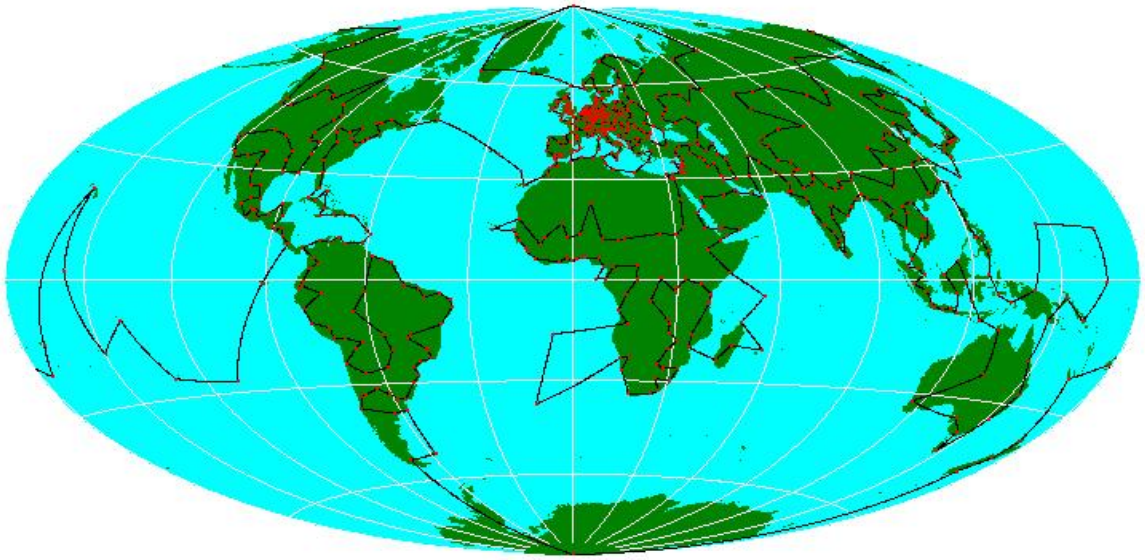
# Algoritmid ja andmestruktuurid

- Kombinatoorsed optimiseerimisülesanded
- Tagasivõtmisega (*backtracking*) algoritmid
- Hargne ja kärbi (*branch and bound*) algoritmid

# Rändkaupmehe (TSP) ülesanne

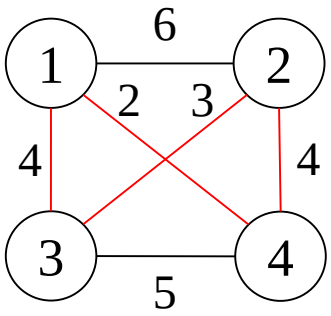
- Leida etteantud graafist lühim tee mis:
  - Algab ja lõpeb samas tipus (kohas)
  - Läbib igat tippu (kohta) täpselt ühe korra
- ATSP – asümmeetriline ülesanne, teepikkused AB ja BA ei pea olema võrdsed
- Eukleidiline TSP
  $AB + BC \geq AC$



