User Manual

Contents

1 Introduction

This document describes how to use *Triangle Algorithm* and *Spherical Triangle Algorithm*, which are part of a Matlab package for *Convex Hull Membership* (CHM) problems, a fundamental problem in Linear Programming, Computational Geometry, Machine Learning and Statistics. It also serves as a query problem in many applications, e.g., Topic Modeling, LP Feasibility and Data Reduction. Definitions and technical details are included in the main article [5].

2 Installation

The algorithms are implemented using Matlab 2019 [1]. To begin, one needs to install Matlab. The algorithms also works on Octave 5.2.0 with additional statistics package. Having started Matlab, access to all the user callable functions is obtained using:

```
%%%
addpath(Directory containing Spherical Triangle Algorithm)
%%%
```

Note that the path needs to be added each time Matlab is started.

3 Main Functions

3.1 *Tri_algo*

```
[inorout,p_prime,alpha_coe,gap,iter_num]...
=Tri_Algo(input_a,epsilon,query,varargin)
```

Description

The function `Tri_algo` is an implementation of the *Triangle Algorithm* (TA) described in [4]. The Triangle Algorithm is an iterative algorithm for solving the approximate CHM problem : given a set $S = \{v_1, \dots, v_n\} \subset \mathbb{R}^m$, a distinguished point $p \in \mathbb{R}^m$, and $\varepsilon \in (0,1)$, solving CHM means either computing an *ε-approximate solution*, i.e. $p_\varepsilon \in conv(S)$ so that $\|p - p_\varepsilon\| \leq \varepsilon R, \quad R = \max\{\|v_i - p\| : v_i \in S\}$ or *witness*: a point $p' \in conv(S)$ for which there is no $v \in S$ with $\|p' - v\| \geq \|p - v\|$. If p' is a witness, the orthogonal bisecting hyperplane to pp' separates p from $conv(S)$.

Parameters

Input:

- *input_a*: An $m \times n$ dimensional matrix representing the set of n points in dimension m.
- *epsilon*: A scalar in $(0,1)$; precision parameter. In practice, the *epsilon* is usually chosen between $[0.01, 0.001]$. The number of iteration in the worst case has $O(1/\varepsilon^2)$ iterations.

[1]Any advice or opinions posted here are our own, and in no way reflect that of MathWorks.

- *varargin*: This variable is set to be an $n \times 1$ vector with non-negative entries that sum up to 1 if manual initialization is used. Otherwise leave it blank.

- *query*: An $m \times 1$ vector representing p, the query point in dimension m.

Output:

- *innout*: $0, 1$ binary scalar. Its value is 0 if a witness is found, 1 if an ϵ-approximation solution is found.

- *alpha_coe* : An $n \times 1$ vector. The value of α so that $A\alpha = p'$ where p' is an ε solution.

- *p_prime*: An $m \times 1$ vector. Witness or an ε-approximation solution of query point.

- *gap*: Euclidean distance between *p_prime* and query point p.

- *iter_num*: Number of iterations for algorithm to stop.

Examples

This example uses `Tri_algo` to solve the convex hull membership problem. For simplicity, the data points are generated from a unit sphere. The query point p is generated using a random convex combination of the given set of points. In the example the number of points, n, is set to 20 and the dimension, m, to 10.

```
m=10;
n=20;
epsilon=0.01;
%% Generate points from unit sphere
matA=Random_pts(m,n,'unit ball');
%%
%% query points p is in the convex hull
p=Random_cvx(matA,1);
%%
tic
[inorout,p_prime,alpha_coe,gap,iter_num]...
=Tri_Algo(matA,epsilon,p);
toc
alpha_coe
```

```
p_prime
inorout
dist
```

