kreher-stinson

Algorithms from the book implemented in GAP

Version 0.1

8 April 2016

Bertín Hernández-Trejo
Rafael Villarroel-Flores
Citlalli Zamora-Mejía

Bertín Hernández-Trejo Email: bertin13@gmail.com

Rafael Villarroel-Flores Email: rvf0068@gmail.com
Homepage: http://rvf0068.github.io

Citlalli Zamora-Mejía Email: cizame@gmail.com

Copyright

Contents

Chapter 1

Generating Combinatorial Objects

1.1 Subsets

1.1.1 KSSubsetLexRank

▷ KSSubsetLexRank(*n*, *T*) (function)

Returns the rank of *T* as a subset of the set of numbers from 1 to *n* (Algorithm 2.1). An error is produced if *T* is not a subset of the set $\{1..n\}$.

Example

```
gap> KSSubsetLexRank(4,[1,2,3]);
14
gap> KSSubsetLexRank(4,[2,4]);
5
gap> KSSubsetLexRank(4,[]);
0
gap> KSSubsetLexRank(4,[1,2,3,4]);
15
gap> KSSubsetLexRank(4,[1,2,3,4,5]);
Error, the set [ 1, 2, 3, 4, 5 ] is not a subset of [1 ..4]
```

1.1.2 KSSubsetLexUnrank

▷ KSSubsetLexUnrank(*n*, *r*) (function)

Returns the subset of $\{1..n\}$ whose rank is *r*. (Algorithm 2.2). The number r has to be greater than 0 and less than $2^n - 1$.

Example

```
gap> KSSubsetLexUnrank(4,14);
[ 1, 2, 3 ]
gap> KSSubsetLexUnrank(4,5);
[ 2, 4 ]
gap> KSSubsetLexUnrank(4,0);
[  ]
gap> KSSubsetLexUnrank(4,15);
[ 1, 2, 3, 4 ]
```

```
gap> KSSubsetLexUnrank(4,17);
Error, there is no subset of [1 ..4] of rank 17
```

1.1.3 KSkSubsetLexRank

▷ KSkSubsetLexRank(*T*, *k*, *n*) (function)

Finds the rank of *T*, among all *k*-subsets of $\{1,2,\ldots,n\}$. If *T* is not a *k*-subset of $\{1,2,\ldots,n\}$, then an error is produced.

Example
```
gap> KSkSubsetLexRank([1,2,3],3,5);
0
gap> KSkSubsetLexRank([1,3,4],3,5);
3
gap> KSkSubsetLexRank([3,4,5],3,5);
9
gap> KSkSubsetLexRank([1,2,3,4],3,5);
Error, the set [ 1, 2, 3, 4 ] is not a 3-subset of [1 .. 5]
gap> KSkSubsetLexRank([1,3,6],3,5);
Error, the set [ 1, 3, 6 ] is not a 3-subset of [1 .. 5]
```

1.1.4 KSkSubsetLexUnrank

▷ KSkSubsetLexUnrank(*r*, *k*, *n*) (function)

Given an integer *r* between 0 and $\binom{n}{k}-1$, returns the *k*-subset of an *n*-set with rank *r*.

Example
```
gap> KSkSubsetLexUnrank(0,3,5);
[ 1, 2, 3 ]
gap> KSkSubsetLexUnrank(3,3,5);
[ 1, 3, 4 ]
gap> KSkSubsetLexUnrank(9,3,5);
[ 3, 4, 5 ]
gap> KSkSubsetLexUnrank(-1,3,5);
Error, there is no 3-subset of [1 .. 5] of rank -1
gap> KSkSubsetLexUnrank(10,3,5);
Error, there is no 3-subset of [1 .. 5] of rank 10
```

1.2 Permutations

1.2.1 KSPermLexRank

▷ KSPermLexRank(*n*, *pi*) (function)

Given a permutation *pi* of $\{1..n\}$, returns the rank of *pi*. (Algorithm 2.15)

1.2.2 KSPermLexUnrank

▷ `KSPermLexUnrank(n, r)` (function)

Returns the permutation of $\{1..n\}$ with rank r. (Algorithm 2.16)

Chapter 2

Bactracking

2.1 Knapsack

2.1.1 KSCheckKnapsackInput

▷ `KSCheckKnapsackInput(`*K*`)` (function)

Checks for valid input data for the Knapsack problems (Problems 1.1-1.4). *K* is a list, which first element is the vector of profits, the second is the vector of weights, and the third is the capacity of the knapsack, which must be an integer.

2.1.2 KSKnapsack1

▷ `KSKnapsack1(`*K*`)` (function)

Implementation of Algorithm 4.1. *K* is a list, which elements are profits, weights, capacity.

2.1.3 KSKnapsack2

▷ `KSKnapsack2(`*K*`)` (function)

Implementation of Algorithm 4.3. *K* is a list, which elements are profits, weights, capacity.

