.LoadInt32(&rs.priority) == p
31 }

```

Реализация системы непересекающихся множеств

```

1  type UF struct {
2      parent []int
3  }
4
5  // Find с компрессией пути
6  func (uf *UF) Find(x int) int {
7      for {
8          p := atomic.LoadInt32(&uf.parent[x])
9          if p < 0 {
10             return x
11         }
12         grand := atomic.LoadInt32(&uf.parent[int(p)])
13         atomic.CompareAndSwapInt32(&uf.parent[x], p, grand)
14         x = p
15     }
16 }
17

```

```
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 }
```