Implementations of Kruskal algorithm

Combinatorial optimization

Giovanni Righini

UNIVERSITÀ DEGLI STUDI DI MILANO

Median-of-medians

Median selection problem: find the median in a given set (unsorted list L) of n values.

Median-of-medians algorithm.

- Split L in subsets of 5 elements each: $S_i \ \forall i = 1, \ldots, n/5$.
- Find the median x_i for each S_i.
- Find the median M of the x_i values.
- Partition L into three lists:
  - L_1 with values smaller than M;
  - L_2 with values equal to M;
  - L_3 with values larger than M.
- Discard L_1 or L_3 depending on the sought element k.
- Recursively call the procedure on the remaning part of L.

```
// Procedure Select(L, u, v, k).
if (v − u + 1 ≤ 10) then
   Sort(L, u, v)
   return  L[u − 1 + k]
else
   s ← ⌈(v − u + 1)/5⌉
   for i = 1, . . . , s do
      S(i) ← {u + 5(i − 1), . . . , min{u + 5i − 1, v}}
      x(i) ← Median − of − 5(S(i))
   M ← Median({x(1), . . . , x(s)})
   Partition(L, M, L1, L2, L3)
   if (k ≤ |L1|) then
      Select(L1, u, u + |L1| − 1, k)
   else
      if (k > |L1| + |L2|) then
         Select(L3, v − |L3| + 1, v, k − |L1| − |L2|)
      else
         return  M
```

Complexity

Median of 5: at most 6 pair-wise comparisons.

Median of a set S of s elements: call Select$(S, \lceil s/2 \rceil)$.

Max. size s of S is $n/5$.

By definition, at least half of the medians x_i (i.e. at least $n/10$) are $\leq M$ ($\geq M$).

For each of them, 3 values in S_i are $\leq M$ ($\geq M$).

Then, $|L_1| \geq 3n/10$ ($|L_3| \geq 3n/10$).

Hence $|L_3| \leq 7n/10$ ($|L_1| \leq 7n/10$).

Complexity

$T(n)$: n. of comparisons to find the median of n elements.

Finding all medians x_i takes at most $6n/5$ comparisons.
Finding M takes $T(n/5)$.
Partitioning L takes $n-1$ comparisons.
Finding the median of the remaining list takes at most $T(7n/10)$.

Therefore

$$T(n) \leq \frac{6n}{5} + T\left(\frac{n}{5}\right) + n + T\left(\frac{7n}{10}\right).$$

Hence

$$T(n) \leq \frac{11n}{5} \sum_{k=0}^{\infty} \left(\frac{1}{5} + \frac{7}{10}\right)^k.$$

Since $\frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$, the geometric series converges to a constant.

Therefore $T(n)$ is $O(n)$ (it can be reduced to 2.95 n).

Selection problem

Selection problem: find the kth smallest value in a given set (unsorted list N) of n values.

Select algorithm (Blum et al., 1973).

- Split N in n/c subsets of c elements each and sort each of them.
- Find the dth smallest element (the representative) in each subset.
- Find the bth smallest element, M, in the set of representatives.
- Partition the n/c subsets into three parts:
  - Left: $b-1$ subsets with representatives $\leq M$;
  - Center: the subset containing M;
  - Right: $n/c-b$ subsets with representatives $\geq M$.
- L: entries in the left and center columns on rows from 1 to d.
- G: entries in the right and center columns on rows from d to c.
- Compare each element with M.
- Discard either L or G and repeat, until $k-1$ smallest entries or $n-k$ largest entries have been discarded.

Selection problem

A

G

m

← T = {d-th smallest element from each column}

L

B

b-1 columns with d-th smallest < m

n/c - b columns with d-th smallest > m

Complexity

$h(c)$: n. of comparisons needed to sort c elements.

Ford and Johnson (1959): $h(c) = \sum_{j=1}^{c} \lceil \log_2 \left(\frac{3}{4}j\right) \rceil$.

$T(n)$: n. of comparisons needed to find the kth smallest element in a set of n.