2.2 Generating all cliques

2.2.1 KSAllCliques

▷ `KSAllCliques(`*graph*`)` (function)

Implementation of Algorithm 4.4. A graph G is defined by the list *graph*, which must be a list of subsets of $\{1,...,n\}$, for some integer n. The neighbors of vertex i are the elements of `graph[i]`.

2.3 Exact cover

2.3.1 KSExactCover

▷ `KSExactCover(`*`number, cover`*`)` (function)

Finds an subcollection of *cover* (which is a set of subsets of $\{1,..,number\}$) that is an exact cover of $\{1,..,number\}$, if it exists.

2.3.2 KSRandomSubsetOfSubsets

▷ `KSRandomSubsetOfSubsets(`*`n, delta`*`)` (function)

Generates a random subset of the set of all subsets of $\{1..n\}$, with density *delta*. This can be used as an instance of the ExactCover problem.

2.4 Bounding functions

2.4.1 KSSortForRationalKnapsack

▷ `KSSortForRationalKnapsack(`*`K`*`)` (function)

Given an instance *K* of the Knapsack Problem, where the two first components of *K* represent profits and weights, this function returns a list, where the first component is the same instance of the problem, but the profits and weights have been sorted in non-increasing order of values of *profits[i]*/*weights[i]*. The second component is the permutation applied to the original problem.

2.4.2 KSRationalKnapsackSorted

▷ `KSRationalKnapsackSorted(`*`K`*`)` (function)

Solves the rational Knapsack problem for the instance *K*. Profits and weights must be sorted in non-increasing order of values of *profits[i]*/*weights[i]*.

2.4.3 KSRationalKnapsack

▷ `KSRationalKnapsack(`*`K`*`)` (function)

Solves the rational Knapsack problem for the instance *K*.

2.4.4 KSKnapsack3

▷ `KSKnapsack3(`*`K`*`)` (function)

Solves the Knapsack problem for the instace *K*, using the function KSRationalKnapsack as bounding function.

2.4.5 KSRandomKnapsackInstance

▷ `KSRandomKnapsackInstance(size, maximum_weight)` (function)

Returns a random instance of a Knapsack problem, for *size* objects. The maximum weight is *maximum_weight*. For each i, the profit $P[i]$ is $2*W[i]*\varepsilon$, where ε is a random number between 0.9 and 1.1.

2.4.6 KSRandomTSPInstance

▷ `KSRandomTSPInstance(n, Wmax)` (function)

Returns a random instance of the TSP problem, which is a symmetric *n* by *n* matrix, such that its ij entry is the cost to travel from city i to city j. The entries in the diagonal are made equal to ∞. Each cost is a random integer between 1 and *Wmax*.

2.4.7 KSTSP1

▷ `KSTSP1(G)` (function)

Solves the TSP problem, for the instance *G*, traversing the whole tree space.

2.4.8 KSMinCostBound

▷ `KSMinCostBound(V, G)` (function)

A bounding function for the TSP problem.

2.4.9 KSReduce

▷ `KSReduce(M)` (function)

Reduce function for matrices, which will be useful to implement a secound bounding function for the TSP problem.

2.4.10 KSReduceBound

▷ `KSReduceBound(V, M)` (function)

A second bounding function for the TSP problem. *V* is a partial solution, and *M* is the problem instance. This implements Algorithm 4.12.

2.4.11 KSTSP2

▷ `KSTSP2(G, F)` (function)

Solves the TSP problem for instance *G*, using the bounding function *F*.

2.4.12 KSMaxClique1

▷ `KSMaxClique1(G)` (function)

Adapts the function that lists the complete subgraphs of *G*, to find the size of the largest clique of *G*. This implements Algorithm 4.14.

2.4.13 KSMaxClique2

▷ `KSMaxClique2(G, F)` (function)

Finds the size of the maximum clique in the graph *G*, using the bounding function *F*. This implements Algorithm 4.19.

2.4.14 KSSizeBound

▷ `KSSizeBound(XX, G, Cl)` (function)

A bounding function for the MaxClique problem. *XX* is a complete subgraph of *G*, and *Cl* is the set of candidates to extend *XX*.

2.4.15 KSGenerateRandomGraph

▷ `KSGenerateRandomGraph(n)` (function)

Returns a list of edges of a random graph on *n* vertices. This implements Algorithm 4.20.

2.4.16 KSEdgeListToAdjacencyList

▷ `KSEdgeListToAdjacencyList(Ged, n)` (function)

Given the list of edges *Ged* of a graph with *n* vertices, returns the adjacency list of such graph.

2.4.17 KSGreedyColor

▷ `KSGreedyColor(G)` (function)

Colors the vertices of a graph *G* using a greedy strategy. This implements Algorithm 4.16.

2.4.18 KSSamplingBound

▷ `KSSamplingBound(XX, G, Cl)` (function)

A bounding function for the MaxClique problem. *XX* is a complete subgraph of *G*, and *Cl* is the set of candidates to extend *XX*. This function uses a fixed greedy coloring of the graph *G*. Implements Algorithm 4.17.