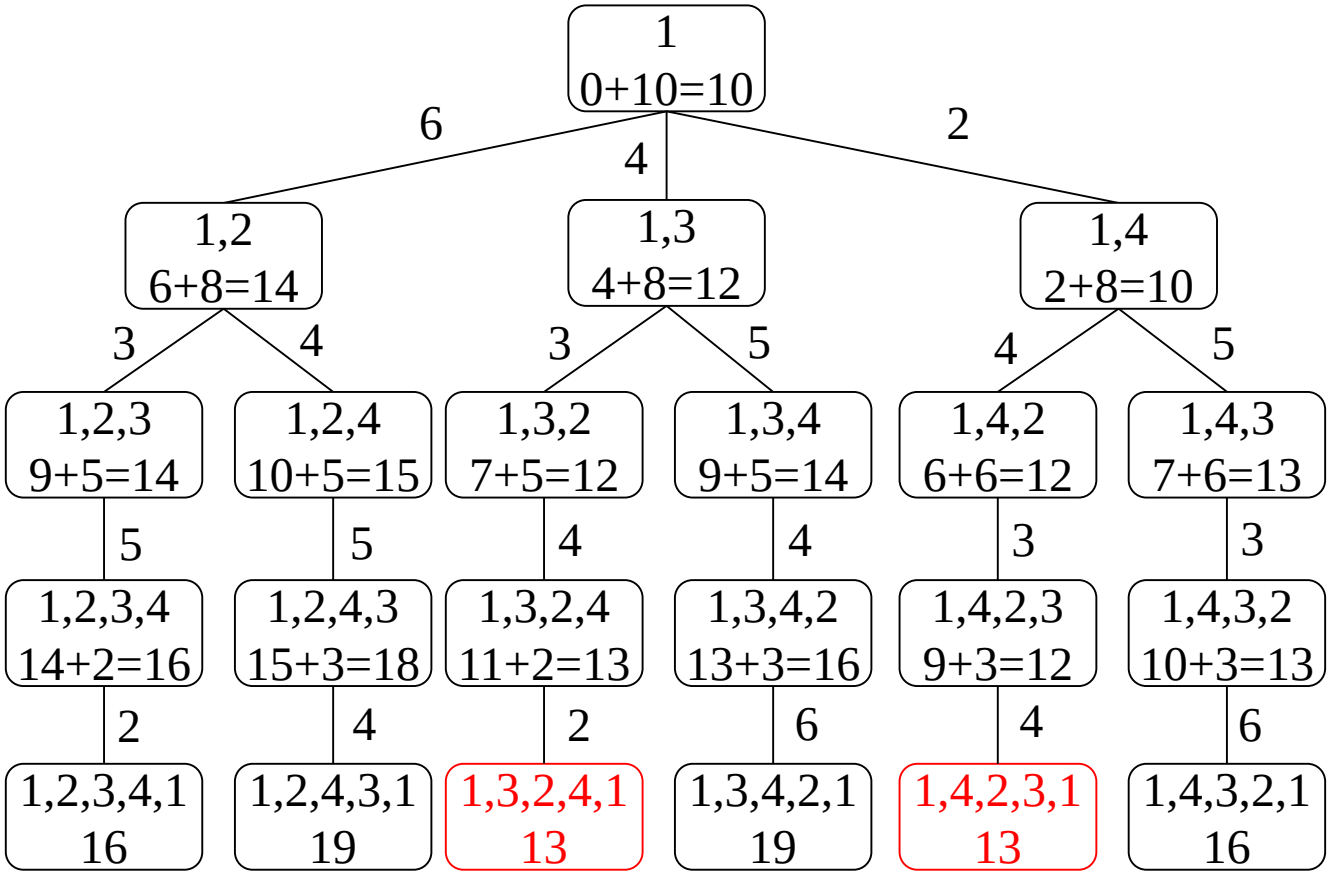
# Kärpimine - *bound*()

- hinnang väljendab teepikkust, millest lühemat teed pole antud seisust enam võimalik saavutada
- läbitud teepikkuse ja läbimata linnade läbimise miinimumhinnangu summa
  - igast linnast väljuvate minimaalsete teede järgi
    - arvutatakse igast linnast väljuv minimaalne tee
    - hinnang on summa juba läbitud teest ja minimaalsetest väljuvatest teedest linnade kohta, kust ei ole veel välja sõidetud (sh ka teekonna viimane liińn)
    - minimaalsed teed järgnevas näites [2 3 2 4] ja hinnang kui pole läbitud veel ühtegi linna on nende summa 11
  - korrigeeritud minimaalsete teede järgi
    - linnast väljuv minimaalse tee pikkus leitakse ainult läbimata linnade alusel (näide järgmisel slaidil)
  - hinnagu algoritm tuleb implementeerida efektiivselt

