itap

Information Theoretic Achievability Prover

1.0

25/08/2015

Jayant Apte

John Walsh

Jayant Apte
Email: jayant91089@gmail.com
Homepage: https://sites.google.com/site/jayantapteshomepage/
Address: Department of Electrical and Computer Engineering
Drexel University
Philadelphia, PA 19104

John Walsh
Email: jwalsh@coe.drexel.edu
Homepage: http://www.ece.drexel.edu/walsh/web/
Address: Department of Electrical and Computer Engineering
Drexel University
Philadelphia, PA 19104

Contents

Chapter 1

Introduction

ITAP stands for Information Theoretic Achievability Prover. So basically, it is intended for coming up with achievability proofs using a computer. As of now, it supports the following:

- Achievability proofs of multi-source network coding using vector-linear codes: answer questions like 'Is this rate vector achievable using a vector linear code over the given finite field?'. If the answer is 'yes' it also returns the vector linear code it found as a certificate of achievability. Otherwise, it just says 'no'.
- Achievability proofs with multi-linear secret sharing schemes: answer questions 'Is there a multi-linear secret sharing scheme over $GF(q)$ for this access structure?'
- Representability test for integer polymatroids over a given finite field: answer questions like 'Is the integer polymatroid associated with this rank vector linear over $GF(q)$?

All three questions above are very similar, in that, we are looking for a linear representation of an integer polymatroid satisfying certain properties (satisfying network coding constraints, access structure and having a specified rank vecor resp.) In the most general form an achievability prover should be able to tell if there exists a joint distribution satisfying certain constraints on entropy function which remains a fundamental open problem. ITAP tries answering this in a more restricted sense, i.e. with vector linear codes. The algorithm underlying `itap` is called Leiterspiel or the algorithm of snakes and ladders. See [BBF+06] for details.

Chapter 2

Installation

To get the latest version of this GAP 4 package download one of the archive files itap-x.x.zip, itap-x.x.tar.gz, itap-x.x.zoo, itap-x.x.tar.bz2 from `https://github.com/jayant91089/itap` or `http://www.ece.drexel.edu/walsh/aspitrg/software.html` and unpack it using

```
gunzip itap-x.x.zip
```

or

```
gunzip itap-x.x.tar.gz
```

respectively and so on. Do this preferably (but not necessarily) inside the `pkg` subdirectory of your GAP 4 installation. It creates a subdirectory called `itap`. This completes the installation of the package. One can then start using `itap` by invoking

gap>

Code

```
LoadPackage( "itap");
```

from within GAP. **Optional Dependencies**: `itap` can optionally use gap package `FinInG`. `FinInG` provides allows internal functions to use projective semilinear group $P\Gamma L(k,q)$, which is the group of all collineations of the vector space $GF(q)^k$. See `http://cage.ugent.be/fining/` for information on how to obtain and install this package. `LoadPackage( "itap")` automatically checks if `FinInG` is available and loads it in case it is available. If `FinInG` is absent, a light version of `itap` is loaded, where the internal functions default to using projective general linear group $PGL(k,q) \leq P\Gamma L(k,q)$. Note that this distinction is irrelevent if q is a prime number as $PGL(k,q) \cong P\Gamma L(k,q)$.

Chapter 3

Usage

3.1 Available functions

In this section we shall look at the functions provided by `itap`.

3.1.1 proverep

▷ `proverep(rankvec, nvars, F, optargs)` (function)

Returns: A list

This function checks if there is a linear representation of an integer polymatroid rank vector. It accepts following arguments:

- `rankvec` - A list of integers specifying a polymatroid rank vector
- `nvars` - Number of ground set elements
- `F` - The finite field over which we wish to find a linear representation. It can be defined by invoking the following in GAP:

Code
```
q:=4;;
F:= GF(q);; # here q must be a prime power
```

- `optargs` is a list of optional arguments `[lazy,..]` where
  - `lazy` disables lazy evaluation of transporter maps if set to `false`, which is enabled by default in GAP.