2.4.19 KSInducedSubgraph

▷ `KSInducedSubgraph(G, L)` (function)

Returns the adjacency list of the subgraph of *G* induced by the vertices in *L*.

2.4.20 KSGreedyBound

▷ `KSGreedyBound(XX, G, Cl)` (function)

A bounding function for the MaxClique problem. *XX* is a complete subgraph of *G*, and *Cl* is the set of candidates to extend *XX*. This uses a greedy coloring of the subgraph of *G* induced by *L*.

2.4.21 KSGenerateRandomGraph2

▷ `KSGenerateRandomGraph2(n, delta)` (function)

Returns the list of edges of a random graph on *n* vertices with edge density *delta*.

2.4.22 KSTSP3

▷ `KSTSP3(G, F)` (function)

Solves the TSP problem for instance *G*, using bounding function *F*, applying the branch and bound technique.

2.5 Exercises

2.5.1 KSQueens

▷ `KSQueens(size)` (function)

Solves the n queens problem for a *size* × *size* board. (Exercise 4.1.(a))

Example
```
gap> KSQueens(4);
[ 2, 4, 1, 3 ]
[ 3, 1, 4, 2 ]
```

2.5.2 KSWalks

▷ `KSWalks(number)` (function)

Finds all non-overlapping walks in the plane of length *number*. (Exercise 4.1.(b))

Chapter 3

Heuristic Search

3.1 Uniform graph partition

3.1.1 KSRandomkSubset

▷ `KSRandomkSubset(k, n)` (function)

Returns a randomly chosen k-subset of the set of integers from 1 to n.

3.1.2 KSSelectPartition

▷ `KSSelectPartition(n)` (function)

Returns a random partition of the set $\{1, 2, \ldots, 2n\}$ into two subsets of size n each. (Algorithm 5.7)

3.1.3 KSCost

▷ `KSCost(G, P)` (function)

Returns the cost of the partition P of the vertices of the weighted graph G.

3.1.4 KSGain

▷ `KSGain(G, P, u, v)` (function)

P is a partition in equal parts of the vertices of G. This function calculates the change in the value of the cost function when interchanging the vertex u from the first set in the partition P with the vertex v which is in the second set of the partition.

3.1.5 KSRandomCostMatrix

▷ `KSRandomCostMatrix(n, Wmax)` (function)

Returns a symmetric n by n matrix, such that its entries are random integers from 0 to $Wmax$, and with zeros in the main diagonal.

3.1.6 KSAscend

▷ `KSAscend(G, P)` (function)

Given a partition *P* of the vertices of the weighted graph *G*, it returns a partition *Q* with less cost than *P*, by exchanging one vertex of the partition, if such partition exists. Otherwise, returns the same partition *P*.

3.2 Steiner systems

3.2.1 KSConstructBlocks

▷ `KSConstructBlocks(v, other)` (function)

Constructs a list of blocks of length *v* from the list of lists *other*. (Algorithm 5.12)

3.2.2 KSRevisedStinsonAlgorithm

▷ `KSRevisedStinsonAlgorithm(v)` (function)

Constructs a Steiner triple system with *v* points, using a hill-climbing algorithm. Implements Algorithm 5.19.

3.3 The knapsack problem

3.3.1 KSKnapsackSimulatedAnnealing

▷ `KSKnapsackSimulatedAnnealing(K, cmax, T0, alpha)` (function)

Implements Algorithm 5.20. *K* is the instance of the Knapsack problem to solve. *cmax* is the number of iterations to be done. *T0* is the initial "temperature" and *alpha* is the parameter of the "cooling schedule".

3.3.2 KSRandomFeasibleSolutionKnapsack

▷ `KSRandomFeasibleSolutionKnapsack(K)` (function)

Returns a randomly chosen feasible solution to the Knapsack problem instance *K*.

3.3.3 KSKnapsackTabuSearch

▷ `KSKnapsackTabuSearch(K, cmax, L)` (function)

Searches for an optimal solution to the Knapsack problem instance *K* using a tabu search list. *cmax* is the maximum number of iterations, and *L* the length of iterations a tabu search should be kept.

3.4 Heuristics for the TSP

3.4.1 KSGainTSP

▷ `KSGainTSP(XX, i, j, M)` (function)

Gain function for the Traveling Salesman Problem.

3.4.2 KSSteepestAscentTwoOpt

▷ `KSSteepestAscentTwoOpt(XX, M)` (function)

Given an instance of the TSP problem *M*, and an initial permutation *XX*, applies steepest ascent heuristic.

3.4.3 KSSelect

▷ `KSSelect(popsize, M)` (function)

Returns a population of size *popsize* for the TSP problem *M*.

3.4.4 KSPartiallyMatchedCrossover

▷ `KSPartiallyMatchedCrossover(n, alpha, beta, j, k)` (function)

One way to obtain two new permutations from permutations *alpha*, *beta*.

Index