# Rändkaupmehe näide (täielik puu)



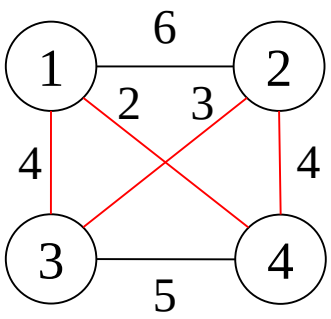

| | 1 | 2 | 3 | 4 | min |
|---|---|---|---|---|---|
| 1 | 0 | 6 | 4 | 2 | 2 |
| 2 | 6 | 0 | 3 | 4 | 3 |
| 3 | 4 | 3 | 0 | 5 | 3 |
| 4 | 2 | 4 | 5 | 0 | 2 |
| **kokku** | | | | | **10** |



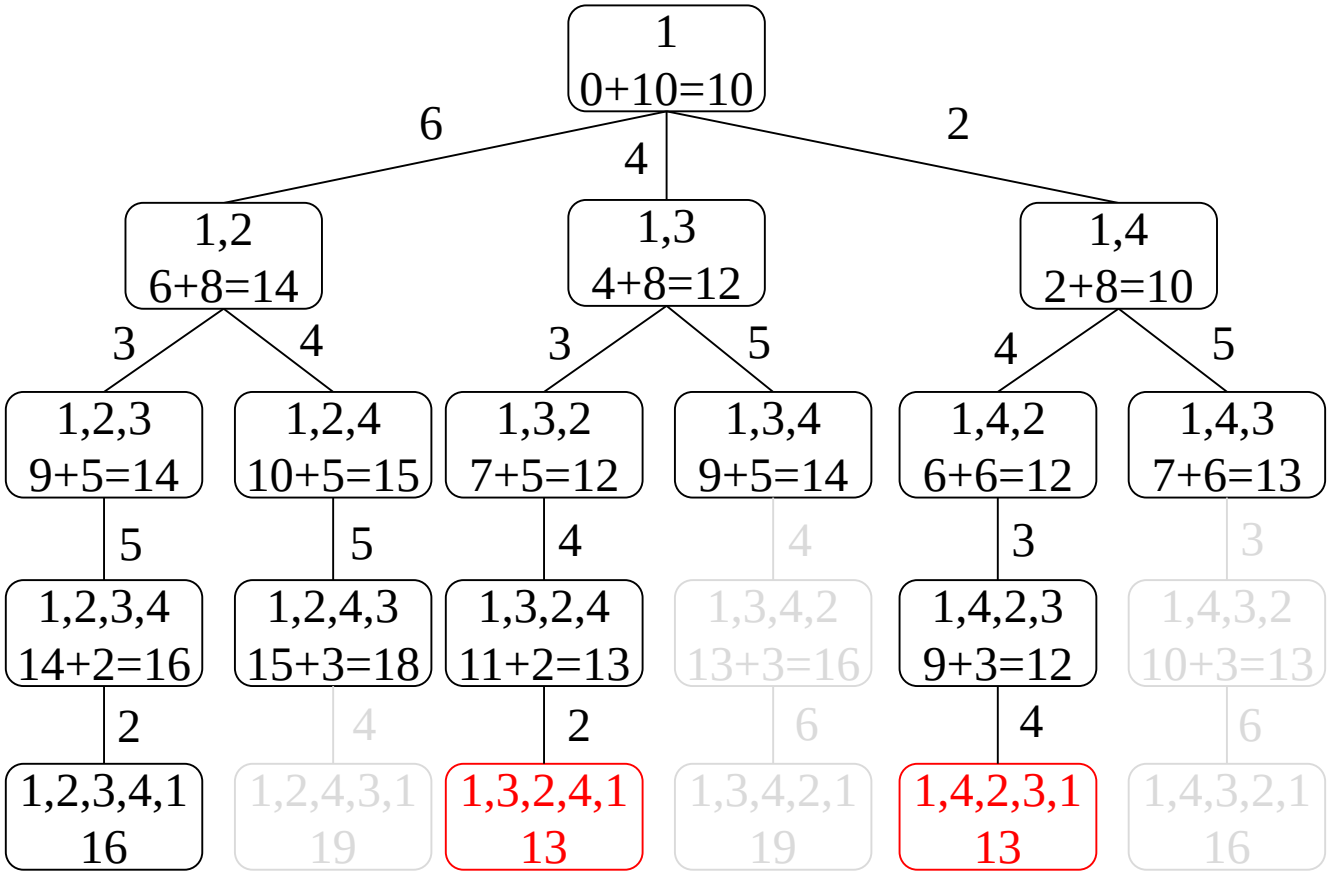
# Rändkaupmehe tagasivõtmisega algoritm

```
int läbitud[N];        // läbitud linnade numbrid läbimise järjekorras
int läbimata[N];       // läbimata[i]=0 kui linn i läbimata
int parim_tee[N] = ahne_hinnang();
int parim_teepikkus = teepikkus(parim_tee);
tsp(int läbitud_arv) {
   if(läbitud_arv == N){
      teepikkus = teepikkus(läbitud);        // koos teega algusse
      if(teepikkus < parim_teepikkus){
           parim_teepikkus = teepikkus;
           parim_tee = läbitud;              // tuleb teha koopia
      }
   elif(bound(läbitud, läbimata) < parim_teepikkus)
      for i in läbimata {
           läbitud[läbitud_arv+1] = i;
           läbimata[i] = 1;
           tsp(läbitud_arv+1);
   }
}
```