The output of above program is:

```
alpha_coe                       p_prime
=                               =
      0                         0.0008
0.0137                          -0.0611
0.0355                          -0.0220
0.0239                          -0.0541
0.2518                          0.0501
0.1666                          -0.0826
0.0676                          -0.0605
0                               -0.0270
0                               0.0398
0                               0.0310
0
0.0495
0.2012
0
0
0.0616
0.0122
0
0.0417
0.0745

inorout =1
dist=0.0104
Elapsed time is
0.032385 seconds.
```

3.2 *Spherical_TA*

```
[innout,p_prime,alpha_coe,gap,iter_num]...
=Spherical_TA(input_a,epsilon,alpha_0,query)
```

Description

`Spherical_TA` is an implementation of the *Spherical Triangle Algorithm*, which solves the CHM problem by converting a CHM into a spherical CHM. The *Spherical-CHM* is the case of CHM, where $p = 0$ and each $v_i \in S$ has unit norm. Given a data set $S^r = \{v_1^r, ...v_n^r\}$ and p^r, we set $p = 0$ and set $S = \{v_1, ..., v_n\}$, where $v_i = (v_i^r - p^r)/\|v_i^r - p^r\|$. This step scales every point onto a unit sphere. The function `Spherical_TA` then applies the Triangle Algorithm to solve the scaled problem. The solution to the Spherical CHM is then converted back to a solution of the original CHM query. Details are provided in [5].

Implementation

Input:

- *input_a*: An $m \times n$ dimensional matrix representing the set of n points in dimension m.
- *epsilon*: A scalar in $(0, 1)$; precision parameter. In practice, the *epsilon* is usually chosen between $[0.01, 0.001]$. The number of iteration in the worst case has $O(1/\varepsilon^2)$ iterations.
- *alpha_0*: An $n \times 1$ vector with non negative entries that sum up to 1; initialization of the coefficient.
- *query*: An $m \times 1$ vector representing the query point p in dimension m.

Output:

- *innout*: $0, 1$ binary scalar. Its value is 0 if a witness is found, and 1 if an ϵ approximation solution is found.
- $alpha_coe$: An $n \times 1$ vector. The value of α so that $A\alpha = p'$ where p' is an ε solution.
- *p_prime*: An $m \times 1$ vector. Witness or an ε-approximation solution of query point.
- *gap*: Euclidean distance between *p_prime* and the query point p.
- *iter_num*: Number of iterations for the algorithm to stop.

Examples

This example uses `Spherical_TA` to solve CHM. For simplicity, the data points are generated from a unit sphere. The query point p is generated using a random convex combination of a given set of points. In the example the number of points, n, is set to 20 and the dimension, m, to 10.

```
m=10;
n=20;
%% Generate points from unit sphere
matA=Random_pts(m,n,'unit ball');
%%
%% query points p is in the convex hull
p=Random_cvx(matA,1);
%%
tic
[inorout,p_prime,alpha_coe,gap,iter_num]...
=Spherical_TA(matA,epsilon,zeros(n,1),p);
toc
alpha_coe
p_prime
inorout
dist
```

The output of above program is:

```
alpha_coe
=
0.0482
0
0.1060
0.0288
0
0.0461
0.0057
0
0
0
0.0180
0.0243
0.0101
```

```
0.2003
0.0206
0.2211
0
0.1607
0.0279
0.0820

p_prime
=
-0.0038
0.0006
-0.0163
0.0006
-0.0048
0.0076
-0.0015
0.0036
0.0074
0.0005

inorout =1
dist=0.0058
Elapsed time is
0.019650 seconds.
```

3.3 *AVTA_PLUS*

`[index]=AVTA_PLUS(matrix_A,gamma,varargin)`

Description

The function `AVTA_PLUS` is an implementation of *AVTA+* in [5]. *AVTA* [1] solves the exact or approximate version of the *irredundancy problem* [6]: given a set of n points $S = \{v_1, \ldots, v_n\}$ in $\mathbb{R}^m$, it computes all vertices of its convex hull. *AVTA+* replaces the use of Triangle Algorithm in AVTA by Spherical-TA as a faster membership query oracle. The function also includes an implementation of J-L random projection [3] for dimension reduction.