- Sorting $\frac{n}{c}$ subsets of c elements each: $\frac{n}{c}h(c)$.
- Finding M among the representatives: $T\left(\frac{n}{c}\right)$.
- Comparing each element with M: n.
- Solving the remaining problem: $T(n - \min\{|L|, |G|\})$.

$$T(n) = \frac{n}{c}h(c) + T(n/c) + n + T(n - \min\{|L|, |G|\}).$$

Complexity

$$|L| = bd \quad |G| = \left(\frac{n}{c} - b + 1\right)(c - d + 1)$$

By selecting $b = n/42$, $c = 21$, $d = 11$:

$$|L| = \frac{11}{42}n \quad |G| = \left(\frac{n}{21} - \frac{n}{42} + 1\right) 11 = \frac{11}{42}n + 11.$$

$$h(21) = 66.$$

Therefore

$$T(n) \leq \frac{66}{21}n + T\left(\frac{n}{21}\right) + n + T\left(\frac{31}{42}n\right).$$

Base of the induction: for small enough n, $h(n)$ is linear (e.g. for $n < 10^5$, $h(c) < 19n$).

$$T(n) \leq \frac{29}{7}n \sum_{k=0}^{\infty} \left(\frac{1}{21} + \frac{31}{42}\right)^k = \frac{29}{7}n \frac{1}{1 - \frac{11}{14}} = \frac{29}{7}n\frac{14}{3} = \frac{58}{3}n.$$

Hence $T(n)$ is $O(n)$.

Partial sorting problem

Partial sorting problem: find and sort the kth smallest values in a given set N of m values.

SelectionSort algorithm.

- For $i = 1, \ldots, k$,
  - scan the list from $N[i]$ and find the smallest value;
  - swap the smallest element with $N[i]$.

Time complexity: $O(km)$. Space complexity: $O(m)$.

Using a (big) binary heap (C++ STL's partial_sort function):

- Build a binary heap with the m elements.
- For k times, extract the root and rearrange the heap.

Complexity: $O(m + k \log m)$. Space complexity: $O(m)$.

Partial sorting problem

Using a (small) binary heap:

- Build a binary max-heap with the first k elements of N.
- For $i = k+1, \ldots, m$ times, if $N[i]$ is smaller than the current root of the heap, then replace the root and update the heap.
- Sort the elements of the heap.

Complexity: $O((k+m)\log k)$. Space complexity: $O(k)$.
Valid alternative if $k << m$ or N is processed on-line.

QuickSelSort algorithm, combining Select and QuickSort.

- find the kth smallest element in N with Select (the pivot);
- scan the list and retain the elements smaller than the pivot (if needed);
- sort the selected elements.

Complexity: $O(m + k\log k)$. Space complexity: $O(m)$.

Partial QuickSort (Martinez, 2004)

Input for each recursive call of PartialQuickSort(N, i, j, k):

- an array N;
- an interval $[i, \ldots, j]$, with $i \leq j$;
- a number k of smallest elements to sort, with $i \leq k$.

Base of the recursion: $i = j$.

Recursive step:

- select a pivot in position $p \in [i, \ldots, j]$;
- partition $[i, \ldots, j]$ as in QuickSort; let p' be the final position of the pivot;
- recursively call PartialQuickSort$(N, i, p' - 1, k)$;
- if $k > p'$, then recursively call PartialQuickSort$(N, p' + 1, j, k)$.

Partial QuickSort

```
// Procedure PartialQuickSort(N, i, j, k)
if (i < j) then
   p ← Pivot(N, i, j)
   p' ← Partition(N, i, j, p)
   PartialQuickSort(N, i, p' − 1, k)
   if (p' < k) then
      PartialQuickSort(N, p' + 1, j, k)
```

Incremental sorting problem (Paredes and Navarro, 2006)

Incremental sorting problem: on-line version of the Partial sorting problem (k is not given).

Given set N of m numbers, output the elements of N from the smallest one to the largest one, so that the process can be stopped after k elements have been output, for any k (not given in input).