# Rändkaupmehe näide sügavuti otsinguga

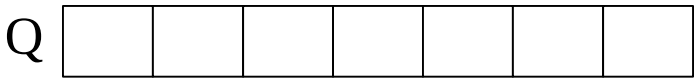


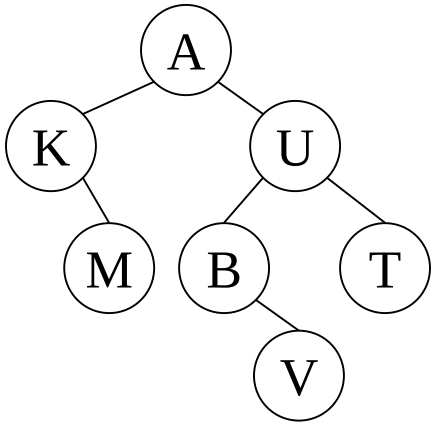

| | 1 | 2 | 3 | 4 | min |
|---|---|---|---|---|---|
| 1 | 0 | 6 | 4 | 2 | 2 |
| 2 | 6 | 0 | 3 | 4 | 3 |
| 3 | 4 | 3 | 0 | 5 | 3 |
| 4 | 2 | 4 | 5 | 0 | 2 |
| kokku | | | | | 10 |



# Laiuti puu-otsing

```
int breadth_first_search (tree T){
  queue_of_node Q;
  node u, v,
  initialize(Q);  // Initialize Q to be empty.
  v = root of T;
  visit v;        // Visit root.
  enqueue(Q, v);
  while (! empty(Q)){
   dequeue(Q, v);
   for (each child u of v) {  // Visit each child.
      visit u;
      enqueue(Q, u);
   }
  }
}
```



# H&K laiuti otsinguga

```
int breadth_first_branch_and_bound (state_space_tree T){
  queue_of_node Q;
  node u, v,
  initialize(Q); // Initialize Q to be empty.
  v = root of T; // Visit root.
  enqueue(Q, v);
  best = ∞
  while (! empty(Q)){
   dequeue(Q, v);
   for (each child u of v) { // Visit each child.
      if (u is a solution and
          value(u) is better than best) // solution found
         best = value(u);
      if (bound(u) is better than best) // promising child
         enqueue(Q, u);
   }
  }
  return best
}
```