Input:

- *matrix_A*: An $m \times n$ matrix representing set of n points in dimension m.

- *gamma*: A scalar in $(0, 1)$; precision parameter.In practice, the *gamma* is usually chosen between $[0.05, 0.001]$.
- *varargin*: If random projection is used for dimension reduction, this variable is set to be $'Rp'$, dimension of random projection. Otherwise leave it blank. For example,
  `[index]=AVTA_PLUS(matrix_A,gamma,'Rp',50)`
  uses random projection with dimension 50.
  `[index]=AVTA_PLUS(matrix_A,gamma)`
  doesn't not apply random projection.

Output:

- *index*: Integer vector, subset of $[n]$. Index of vertices.

This example uses `AVTA_PLUS` to solve the irredundancy problem using points generated from a unit sphere.The problem specified below uses $n = 100$ points, dimension $m = 10$ and $K = 20$ vertices.

```
m=20;
K=20;
n=100;
vtx_A=Random_pts(m,K,'normal');
inhulldata=Random_cvx(vtx_A,n,'unif');
matA=[inhulldata,vtx_A];
[m,n]=size(matA);
tic
[index]=AVTA_PLUS(matA,0.01);
toc
index
```

The output of above program is:

```
Elapsed time is
0.181165 seconds.

index =
101
102
103
104
105
```

4 Auxiliary Functions

4.1 *Random_pts*

```
[mat_A]=Random_pts(m,n,varargin)
%%This function generates random data points.
```

Input:

- n: Number of points.
- m: Dimension.
- $varargin$:
  - $'normal`$: The data points are generated using standard Gaussian distributution.
  - $'unif`$: Thedata points are generated using standard uniform distribution.
  - $'unitball`$: The data points are uniformly generated on a unit sphere.

Output:

- mat_A: The $m \times n$ matrix containing the set of random generated points.

4.2 *Random_cvx*

```
[mat_A]=Random_cvx(vtx_A,n,varargin)
%% Given a set of points as generators, this function generates n
%% points using a convex combination of the generators.
```

Input:

- n: Number of points.
- m: Dimension.
- $varargin$:
  - $['dir', para]$: The data points are generated using the standard dirichlet distribution with $para$ as hyper parameter.
  - $'unif$‘: The data points are generated using the scaled standard uniform distribution

Output:

- mat_A: The $m \times n$ matrix representing set of random generated points.

5 Demo

The `Demo.m` driver uses data generated using a Gaussian distribution. Following three algorithms are used to solve the CHM problem: the Simplex method [2] (using package provided by Matlab), the TA [4], and the Spherical-TA. The output in Figure 1 consists of two plots that compare the run times of the three methods on all the tests executed. The two plots corresponds to the two cases in CHM when the query p is feasible or infeasible.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% This is a demo for Spherical Triangle Algrithm
%% Full paper available on Arxiv:
%% Spherical Triangle Algorithm: A Fast
%% Oracle for Convex Hull Membership Queries
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
clear all
options = optimset('linprog');
options.Display = 'off';
%% Number of repetition
Num_rep=3;
%%
%% Problem size (dim,number of points)
Problem_size=[100,1000;200,2000;500,5000;1000,10000];
%%
%% Precision parameter
epsilon=0.01;
%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%Query point infeasible
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%Triangle Algorithm running time
time_ta=zeros(1, length(Problem_size(:,1)));
%%
%%Spherical Triangle Algorithm running time
time_sphere_ta=zeros(1, length(Problem_size(:,1)));
%%
%% Linear Programming running time
time_lp=zeros(1, length(Problem_size(:,1)));
%%
%% Itertion number Spherical Triangle Algorithm
iter_ta=zeros(1, length(Problem_size(:,1)));
%%
%% Itertion number Triangle Algorithm
iter_sphere_ta=zeros(1, length(Problem_size(:,1)));
%%
for ii=1:length(Problem_size(:,1))
Num_dim=Problem_size(ii,1);
Num_vtx=Problem_size(ii,2);
for kk=1:Num_rep
%%%Data generation
vtx_A=Random_pts(Num_dim,Num_vtx,'normal');
matA=vtx_A;
[m,n]=size(matA);
%p=Random_cvx(vtx_A,1,'dir');%% Query points
p=Random_pts(Num_dim,1,'normal');
```