Iterate Select.

- For $i = 1, \ldots, k$, select the smallest unselected element in N.

Time complexity: $O(km)$. Space complexity: $O(m)$.

Using a (big) binary heap:

- Build a binary heap with the m elements.
- Iteratively, extract the root and rearrange the heap.

Complexity: $O(m + k \log m)$. Space complexity: $O(m)$.

These naive methods are dominated by PartialQuickSort.

Incremental Quick Select

Algorithm IncrementalQuickSelect (Paredes and Navarro, 2006).

At each iteration, the algorithm finds the smallest element among those not yet selected.

The algorithm exploits the same recursion of QuickSort, but it can stop for each value of k.

As in iterative implementations of QuickSort, it keeps a stack of pivots already set at their correct positions in previous iterations.

Incremental Quick Select

```
[(49) 81 74 12 58 92 86 33 67 18 25 37 51 63 29 41]      s = {16}

[(33) 41 29 12 37 25 18] 49 [67 86 92 58 51 63 74 81]     s = {16, 7}

[(25) 18 29 12] 33 [37 41]                                s = {16, 7, 4}

[(12) 18] 25 [29]                                         s = {16, 7, 4, 2}

12 [18]                                                   s = {16, 7, 4, 2, 0}
```

Figure: Positions are numbered from 0 to 15.

Incremental Quick Select

1: // Procedure IQS(N, S, k)
2: **if** Top(S) $= k$ **then**
3: Pop(S)
4: **return** $N[k]$
5: $p \leftarrow$ Pivot(N, k, Top(S) $- 1$)
6: $p' \leftarrow$ Partition(N, k, Top(S) $- 1, p$)
7: Push(S, p')
8: **return** IQS(N, S, k)

Worst-case complexity: $O(m^2)$.
Average-case complexity: $O(m + k \log k)$.

Comparing MST algorithms

Computational complexity:

- Fibonacci heaps (Fredman and Tarjan, 1987, Gabow et al. 1986) for Prim algorithm: $O(m + n \log n)$.
- Union-Find for Kruskal algorithm: $O(m \log n)$ for sorting and $O(m + n \log n)$ for the MST.

A randomized algorithm by Karger (1993) runs in linear expected time.

In practice:

- Kruskal: effective when the edge weights are drawn from a small range of integers or when the graph is sparse;
- Prim: effective in all the other cases, especially when the graph is dense.

Experimental comparisons (Moret and Shapiro, 1991, Bazlamaçcı and Hindi, 2001):

- Prim algorithm implemented with a binary heap ($O(m \log n)$) is faster than Kruskal.
- Both algorithms are faster than the more recent ones.

Implementing Kruskal algorithm

Naive implementation: first the edge list is sorted; then it is scanned to compute a MST.

“On demand sorting”: keep the edges partially sorted in a binary heap, constructed by QuickSort when needed, i.e. when all previously sorted edges have been examined.

QuickSortKruskal (QSK): no binary heap is constructed. Kruskal algorithm is executed from within QuickSort: the two steps of sorting and selecting the edges of the MST proceed in parallel.

Quick Sort Kruskal

QSK: the recursive procedure is run on the leftmost part first and on the rightmost part later, for each interval to be sorted.

Base of the recursion: interval with a single element.

Every time QuickSort completes the sorting of the leftmost part of an interval, the pivot element is considered for insertion in the MST as in Kruskal algorithm, before sorting the rightmost part of the interval.

- edges are sorted only when needed;
- no additional data-structure and initialization.

QSK is especially effective when the input graph is given as an edge list.