**bound - parima võimaliku tulemuse hinnang**

**kas tipp on lootust-andev - kärpimine**

# Sügavuti otsing ilma rekursioonita

```
int depth_first_branch_and_bound (state_space_tree T){
  stack_of_node S;
  node u, v,
  initialize(S); // Initialize Q to be empty.
  v = root of T; // Visit root.
  push(S, v);
  best = ∞
  while (! empty(S)){
   pop(S, v);
   for (each child u of v) { // Visit each child.
      if (u is a solution and
          value(u) is better than best) // solution found
         best = value(u);
      if (bound(u) is better than best) // promising child
         push(S, u);
   }
  }
  return best
}
```

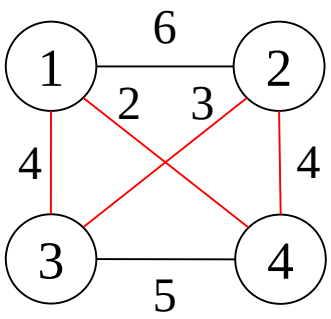
# H&K parim-enne otsinguga

```
int best_first_branch_and_bound (state_space_tree T){
  priority_queue_of_node PQ;
  node u, v,
  initialize(PQ); // Initialize Q to be empty.
  v = root of T;  // Visit root.
  enqueue(PQ, v);
  best = ∞
  while (! empty (PQ)){
   dequeue(PQ, v);
   if (bound(v) is worse than best) break;
   for (each child u of v) { // Visit each child.
      if (u is a solution and
          value(u) is better than best) // solution found
         best = value(u);
      if (bound(u) is better than best) // promising child
         enqueue(PQ, u);
   }
  }
  return best }
```

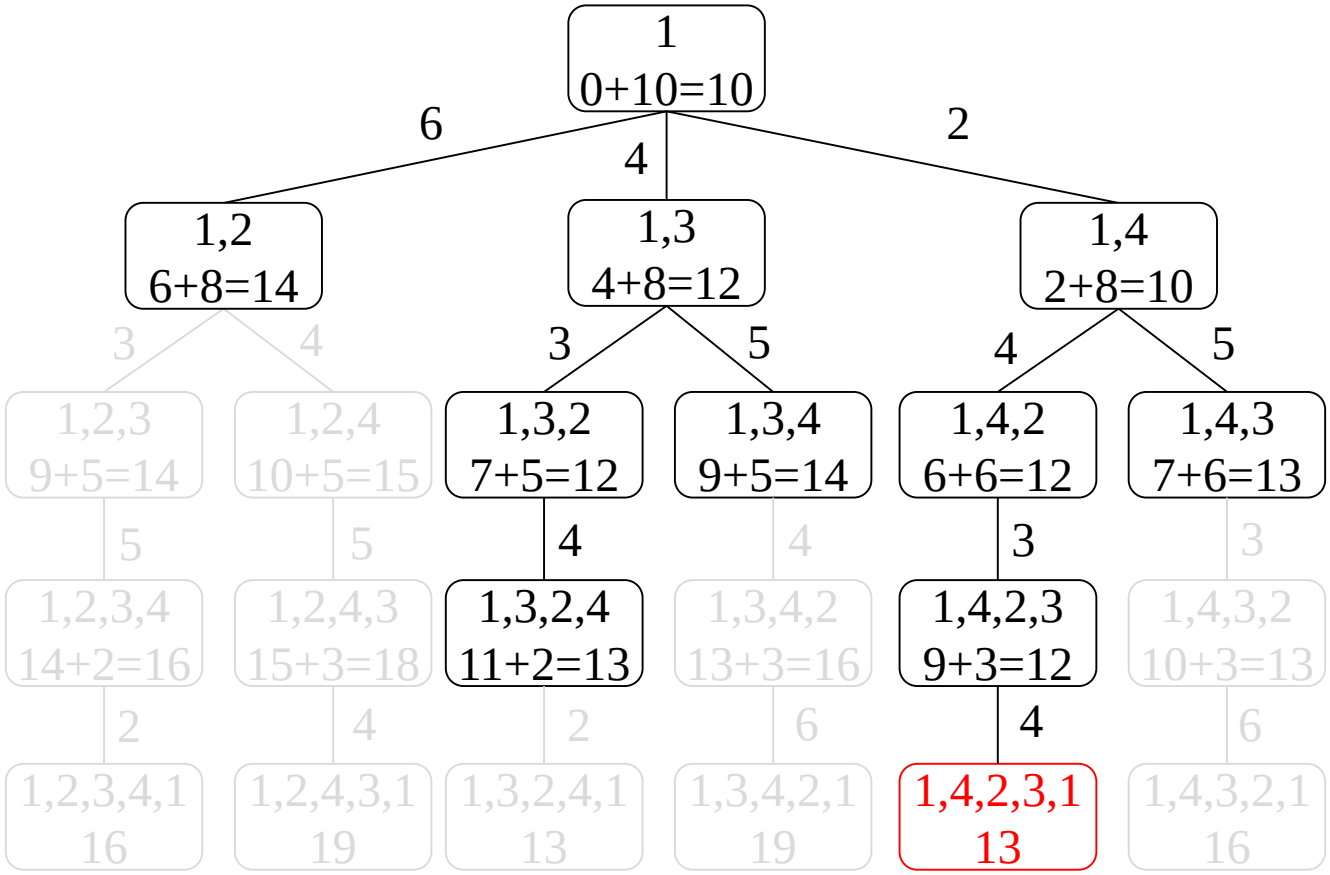
**Kontrolli uuesti, et tipp oleks ikka veel lootustandev**