```
%%%
disp('Triangle Algorithm starts')
%%Triangle Algorithm
tic;
[inorout,p_prime,alpha_coe,dist,ta_iter]=Tri_Algo(matA,epsilon,p);
ta_end=toc;
time_ta(ii)=time_ta(ii)+ta_end;
iter_ta(ii)=iter_ta(ii)+ta_iter;
disp('Triangle Algorithm time')
disp(ta_end)
%%Spherical Triangle Algorithm
disp('Spherical Triangle Algorithm  starts')
tic;
[inorout,p_prime,alpha_coe,dist,iter_num]...
=Spherical_TA(matA,epsilon,zeros(n,1),p);
sphere_ta_end=toc;
time_sphere_ta(ii)=time_sphere_ta(ii)+sphere_ta_end;
iter_sphere_ta(ii)=iter_sphere_ta(ii)+iter_num;
disp('Spherical Triangle Algorithm  time')
disp(sphere_ta_end)
%%Linear Programming
disp('Linear Programming starts')
tic;

n_var=n;
n_con=m;
x_y_size=n_con+n_var;
cvx_aeq=ones(1,n_var);
Aeq=[matA,eye(n_con);ones(1,n_var),zeros(1,n_con)];
beq=[p;1];
lb=zeros(x_y_size,1);
tic
options = optimset('linprog');
options.Display = 'off';

[bb,fval] = linprog([zeros(n_var,1);ones(n_con,1)],[],[]...
,Aeq,beq,lb,[],options);
lp_t=toc;
time_lp(ii)=time_lp(ii)+lp_t;
disp('Linear Programming time')
disp(lp_t)
```

```
end
end

figure(1);
hold on
title('query point infeasible')
plot(log((time_ta(1,:))),'DisplayName','TA','LineWidth',1.5)
plot(log((time_sphere_ta(1,:))),'DisplayName',...
'Spherical TA','LineWidth',1.5)
plot(log(time_lp(1,:)),'DisplayName','LP','LineWidth',1.5)  ;
% plot(mean(time_mirror_ta(2,:)))
legend('show','Location','northwest')%,'Orientation','horizontal')
hold off;









%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%Query point feasible
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%%Triangle Algorithm running time
time_ta=zeros(1, length(Problem_size(:,1)));
%%
%%Spherical Triangle Algorithm running time
time_sphere_ta=zeros(1, length(Problem_size(:,1)));
%%
%% Linear Programming running time
time_lp=zeros(1, length(Problem_size(:,1)));
%%
%% Itertion number Spherical Triangle Algorithm
iter_ta=zeros(1, length(Problem_size(:,1)));
%%
```

```
%% Itertion number Triangle Algorithm
iter_sphere_ta=zeros(1, length(Problem_size(:,1)));
%%
for ii=1:length(Problem_size(:,1))
Num_dim=Problem_size(ii,1);
Num_vtx=Problem_size(ii,2);
for kk=1:Num_rep
%%%Data generation
vtx_A=Random_pts(Num_dim,Num_vtx,'normal');
matA=vtx_A;
[m,n]=size(matA);
p=Random_cvx(vtx_A,1,'dir');%% Query points
%         p=Random_pts(Num_dim,1,'normal');
%%%
disp('Triangle Algorithm starts')
%%Triangle Algorithm
tic;
[inorout,p_prime,alpha_coe,dist,ta_iter]=Tri_Algo(matA,epsilon,p);
ta_end=toc;
time_ta(ii)=time_ta(ii)+ta_end;
iter_ta(ii)=iter_ta(ii)+ta_iter;
disp('Triangle Algorithm time')
disp(ta_end)
%%Spherical Triangle Algorithm
disp('Spherical Triangle Algorithm  starts')
tic;
[inorout,p_prime,alpha_coe,dist,iter_num]...
=Spherical_TA(matA,epsilon,zeros(n,1),p);
sphere_ta_end=toc;
time_sphere_ta(ii)=time_sphere_ta(ii)+sphere_ta_end;
iter_sphere_ta(ii)=iter_sphere_ta(ii)+iter_num;
disp('Spherical Triangle Algorithm  time')
disp(sphere_ta_end)