Returns a list `[true,code]` if there exists such a representation and `code` is the vector linear code over $GF(q)$ Returns a list `[false]` otherwise, indicating that no such representation exists

3.1.2 proverate

▷ `proverate(ncinstance, rvec, F, optargs)` (function)

Returns: A list

This function checks if there is a vector linear code achieving the rate vector `rvec` for the network coding instance `ncinstance`. It accepts following arguments:

- ncinstance is a list [cons,nsrc,nvars] containing 3 elements:
  - cons is a list of network coding constraints.
  - nsrc is the number of sources.
  - nvars is the number of random variables associated with the network.
- rvec - A list of nvars integers specifying a rate vector
- F is the finite field over which we wish to find the vector linear code. It can be defined by invoking the following in GAP:

Code
```
q:=4;;
F:= GF(q);; # here q must be a prime power
```

- optargs is a list of optional arguments [lazy,..] where lazy disables lazy evaluation of transporter maps if set to false, which is enabled by default.

Returns a list [true,code] if there exists such a representation and code is the vector linear code over $GF(q)$ Returns a list [false] otherwise, indicating that no such code exists

NOTE: Certain naming convensions are followed while defining network coding instances. The source messages are labeled with [1...nsrc] while rest of the messages are labeled [nsrc...nvars]. Furthermore, the list cons includes all network constraints except source independence. This constraint is implied with the labeling i.e. itap enforces it by default for the messages labeled [1...nsrc]. See next section for usage examples.

3.1.3 provess

▷ provess(*Asets, nvars, svec, F, optargs*) (function)

Returns: A list

This function checks if there is a multi-linear secret sharing scheme with secret and share sizes given by svec and the access structure Asets with one dealer and nvars-1 parties. It accepts following arguments:

- Asets - A list of authorized sets each specified as a subset of $[\texttt{nvars} - 1]$
- nvars - Number of participants (including one dealer)
- svec - vector of integer share sizes understood as number of $\mathbb{F}_q$ symbols, with index 1 giving secret size and index $i, i \in \{2, ..., \texttt{nvars}\}$ specifying share sizes of different parties
- F - The finite field over which we wish to find a multi-linear scheme. It can be defined by invoking the following in GAP:

Code
```
q:=4;;
F:= GF(q);; # here q must be a prime power
```

- optargs is a list of optional arguments [lazy,..] where
  - lazy disables lazy evaluation of transporter maps if set to false, which is enabled by default in GAP.

Returns a list [true,code] if there exists such a scheme and code is the vector linear code over $GF(q)$ Returns a list [false] otherwise, indicating that no such scheme exists

NOTE: No input checking is performed to verify if input Asets follows labeling convensions. The map returned with a linear scheme is a map $f : [\mathtt{nvars}] \to [\mathtt{nvars}]$ with dealer associated with label 1 and parties associated with labels $\{2, ..., \mathtt{nvars}\}$. See next section for usage examples.

3.1.4 DisplayCode

▷ DisplayCode(*code*) (function)

Returns: nothing

Displays a network code (or an integer polymatroid). It accepts 1 argument:

- code - A list [s,map] containing 2 elements:
  - s - A list of subspaces where is subspace is itself a list of basis vectors
  - map - A GAP record mapping subspaces in s to network messages or to polymatroid ground set elements

Returns nothing

Example
```
gap> s:=[ [ [ 0*Z(2), 0*Z(2), Z(2)^0 ] ], [ [ 0*Z(2), Z(2)^0, 0*Z(2) ] ],\
> [ [ 0*Z(2), Z(2)^0, Z(2)^0 ] ], [ [ Z(2)^0, 0*Z(2), 0*Z(2) ] ],\
> [ [ Z(2)^0, 0*Z(2), Z(2)^0 ] ], [ [ Z(2)^0, Z(2)^0, 0*Z(2) ] ],\
> [ [ Z(2)^0, Z(2)^0, Z(2)^0 ] ] ];;
gap> map:=rec( 1 := 1, 2 := 2, 3 := 4, 4 := 3, 5 := 6, 6 := 5, 7 := 7 );;
gap> DisplayCode([s,map]);;
1->1
 . . 1
=============================
2->2
 . 1 .
=============================
3->4
 . 1 1
=============================
4->3
 1 . .
=============================
5->6
 1 . 1
=============================
6->5
1 1 .
=============================
7->7
 1 1 1
=============================
```

3.2 A catalog of examples

`itap` comes equipped with a catalog of examples, which contains well-known network coding instances and integer polymatroids. This catalog is intended to be of help to the user for getting started with using `itap`. Most of the network coding instances in this catalog can be found in [Yeu08] and [DFZ07]. Some of the integer polymatroids in the catalog are taken from `http://code.ucsd.edu/zeger/linrank/`. These polymatroids correspond to the extreme rays of the cone of linear rank inequalities in 5 variables, found by Dougherty, Freiling and Zeger. See [DFZ09] for details.