Quick Sort Kruskal

```
// QuickSortKruskal. IN: ℰ. OUT: T.
InitUnionFind
T ← ∅
count ← 0
QuickSortK(1, m)
```

$\mathcal{E}$: input list of edges.
T: list of selected edges.
count: cardinality of T.

```
// Procedure QuickSortK(p, q)
e+ ← q
if p < q then
   e- ← p
   while e- ≤ e+ do
      while E[e+].cost > E[p].cost do
         e+ ← e+ - 1
      while (e- ≤ e+) ∧ (E[e-].cost ≤ E[p].cost) do
         e- ← e- + 1
      if e- < e+ then
         Swap(e-, e+); e- ← e- + 1; e+ ← e+ - 1
   Swap(p, e+)
if e+ > p then
   QuickSortK(p, e+ - 1)
if count < n - 1 then
   TestEdge(e+)
if (count < n - 1) ∧ (e+ < q) then
   QuickSortK(e+ + 1, q)
```

Quick Sort Kruskal

The pivot element in position e^+ is tested by *TestEdge*(e^+)

- after sorting the edges with cost smaller than the pivot ($QuickSortK(p, e^+ - 1)$)
- before sorting the edges with cost larger than the pivot ($QuickSortK(e^+ + 1, q)$).

// Procedure TestEdge(e)
$i \leftarrow \mathcal{E}[e].i$
$j \leftarrow \mathcal{E}[e].j$
if $head[i] \neq head[j]$ **then**
    $count \leftarrow count + 1$
    $T \leftarrow T \cup \{[i, j]\}$
    if $card[head[i]] \geq card[head[j]]$ **then**
        $Append(head[j], head[i])$
    else
        $Append(head[i], head[j])$

The Star Quick Sort Kruskal algorithm

Input graph: set of stars.

Naive solution 1: produce a sorted list of edges by merging the subsets and then sorting the resulting list or heap.

Naive solution 2: produce a sorted list of edges by separately sorting each star and then merging them into a unique sorted list or heap.

In StarQuickSortKruskal (SQSK) stars are not merged.

A sorted list is produced on demand by QuickSort from the edges of each star; the first not-yet-examined edge in each list is the candidate edge for its vertex.

All candidate edges are partially sorted in a binary heap.

Iteratively:

- the best candidate e is selected;
- QuickSort is re-activated to find the next candidate for the endpoints of e;
- the heap is rearranged.

The Star Quick Sort Kruskal algorithm

Main idea: save a potentially large fraction of the computing time that would be spent to (partially) sort each vertex star in a binary heap or a in sorted list, since only a very small fraction of the edges in each star is likely to be considered by Kruskal algorithm.

The candidate edge in each star is the last edge of the sorted part of the star; all the edges with a cost larger than the candidate remain unsorted.

Pro: the sorting step does not work on a (typically large) set of edges, but on several (much smaller) subsets.

Drawback: the selection of each edge requires some non-trivial steps.

Since sorting requires super-linear computing time, it is intuitively convenient to (partially) sort n subsets of cardinality m/n rather than a unique set of cardinality m, especially when the input is given as a set of vertex stars.

SQSK: iterative Quick Sort

Since in SQSK QuickSort must be executed step-by-step only on-demand, an iterative implementation is needed.

A step of QuickSort is executed by QSstep(v) on $Star(v)$ every time v is the endpoint of the selected edge.

The effect of QSstep is to define at least one more sorted element in $Star(v)$.

$Stack(v)$: a stack associated with the star of each vertex v.

The stack contains the intervals in which the star has been partitioned by QuickSort.

Every time the current interval $[p, \ldots, q]$ is divided into a left interval $[p, \ldots, k-1]$ and a right interval $[k+1, \ldots, q]$ by a pivot element k,

- the right interval $[k+1, \ldots, q]$ is put into the stack,
- the pivot element k is put in its final position between the two intervals,
- the left interval $[p, \ldots, k-1]$ becomes the new current range.

SQSK: iterative QuickSort

$h(v)$: position of last sorted element in $Star(v)$.

If $h(v)+1$ coincides with the leftmost element of the interval on top of $Stack(v)$, then the new candidate edge is already available: increase $h(v)$ and stop.