**kas tipp on lootust-andev - kärpimine**

# Rändkaupmehe näide
## parim enne otsinguga




| | 1 | 2 | 3 | 4 | min |
|---|---|---|---|---|---|
| 1 | 0 | 6 | 4 | 2 | 2 |
| 2 | 6 | 0 | 3 | 4 | 3 |
| 3 | 4 | 3 | 0 | 5 | 3 |
| 4 | 2 | 4 | 5 | 0 | 2 |
| kokku | | | | | 10 |



# H&K parim-enne otsinguga

```
int best_first_branch_and_bound (state_space_tree T){
  priority_queue_of_node PQ;
  node u, v,
  initialize(PQ); // Initialize Q to be empty.
  v = root of T;  // Visit root.
  enqueue(PQ, v);
  best = greedySolution(T);
  while (! empty (PQ)){
   dequeue(PQ, v);
   if (bound(v) is worse than best) break;
   for (each child u of v) { // Visit each child.
      if (u is a solution and
          value(u) is better than best) // solution found
         best = value(u);
      if (bound(u) is better than best) // promising child
         enqueue(PQ, u);
   }
  }
  return best }
```

**Esmane parima lahenduse hinnang kiire ahne algoritmiga**

# Erinevad otsimisstrateegiad

- Sügavuti
  - otsimisjärjekorda esitav andmestruktuur: *stack* (FILO)
  - sobib paremini, kui lahendused on puus sügaval
- Laiuti
  - otsimisjärjekorda esitav andmestruktuur: *queue* (FIFO)
  - sobib paremini, kui lahendused ei ole puus sügaval
  - vajab tavaliselt rohkem mälu,
    tihti on mäluvajadus väga suur
- Parim esimesena
  - otsimisjärjekorda esitav andmestruktuur: *priority queue*
  - sobib parima lahenduse leimiseks
  - efektiivsem kärpimine
  - mäluvajadus suur