3.2.1 FanoNet

▷ `FanoNet()` (function)

Returns: A list

Returns the Fano instance. It accepts no arguments. Returns a list.

Example

```
gap> FanoNet();
[ [ [ [ 1, 2 ], [ 1, 2, 4 ] ], [ [ 2, 3 ], [ 2, 3, 5 ] ],
      [ [ 4, 5 ], [ 4, 5, 6 ] ], [ [ 3, 4 ], [ 3, 4, 7 ] ],
      [ [ 1, 6 ], [ 3, 1, 6 ] ], [ [ 6, 7 ], [ 2, 6, 7 ] ],
      [ [ 5, 7 ], [ 1, 5, 7 ] ] ], 3, 7 ]
gap> rlist:=proverate(FanoNet(),[1,1,1,1,1,1,1],GF(2),[]);;
gap> rlist[1]; # Fano matroid is representable over GF(2)
true
gap> DisplayCode(rlist[2]);
1->1
 . . 1
==============================
2->2
 . 1 .
==============================
3->4
 . 1 1
==============================
4->3
 1 . .
==============================
5->6
 1 . 1
==============================
6->5
 1 1 .
==============================
7->7
 1 1 1
==============================
gap> rlist:=proverate(FanoNet(),[1,1,1,1,1,1,1],GF(3),[]);;
gap> rlist[1]; # Fano matroid is not representable over GF(3)
false
```

3.2.2 NonFanoNet

▷ `NonFanoNet()` (function)

Returns: A list

Returns the NonFano instance. It accepts no arguments. Returns a list.

Example
```
gap> NonFanoNet();
gap> gap> NonFanoNet();
[ [ [ [ 1, 2, 3 ], [ 1, 2, 3, 4 ] ], [ [ 1, 2 ], [ 1, 2, 5 ] ],
      [ [ 1, 3 ], [ 1, 3, 6 ] ], [ [ 2, 3 ], [ 2, 3, 7 ] ],
      [ [ 4, 5 ], [ 3, 4, 5 ] ], [ [ 4, 6 ], [ 2, 4, 6 ] ],
      [ [ 4, 7 ], [ 1, 4, 7 ] ] ], 3, 7 ]
```

3.2.3 VamosNet

▷ `VamosNet()` (function)

Returns: A list

Returns the Vamos instance. It accepts no arguments. Returns a list.

Example
```
gap> VamosNet();
[ [ [ [ 1, 2, 3, 4 ], [ 1, 2, 3, 4, 5 ] ],
      [ [ 1, 2, 5 ], [ 1, 2, 5, 6 ] ],
      [ [ 2, 3, 6 ], [ 2, 3, 6, 7 ] ],
      [ [ 3, 4, 7 ], [ 3, 4, 7, 8 ] ],
      [ [ 4, 8 ], [ 2, 4, 8 ] ],
      [ [ 2, 3, 4, 8 ], [ 1, 2, 3, 4, 8 ] ],
      [ [ 1, 4, 5, 8 ], [ 1, 2, 3, 4, 5, 8 ] ],
      [ [ 1, 2, 3, 7 ], [ 1, 2, 3, 4, 7 ] ],
      [ [ 1, 5, 7 ], [ 1, 3, 5, 7 ] ] ], 3, 7 ]
```

3.2.4 U2kNet

▷ `U2kNet()` (function)

Returns: A list

Returns the instance associated with uniform matroid U^2_k. It accepts one argument `k` specifying the size of uniform matroid. Returns a list.

Example
```
gap> U2kNet(4);
[ [ [ [ 1, 2 ], [ 1, 2, 3 ] ], [ [ 1, 2 ], [ 1, 2, 4 ] ],
      [ [ 1, 3 ], [ 1, 2, 3 ] ], [ [ 1, 3 ], [ 1, 3, 4 ] ],
      [ [ 1, 4 ], [ 1, 2, 4 ] ], [ [ 1, 4 ], [ 1, 3, 4 ] ],
      [ [ 2, 3 ], [ 1, 2, 3 ] ], [ [ 2, 3 ], [ 2, 3, 4 ] ],
      [ [ 2, 4 ], [ 1, 2, 4 ] ], [ [ 2, 4 ], [ 2, 3, 4 ] ],
      [ [ 3, 4 ], [ 1, 3, 4 ] ], [ [ 3, 4 ], [ 2, 3, 4 ] ]
    ], 2, 4 ]
```

3.2.5 ButterflyNet

▷ `ButterflyNet()` (function)

Returns: A list

Returns the Butterfly instance. It accepts no arguments. Returns a list.