Otherwise:

- extract $[p, q]$ to be sorted from $Stack(v)$;
- select a pivot element k in $[p, q]$;
- temporarily put the pivot in the first position;
- separate all elements smaller than or equal to the pivot (left) from the others (right);
- put the pivot back between the two sub-intervals;
- store the right interval $[k+1, q]$ in $Stack(v)$, unless its cardinality is less than 2;
- repeat on the left interval $[p, k-1]$, until its cardinality is less than 2.

After this step at least one more edge has been sorted and $h(v)$ can be increased by 1, indicating the next candidate in $Star(v)$.

// Procedure QSstep(v)
if $(Stack(v) \neq nil) \wedge (h(v) + 1 = \text{Top}(v).p)$ **then**
  $[p, q] \leftarrow \text{Pop}(v)$
  while $p < q$ **do**
    $k \leftarrow \text{Pivot}(v, p, q)$
    $\text{Swap}(v, k, p)$; $e^+ \leftarrow q$; $e^- \leftarrow p + 1$
    while $e^- \leq e^+$ **do**
      while $Star(v)[e^+].cost > Star(v)[p].cost$ **do**
        $e^+ \leftarrow e^+ - 1$
      while $(e^- \leq e^+) \wedge (Star(v)[e^-].cost \leq Star(v)[p].cost)$ **do**
        $e^- \leftarrow e^- + 1$
      if $e^- < e^+$ **then**
        $\text{Swap}(v, e^-, e^+)$; $e^- \leftarrow e^- + 1$; $e^+ \leftarrow e^+ - 1$
    if $(e^+ \neq p)$ **then**
      $\text{Swap}(v, p, e^+)$
    if $e^+ + 1 < q$ **then**
      $\text{Push}(v, e^+ + 1, q)$
    $q \leftarrow e^+ - 1$
$h(v) \leftarrow h(v) + 1$

SQSK: initialization

// Procedure InitSQSK
for $i = 1, \ldots, n$ **do**
  $Stack(i) \leftarrow \emptyset$
  Push$(i, [1, |Star(i)|])$
  $h(i) \leftarrow 0$
  QSstep(i)
BuildHeap$(Heap)$
$T \leftarrow \emptyset$
$count \leftarrow 0$
InitUnionFind

SQSK: iteration

```
// Procedure SQSK. IN: Star. OUT: T.
InitSQSK
while count < n − 1 do
    i ← Heap[1]
    j ← Star(i)[1].vertex
    w ← Star(i)[1].cost
    QSstep(i)
    (QSstep(j))
    IncreaseKey(i)
    (IncreaseKey(j))
    if head[i] ≠ head[j] then
        T ← T ∪ {[i, j]}
        count ← count + 1
        UpdateUnionFind(i, j)
```

Parenthesized instructions are needed if there are duplicates (each edge belongs to two stars).

The pivot selection

The selection of the pivot is crucial role for the performance of all algorithms (like QSstep) that need to partition intervals recursively.

QuickSort.
Purpose: completely sort a set of values.
Pivot selection: try to achieve a balanced partition.

SQSK.
Purpose: quickly compute the next candidate edge when required.
Pivot selection: go for an unbalanced partition.

Most part of each star is likely to be useless: the initial left interval should better be much smaller than the initial right interval.

The pivot selection

Heuristic rule: select r elements at random with uniform probability distribution in $[p, \ldots, q]$ and take the min cost one as the pivot element.

The larger r, the more unbalanced the resulting partition is likely to be.

Rule of thumb: set r to $\min\{(q-p+1)/100, \bar{r}\}$.

Set $\bar{r}$ to a larger value in InitSQSK and to smaller values otherwise.

There is room for heuristics, especially self-adaptive heuristics.

Experimental results

QSK: an iteration of Kruskal algorithm is done sometimes during the execution of QuickSort.
SQSK: an iteration of QuickSort is done sometimes during the execution of Kruskal algorithm.

QSK is designed to work on a list of edges.
SQSK is designed to work on a set of vertex stars.

Both QSK and SQSK can be faster than Prim algorithm.

Besides size and density, the computing time is also affected by the clustering degree of the input graph:
Non-clustered: SQSK wins.
Clustered: Prim wins.