%%Linear Programming
disp('Linear Programming starts')
tic;

n_var=n;
n_con=m;
x_y_size=n_con+n_var;
```

```
cvx_aeq=ones(1,n_var);
Aeq=[matA,eye(n_con);ones(1,n_var),zeros(1,n_con)];
beq=[p;1];
lb=zeros(x_y_size,1);
tic
options = optimset(’linprog’);
options.Display = ’off’;

[bb,fval] = linprog([zeros(n_var,1);ones(n_con,1)]...
,[],[],Aeq,beq,lb,[],options);
lp_t=toc;
time_lp(ii)=time_lp(ii)+lp_t;
disp(’Linear Programming time’)
disp(lp_t)
end
end

figure(2);
hold on
title(’query point feasible’)
plot(log((time_ta(1,:))),’DisplayName’,’TA’,’LineWidth’,1.5)
plot(log((time_sphere_ta(1,:)))...
,’DisplayName’,’Spherical TA’,’LineWidth’,1.5)
plot(log(time_lp(1,:)),’DisplayName’,’LP’,’LineWidth’,1.5)  ;
% plot(mean(time_mirror_ta(2,:)))
legend(’show’,’Location’,’northwest’)%,’Orientation’,’horizontal’)
hold off;
```

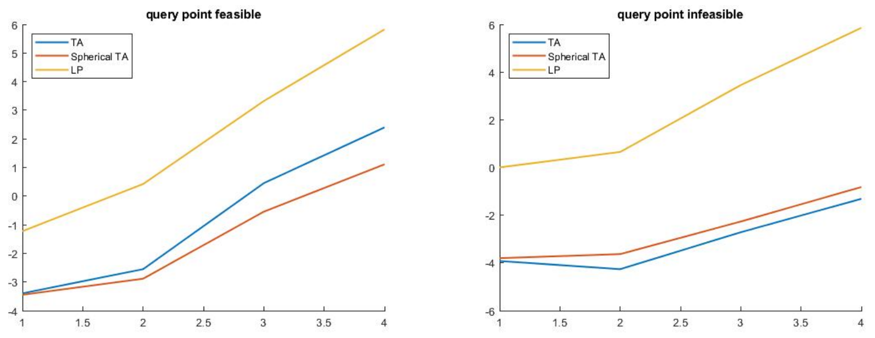

Figure 1: Running time of three algorithms on different size of problems.

References

[1] Pranjal Awasthi, Bahman Kalantari, and Yikai Zhang. Robust vertex enumeration for convex hulls in high dimensions. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics*, pages 1387–1396, 2018. Full version available at https://arxiv.org/abs/1802.01515.

[2] Vasek Chvátal. *Linear programming.* Macmillan, 1983.

[3] William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1, 1984.

[4] Bahman Kalantari. A characterization theorem and an algorithm for a convex hull problem. *Annals of Operations Research*, 226(1):301–349, 2015.

[5] Bahman Kalantari and Yikai Zhang. Spherical triangle algorithm: A fast oracle for convex hull membership queries. *arXiv preprint arXiv:1810.07346*, 2019.

[6] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. *Handbook of discrete and computational geometry.* Chapman and Hall/CRC, 2017.