Example
```
gap> ButterflyNet();
[ [ [ [ 1, 2 ], [ 1, 2, 3 ] ], [ [ 1, 3 ], [ 1, 2, 3 ] ],
     [ [ 2, 3 ], [ 1, 2, 3 ] ] ], 2, 3 ]
gap> U2kNet(3)=ButterflyNet();
true
```

3.2.6 Subspace5

▷ Subspace5() (function)

Returns: A list

Returns the extreme rays of cone formed by linear rank inequalities in 5 variables. It accepts no arguments. Returns a list.

Example
```
gap> rays5:=Subspace5();;
gap> Size(rays5);
162
gap> rlist:=proverep(rays5[46],5,GF(2),[])
> rlist[1];
true
gap> gap> DisplayCode(rlist[2]);
1->4
 . . . 1
=============================
2->5
 . . 1 .
=============================
3->3
 . 1 . .
=============================
4->2
 1 . . .
 . . 1 1
=============================
5->1
 1 . . 1
 . 1 1 1
=============================
```

3.2.7 BenaloahLeichter

▷ BenaloahLeichter() (function)

Returns: A list of lists specifing authorized subsets of $\{1,2,3,4\}$

Returns the access structure associated with secret sharing scheme of Benaloah and Leichter that was used to show that share sizes can be larger than secret size. See [BL90] for details. It accepts no arguments. Returns a list.

Example
```
gap> B:=BenaloahLeichter();
[ [ 1, 2 ], [ 2, 3 ], [ 3, 4 ] ]
```

```
gap> rlist:=provess(B,5,[2,2,3,3,2],GF(2),[]);;
gap> rlist[1];
true
gap> DisplayCode(rlist[2]);
1->1
 . . . . 1 .
. . . . . 1
=============================
2->2
. . 1 . . .
. . . 1 . .
=============================
3->3
. 1 . . . .
. . 1 . . 1
. . . 1 1 .
=============================
4->5
1 . . . . .
. 1 . . . .
=============================
5->4
1 . . . . 1
. 1 . . 1 .
. . 1 . . .
=============================
```

3.2.8 Threshold

▷ `Threshold()` (function)

Returns: A list of lists specifing authorized subsets of $[n]]$

Returns the access structure associated with Shamir's (k,n) threshold scheme. See [Sha79] for details. It accepts following arguments:

- `n` - number of shares
- `k` - threshold

Example

```
gap> T:=Threshold(4,2);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
gap> rlist:=provess(T,5,[1,1,1,1,1],GF(2),[]);
[ false ]
gap> rlist:=provess(T,5,[1,1,1,1,1],GF(3),[]);
[ false ]
gap> rlist:=provess(T,5,[1,1,1,1,1],GF(5),[]);;
gap> rlist[1];
true
gap> DisplayCode(rlist[2]);
1->1
 . 1
=============================
2->2
```

```
 1 .
==============================
3->3
 1 1
==============================
4->4
 1 2
==============================
5->5
1 4
==============================
```

References

[BBF$^+$06] A. Betten, M. Braun, H. Fripertinger, A. Kerber, A. Kohnert, and A. Wassermann. *Error-Correcting Linear Codes: Classification by Isometry and Applications*. Algorithms and Computation in Mathematics. Springer Berlin Heidelberg, 2006. 3

[BL90] J. Benaloh and J. Leichter. Generalized secret sharing and monotone functions. In *Proceedings on Advances in Cryptology*, CRYPTO '88, pages 27–35, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 10

[DFZ07] R. Dougherty, C. Freiling, and K. Zeger. Networks, matroids, and non-shannon information inequalities. *Information Theory, IEEE Transactions on*, 53(6):1949–1969, 2007. 8

[DFZ09] R. Dougherty, C. Freiling, and K. Zeger. Linear rank inequalities on five or more variables. *arXiv cs.IT/0910.0284v3*, 2009. 8

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979. 11

[Yeu08] R. W. Yeung. *Information Theory and Network Coding*. Springer, 2008. 8

Index